Intel Architecture 32bit Assembly Language

KAIST 전산학부 이주안

Three Levels of Languages

프로그래밍 언어에는 3가지 레벨Level이 있다고 말합니다. Python, C언어 등이 포함된 고급언어High-level Programming Language, 어셈블리 언어Assembly Language, 기계 언어Machine Language입니다. 이 레벨을 나누는 기준은 명령 구조Instruction Set Architecture, ISA에 대한 추상화Abstraction인데, 여기서는 단순히 "기계어를 인간이 얼마나 잘 알아들을 수 있게 포장했는가" 정도로 이해 하셔도 좋습니다. 0과 1로 이루어진 기계어보다는 단어와 10진법 수로 한번 포장해 두는 것이 알아듣기 쉽고, 단어로 한번 포장하는 것보다는 if, for, while 같은 문법 구조를 추가하는 것이, 그것보다는 Python과 같은 직관적이고 강력한 문법 구조를 가진 것이 더 High-level 입니다.

Assembly Language

위 세가지 단계로 볼 때 어셈블리 언어는 0과 1보다는 인간에 가깝고, C보다는 인간에 먼 언어라고 직관적으로 이해할 수 있습니다. 어셈블리 언어는 단순히, 기계어 명령어 집합, 가령 1과 2를 더해라, 메모리Memory에 2를 저장해라, 등의 명령어를 1대 1로 대응시켜 둔 언어에 지나지 않습니다. 맛보기로 먼저 보자면,

```
movl $1, %eax
addl $2, %eax
```

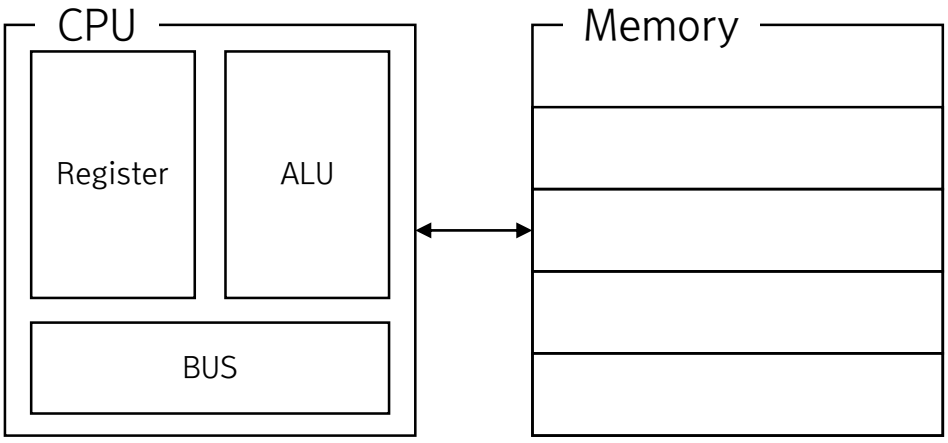
1과 2를 더하는 코드는 위와 같이 작성할 수 있습니다. 내용 자체는 아직 몰라도 괜찮습니다. 단순히 + 기호를 써서 적을 수 있는 고급 언어에 대해서는 사뭇 복잡해진 걸 알 수 있습니다.
C, Python 등의 언어가 있음에도 이렇게 복잡하고 어려운 언어를 쓰는 이유는 기계를 직접 다루기에 효율적이고, 빠른 코드 작성이 가능하고 컴퓨터에 대한 보다 직관적인 이해가 가능하기 때문입니다. 공부하면서 알아봅시다.

Intel Architecture 32bit (IA32)

어셈블리 언어에도 많은 종류가 있습니다. 어셈블리 언어는 기계에 매우 영향을 많이 받는 언어이기 때문에 CPU마다, Memory 마다 그 종류가 달라질 수 밖에 없습니다. 하지만 세상에는 제일 많이 쓰는 체계나, 상품이 있기 마련이고, 우리는 Intel이 만든 32bit 어셈블리 체계를 활용한 컴퓨터를 주로 사용해왔습니다. 가장 많이 사용되고 있고, 가장 많이 검증을 거친 언어가 되는 것입니다. 따라서 수업에서는 물론, 본 자료에서도 IA32 어셈블리를 사용하겠습니다.

Computer Architecture

어셈블리는 기계에 정말 가까운 언어이고, 기계를 직접적으로 다루는 언어이기 때문에 컴퓨터 구조에 대한 이해가 꼭 필요합니다. 따라서 본격적인 프로그래밍에 들어가기 전에 컴퓨터 구조 **Computer Architecture/Organization**에 대해 간단히 다뤄보겠습니다.



위 그림은 정말 간단한 컴퓨터의 구조를 보여주는 그림입니다. CPU**Central Processing Unit**은 간단히 컴퓨터의 모든 연산 처리를 관장하는 부분입니다. 연산 처리는 생각보다 범주가 넓어서, 연산의 입력값**input**과 출력값**output**을 처리하는 부분, 연산 과정의 임시적인 값들을 저장하는 부분, 실제 수리 연산과 논리 연산을 수행하는 부분 등으로 나뉘게 됩니다. 컴퓨터에서 다루는 메모리**Memory**는 전산학적으로 굉장히 범주가 넓은 단어입니다. 구글 드라이브와 같은 것도 메모리이고, "컴퓨터 용량이 얼마야" 할 때도 메모리이고, 프로그래밍에서 변수를 선언하는 것도 메모리입니다. 앞으로 주로 이야기할 메모리**Memory**는 앞으로 보통 앞의 수많은 메모리 중 DRAM**Dynamic Random-Access Memory**를 지칭하는 용어로 쓰일 것인데, 여기서는 간단히, 각각의 프로그램이 사용하는 추상적인 메모리 공간으로 생각하면 충분할 것 같습니다. 전체 메모리의 일부고, 각 프로그램이 자유롭게 사용할 수 있도록 할당된 영역이라는 것이죠. 하나하나 자세히 이야기해 보겠습니다.

Register

레지스터**Register**는 컴퓨터에 존재하는 메모리 영역 중 가장 작지만, 가장 빠른 메모리입니다. 위의 그림을 보면 CPU 안에 있는 요소 중 하나인데, CPU에서 연산을 할 때 연산에 필요한 각 과정에서의 메모리로 활용을 하고, 보다 장기적인 기억을 위해서는 메모리**Memory** 영역에 저장을 합니다. 이러한 레지스터는 개수와 크기가 이미 결정되어 있는데, IA32의 레지스터는 아래와 같습니다.

%eax %ebx %ecx %edx %esi %edi %esp %ebp

IA32에는 위와 같이 8개의 프로그래머가 사용 가능한 레지스터가 있으며, 각각의 레지스터는 32bit의 크기입니다. 또, 어셈블리에서 %로 표기된 것은 레지스터이기도 합니다. 어셈블리에서는 추가적인 변수를 선언한다는 개념이 다른 고급 언어와는 다르기 때문에 앞으로 위의 레지스터들을 조합해서 프로그래밍을 하게 됩니다.

비록 레지스터의 개수는 8개지만, 여러 관례와 안정성의 문제로 레지스터가 주로 사용되는 목적은 정해져 있는데요, 아래의 4개가 현재로써는 범용적으로 사용되고 있습니다.

