CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 5 : 3.6 제어문

http://eslab.cnu.ac.kr

* Some slides are from Original slides of RBE

전달사항

다음주 예습숙제 : 3.7~3.7.4, pp.253~263

오늘 배울 내용

제3장 프로그램의 기계수준 표현

- 데이터의 이동
- 제어 명령(3.6)
- 프로시져
- 보안 응용

현상태 점검 : 우리는 지금..

IA-32 어셈블리어를 배운다 – Forward Engineering

컴퓨터 구조를 이해한다

컴파일러가 만들어내는 어셈블리어를 분석한다 – Reverse Engineering

오늘의 주제 : 제어문

제어문 ? Control

제어문은 언제 사용하는가?

C 프로그램에서 사용하는 제어문에는 어떤 것들이 있는가 ?

어셈블리어에서는 저수준에서 이러한 제어문을 구현한다

- 기본구조는 조건결과에 의해 프로그램의 여러 부분으로 점프하는 구조다

상태 플래그 Condition Codes

1비트 플래그 Single Bit Registers

CF Carry Flag SF Sign Flag

ZF Zero Flag OF Overflow Flag

연산명령어의 결과로 세트됨

`addl` *Src,Dest*

C : `t = a + b` (a = Src, b = Dest)

- CF : MSB로부터 받아 올림이 발생한 경우에 1로 세팅됨
  - 비부호형 오버플로우 검출시 사용
- ZF : 연산의 결과 0이 된 경우 1로 세팅됨
- SF : 연산결과 음수가 생성된 경우 1로 세팅됨
- OF : 연산 결과 2의 보수 오버플로우(양/음 모두 포함)가 생긴경우

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

주의:플래그들은 `leal` 명령어로는 변경되지 않는다

상태 플래그 세팅 명령어 - CMP

Compare 명령

`cmpl` *Src2, Src1* # cmpl b, a

- `Src1 – Src2` 를 계산하지만 결과가 저장되지는 않는다
- CF : MSB로부터 받아 올림이 발생한 경우에 1로 세팅됨
  - 비부호형 오버플로우 검출시 사용
- ZF : 연산의 결과 0이 된 경우 1로 세팅됨
- SF : 연산결과 음수가 생성된 경우 1로 세팅됨
- OF : 연산 결과 2의 보수 오버플로우(양/음 모두 포함)가 생긴경우
  - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

달라진 조건에 유의 !

상태 플래그 세팅 명령어 - TEST

Test 명령어를 이용한 플래그 세팅

`testl` *Src2,S rc1*

- *Src1* & *Src2* 의 결과로 플래그가 세팅됨
  - *오퍼랜드를 마스크 시키는데 용이*
- 단순 and 연산을 수행하지만, 결과를 저장하지 않는다는 것이 차이
- ZF set when a&b == 0
- SF set when a&b < 0

상태 플래그의 이용명령 - SET

명령어 : SetX [R/M] (바이트 R or M)

- 상태 플래그들의 조합의 결과를 한 바이트 오퍼랜드에 저장
- 접미어는 연산의 길이가 아니다!!!!

Q. Why OF ?

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

상태 플래그 이용 명령 - SET

SetX 명령

- 상태 플래그들의 조합의 결과를 한 바이트에 저장
- 8 개의 바이트 레지스터를 이용
  - EAX, EBX, ECX, EDX를 이용
  - 다른 3 바이트들은 변화 없음
  - movzbl 를 일반적으로 이용

%eax	%ah	%al
%edx	%dh	%dl
%ecx	%ch	%cl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

