Computer Architecture

Network Computing Lab.

Arm assembly Lab 4

Contents

- **메모리 test 및 메모리 내용내 데이터 패턴 검색 예제를 통해 메모리 접근 및 응용 assembly 프로그래밍을 배운다.**
  - 4-1 : 메모리 크기 검사
  - 4-2 : 메모리 내용내 데이터 패턴 검색
  - 4-3 : 메모리 내용내 Top 1 데이터 패턴 검색
  - HW8: P21, 29. 30에 나오는 실습A ~ C 수행하세요.결과를 캡쳐/설명 추가하여 보고서로 제출하세요. 실습C에서 작성한 소스코드는 file로 제출하세요.

Lab 4-1 : 메모리 크기 검사

- C 프로그램에서 테스트할 memory 시작주소(32bit), memory 끝 주소(32bit)를 r0, r1을 통해 어셈블리 코드로 전달한다.
- 메모리 테스트시 두 종류의 data pattern '01010101'와 '10101010'을 각각 사용한다.
- 위 pattern을 메모리 특정 번지에 쓰고 나서 다시 읽었을 때 같은 값이면 그 메모리 번지는 정상이고 만일 다르면 그 번지 메모리가 없거나 메모리 불량이다.
- 메모리 테스트는 byte 단위로 수행하며, 테스트 도중 에러가 발생하면 그 전 주소까지의 byte size를 r0를 통해 C로 반환한다.
- 이 과정을 거치면 현재 메모리 크기를 알 수 있다.
- 시스템 부팅과정에서 실제 메모리 크기를 확인할 때 사용되는 방법이다.

Lab 4-1 : 메모리 크기 검사

Debugging

```
 3 #define TEST_ADDR1 0xf6fff07c
26 void example1(void)
27 {
28   int result = 0;
29
30   result = memory_size_check(TEST_ADDR1, TEST_ADDR1 + (1 << 8));
31
32   printf("check array size: %d\n", result);
33 }
```

예제 1번 코드입니다.
예제마다 main 함수를 계속 수정할 필요 없이 example1() 함수에서 memory_size_check() 어셈블리 함수를 호출하고, main 함수에서 example1() 함수를 호출합니다.

```
 1 .text
 2 .global memory_size_check
 3 .type memory_size_check, STT_FUNC
 4
 5 memory_size_check:
 6   stmfd sp!, {r4-r6}
 7   mov r2, #0x55
 8   mov r3, #0xaa
 9
10   mov r4, r0
11   mov r5, #0
12
13 loop:
14   strb r2, [r4]
15   ldrb r6, [r4]
16   cmp r2, r6
17   bne exit
18   strb r3, [r4]
19   ldrb r6, [r4]
20   cmp r3, r6
21   bne exit
22   addeq r5, r5, #1
23   addeq r4, r4, #1
24   cmp r1, r4
25   bhi loop
26
27 exit:
28   mov r0, r5
29   ldmfd sp!, {r4-r6}
30   mov pc, lr
31
32 .end
```

Lab 4-1 : 메모리 크기 검사

- Debugging

```
 3 #define TEST_ADDR1 0xf6fff07c
26 void example1(void)
27 {
28   int result = 0;
29
30   result = memory_size_check(TEST_ADDR1, TEST_ADDR1 + (1 << 8));
31
32   printf("check array size: %d\n", result);
33 }
```

(base addr, end addr)

main 함수 example1() 함수를 호출하는데 첫 번째 인자는 확인할 메모리의 base 주소, 두 번째 인자는 확인할 메모리의 마지막 주소 입니다.
현재 입력된 값은 프로그램에 의존성이 없는 메모리 주소를 미리 찾아 넣어 놓았습니다. 만약 임의로 수정한다면 컴파일은 가능하지만 실행시 core dump가 발생할 수 있습니다.

Lab 4-1 : 메모리 크기 검사

```
1 .text
2 .global memory_size_check
3 .type memory_size_check, STT_FUNC
4
5 memory_size_check:
6   stmfd sp!, {r4-r6}
7   mov r2, #0x55
8   mov r3, #0xaa
9
10  mov r4, r0
11  mov r5, #0
12
13 loop:
14  strb r2, [r4]
15  ldrb r6, [r4]
16  cmp r2, r6
17  bne exit
18  strb r3, [r4]
19  ldrb r6, [r4]
20  cmp r3, r6
21  bne exit
22  add r5, r5, #1
23  add r4, r4, #1
24  cmp r1, r4
25  bhi loop
26
27 exit:
28  mov r0, r5
29  ldmfd sp!, {r4-r6}
30  mov pc, lr
31
32 .end
```

0101 0101 pattern 1
1010 1010 pattern 2

Pattern 1, 2를 각각 R2, R3에 대입합니다.

