시스템 프로그래밍 (2016)

강의 5 : 산술 명령과 제어문

* Some slides are from Original slides of
Bryant and O'Hallaron's Computer Systems:
A Programmer's Perspective

강의 일정

주	날짜	강의실	날짜	실습실
1	9월 1일(목)	**소개 강의**	9월 6일(화)	**리눅스 개발환경 익히기 (VI, 쉘 기본명령어들)**
2	9월 8일(목)	**정수 표현 방법**	9월 13일(화)	GCC & Make, shell script
3	9월 15일(목)	**추석 휴강**	9월 20일(화)	**C 복습과 GDB 사용하기 1 (소스 수준 디버깅)**
4	9월 22일(목)	**실수 표현 방법**	9월 27일(화)	**Data lab (GDB활용)**
5	9월 29일(목)	**어셈1 - 데이터이동**	10월 4일(화)	**어셈1 – move(실습),**
6	10월 6일(목)	**어셈2 – 제어문**	10월 11일(화)	**어셈2- 제어문 (실습)**
7	10월 13일(목)	**어셈3 – 프로시저**	10월 18일(화)	**어셈3-프로시저(실습)**
8	10월 20일(목)	**어셈보충/중간시험**	10월 25일(화)	**GDB 사용하기2(어셈수준)**
9	10월 27일(목)	**보안**(buffer overflow)	11월 1일(화)	Binary bomb 1 (GDB활용)
10	11월 3일(목)	**프로세스 1**	11월 8일(화)	Binary bomb 2 (GDB활용)
11	11월 10일(목)	**프로세스 2**	11월 15일(화)	Tiny shell 1
12	11월 17일(목)	**시그널**	11월 22일(화)	Tiny shell 2
13	11월 24일(목)	**동적메모리 1**	11월 29일(화)	Malloc lab1
14	12월 1일(목)	**동적메모리 2**	12월 6일(화)	Malloc lab2
15	12월 8일(목)	**기말시험**	12월 13일(화)	Malloc lab3

목차(제어문)

1. 주소계산 및 산술 논리 연산 명령어
2. 특수 연산(128비트 곱셈, 나눗셈연산 명령)
3. 제어문 … 상태플래그, condition code
4. Setx 명령
5. JMP, JX 명령어
6. CMOV 조건 데이터 이동 명령어
7. LOOP 문 (do-while, while, for문) 번역
8. switch 문 번역

1. 주소계산 및 산술 논리 연산 명령어

주소 계산 명령

- **`leaq` *Src*, *Dst***
  - *Src* 는 주소지정 모드 표현
  - *Dst* 는 Src로 지정된 주소값을 갖게됨

- 사용 예
  - 메모리 참조없는 주소 계산임
    - 예: **`p = &x[i];`** 이 명령을 번역하면 …
  - x + k*y 형태의 산술 수식 계산에 사용됨(상수 곱셈 수식)
    - K = 1, 2, 4, or 8
- 예

```
long m12(long x)
{
  return x*12;
}
```

컴파일러 생성 어셈 코드 :

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

연습문제. leaq 명령

- %rax = x, %rcx = y 일때, 다음 명령 수행 결과인 %rdx를 x, y의 수식으로 표시하시오

- leaq 9(%rax, %rcx, 2), %rdx

산술 연산 명령들 일부

- 피연산자가 두개의 명령들:

포맷		계산 내용	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***shlq***라고도함…
`sarq`	*Src,Dest*	Dest = Dest >> Src	산술
`shrq`	*Src,Dest*	Dest = Dest >> Src	논리
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- 인자 순서에 주의!
- 부호 있는 수와 없는 수 간의 구분이 없음(왜?)

산술 연산 명령들 일부

- 시프트 연산
  - SAL/SHL k, D D ← D <<k
  - SAR k, D D ← D >> k (산술시프트)
  - SHR k, D D ← D >> k (논리시프트)
- 단일 피연산자 명령

incq	*Dest*	*Dest = Dest + 1*
decq	*Dest*	*Dest = Dest – 1*
negq	*Dest*	*Dest = – Dest*
notq	*Dest*	*Dest = ~Dest*

- 더 많은 명령어들은 책이나 참고문헌...