뭐하는 프로그램?

```
int gt (int x, int y)
{
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

cmp명령은 비교순서에 유의!

Jump 명령어

jX Label

- 상태 플래그와 연관하여 조건(condition)이 참인 경우에 프로그램의 다른 곳으로 점프함

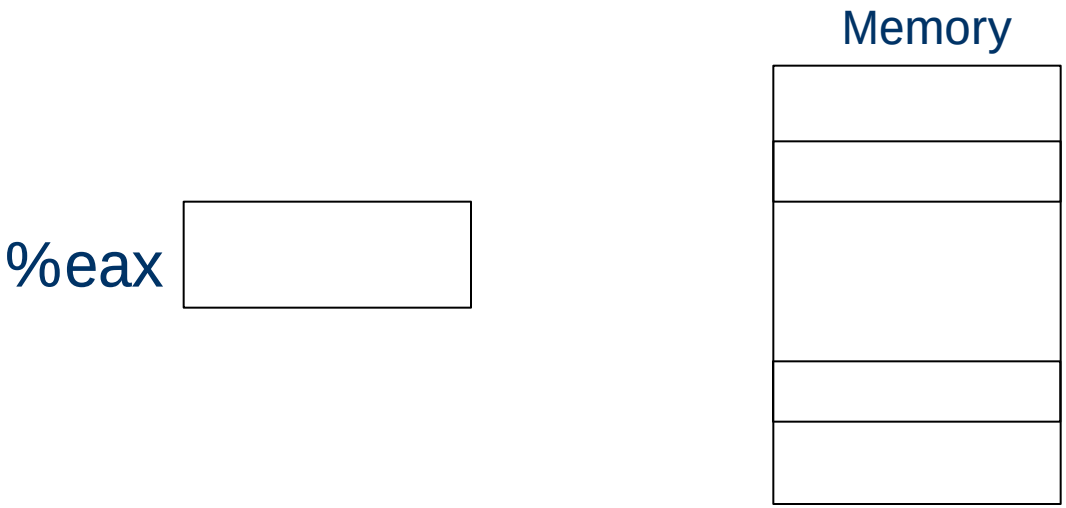
jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

간접 점프 !

jmp L1 무조건 L1으로 가라

jmp *%eax eax 레지스터에 저장된 주소로 가라

jmp *(%eax) ???



조건형 분기문의 예

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    cmpl %eax,%edx
    jle L9
    movl %edx,%eax
L9:

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

eax 의 사용에 주목!

조건형 분기문의 번역

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)goto done;
  rval = x;
done:
  return rval;
}
```

- C 에서는 goto 명령이 있다
  - 기계어 수준과 유사한 형식
- 일반적으로는 나쁜 프로그램 형태로 평가됨
- C 언어의 조건문은 조건형 점프와 무조건 점프를 이용해서 변역된다

```
   movl 8(%ebp),%edx     # edx = x
   movl 12(%ebp),%eax    # eax = y
   cmpl %eax,%edx        # x - y
   jle L9                # if x <= y goto done
   movl %edx,%eax        # eax = x
L9:                      # done:
```

x ≤ y 이면 실행안됨

좀 더 복잡한 조건형 분기문 번역의 예

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Set Up

Body1

Finish

Body2

어셈블리 번역의 구조를 닮은 C 코드

```
int goto_ad(int x, int y)
{
  int result;
  if (x<=y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

Body1

```
   # x in %edx, y in %eax
   cmpl   %eax, %edx     # Compare x:y
   jle    .L7            # <= Goto Else
   subl   %eax, %edx     # x-= y
   movl   %edx, %eax     # result = x
.L8:  # Exit:
```

Body2

```
.L7:  # Else:
   subl   %edx, %eax     # result = y-x
   jmp    .L8            # Goto Exit
```

연습문제 1. 분기문

```
    movl   $0x2, %eax          (1)
    movl   $0x4, %edx          (2)
    shl    $0x1, %eax          (3)
    cmpl   %eax, %edx          (4)
    jne    .L2                 (5)
.L1: nop                       (6)
.L2: movl %edx, %eax           (7)
```

1) 위 코드를 (3) 까지 실행할 때 %eax, 와 %edx 값은 각각 얼마인가?

2) (4) 를 실행한 후의 condition codes 의 레지스터 값은 각각 어떻게 되는가?

CF: ZF: SF: OF:

3) (5)를 실행한 다음 실행되는 코드는 무엇인가?

어셈블리어에서의 루프구현

- 루프가 무엇인가?
- 루프는 언제 사용하겠는가?
- C 언어에서 제공하는 루프명령은 무엇이 있는가?
- 어셈블리어에는 루프명령이 없다
  - 있을 수도 있다 T.T
  - 복잡하고 편리한 루프구조는 없다
- 어셈블리어에서 루프는 조건비교와 점프로 구현된다
- 대부분의 컴파일러에서는 루프문들을 do-while 형태로 구현한다

"Do-While" 루프 예

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- 루프문에 있어서 코드의 앞쪽으로 분기하는 형태
- while 조건이 만족되는 한 앞쪽으로 분기 실행

"Do-While" 루프의 컴파일

레지스터 할당

%edx	x
%eax	result

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
_fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x - 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

일반적인 Do-While 문의 번역

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

- *Body* 는 C 문장이 오면 됨

