CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 5 : 3.6 제어문

http://eslab.cnu.ac.kr

* Some slides are from Original slides of RBE

강의 일정

주	날짜	강의실 (화)	날짜	실습실 (목)
1	9월 5일	1.Intro		
2	9월 12일	2.정수	9월 10일	리눅스 개발환경 익히기
3	9월 19일	3.부동소숫점	9월 17일	GCC & Make
4	9월 26일	추석휴일	9월 24일	추석휴일
5	10월 3일	개천절	10월 1일	(이론)4. 어셈1 – move
6	10월 10일	5.어셈2 – 제어문	10월 8일	데이타랩
7	10월 17일	6.어셈3 – 프로시져 I	10월 15일	어셈블리어/GDB
8	10월 24일	7.어셈3 – 프로시져 II	10월 22일	폭탄랩
9	10월 31일	중간고사(저녁 7시)	10월 29일	
10	11월 7일	8.프로세스 1	11월 5일	
11	11월 14일	9.프로세스 2	11월 12일	Tiny shell 1
12	11월 21일	10.시그널	11월 19일	Tiny shell 2
13	11월 28일	11.동적메모리 1	11월 26일	Tiny shell 3
14	12월 5일	12.동적메모리 2	12월 3일	Malloc lab1
15	12월 12일	기말고사(저녁7시)	12월 17일	Malloc lab2
16	12월 19일	Wrap-up/종강	12월 13일	Malloc lab3