산술연산 명령어 사용 사례

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq   %rcx, %rax
    ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

산술연산 명령어 사용 사례

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
    leaq    (%rdi,%rsi), %rax    # t1
    addq    %rdx, %rax           # t2
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx             # t4
    leaq    4(%rdi,%rdx), %rcx   # t5
    imulq   %rcx, %rax           # rval
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**.
%rdx	**t4** .
%rcx	**t5** .

연습

주소	값	레지스터	값
0x100	0xFF	rax	0x100
0x108	0xAB	rcx	0x1
0x110	0x13	rdx	0x3
0x118	0x11		
0x120	0x12		

명령	destination	값
Addq %rcx, (%rax)		
Subq %rdx, 8(%rax)		
Imulq $16, (%rax,%rdx,8)		
Incq 16(%rax)		
Decq %rcx		
Subq %rdx, %rax		

2. 특수 연산(128비트 곱셈, 나눗셈연산 명령)

특수 연산 명령어들(128bit연산)

명령어	효과	설명
Imulq S	R[%rdx]:R[%rax] ← S x R[%rax]	부호형 완전 곱셈
Mulq S	R[%rdx]:R[%rax] ← S x R[%rax]	비부호형 완전 곱셈
Cqto	R[%rdx]:R[%rax] ← SingExt(R[%rax])	Quad to Oct Word 변환
Idivq S	R[%rdx] ←R[%rdx]:R[%rax] mod S R[%rax] ←R[%rdx]:R[%rax] / S	부호형 나누기
Divq S	R[%rdx] ←R[%rdx]:R[%rax] mod S R[%rax] ←R[%rdx]:R[%rax] / S	비부호형 나누기

C 코드(128비트 곱셈 연산)

```
#include <inttypes.h> #128비트 지원안함.

typedef unsigned __int128 uint128_t; #gcc __int128 이용

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
        *dest = x * (uint128_t) y;
}
```

어셈 코드(128비트 곱셈연산)

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {

Dest in %rdi, x in %rsi, y in %rdx

```
store_uprod:
movq  %rsi, %rax
mulq  %rdx
movq  %rax, (%rdi)
movq  %rdx, 8(%rdi)
ret
```

C 코드(나눗셈연산예제)

```
void remdiv(long x, long y, long *qp, long *rp) {
        long q = x / y;
        long r = x % y;
        *qp = q;
        *rp = r;
}
```

어셈 코드(나눗셈연산예제)

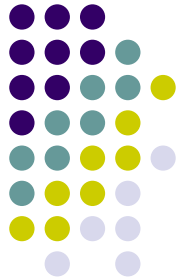
void remdiv(longx, longy, long *qp, long *rp) {
x in %rdi, y in %rsi, qp in %rdx, rp in %rcx

```
remdiv:
movq %rdx, %r8
movq %rdi, %rax
cqto
idivq %rsi
movq %rax, (%r8)
movq %rdx, (%rcx)
ret
```

연습: 어떤 코드가 생성될까?

```
void uremdiv(unsigned longx, unsigned longy,
unsigned long *qp, unsigned long *rp) {
        unsigned long q = x / y;
        unsigned long r = x % y;
        *qp = q;
        *rp = r;
}
```

