

CSC 6580

Spring 2020

Instructor: Stacy Prowell

Assembly Language



Resources

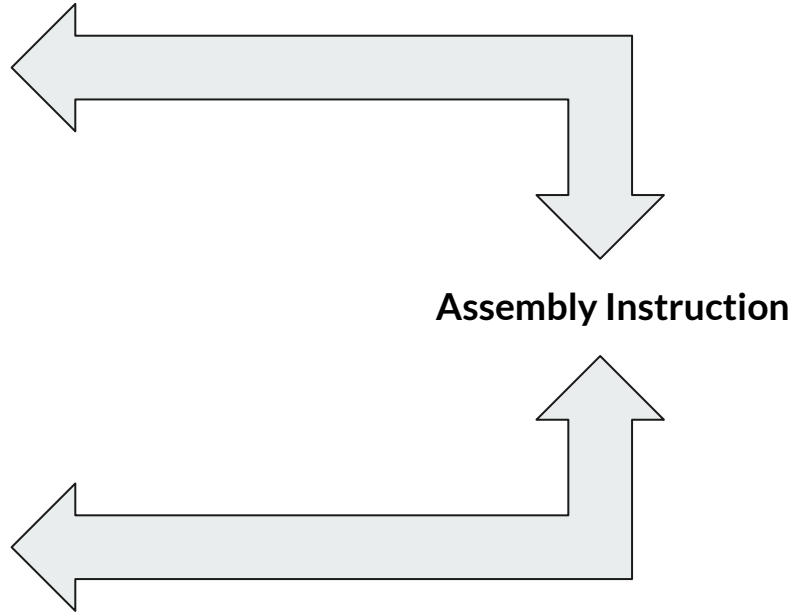
- ODA <https://onlinedisassembler.com/odaweb>
- X86 Opcode and Instruction Reference <https://ref.x86asm.net>
- Intel Documentation (search Intel 64 and IA-32 Architectures Software Developer's Manual)
- The incomplete reference <https://felixcloutier.com/x86>
- The Compiler Explorer <https://godbolt.org/>

Processor State

- Registers (**RAX**, **RBX**, ...)
- Flags (**ZF**, **CF**, **SF**, ...)
- Modes
- Input / Output
- other...

Memory

- DRAM via L1, L2, L3 cache



Assembly instructions **read** and/or **modify** processor state and memory (through the processor's interface, the cache controller)



Assembly Instructions

There are a lot. How many? The number varies, depending on how you count.

<https://bit.ly/2GmdT7h>

Measure	Count	Comment
AT&T mnemonic (e.g., <code>addl</code>)	1,279	Counts the number of unique mnemonics in AT&T syntax.
Intel mnemonic (e.g., <code>add</code>)	981	This is a rough estimate of the number of different kinds of operations the x86 instruction set can perform, ignoring the operand type and size. There are various caveats as described earlier, such as some operations having multiple different mnemonics.
Mnemonic and operand types (e.g., <code>add_r32_imm32</code>)	3,683	Distinguishes instructions in every way possible and is thus a sort of upper bound on the number of instructions. If you are looking for a very fine-grained notion of instruction, this is a good measure.
Mnemonic and operand width (e.g., <code>add_32_32</code>)	2,034	This is an estimate of the number of different kinds of instructions, if an operation on 8 bits is considered to be different from the same operation on 16 bits (because sometimes, these actually have different semantics, as shown with the zeroing of the upper 32 bits for the <code>addl</code> instruction). Does not distinguish instructions that operate on registers vs. constants vs. memory locations.
Every valid bit sequence (e.g., <code>0x83</code> , <code>0xC0</code> , <code>0x01</code>)	?	Counts every bit sequence that makes a valid x86 instruction. Unfortunately there are too many to count.



Assembly Instructions

It is arguable that the official documentation is not complete, either.

See the “Sandsifter” project:

<https://github.com/xoreaxeaxeax/sandsifter>

So *even assemblers* don't understand the complete x86 ISA.

Breaking the x86 ISA

Christopher Domas
xoreaxeaxeax@gmail.com

July 27, 2017

Abstract— A processor is not a trusted black box for running code; on the contrary, modern x86 chips are packed full of secret instructions and hardware bugs. In this paper, we demonstrate how page fault analysis and some creative processor fuzzing can be used to exhaustively search the x86 instruction set and uncover the secrets buried in a chipset. The approach has revealed critical x86 hardware glitches, previously unknown machine instructions, ubiquitous software bugs, and flaws in enterprise hypervisors.

Both vary based on the *specific processor* tested.

source:

<https://github.com/xoreaxeaxeax/sandsifter>



Assembly Instructions

You are *not expected* to become an expert on assembly language on any platform; there are *very few* of those, and even their knowledge is incomplete and sometimes wrong. You just need to know the basics.

New features are added all the time, for a variety of reasons and sometimes for specific, special cases. Compiler writers struggle to keep up.

(Curious about how compilers work? LLVM has good documentation.
<https://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.pdf>)



Aside: Analyzing Programs

Let's answer some questions about this program.

- How many times does it loop?