오늘 배울 내용

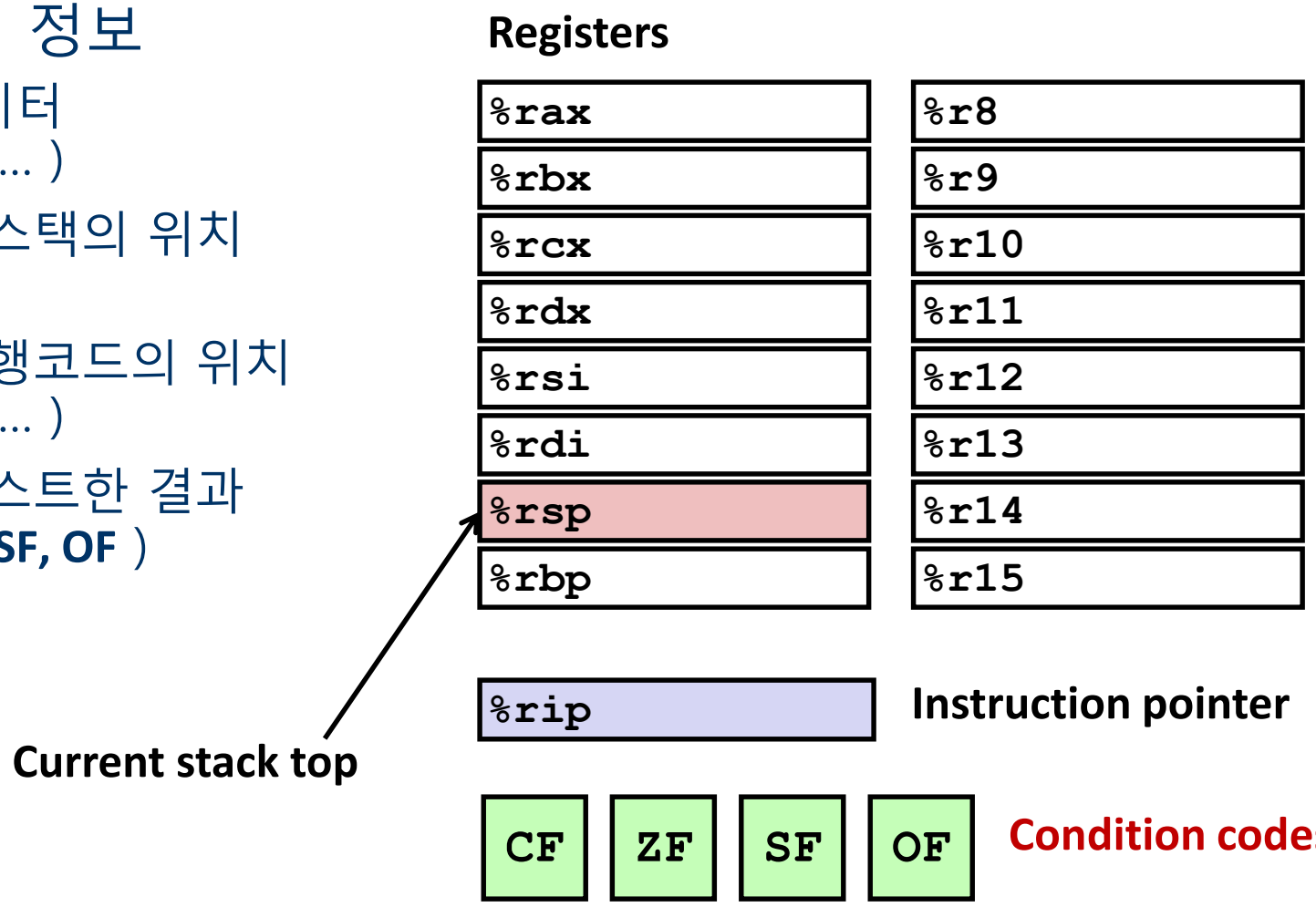
제3장 프로그램의 기계수준 표현

- 데이터의 이동
- 제어 명령(3.6)
- 프로시져
- 보안 응용

프로세서의 상태(x86-64)

실행하고 있는
프로그램의 정보

- 임시데이터
  (**%rax**, …)
- 런타임 스택의 위치
  (**%rsp**)
- 현재 실행코드의 위치
  (**%rip**, …)
- 현재 테스트한 결과
  (**CF, ZF, SF, OF**)



Condition Codes(조건코드, 간접세팅)

1비트 레지스터

- **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
- **ZF** Zero Flag **OF** Overflow Flag (for signed)

산술연산의 결과로 값이 바뀜

예제 : **`addq`** *Src,Dest* ↔ **`t = a+b`**

CF set 가장 중요한 비트(MSB)에서 캐리 발생 (unsigned overflow)

ZF set if **`t == 0`**

SF set if **`t < 0`** (as signed)

OF set 2의 보수(signed) 오버플로우 발생

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

`leaq` 명령어로는 값이 바뀌지 않음에 주의!

Condition Codes (직접세팅: 비교명령어)

비교명령어를 이용한 직접적인 값 변화

- **cmpq** *Src2*, *Src1*
- **cmpq b,a a-b** 를 계산하되, 결과를 저장하지 않는다

- **CF set** 가장 중요한 비트(MSB)에서 캐리가 발생하면 1로 설정됨(unsigned 비교에서 사용)
- **ZF set** if **a == b**
- **SF set** if **(a-b) < 0** (부호형)
- **OF set** if 2의 보수(signed) 오버플로우가 발생한 경우

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (직접세팅: Test 명령어)

Test 명령어를 이용한 직접 세팅

- **`testq`** *Src2*, *Src1*
  - **`testq b,a`** **`a&b`** 연산을 수행하지만, 결과를 저장하지 않는다

- *Src1* & *Src2* 결과에 따라 조건코드를 설정한다
- 오퍼랜드 중의 하나가 마스크mask로 이용할 때 유용하다

- **ZF set** **`a&b == 0`** 일때
- **SF set** **`a&b < 0`** 일때

조건 코드값 읽어오기

SetX 명령어

- 목적지의 하위 바이트를 조건코드 조합에 따라 0 또는 1로 설정
- 나머지 7바이트는 변경없다
- cmp 명령 실행 후에 적용

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)

x86-64 정수 레지스터

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- 하위 바이트를 사용할 수 있다

조건코드 읽어오기(계속)

SetX 명령어 :