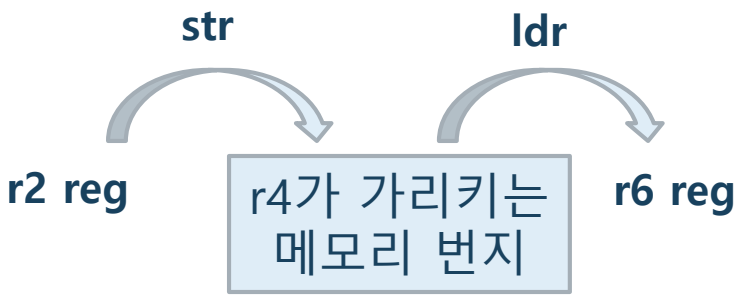
레지스터 사용
r0:메모리 시작주소
r1:메모리 끝주소
r2:test pattern 1
r3:test pattern 2
r4:현재 메모리주소
r5:counter (존재하는 메모리 byte수)
r6:데이터 읽어오는 임시레지스터

Lab 4-1 : 메모리 크기 검사

```
 1 .text
 2 .global memory_size_check
 3 .type memory_size_check, STT_FUNC
 4
 5 memory_size_check:
 6   stmfd sp!, {r4-r6}
 7   mov r2, #0x55
 8   mov r3, #0xaa
 9
10   mov r4, r0
11   mov r5, #0
12
13 loop:
14   strb r2, [r4]
15   ldrb r6, [r4]
16   cmp r2, r6
17   bne exit
18   strb r3, [r4]
19   ldrb r6, [r4]
20   cmp r3, r6
21   bne exit
22   add r5, r5, #1
23   add r4, r4, #1
24   cmp r1, r4
25   bhi loop
26
27 exit:
28   mov r0, r5
29   ldmfd sp!, {r4-r6}
30   mov pc, lr
31
32 .end
```

r0를 통해 받아온 메모리 base 주소 값이 r4에 있습니다. r4가 가리키는 주소에 str, ldr를 해서 값이 정상적으로 store/load 동작을 하는지 확인하는 부분입니다.
r2와 r6의 레지스터 값이 다르면 메모리가 불량하다는 것을 의미합니다.

Pattern 1 비교

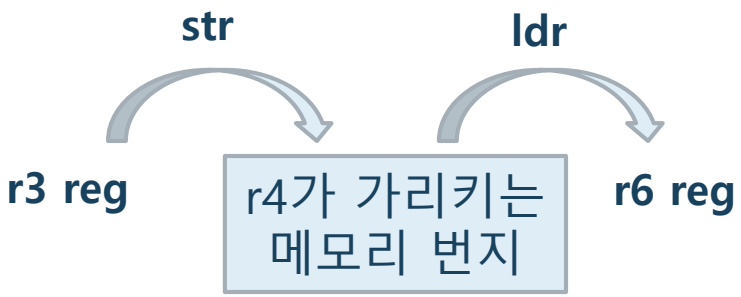


Lab 4-1 : 메모리 크기 검사

```
 1 .text
 2 .global memory_size_check
 3 .type memory_size_check, STT_FUNC
 4
 5 memory_size_check:
 6   stmfd sp!, {r4-r6}
 7   mov r2, #0x55
 8   mov r3, #0xaa
 9
10   mov r4, r0
11   mov r5, #0
12
13 loop:
14   strb r2, [r4]
15   ldrb r6, [r4]
16   cmp r2, r6
17   bne exit
18   strb r3, [r4]
19   ldrb r6, [r4]
20   cmp r3, r6
21   bne exit
22   add r5, r5, #1
23   add r4, r4, #1
24   cmp r1, r4
25   bhi loop
26
27 exit:
28   mov r0, r5
29   ldmfd sp!, {r4-r6}
30   mov pc, lr
31
32 .end
```

두 번째 패턴을 이용해 같은 메모리 번지를 테스트합니다. 이전과 동일한 방식입니다. str/ldr 방식으로 두 번 체크하는 거지요.

Pattern 2 비교



Lab 4-1 : 메모리 크기 검사

```
 1 .text
 2 .global memory_size_check
 3 .type memory_size_check, STT_FUNC
 4
 5 memory_size_check:
 6   stmfd sp!, {r4-r6}
 7   mov r2, #0x55
 8   mov r3, #0xaa
 9
10   mov r4, r0
11   mov r5, #0
12
13 loop:
14   strb r2, [r4]
15   ldrb r6, [r4]
16   cmp r2, r6
17   bne exit
18   strb r3, [r4]
19   ldrb r6, [r4]
20   cmp r3, r6
21   bne exit
22   add r5, r5, #1
23   add r4, r4, #1
24   cmp r1, r4
25   bhi loop
26
27 exit:
28   mov r0, r5
29   ldmfd sp!, {r4-r6}
30   mov pc, lr
31
32 .end
```

두 번의 검사를 통과했다면, 정상적인 메모리를 byte 단위로 count합니다. count를 위해 r5를 사용합니다.
검사를 완료했으니 이제 다음 주소를 검사해야 하겠죠? r4가 현재 base 주소입니다. r4 +1을 해서 검사할 주소를 1byte 이동합니다.

