Floating point - summary 1

- **More in section**
- Numbers are represented as [Mantissa]*(2**[Exponent])
  - IEEE 754
    - Mantissa is *normalized* sign/magnitude; normalization means the number always has a leading 1 (e.g. 1.00101) and that leading 1 is dropped.
    - exponent uses some crazy base format (value = exponent - base).
      - Exponents at the extent of the range (0x0...0 and 0xf...f) are special and represent unusual numbers:
        - Sign=0, Exp=0, Significand=0 = +0
        - Sign=1, Exp=0, Significand=0 = -0
        - Sign=0, Exp=111..1, Significand=0 = +infinity
        - Sign=1, Exp=111..1, Significand=0 = -infinity
        - Sign=0/1, Exp=111..1, Significand = 1????? = “quiet” NaN
        - Sign=0/1, Exp=111..1, Significand = 0??1?? = “signaling” NaN
  - There *are other formats*. Most of these are internal to a processor, but not all.

Floating point - summary 2

- Your view as a software developer is typically:
  - float = 32 bit FP value
  - double = 64 bit FP value
  - **avoid:** long double = non-standard FP value. Varies between 64, 80 and 128 bits
- Unless you need IEEE 754 standard FP, then you get “whatever” FP
  - On Intel x86 machines this means that computations that never leave the processor are computed with greater precision than the values. e.g.:
    - float x = MAX_FLOAT, y = MAX_FLOAT, z; z = x * 2 - y;
      - IEEE 754: z = +infinity Intel: z = MAX_FLOAT or +infinity (depends).
  - Typically IEEE 754 is a tad slower because of all the corner case implementation details supported.
    - Where you care is at the edges, in particular how things round. Numerically stable algorithms are designed to work with the particular rounding modes IEEE FP provides.

Floating point - summary 3

- Advice #1: If you end up writing a lot of FP code, you probably should buy “Numerical Recipes in C”, which is an atrocious book where the implementations are badly reformatted FORTRAN code, but it is the defining text on this.
- Advice #2: If you are writing code like: if (a == b) and a and b are FP values then you probably have an error in your thinking. e.g.:
    - double a = 1.0 / 3.0, b; b = a * 3.0; if (a == 1.0) { } // BROKEN
    - Consider instead: if (is_close(a, b, epsilon)) { }

Checkpoint

- So far in class we have:
    - Provided a broad overview
    - Focused a lot on **data representation**
        - Dwelled extensively on integers (2’s complement)
        - Briefly mentioned how bits are mapped to characters (ASCII, Unicode)
        - Discussed how strings are stored in C and alternative approaches
        - Did a whirlwind tour of fixed and floating point
    - Labored over a few C eccentricities
        - pointers
        - bit manipulations
- Things I hope you should be able to do by now:
    - Write the function int atoi(const char *s)
        - **This was Corensic’s standard interview question and by and large only 1/5th of the people we interviewed, representing 1/500th of the resumes we received can do this correctly.**

Up next: the HW/SW interface

Your view as the developer

```
int atoi(char *s) {
    int v = 0, sign=1;
    if (*s == '-') {
        sign=-1;
        ++s;
    }
    while (*s &&
        _is_number(*s)) {
        v = v * 10 +
            _ascii_to_digit(*s);
        ++s;
        }
    return sign * v;
}
```

gcc,
cl

A “human readable” view of the ISA

```
_atoi:
Leh_func_begin1:
    pushq   %rbp
Ltmp0:
    movq    %rsp, %rbp
Ltmp1:
    subq    $32, %rsp
Ltmp2:
    movq    %rdi, -8(%rbp)
    movl    $0, -20(%rbp)
    movl    $1, -24(%rbp)
    movq    -8(%rbp), %rax
    movb    (%rax), %al
    cmpb    $45, %al
    jne LBB1_2
    movl    $-1, -24(%rbp)
    movq    -8(%rbp), %rax
    movabsq $1, %rcx
```

gas,
MASM/
ml64
+
ld,
link

As the processor sees it

```
 55 48 89 e5 48 83 ec 20
48 89 7d f8 c7 45 ec 00
 00 00 00 c7 45 e8 01 00
00 00 48 8b 45 f8 8a 00
 3c 2d 75 1c c7 45 e8 ff  ff
ff ff 48 8b 45 f8 48
 b9 01 00 00 00 00 00 00
00 48 01 c8 48 89 45 f8
 eb 3c 8b 45 ec 6b c0 0a
48 8b 4d f8 8a 09 0f be
 c9 30 d2 89 cf 89 45 e4
88 d0 e8 00 00 00 00 89
 c1 8b 55 e4 01 ca 89 55
ec 48 8b 4d f8 48 ba 01
```

x86 / x64 ISA

- Why do we study x86 / x64 in this class?
  - Like it or not, it is the dominant desktop/server/laptop architecture
  - It is not simple. It is burdened by legacy:
    - x64 (64 bit) is based on x86 (32 bit) which is based on x86 (16 bit) which was designed to supplant the 8080 (8 bit).
      - 8080 is not 8086, but lives on! Many a microwave, thermostat and other tiny computer uses this ISA that dates from 1974!
    - **To this day,** 64 bit chips from Intel/AMD start out in an 8086 compatibility mode (euphemistically called “real mode”)
    - There is also an orphaned offshoot (the 80286) which is a 16 bit “protected mode” 8086 that is **still supported**.

x86 / x64 ISA in the market place

- AMD and Intel have a curious history
  - In the 80’s and 90’s there were a few “clone” CPU vendors, AMD, Cyrix, Transmeta, Chips and Technologies, IBM (the only licensed clone)
  - AMD originally made parts that were ISA and pin-compatible replacements for Intel parts.
    - Massive lawsuits ensued.
    - Eventually AMD and Intel reached a cross-licensing de taunt through the 486 generation, at which point AMD and Intel started to go their separate ways
      - This means the “core” 32 bit x86 architecture is the same, and they vary along the edges: vector instruction set extensions, virtualization extensions, etc; and are no longer pin compatible.
    - In the late 90’s it was apparent to everyone x86 had to go 64 bits.
      - Intel developed their own ISA extension, IA-64 (otherwise known as Itanium) which didn’t look anything like x86/IA-32. Itanium chips could run IA-32 or IA-64
      - AMD went to MSFT and said “what do you want?”. Thus was born AMD64 (or x86-64 or just x64). IA-64 never caught on; 2003/04 Intel licensed x64 from AMD.
- AMD and Intel reached another de taunt recently (with a ~ $1B payout to AMD). But the companies continue to go their separate ways. Thus the “core ISA” x86 & x64 is **almost but not entirely** the same, the **extensions are not.**

Architecture v Microarchitecture

- Architecture or Microarchitecture?
  - Main memory?
  - Virtual memory?
  - TLB?
  - Registers?
  - Register usage?
  - Caches?
  - Instructions?