- 조건코드 조합을 한 바이트로 표시

주소지정 가능한 바이트 레지스터 한 개를 사용

- 다른 바이트 값은 불변
- 대개 **movzbl** 추가로 사용

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

점프

jX Label

- 조건코드에 따라서 코드의 실행 위치를 이동

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)

조건부 분기예제(올드 스타일)

생성방법

```
hskim> gcc -Og -S -fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Goto 코드로 표현하기

C 언어는 **goto** 문을 사용할 수 있다

레이블로 표시된 위치로 점프하게 된다

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

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

일반적인 조건부 수식의 번역(분기문을 사용)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- then & else 수식에 대해 별도의 코드 영역을 생성
- 해당 코드부분만 실행

조건부 이동

조건부 이동명령어

- 표현 형태:
  if (Test) Dest ← Src
- 1995년 이후 x86 프로세서에서 지원
- GCC 에서도 사용하려고 함

왜?

- 분기문은 파이프라인의 인스트럭션 흐름을 매우 방해한다
- 조건부 이동명령은 제어의 이동이 필요 없다

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

조건부 이동명령 예제

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax  # x
    subq    %rsi, %rax  # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx  # eval = y-x
    cmpq    %rsi, %rdi  # x:y
    cmovle  %rdx, %rax  # if <=, result = eval
    ret
```

"Do-While" 루프 예제

C Code

```
long pcount_do
   (unsigned long x) {
   long result = 0;
   do {
     result += x & 0x1;
     x >>= 1;
   } while (x);
   return result;
}
```

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

인자 x에서 1이 몇 개인지 계산

조건부 분기를 이용해서 루프를 계속할지 빠져나올지 결정

"Do-While" 루프의 컴파일

Goto Version

```
long pcount_goto
   (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   rep; ret
```

일반적인 "Do-While" 의 번역

C Code

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

Goto Version

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

Body:

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

일반적인 "While" 문의 번역 #1

"중간으로 점프형태" 번역

-Og 를 사용

While version

```
while (Test)
   Body
```

Goto Version

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

While 루프 예제 #1

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
   (unsigned long x) {
   long result = 0;
   goto test;
 loop:
   result += x & 0x1;
   x >>= 1;
 test:
   if(x) goto loop;
   return result;
}
```

do-while 버전과 비교해보자

시작부의 goto는 루프를 test에서 시작한다

일반적인 "While"문의 번역 #2

While version

```
while (Test)
  Body
```

"Do-while" 로 변환
Used with **–O1**

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:
```

While 루프 예제 #2

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Do-While Version

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

최초의 조건이 루프로의 진입을 통제

"For" 루프

일반형

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

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

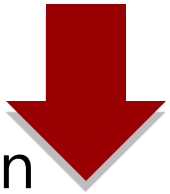
Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