Lab 4-1 : 메모리 크기 검사

```
 1 .text
 2 .global memory_size_check
 3 .type memory_size_check, STT_FUNC
 4
 5 memory_size_check:
 6   stmfd sp!, {r4-r6}
 7   mov r2, #0x55
 8   mov r3, #0xaa
 9
10   mov r4, r0
11   mov r5, #0
12
13 loop:
14   strb r2, [r4]
15   ldrb r6, [r4]
16   cmp r2, r6
17   bne exit
18   strb r3, [r4]
19   ldrb r6, [r4]
20   cmp r3, r6
21   bne exit
22   add r5, r5, #1
23   add r4, r4, #1
24   cmp r1, r4
25   bhi loop
26
27 exit:
28   mov r0, r5
29   ldmfd sp!, {r4-r6}
30   mov pc, lr
31
32 .end
```

검사하는 도중 메모리 불량이 있다면
그 전 검사를 완료한 메모리까지 리턴하기 위해 exit로 분기합니다.

r1에는 검사할 메모리의 마지막 주소 값을 가지고 있습니다. 마지막에 도달했으면 loop로 분기하지 않고 호출했던 함수로 되돌아 갑니다.

Lab 4-1 : 메모리 크기 검사

- Debugging

메모리 검사 시작 주소를 0xf6fff07c로 주고,
256 (1 << 8) byte만큼 검사를 해보겠습니다.

```
 3 #define TEST_ADDR1 0xf6fff07c
26 void example1(void)
27 {
28   int result = 0;
29
30   result = memory_size_check(TEST_ADDR1, TEST_ADDR1 + (1 << 8));
31
32   printf("check array size: %d\n", result);
33 }
```

Lab 4-1 : 메모리 크기 검사

Load Binary /path/to/target/executable -and -flags

show filesystem | fetch disassembly | reload file | jump to line | /home/youngjoon/CA_lab/lab4/memory_size_check.s:10 (33 lines total)

```
1   .text
2   .global memory_size_check
3   .type memory_size_check, STT_FUNC
4
5   memory_size_check:
6           stmfd sp!, {r4-r6}
7           mov r2, #0x55
8           mov r3, #0xaa
9
10          mov r4, r0
11          mov r5, #0
12
13  loop:
14          strb r2, [r4]
15          ldrb r6, [r4]
16          cmp r2, r6
17          bne exit
18          strb r3, [r4]
19          ldrb r6, [r4]
20          cmp r3, r6
21          bne exit
22          add r5, r5, #1
23          add r4, r4, #1
```

디버거를 실행시키고 memory_size_check 함수에 breakpoint를 설정하고 진입합니다.
그러면 r0에 base 주소가 들어가게 됩니다. 레지스터 섹션에서 r0값을 클릭해 메모리가 변하는 것을 확인해봅시다.

Lab 4-1 : 메모리 크기 검사

```
5   memory_size_check:
6           stmfd sp!, {r4-r6}
7           mov r2, #0x55
8           mov r3, #0xaa
9
10          mov r4, r0
11          mov r5, #0
12
13  loop:
14          strb r2, [r4]
15          ldrb r6, [r4]
16          cmp r2, r6
17          bne exit
18          strb r3, [r4]
19          ldrb r6, [r4]
20          cmp r3, r6
21          bne exit
22          add r5, r5, #1
23          add r4, r4, #1
24          cmp r1, r4
25          bhi loop
26
27  exit:
28          mov r0, r5
29          ldmfd sp!, {r4-r6}
30          mov pc, lr
31
```

왼쪽과 같이 breakpoint를 설정하고 c를 눌러보면 1byte 단위로 메모리를 체크하는 것을 오른쪽과 같이 메모리 섹션에서 확인 가능할 겁니다.

memory

0xf6fff07c | 0xf6fff14f | 16

address	hex
more	
0xf6fff07c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff08c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff09c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff0ac	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff0bc	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff0cc	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff0dc	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff0ec	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff0fc	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff10c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff11c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff12c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff13c	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
0xf6fff14c	aa aa aa aa
more	

Lab 4-2 : 메모리 내용내 데이터 패턴 검색

- 특정 메모리 영역내에서 검색 데이터 패턴(2 byte 크기)이 몇 개 존재하는지 그 개수를 구하는 어셈블리 프로그램이다.
- C 프로그램으로부터, 검색할 메모리 시작 주소는 r0, 끝 주소는 r1, 검색 패턴은 r2 통해 전달받고 특정 메모리 영역내 검색 패턴의 개수 값(counter)은 r0 통해 리턴된다.
- 시작 주소부터 끝 주소까지 2 byte를 단위로 검사하는 것을 반복한다.
  - 검색 성공시 현재 주소를 2 byte씩 증가시킨다. 검색 실패시는 1 byte만 증가시킨다.