Architecture v Microarchitecture

- Architecture or Microarchitecture?
  - Main memory? Architecture
  - Virtual memory? Architecture
  - TLB? Microarchitecture
  - Registers? Architecture
  - Register usage?   Convention (mostly), Architecture (some)
  - Caches? Microarchitecture (more or less)
  - Instructions? Architecture

x64 ISA

- Two types of memory
  - Registers
    - Direct access for data: ADD %rax, %rdx // rdx = rdx + rax; rflags....
    - Indirect access for flags: CMP %rax, %rbx // rflags.zf = (rax == rbx), ...
  - Main memory
    - Directly accessed: MOV *%rdx, %rax // rax = memory[rdx]
    - Stack accessed: POP %rax // rax = memory[rsp]; rsp = rsp + 8
    - Generally speaking there are 3 regions of memory for your process: code, data and stack. But as previously discussed, there tends to be multiple disjoint code and data locations, and *each thread* has its own stack.

x64 ISA

- Three broad classes of instructions:
  - Moving data (mov *%rdx, %rax)
  - Computing on data (add %rax, %rdx)
  - Branching (CMP %rax, %rdx; JE location)
- On x86/x64 these classes are not disjoint, e.g.:
  - ADD *%rdx, %rax (rax = memory[rdx] + rax)
  - SUB %rdx, %rax; JLZ location (SUB sets the flags JLZ jumps on)
- There are more instructions than these classes:
  - Instructions to access the OS (e.g. INT and SYSCALL)
  - Instructions the OS uses to manipulate processes (e.g. lgdt)
  - Instructions the OS uses to access “miscellaneous potentially non standard junk” (e.g. wrmsr)
  - Instructions to access the performance monitoring hardware (e.g. rdtsc)
  - etc, etc, etc

x64 ISA

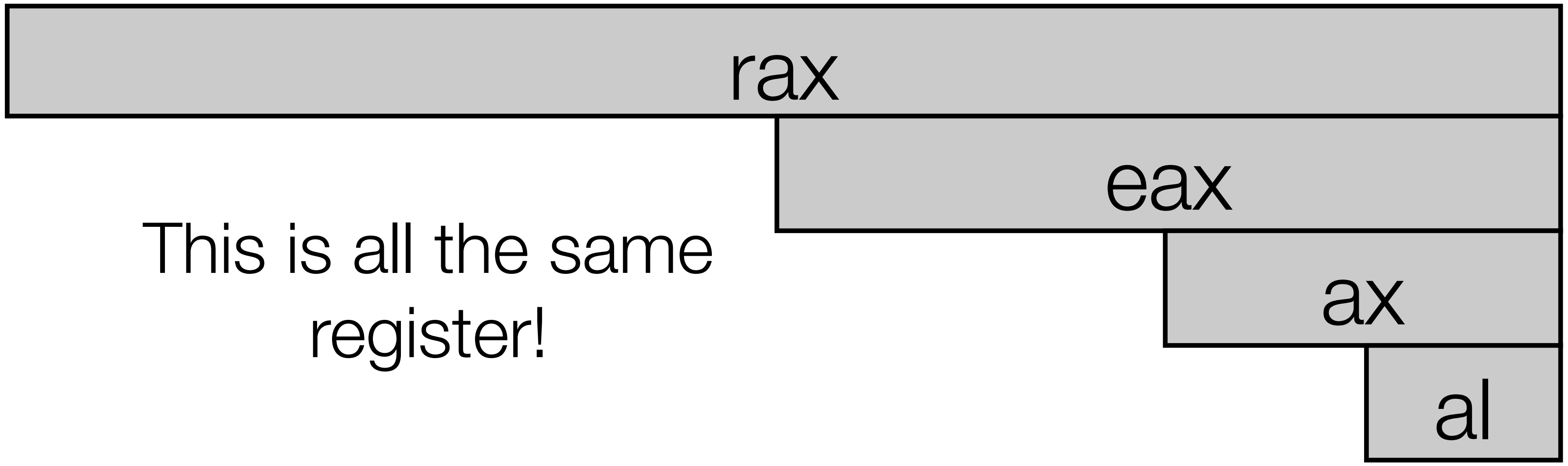
- **Almost true**: one only manipulates small pieces of data:
  - Integers, 1, 2, 4, 8 bytes
    - These data types are referred to as “b, s, l, and q” in gcc and “BYTE, WORD, DWORD, and QWORD” in MASM land.
    - E.g.: gcc: movq *%rdx, %rax
    - E.g.: MASM: MOV rax, QWORD PTR [rdx]
  - Floating point values, 32 and 64 bit values (and the non standard 80)
- x64/x86 supports numerous accessors that break this
  - x64 can do memcpy in 1 hardware instruction
  - x86/x64 supports “vectors of” integers
  - Certain OS instructions directly manipulate hardware tables

x64 ISA

16 registers:

rax, rcx, rbx, rdx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14

These registers are 64 bits wide, but it is possible to access smaller fields within them:



This is all the same register!

It is also possible to access other subfields (e.g. ah = top half of ax), but the need to do so is low and if you have to, you’ll have to look it up anyway :-)

Why 16 registers?

Not all registers are created equal...

- For some operations, RAX is an implicit destination register. For others, such as multiply, RDX:RAX is.
- RCX is often an implicit source/destination register meaning “count”
- RSP is an implicit source/destination register for “top of stack” which, by the way, is at the bottom of the stack in memory :-)
- RBX and RDX have more flexibility in the address computation department
- RSI is an implicit source/destination register meaning “source index”
- RDI is an implicit source/destination register meaning “destination index”

- **These are not conventions, these differences are baked into the ISA**

- Why does this exist?

x64 Register conventions

- **Conventions != ISA They are strongly worded suggestions**
- Why have conventions?
  - Codifies "best practices"
  - So that software from different vendors can interact
    - The conventions that most impact your life surround procedure call and system call invocations (but there are more!). We'll focus a lot on procedure calls in this class, and once you get that, the system call stuff will be trivial.
- x86 has no conventions -- period. Object code compiled with one vendor's compiler **cannot successfully call/link** with object code compiled with another vendors compiler.
  - Even code **from the same compiler** cannot link to itself if it is not compiled with the same flags! (e.g. __fastcall)

x64 Calling Conventions

- **x64 imposes 2 broad calling conventions**
  - One for Unix-based OS’s such as Linux, FreeBSD, Mac OS X, etc
  - One for Windows based OS’s
  - Why are there 2? I have no idea..
  - **To make this tractable, we are going to focus ONLY on Unix-based systems. But please please be aware that it is very different on Windows based ones. If you write code for that platform you will have to look it up. Search for “x64 API calling conventions” or go here http://x86-64.org/documentation/abi.pdf**
- What comprises a calling convention?
  - Passing arguments to the procedure
  - Obtaining the return value
  - Assurances about state that is preserved
  - Assurances about state that *may not be* preserved
  - Subtlies of stack usage
- **Warning:** Experience has shown this is a deceptively difficult topic. It is going to sound simple, but many many people go into the weeds here...