How long does it take? We'll compile it, and then disassemble it using `objdump` ^{**}:

```
$ gcc -c delay.c
$ objdump -d delay.o
```

```
void delay() {
    unsigned char i, j;
    j = 0;
    while(--j) {
        i = 0;
        while(--i);
    }
}
```

^{**} We're going to talk more about `objdump` later

Aside: Analyzing Programs

```
void delay() {  
    unsigned char i, j;  
    j = 0;  
    while(--j) {  
        i = 0;  
        while(--i);  
    }  
}
```

gcc -c delay.c

0000000000000000 <delay>:

```
0: 55  
1: 48 89 e5  
4: c6 45 ff 00  
8: eb 0e  
a: c6 45 fe 00  
e: 80 6d fe 01  
12: 80 7d fe 00  
16: 75 f6  
18: 80 6d ff 01  
1c: 80 7d ff 00  
20: 75 e8  
22: 90  
23: 5d  
24: c3
```

```
push    rbp  
mov     rbp, rsp  
mov     BYTE PTR [rbp-0x1], 0x0  
jmp     18 <delay+0x18>  
mov     BYTE PTR [rbp-0x2], 0x0  
sub     BYTE PTR [rbp-0x2], 0x1  
cmp     BYTE PTR [rbp-0x2], 0x0  
jne     e <delay+0xe>  
sub     BYTE PTR [rbp-0x1], 0x1  
cmp     BYTE PTR [rbp-0x1], 0x0  
jne     a <delay+0xa>  
nop  
pop     rbp  
ret
```

Aside: Analyzing Programs

```
void delay() {  
    unsigned char i, j;  
    j = 0;  
    while(--j) {  
        i = 0;  
        while(--i);  
    }  
}
```

gcc -c -O delay.c

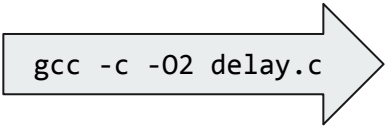
0000000000000000 <delay>:

```
0:  ba ff ff ff ff  
5:  b8 00 00 00 00  
a:  2c 01  
c:  75 fc  
e:  80 ea 01  
11: 75 f2  
13: c3
```

```
mov    edx,0xffffffff  
mov    eax,0x0  
sub    al,0x1  
jne    a <delay+0xa>  
sub    dl,0x1  
jne    5 <delay+0x5>  
ret
```

Aside: Analyzing Programs

```
void delay() {  
    unsigned char i, j;  
    j = 0;  
    while(--j) {  
        i = 0;  
        while(--i);  
    }  
}
```



```
gcc -c -O2 delay.c
```

```
0000000000000000 <delay>:  
    0:   c3
```

```
ret
```



Aside: Analyzing Programs

If you analyze source code, you are analyzing (at best) an approximation of the real behavior.

Compilers can, *and do*, omit code you wrote, insert code you didn't write, and change your code for a variety of purposes. If you want to know (more completely) what a program is going to do, you have to analyze the machine code. Even the output of a disassembler might not be true (which we will see later).

A Bit on the X86 Instruction Set Architecture (ISA)



Some X86 Architecture: 16-bit registers

16-bit registers

- Accumulator AX
- Base BX
- Counter CX
- Data DX
- Stack Pointer SP
- (Stack) Base Pointer BP
- Source Index SI
- Destination Index DI
- Instruction Pointer IP

The AX, BX, CX, and DX registers are high / low byte-addressable.

- AX = AH.AL
- BX = BH.BL
- CX = CH.CL
- DX = DH.DL

8 bits: A byte

16 bits: A word



Some X86 Architecture: 32-bit registers

Extended registers:

- EAX, EBX, ECX, EDX
- ESP, EBP
- ESI, EDI
- EIP

Register aliasing:

EAX

= (16 bits).AX

= (16 bits).AH.AL

8 bits: A byte

16 bits: A word

32 bits: A dword (double word)



Some X86_64 Architecture: 64-bit registers

“R” is for register.

- RAX, RBX, RCX, RDX
- RSP, RBP
- RSI, RDI
- RIP

But also...

- R8, R9, R10, R11, R12, R13, R14, R15

(RAX through RDI count as R0-R7.)

Aliasing:

RAX

= (32-bits).EAX

= (32-bits).(16-bits).AX

= (32-bits).(16-bits).AH.AL

8 bits: A byte

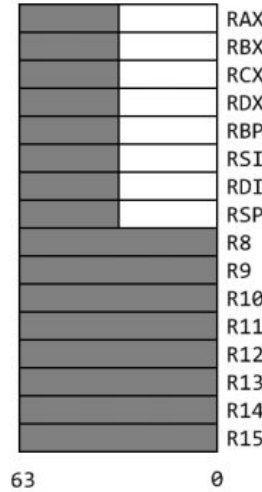
16 bits: A word

32 bits: A dword (double word)

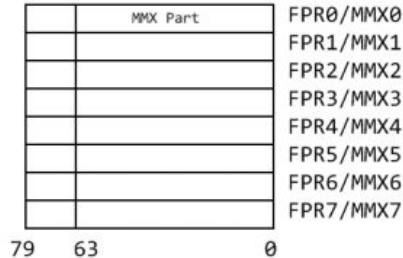
64 bits: A qword (quadword)

Some X86_64 Architecture: 64-bit registers & 80-bit registers & 128-bit registers

General Purpose Registers (GPRs)

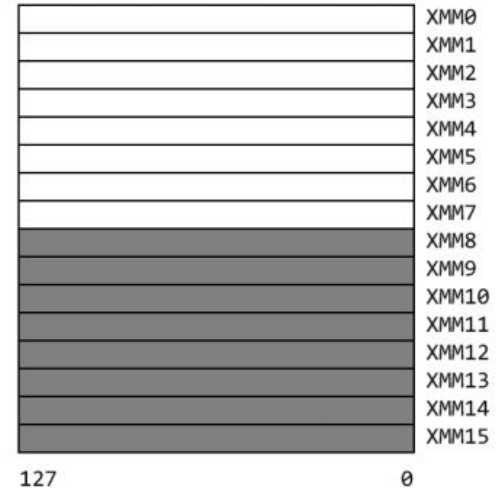
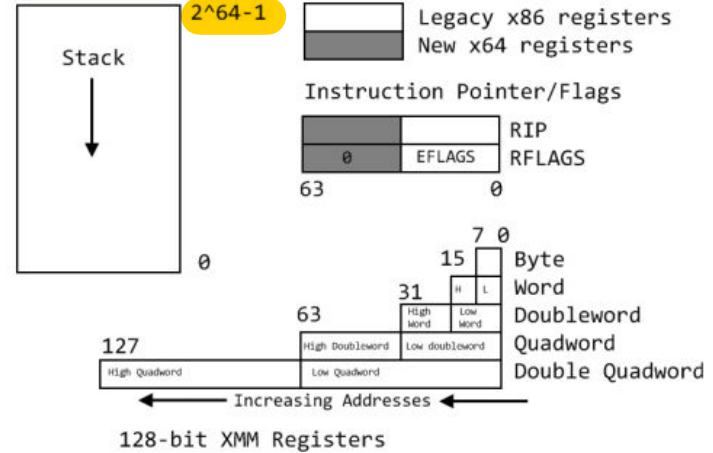