Lab 4-2 : 메모리 내용내 데이터 패턴 검색

- Debugging

```
 5 #define TEST_ADDR2 0xf6fff180
 6 #define TEST_PATTERN 0x0000f6ff
35 void example2(void)
36 {
37   int result = 0;
38
39   result = pattern_search(TEST_ADDR2, TEST_ADDR2 + (1 << 8), TEST_PATTERN);
40
41   printf("pattern count: %d\n", result);
42 }
```

함수 인자 정보는 다음과 같습니다.
r0 : TEST_ADDR2 -> base 주소
r1 : TEST_ADDR2 + 256(1 << 8) -> end 주소
r2 : TEST_PATTERN -> 패턴

Lab 4-2 : 메모리 내용내 데이터 패턴 검색

Network Computing Lab.

- Debugging

```
1   .text
2   .global pattern_search
3   .type pattern_search, STT_FUNC
4
5   pattern_search:
6           push {r4, r5, lr}
7           mov r3, #0
8           ldr r5, =0x0000ffff
9
10  loop:
11          ldr r4, [r0], #2
12          and r4, r4, r5
13          cmp r4, r2
14          addeq r3, r3, #1
15          subne r0, r0, #1
16          cmp r0, r1
17          bls loop
18          mov r0, r3
19
20          pop {r4, r5, pc}
21
22  .end
```

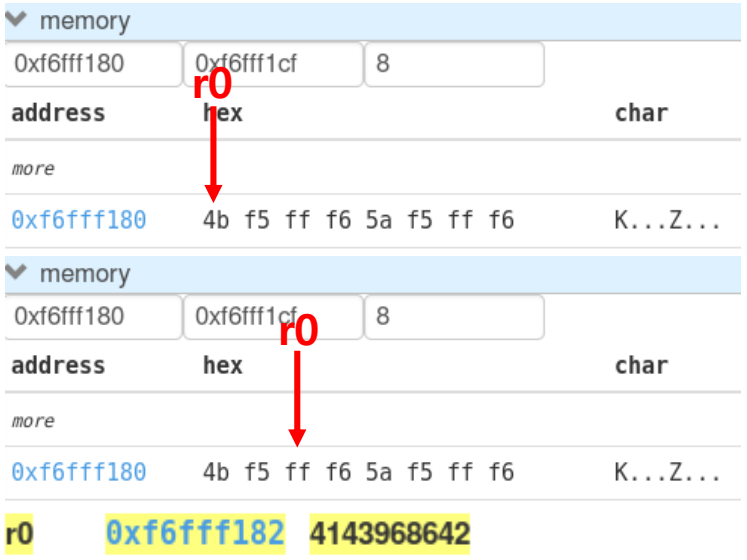
레지스터 사용
r0:메모리 시작주소
r1:메모리 끝주소
r2:test pattern
r3:counter
r4:데이터 읽어오는 임시레지스터
r5:mask pattern (0x0000ffff)

Lab 4-2 : 메모리 내용내 데이터 패턴 검색

Debugging

```
1   .text
2   .global pattern_search
3   .type pattern_search, STT_FUNC
4
5   pattern_search:
6           push {r4, r5, lr}
7           mov r3, #0
8           ldr r5, =0x0000ffff
9
10  loop:
11          ldr r4, [r0], #2
12          and r4, r4, r5
13          cmp r4, r2
14          addeq r3, r3, #1
15          subne r0, r0, #1
16          cmp r0, r1
17          bls loop
18          mov r0, r3
19
20          pop {r4, r5, pc}
21
22  .end
```

먼저 pattern_search 함수에 breakpoint를 설정하고 내부로 진입합니다. 그리고 r0에 있는 base 주소를 메모리 섹션에 찍어봅시다. 메모리 섹션이 준비되었으면 12번 line에 breakpoint를 설정하고 r4에 값이 load된 후에 r0가 어떻게 증가되는지 확인해봅시다. 아래는 r0 변화 전 후의 비교입니다. Post-indexing 방식으로 값을 load하고 메모리 번지를 +2 했습니다. 읽어간 메모리는 빨간색으로 표시되었습니다.