```
{
   Statement1;
   Statement2;
      ...
   Statementn;
}
```

- *Test* 는 수식으로 정수값을 출력함
  = 0 이면 거짓, ≠0 이면 참으로 해석

"While" 루프의 2가지 번역방법

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

방법 1 : 일단 goto로

```
int fact_while_goto(int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

컴파일러의 "While" 루프의 번역

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- 루프내에 do-while 구조를 이용
- 루프 이전에 추가의 조건 시험 문이 들어감

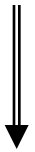
방법 2 : do-while 스타일로

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

일반적인 "While"문의 번역

C Code

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

"For" 루프

```
int result;
for (result = 1;
     p != 0;
     p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

```
for (Init; Test; Update )
    Body
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"For" → "While"

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

"For" 루프의 컴파일 과정

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
done:
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

연습문제 2. For 문

```
int fun_b(unsigned x) {
    int val = 0;
    int i;
    for ( ________ ; ________ ; ________ ) {
        ________
    }
    return val;
}
```

```
    x at %ebp+8
1       movl    8(%ebp), %ebx
2       movl    $0, %eax
3       movl    $0, %ecx
4    .L13:
5       leal    (%eax,%eax), %edx
6       movl    %ebx, %eax
7       andl    $1, %eax
8       orl     %edx, %eax
9       shrl    %ebx                Shift right by 1
10      addl    $1, %ecx
11      cmpl    $32, %ecx
12      jne     .L13
```

1) 위 c함수를 우측의 어셈블리로 번역하였을 때,
c 함수의 빈곳을 채우시오.

2) 이 함수가 무슨 계산을 수행하는지 설명하시오.

Switch 문

컴파일 방법

- 다수의 조건문으로 구성
  - case가 적으면 유리
  - case가 많으면 느려짐
- Jump Table 방법
  - Lookup branch target
  - 다수의 조건문이 불필요
  - case의 수가 작은 정수이면 구현가능
- GCC 에서는
  - case 문의 구조에 따라 번역방법을 결정

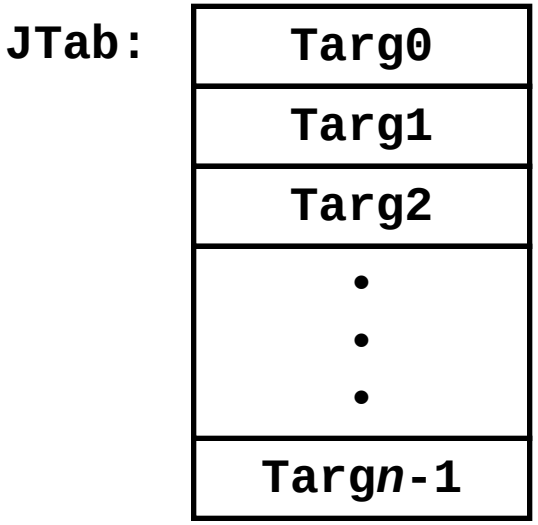
```
char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Jump Table의 구조

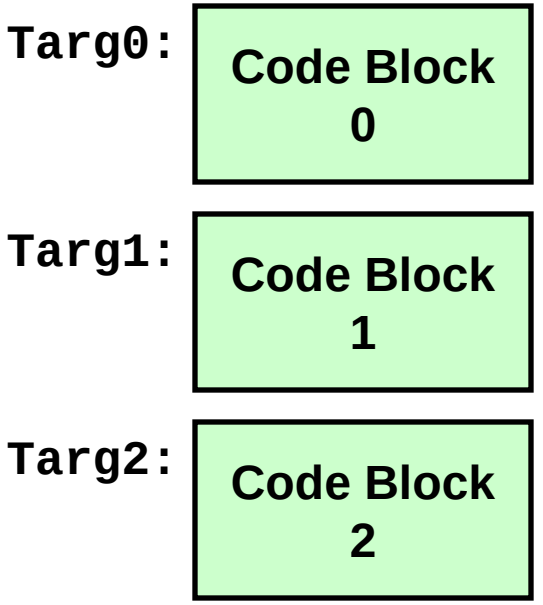
Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Approx. Translation