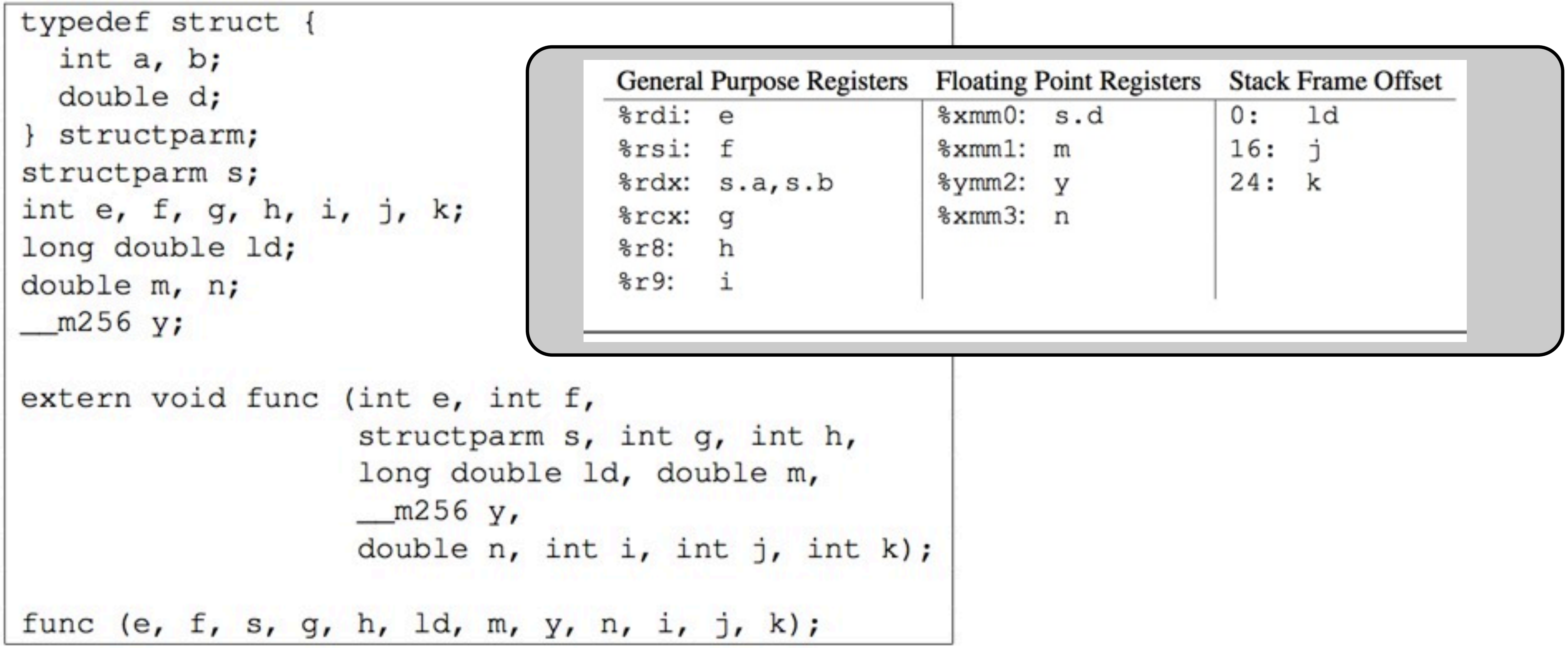
- PLEASE SIT CLOSE
- SKY DECIDED 1-5am WAS PLAY TIME :-)

x64 Calling conventions - Part 1

- The first 6 *integer* arguments to a function are passed through registers:
    - uint64_t foobar(uint64_t a1, uint64_t a2, uint64_t a3,
      uint64_t a4, uint64_t a5, uint64_t a6) { return 10; }
      x = foobar(1, 2, 3, 4, 5, 6);
        - When calling the function:
            - rdi = 1 rsi = 2 rdx = 3 rcx = 4 r8 = 5 r9 = 6
        - When returning a value from the function:
            - rax = 10
    - Even if the type is less than 64 bits:
        - char foobar(char x, int64_t y) { return ‘1’; } foobar(‘2’, -3);
        - edi = ‘2’ and rsi = -3
            - **Note:** notice how the “char” was extended to only 32 bits! This is a C thing. Not an x64 calling convention thing.

x64 - Calling Conventions - Part 2

More than 6 arguments can be passed through registers if they are of different types. The remainder need to go on the stack

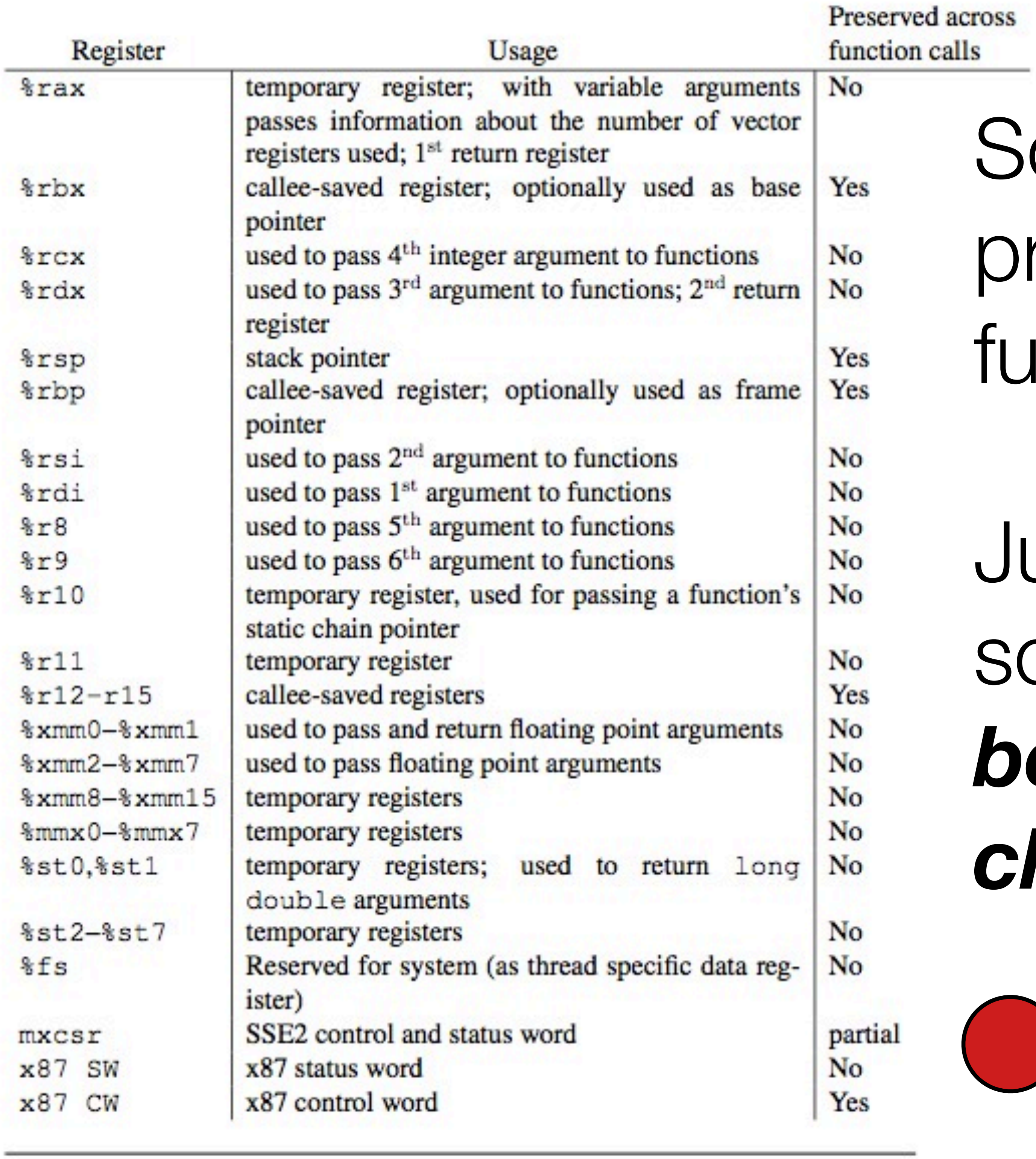


```
typedef struct {
  int a, b;
  double d;
} structparm;
structparm s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;

extern void func (int e, int f,
                  structparm s, int g, int h,
                  long double ld, double m,
                  __m256 y,
                  double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);
```