%eax %ebx %ecx %edx

즉, 단순한 산술 연산을 위해 사용할 수 있는 레지스터는 8개가 아니라 4개라는 것이죠. 아래 4가지는 주소 값을 저장하기 위해 사용됩니다. C언어와 굳이 비교하자면 포인터 변수와 유사한 역할을 하게 됩니다.

%esi %edi %esp %ebp

하지만 위 4개의 레지스터는 주로 다른 값을 이미 저장하고 있기 때문에 특수한 목적이 아니면 사용하지 않는 것이 좋습니다. 앞서, "포인터 변수와 유사한 역할"이라고 말했지만, 어셈블리의 레지스터는 그 사용법이 특정되어 있지 않은 32bit 메모리 공간일 뿐입니다. 또한 메모리 주소 역시 하나의 정수 값에 지나지 않죠. 따라서 어셈블리에서는 %eax 같은 레지스터가 값은 물론 주소까지 저장할 수 있고, 실제로도 그렇게 많이 사용합니다.
사실, 프로그래머가 사용할 순 없지만 존재하는 레지스터가 하나 더 있습니다. %eip 인데요, EIP 레지스터는 다음으로 실행될 프로그램의 위치를 저장합니다. EIP에 들어가는 값은 연속적으로 증가할 수도 있고, 또는 다른 줄로 한번에 Jump 할 수도 있습니다. 이 과정이 C에서 if나 for, while 등으로 나타나게 됩니다. 레지스터를 실제로 이용하는 법은 조금 있다 다뤄보도록 하겠습니다.

ALU Arithmetic and Logic Unit

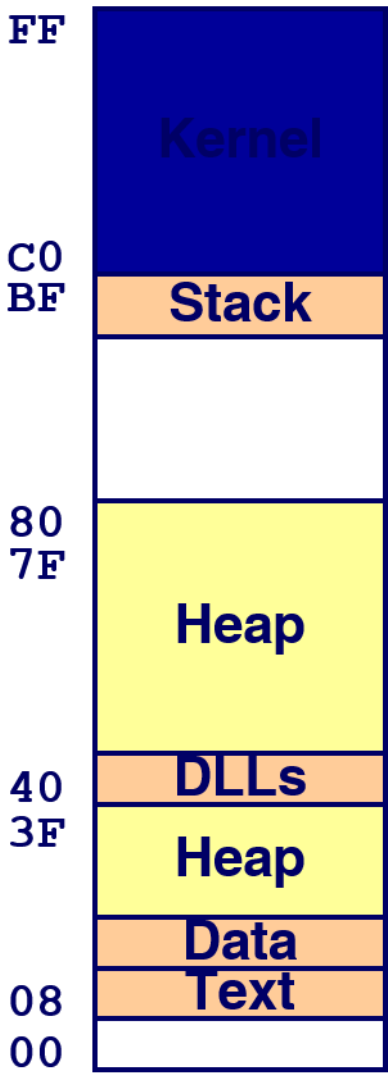
산술 논리 연산 장치ALU는 CPU내의 실질적인 연산을 진행합니다. 우리가 더하기, 빼기, &, | 과 같은 연산을 할 때 CPU 내부의 ALU가 위의 레지스터들을 활용해서 실질적인 연산을 진행하게 됩니다. 이러한 연산은, C언어와 사뭇 다른 부분이 있는데요. C언어에서는 직접 알려주지 않는 Overflow 등의 연산을 플래그flag를 통해 참조할 수 있게 해두었습니다. 간단히, overflow가 일어나면 OF Overflow Flag라는 이름의 플래그를 켜서 프로그래머가 이를 확인하고 대응할 수 있습니다. 이러한 플래그는 ZF Zero Flag, OF Overflow Flag, CF Carry Flag, SF Sign Flag 등이 있고, 이에 대한 사용법은 조금 있다 다뤄보겠습니다.

BUS

CPU와 메모리, 입력 및 출력 연산을 담당하는 요소들은 수없이 많습니다. 당장에 지금까지 언급된 것만 9개의 레지스터와 1개의 ALU, 각종 플래그들이 있죠. 수학적으로 이러한 요소들이 모두 상호작용할 수 있게 하려면 $_NC_2$개의 연결 통로가 필요합니다. 하지만 이 방식은 너무나도 높은 비용이 소요됩니다. 따라서 실제 컴퓨터에서는 길고 넓은 연결 통로를 하나 만들고 모든 요소들을 이 통로에 연결해 두었습니다. 더욱 긴 연결 통로가 필요하고, 누가 언제 이 통로를 사용하느냐에 대한 복잡한 규칙이 필요하겠지만 비용은 더욱 적게 들겠죠. 이 통로가 BUS입니다.

Memory

메모리Memory는 굉장히 중요한 부분입니다. 실제로 값을 저장하는 부분이 이곳이고, 메모리를 사용하는 프로그램 조차도 컴퓨터에서는 하나의 데이터로써 메모리 위에 저장됩니다. 따라서 데이터Data와 데이터가 저장된 주소Memory Address를 잘 사용해주는 과정은 매우 중요합니다. 여러분이 어렵지 않게 만나 보셨을 Segmentation Fault가 대부분 여기서 발생하는 에러죠.



왼쪽의 이미지가 실제 메모리 구조입니다. 동적 할당Dynamic Allocation을 제대로 공부했다면 익숙해야 할 힙 영역Heap Memory가 보이고, 자료 구조에서 봤던 스택Stack이라는 이름도 보입니다.

커널Kernel은 운영체제가 다루는 메모리 영역입니다. 프로그래머가 다룰 수도, 행여나 다뤄서도 안되는 메모리 영역입니다.

스택Stack은 앞으로 가장 많이 사용하게 될 영역입니다. 실제로 IA32 에서 레지스터 외에 저장되는 값들을 저장하거나, 서브루틴Subroutine이나 프로시져Procedure를 만들기 위해 사용하는 영역입니다. 스택은 그 이름처럼 스택 자료구조를 가지고 있으며, 스택의 시작 위치base pointer를 %ebp가, 스택의 제일 위stack pointer를 %esp가 가지고 있습니다. 스택에서 중요하게 봐야할 점은, 다른 메모리 영역과는 다르게 거꾸로 자란다grow down는 사실인데요, 나중에 중요하게 다뤄질 이야기입니다.

힙Heap은 동적 할당 등에 사용되는 영역입니다. C의 malloc, calloc, realloc 등의 함수가 힙 영역에 메모리를 할당합니다.

DLLs는 링커Linker가 외부에서 가져온 코드를 합칠 때 사용하는 메모리 영역입니다. 링커가 무엇인지 모른다면 C언어 첫 부분을 다시 공부하고 올까요...?

데이터Data는 전역 변수Global Variable등이 저장되는 영역입니다.

텍스트Text는 실제 실행되는 코드 등의 저장되는 영역입니다.

설명의 양을 보면 알겠지만, IA32에서는 최소한 스택의 개념은 확실히 알아두고 가시는 것이 좋습니다.