```
target = JTab[op];
goto *target;
```

Switch 문 예제

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    . . .
  }
}
```

Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

Setup:

```
unparse_symbol:
   pushl %ebp          # Setup
   movl %esp,%ebp      # Setup
   movl 8(%ebp),%eax   # eax = op
   cmpl $5,%eax        # Compare op - 5
   ja .L49             # If > goto done
   jmp *.L57(,%eax,4) # goto Table[op]
```

전체 코드

```
unparse_symbol:
    pushl %ebp                  # Setup
    movl %esp,%ebp              # Setup
    movl 8(%ebp),%eax           # eax = op
    cmpl $5,%eax                # Compare op - 5
    ja .L49                     # If > goto done
    jmp *.L57(,%eax,4)          # goto Table[op]

.L51:
    movl $43,%eax               # '+'
    jmp .L49
.L52:
    movl $42,%eax               # '*'
    jmp .L49
.L53:
    movl $45,%eax               # '-'
    jmp .L49
.L54:
    movl $47,%eax               # '/'
    jmp .L49
.L55:
    movl $37,%eax               # '%'
    jmp .L49
.L56:
    movl $63,%eax               # '?'
    # Fall Through to .L49
.L49:                           # Done:
    movl %ebp,%esp              # Finish
    popl %ebp                   # Finish
    ret                         # Finish
```

```
.section .rodata
   .align 4
.L57:
   .long .L51       #Op = 0
   .long .L52       #Op = 1
   .long .L53       #Op = 2
   .long .L54       #Op = 3
   .long .L55       #Op = 4
   .long .L56       #Op = 5
```

Setup 부 설명

Symbolic Labels

- .L*XX* 와 같은 레이블들은 어셈블러에 의해 주소로 변환된다

Table Structure

- 각 분기 대상 주소들은 4바이트이다
- Base address at `.L57`

Jumping

`jmp .L49`

- **점프할 목표는** `.L49`로 표시한다.

`jmp *.L57(,%eax,4)`

- **jump table 의 시작주소는** .L57이다
- **Register** %eax **는** op 값을 저장하고 있다
- **점프 테이블은 4바이트의 주소를 저장하고 있으므로 4의 배수로 오프셋을 표시**
- **최종 EA는** .L57 + op*4 로 계산됨

점프 테이블

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

Switch 문의 완성

```
.L49:                           # Done:
    movl %ebp,%esp              # Finish
    popl %ebp                   # Finish
    ret                         # Finish
```

Puzzle

- op 값이 잘못된 값이 들어갈 때, 리턴되는 값은 무엇인가?

Jump Table의 장점

- *k*-way branch in $O(1)$ operations (?)

Object Code

Setup

- Label .L49 는 0x804875c 로 정해진다
- Label .L57 는 0x8048bc0 로 정해진다

```
08048718 <unparse_symbol>:
8048718:55                  pushl  %ebp
8048719:89 e5               movl   %esp,%ebp
804871b:8b 45 08            movl   0x8(%ebp),%eax
804871e:83 f8 05            cmpl   $0x5,%eax
8048721:77 39               ja     804875c <unparse_symbol+0x44>
8048723:ff 24 85 c0 8b jmp     *0x8048bc0(,%eax,4)
```

```
08048718 <unparse_symbol>:
8048718:   55                      pushl  %ebp
8048719:   89 e5                   movl   %esp,%ebp
804871b:   8b 45 08                movl   0x8(%ebp),%eax
804871e:   83 f8 05                cmpl   $0x5,%eax
8048721:   77 39                   ja     804875c <unparse_symbol+0x44>
8048723:   ff 24 85 c0 8b          jmp    *0x8048bc0(,%eax,4)
8048730:   b8 2b 00 00 00    movl   $0x2b,%eax
8048735:   eb 25             jmp    804875c <unparse_symbol+0x44>
8048737:   b8 2a 00 00 00    movl   $0x2a,%eax
804873c:   eb 1e             jmp    804875c <unparse_symbol+0x44>
804873e:   89 f6             movl   %esi,%esi
8048740:   b8 2d 00 00 00    movl   $0x2d,%eax
8048745:   eb 15             jmp    804875c <unparse_symbol+0x44>
8048747:   b8 2f 00 00 00    movl   $0x2f,%eax
804874c:   eb 0e             jmp    804875c <unparse_symbol+0x44>
804874e:   89 f6             movl   %esi,%esi
8048750:   b8 25 00 00 00    movl   $0x25,%eax
8048755:   eb 05             jmp    804875c <unparse_symbol+0x44>
8048757:   b8 3f 00 00 00    movl   $0x3f,%eax
```