80-bit floating point
and 64-bit MMX registers
(overlaid)



Also: 6 segment registers, control, status, debug, more

Address Space





Some X86 Architecture: Segment Registers

- Code segment CS
- Data segment DS
- Stack segment SS
- Extra segment registers ES, FS, GS

The extra segment registers are typically used by the operating system.

286: CS, DS, SS, ES

386: Added FS, GS

You can read about *protected mode* and the LDT and GDT. Head to Wikipedia and search Global Descriptor Table.



Some X86 Architecture: **Flags (FLAGS)**

- Carry CF
- Parity PF (even = 1, odd = 0)
- Adjust / Auxiliary AF (4-bit carry)
- Zero ZF
- Sign SF
- Trap TF (single-step)
- Interrupt Enable IF (allowed = 1)
- Direction DF (increment = 0)
- Overflow OF
- Privilege IOPL (2 bits)

Direction flag? See the [MOVS](#) family of instructions.



Some X86 Architecture: More flags (**EFLAGS**)

- Carry **CF**
- Parity **PF** (even = 1, odd = 0)
- Adjust / Auxiliary **AF** (4-bit carry)
- Zero **ZF**
- Sign **SF**
- Trap **TF** (single-step)
- Interrupt Enable **IF** (allowed = 1)
- Direction **DF** (increment = 0)
- Overflow **OF**
- Privilege **IOPL** (2 bits)

- Resume **RF**
- Virtual 8086 **VM** (compatibility = 1)
- Alignment Check **AC** (enabled = 1)
- Virtual Interrupt **VIF**
- Virtual Interrupt Pending **VIP**
- CPU ID **ID**
- Virtual Address Descriptor **VAD** (allowed = 1)

Intel x86 FLAGS register ^[1]						
Bit #	Mask	Abbreviation	Description	Category	=1	=0
FLAGS						
0	0x0001	CF	Carry flag	Status	CY(Carry)	NC(No Carry)
1	0x0002		Reserved, always 1 in EFLAGS ^{[2][3]}			
2	0x0004	PF	Parity flag	Status	PE(Parity Even)	PO(Parity Odd)
3	0x0008		Reserved ^[3]			
4	0x0010	AF	Adjust flag	Status	AC(Auxiliary Carry)	NA(No Auxiliary Carry)
5	0x0020		Reserved ^[3]			
6	0x0040	ZF	Zero flag	Status	ZR(Zero)	NZ(Not Zero)
7	0x0080	SF	Sign flag	Status	NG(Negative)	PL(Positive)
8	0x0100	TF	Trap flag (single step)	Control		
9	0x0200	IF	Interrupt enable flag	Control	EI(Enable Interrupt)	DI(Disable Interrupt)
10	0x0400	DF	Direction flag	Control	DN(Down)	UP(Up)
11	0x0800	OF	Overflow flag	Status	OV(Overflow)	NV(Not Overflow)
12-13	0x3000	IOPL	I/O privilege level (286+ only), always 1 ^[clarification needed] on 8086 and 186	System		
14	0x4000	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System		
15	0x8000		Reserved, always 1 on 8086 and 186, always 0 on later models			
EFLAGS						
16	0x0001 0000	RF	Resume flag (386+ only)	System		
17	0x0002 0000	VM	Virtual 8086 mode flag (386+ only)	System		
18	0x0004 0000	AC	Alignment check (486SX+ only)	System		
19	0x0008 0000	VIF	Virtual interrupt flag (Pentium+)	System		
20	0x0010 0000	VIP	Virtual interrupt pending (Pentium+)	System		
21	0x0020 0000	ID	Able to use CPUID instruction (Pentium+)	System		
22-31	0xFFC0 0000		Reserved	System		
RFLAGS						
32-63	0xFFFF FFFF... ...0000 0000		Reserved			

All the flag bits live in the FLAGS register (and EFLAGS and RFLAGS).

You cannot access this register directly... but you can manipulate it using special instructions:

- PUSHF pushes the FLAGS register content on the top of the stack
- POPF loads the FLAGS register from the top of the stack

Similar instructions extend to EFLAGS and RFLAGS.

- LAHF moves FLAGS bits 0-7 into AH
- SAHF moves AH into bits 0-7 of FLAGS

source: https://en.wikipedia.org/wiki/FLAGS_register

Creating an Assembly Routine



A Very Simple Example

Given a list of unsigned ints, find the largest. Doing this in C is fairly obvious.

```
unsigned int max(unsigned int count, unsigned int vals[]) {  
    unsigned int max = 0;  
    for (int index = 0; index < count; ++index) {  
        if (vals[index] > max) max = vals[index];  
    } // end of for loop  
    return max;  
}
```



A Very Simple Example

On my 64-bit Ubuntu system, an `unsigned int` is 4 bytes long (32 bits).

Basic function boilerplate first.

- Save the stack pointer: `RSP`
- Use `RBP` to access the stack

How do we get the arguments? How do we return the answer?

```
push rbp
mov rbp, rsp
;
; body
;
mov rsp, rbp ; Not always done.
pop rbp
ret
```

Aside: Calling Convention

Linux, MacOS X, Solaris, and most operating systems use the AMD64 calling convention.

