2021 Fall System Programming

Assembly & BombLab

2021. 10. 18

유주원
clearlyhunch@gmail.com

Embedded System Lab.
Computer Engineering Dept.
Chungnam National University

실습 소개

- 과목 홈페이지
  - 충남대학교 사이버캠퍼스 (http://e-learn.cnu.ac.kr/)
- 연락처
  - 유주원
  - 공대 5호관 533호 임베디드 시스템 연구실
  - clearlyhunch@gmail.com
    - Email 제목은 **'[시스템프로그래밍][01][학번_이름]용건'** 으로 시작하도록 작성

Assembly

CNU Embedded System Laboratory

어셈블리어?

- 어셈블리어(Assembly Language)는 0과 1의 이진수 프로그래밍을 좀 더 편하게 하기 위해 비트 패턴을 명령어로 만든 언어
  - 하드웨어 디바이스 드라이버, 일반 프로그램의 특정 기능 최적화 등에 사용

- 어셈블리어는 시스템의 구조에 따라 문법과 명령어 셋(set) 등이 다르다.
  - masm(MS), TASM(Borland), NASM(open source), GAS(GNU)

어셈블리 명령어 구조

[Label]:
    [operator] [operand1], [operand2] // [comment]

- Label
  - 해당 명령어의 주소를 나타내는 라벨
- Operator
  - 연산자 코드
  - eg. MOV, ..
- Operand
  - 피연산자
  - 주로 레지스터가 들어감.
  - eg. $0, %rax, %rdi, …
- Comment
  - 주석
  - !, #, // 모두 가능

CNU Embedded System Laboratory

어셈블리 프로그램 구조

- .section
  - 메모리 영역을 지정
  - .data
    - 프로그램에서 사용하는 변수를 저장하는 영역
    - 전역 변수, 정적 변수, 배열, 구조체 등
  - .text
    - 실제로 실행되는 코드를 저장하는 영역
- .global main
  - 프로그램의 시작점

```
.section .data
label:

.section .text

.global main
main:

ret
```

레지스터

- 프로세서 내에서 데이터를 저장하는 공간
- 일반적으로 현재 계산중인 값을 저장하는데 쓰임.
- 64bit 길이 (64bit 머신)

- rsp
  - Stack Pointer: stack의 상위 주소를 가리키는 레지스터
- rbp
  - Base Pointer: stack의 base 주소를 가리키는 레지스터
- rip
  - Instruction Pointer(Program Counter): 실행 할 명령의 주소를 가리키는 레지스터.
  - 각각의 명령이 실행 될 때, rip에 CPU가 현재 실행하고 있는 주소가 저장됨

CNU Embedded System Laboratory

참고 - 어셈블리어

- 주의사항
  1) 우리는 AT&T 의 문법 규칙을 따른다.
  2) GAS 어셈블러를 사용하기때문에 인텔 어셈블리어의 문법 규칙이 아니라 AT&T와 인텔의 구문은 source와 destination을 반대로 사용한다.예를 들면 다음과 같다.
     - 인텔: mov rax, 4
     - AT&T: mov $4, %rax
  3) AT&T 구문에서는 피연산자 중 값은 $로 시작한다. 인텔 구문에서는 그렇지 않다. 예를 들면 다음과 같다.
     - 인텔: push 4
     - AT&T: pushl $4
  4) AT&T 구문에서 메모리 피연산자의 크기는 opcode 이름의 마지막 글자로 결정된다. opcode는 b(8bits), w(16bits), l(32bits)로 각각 정해진 메모리 참조를 나타낸다. 인텔 구문은 메모리 피연산자 접두어(byte ptr, word ptr, dowrd ptr)로 결정된다.
     - 인텔: mov al, byte ptr foo
     - AT&T: movb foo, %al

CNU Embedded System Laboratory

출력 함수의 사용법

- 어셈블리어로 아래의 C언어 스타일과 같은 형식의 printf 결과를 보이는 프로그램
- printf("val1 = %d val2 = %d val3 = %d\n", val1, val2, val3);
  - 매개변수 레지스터 : rdi, rsi, rdx, rcx 등등
  - 리턴 값 : rax

소스코드

```
.section .data
msg:
    .string "val1 = %d val2 = %d val3 = %d \n"
val1:
    .int 100
val2:
    .int 200
val3:
    .int 300

.section .text
.global main

main:
    movq    $msg, %rdi
    movq    val1, %rsi
    movq    val2, %rdx
    movq    val3, %rcx

    movq    $0, %rax
    call    printf
    movq    $0, %rax
    ret
```

tab 키 사용

컴파일 및 실행

```
sys01@2020sp:~/week05$ vi ex02.s
sys01@2020sp:~/week05$ gcc -no-pie -o ex02.out ex02.s
sys01@2020sp:~/week05$ ./ex02.out
val1 = 100 val2 = 200 val3 = 300
```

CNU Embedded System Laboratory

scanf와 printf

```
1  .section .data
2  scanf_format:
3      .string "%d"
4  printf_format:
5      .string "your : %d \n"
6  input :
7      .int 0
8
9
0  .section .text
1  .global main
2
3  main:
4
5      #call function scanf
6      #scanf("%d", &input);
7      movq $scanf_format, %rdi
8      movq $input, %rsi
9      movq $0, %rax
0      call scanf
1
2      #call function printf
3      #printf("your : %d \n", input);
4      movq $printf_format, %rdi
5      movq input, %rsi
6      movq $0, %rax
7      call printf
8
9      ret
```

- scanf_format 은 scanf로 입력을 받을 형식을 저장
- prinft_format 은 printf 로 출력할 형식을 저장
- input은 입력 값을 저장한 변수
- Scanf(%d", &input);
- 변수 앞의 $는 &개념.
- scanf 로 입력을 받는다
- 이때 입력 받은 값은 변수 input에 저장되기 때문에 레지스터 rsi 로 옮겨 줘야한다.
- 입력 받은 값을 printf 로 출력!