Endianness

아래의 C 코드를 본인 컴퓨터에 실행해 봅시다.

```
#include<stdio.h>
int main(){
	int value = 0x12345678;
	char* pt = (char*)&value;
	for(int i=0; i<4; i++) printf("%d₩n", *pt++);
	return 0;
}
```

보통 0x12, 0x34, 0x56, 0x78의 값이 순서대로 나오길 기대하지만 몇몇 컴퓨터에서는 다르다는 것을 보셨을 거라 생각합니다. 컴퓨터에서 바이트를 배열하는 방법을 바이트 순서Byte Order라고 하는데, 큰 단위가 앞에 나오는 빅 엔디안Big Endian과 작은 단위가 앞에 나오는 리틀 엔디안Little Endian으로 보통 구별됩니다. 예상과 다르게 0x78부터 나오는 바이트 순서를 리틀 엔디안이라고 부르죠.

데이터 저장 및 이동

데이터를 레지스터에 저장하거나 데이터를 옮길 때는 mov 라는 명령어를 사용합니다.
아래는 %eax에 10진법 수 1을 저장하는 코드입니다.

```
movl $1, %eax
```

mov 명령어 뒤에 l이라는 접미사가 붙은 것을 볼 수 있습니다. 소문자 L은 long의 약자로, 4바이트 크기의 메모리를 옮긴다는 것을 의미합니다. 또 $ 접두사는 상수값 1을 저장하겠다는 의미입니다. 즉, 32bit 크기의 레지스터 %eax를 값 1로 꽉 채우겠다는 의미죠.
소문자 L이 4byte를 의미한다는 것은, 다른 크기의 레지스터와 다른 크기의 데이터를 옮기는 명령어도 따로 존재한다는 뜻이겠죠. Long의 약자인 소문자 L이 4바이트, Word의 약자인 소문자 W가 2바이트, Byte의 약자인 소문자 B가 1바이트입니다. 아래는 각 레지스터의 위치를 접근하는 레지스터 명령어입니다.

31 16	15 8	7 0	16bit	32bit
	%ah	%al	%ax	%eax
	%bh	%bl	%bx	%ebx
	%ch	%cl	%cx	%ecx
	%dh	%dl	%dx	%edx
	%si			%esi
	%di			%edi

가령 아래 코드는, 1바이트짜리 크기의 %ab를 %bl 위치로 옮기는 코드입니다.

```
movb %ab, %bl
```

Before

	1	2	%eax
	3	4	%ebx

After

	1	2	%eax
	3	1	%ebx

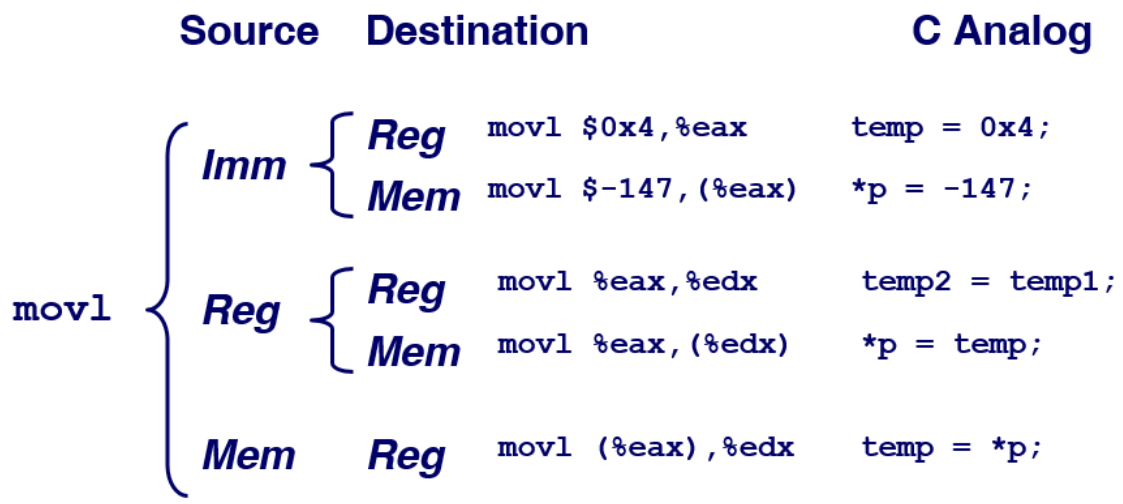
주소 지정 방식Addressing Mode

앞서 언급 되었듯 레지스터는 값 뿐만 아니라 주소 값 또한 저장할 수 있습니다. mov는 각 레지스터에 저장된 값이 값일 때와 주소일 때에 따라 실제 값에 접근할 수 있는 세 가지 방법을 제시합니다. 각각은 레지스터에 담긴 주소 값을 따라가는 직접 주소 지정방식Direct Addressing Mode, 레지스터에 담긴 주소가 실제 값이 저장된 주소를 가리키고 있는 간접 주소 지정방식Indirect Addressing Mode, 상수 값을 해당 레지스터에 직접 넣는 즉시 주소 지정방식Immediate Addressing Mode이 있습니다.
주소 지정 방식은 어셈블리에서 괄호로 둘러싸여 표현되는데요, 간단히 아래의 두 코드에서 차이를 볼 수 있습니다.

```
movl $1, %ebx
movl (%eax), %ebx
```

가령 위는 즉시 주소 지정방식으로 값을 넣은 경우이고, 아래는 간접 주소 지정방식으로 %eax가 가지고 있는 주소 값이 가리키는 값을 %ebx에 넣은 경우입니다.

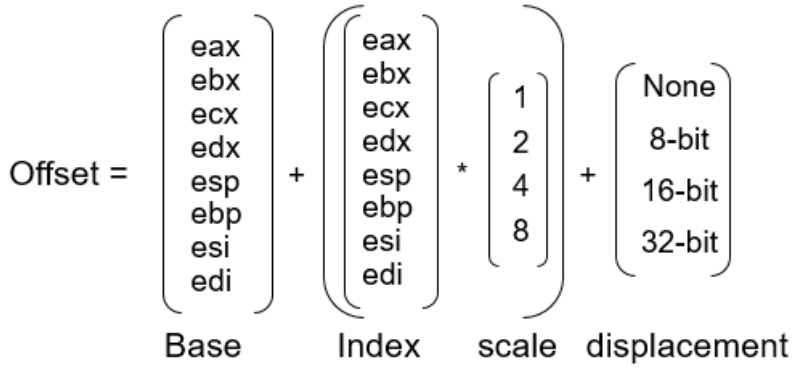
Mov 연산 조합



이 사진은 지정 방식에 따른 movl 의 조합법입니다. 위에서 업급된 지정 방식을 조합할 수 있지만 메모리-메모리 mov 연산은 되지 않는 다는 것에 주의해야 합니다.

주소 지정 방식 활용

위의 주소 지정 방식을 C의 포인터 연산처럼 활용할 수 있는 방법이 있습니다.



- Displacement — `movl foo, %ebx`
- Base — `movl (%eax), %ebx`
- Base + displacement — `movl foo(%eax), %ebx` / `movl 1(%eax), %ebx`
- (Index * scale) + displacement — `movl (,%eax,4), %ebx`
- Base + (index * scale) + displacement — `movl foo(%edx,%eax,4),%ebx`

위 사진만으로는 설명이 부족한 것 같으니 예제와 함께 알아보겠습니다.