<https://bit.ly/2TU24Nw>

There is actually a lot of information in a calling convention.

Microsoft (and UEFI!) has their own (not shown):

<https://bit.ly/2TWA6kd>

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1 st return register	No
%rbx	callee-saved register	Yes
%rcx	used to pass 4 th integer argument to functions	No
%rdx	used to pass 3 rd argument to functions; 2 nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2 nd argument to functions	No
%rdi	used to pass 1 st argument to functions	No
%r8	used to pass 5 th argument to functions	No
%r9	used to pass 6 th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes
%xmm0-%xmm1	used to pass and return floating point arguments	No
%xmm2-%xmm7	used to pass floating point arguments	No
%xmm8-%xmm15	temporary registers	No
%mm0-%mm7	temporary registers	No
%k0-%k7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2-%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes



Aside: Calling Convention: TL;DR?

The registers **RDI**, **RSI**, **RDX**, **RCX**, **R8**, and **R9** (in order) should be used for *integer and memory address arguments* and **XMM0**, **XMM1**, **XMM2**, **XMM3**, **XMM4**, **XMM5**, **XMM6**, and **XMM7** (in order) should be used for *floating point arguments*.

For *system calls*, **R10** is used instead of **RCX**.

Additional arguments are passed on the stack and the *return value* is stored in **RAX** if it is an integer or pointer, and in **XMM0** if it is a floating point result. (Sometimes a second register might be used, too.)

You must make sure you *do not modify*, or you *save and restore*, the value in **RBP**, **RBX**, **R12**, **R13**, **R14**, and **R15**.



A Very Simple Example

Function boilerplate:

- Save base pointer (because it is probably being used by the calling function) on the stack.
- Move the stack pointer into the base pointer.
- Do stuff...
- Reset the stack pointer to the (unmodified) base pointer.
- Restore the base pointer from the stack.
- Return.

```
push rbp
mov rbp, rsp
    ; Array length is in EDI
    ; Array pointer is in RSI
    ; Return value should be in RAX
mov rsp, rbp
pop rbp
ret
```



A Very Simple Example

Need a scratch register to hold the maximum value (the `max` variable in the original program); might as well be `RAX`. Be sure to use a register whose value is *not guaranteed* to be preserved across a function call.

Need a pointer into the array (the `index` variable in the original program). Let's use `ECX` (`unsigned int` is 32 bits), since it is a scratch register not expected to be preserved.

```
push rbp
mov rbp, rsp
mov rax, 0
mov ecx, 0
    ; Loop goes here
mov rsp, rbp
pop rbp
ret
```



A Very Simple Example

We need the loop index to run from zero up to the value in **EDI**.

Loops in assembly are almost always *do-while* loops. That is, the body is *always* executed at least once.

To get a proper while loop we need to test if **EDI** is zero before we start, or we'll have a bug!

```
push rbp
mov rbp, rsp
mov rax, 0
mov ecx, 0
```

loop:

; Compare goes here

```
inc ecx
cmp ecx, edi
jl loop
mov rsp, rbp
pop rbp
ret
```



A Very Simple Example

We need the loop index to run from zero up to the value in **EDI**.

Loops in assembly are almost always *do-while* loops. That is, the body is *always* executed at least once.

To get a proper while loop we need to test if **EDI** is zero before we start, or we'll have a bug!

```
push rbp
mov rbp, rsp
mov rax, 0
cmp edi, 0
je out
mov ecx, 0
loop:
    ; Compare goes here
    inc ecx
    cmp ecx, edi
    jl loop
out:
mov rsp, rbp
pop rbp
ret
```




A Very Simple Example

Next we need to see if we have found a better maximum value and, if so, replace the current maximum.

To do this, we can use a computed (relative) address.

```
push rbp
mov rbp, rsp
mov rax, 0
cmp edi, 0
je out
mov ecx, 0

loop:
    ; Compare goes here
    inc ecx
    cmp ecx, edi
    jl loop

out:
    mov rsp, rbp
    pop rbp
    ret
```

Aside: Addressing

$$\left\{ \begin{array}{l} \text{FS : } \left[\begin{array}{c} \vdots \\ \text{GPR} \\ \vdots \end{array} \right] + \left(\left[\begin{array}{c} \vdots \\ \text{GPR} \\ \vdots \end{array} \right] * \left[\begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \end{array} \right] \right) \\ \text{GS : } \left[\begin{array}{c} \vdots \\ \text{GPR} \\ \vdots \end{array} \right] \end{array} \right\} + \text{displacement}$$

RIP

The X86 ISA computes addresses in a way that is helpful for pointers.

The address is:

$$[base + index * scale + displacement]$$

The *base* and *index* are registers, and the *displacement* is an integer. The *scale* may be 1, 2, 4, or 8. The register sizes must be the same.



Aside: Address Examples

These are all in Intel syntax. We'll cover AT&T syntax later.

Assume `EAX` contains 512 and `EBX` contains 5.

<code>mov BYTE [60], al</code>	Copy the byte at address 60 (in the data segment <code>DS</code>) into <code>AL</code>
<code>mov WORD [rax], ax</code>	Copy the word (16 bits, LE) at address 512 into <code>AL</code>
<code>cmp DWORD [rax+60], ebx</code>	Compare the double word (32 bits, LE) at address $512 + 60 = 572$ with the value in <code>EBX</code>
<code>mov rsi, QWORD [rax+rbx*8]</code>	Copy the quad word (64 bits, LE) stored in <code>RSI</code> to address $512 + 5*8 = 552$
<code>lea esi, [rax+rbx*4+60]</code>	Copy the address $512 + 5*4 + 60 = 592$ into <code>ESI</code>



Aside: Math

Load effective address (LEA) is great for simple math.