CNU Embedded System Laboratory

scanf와 printf

- 앞의 코드를 컴파일 .
- 실행 후, scanf와 printf 사용법을 숙지.

```
sys00@localhost:~/lab05/TA$ vi ex03.s
sys00@localhost:~/lab05/TA$ gcc -o ex03 ex03.s
sys00@localhost:~/lab05/TA$ ./ex03
1234
your input number : 1234
```

  - 컴파일 및 실행 결과

CNU Embedded System Laboratory

연산 명령어

❖ 연산 명령 addq를 이용하여 작성한 프로그램과 출력.

실행결과

```
sys00@localhost:~/lab05/student$ gcc -o ex01 ex01.s
sys00@localhost:~/lab05/student$ ./ex01
val1= 100 val2 =  200 result = 300
```

```
.section .data
message :
    .string "val1= %d val2 =  %d result = %d #n"
val1:
    .int 100
val2:
    .int 200


.section .text
.globl main

main:
    # move a message to a register(=reg) rdi
    movq    $message, %rdi

    # move variables val1 & val2 to reg rsi & rdx
    movq    val1, %rsi
    movq    val2, %rdx

    # add  rsi(val1) & rdx(val2), rdx = rdx + rsi
    addq    %rsi, %rdx

    # move the result to reg rcx
    # since val2 was overlapped by the result, move val2 to rdx
    movq    %rdx, %rcx
    movq    val2, %rdx

    # call function printf
    movq    $0, %rax
    call    printf

    #return
    ret
```

CNU Embedded System Laboratory

Setting Condition

- 비교 연산자를 통한 조건(상태) 플래그 설정
  - 오버 플로우, 제로, 부호, 캐리 플래그 등이 있다.

- cmpq Dest, Src
  - cmpq Dest, Src는 Src – Dest 연산을 통해 값을 비교함

- 조건 플래그 (Condition Flag)
  - CF(Carry Flag) : MSB(Most Significant Bit)로 부터의 자리 올림(carry) 혹은 빌림(borrow)이 발생할 경우에 1로 설정
    - 부호 없는 산술 연산에서 오버 플로우를 검출할 때 사용
  - ZF(Zero Flag) : dest 와 src 의 값이 같은 경우(Src- Dest == 0) 1로 설정
  - SF(Sign Flag) : Src – Dest 의 부호를 나타낸다. 0은 양수, 1은 음수.
  - OF(Overflow Flag) : 부호 있는 연산 결과가 오버 플로우가 발생된 경우. 오버 플로우가 발생한 경우 1로 설정.
    - 양수/음수의 2의 보수 오버 플로우를 발생시킨 것을 표시

CNU Embedded System Laboratory

실습5. Jump 명령의 이용 - 분기

- jx Label - if-else, while등 조건 문으로 사용 가능
  - 컨디션 코드들에 따라서 조건 분기 및 무조건 분기

jx	Condition	Description
jmp (jump)	1	무조건 분기
je (equal)	ZF (ZF = 1)	ZF가 1인 경우(Dest == Src)
jne (not e)	~ZF	ZF가 0인 경우(Dest != Src)
js (sign)	SF	SF가 1인 경우(음수)
jns (not s)	~SF	SF가 0인 경우(양수)
jg (greater)	~(SF^OF)&~ZF (ZF = 0 and SF == OF)	큰 경우 > (Signed)
jge (greater e)	~(SF^OF)	크거나 같은 경우 ≥ (Signed)
jl (less)	(SF^OF)	작은 경우 < Signed)
jle (less e)	(SF^OF)|ZF	작거나 같은 경우 ≤ (Signed)
ja (above)	~CF&~ZF (CF = 0 and ZF = 0)	앞의 숫자가 큰 경우 > (Unsigned)
jb (below)	CF	뒤의 숫자가 큰 경우 < (Unsigned)

CNU Embedded System Laboratory

실습5. Jump 명령의 이용 - 분기

```
.section .data
scanf_str:
    .string "%d %d"

printf_str:
    .string "%d is greater #n"

val1:
    .int 0
val2:
    .int 0

.section .text
.globl  main
main:
    movl    $val1, %esi
    movl    $val2, %edx
    movq    $scanf_str, %rdi

    movq    $0, %rax
    call    scanf

    # move results to register to compare
    movl    val1, %esi
    movl    val2, %edx

    #compare
    cmpl    %edx, %esi

    #choose the greater one to print
    jg  greater
    movl    %edx, %esi

greater:
    movq    $printf_str, %rdi
    movq    $0, %rax
    call    printf

    ret
```

- 다음은 두 수를 입력 받아 둘 중 더 큰 수를 출력하는 프로그램이다.

실행 결과

```
sys00@localhost:~/lab05/TA$ ./ex04
10 15
15 is greater
```

CNU Embedded System Laboratory

실습6. Loop의 이용

```
.section .data
scanf_str:
    .string "%d"
printf_str:
    .string "result : %d \n"
i:
    .int 0
sum:
    .int 0
n:
    .int 0

.section .text
.globl  main

main:

    movl    $n, %esi
    movq    $scanf_str, %rdi

    movq    $0, %rax
    call    scanf

    movl    sum, %edx
    movl    i, %ecx

loop:
    addl    %ecx, %edx      #sum += i
    incl    %ecx            #i++
    cmpl    n,  %ecx        #
    jle     loop            #if ecx(i) <= n , jump


    movl    %edx, %esi      #print sum

    movq    $printf_str, %rdi
    movq    $0, %rax
    call    printf
    ret
```

- 사용자로부터 n을 입력 받아 0에서부터 n까지 합을 출력하는 프로그램이다.
  - 다음을 입력하여 아래와 같이 결과를 출력.