Example – Offset

movl 2000, %ebx	주소가 2000인 위치의 값을 ebx에 넣음
movl (%eax), %ebx	%eax의 값이 주소인 위치의 값을 ebx에 넣음
movl 2000(%eax), %ebx	2000 + %eax 위치의 값을 ebx에 넣음
movl (,%eax,4), %ebx	%eax * 4 위치의 값을 ebx에 넣음
movl 2000(%edx, %eax, 4), %ebx	2000 + edx + eax * 4 위치의 값을 ebx에 넣음

위의 Offset을 만드는 방법은 array의 요소들을 참조할 때 많이 사용하는데요, 가령 아래와 같은 C언어 코드가 있다고 해봅시다.

```
n = arr[i]; // int arr[10];
```

위와 같은 C언어 코드를 어셈블리로 옮기기 위해서는 아래와 같이 사용합니다.

```
movl arr(,%ecx,4), %eax
```

%ecx가 i이고, %eax가 n이라면 arr + %ecx * 4의 위치에 arr[i]가 있습니다. C언어에서는 arr + %ecx가 되겠지만 어셈블리에서는 int형이 4byte란 것도 감안을 해 주어야 합니다. 따라서 위의 코드가 C언어의 코드와 같아지게 되는 것이죠.

각종 산술 논리 연산

addl Src, Dest	Dest = Dest + Src
subl Src, Dest	Dest = Dest - Src
imull Src, Dest	Dest = Dest * Src (For signed)
imul Dest	%edx = %eax * Dest의 상위 32bit (For Signed) %eax = %eax * Dest의 하위 32bit
mull Dest	%edx = %eax * Dest의 상위 32bit (For Unsigned) %eax = %eax * Dest의 하위 32bit
div Dest	%eax = %edx:%eax 를 Dest로 나눈 몫 (For Unsigned) %edx = %edx:%eax 를 Dest로 나눈 나머지 * %edx:%eax는 %edx와 %eax를 연결한 64bit 크기 메모리
idiv Dest	%eax = %edx:%eax 를 Dest로 나눈 몫 (For Signed) %edx = %edx:%eax 를 Dest로 나눈 나머지
sall Src, Dest	Dest = Dest << Src
sarl Src, Dest	Dest = Dest >> Src (Arithmetic)
shll Src, Dest	Dest = Dest << Src
shrl Src, Dest	Dest = Dest >> Src (Logical)
xorl Src, Dest	Dest = Dest ^ Src
andl Src, Dest	Dest = Dest & Src
orl Src, Dest	Dest = Dest | Src
incl Dest	Dest = Dest + 1
decl Dest	Dest = Dest – 1
negl Dest	Dest = - Dest
notl Dest	Dest = ~ Dest

어셈블리도 C언어처럼 다양한 연산자들을 지원하기 때문에 산술 연산이 어렵지 않습니다. 하지만 많은 경우 더하기와 shift연산으로 곱하기와 나누기를 구현하는 것이 효율적입니다. 아래에 몇가지 예시를 통해 곱하기와 나누기를 shift와 더하기로 구현해보겠습니다.

Example – Shift와 Add로 Mul 구현

n = a*3 + 1;
// n: %edx, a: %eax
	movl %eax, %edx
	addl %eax, %edx
	addl %eax, %edx
	incl %edx

n = a*17;	movl %eax, %edx
	shll $4, %edx
	addl %eax, %edx

n = a*20;	movl %eax, %edx
	shll $2, %edx
	addl %eax, %edx
	shll $2, %edx

n = a/8;	movl %eax, %edx sarl $3, %edx

Load Effective Address

mov는 주소 방식을 활용하기 때문에 레지스터에 저장된 값을 산술 연산해 주소 자체를 저장하는 방법이 없습니다. 이를 해결하기 위해 주소를 직접 다루는 연산이 lea입니다.

movl 4(%eax, %ecx, 4), %ebx
leal 4(%eax, %ecx, 4), %ebx

가령 위의 두 가지 예시를 비교해보면, 위의 코드는 4+%eax+%ecx*4 에 해당하는 주소가 가리키는 값을 %ebx에 넣고, 아래의 코드는 4+%eax+%ecx*4 자체를 %ebx에 넣습니다. Leal은 포인터 연산을 직접적으로 하는 방법이라고 생각하면 편합니다.

조건문Conditional Statement

C의 if문에 해당하는 조건문Conditional Statement과 같은 문법은 어셈블리에서는 직접적으로 표현되지 않습니다. 즉, if와 같은 키워드가 있는 것이 아닌, 위에서 설명한 %eip에 저장된 값을 수정하며 해당 위치로 직접 이동을 하는 형태이죠.

```
if(x < 3){
    x -= 3;
} else {
    x += 3;
}
```

즉, 위와 같은 코드에서는 x > 3을 확인한 후, 조건을 만족하면 %eip에 2를, 아니면 %eip에 4를 넣음으로써 Jump를 하는 형태입니다. 위 코드를 어셈블리로 바꿔보면,

```
        cmpl %eax, 3
        jle else
        subl $3, %eax
        jmp endif
else:
        addl $3, %eax
endif:
```

cmpl은 %eax와 3을 비교합니다. cmpl Src, Dest의 형태이니 Dest – Src, 즉 3 - %eax를 0과 비교합니다. cmpl은 연산의 결과를 어딘가에 저장하는 것이 아닌 위에서 언급된 플래그flag의 상태를 바꿉니다. 그리고 그 플래그들의 상태를 j로 시작하는 점프 명령어들이 참조하여 연산을 진행합니다. 점프에 대한 명령어들은 아래와 같습니다.

점프 명령어Jump Commands

jX	설명
jmp	Jump
je	Jump if equal to 0
jne	Jump if not equal to 0
js	Jump if sign flag is on (Jump if negative)
jns	Jump if sign flag is not on (Jump if not negative)
jg	Jump if greater than 0 (For signed)
jge	Jump if greater than or equal to 0 (For signed)
jl	Jump if less than 0 (For signed)
jle	Jump if less than or equal to 0 (For signed)
ja	Jump if above 0 (For unsigned)
jb	Jump if below 0 (For unsigned)

위의 코드를 다시 해석해보면, 3 - %eax가 0보다 작거나 같은경우(jle), else 부분으로 이동합니다. 어셈블리에서는 이동할 위치를 키워드와 콜론(:)을 통해 표현할 수 있습니다. 3 - %eax가 0보다 작거나 같은 경우에 else로 이동하니, 0보다 큰 경우, 즉 %eax가 3보다 작은 경우에 if문 안으로 들어가게 되는 것이죠.
그리고 %eax에 대한 뺄셈 연산을 수행한 후 endif 로 이동합니다. Else로 이동했다면 덧셈 연산을 수행한 후 endif 로 자연스럽게 이동하게 되죠. 따라서 어셈블리 코드가 C언어 코드를 표현하게 됩니다.

Example – if statement

```
if(x > y){
    x -= y;
} else {
    y -= x;
}
// x: %eax, y: %ebx
```

```
    cmpl %eax, %ebx
    jge else
    subl %ebx, %eax
    jmp endif
else:
    subl %eax, %ebx
endif:
```

루프문Loop Statement

루프문도 결국 if와 같은 방식으로 진행됩니다. 대신 if에서 jump와 같은 분기를 한번만 하는 것이 아니고, 루프의 시작 위치에도 주소를 적어서 돌아갈 수 있게끔 만드는 것이죠. Do-while, while, for에 대한 예시를 각각 알아봅시다. 보통의 경우, 어렵다면 for -> while -> do-while -> goto 문으로 C언어를 바꾼 후 어셈블리로 변환하는 과정이 어려움을 줄이는 과정입니다.