Need to compute $756 + \text{EAX} * 4$?

```
lea eax, [756+eax*4]
```



A Very Simple Example

Compare `EAX` to the array value and, if `EAX` is greater, skip replacing it's value.

This kind of “jump over” logic is common.

```
push rbp
mov rbp, rsp
mov rax, 0
cmp edi, 0
je out
mov ecx, 0

loop:
    cmp eax, DWORD [rsi+rcx*4]
    jg over
    mov eax, DWORD [rsi+rcx*4]
over:
    inc ecx
    cmp ecx, edi
    jl loop

out:
    mov rsp, rbp
    pop rbp
    ret
```



A Very Simple Example

We don't ever modify the stack, so *technically* we can dispense with the stack stuff.

```
push rbp  
mov rbp, rsp  
mov rax, 0  
cmp edi, 0  
je out  
mov ecx, 0  
loop:  
    cmp eax, DWORD [rsi+rcx*4]  
    jg over  
    mov eax, DWORD [rsi+rcx*4]  
over:  
    inc ecx  
    cmp ecx, edi  
    jl loop  
out:  
mov rsp, rbp  
pop rbp  
ret
```



A Very Simple Example

We don't ever modify the stack, so *technically* we can dispense with the stack stuff.

It's usual to replace:

- `cmp rdi, 0` with `test rdi, rdi`
- `mov rax, 0` with `xor rax, rax`

```
mov rax, 0
cmp edi, 0
je out
mov ecx, 0

loop:
    cmp eax, DWORD [rsi+rcx*4]
    jg over
    mov eax, DWORD [rsi+rcx*4]

over:
    inc ecx
    cmp ecx, edi
    jl loop

out:
    ret
```



A Very Simple Example

We don't ever modify the stack, so *technically* we can dispense with the stack stuff.

It's usual to replace:

- `cmp rdi, 0` with `test rdi, rdi`
- `mov rax, 0` with `xor rax, rax`

```
xor rax, rax
test edi, edi
je out
xor ecx, ecx
loop:
    cmp eax, DWORD [rsi+rcx*4]
    jg over
    mov eax, DWORD [rsi+rcx*4]
over:
    inc ecx
    cmp ecx, edi
    jl loop
out:
    ret
```




A Very Simple Example

Let's assemble it! Put the code in a file called

`max.asm`.

We can call the function if we export it (then we can import it in C). Use `global` for that.

We can assemble it with NASM**:

```
nasm -f elf64 max.asm
```

```
; nasm -f elf64 max.asm
```

```
; gcc -static -o max max.o
```

```
global max
```

```
max:
```

```
xor rax, rax
test edi, edi
je out
xor ecx, ecx
```

```
loop:
```

```
cmp eax, DWORD [rsi+rcx*4]
jg over
mov eax, DWORD [rsi+rcx*4]
```

```
over:
```

```
inc ecx
cmp ecx, edi
jl loop
```

```
out:
```

```
ret
```

** The Netwide Assembler: <https://nasm.us/>



Aside: Linking Assembly and C

- Create `.c` files and compile them (`gcc -c`).
- Create `.asm` files and assemble them (`nasm -felf64`).
- Link everything (`ld` / `gcc`).
- From assembly use `extern` to declare C functions you need. Then set up the arguments and `call` the function, according to the calling convention.
- From C use `extern` to declare assembly functions you need. Then make sure the assembly handles the arguments according to the calling convention.



Aside: Linking Assembly and C

- Linking is *hard*. If you can, you should let `gcc` do it. You *can* do it with `ld`, but it can be *hard*.

```
$ gcc -static -m64 -o hello greet.o hello.o
```

```
$ ld -static -melf_x86_64 -o hello greet.o hello.o \  
    --start-group -lc --end-group \  
    /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o \  
    /usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/crtn.o
```

- Really curious? Run the `gcc` link with `-v` at the end.



A Very Simple Example

```
#include <stdio.h>

extern unsigned int max(unsigned int, unsigned int[]);

int main(int argc, char * argv[], char * envp[]) {
    unsigned int values[] = {
        65, 62, 72, 93, 103, 21, 18, 0
    };
    printf("%d\n", max(sizeof(values)/sizeof(values[0]), values));
    return 0;
}
```

Here `max` could be provided by the C function shown earlier, or by the assembly function. It can be provided by anything that satisfies the calling convention.



A Very Simple Example

```
$ nasm -f elf64 max.asm  
$ gcc -o test test.c max.o  
$ ./test  
103  
$ gcc -c -O2 max.c  
$ gcc -o test test.c max.o  
$ ./test  
103
```



A Very Simple Example

Let's look at our program's machine code. We used 0x18, or 24 bytes.

```
0000000000000000 <max>:
  0: 48 31 c0          xor     rax,rax
  3: 85 ff             test    edi,edi
  5: 74 10             je      17 <out>
  7: 31 c9             xor     ecx,ecx

0000000000000009 <loop>:
  9: 3b 04 8e          cmp     eax,DWORD PTR [rsi+rcx*4]
  c: 7f 03             jg      11 <over>
  e: 8b 04 8e          mov     eax,DWORD PTR [rsi+rcx*4]

0000000000000011 <over>:
 11: ff c1             inc     ecx
 13: 39 f9             cmp     ecx,edi
 15: 7c f2             jl      9 <loop>

0000000000000017 <out>:
 17: c3               ret
```

Let's Compare

C source shown earlier compiled by GCC with -O2 and disassembled with objdump.

Recall: EDI is length, RSI is array pointer.