실행 결과

```
sys00@localhost:~/lab05/TA$ ./ex05
10
result : 55
```

CNU Embedded System Laboratory

실습 7. Switch 문의 구현

- Switch 문은 if … else 형태로 구현이 가능하다.

```c
#include <stdio.h>

int main(){

    int x = 1;
    int ch = 0;

    switch(x){
        case 0:
            ch = 65;
            break;
        case 1:
            ch = 66;
            break;
        default:
            ch = 0;
    }

    printf("result: %d \n",  ch);
}
```

C 언어의 switch 문

```
.section .data
printf_str:
    .string "result : %d \n"
x:
    .int 1

.section .text
.globl  main

main:

    movl    x, %ebx

    cmpl    $0, %ebx
    movl    $65,%esi
    je      END

    cmpl    $1, %ebx
    movl    $66, %esi
    je      END

    movl    $0, %esi

END:
    #movl   %eax, %rsi
    movq    $printf_str, %rdi
    movq    $0, %rax
    call    printf
    ret
```

if…else 형태

어셈블리어 함수

- 어셈블리어에서 함수는 아래와 같이 선언한다.
  - .type 함수 명, @function
- 함수의 호출은 아래와 같다.
  - call 함수 명
- 함수의 인자는 정해진 레지스터에 순서대로 저장된다.
  - 인자가 너무 많아 레지스터를 다 사용하면 어떻게 저장될까?

CNU Embedded System Laboratory

실습 8. 함수 이용

- add_func 함수를 호출 후 결과 값 출력
- main 함수에서 다음과 같이 add_func 함수 호출

```
movq    val1, %rsi
movq    val2, %rdx
call    add_func
```

- add_func의 내용은 다음 코드와 같다.

```
int add_func(int a, int b){
        return (a + b);
}
```

- 인자의 값은 사전에 정의된 레지스터 순서를 따른다
  - rdi, rsi, rdx, rcx 등등
- %rax 에 결과 값을 넣으면 결과가 반환된다.

```
.section .data
message :
    .string "%d + %d = %d ₩n"
val1 :
    .int 100
val2 :
    .int 200

.section .text
.global main
main :
    movq    val1, %rsi
    movq    val2, %rdx
    call    add_func

    movq    %rax, %rcx
    movq    val1, %rsi
    movq    val2, %rdx
    movq    $message, %rdi
    movq    $0, %rax
    call    printf

    ret

.type add_func, @function
add_func:
    movq    %rsi, %rax
    addq    %rdx, %rax
    ret
```

함수 이용

```
.section .data
message :
    .string "%d, %d ₩n"
val1 :
    .int 100
val2 :
    .int 200

.section .text
.globl main
main :
    movq    val1, %rsi
    movq    val2, %rdx

    movq    $message, %rdi
    movq    $0, %rax
    call    printf

    movq    val1, %rsi
    movq    val2, %rdx

    call    swap

    movq    val1, %rsi
    movq    val2, %rdx
    movq    $message, %rdi
    movq    $0, %rax
    call    printf

    ret

.type swap, @function
swap:
    movq    val1, %rcx  # temp = val1
    movq    %rdx, val1  # val1 = val2
    movq    %rcx, val2  # val2 = temp

    ret
```

- swap 함수를 호출 후 결과를 출력
- 다음과 같은 명령어를 통해 값이 변경된다

```
movq    val1, %rcx
movq    %rdx, val1
movq    %rcx, val2
```

- 실행 결과

```
sys00@localhost:~/6lab$ ./ex02
100, 200
200, 100
```

CNU Embedded System Laboratory

참고 1. $의 사용 (1/2)

- $는 상수를, %는 레지스터를 표시한다.
  - ex) movq $0, %rsi
- 변수와 사용된다면?

: 우선 변수는 프로그래머가 이해하기 쉽게 라벨로 표시한 것으로, 컴퓨터 입장에서는 단순 주소에 지나지 않는다.

```
.section .data
message :
    .string "%d + %d = %d ₩n"
val1 :
    .int 100
val2 :
    .int 200
```

이 경우, data **영역에** val1 **의 위치에** int **형태로 메모리를 잡아** 100**으로 초기화해 준 것으로 볼 수 있다.**

```
movq    val1, %rsi
```

r: 64bit

즉, 좌측 명령어는 (주소) val1 **에 위치한 값** (100) **을** rsi **로 이동시키는 명령어이다.**

```
movl    $val1, %esi
```

e: 32bit

변수 앞의 $**는** &(**주소**)**를 의미한다. 따라서** $val1 **은** val1(**주소**) **자체를 의미하게 된다. 즉,** %esi **에는** val1 **의 주소값이 이동된다.**

CNU Embedded System Laboratory

참고 1. $의 사용 (2/2)