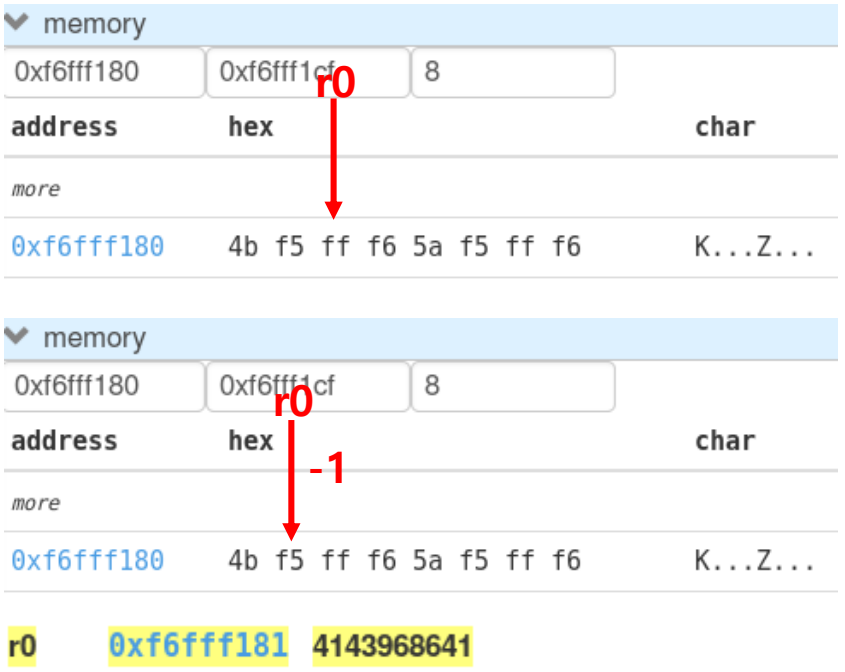
Lab 4-2 : 메모리 내용내 데이터 패턴 검색

Debugging

```
1   .text
2   .global pattern_search
3   .type pattern_search, STT_FUNC
4
5   pattern_search:
6           push {r4, r5, lr}
7           mov r3, #0
8           ldr r5, =0x0000ffff
9
10  loop:
11          ldr r4, [r0], #2
12          and r4, r4, r5
13          cmp r4, r2
14          addeq r3, r3, #1
15          subne r0, r0, #1
16          cmp r0, r1
17          bls loop
18          mov r0, r3
19
20          pop {r4, r5, pc}
21
22  .end
```

메모리의 값과 우리가 찾고자 하는 패턴과 비교한 후에 같으면, count 값(r3)을 +1 합니다.

다르다면, 현재 메모리 번지 값을 가지고 있는 r0를 -1 합니다.



Lab 4-2 : 메모리 내용내 데이터 패턴 검색

- Debugging

```
1   .text
2   .global pattern_search
3   .type pattern_search, STT_FUNC
4
5   pattern_search:
6           push {r4, r5, lr}
7           mov r3, #0
8           ldr r5, =0x0000ffff
9
10  loop:
11          ldr r4, [r0], #2
12          and r4, r4, r5
13          cmp r4, r2
14          addeq r3, r3, #1
15          subne r0, r0, #1
16          cmp r0, r1
17          bls loop
18          mov r0, r3
19
20          pop {r4, r5, pc}
21
22  .end
```

r1에 메모리 end 주소가 있습니다.
메모리 끝까지 패턴 검색을 진행했다면 loop 로 분기하지 않고 18번 라인으로 넘어갑니다

Lab 4-2 : 메모리 내용내 데이터 패턴 검색

Debugging

```
youngjoon@ubuntu:~/CA_lab/lab4$ qemu-arm
check array size: 256
pattern count: 41
```

결과 창을 확인해보면 매칭되는 패턴을 41개 찾았다고 알려줍니다. 메모리 창에서 확인해보면 41개가 존재하는 것을 확인할 수 있습니다.

memory

0xf6fff180 | 0xf6fff25f | 16

address	hex
more	
0xf6fff180	4b f5 ff f6 5a f5 ff f6 6f f5 ff f6 8e f5 ff f6
0xf6fff190	a8 f5 ff f6 ba f5 ff f6 d7 f5 ff f6 e9 f5 ff f6
0xf6fff1a0	1f f6 ff f6 2e f6 ff f6 3f f6 ff f6 52 f6 ff f6
0xf6fff1b0	66 f6 ff f6 78 f6 ff f6 98 f6 ff f6 ad f6 ff f6
0xf6fff1c0	ce f6 ff f6 e0 f6 ff f6 f7 f6 ff f6 a9 f7 ff f6
0xf6fff1d0	ed f7 ff f6 20 f8 ff f6 49 f8 ff f6 7d f8 ff f6
0xf6fff1e0	b7 f8 ff f6 ca f8 ff f6 52 fe ff f6 61 fe ff f6
0xf6fff1f0	8e fe ff f6 a5 fe ff f6 e9 fe ff f6 fb fe ff f6
0xf6fff200	1e ff ff f6 2f ff ff f6 43 ff ff f6 53 ff ff f6
0xf6fff210	89 ff ff f6 be ff ff f6 d4 ff ff f6 e6 ff ff f6
0xf6fff220	00 00 00 00 1a 00 00 00 1f 00 00 00 19 00 00 00
0xf6fff230	bd f2 ff f6 11 00 00 00 64 00 00 00 10 00 00 00
0xf6fff240	d7 b8 1f 00 0e 00 00 00 e8 03 00 00 0d 00 00 00
0xf6fff250	e8 03 00 00 0c 00 00 00 e8 03 00 00 0b 00 00 00

Lab 4-2 : 메모리 내용내 데이터 패턴 검색

Network Computing Lab.

