

1 Register

- %esp = stack pointer
- %ebp = base pointer
- %eax = accumulator, return Werte von Funktionen werden hier abgelegt.
- %ebx = base index (array manipulation)
- %ecx = counter (array manipulation)
- %edx = data / general register
- %esi = source index (string manipulation)
- %edi = destination index (string manipulation)
- %eip = instruction pointer

Ausser %eip und %esp sind alles General Purpose Register, man kann auch %ebx für eine Array-Manipulation verwenden.

1.0.1 movl

movl kann in drei Varianten verwendet werden:

- movl "register", "register"
- movl "register", [Expression]
- movl [Expression], "register"

Generelle Funktion für Expressions: $D(Rb, Ri, S) = Mem[Reg[Rb] + S \cdot Reg[Ri] + D]$

- D: Konstante in Byte(4 Byte für 64b)
- Rb: Base Register
- Ri: Index Register, können alle sein ausser %esp und %ebp
- S: Skalar in Zweierpotenz

	Ausdruck	Berechnung	Adresse im Hauptspeicher
Beispiele:	0x8(%edx)	0xf000 + 0x8	0xf008
	(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
	0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

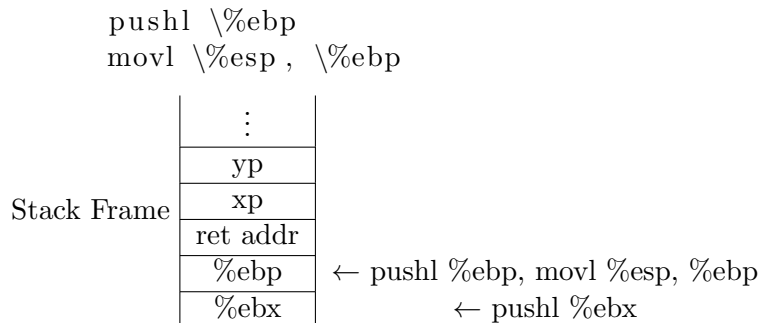
2 Function Call

2.1 Stack Frame

%ebp zeigt immer auf die "Basis" des stacks, heisst alle Adressen kleiner als %ebp gehören zur momentan ausgeführten Methode. Die Parameter dieser Methode sind dabei auf den Adressen grösser als %ebp abgespeichert. Die Speicherstelle, auf die %ebp hinzeigt, ist der &ebp Wert der vorherigen Methode. 4(%ebp) beinhaltet die Return-Adresse für diese Methode, alles höher als 4(%ebp) sind Parameter der momentanen Methode.

2.2 Function Call Setup

Nachdem der Aufrufer die Parameter auf den Stack abgelegt und "Call Function" ausgeführt hat.



2.3 Function Call Teardown

```

// allenfalls Returnwert in \%eax speichern
movl \%ebp, \%esp
pop \%ebp
return

```

3 Instruktionen

3.1 Arithmetische Operatoren

3.1.1 Binäre Operatoren

Alle binären Operatoren lesen aus dem Source Register und den berechneten Wert in das Destination Register.

Befehl	Beschreibung
addl	Dest += Source
subl	Dest -= Source
imull	Dest *= Source
sall	Dest << Source
sarl	Dest >> Source, fällt mit 1 auf falls MSB = 1
shrl	Dest >> Source, fällt immer mit 0 auf
leal	siehe LEA Instruction.
xorl	...
andl	...
orl	...

3.1.2 Unäre Operatoren

Befehl	Beschreibung
incl	increment
decl	decrement
negl	negate
notl	not operator

3.1.3 LEA Instruction

Vom Internet: LEA, the only instruction that performs memory addressing calculations but doesn't actually address memory. LEA accepts a standard memory addressing operand, but does nothing more than store the calculated memory offset in the specified register, which may be any general purpose register.

What does that give us? Two things that ADD doesn't provide:

the ability to perform addition with either two or three operands, and the ability to store the result in any register; not just one of the source operands.

3.2 Compare und Konditionen

cmpl	compare
testl	(jmp %eax %eax) $\hat{=}$ $\hat{=}$ 0, ob %eax grÖßer, kleiner oder = 0 ist.

3.2.1 Flags

Abkürzung	Name	wird gesetzt durch
ZF	Zero Flag	wird von testl gesetzt.
SF	Signed Flag	wird von testl gesetzt.
OF	Overflow Flag	von arithmetischen Operationen gesetzt.
CF	Carry Flag	von arithmetischen Operationen gesetzt.

Befehl	Ausdruck	Beschreibung
sete	<i>ZF</i>	Equal / Zero
setne	<i>ZF</i>	Not Equal / Not Zero
sets	<i>SF</i>	Negative
setns	<i>SF</i>	Nonnegative

3.3 Control Statements

jmp (label)	jump
je (label)	jump equal
jne (label)	jump not equal
js (label)	jump negative
jns (label)	jump not negative
jg (label)	jump greater
jge (label)	jump greater or equal
jl (label)	jump less
jle (label)	jump less or equal
ja (label)	jump above (unsigned)
jb (label)	jump below (unsigned)
push Src	
pop Dest	
call (label)	
ret	

4 Loops und If's

4.1 If Statement

C Code:

```
int absdiff(int x,int y)
{
    int result;

    if(x > y)
        result = x-y;
    else
        result = y-x;

    return result;
}
```

Assembler:

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

    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    compl %eax,%edx
    jle .L9
    movl %edx,%eax

.L9:
    movl %ebp,%esp
    popl %ebp
    ret
```

4.2 Loops

5 Bitwise Magix

```
int bitXor(int x, int y) {
    return ~x & y;
}

int isEqual(int x, int y) {
    return !(x ^ y);
}
```