0000000000000000 <max>:

```
0: 85 ff
2: 74 24
4: 8d 47 ff
7: 48 8d 4c 86 04
c: 31 c0
e: 66 90
10: 8b 16
12: 39 d0
14: 0f 42 c2
17: 48 83 c6 04
1b: 48 39 ce
1e: 75 f0
20: c3
21: 0f 1f 80 00 00 00 00
28: 31 c0
2a: c3
```

```
test    edi,edi
je      28 <max+0x28>
lea     eax,[rdi-0x1]
lea     rcx,[rsi+rax*4+0x4]
xor     eax,eax
xchg    ax,ax
mov     edx,DWORD PTR [rsi]
cmp     eax,edx
cmovb   eax,edx
add     rsi,0x4
cmp     rsi,rcx
jne     10 <max+0x10>
ret
nop     DWORD PTR [rax+0x0]
xor     eax,eax
ret
```

Let's Compare

What's going on here?

```
lea eax,[rdi-0x1]
```

$eax = edi - 1$

```
lea rcx,[rsi+rax*4+0x4]
```

$rcx = rsi + (edi - 1) * 4 + 4$

$rcx = rsi + edi * 4 - 4 + 4$

$rcx = rsi + edi * 4$

Now **rcx** is the address just past the end of the array. No idea why this is done.

0000000000000000 <max>:

0:	85 ff	test	edi,edi
2:	74 24	je	28 <max+0x28>
4:	8d 47 ff	lea	eax,[rdi-0x1]
7:	48 8d 4c 86 04	lea	rcx,[rsi+rax*4+0x4]
c:	31 c0	xor	eax,eax
e:	66 90	xchg	ax,ax
10:	8b 16	mov	edx,DWORD PTR [rsi]
12:	39 d0	cmp	eax,edx
14:	0f 42 c2	cmovb	eax,edx
17:	48 83 c6 04	add	rsi,0x4
1b:	48 39 ce	cmp	rsi,rcx
1e:	75 f0	jne	10 <max+0x10>
20:	c3	ret	
21:	0f 1f 80 00 00 00 00	nop	DWORD PTR [rax+0x0]
28:	31 c0	xor	eax,eax
2a:	c3	ret	

Let's Compare

Why?

It's a no-op. It changes no registers, and no flags, and just consumes two bytes to align the start of the loop on a 8-byte boundary.

This happens a lot.

```
0000000000000000 <max>:
0: 85 ff          test    edi,edi
2: 74 24          je      28 <max+0x28>
4: 8d 47 ff       lea     eax,[rdi-0x1]
7: 48 8d 4c 86 04 lea     rcx,[rsi+rax*4+0x4]
c: 31 c0          xor     eax,eax
e: 66 90          xchg    ax,ax
10: 8b 16          mov     edx,DWORD PTR [rsi]
12: 39 d0          cmp     eax,edx
14: 0f 42 c2       cmovb   eax,edx
17: 48 83 c6 04     add     rsi,0x4
1b: 48 39 ce       cmp     rsi,rcx
1e: 75 f0          jne     10 <max+0x10>
20: c3            ret
21: 0f 1f 80 00 00 00 00 nop     DWORD PTR [rax+0x0]
28: 31 c0          xor     eax,eax
2a: c3            ret
```

Let's Compare

Get the next element of the array (pointed to by **RSI**) and compare it to **EAX**. Note how few bytes are used.

0000000000000000 <max>:

0:	85 ff	test	edi,edi
2:	74 24	je	28 <max+0x28>
4:	8d 47 ff	lea	eax,[rdi-0x1]
7:	48 8d 4c 86 04	lea	rcx,[rsi+rax*4+0x4]
c:	31 c0	xor	eax,eax
e:	66 90	xchg	ax,ax
10:	8b 16	mov	edx,DWORD PTR [rsi]
12:	39 d0	cmp	eax,edx
14:	0f 42 c2	cmovb	eax,edx
17:	48 83 c6 04	add	rsi,0x4
1b:	48 39 ce	cmp	rsi,rcx
1e:	75 f0	jne	10 <max+0x10>
20:	c3	ret	
21:	0f 1f 80 00 00 00 00	nop	DWORD PTR [rax+0x0]
28:	31 c0	xor	eax,eax
2a:	c3	ret	

Let's Compare

The **CMOVx** family of instructions are *conditional moves*.

CMOVB is conditional move *below*. That is, if the prior compare showed that **EAX** was below (less than) **EDX**, then do the move.

There are a lot of these.

This eliminates our “if” branching logic.

```
0000000000000000 <max>:
0: 85 ff          test    edi,edi
2: 74 24          je      28 <max+0x28>
4: 8d 47 ff       lea     eax,[rdi-0x1]
7: 48 8d 4c 86 04 lea     rcx,[rsi+rax*4+0x4]
c: 31 c0          xor     eax,eax
e: 66 90          xchg    ax,ax
10: 8b 16          mov     edx,DWORD PTR [rsi]
12: 39 d0          cmp     eax,edx
14: 0f 42 c2       cmovb   eax,edx
17: 48 83 c6 04     add     rsi,0x4
1b: 48 39 ce       cmp     rsi,rcx
1e: 75 f0          jne     10 <max+0x10>
20: c3            ret
21: 0f 1f 80 00 00 00 00 nop     DWORD PTR [rax+0x0]
28: 31 c0          xor     eax,eax
2a: c3            ret
```

Let's Compare

If we switch our program to using **CMOVB**, we save only one byte (because the instruction is longer) but we eliminate a branch inside a loop, which can significantly improve performance.

```
0000000000000000 <max>:
0: 85 ff          test    edi,edi
2: 74 24          je      28 <max+0x28>
4: 8d 47 ff       lea     eax,[rdi-0x1]
7: 48 8d 4c 86 04 lea     rcx,[rsi+rax*4+0x4]
c: 31 c0          xor     eax,eax
e: 66 90          xchg    ax,ax
10: 8b 16          mov     edx,DWORD PTR [rsi]
12: 39 d0          cmp     eax,edx
14: 0f 42 c2       cmovb   eax,edx
17: 48 83 c6 04     add     rsi,0x4
1b: 48 39 ce       cmp     rsi,rcx
1e: 75 f0          jne     10 <max+0x10>
20: c3             ret
21: 0f 1f 80 00 00 00 00 nop     DWORD PTR [rax+0x0]
28: 31 c0          xor     eax,eax
2a: c3             ret
```

Let's Compare

Now advance **RSI** to point to the next value in the array, and compare it to **RCX** to see if we are done. If not, branch backward.