Object Code

Jump Table

- disassembled code 에는 나타나지 않는다
- GDB로 확인가능

```
gdb code-examples
(gdb) x/6xw 0x8048bc0
```

  - Examine 6 hexadecimal format "words" (4바이트씩 출력)
  - "help x" 명령을 사용하면 사용정보가 나온다

```
0x8048bc0 <_fini+32>:
  0x08048730
  0x08048737
  0x08048740
  0x08048747
  0x08048750
  0x08048757
```

데이터 세그먼트에서 Jump table 읽어내기

Jump Table 은 Read Only Data Segment (.rodata)에 저장되어 있다

- 여러분이 작성한 코드의 상수값들이 저장되는 구역

objdump로 조사할 수 있다

```
objdump code-examples –s –-section=.rodata
```

- 지정한 세그먼트의 데이터들을 볼 수 있다.

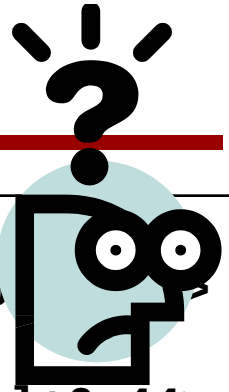
읽기 어려움

- reversed byte ordering

```
Contents of section .rodata:
 8048bc0 30870408 37870408 40870408 47870408  0...7...@...G...
 8048bd0 50870408 57870408 46616374 28256429  P...W...Fact(%d)
 8048be0 203d2025 6c640a00 43686172 203d2025   = %ld..Char = %
 …
```