Example – loops

do{
  n /= 2;
} while(n%2 == 0);
// n: %eax
	loop:
	  sarl $1, %eax
	  andl $1, %eax
	  je endloop
	  jmp loop
	endloop:

while(n){
  x += n--;
}
// x: %eax, n: %ecx
	loop:
	  cmpl $0, %ecx
	  je endloop
	  addl %ecx, %eax
	  decl %ecx
	  jmp loop
	endloop:

for(int i=0; i<n; i++){
  x -= n;
}
// x: %eax, n: %ebx, i: %ecx
	xorl %ecx, %ecx
	loop:
	  cmpl %ecx, %ebx
	  jle endloop
	  subl %ebx, %eax
	  incl %ecx
	  jmp loop
	endloop:

스택 Stack

지금껏 다뤘지만 어셈블리에서는 추가적인 변수 선언이 안되는 채로 4 개의 범용 레지스터만을 조합하여 여러 연산을 시행합니다. 하지만 이게 끝이라면 변수를 4개 이상으로 선언할 수도 없고, 메모리를 장기적으로 저장하지도 못하는 문제가 발생합니다. 이를 위해 앞서 언급된 스택 Stack 이라는 메모리 영역을 사용합니다.
지금껏 사용하지 않았던 %esp 와 %ebp 는 사실 스택의 가장 위쪽과 가장 아래쪽의 위치를 저장하고 있는 레지스터입니다. %esp 는 스택 포인터 Stack Pointer, %ebp 는 베이스 포인터 Base Pointer 의 위치를 가지고 있습니다. 가령, 스택의 가장 위 값을 얻어오고 싶다면 아래처럼 합니다.

```
movl (%esp), %eax
```

%esp 가 값을 가지고 있는 것이 아닌 주소를 가지고 있는 것에 유의하여야 합니다. 따라서 괄호를 이용해 %esp 가 가지고 있는 주소가 가리키는 값을 가져왔습니다.
그렇다면 이 스택에 값을 넣으려면 어떻게 해야할까요? 마찬가지로 코드와 함께 보겠습니다.

```
subl $4, %esp
movl %eax, (%esp)
```

마찬가지로 %esp 가 주소를 가지고 있는 것에 유의하여 코드를 보았습니다. 하지만 어딘가 이상한 것이 보입니다. %esp 가 제일 위 값이라면 4 byte 에 해당하는 크기를 더해줘야 하겠지만, 위 코드에서는 빼주고 있습니다. 이는 앞에서 언급되었던 스택의 거꾸로 자라는 grow down 성질 때문입니다. 스택은 Top 의 주소가 가장 작은 주소로 유지됩니다.
같은 방식으로 값을 빼는 방법은 아래처럼 구현할 수 있습니다.

```
movl (%esp), %eax
addl $4, %esp
```

IA32 에서는 스택을 위한 효율적인 명령어를 이미 구현해 두었습니다. 자료구조를 배웠다면 위의 두 연산을 Push 와 Pop 이라고 부른다는 것을 아실텐데요, 어셈블리에서도 아래와 같이 사용할 수 있습니다.

pushl %eax	# %eax 값을 stack 에 push
popl %eax	# %eax 에 stack 의 top 값을 pop

함수Functions

어셈블리는 C언어와 같이 {와 }로 구별되는 변수의 생존 영역Scope에 대한 개념이 없습니다. 코드 전역에서 사용되는 레지스터가 8개가 있고, 그 중 범용 목적으로 사용 가능한 레지스터는 4개가 있습니다. 이러한 특성과 Jump를 통해 코드를 이동하며 루틴routine이 실행되는 방식은 C와 같은 고급 언어에서 구현되는 함수 형태를 만들고자 할 때 5가지 문제점을 야기하는데요, 각 문제점을 알아보고 하나씩 해결해가며 어셈블리에서의 함수를 만들어보겠습니다.

문제 1. 함수 호출과 반환Calling and Returning

시작은 함수 자체를 어떻게 호출할 것인가에 대한 문제로 시작합니다. 별도의 함수 형태가 정해지지 않은 어셈블리에서는 함수 또한 전체 코드 안에 들어있는 루틴일 뿐입니다. 이를 보통 서브루틴Subroutine이라 부릅니다. 우선 간단히, 점프를 한 후, 점프를 한 바로 다음 줄로 돌아오는 방식으로 함수를 구현해봅시다.

```
P:                  # Function P
        jmp R       # Call R
Rtn_point1:
```

```
R:                          # Function R
        ...
        jmp Rtn_point1      # Return
# 아참, 어셈블리에서 주석은 #으로~
```

당장에는 해결된 것으로 보입니다. 하지만 아래의 코드에서 바로 문제가 발생합니다.

```
P:
        jmp R
Rtn_point1:
        jmp R
Rtn_point2:
```

```
R:
        ...
        jmp Rtn_point1
```

위 코드에서 두번째 점프는 Rtn_point2로 돌아오는 것을 기대하지만, Rtn_point1으로 이동합니다. 함수는 전체 코드에서 한번만 실행될 것이라 보장되지 않습니다. 따라서 고정된 위치로 돌아가는 경우 다른 위치에서 함수를 호출했을 때 함수 호출이 끝난 후 돌아오는 것이 보장되지 않습니다. 다른 해결법을 제시해봅시다. 만약에 레지스터에 돌아갈 곳을 저장할 수 있고, 레지스터에 저장된 위치로 점프를 할 방법이 있다면요? 아래와 같은 상황입니다.

```
P:
    movl $Rtn_point1, %eax
    jmp R
Rtn_point1:
    movl $Rtn_point2, %eax
    jmp R
Rtn_point2:
```

```
R:
    ...
    jmp *%eax
```

레지스터 앞에 *이 붙은 것에 대해선 우선 걱정할 필요가 없습니다. 위 코드도 문제가 있고, 사용하지 않을 문법입니다. 위 코드도 결국 함수 호출 문제를 완전하게 해결하지 못했습니다. 아래 경우를 보겠습니다.

```
P:                  Q:                  R:
        movl $rtn1, %eax        movl $rtn2, %eax        ...
        jmp Q               jmp R               jmp *%eax
rtn1:               rtn2:
                            jmp *%eax
```

레지스터에 반환될 위치를 저장하는 방식은 함수가 함수를 호출할 때 문제가 발생합니다. 위의 예시에서, P함수에서 돌아와야 할 좌표를 %eax에 저장했지만, Q에서 R을 호출함으로써 %eax를 수정해 버렸습니다. 따라서 Q에서 P로 돌아가려고 해도 돌아갈 좌표를 알 수가 없습니다.
위 문제를 해결하기 위해서는 돌아갈 주소를 레지스터와 독립적인 좌표에 저장할 필요성이 있어 보입니다. 또한, P와 Q에서 순서대로 저장한 주소를 거꾸로 돌아가기 위해서는 Q와 P 순서대로 꺼내서 참조할 수 있어야 합니다. 독립적인 저장 공간, 넣은 순서와 반대의 꺼내는 순서. 앞서 말했던 메모리 스택Memory Stack을 사용하기에 적절해 보입니다.
IA32에서는 위 문제를 스택Stack영역을 이용해 해결했습니다. 아래의 코드를 보겠습니다.

```
P:          Q:          R:
        call R      call R      ret
        call Q      ret
```