Check out the bytes for **jne 10**. Note that **0xf0** is **-0x10** in 8-bit two's complement, and the first address after the branch instruction is hex **0x20**, and **0x20-0x10 = 0x10**.

0x2b is $32+11 = 43$ bytes (but 9 are nops)

```
0000000000000000 <max>:
0: 85 ff          test    edi,edi
2: 74 24          je      28 <max+0x28>
4: 8d 47 ff       lea     eax,[rdi-0x1]
7: 48 8d 4c 86 04 lea     rcx,[rsi+rax*4+0x4]
c: 31 c0          xor     eax,eax
e: 66 90          xchg    ax,ax
10: 8b 16          mov     edx,DWORD PTR [rsi]
12: 39 d0          cmp     eax,edx
14: 0f 42 c2       cmovb   eax,edx
17: 48 83 c6 04     add     rsi,0x4
1b: 48 39 ce       cmp     rsi,rcx
1e: 75 f0          jne     10 <max+0x10>
20: c3            ret
21: 0f 1f 80 00 00 00 00 nop     DWORD PTR [rax+0x0]
28: 31 c0          xor     eax,eax
2a: c3            ret
```



A Better Solution

We can actually do better. There is a special instruction family: **LOOP**. These instructions decrement **ECX**, check for zero, and may check other flags. They were created to do looping.

All we have to do is run the loop backward, and modify the input arguments (which, after all, are not guaranteed to be preserved).

This reduces our program size to just 21 bytes. Does it matter? Sometimes it does. But this is also *faster*.

```
; nasm -f elf64 max.asm  
; gcc -static -o max max.o
```

```
global max
```

```
max:
```

```
xor rax, rax  
test edi, edi  
je out  
mov ecx, edi
```

```
loop:
```

```
mov edx, DWORD [rsi+rcx*4-4]  
cmp eax, edx  
cmovb eax, edx  
loop loop
```

```
out:
```

```
ret
```



A Better Solution

NASM has a nice macro facility. We can add a macro to reference an element of the array. This can make the code less error-prone.

We could also rename the registers... but this might be a bad idea because it could lead to confusion.

```
; nasm -f elf64 max.asm  
; gcc -static -o max max.o
```

```
global max
```

```
%define value(b) DWORD [rsi+(b)*4-4]
```

```
max:
```

```
    xor rax, rax  
    test edi, edi  
    je out  
    mov ecx, edi
```

```
loop:
```

```
    mov edx, value(rcx)  
    cmp eax, edx  
    cmovb eax, edx  
    loop loop
```

```
out:
```

```
    ret
```



What's the Point?

There are a *lot* of instructions, some added for specific cases.

An ordinary algorithm can start out pretty obvious... and rapidly become difficult to understand.

It is very easy to make hard-to-find errors in assembly.

Writing assembly will still often beat the compiler because you, the programmer, have information (the loop can run backwards) that the compiler doesn't.

Writing assembly can be *tough*.

If you really want to obfuscate a program, use convoluted assembly.

Calling Library Routines from Assembly



A More Complex Example

Let's write a stand-alone program to compute the square root of a number. We'll use the C libraries. We will also use the C runtime, so we need a `main` function.

- `strtod` - convert `char *` to `double`
- `sqrt` - compute the square root of a `double`
- `printf` - print the results
- `puts` - print a string without formatting

To start a stand-alone program we should define the `main` symbol as the entry point of our code.

To access the C functions, we use `extern`.



A More Complex Example

Command line arguments are just like those of `main`.

<code>int argc</code>	<code>RDI</code>
<code>char * argv[]</code>	<code>RSI</code> Entries are quad word pointers, so 8 bytes each with a terminating NULL pointer
<code>char * envp[]</code>	<code>RDX</code>

The program name is in `[RSI]`. The first argument (if any) is in `[RSI+8]`, etc.

Remember that `RDI` is one plus the number of arguments (since it counts the program name as the first).



A More Complex Example

Let's get started.

First let's write code to check the number of arguments and to make sure everything works like we would expect.

Compile with:

```
nasm -felf64 sqrt1.asm  
gcc -static -o sqrt1 sqrt1.o
```

(Instead of `-static`, try `-no-pie`.)

```
; nasm -f elf64 sqrt1.asm  
; gcc -static -o sqrt1 sqrt1.o
```

```
global main  
extern puts
```

```
section .text
```

```
main:
```

```
    cmp rdi, 2  
    je okay1  
    mov rdi, eargs  
    call puts  
    mov eax, 1  
    ret
```

```
okay1:
```

```
    mov rdi, [rsi+8]  
    call puts  
    mov rdi, endl  
    call puts  
    mov eax, 0  
    ret
```

```
section .data
```

```
eargs: db "ERROR: Expected exactly one argument.", 10, 0  
endl:  db 10,0
```



Aside: Sections and Segments

The assembly file we created had *sections*. Specifically, it mentioned two.

- `.text` contains unmodifiable executable instructions; accumulated into the code segment `CS`
- `.data` contains unmodifiable fixed data; accumulated into the data segment `DS`

If we wanted to have *modifiable* data, we could have included a `.bss` section, too. This is (historically) the *block started by symbol* area of fixed-size, uninitialized, modifiable variables. This is adjacent to the data segment, and is also referenced via the `DS` segment register.

The page table can be set up to control read / write access to the pages where these reside, but you seldom need to worry about that.