- **실습 A**
  - P19의 코드에서, 14번 라인 addeq 명령어에서 조건이 만족되면 counter(r3)는 1만큼 증가하고 메모리 주소(r4)는 2만큼 증가합니다.
  - 14번 라인에 breakpoint를 걸어 addeq에서 조건이 참일 경우가 처음 나타날 때의 r3, r4값, r0값 - 2가 가리키는 메모리 값을 화면 캡처하여 제출하세요.

Lab 4-3 : Top1 패턴 검색

- 이번엔 특정 메모리 영역에서 가장 많이 나타나는 1 byte 짜리 데이터 패턴을 검색합니다.
- 메모리 구간에서 0x00 ~ 0xff까지의 패턴을 모두 검사합니다.
- 예를 들어, 패턴이 0x00 일 경우 그 메모리 구간에 들어 있는 총 횟수를 계산하여 r4에는 패턴의 개수, r5에는 패턴을 저장합니다.(현재로는 이 패턴이 Top1) 패턴 값을 0x01로 증가하여 같은 과정을 반복합니다. 만일 0x01의 총 개수가 저장된 Top1의 개수보다 크면 Top1 패턴, Top1의 개수를 update합니다. 이 과정을 패턴이 0xff 될 때까지 반복한 뒤 저장된 Top1, Top1의 개수가 그 메모리 구간에서 가장 많이 나타나는 패턴 및 패턴의 개수 입니다.

Lab 4-3 : Top1 패턴 검색

Network Computing Lab.

- Debugging

```
 5 #define TEST_ADDR2 0xf6fff180
 6 #define TEST_PATTERN 0x0000f6ff
44 void example3(void)
45 {
46   int result[2];
47
48   top1_pattern_search(TEST_ADDR2, TEST_ADDR2 + (1 << 8), 0, result);
49
50   printf("Top1 pattern: %x\n", result[0]);
51   printf("Top1 pattern count: %d\n", result[1]);
52 }
53
```

top_pattern_search 함수를 호출 할 때 인자 4개가 r0, r1, r2, r3를 통해 전달됩니다. result 배열이 저장된 주소가 r3에 들어 있습니다. top_pattern_search 함수를 호출한 뒤 result[0]에는 Top1 패턴이 저장되어 돌아오고 result[1]에는 Top1 패턴의 총 횟수가 저장되어 돌아옵니다.

31번 라인에 보면 Top1 패턴이 저장된 r5를 r3가 가리키는 메모리(result[0])에 저장합니다. 32번 라인에 보면 Top1 패턴 횟수가 저장된 r4를 r3+4 가 가리키는 메모리(result[1])에 저장합니다.

```
 1 .text
 2 .global top1_pattern_search
 3 .type top1_pattern_search, STT_FUNC
 4
 5 top1_pattern_search:
 6   stmfd sp!, {r4-r9, lr}
 7   mov r4, #0
 8   mov r5, #0
 9
10 loop1:
11   mov r6, #0
12   mov r7, r0
13
14 loop2:
15   ldrb r8, [r7], #1
16
17   cmp r2, r8
18   addeq r6, r6, #1
19
20   cmp r7, r1
21   bls loop2
22
23   cmp r6, r4
24   movgt r4, r6
25   movgt r5, r2
26
27   cmp r2, #0xff
28   addlt r2, r2, #1
29   blt loop1
30
31   str r5, [r3]
32   str r4, [r3, #4]
33   ldmfd sp!, {r4-r9, lr}
34
35   mov pc, lr
36
37 .end
```

Lab 4-3 : Top1 패턴 검색

- Debugging

레지스터 사용
r0:메모리 시작주소
r1:메모리 끝주소
r2:test pattern 1
r3:결과저장용 배열주소. result[0] pattern 저장, result[1] counter 값
r4:Top1 counter
r5:Top1 pattern
r6:counter (pattern 수)
r7:현재 메모리주소
r8:데이터 읽어오는 임시레지스터

```
1 .text
2 .global top1_pattern_search
3 .type top1_pattern_search, STT_FUNC
4
5 top1_pattern_search:
6   stmfd sp!, {r4-r9, lr}
7   mov r4, #0
8   mov r5, #0
9
10 loop1:
11   mov r6, #0
12   mov r7, r0
13
14 loop2:
15   ldrb r8, [r7], #1
16
17   cmp r2, r8
18   addeq r6, r6, #1
19
20   cmp r7, r1
21   bls loop2
22
23   cmp r6, r4
24   movgt r4, r6
25   movgt r5, r2
26
27   cmp r2, #0xff
28   addlt r2, r2, #1
29   blt loop1
30
31   str r5, [r3]
32   str r4, [r3, #4]
33   ldmfd sp!, {r4-r9, lr}
34
35   mov pc, lr
36
37 .end
```

byte단위로 메모리 값을 load하고 패턴과 비교합니다.