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: e	%xmm0: s.d	0: ld
%rsi: f	%xmm1: m	16: j
%rdx: s.a,s.b	%ymm2: y	24: k
%rcx: g	%xmm3: n	
%r8: h		
%r9: i		



Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
`%rbx`	callee-saved register; optionally used as base pointer	Yes
`%rcx`	used to pass 4th integer argument to functions	No
`%rdx`	used to pass 3rd argument to functions; 2nd return register	No
`%rsp`	stack pointer	Yes
`%rbp`	callee-saved register; optionally used as frame pointer	Yes
`%rsi`	used to pass 2nd argument to functions	No
`%rdi`	used to pass 1st argument to functions	No
`%r8`	used to pass 5th argument to functions	No
`%r9`	used to pass 6th argument to functions	No
`%r10`	temporary register, used for passing a function's static chain pointer	No
`%r11`	temporary register	No
`%r12-r15`	callee-saved registers	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
`%mmx0–%mmx7`	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

Some registers *must* be preserved by the called function

Just as important, some registers ***must be assumed to be clobbered***.

WARNING: This is a huge source of confusion for people. Read that phrase in bold again slowly.

x64 Calling Conventions - Example

```
uint64_t add2(uint64_t x, uint64_t y) {
    return x + y;
}

.text
.globl add2

add2:
   pushq %rbp
   movq %rsp, %rbp
   movq %rdi, %rax
   addq %rsi, %rax
   popq %bp
   ret
```

This is a called a ***pseudo directive***. *“.text” is the Unix way of saying “this is code”*

The “.globl” pseudo directive (yes it is really spelled that way) is a way of telling the assembler to mark the symbol as “global” for the purposes of linking. This means the symbol will be visible outside of the current object file.

x64 Calling Conventions - Example

```
uint64_t add2(uint64_t x, uint64_t y) {
    return x + y;
}

.text
.globl _add2

_add2:
   pushq %rbp
   movq %rsp, %rbp
   movq %rdi, %rax
   addq %rsi, %rax
   popq %bp
   ret
```

Not all systems (Linux does not for now), Mac OS X does. Windows does. Etc., prefix C symbols with another symbol.

They do this to avoid name collisions. Historically the prefix character has been “_” but some systems have used “.”, or “__” and others have used “$” either at the beginning or the end.

The “reality on the ground” as a software developer is you just need to figure out what your tool chain does and do that. **There is no standard.**

x64 Calling Conventions - Example

```
uint64_t add2(uint64_t x, uint64_t y) {
    return x + y;
}
```

```
.text
.globl add2

add2:
   pushq %rbp
   movq %rsp, %rbp
   movq %rdi, %rax
   addq %rsi, %rax
   popq %rbp
   ret
```

The pushq/popq that surround this function are there to maintain the “frame pointer”. A Frame pointer is **not part of the calling conventions**, but rather it is a feature of the **runtime environment**. It can be changed by compiler flags, but on Mac OS X & Linux systems a frame pointer is used by default (although on Linux systems it can get optimized away). Inside the Windows 64 kernel no frame pointer is used. Inside Windows user land, a frame pointer is used for native code.

A frame pointer aids the debugger (it facilities easier stack unwinding), and it can make assembly programming easier.

x64 Calling Conventions - Example

```
uint64_t add2(uint64_t x, uint64_t y) {
    return x + y;
}

.text
.globl add2

add2:
   pushq %rbp
   movq %rsp, %rbp
   movq %rdi, %rax
   addq %rsi, %rax
   popq %rbp
   ret
```

```
uint64_t test() {
    return add2(3, 4);
}
.text
.globl test

test:
   pushq %rbp
   movq %rsp, %rbp
   movl $3, %edi
   movl $4, %esi
   call add2
   popq %rbp
   ret
```

x64 Calling Conventions - Example

```
uint64_t test(uint64_t r) {
    return add2(3, 4) + r;
}
        .text
        .globl test
        test:
           pushq %rbp
           pushq %rdi
           movl $3 %edi
           movl $4 %esi
           subq $8, %rsp
           call add2
           add $8 %rsp
           popq %rdi
           add %rdi, %rax
           popq %rbp
           ret
```

What is going on here?