위 코드에서는 P가 Q와 R을 순차적으로 호출call하고, Q는 R을 호출한 후에, R은 아무 행동 없이 반환return합니다. 앞서 언급된 문제를 다 넣어둔 코드인데요, IA32에서는 call과 ret을 아래와 같이 정의하면서 문제를 해결했습니다.

call addr	pushl %eip jmp addr
ret	popl %eip

%eip는 프로그램이 다음으로 실행될 좌표를 저장하는 레지스터(또는 프로그램 카운터Program Counter)라는 것을 말했습니다. 물론 %eip는 프로그래머가 직접 사용할 수 없는 레지스터이기 때문에 위와 같은 직접적인 정의는 불가능하지만, call과 ret은 위의 의미와 완벽하게 일치하게 작동합니다. 즉, call은 다음으로 실행될 좌표인 %eip를 스택에 push한 후, addr로 jmp합니다. ret은 반대로 스택에 저장된 값, 즉 돌아갈 좌표를 %eip에 pop하며 해당 위치로 이동하게 합니다.

문제 2. 전달 인자Parameters

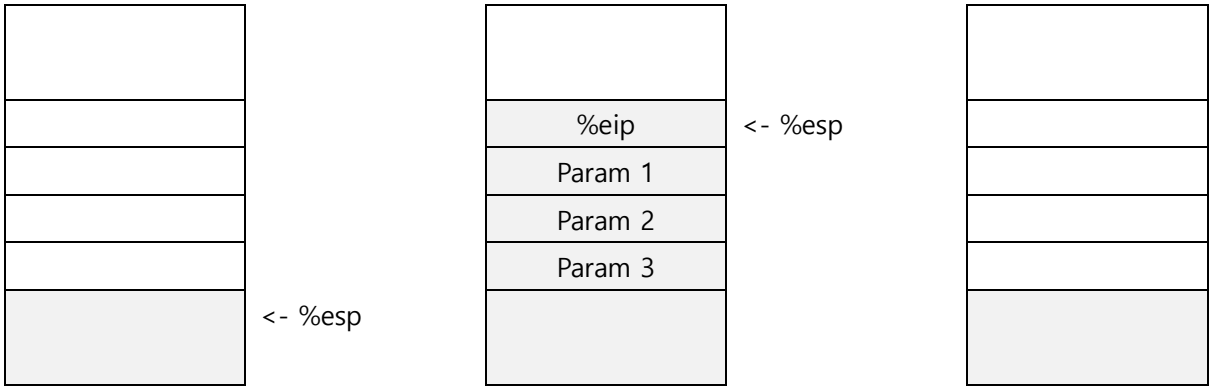
전달 인자Parameter도 함수에서는 매우 중요한 요소 중 하나입니다. 앞의 문제와 마찬가지로 직관적으로 해결해 봅시다. 어셈블리에서의 레지스터는 C언어에서의 전역 변수Global Variable과 같은 역할을 하니 레지스터에 값을 넘기면 어떨까요?

```
f:
        movl $3, %eax
        movl $4, %ebx
        movl $5, %ecx
        call g
```

```
g:
        movl $6, %eax
        call h
        # use %eax !!
        ret
```

```
h:
        # use %eax
        ret
```

위의 코드는 전달 값을 레지스터에 저장할 때의 문제점을 아주 간단히 보여줍니다. g와 h가 모두 전달 값을 %eax로 받고 있고, g는 전달 인자를 사용하기 전에 h를 호출합니다. 위 경우 %eax는 중복 사용되고, g와 h중 하나는 올바른 전달 인자를 사용하지 못합니다. 뿐만 아니라 크기가 32bit인 데이터만을 전달할 수 있고, 그 개수가 4개를 넘어가지 못한다는 문제도 발생합니다.
개수의 제한이 없이 데이터를 전달해야 하고, 그 값을 레지스터와 독립된 영역에 보존해야 할 것 같습니다. IA32에서는 위 문제도 스택Stack을 활용해서 아주 간단히 해결합니다.



즉, 위의 그림처럼 전달 인자를 넣고, 함수 호출call을 하며 %eip를 저장해줍니다. 그리고 함수 호출이 완료 된 이후 Stack을 다시 정리해주는 방식을 사용합니다. 위 방식에서는 전달 인자를 거꾸로 넣어주고 있습니다. 스택은 먼저 들어간 데이터가 더 밑에 싸이는 형태이기 때문에, 거꾸로 넣어야 앞 순서부터 함수에서 처리할 수 있게 됩니다.

```
f:
        pushl $5
        pushl $4
        pushl $3
        call g
        addl $12, %esp
```

```
g:
        movl 4(%esp), somewhere
        movl 8(%esp), somewhere
        movl 12(%esp), somewhere
        ret
```

꽤나 깔끔한 해결 방법입니다. f에서는 3개의 전달 인자를 넣은push 후 %esp에 12를 더해주며 4*3칸의 스택을 한번에 정리했습니다. popl을 한 데이터를 저장할 필요가 없다면, popl을 3번하는 것보다 12를 더해주는 것이 간편한 방법입니다. g에서는 %esp+4의 위치부터 순서대로 전달 인자를 저장합니다. %esp 위치에는 돌아가야 할 %eip 값이 저장되어 있기 때문입니다.
위 해결 방법은 꽤나 완벽해 보이지만 아쉬운 곳이 존재합니다. 문제 해결을 위해 사용한 스택은 전달 인자Parameters와 함수 반환return 이외에도 쓰이고, 이 뜻은 %esp의 값이 언제든 바뀔 수 있다는 뜻입니다. 따라서 %esp에 따라 상대적인 위치로 전달 인자를 참조하는 방식은 위험합니다.
이를 해결하기 위해 IA32에서는 %ebp를 활용합니다. 스택에 대한 포인터지만 사용하지는 않고 있었죠. 아래의 코드처럼 %ebp를 끌어올려 %ebp에 대한 상대적인 위치로 전달 인자를 참조하는 것이죠. 코드를 보겠습니다.

```
f:
        pushl $2
        pushl $1
        call g
        addl $8, %esp
```

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

        movl 8(%ebp), somewhere
        movl 12(%ebp), somewhere

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

위처럼 호출된 함수callee가 처음 실행될 때 %ebp를 스택에 Push한 후 시작합니다. 현재 스택에는 전달 인자가 역순으로 쌓여 있고, 그 위에는 돌아갈 위치, 그리고 %ebp의 원래 값이 저장되어 있는 형태입니다. %ebp를 스택에 굳이 저장하는 이유는, %ebp가 원래의 루틴, 또는 호출한 함수caller에서 어떤 목적으로 사용되고 있었을지 모르기 때문에 저장해두는 과정이 필요하기 때문입니다. 이렇게 외부의 값을 보존하기 위해 불리는 함수callee에서 시행하는 전처리를 프롤로그prolog, 함수가 호출된 후 하는 후처리를 에필로그epilog라고 합니다.
위의 방식대로라면 %ebp를 함수 내부에서 바꿔주지 않으면 전달 인자에 대한 상대 좌표가 항상 유지됩니다. %esp를 통해 참조하는 것보다 훨씬 안정적인 방식이라고 할 수 있습니다.