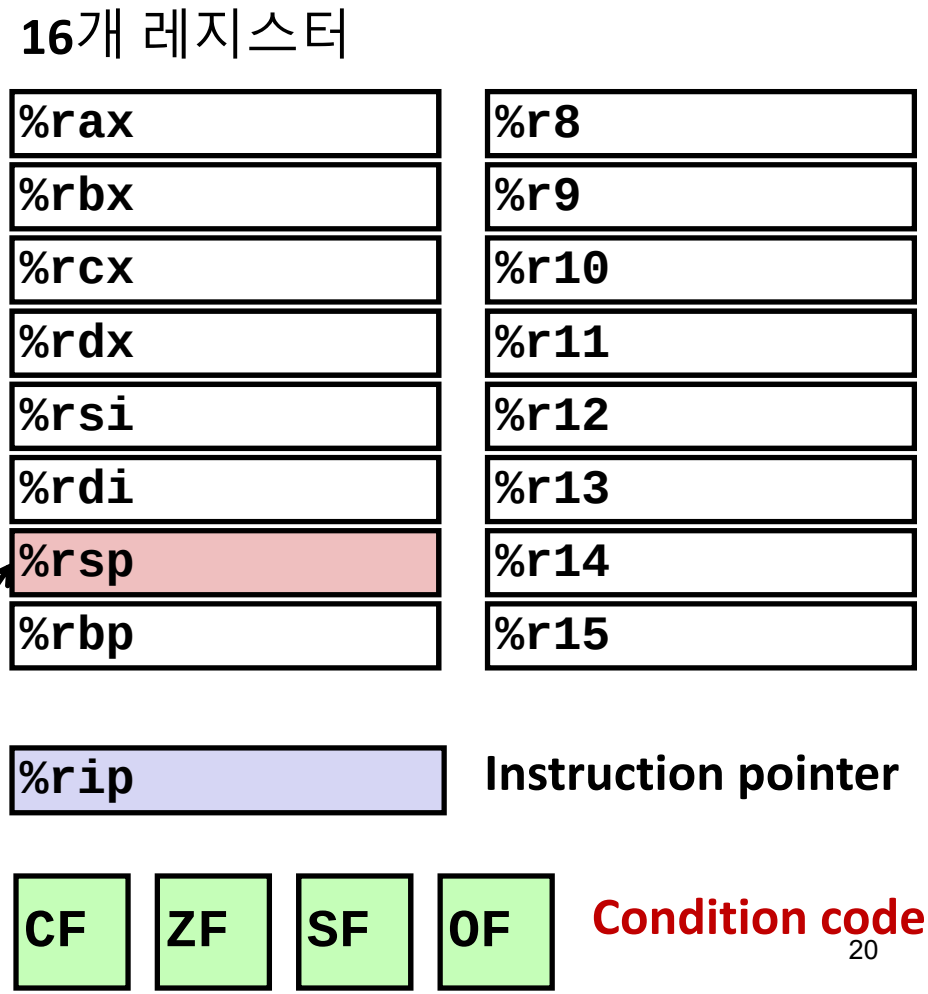
- 앞의 remdiv()를 수정하여 완성함!!



3. 상태플래그/CONDITION CODE와 CMP, TEST 명령

X64 processor state

현재 실행 프로그램관련 정보
- 임시 데이터
  (**%rax**, …)
- 실행시간 스택 위치
  (**%rsp**)
- 현재 코드의 제어포인트(실행) 위치
  (**%rip**, …)
- 최근 테스트 상태(제어 조건)
  (**CF, ZF, SF, OF**)



상태 플래그 (Condition Codes) 자동설정됨

- 1비트 플래그 Single Bit Registers

  CF Carry Flag(비부호연산용) SF Sign Flag(부호연산용)

  ZF Zero Flag OF Overflow Flag(부호연산용)

- 연산명령어의 결과로 자동 세팅됨

  `addq` *Src*,*Dest* ⟷ *t* ← *a* + *b*

  - CF : MSB로부터 carry가 발생한 경우에 1로 세팅됨(unsigned용)
  - ZF : t==0인 경우 1로
  - SF : t < 0인 경우 1로
  - OF : 2의 보수 오버플로우(signed에서 의미)발생시 1로

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

- `leal` 명령은 flag에 영향주지 않음.(주소명령이라)
- Signed와 unsigned를 위해 모두 설정된다.

상태 플래그 세팅 명령어 - CMP

- Compare 명령의 flag자동설정

  `cmpq` *Src2*, *Src1* # cmpq b, a는 dest없는 a-b 효과

  - CF : MSB에서 carry발생시 1로 (unsigned)
  - ZF : a==b일때 1로 세팅됨
  - SF : a < b일때 1로 세팅됨
  - OF : 2's complement (signed) overflow 발생시 설정
    - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

상태 플래그 세팅 명령어 - TEST

- Test 명령어를 이용한 플래그 세팅
  `testq` *Src2,Src1 : a&b를 dest없이 하는 효과*

  - Src1 & Src2로 플래그설정, 하나는 대개 mask
  - 단순 and 연산을 수행하지만, dest없음.

  - ZF set when a&b `==` `0`
  - SF set when a&b `<` `0`

4. SETX 명령

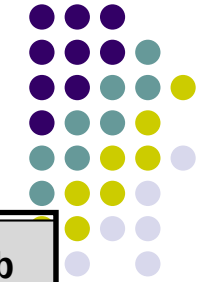
상태 플래그의 이용명령 - SET

- 명령어 : SetX [byte Register/Memory] (최하단 바이트 R or M)
  - 상태 플래그들의 조합의 결과를 한 바이트 오퍼랜드에 저장

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)

x86-64 Integer Registers



%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

- Can reference low-order byte

Setl byte : (SF ^ OF) ?