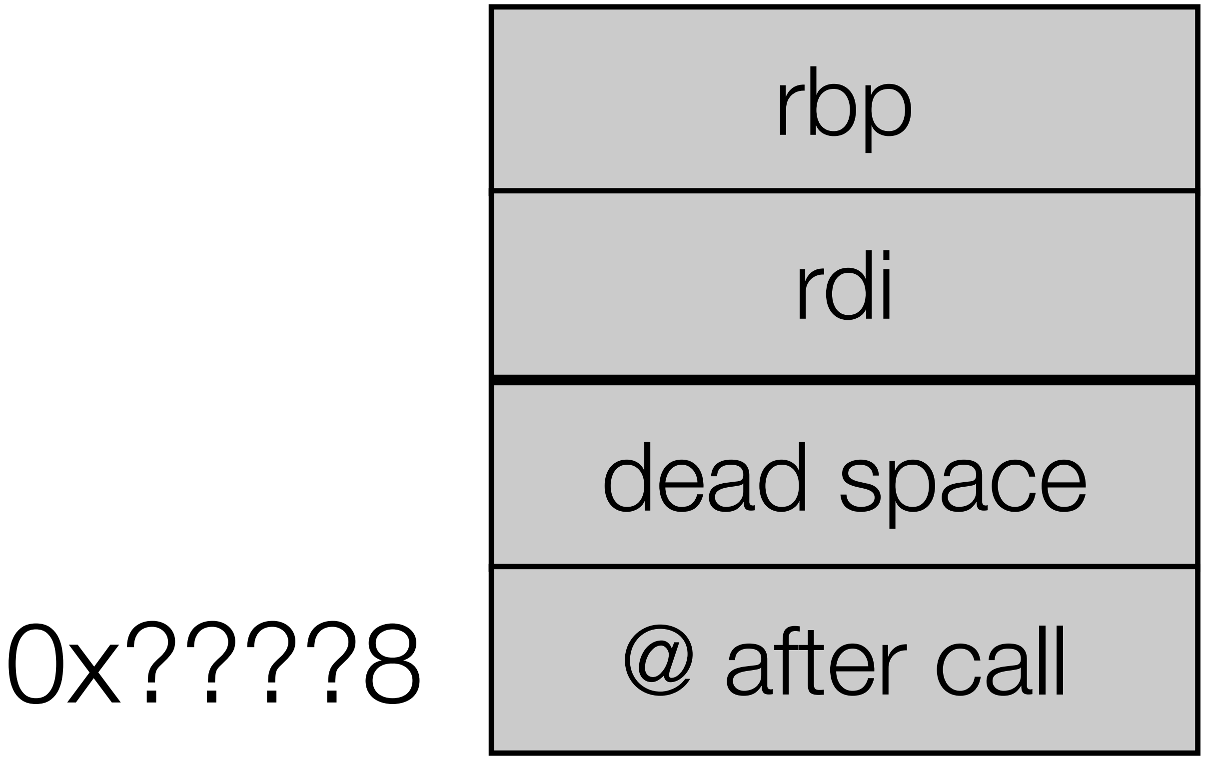
x64 Calling Conventions - Example

```
uint64_t test(uint64_t r) {
    return add2(3, 4) + r;
}
        .text
        .globl test
        test:
           pushq %rbp
           pushq %rdi
           movl $3 %edi
           movl $4 %esi
           subq $8, %rsp
           call add2
           add $8 %rsp
           popq %rdi
           add %rdi, %rax
           popq %rbp
           ret
```

On x64 there is an expectation that the stack pointer is a multiple of 16, plus 8, on entry to a function.
i.e., (%rsp + 8) mod 16 = 0

x64 Calling Conventions - Example

```
uint64_t test(uint64_t r) {
    return add2(3, 4) + r;
}
        .text
        .globl test
        test:
           pushq %rbp
           pushq %rdi
           movl $3 %edi
           movl $4 %esi
           subq $8, %rsp
           call add2
           add $8 %rsp
           popq %rdi
           add %rdi, %rax
           popq %rbp
           ret
```



Checkpoint!

- Memory, registers
- Intro to basic ops: Memory interfacing, arithmetic, control
- Dwelling on calling conventions
- Up next: More examples that become progressively more complex

Feels like it’s about time for a midterm doesn’t it?

How about Feb 10th.

Topics: data representation, assembly, and limited C programming

A slightly more complex example

```
#include <stdio.h>
#include <inttypes.h>

void foobar(uint64_t x, uint64_t y) {
    printf("The sum of x and y is %lld\n", x + y);
}
```

gcc -O3 -S t.c

```
        .file   "t.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string         "The sum of x and y is %lld\n"
        .text
        .p2align 4,,15
        .globl foobar
        .type  foobar, @function
foobar:
.LFB15:
        .cfi_startproc
        addq   %rdi, %rsi
        xorl   %eax, %eax
        movl   $.LC0, %edi
        jmp    printf
        .cfi_endproc
.LFE15:
        .size  foobar, .-foobar
        .ident "GCC: (GNU) 4.6.1 20110908 (Red Hat 4.6.1-9)"
        .section       .note.GNU-stack,"",@progbits
```

A slightly more complex example

```
#include <stdio.h>
#include <inttypes.h>

void foobar(uint64_t x, uint64_t y) {
    printf("The sum of x and y is %lld\n", x + y);
}
```

gcc -O3 -S t.c

```
        .file   "t.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string         "The sum of x and y is %lld\n"
        .text
        .p2align 4,,15
        .globl foobar
        .type  foobar, @function
foobar:
.LFB15:
        .cfi_startproc
        addq %rdi, %rsi
        xorl   %eax, %eax
        movl  $.LC0, %edi
        jmp   printf
        .cfi_endproc
.LFE15:
        .size  foobar, .-foobar
        .ident "GCC: (GNU) 4.6.1 20110908 (Red Hat 4.6.1-9)"
        .section        .note.GNU-stack,"",@progbits
```

```
        .section        .rodata
format_string:
        .string         "The sum of x and y is %lld\n"

        .text
        .globl foobar

foobar:
        pushq           %rbp
        movq            %rsp, %rbp
        addq            %rdi, %rsi
        movl            $format_string, %edi
        call            printf
        pop             %rbp
        ret
```

A slightly more complex example

```
#include <stdio.h>
#include <inttypes.h>

void foobar(uint64_t x, uint64_t y) {
    printf("The sum of x and y is %lld\n", x + y);
}
```

```
        .section .rodata
format_string:
        .string   "The sum of x and y is %lld\n"

        .text
        .globl    foobar

foobar:
        pushq     %rbp
        movq      %rsp, %rbp
        addq      %rdi, %rsi
        movl      $format_string, %edi
        call      printf
        pop       %rbp
        ret
```

Indicates “read only data” segment.

A slightly more complex example

```c
#include <stdio.h>
#include <inttypes.h>

void foobar(uint64_t x, uint64_t y) {
    printf("The sum of x and y is %lld\n", x + y);
}
```

```
        .section .rodata
format_string:
        .string "The sum of x and y is %lld\n"

        .text
        .globl    foobar

foobar:
        pushq    %rbp
        movq     %rsp, %rbp
        addq     %rdi, %rsi
        movl     $format_string, %edi
        call     printf
        pop      %rbp
        ret
```

Indicates a NULL terminated ASCII string. Note that on Mac OS X this directive is called .asciz

```c
#include <stdio.h>
#include <inttypes.h>

void foobar(uint64_t x, uint64_t y) {
    printf("The sum of x and y is %lld\n", x + y);
}
```

A slightly more complex example

```
	.section .rodata
format_string:
	.string	"The sum of x and y is %lld\n"

	.text
	.globl	foobar

foobar:
	pushq	%rbp
	movq	%rsp, %rbp
	addq	%rdi, %rsi
	movl	$format_string, %edi
	call	printf
	pop	%rbp
	ret
```

Our old friends...

On Linux x64 systems .rodata is stored “low” in memory (just above the code). This means the address of items in it are < 32 bits. This means you can specify them directly.

This is not the case on Mac OS X and other Unixy systems.