문제 3. 지역 변수Local Variable

프로그래밍의 대부분의 변수Variable는, 특히 C언어의 모든 변수는 생존 영역Scope를 가집니다. 따라서 모든 지역 변수Local Variable은 그 생명이 일시적이고 크기가 이미 결정되어 있습니다. 따라서 생존 영역에 영향을 받지 않고 크기가 32bit로 고정된 레지스터를 쓰기에는 비효율적입니다. IA32에서는 이런 지역 변수 역시, 스택Stack을 이용하여 해결합니다.
보다 간단히, %ebp를 스택에 Push한 이후 필요한 만큼의 공간을 Push를 통해 확보하고 %ebp의 상대 좌표를 이용해 사용하는 방식입니다. 아래의 코드로 확인해보겠습니다.

Example – add two integers

간단히 두 정수를 합하는 코드를 통해 지역 변수Local Variable의 사용법을 알아보겠습니다.

```
add2:
        pushl %ebp
        movl %esp, %ebp # Prolog

        subl $4, %esp # allocate 4byte for local variable

        movl 8(%ebp), %eax // move first parameter to local variable
        movl %eax, -4(%ebp) // Note: movl doesn't support mem-mem operation
        movl 12(%ebp), %eax
        addl %eax, -4(%ebp)

        movl -4(%ebp), %eax // return value is stored in %eax

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

지금까지 배운 것을 다 활용해 두개의 정수를 더하는 함수를 작성해 보았습니다. 위 코드에서는 이 함수의 반환값Return Value가 %eax에 된다고 생각해 보았습니다. 코드 이해가 어렵다면 스택 구조를 직접 그려보면 이해가 잘 될 것입니다.

문제 4. 레지스터 값 보존Register Handling

지금껏 문제들에서 모두 해결책으로 레지스터를 사용하려 시도해 보았지만 적절하지 못했습니다. 어셈블리에서는 레지스터가 전역 변수Global Variable과 같은 개념으로 쓰이고, 루틴routine과 서브 루틴subroutine이 명확히 구별되지 않는 특성 때문이었습니다. 이러한 특성은 레지스터 자체도 하나의 문제로 만듭니다. 루틴에서 사용하던 레지스터의 값을 보존하지 않는다면, 서브 루틴에서 레지스터의 값이 변형되지 않을 것이라 보장할 수 없기 때문입니다.
간단히, 어셈블리에는 한정된 개수의 레지스터만이 존재하고, 서브 루틴에서 레지스터의 값을 덮어 쓸 경우 루틴으로 돌아갔을 때 그 값이 보존되지 않는 것이 문제입니다. IA32에서는 마찬가지 해결책으로 스택Stack을 사용하면서, 호출하는 함수caller과 호출되는 함수callee간에 레지스터를 보존하는 책임이 있는 관례Convention을 만들어 뒀습니다.

caller-save					callee-save
%eax	%ecx	%edx	%ebx	%esi	%edi

코드로 예시를 먼저 보도록 하겠습니다.

Example – caller and callee-save registers

```
caller:
        # store caller-save registers
        pushl %eax
        pushl %ecx
        pushl %edx

        # store parameters and call function
        pushl $2
        pushl $1
        call callee
        addl $8, %esp # empty params
```

```
callee:
        # prolog
        pushl %ebp
        movl %esp, %ebp

        # store callee-save registers
        pushl %ebx
        pushl %esi
        pushl %edi

        # allocate local variable
        subl $4, %esp

        # operations using params
        movl 8(%ebp), somewhere
        movl 12(%ebp), somewhere

        # save something in local variable
        movl something, -16(%ebp)

        # restore callee-save registers
        movl -4(%ebp), %ebx
        movl -8(%ebp), %esi
        movl -12(%ebp), %edi

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

혼동을 방지하기 위해 지금까지 다뤘던 내용들을 모두 포함한 코드를 하나 작성했습니다. caller-save와 callee-save 레지스터를 저장하는 과정을 유의해서 보시면 될 것 같습니다.
사실, 레지스터를 보존하는 것은 말 그대로 관례Convention이고, 어느 쪽이 저장해야 한다는 명확한 규칙이나 필수적인 과정이 아닙니다. 하지만 저장하지 않았을 때 정해진 관례대로 책임이 있다는 것을 명시하기 위해 나눈 것이니, 저장을 해 두는 것이 좋은 습관입니다. 만약 특정 레지스터를 쓰지 않을 것이란 확신이 있는 경우, 가령 대부분의 경우에서 %esi와 %edi는 callee에서 경우에 따라 저장하지 않아도 무방합니다.

문제 5. 반환 값Return Value

드디어 마지막 문제입니다. 지금껏 함수를 열심히 작성한 것은 맞는데, 반환 값은 어떻게 전달해야 하는 것일까요? 이 경우 정석이라면 위의 수 많은 문제들과 마찬가지로 호출하는 함수Caller의 스택Stack에 반환 값을 넣어주는 것이 맞습니다. 함수를 호출하기 전에 전달 인자를 넣고, 반환 값의 크기에 해당하는 빈 공간을 스택에 만들어 둔 후 그곳에 값을 넣는 방식이죠.
하지만 IA32를 개발하는 많은 프로그래머들은 효율성을 위해 반환 값의 문제는 보통 레지스터를 활용합니다. 즉, 반환 값을 보통 %eax에 넣는데요, 앞서 %eax는 caller-save 레지스터이니 %eax에 값을 넣고 caller가 %eax를 복구하기 전에 반환 값을 사용하게 합니다. 아래는 짧은 예시입니다.

caller:	callee:
pushl %eax # save register call callee movl %eax, %ebx popl %eax	movl $1, %eax ret

위 코드에선 스택Stack을 %eax를 저장하는 용도와 call시 자동으로 저장되는 것 이외에는 쓰고 있지 않습니다. caller 부분의 4번째 줄에서 %eax를 %ebx에 이동하고 있는데요, callee에서 %eax에 저장했을 함수의 반환 값을 다른 곳에 옮겨 두고 %eax를 복구하고 있습니다.

함수: 정리

어셈블리에서 발생할 수 있는 함수 구현에서의 5가지 문제를 다뤄보았고, 해결해 보았습니다.

1) 함수 호출과 반환Call and Return
2) 전달 인자Parameters
3) 지역 변수Local Variable
4) 레지스터 값 보존Register Handling
5) 반환 값Return Value

어셈블리에서는 함수를 구현하기 위해 꽤나 명확한 해결 방법을 제시하고 있지만, 언어 자체의 특성상 잘못 사용하면 위험한 경우가 많습니다. 위 5가지 단계를 잘 기억해서 함수를 구현합시다.