- Less than (Src < Dest ➔ SF^OF=1)
  1. Src > 0, Dest > 0 (4 – 7 = -3) : SF=1
  2. Src < 0, Dest < 0 (-4 - -3 = -1) : SF=1
  3. Src > 0, Dest < 0 : 불가능한 경우 (Src가 큼)
  4. Src < 0, Dest > 0 : (두가지로 구분)
     1. |Src-Dest| <= |Tmin| : (-4 -3=-7, -4 -4 = -8) SF=1
     2. |Src-Dest| > |Tmin| (-4 -5 = -9) ➔ 표현 불가(OF)
        1. 1100
        2. 1011
        3. 1 0111 (SF = 0, OF=1)

Set명령어 사용하기

Int comp(data_t a, data_t b)
a in %rdi, b in %rsi

comp:

```
        cmpq    %rsi, %rdi      # a, b 비교
        setl    %al             # a < b이면, al을 1로
        movzbl  %al, %eax       #%eax, %rax 나머지부분을 0으로
        ret
```

상태 플래그 이용 명령 – SET (1)

- SetX 명령
  - 상태 플래그를 한 바이트에 저장
  - 지정 가능한 바이트 사용
    - `movzbl` 로 마침

```
int gt (long x, long y)
{
  return x > y;
}
// x in %rdi, y in %rsi,
// return value in %rax/eax
```

Body

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

cmp명령은 비교순서에 유의!

상태플래그/Condition Code(X)를 사용하는 세가지 명령

1. 바이트 R/M를 0/1로 (CC의 조합에 의해)
   - setX
2. 프로그램 상의 특정 위치로 조건 점프
   -jX
3. CC의 조합/조건에 따라 데이터 이동
   - cmovX

5. JMP, JX 명령

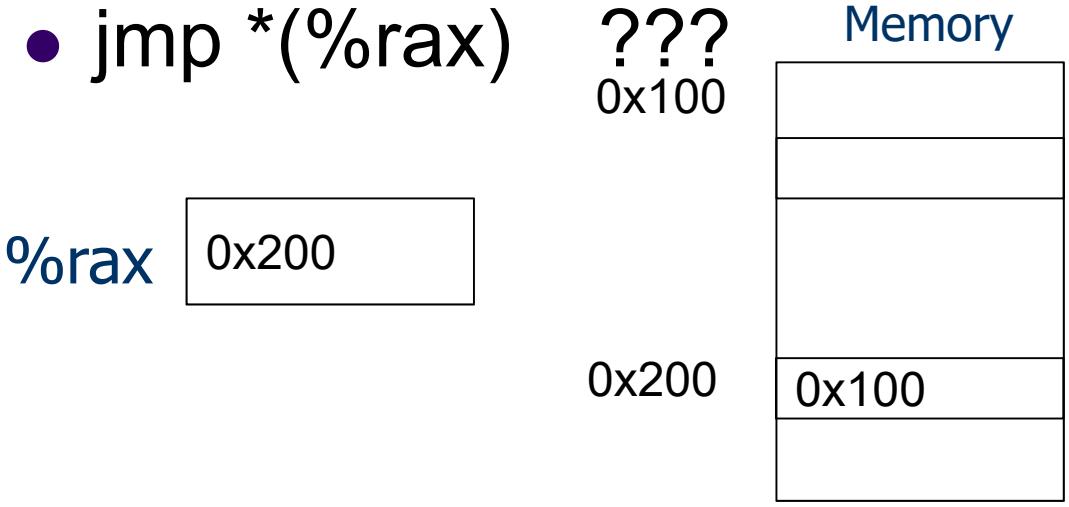
Jump 명령어 (2)

- jX Label
  - 상태 플래그에 따라 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 *%rax %rax에 저장된 주소로 가라
- jmp *(%rax) ???