Yet another example

```
.data
    .comm  my_array,128,32

.text
    .globl   initialize_array

initialize_array:
   push      %rbp
   mov       %rsp, %rbp
   movl      $0, %eax
LoopHere:
   movl      %eax, my_array(%rax)
   addq      $4, %rax
   cmpq      $512, %rax
   jne       LoopHere
   pop       %rbp
   ret
```

```
#include <stdio.h>
#include <inttypes.h>

int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
         my_array[i] = i;
}
```

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

Yet another example

```
.data
     .comm  my_array,128,32

.text
     .globl    initialize_array

initialize_array:
   push      %rbp
   mov       %rsp, %rbp
   movl      $0, %eax
LoopHere:
   movl      %eax, my_array(%rax)
   addq      $4, %rax
   cmpq      $512, %rax
   jne       LoopHere
   pop       %rbp
   ret
```

Indicates this stuff belongs in the “.data” or “.bss” segment. The .bss or “block started by symbol” in 1950’s parlance, is static data that starts out as 0. The “.data” segment is data that is initialized in some way.

.comm symbol, size, bits
Make space and put it in .bss

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

Yet another example

```
.data
my_array: .zero 512

.text
      .globl    initialize_array

initialize_array:
    push      %rbp
    mov       %rsp, %rbp
    movl      $0, %eax
LoopHere:
    movl      %eax, my_array(%rax)
    addq      $4, %rax
    cmpq      $512, %rax
    jne       LoopHere
    pop       %rbp
    ret
```

Indicates this stuff belongs in the “.data” or “.bss” segment. The .bss or “block started by symbol” in 1950’s parlance, is static data that starts out as 0. The “.data” segment is data that is initialized in some way.

Allocate it but place it in .data

```c
int my_array[128] = { 5 };

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

Yet another example

```
.data
my_array: .long 5
          .zero 508
.text
    .globl   initialize_array

initialize_array:
   push      %rbp
   mov       %rsp, %rbp
   movl      $0, %eax
LoopHere:
   movl      %eax, my_array(%rax)
   addq      $4, %rax
   cmpq      $512, %rax
   jne       LoopHere
   pop       %rbp
   ret
```

Indicates this stuff belongs in the “.data” or “.bss” segment. The .bss or “block started by symbol” in 1950’s parlance, is static data that starts out as 0. The “.data” segment is data that is initialized in some way.

Make my_array[0] = 5 and the rest 0.

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

Yet another example

```
.data
my_array: .zero 512

.text
     .globl    initialize_array

initialize_array:
    push      %rbp
    mov       %rsp, %rbp
    movl      $0, %eax
LoopHere:
    movl      %eax, my_array(%rax)
    addq      $4, %rax
    cmpq      $512, %rax
    jne       LoopHere
    pop       %rbp
    ret
```

Like read-only data, static data is “low” in memory and so is addressable directly.

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

Yet another example

```
.data
my_array: .zero 512

.text
      .globl    initialize_array

initialize_array:
    push      %rbp
    mov       %rsp, %rbp
    movl      $0, %eax
LoopHere:
    movl      %eax, my_array(%rax)
    addq      $4, %rax
    cmpq      $512, %rax
    jne       LoopHere
    pop       %rbp
    ret
```

Important: movl $0, %eax sign extends into the full 64 bit value of 0. This means %rax is equal to 0x0000000000000000 **NOT** 0x????????00000000

Yet another example

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

```
.data
my_array: .zero 512

.text
      .globl    initialize_array

initialize_array:
    push      %rbp
    mov       %rsp, %rbp
    movl      $0, %eax
LoopHere:
    movl      %eax, my_array(%rax)
    addq      $4, %rax
    cmpq      $512, %rax
    jne       LoopHere
    pop       %rbp
    ret
```

Note the use of a 32 bit value to be stored, **and**, a 64 bit address which is %rax + constant where the constant is the starting address of my_array

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                     my_array[i] = i;
}
```

Yet another example

```
.data
my_array: .zero 512

.text
      .globl    initialize_array

initialize_array:
    push      %rbp
    mov       %rsp, %rbp
    movl      $0, %eax
LoopHere:
    movl      %eax, my_array(%rax)
    addq      $4, %rax
    cmpq      $512, %rax
    jne       LoopHere
    pop       %rbp
    ret
```

Observe the 2 step

“if (rax != 512) goto LoopHere”

The cmpq sets bits in the rflags register. The JNE “jump not equal” either jumps to LoopHere if the previous comparison is “not equal” or falls through.

```
int my_array[128];

void initialize_array() {
    int i;
    for(i = 0; i < 128; i++)
                    my_array[i] = i;
}
```

Yet another example

```
.data
my_array: .zero 512

.text
    .globl   initialize_array

initialize_array:
   push      %rbp
   mov       %rsp, %rbp
   movl      $0, %eax
LoopHere:
   movl      %eax, my_array(%rax)
   addq      $4, %rax
   cmpq      $512, %rax
   jne       LoopHere
   pop       %rbp
   ret
```

Note that LoopHere is visible within the entire scope of the file, but not declared ".globl" so it is not visible outside the object file (the linker cannot see it). But heed the first part of that sentence closely. LoopHere is visible within the entire scope of the file...

Yet another yet another example

```
#include <stdio.h>
#include <inttypes.h>

void initialize_array(int *my_array) {
    int i;
    for(i = 0; i < 128; i++)
        my_array[i] = i;
}
```

Yet another yet another example

```
void initialize_array(int *my_array) {
    int i;
    for(i = 0; i < 128; i++)
            my_array[i] = i;
}
```

```
    .text
    .globl initialize_array

initialize_array:
    push    %rbp
    mov     %rsp, %rbp
    movl    $0, %eax
LoopHere:
    movl    %eax, (%rdi,%rax,4)
    addq     $1, %rax
    cmpq     $128, %rax
    jne       LoopHere
    pop      %rbp
    ret
```

Wowa! What is this?

(%rdi, %rax, 4) means use the address %rdi + 4 * %rax

Why is it written in the strange (%rdi, %rax, 4) notation? I have no idea. MASM is much saner here, with [%rdi + 4* %rax].

NOTE: Arbitrary multiplies are **NOT ALLOWED.** Only 2, 4, 8.

Yet another yet another example

```
void initialize_array(int *my_array) {
    int i;
    for(i = 0; i < 128; i++)
            my_array[i] = i;
}
```

```
    .text
    .globl initialize_array

initialize_array:
    push    %rbp
    mov     %rsp, %rbp
    movl    $0, %eax
LoopHere:
    movl    %eax, (%rdi,%rax,4)
    addq     $1, %rax
    cmpq     $128, %rax
    jne      LoopHere
    pop     %rbp
    ret
```

Since we are multiplying %rax by 4 on the address calculation, we count up to 128 here, not 512.

Checkpoint!

- By now you should be able to write simple functions in x64 assembly that are callable by C code. In increasing order of complexity, try and write the following functions on your own time. I **highly recommend you do this**:
    - int strlen(char *s)
    - void memcpy(void *dest, void *src, int length)
    - int strcpy(char *dest, char *src)
    - int atoi(char *s)

- Up next: broad overview of instructions
- After that: what would Brian Boitano do? i.e., understanding how gcc does it.