- E.g., 30870408 => 0x08048730 로 해석해야 한다

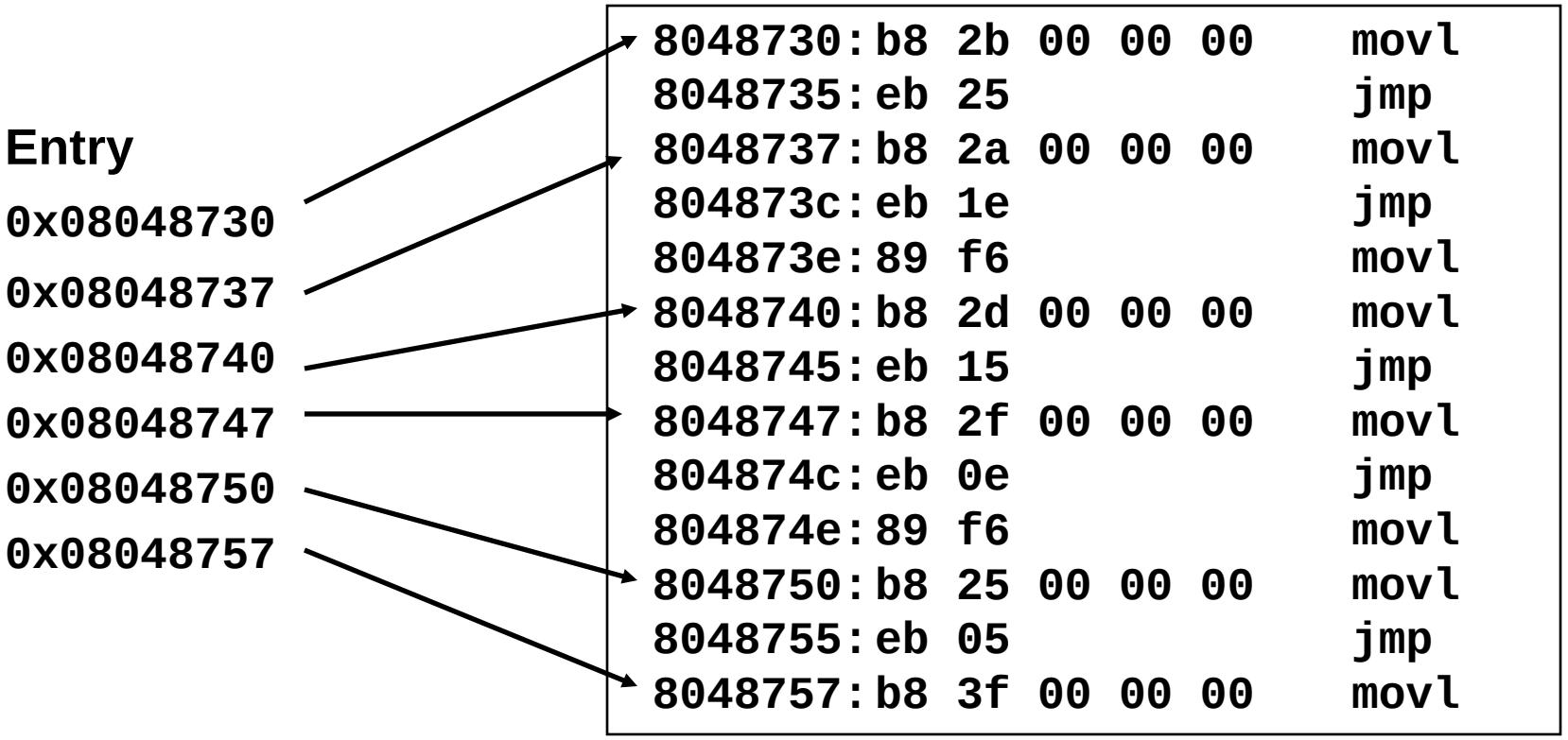
Target 디스어셈블 결과



```
8048730:b8 2b 00 00 00   movl   $0x2b,%eax
8048735:eb 25            jmp    804875c <unparse_symbo
8048737:b8 2a 00 00 00   movl   $0x2a,%eax
804873c:eb 1e            jmp    804875c <unparse_symbol+0x44>
804873e:89 f6            movl   %esi,%esi
8048740:b8 2d 00 00 00   movl   $0x2d,%eax
8048745:eb 15            jmp    804875c <unparse_symbol+0x44>
8048747:b8 2f 00 00 00   movl   $0x2f,%eax
804874c:eb 0e            jmp    804875c <unparse_symbol+0x44>
804874e:89 f6            movl   %esi,%esi
8048750:b8 25 00 00 00   movl   $0x25,%eax
8048755:eb 05            jmp    804875c <unparse_symbol+0x44>
8048757:b8 3f 00 00 00   movl   $0x3f,%eax
```

- movl %esi,%esi 은 아무 하는 일도 없는 명령어
- 왜 이런 짓을 ?

디스어셈블된 Target 과 테이블 주소



Sparse Switch 의 경우에는 ?

```
/* Return x/111 if x is multiple
   && <= 999.  -1 otherwise */
int div111(int x)
{
  switch(x) {
  case   0: return 0;
  case 111: return 1;
  case 222: return 2;
  case 333: return 3;
  case 444: return 4;
  case 555: return 5;
  case 666: return 6;
  case 777: return 7;
  case 888: return 8;
  case 999: return 9;
  default: return -1;
  }
}
```

- 점프 테이블을 사용하기에는 비현실적!
  - 1000개의 엔트리가 필요
- if-then-else 를 사용하면 최대 9번만 비교하면 된다

Sparse switch code

```
    movl 8(%ebp),%eax  # get x
    cmpl $444,%eax     # x:444
    je L8
    jg L16
    cmpl $111,%eax     # x:111
    je L5
    jg L17
    testl %eax,%eax    # x:0
    je L4
    jmp L14

    . . .
```

- x를 모든 값과 비교
- 그 결과에 따라 다른 곳으로 점프함

```
    . . .
L5:
    movl $1,%eax
    jmp L19
L6:
    movl $2,%eax
    jmp L19
L7:
    movl $3,%eax
    jmp L19
L8:
    movl $4,%eax
    jmp L19
    . . .
```

요약

C 언어의 제어문
- if-then-else
- do-while
- while
- switch

어셈블리어의 제어문
- jump
- Conditional jump

컴파일러
- 보다 복잡한 어셈블리 코드를 생성해 낸다

표준 기법
- 모든 루프는 do-while로 변환 된다
- Switch문은 점프테이블로 구성 될 수 있다