"For" 루프 → While 루프

For Version

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



While Version

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

For-While 변환

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch 문 예제

- 다중 레이블 사용
  - case 5 & 6
- 통과 cases
  - case 2
- 빠진 cases
  - case 4

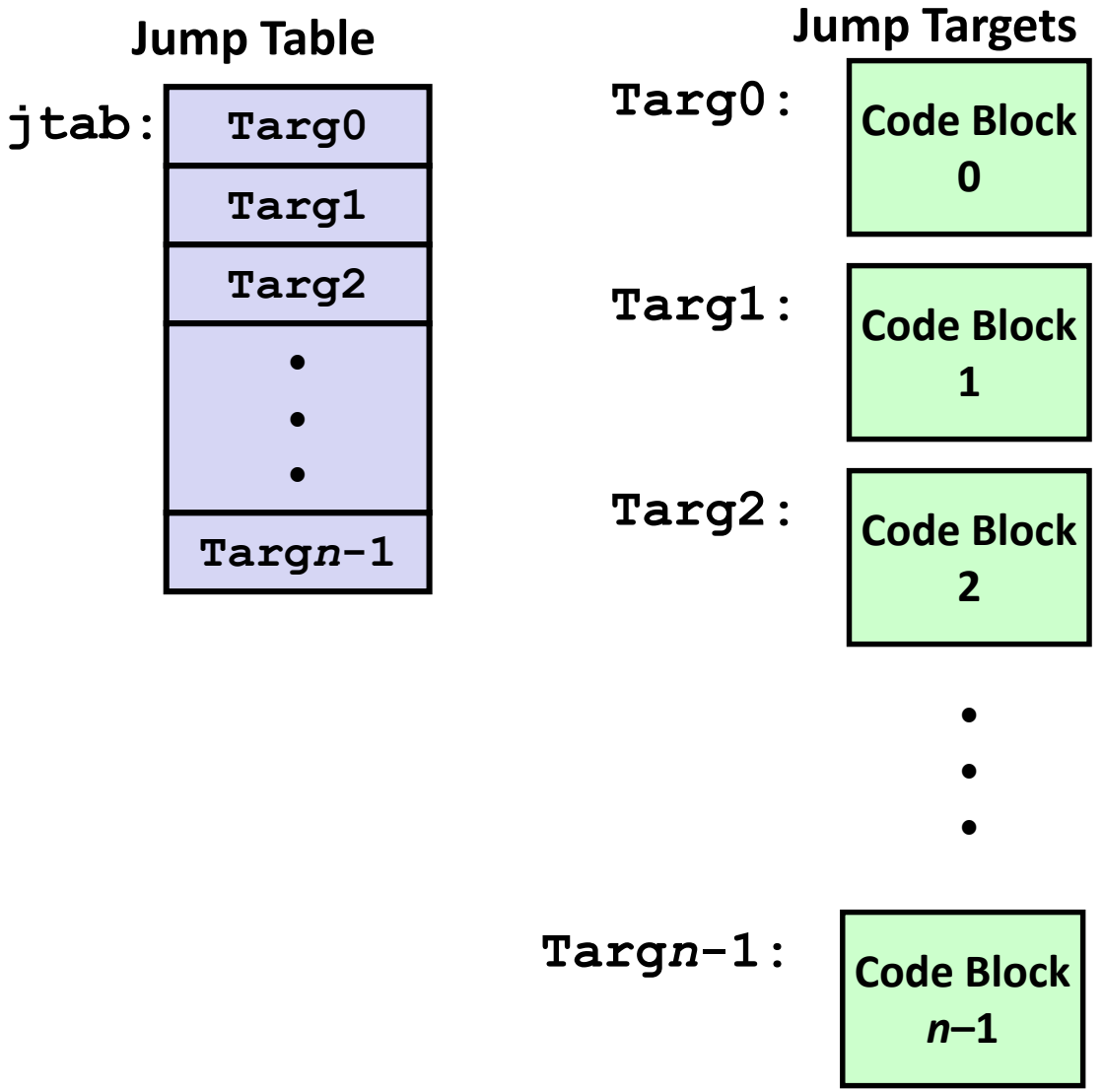
점프 테이블 구조

Switch 문

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

번역 (확장형 C)

```
goto *JTab[x];
```



Switch 문 예제

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Switch 문 예제

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

간접점프 →

데이터 섹션

테이블 구조

- 각 타겟은 8바이트를 필요로 함
- 시작주소는 **.L4**

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

점프하기

- 직접점프: **jmp .L8**
- 점프 대상은 레이블**.L8**로 표시

- 간접점프: **jmp *.L4(,%rdi,8)**
- 점프테이블의 시작: **.L4**
- 8의 배수로 증가해야 함(주소들이 8바이트이므로)
- 점프 타겟은 유효주소 **.L4 + x*8** 로부터 얻어짐
  - 0 ≤ **x** ≤ 6 에 대해서만 성립

점프 테이블

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```

코드 블록 (x == 1)

```
  switch(x) {
  case 1:         // .L3
        w = y*z;
        break;
    . . .
}
```

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value