조건형 분기 예(Old Style)

- 생성방법:
- **$ 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

gcc –O options

```
-O2 Optimize even more.  GCC performs nearly all supported
    optimizations that do not involve a space-speed tradeoff.  As
    compared to -O, this option increases both compilation time and the
    performance of the generated code.

    -O2 turns on all optimization flags specified by -O.  It also turns
    on the following optimization flags: -fthread-jumps
    -falign-functions  -falign-jumps -falign-loops  -falign-labels
    -fcaller-saves -fcrossjumping -fcse-follow-jumps  -fcse-skip-blocks
    -fdelete-null-pointer-checks -fdevirtualize
    -fdevirtualize-speculatively -fexpensive-optimizations -fgcse
    -fgcse-lm -fhoist-adjacent-loads -finline-small-functions
    -findirect-inlining -fipa-cp -fipa-cp-alignment -fipa-sra -fipa-icf
    -fisolate-erroneous-paths-dereference -flra-remat
    -foptimize-sibling-calls -foptimize-strlen -fpartial-inlining
    -fpeephole2 -freorder-blocks -freorder-blocks-and-partition
    -freorder-functions -frerun-cse-after-loop -fsched-interblock
    -fsched-spec -fschedule-insns  -fschedule-insns2 -fstrict-aliasing
    -fstrict-overflow -ftree-builtin-call-dce -ftree-switch-conversion
    -ftree-tail-merge -ftree-pre -ftree-vrp -fipa-ra

    Please note the warning under -fgcse about invoking -O2 on programs
    that use computed gotos.

    NOTE: In Ubuntu 8.10 and later versions, -D_FORTIFY_SOURCE=2 is set
    by default, and is activated when -O is set to 2 or higher.  This
    enables additional compile-time and run-time checks for several
    libc functions.  To disable, specify either -U_FORTIFY_SOURCE or
    -D_FORTIFY_SOURCE=0.

-O3 Optimize yet more.  -O3 turns on all optimizations specified by -O2
    and also turns on the -finline-functions, -funswitch-loops,
    -fpredictive-commoning, -fgcse-after-reload, -ftree-loop-vectorize,
    -ftree-loop-distribute-patterns, -ftree-slp-vectorize,
    -fvect-cost-model, -ftree-partial-pre and -fipa-cp-clone options.
```

gcc –O options

-O0 Reduce compilation time and make debugging produce the expected results. This is the default.

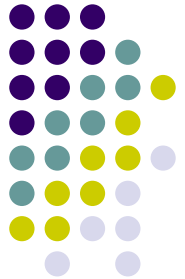
-Os Optimize for size. **-Os** enables all **-O2** optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

-Os disables the following optimization flags: **-falign-functions -falign-jumps -falign-loops -falign-labels -freorder-blocks -freorder-blocks-and-partition -fprefetch-loop-arrays**

-Ofast
Disregard strict standards compliance. **-Ofast** enables all **-O3** optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on **-ffast-math** and the Fortran-specific **-fno-protect-parens** and **-fstack-arrays**.

-Og Optimize debugging experience. **-Og** enables optimizations that do not interfere with debugging. It should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

조건형 분기문의 예



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

```
_max:
    cmpl %esi, $edi
    jle .L3
    movl %edi,%eax
L3:
    movl %esi,%eax
    ret
```

eax 의 사용에 주목!
return value!!

x in %edi, y in %esi

점검문제 1. 분기문

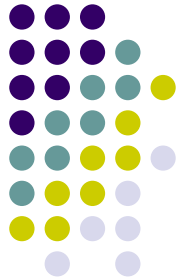
```
    movl   $0x2, %eax          (1)
    movl   $0x4, %edx          (2)
    shl    $0x1, %eax          (3)
    cmpl   %eax, %edx          (4)
    jne    .L2                 (5)
.L1: nop                       (6)
.L2: movl %edx, %eax           (7)
```

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?



6. 조건적인 데이터 이동 명령어 (CMOV)

조건에 따른 이동 명령(CMOV)

- 조건적인 이동 명령
  - 명령이 지원하는 방식:

    if (Test) Dest ← Src
  - 1995후반 x86 에서 지원
  - GCC도 cmov 명령 사용
- 왜 이런 명령을?
  - 분기 명령은 명령어 파이프라인에서 매우 바람직하지 않음.
  - cmov는 제어 이동을 하지 않음(분기명령처럼)

C 코드 표현 예

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

풀어헤진 형태(cmov 용)

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

조건에 따른 이동 명령 사례

```
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
```

조건 이동 명령 적용에 주의할 경우

계산이 복잡한 경우(then, else 부분 모두)

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- 두 값을 모두 계산해두어야 함
- 계산이 매우 단순한 경우에 의미있음

미리 해두기에는 위험한 계산

```
val = p ? *p : 0;
```

- 두 값이 모두계산되어야 하는데
- p==0인 경우에, *p를 미리 계산하는 건 위험함!!