2. GDB를 이용해 값을 확인하면 보다 이해가 쉽다.

```
(gdb) l
15
16      .section .text
17      .globl  main
18      main:
19              movl    $val1, %esi
20              movl    $val2, %edx
21              movq    $scanf_str, %rdi
22
23              movq    $0, %rax
24              call    scanf
```

```
(gdb) disas
Dump of assembler code for function main:
=> 0x000000000040057d <+0>:     mov    $0x60107d,%esi
   0x0000000000400582 <+5>:     mov    $0x601081,%edx
   0x0000000000400587 <+10>:    mov    $0x601048,%rdi
   0x000000000040058e <+17>:    mov    $0x0,%rax
   0x0000000000400595 <+24>:    callq  0x400480 <scanf@plt>
   0x000000000040059a <+29>:    mov    0x60107d,%esi
   0x00000000004005a1 <+36>:    mov    0x601081,%edx
   0x00000000004005a8 <+43>:    cmp    %edx,%esi
   0x00000000004005aa <+45>:    je     0x4005c0 <equal>
   0x00000000004005ac <+47>:    mov    $0x601063,%rdi
   0x00000000004005b3 <+54>:    mov    $0x0,%rax
```

```
Breakpoint 1, main () at ex04.s:19
19              movl    $val1, %esi
4: /x $esi = 0xffffe668
3: &val1 = (<data variable, no debug info> *) 0x60107d
2: val1 = 0
(gdb) n

Breakpoint 2, main () at ex04.s:20
20              movl    $val2, %edx
4: /x $esi = 0x60107d
3: &val1 = (<data variable, no debug info> *) 0x60107d
2: val1 = 0
```

CNU Embedded System Laboratory

참고 2. rax, floating point?

- C library 를 호출할 때, floating point 를 몇 개 사용할 것인지에 대해 결정해 줘야한다. 이때 rax 에 저장하여 전달한다.

```c
#include <stdio.h>

int main()
{
    int a = 10;

    printf("%d ₩n", a);

    return 0;
}
```

```
.LC0:
    .string "%d ₩n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB24:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    $10, %edx
    movl    $.LC0, %esi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk
    movl    $0, %eax
```

```c
#include <stdio.h>

int main()
{
    float a = 10.0;

    printf("%f ₩n", a);

    return 0;
}
```

```
.LC1:
    .string "%f ₩n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB24:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movsd   .LC0(%rip), %xmm0
    movl    $.LC1, %esi
    movl    $1, %edi
    movl    $1, %eax
    call    __printf_chk
    movl    $0, %eax
```

```c
#include <stdio.h>

int main()
{
    float a = 10.0;
    float b = 11.0;

    printf("%f %f ₩n", a, b);

    return 0;
}
```

```
.LC2:
    .string "%f %f ₩n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB24:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movsd   .LC0(%rip), %xmm1
    movsd   .LC1(%rip), %xmm0
    movl    $.LC2, %esi
    movl    $1, %edi
    movl    $2, %eax
    call    __printf_chk
```

CNU Embedded System Laboratory

참고 3 - 레지스터

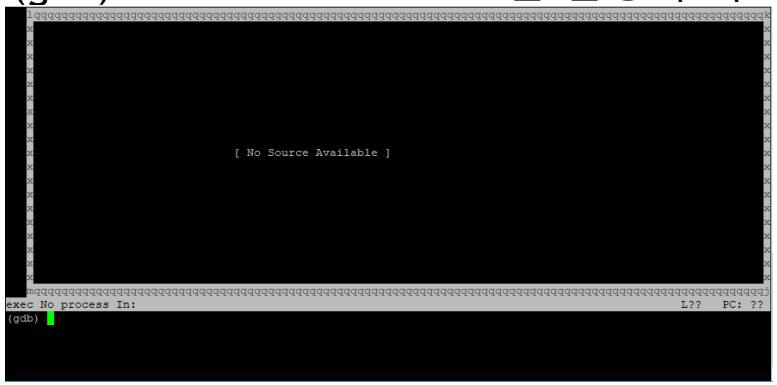
- 레지스터는 산술 연산, 논리 연산, 전송 조작을 행할 때 데이터나 명령을 일시적으로 기억해 두는 장소이다.
- 교재 180page 참조

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

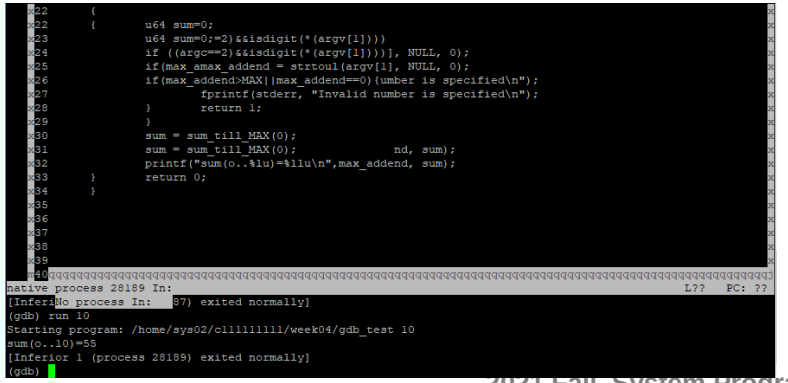
GDB

참고 4 - GDB TUI(Text User Interface)란? (1/4)

- GDB 사용의 편의를 돕기 위한 시각 옵션이 TUI(Text User Interface) 이다.
  - (gdb) tui enable // TUI를 활성화 시켜준다.



  - (gdb) layout src // source code를 상시로 볼 수 있는 창이 생긴다.
  - (gdb) run // 디버깅을 시작하면 실행하고 있는 코드의 내용이 출력.



시각적 도움으로
디버깅이 수월해 진다.

CNU Embedded System Laboratory

참고 4 - GDB TUI – layout (2/4)

- GDB 사용의 편의를 돕기 위한 시각 옵션이 TUI(Text User Interface) 이다.
  - 디버깅 정보들을 분리된 창에서 상시로 확인 할 수 있게 해준다.
  - Source file, Assembly, register info, gdb command 총 4가지 종류의 정보를 띄울 수 있다.