코드 구조Code Structure

KAIST 전자공학을 위한 프로그래밍 구조 4번째 과제의 스켈레톤 코드Skeleton Code를 같이 해석하며 어셈블리 코드의 코드 구조를 파악해 보도록 하겠습니다. 다소 불필요한 주석은 지웠습니다.

```
        .equ    ARRAYSIZE, 20
        .equ    EOF, -1

.section ".rodata"

scanfFormat:
        .asciz "%s"
### ----------------------------------------------------------------

        .section ".data"

### ----------------------------------------------------------------

        .section ".bss"
buffer:
        .skip   ARRAYSIZE

### ----------------------------------------------------------------

        .section ".text"

        .globl  main
        .type    main,@function

main:

        pushl    %ebp
        movl     %esp, %ebp

input:
        pushl    $buffer
        pushl    $scanfFormat
        call     scanf
        addl     $8, %esp
```

```
        ## check if user input EOF
        cmp     $EOF, %eax
        je      quit

quit:
        ## return 0
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

어셈블러 지시자Assembler Directives

어셈블리는 언어 자체의 특성 외에도 어셈블러Assembler에게 특정한 명령을 하거나 신호를 주는 지시자Directives를 가지고 있습니다. 마치 C언어가 전처리기Preprocessor에게 #으로 시작하는 코드를 통해 지시를 하는 것과 유사합니다. 어셈블리에서는 이러한 지시자가 점(.)으로 시작합니다.

.equ symbol, expression	symbol의 값을 expression으로 설정합니다. C에서의 #define처럼 생각하면 될 것 같습니다.
.asciz strings	₩0이 포함된 strings 문자열을 정의합니다.
.skip size[,fill]	size-byte 크기의 공간을 fill로 채웁니다. fill이 생략된 경우 0으로 채웁니다.
.globl symbol	링커Linker에게 symbol을 표시합니다. 간단히, 위 코드의 경우 시작 지점을 main이라 알립니다.
.section	앞서 배웠던 스택 구조에서 어느 영역에 값을 저장할 것인지를 지정합니다. 가령, ".rodata" 는 read-only data로, 문자열 상수String literal와 같은 값들을 저장합니다. ".data" 는 전역 변수나 이미 값이 초기화된 값들을 저장합니다. ".bss"는 .data와 달리 초기화 되지 않은 값들을 저장합니다. C언어에서 "static int i;"와 같은 코드가 이곳에 저장됩니다. ".text"는 코드 영역입니다.

어셈블리 프로그래밍을 할 때, 위의 지시자를 이용해서 적절한 공간이나 변수, 상수를 선언하고 사용할 수 있게 됩니다. 지금껏 앞서 다뤘던 어셈블리 프로그래밍은 실제로 ".text" 영역에서 이루어지던 코드들로 해석하면 편할 것 같습니다.

Examples

예제가 많지 않아 직관적인 이해가 어렵고, 자료가 충분하지 않은 어셈블리를 보조하기 위해 예제를 준비했습니다. 하지만 아래의 코드들은 제가 직접 실행해본 코드가 아닙니다. 문제가 있다면 알려주시기 바랍니다.
또한, 여기 있는 코드가 과제 등에 직접 사용되더라도 그대로 사용하시면 안됩니다. 물론, 가능한 한 모든 곳에서 범용적으로 재사용 될 수 있을법한 코드와 사용되지 않을 법한 코드만을 예시로 만들긴 하였습니다.

printf("Hello, World!₩n");	.section "rodata" helloWorldString: .asciz "Hello, World!₩n" ~~ pushl $helloWorldString call printf addl $4, %esp

if(a == b)
printf("True₩n");
else
printf("False₩n");
// a: %eax, b: %ebx
	.section "rodata"
	trueString:
	.asciz "True₩n"
	falseString:
	.asciz "False₩n"
	
	~~
	cmpl %eax, %ebx
	jne else
	pushl $trueString
	call printf
	addl $4, %esp
	jmp endif
	else:
	pushl $falseString
	call printf
	addl $4, %esp
	endif:

a = '+' // a: %eax	movl $43, %eax # Note: '+' is 43 in ASCII
a = '+'	.equ PLUS, 43 movl $PLUS, %eax

int square_sum(int a, int b){
  return a*a + b*b;
}
	square_sum:
	  # prolog
	  pushl %ebp
	  movl %esp, %ebp
	
	  # store callee-save registers
	  pushl %ebx
	  pushl %esi
	  pushl %edi
	
	  # calculation
	  movl 8(%ebp), %eax
	  imull %eax, %eax
	  movl 12(%ebp), %ebx
	  imull %ebx, %ebx
	  addl %ebx, %eax
	
	  # restore callee-save registers
	  movl -4(%ebp), %ebx
	  movl -8(%ebp), %esi
	  movl -12(%ebp), %edi
	
	  # epilog
	  movl %ebp, %esp
	  popl %ebp
	  ret

int inc(int n){
  return n+1;
}
	inc:
	  # prolog
	  pushl %ebp
	  movl %esp, %ebp
	
	  # save n+1 in %eax
	  movl 8(%ebp), %eax
	  incl %eax
	
	  # epilog
	  movl %ebp, %esp
	  popl %ebp
	  ret

```
for(int i=1; i<10; i++){
        printf("2 * %d = %d₩n", i, 2*i);
}
```

```
.section "rodata"
printfFormat:
        .asciz "2 * %d = %d₩n"
~~
        movl $1, %ecx
loop:
        cmpl $10, %ecx
        jge endloop
        movl %ecx, %eax
        shll $1, %eax
        pushl %eax
        pushl %ecx
        pushl $printfFormat
        addl $12, %esp
        incl %ecx
        jmp loop
endloop:
```

```
int arr[4] = {1,3,5,7};
int sum = 0;
for(int i=0; i<4; i++)
      sum += arr[i]*arr[i];
// assume address 'arr' is given
// sum: %eax
```

```
        xorl %ecx, %ecx
        xorl %eax, %eax
loop:
        cmpl $4, %ecx
        jge endloop
        movl arr(,%ecx,4), %edx
        imull %edx, %edx
        addl %edx, %eax
        incl %ecx
        jmp loop
endloop:
```

Note

본 자료는 KAIST EE209 전자공학을 위한 프로그래밍 구조의 어셈블리 언어를 보조하기 위해 만들어진 자료입니다. 하지만 IA32 Assembly에 대한 전반적인 내용을 다루고 있기 때문에, 다른 수업 또는 Assembly를 공부하기 위한 보조 자료로 활용하셔도 좋습니다.
편한 이해를 위해 전산학적 엄밀성을 포기했거나, 자세한 설명을 기재하지 않은 경우가 있습니다. 틀린 내용이 있을 경우 아래의 연락처로 연락 주시면 수정 후 재배포하도록 하겠습니다.

Reference

PPT 자료, 전자공학을 위한 프로그래밍 구조, EE209, KAIST
HW 4, 전자공학을 위한 프로그래밍 구조, EE209, KAIST
PPT 자료, 시스템 프로그래밍, CS230, KAIST
Computer Systems: A Programmer's Architecture
PPT 자료, 전산기조직, CS311, KAIST
어셈블러 지시자, http://tigcc.ticalc.org/doc/gnuasm.html

Version

2018년 5월 3일 오전 2시 1분 업데이트
: Assembly Functions 이전까지 제작

2018년 5월 5일 오후 11시 3분 업데이트
: 초판 완성

Contact

juanlee@kaist.ac.kr