부작용(side-effect)이 있을 수 있는 계산

```
val = x > 0 ? x*=7 : x+=3;
```

- 두 값이 모두 계산되어야 하는데…
- x에 대한 값의 변화까지 가져오는 계산이라서 . . .

do-while, while, for 문

7. 순환문/반복문 번역

어셈블리어에서의 루프구현

- 루프, do-while, while, for문을 어셈에서는?

- 어셈블리어에는 편리한 루프명령이 없음
- 어셈블리어에서 루프는 조건비교와 점프로 구현
- 대부분 컴파일러에서는 루프문들을 do-while 형태로 구현

“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(어셈과 가까움)

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

- 인자 x에 있는 1의 수를 세는 기능
- 조건 분기를 사용해서, 루프 계속 또는 중단 구현

“Do-While” 루프 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 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

일반적인 "While"문의 번역

C Code

```
while (Test)
  Body
```

1. Goto Version

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

Do-While Version

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

2. 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:
```

8. SWITCH 문 번역

```
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

- 다양한 case들
- 다중 case 레이블
  - case: 5 & 6
- Fall through 사례
  - case: 2
- 빠진 case
  - case: 4

점프테이블(Jump Table) 구조

Switch Form

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

점프 테이블

```
jtab: Targ0
      Targ1
      Targ2
      •
      •
      •
      Targn-1
```

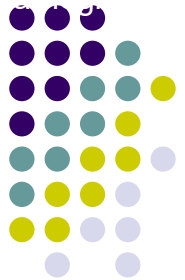
점프 타켓

```
Targ0:   코드 블록 0
Targ1:   코드 블록 1
Targ2:   코드 블록 2
         •
         •
         •
Targn-1: 코드 블록 n–1
```

번역 (확장된 형태의 C 표현)

```
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)
```

값의 범위 고려!!

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

w 초기화 안됨

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]
```

Indirect jump

어셈블러 설정 (switch문처리)

- 테이블 구조
  - 각 타겟은 8 바이트 차지
  - 베이스 주소는 **.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
```

- 점프
  - 직접 점프는: **jmp .L8**
  - 점프 타켓은 레이블 **.L8** 표시

  - 간접 점프: **jmp *.L4(,%rdi,8)**
  - 점프 테이블의 시작주소: **.L4**
  - scale factor는 8로 해야 함 (주소가 8바이트 자치)
  - 타켓 주소는: **.L4 + x*8 (0**과 **6**사이의 **x**에 대해서**)**