A few arithmetic instructions

Format		Computation
`add`	Src,Dest	Dest = Dest + Src
`sub`	Src,Dest	Dest = Dest - Src
`imul`	Src,Dest	Dest = Dest * Src
`sal`	Src,Dest	Dest = Dest << Src
`sar`	Src,Dest	Dest = Dest >> Src
`shr`	Src,Dest	Dest = Dest >> Src
`xor`	Src,Dest	Dest = Dest ^ Src
`and`	Src,Dest	Dest = Dest & Src
`or`	Src,Dest	Dest = Dest | Src
`inc`	Dest	Dest = Dest + 1
`dec`	Dest	Dest = Dest - 1
`neg`	Dest	Dest = -Dest
`not`	Dest	Dest = ~Dest

Remember: the type has to be inferable or explicit (recommended). e.g. “addl” or “addq”, etc

Branch instructions

CF: Carry Flag
ZF: Zero Flag
SF: Sign Flag
OF: Overflow Flag

cmp - subtract
test - and

jX	Condition	Description
`jmp`	`1`	**Unconditional**
`je`	`ZF`	**Equal / Zero**
`jne`	`~ZF`	**Not Equal / Not Zero**
`js`	`SF`	**Negative**
`jns`	`~SF`	**Nonnegative**
`jg`	`~(SF^OF)&~ZF`	**Greater (Signed)**
`jge`	`~(SF^OF)`	**Greater or Equal (Signed)**
`jl`	`(SF^OF)`	**Less (Signed)**
`jle`	`(SF^OF)|ZF`	**Less or Equal (Signed)**
`ja`	`~CF&~ZF`	**Above (unsigned)**
`jb`	`CF`	**Below (unsigned)**

Jump targets

- here:
  jmp here   # ordinary jumps to labels
  - Note that in x86 (32 bit) this is encoded to the hardware as jmp #constant. For x64 (64 bit) if the #constant is small, such as on Linux, then it can be encoded directly like that. But if the constant is large, as it would be on Mac OS X, then it is encoded as “rip relative”. For the most part, you do not have to think about this as a software developer unless you end up writing a hypervisor, assembler, JIT, etc, etc.
- jmp %rax   # jump to the address specified in the register
- jmp *%rax   # jump to the address stored at the memory
  location pointed to by %rax
- Similar target labels work for CALL/RET

Memory addressing

- **Most General Form**

D(Rb,Ri,S)	**Mem[Rb+S*Ri+ D]**

  - D: Constant “displacement” 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
    - Unlikely you’d use `%rbp`, either
  - S: Scale: 1, 2, 4, or 8

- **Special Cases**

(Rb,Ri)	**Mem[Rb+Ri]**
D(Rb,Ri)	**Mem[Rb+Ri+D]**
(Rb,Ri,S)	**Mem[Rb+S*Ri]**

Syntax:

in gcc *%rax is the same as (%rax)

in MASM must do [rax]

There is an instruction, “lea” that does everything but the load. It stands for “load effective address”. Why do you think such an instruction exists?

Today

- “finish” off assembly 101 (but don’t worry, it will return, oh yes it will..)
  - first we’ll detour through sections some more...
- move on to C
  - The wonderful world of the preprocessor
  - Hello world!

How does gcc “write” assembly?

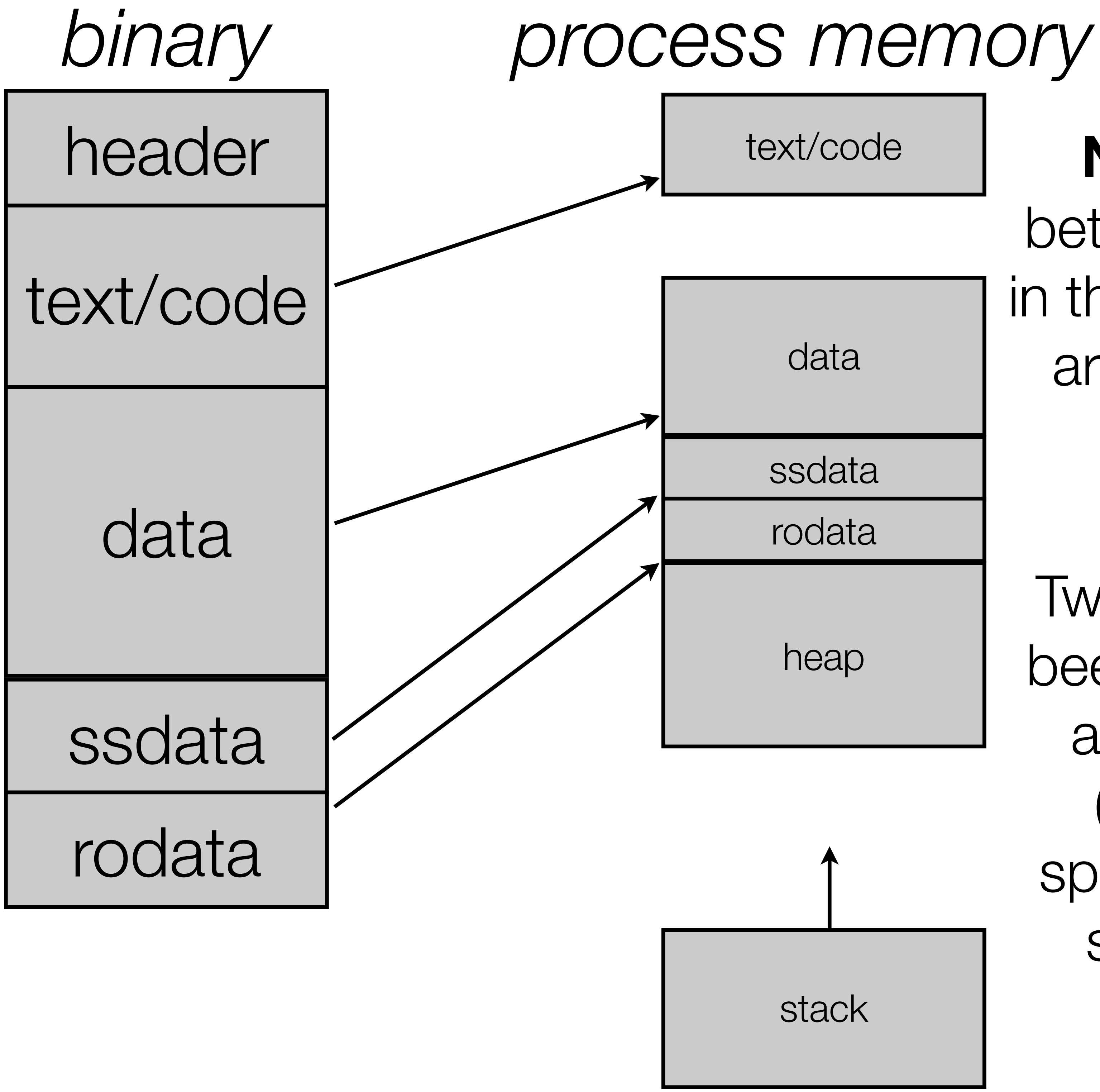
Don’t be ashamed to write things in C and do “gcc -S file.c” to generate a file.s output! Note that “gcc -S file.c” is very “chatty” assembly because it is not optimized. I recommend “gcc -O -S file.c”. And for a good time, try “gcc -O3 -S file.c”

Don’t do this for the homework and *just* turn in the output. But if you are stuck or just need to learn something about the runtime environment, by all means, go for it!