- TUI 시작하기 - 레이아웃
  - gdb를 실행할 때 gdb -tui 옵션을 주고 시작 할 수 있고,
  - gdb를 실행 한 후에는 커멘드 창에
    - tui enable : TUI를 활성화 한다.
    - tui disable : TUI를 비활성화 한다.

    - layout src : Source file 윈도우를 보여준다.
    - layout asm : Assembly 윈도우를 보여준다.
    - layout regs : Register 윈도우를 보여준다.

    - layout split : Src/Asm의 윈도우를 분할하여 2개의 윈도우를 보여준다.
    - layout next : 다음 레이아웃으로 윈도우를 전환한다. (미리 정의된 레이아웃을 제공함)
    - layout prev : 이전 레이아웃으로 윈도우를 전환한다. (미리 정의된 레이아웃을 제공함)

참고 4 - GDB TUI – Focus / 화면조정 (3/4)

- TUI - Focus
  - 현재 출력된 레이아웃의 화면 이동을 하고 싶을 때, 윈도우 선택 기능
    - focus src : Src 윈도우를 선택 한다. (방향키를 통해 src 코드를 확인할 수 있다)
    - focus asm : Asm 윈도우를 선택 한다. (방향키를 통해 asm 코드를 확인할 수 있다)
    - focus regs : Regs 윈도우를 선택 한다.
    - focus cmd : 커멘드 윈도우를 선택한다.
  - 화면 갱신
    - Tui를 사용하다 보면 윈도우들간에 간섭이 생기거나 텍스트들이 넘쳐서 출력되는 경우가 발생한다.
    - refresh : 윈도우/화면을 정리해 주는 명령어가 refresh 이다.
  - 코드와 실행 위치 동기화
    - update : 소스 윈도우와 현재 실행 위치를 업데이트 해준다.
  - 화면 크기 조정
    - info win : 현재 디스플레이 된 윈도우와 그 윈도우 사이즈를 보여준다.
    - winheight name +count : name의 윈도우 크기를 count 만큼 증가시킨다. (ex : winheight src +1)
    - winheight name -count : name의 윈도우 크기를 count 만큼 증가시킨다. (ex : winheight asm -1)

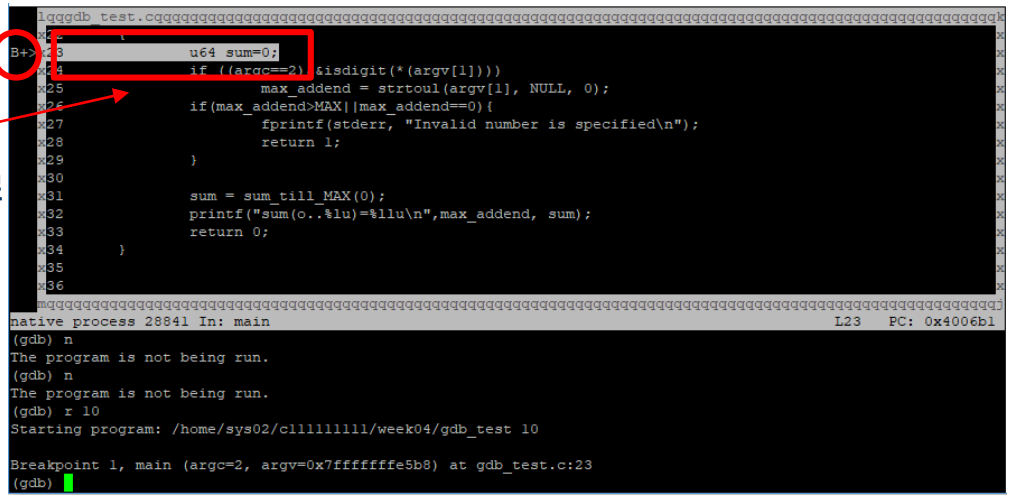
CNU Embedded System Laboratory

참고 4 - GDB TUI 따라하기 (4/4)

- gdb_test.out 을 가지고 tui를 따라해보자.
  - $ gdb gdb_test.out
  - (gdb) layout asm
  - (gdb) run 1 // 프로그램 실행시 src asm으로 코드가 출력된다.
  - (gdb) b main // main 함수로 breakpoint를 잡아준다.
  - (gdb) b sum_till_MAX // sum_till_MAX 함수로 breakpoint를 잡아준다.
  - (gdb) run 10 // 프로그램을 실행하여 디버깅을 진행한다.
  - (gdb) n // 프로그램을 한 줄씩 실행한다.
    // 이때 프로그램 실행 위치를 src 윈도우에서도 따라가므로 디버깅이 쉬워진다

Break point**를 표시해준다.**

현재 실행 위치를 음영으로 강조해주므로 디버깅이 쉬워진다.



CNU Embedded System Laboratory

GDB – 메모리 상태 검사

- 메모리 상태 검사
  - -g 옵션을 사용하지 않고 컴파일 한 실행 파일의 디버깅에 사용한다.
  - x/[출력 횟수] [출력 형식] [출력 단위] [출력 위치]

```
(gdb) x/10xb main
0x4004ed <main>:        0x55    0x48    0x89    0xe5    0x48    0x83    0xec    0x10
0x4004f5 <main+8>:      0xc7    0x45
(gdb) x/8xw main
0x4004ed <main>:        0xe5894855      0x10ec8348      0x0af845c7      0xc7000000
0x4004fd <main+16>:     0x0000fc45      0x458b0000      0xe8c789f8      0x0000000a
(gdb)
```

main의 주소 부터 16진수로 1바이트 씩 10개를 출력

main의 주소 부터 16진수로 4바이트 씩 8개를 출력

GDB – 메모리 상태 검사

출력 형식	설 명
t	2진수로 출력한다.
o	8진수로 출력한다.
d	부호가 있는 10진수(int)로 출력한다.
u	부호가 없는 10진수(unsigned int)로 출력한다.
x	16진수로 출력한다.
c	최초 1바이트 값을 문자 형으로 출력한다.
f	부동 소수점 값 형식으로 출력한다.
a	가장 가까운 심볼의 오프셋을 출력한다.
s	문자열로 출력한다.
i	어셈블리형식으로 출력한다.