Lab 4-3 : Top1 패턴 검색

- Debugging

패턴이 매칭되었으면 count를 +1 합니다.
r6이 count 값이 되겠군요.

```
 1 .text
 2 .global top1_pattern_search
 3 .type top1_pattern_search, STT_FUNC
 4
 5 top1_pattern_search:
 6   stmfd sp!, {r4-r9, lr}
 7   mov r4, #0
 8   mov r5, #0
 9
10 loop1:
11   mov r6, #0
12   mov r7, r0
13
14 loop2:
15   ldrb r8, [r7], #1
16
17   cmp r2, r8
18   addeq r6, r6, #1
19
20   cmp r7, r1
21   bls loop2
22
23   cmp r6, r4
24   movgt r4, r6
25   movgt r5, r2
26
27   cmp r2, #0xff
28   addlt r2, r2, #1
29   blt loop1
30
31   str r5, [r3]
32   str r4, [r3, #4]
33   ldmfd sp!, {r4-r9, lr}
34
35   mov pc, lr
36
37 .end
```

Lab 4-3 : Top1 패턴 검색

- Debugging

현재 패턴에 대한 count 값이 r3입니다.
현재까지 Top1 패턴에 대한 count 값은 r4입니다.
둘을 비교해서 Top1 패턴을 갱신합니다.
r4은 Top1 패턴의 count 이고
r5은 Top1 패턴 입니다.

```
 1 .text
 2 .global top1_pattern_search
 3 .type top1_pattern_search, STT_FUNC
 4
 5 top1_pattern_search:
 6   stmfd sp!, {r4-r9, lr}
 7   mov r4, #0
 8   mov r5, #0
 9
10 loop1:
11   mov r6, #0
12   mov r7, r0
13
14 loop2:
15   ldrb r8, [r7], #1
16
17   cmp r2, r8
18   addeq r6, r6, #1
19
20   cmp r7, r1
21   bls loop2
22
23   cmp r6, r4
24   movgt r4, r6
25   movgt r5, r2
26
27   cmp r2, #0xff
28   addlt r2, r2, #1
29   blt loop1
30
31   str r5, [r3]
32   str r4, [r3, #4]
33   ldmfd sp!, {r4-r9, lr}
34
35   mov pc, lr
36
37 .end
```

Lab 4-3 : Top1 패턴 검색

- Debugging

패턴을 0x00 ~ 0xFF까지 검색한다고 했죠? 0xFF까지 진행하지 않았다면 loop1으로 분기하여 다시 검색할 준비를 합니다.

패턴을 바꾸고 주소를 다시 초기화하고 count 값도 초기화 합니다.

```
1 .text
2 .global top1_pattern_search
3 .type top1_pattern_search, STT_FUNC
4
5 top1_pattern_search:
6   stmfd sp!, {r4-r9, lr}
7   mov r4, #0
8   mov r5, #0
9
10 loop1:
11   mov r6, #0
12   mov r7, r0
13
14 loop2:
15   ldrb r8, [r7], #1
16
17   cmp r2, r8
18   addeq r6, r6, #1
19
20   cmp r7, r1
21   bls loop2
22
23   cmp r6, r4
24   movgt r4, r6
25   movgt r5, r2
26
27   cmp r2, #0xff
28   addlt r2, r2, #1
29   blt loop1
30
31   str r5, [r3]
32   str r4, [r3, #4]
33   ldmfd sp!, {r4-r9, lr}
34
35   mov pc, lr
36
37 .end
```

Lab 4-3 : Top1 패턴 검색

- Debugging

```
^Cyoungjoon@ubuntu:~/CA_lab/lab4$ qemu-arm -
check array size: 256
pattern count: 41
Top1 pattern: 0
Top1 pattern count: 65
```

결과창은 이런 식으로 오른쪽 메모리 기준으로 했을 때 Top1의 패턴을 출력하고 이것의 count 값도 출력합니다.