A More Complex Example

Declare `main` and identify external functions.

```
global main  
extern printf  
extern strtod  
extern sqrt  
extern puts
```



A More Complex Example

Add a `.data` section at the end with some defined strings.

```
global main
extern printf
extern strtod
extern sqrt
extern puts

;...

section .data
eargs:db "ERROR: Expected exactly one argument.",10,0
endl: db 10,0
fmt:  db "sqrt(%f) = %f",10,0
```



A More Complex Example

Add the `.text` section with the usual function boilerplate.

This is a good idea whether or not the stack is actually used. You can optimize it later; but don't *prematurely* optimize!

The `leave` instruction is exactly the same as:

```
mov rsp, rbp
pop rbp
```

```
section .text

main: push rbp
      mov rbp, rsp

      ; ...

done: leave
      ret
```




Aside: Local Variables and the Stack

Since we will be calling other functions, we need to worry about `EDI` and `RSI`. We can save these on the stack. We also need a local variable to hold the floating point value of the argument.

The stack grows "backward" in memory; as you push items, the stack pointer decreases. We can reserve memory on the stack by subtracting the amount we need from the current stack pointer.

We want to store `EDI` (4 bytes) and `RSI` (8 bytes). We also want to store another quad word (8 bytes). Thus we need 20 bytes: `sub rsp, 20`

- `EDI` will be stored at `[RBP-4]`, so the four bytes go in `[RBP-4]`, `[RBP-3]`, `[RBP-2]`, and `[RBP-1]`.
- `RSI` will be stored at `[RBP-12]`
- The local variable will be at `[RBP-20]`



Aside: Local Variables and the Stack

The stack should always be aligned on a 16-byte boundary before a function call. If you do not do this, and the function uses **XMM** registers, you *will* get a segfault. (You can get away with not aligning if you don't... maybe.)

It is the caller's job to make sure the stack is aligned. The stack is aligned on a 16-byte boundary before a call.

- The function is called. The return address (8 bytes) is pushed onto the stack. *The stack is no longer aligned.*
- The first thing most functions do is **push rbp** (8 bytes). The stack is, again, aligned.

If we want to reserve space, we have to make sure the stack is aligned. For 20 bytes we can do this two ways.

Round up to the next multiple of 16:

```
sub rsp, 32
```

Clear the bottom four bits of the pointer:

```
sub rsp, 20
```

```
and rsp, -16
```



A More Complex Example

Reserve space on the stack and store **EDI** and **RSI**.

```
section .text

main: push rbp
      mov rbp, rsp

      sub rsp, 32
      mov DWORD [rbp-4], edi
      mov DWORD [rbp-12], rsi

      ; ...

done: leave
      ret
```



A More Complex Example

Let's make sure we got one command line argument (well, actually two: the program name and the first argument).

If we did not, we print an error message and set **EAX** (the return value) to a non-zero value to indicate an error.

```
section .text

main: push rbp
      mov rbp, rsp

      sub rsp, 32
      mov DWORD [rbp-4], edi
      mov DWORD [rbp-12], rsi

      cmp edi, 2
      je okay1
      mov rdi, eargs
      call puts
      mov eax, 1
      jmp done

okay1:

      ; ...

done: leave
      ret
```



A More Complex Example

We use `strtod` to convert a text string into a `double` value. This takes two arguments:

- The string to convert
- A pointer-pointer, or `NULL`

We need the pointer to the string to be in `RDI`, we need `RSI` to be 0, and we will get the (floating point) result in `XMM0`, which we store in the reserved space on the stack.

The `movsd` instruction moves a `double` value.

```
    ; ...  
  
okay1:  
  
    mov rdi, QWORD [rbp-12] ; argv  
    mov rdi, QWORD [rdi+8]  ; argv[1]  
    mov rsi, 0  
    call strtod  
    movsd QWORD [rbp-20], xmm0  
  
    ; ...  
  
done: leave  
      ret  
  
    ; ...
```



A More Complex Example

Next we need to take the square root.

We need the first (floating point) argument to be in `XMM0`, and the return value will be in `XMM0`, which is why we had to save on the stack earlier.

We can just call `sqrt`.

```
    ; ...  
  
okay1:  
  
    mov rdi, QWORD [rbp-12] ; argv  
    mov rdi, QWORD [rdi+8]  ; argv[1]  
    mov rsi, 0  
    call strtod  
    movsd QWORD [rbp-20], xmm0  
  
    call sqrt  
  
    ; ...  
  
done: leave  
    ret  
  
    ; ...
```



A More Complex Example

Now we need to print the result.

We need the original value to be in `XMM0`, and the square root to be in `XMM1`.

We put the format string into `RDI`. `EAX` holds the number of floating point arguments being provided (two in this case).

We put zero in `EAX` (the return value) and fall-through to `done`.

```
; ...
```

```
okay1:
```

```
mov rdi, QWORD [rbp-12] ; argv
mov rdi, QWORD [rdi+8]  ; argv[1]
mov rsi, 0
call strtod
movsd QWORD [rbp-20], xmm0
```

```
call sqrt
```

```
movsd xmm1, xmm0
movsd xmm0, QWORD [rbp-20]
mov rdi, fmt
mov eax, 2
call printf
mov eax, 0
```

```
done: leave
      ret
```

```
; ...
```

Homework

Due: Tuesday, 4 February



Homework: Several Square Roots

Modify the square root program to compute and print the square root of all arguments to the program. There may be zero arguments. You can use [argc](#), or you can watch for the [NULL](#) pointer in [argv](#) to iterate through the list, but watch out for registers being clobbered by the functions you call! Call your program [sqrt_list](#).

```
$ sqrt_list
$ sqrt_list 16 9 65536 2
sqrt(16.000000) = 4.000000
sqrt(9.000000) = 3.000000
sqrt(65536.000000) = 256.000000
sqrt(2.000000) = 1.414214
```

Next time:
Inline assembly