GDB – 메모리 상태 검사

출력 단위	설 명
b	1 바이트 단위 (byte)
h	2 바이트 단위 (half word)
w	4 바이트 단위 (word)
g	8 바이트 단위 (giant word)

CNU Embedded System Laboratory

GDB – 어셈블리 코드 보기

- gdb에서 어셈블리 코드를 볼 수 있다.
  - disas [함수 명]: 함수의 어셈블리 코드를 출력한다.
  - disas [시작 주소] [끝 주소]: 주소 범위의 어셈블리 코드를 출력한다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     sub    $0x10,%rsp
   0x00000000004004f5 <+8>:     movl   $0xa,-0x8(%rbp)
   0x00000000004004fc <+15>:    movl   $0x0,-0x4(%rbp)
   0x0000000000400503 <+22>:    mov    -0x8(%rbp),%eax
   0x0000000000400506 <+25>:    mov    %eax,%edi
   0x0000000000400508 <+27>:    callq  0x400517 <sum_till_MAX>
   0x000000000040050d <+32>:    mov    %eax,-0x4(%rbp)
   0x0000000000400510 <+35>:    mov    $0x0,%eax
   0x0000000000400515 <+40>:    leaveq
   0x0000000000400516 <+41>:    retq
End of assembler dump.
(gdb)
```

CNU Embedded System Laboratory

GDB 사용 방법

1. gdb를 이용한 레지스터 값 확인
   I. gdb에서 p(print)명령어를 이용하여 레지스터의 값을 확인 할 수 있다.
   II. 사용법: p $[레지스터 명]

```
(gdb) b 15
Breakpoint 1 at 0x40051e: file gdb_test.c, line 15.
(gdb) r
Starting program: /home/sys02/sys02/gdb_test

Breakpoint 1, sum_till_MAX (max=10) at gdb_test.c:16
16              int i = 0;
(gdb) p $rax
$1 = 10
(gdb) p $rbx
$2 = 0
(gdb) p $rcx
$3 = 0
```

프로그램 실행

레지스터 값 확인

CNU Embedded System Laboratory

GDB 사용 방법

2. 레지스터 값 전체를 한번에 확인하는 명령어는 다음과 같다.
   I. 사용법: info register
      - 간단하게 i r이라고 입력해도 동일하게 동작한다.

```
(gdb) info register
rax            0xa      10
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffe698   140737488348824
rsi            0x7fffffffe688   140737488348808
rdi            0xa      10
rbp            0x7fffffffe580   0x7fffffffe580
rsp            0x7fffffffe580   0x7fffffffe580
r8             0x7ffff7dd4e80   140737351863936
r9             0x7ffff7dea530   140737351951664
r10            0x7fffffffe430   140737488348208
r11            0x7ffff7a36e50   140737348070992
r12            0x400400 4195328
r13            0x7fffffffe680   140737488348800
r14            0x0      0
r15            0x0      0
rip            0x40051e 0x40051e <sum_till_MAX+7>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
---Type <return> to continue, or q <return> to quit---
gs             0x0      0
```

과제 :
Bomb LAB

조교의 지시 전에 시작하지 마세요.

CNU Embedded System Laboratory

Bomb Lab - 소개

1. Bomb Lab은 여러 단계로 이루어진 프로그램이다.

2. 각 단계마다 폭탄을 해체할 수 있는 암호를 입력해야 한다.
   I. 만약 여러분이 정확한 암호를 입력한다면 해당 문구의 폭탄은 해체되며, 다음 단계로 넘어간다.
   II. 반면에 입력한 암호가 틀리면, 폭탄이 터지고 화면에 "BOOM!!!"이라는 메시지가 출력되고 프로그램이 종료된다.

```
sys01@eslab-server:~/sys01_bomb/bomb1$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!