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;
    }
```

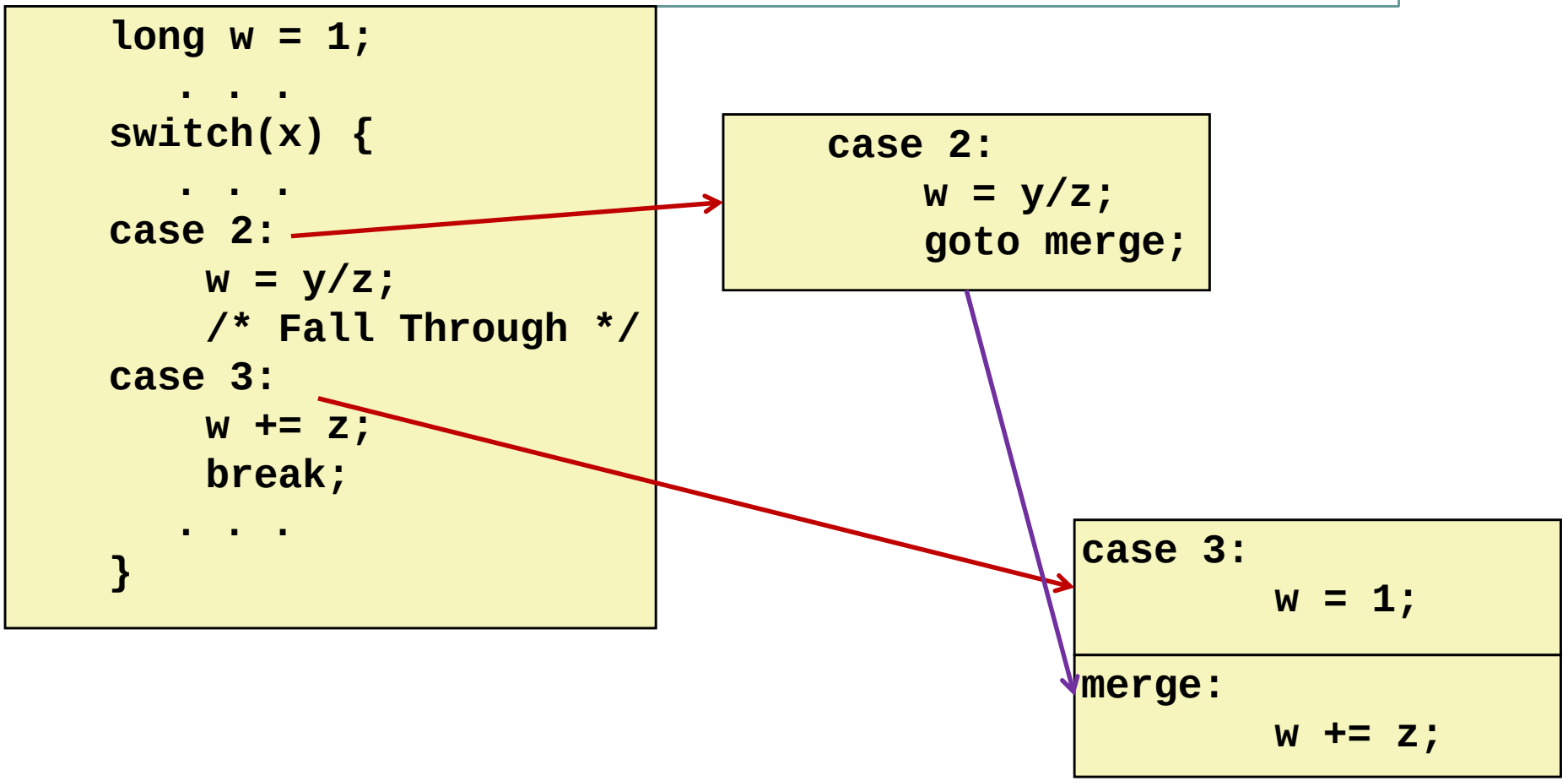
코드 블록 (x == 1)

```
  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

Fall-Through 처리



코드 블록 (x == 2, x == 3)

```
    long w = 1;
       . . .
    switch(x) {
       . . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
       . . .
    }
```

```
.L5:                        # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx             #  y/z
   jmp     .L6              #  goto merge
.L9:                        # Case 3
   movl    $1, %eax         #  w = 1
.L6:                        # merge:
   addq    %rcx, %rax       #  w += z
   ret
```

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

코드 블록 (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl  $1, %eax        #  w = 1
  subq  %rdx, %rax      #  w -= z
  ret
.L8:                    # Default:
  movl  $2, %eax        #  2
  ret
```

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

요약

- C 언어의 제어문
  - if-then-else
  - do-while
  - while
  - switch
- 어셈블리어의 제어문
  - 무조건 점프문(JMP)
  - 간접 점프(JMP *L(rb,ri,S))
  - 조건 점프문(Jx)
  - 조건 데이터 이동문(CMOVx)
- 컴파일러
  - 보다 복잡한 어셈블리 코드를 생성해 낸다
- 표준 기법
  - 모든 루프는 do-while로 변환된다
  - Switch문은 점프테이블로 구성될 수 있다
- 주소 계산문: leaq
- 산술: add, sub, mul, div
- 논리: and, or, not, neg,
- 128bit: rdx:rax, divq, mulq
- 비교: cmpq
- 테스트: testq

실습과 다음 주 준비

- 이어지는 실습: 어셈2 – 산술문,제어문(실습)

- 다음주 강의 동영상: 어셈블리어 프로그래밍3 – 프로시저
- 예습 질문은 개인과제로 올림.