패턴 0x00이 총 65개로 가장 많은 것으로 확인되네요.

memory

0xf6fff180 | 0xf6fff2a7 | 16

address	hex
more	
0xf6fff180	4b f5 ff f6 5a f5 ff f6 6f f5 ff f6 8e f5 ff f6
0xf6fff190	a8 f5 ff f6 ba f5 ff f6 d7 f5 ff f6 e9 f5 ff f6
0xf6fff1a0	1f f6 ff f6 2e f6 ff f6 3f f6 ff f6 52 f6 ff f6
0xf6fff1b0	66 f6 ff f6 78 f6 ff f6 98 f6 ff f6 ad f6 ff f6
0xf6fff1c0	ce f6 ff f6 e0 f6 ff f6 f7 f6 ff f6 a9 f7 ff f6
0xf6fff1d0	ed f7 ff f6 20 f8 ff f6 49 f8 ff f6 7d f8 ff f6
0xf6fff1e0	b7 f8 ff f6 ca f8 ff f6 52 fe ff f6 61 fe ff f6
0xf6fff1f0	8e fe ff f6 a5 fe ff f6 e9 fe ff f6 fb fe ff f6
0xf6fff200	1e ff ff f6 2f ff ff f6 43 ff ff f6 53 ff ff f6
0xf6fff210	89 ff ff f6 be ff ff f6 d4 ff ff f6 e6 ff ff f6
0xf6fff220	00 00 00 00 1a 00 00 00 1f 00 00 00 19 00 00 00
0xf6fff230	bd f2 ff f6 11 00 00 00 64 00 00 00 10 00 00 00
0xf6fff240	d7 b8 1f 00 0e 00 00 00 e8 03 00 00 0d 00 00 00
0xf6fff250	e8 03 00 00 0c 00 00 00 e8 03 00 00 0b 00 00 00
0xf6fff260	e8 03 00 00 09 00 00 00 4c 04 01 00 08 00 00 00
0xf6fff270	00 00 00 00 07 00 00 00 00 00 00 00 06 00 00 00

Lab 4-3 : Top1 패턴 검색

Network Computing Lab.

- **실습 B**
  - P23의 코드에서, 패턴=0x00일 때 검색이 완료되면 그 패턴과 패턴의 총 횟수가 top1 및 top1 횟수에 저장됩니다. 27번 라인에 breakpoint를 걸고 패턴=0x00일 때 검색이 완료되어 패턴과 패턴의 총 횟수가 r4, r5에 저장되는 될 때 그 값을 캡처하세요.
  - P23의 코드에서, 패턴=0xff일 때 까지 검색이 완료되면 top1 및 top1 횟수가 각각 r4, r5에 저장되어 있습니다. main으로 return 하기 바로 전에 r4, r5를 result[0], result[1]에 저장합니다. 31번 라인에 breakpoint 를 걸어 31번 라인 수행 전 r4, r5 값을 화면 캡처, result[0], result[1]값 (메모리 주소 및 메모리 값) 캡처, 32번 라인 수행 후 result[0], result[1]값 (메모리 주소 및 메모리 값) 캡처하세요.

Lab 4-3 : Top1 패턴 검색

Network Computing Lab.

실습 C

- Lab 4-3은 Top1 패턴 검색결과를 출력합니다. 여기서는 Top 3 패턴 검색결과를 출력하려고 합니다. (Top1, Top2, Top3에 대한 패턴 및 패턴 총 횟수를 각각 출력)
- main에서 result[2]를 result[6]으로 수정한다. result[2], result[3]에는 Top2에 대한 패턴 및 패턴 총 횟수, result[4], result[5]에는 Top3에 대한 패턴 및 패턴 총 횟수를 받아봅시다.
- 어셈블리코드에서는 현재는 Top1 에 대한 패턴 및 패턴 총 횟수만을 저장하는 데 이를 , Top1, Top2, Top3에 대한 패턴 및 패턴 총 횟수 를 각각 저장하고 아래와 같이 update할 수 있도록 확장합니다.
  - 예를 들어 현재 패턴=0x10을 갖고 검색을 완료했다고 가정합시다.
  - 그 패턴의 총 횟수를 Top1의 총 회수와 비교하여 더 크면 그 패턴 및 패턴 총 회수가 Top1이 됩니다. 그 전의 Top1이 Top2가 되고, 그전의 Top2가 Top3가 됩니다.
  - 그 패턴의 총 횟수를 Top1의 총 회수와 비교하여 작으면 Top1은 그대로 두고 그 패턴의 총 회수를 Top2의 총 회수와 비교하여 더 크면 그 패턴 및 패턴 총 횟수가 Top2가 됩니다. 그 전의 Top2가 Top3가 됩니다.
  - 그 패턴의 총 회수를 Top1 및 Top2의 총 회수와 비교하면 작을 때는 Top3와 비교합니다. 만일 Top3의 총 회수보다 크면 그 패턴 및 패턴 총 회수가 Top3가 됩니다.
- main으로 return하기 바로 전에 Top1, Top2, Top3 값들을 result[0] ~ result[5]에 저장합니다. main에서 최종 Top1, Top2, Top3의 패턴 및 패턴 회수를 출력합니다.
- 수행 결과에 대한 화면을 캡쳐하세요. 수정된 소스코드는 file로 제출하세요.