a

BOOM!!!
The bomb has blown up.
Your instructor has been notified.
```

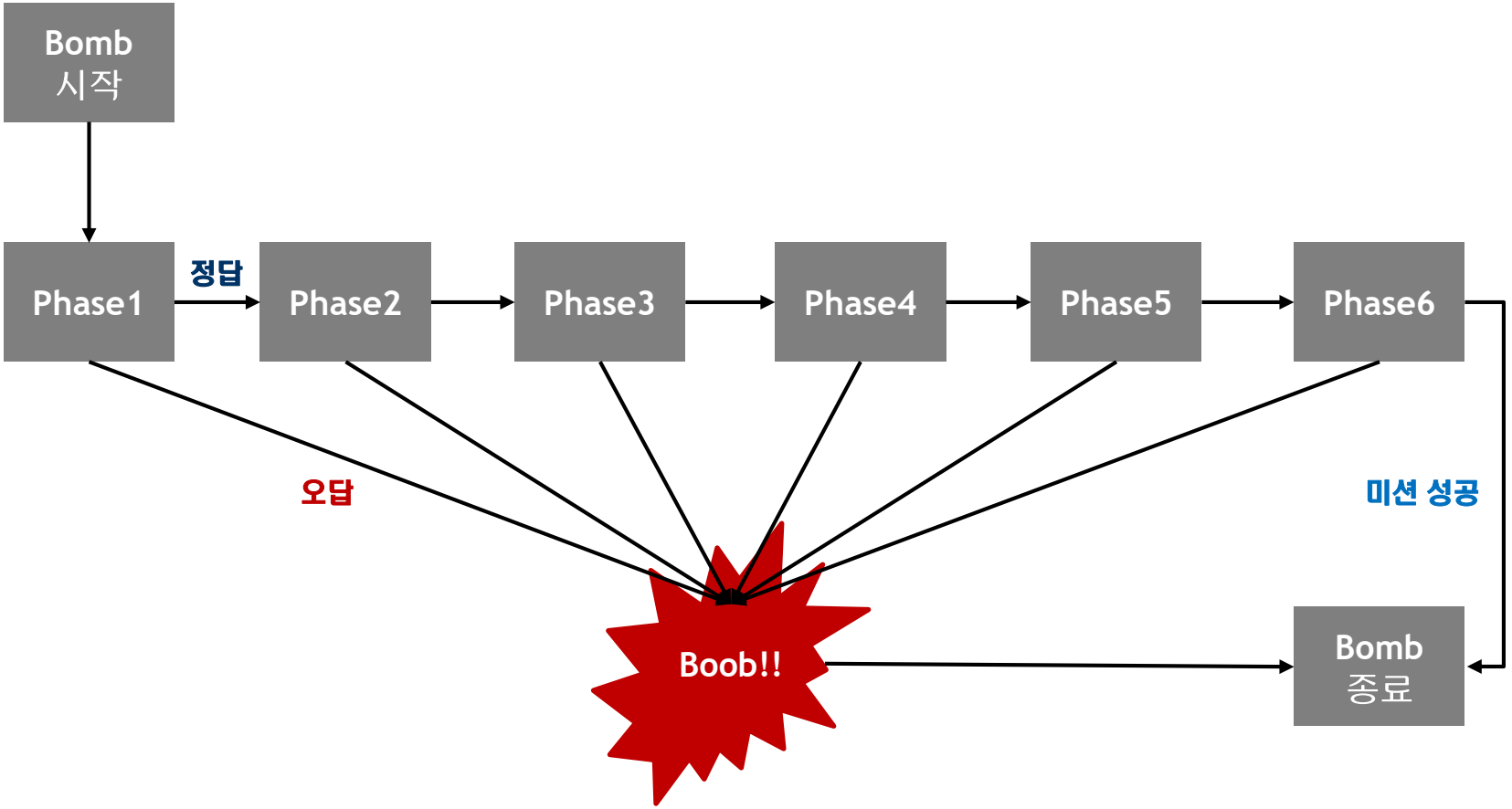
3. 모든 단계에서 정확한 암호를 입력해야 해당 폭탄이 완벽하게 해체된다.

Bomb Lab – 주의 사항

1. 본 프로그램은 http://125.6.36.240:10100/scoreboard 서버에서만 동작하도록 설정되어 있습니다.
2. 모든 사람의 폭탄 해체 암호는 프로그램에 의해 각기 다른 방식으로 생성됩니다.
3. 모든 상황은 서버를 통해 상시 모니터링 되므로 부정행위의 소지가 있는 행동에 각별히 주의바랍니다.
    I. 부정한 방법(바이너리 해킹 등)으로 해체 시도 시, 0점 처리됨과 동시에 기존 과제 모두 0점
4. 받은 Bomb 파일의 관리소홀로 인해 삭제될 경우, 복구가 불가하여 0점 처리 될 수 있으니 주의하시길 바랍니다.
5. Bomb의 해체/폭발 정보는 자동으로 서버로 전송되어 점수가 계산됩니다. 주의사항을 위반하여 발생한 문제에 대해서는 각자가 책임지는 것을 원칙으로 합니다.

CNU Embedded System Laboratory

Bomb Lab – 동작 구조



Bomb Lab - 진행

1. bomb 파일을 gdb를 활용하여, 폭탄의 내용 구성을 확인한 후, 도구들을 이용하여 분석하고 암호를 알아내야 한다.
2. ./bomb을 수행시키면 각 단계별로 암호를 입력 하여 다음 단계로 갈 수 있으며, 다음과 같이 모든 답을 한꺼번에 입력할 수 도 있다.
    I. ./bomb solution.txt
        - solution.txt는 각 단계별 정답을 enter로 구분하여 입력한 파일
        - gdb에서도 사용 가능하다.
            - gdb bomb solution.txt
3. 실행 후 ctrl+c 명령을 이용해서 취소가 가능하므로 실수로 폭탄이 터지지 않도록 주의!

CNU Embedded System Laboratory

Bomb Lab – 폭탄 해체(gdb)

1. 앞에서 보였던 objdump를 이용하는 것과 마찬가지로 gdb를 통해서도 어셈블리어 코드를 확인 할 수 있다.
   - disassemble 명령

```
(gdb) disassemble phase_1
Dump of assembler code for function phase_1:
   0x0000000000400f90 <+0>:     sub    $0x8,%rsp
   0x0000000000400f94 <+4>:     mov    $0x402670,%esi
   0x0000000000400f99 <+9>:     callq  0x401398 <strings_not_equal>
   0x0000000000400f9e <+14>:    test   %eax,%eax
   0x0000000000400fa0 <+16>:    je     0x400fa7 <phase_1+23>
   0x0000000000400fa2 <+18>:    callq  0x401671 <explode_bomb>
   0x0000000000400fa7 <+23>:    add    $0x8,%rsp
   0x0000000000400fab <+27>:    retq
End of assembler dump.
(gdb)
```

2. 해당 명령어는 gdb에서 objdump와 같은 기능을 하는 명령어이다.
   I. disassemble[함수 명]: 함수의 어셈블리 코드를 출력한다.
   II. disassemble[주소 1][주소 2]: 주소 1 ~ 주소 2 범위 사이의 어셈블리 코드를 출력한다.

CNU Embedded System Laboratory

Bomb Lab – Phase 1

1. phase 1의 구조를 보면 strings_not_equal 함수를 호출하는 것을 볼 수 있다. 이를 통해 어떠한 문자열을 입력 받아서 비교를 한 뒤, 오답일 경우 explode_bomb을 호출한다는 것을 추측할 수 있다.

```
(gdb) disassemble phase_1
Dump of assembler code for function phase_1:
   0x0000000000400f90 <+0>:     sub    $0x8,%rsp
   0x0000000000400f94 <+4>:     mov    $0x402670,%esi
   0x0000000000400f99 <+9>:     callq  0x401398 <strings_not_equal>
   0x0000000000400f9e <+14>:    test   %eax,%eax
   0x0000000000400fa0 <+16>:    je     0x400fa7 <phase_1+23>
   0x0000000000400fa2 <+18>:    callq  0x401671 <explode_bomb>
   0x0000000000400fa7 <+23>:    add    $0x8,%rsp
   0x0000000000400fab <+27>:    retq
End of assembler dump.
(gdb)
```

2. 비교하는 값을 찾아야 하는데, 함수의 앞 부분에서 %esi의 값에 $0x402670의 값을 옮기는 것을 볼 수 있다. 이 부분의 값을 보면 아래와 같다.

```
(gdb) x/s 0x402670
0x402670:       "We [redacted]"
(gdb)
```

CNU Embedded System Laboratory

Bomb Lab – Phase 1

3. 앞에서 찾아낸 문자열을 입력하면 아래와 같이 다음 단계로 넘어간다.

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

정답 입력

```
Phase 1 defused. How about the next one?
```

4. 오답일 경우 아래와 같이 폭탄이 터지게 된다.

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

오답 입력

```
BOOM!!!
The bomb has blown up.
Your instructor has been notified.
```

Bomb Lab – 풀이 방법

1. 결국 각 단계의 폭탄을 해체하기 위한 암호를 찾기 위해서는 어셈블리어로 변환된 코드의 구조를 이해해야 한다.

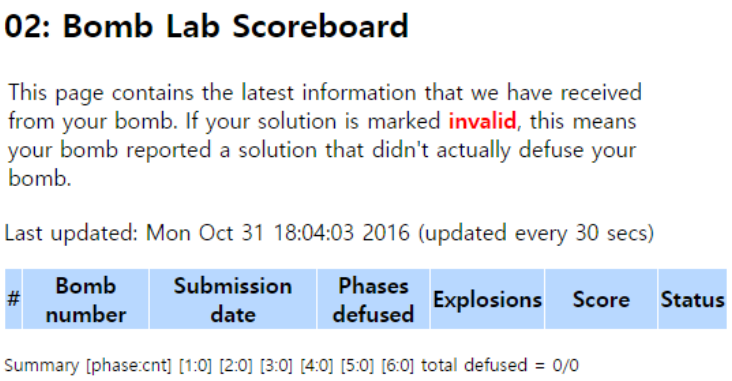
2. 그리고 나서 의심 가는 부분을 gdb를 통해 값을 가져와 확인해야 한다.

CNU Embedded System Laboratory

Bomb Lab – 점수 계산

1. 아래의 웹 페이지를 통해 실시간으로 각자의 진행상황(해체/폭발)을 확인할 수 있습니다.
   I. http://125.6.36.240:10100/scoreboard
   II. 폭탄을 모두 해체할 경우 60점
   III. 폭탄이 터진 경우 2회당 -1점
      1) 2회: -1점, 1회 : -0점
   IV. Hidden phase : 10점

2. Time Attack 방식으로 채점, 모든 폭탄(6 단계)을 제거했을 경우 완료 메일을 조교에게 보낸다(사진캡처 有).
   I. clearlyhunch@gmail.com
   II. 메일 제목: [sys01]폭탄완료_학번_이름



02: Bomb Lab Scoreboard

This page contains the latest information that we have received from your bomb. If your solution is marked **invalid**, this means your bomb reported a solution that didn't actually defuse your bomb.

Last updated: Mon Oct 31 18:04:03 2016 (updated every 30 secs)

#	Bomb number	Submission date	Phases defused	Explosions	Score	Status

Summary [phase:cnt] [1:0] [2:0] [3:0] [4:0] [5:0] [6:0] total defused = 0/0

과제

1. Bomb Lab 기간
   I. 10월 18일 ~ 11월 1일 (2주)
   II. 보고서 마감은 11월 1일 10시 59분
   III. 추가제출 마감은 11월 3일 23시 59분
   IV. 11월 3일 23시 59분에 bomb lab 서버 종료 (시험공부)

2. Bomb Lab 보고서
   I. 결과 화면을 붙임(폭탄 해체 화면과 웹 페이지에서의 본인 결과)
   II. 결과 화면에 학번이 보이도록 캡쳐
   III. 풀이 과정에 대한 자세한 설명(각 단계별 설명 + a)
   IV. 뒷장의 보고서 양식 참조

CNU Embedded System Laboratory

제출 사항

- 사이버캠퍼스에 제출
  - PDF 파일로 제출
    - 한글의 다른 이름으로 저장 기능 활용
  - 파일 제목: [sys01]HW02_학번_이름.pdf
  - 보고서는 제공된 양식 사용(임의로 보고서양식을 만들지 말 것)
- Bomb Lab 종료일시
  - 2021년 11월 3일 수요일 23시 59분 59초까지
- 제출일자
  - 사이버 캠퍼스: 2021년 11월 1일 월요일 10시 59분 59초까지