How does gcc “write” assembly?

- Before delving into this, lets first digress a little and discuss what the runtime environment is like in a C program.
- On Unix systems binary images are stored in ELF (newer) or COFF (older) format. A binary is just a file, like any other file (image, text document, etc). Except that it is formatted in such a way that the operating system knows how to load it into a process space and get it started. Binary files are made up of a header, and several “sections”
  - .text - code
  - .bss - block started by symbol (data that is initialized to 0)
  - .data - data
  - .rodata - read only data
  - And many many more. In fact, you can even stick your own in there if you like (and this is a very very useful programming trick. Search for “linker sets”
  - Typically there are also sections to store debug information.
- Similar (but of course, different) things occur on Windows platforms.

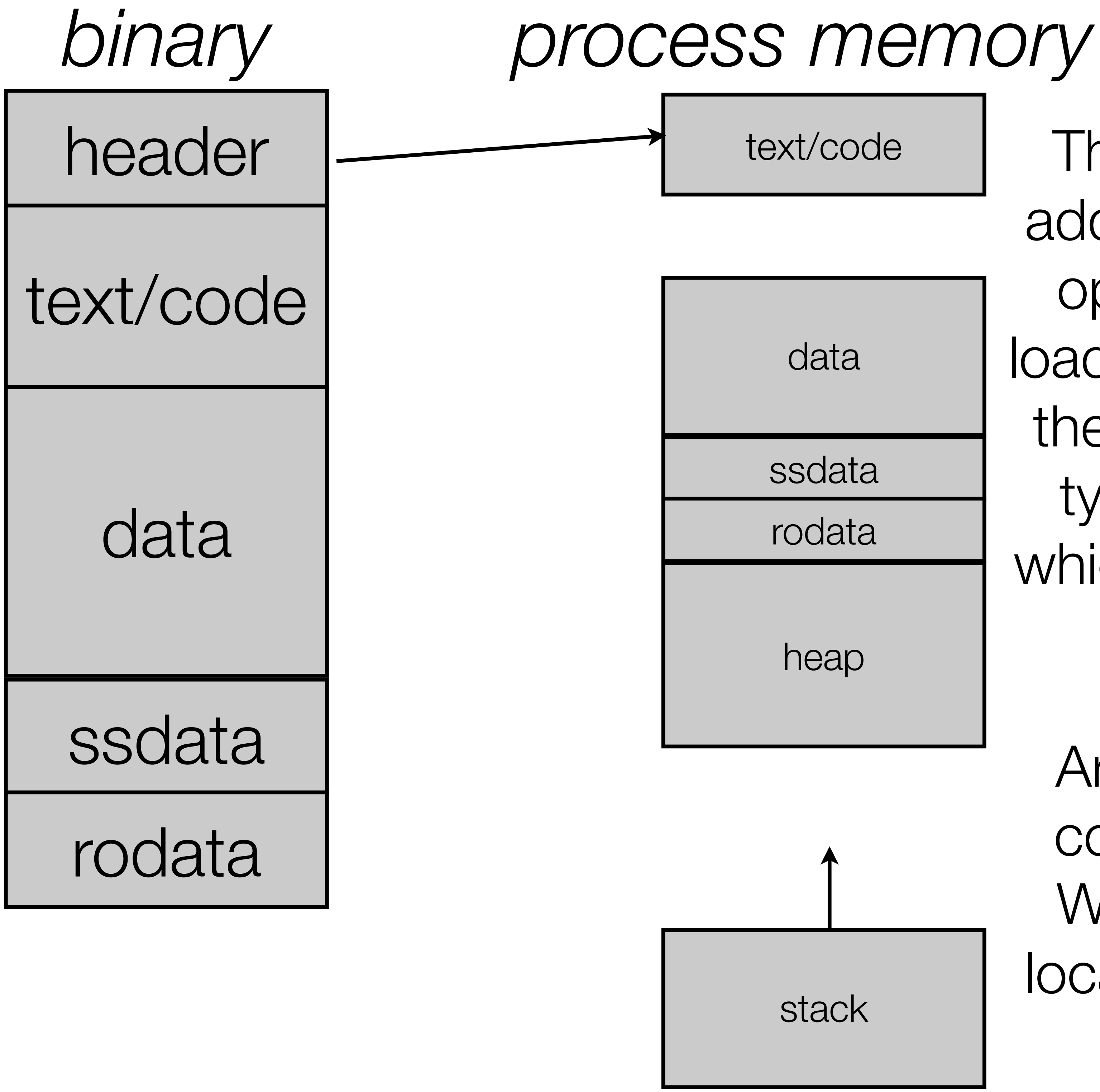
Loading a binary into memory



Note: there is no relationship between the order of the sections in the binary image and where they are located in process memory.

Two new memory locations have been added, a heap (typically just above the data) and the stack (typically at the “top” of user space). The heap grows up, the stack grows down, until they collide!

Loading a binary into memory



The binary also contains a “start address”, which is the location the operating system jumps to after loading the image. On Unix system the default start symbol is “_start” typically located in crt0.o or libc which are linked into your program.

Arguments passed to execv are copied into the top of the stack. When the OS jumps to the start location it also has set rsp to point here.

The libc ecosystem

- C programs typically execute with a library “libc” that provides some useful functionality, including file and console I/O (e.g. printf), and memory management (malloc/free).
- This functionality is initialized *before your main(...) function is invoked.* This is why C programs typically start at _start and not main.
- For 99.9% of C programing you do not have to think about this.
  - But it is very helpful to know that the world works this way.
  - The Linux kernel is just an ELF binary like any other.
    - But it doesn’t have a _start like any other!
  - The hypervisor my old company wrote also was an ELF binary.
    - Even on Windows! We ported the BSD ELF loading code to load our HV
    - Again, just as with Linux, we have to use a custom startup. No libc.
- Eventually _start invokes main(...), and from then on your code executes.

gcc -O4 -S

```
#define ARRAY_LENGTH (128)

static void foo(int *array, int length) {
        int i;
        for(i = 0; i < length; i++) {
                array[i] = i;
        }
}

int main(int argc, char *argv[]) {
        int array[ARRAY_LENGTH];

        foo(array, ARRAY_LENGTH);
        return 0;
}
```

```
        .file   "test.c"
        .section    .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl      main
        .typemain, @function
main:
.LFB1:
        .cfi_startproc
        xorl   %eax, %eax
        ret
        .cfi_endproc
.LFE1:
        .size main, .-main
        .ident      "GCC: (GNU) 4.6.1 20110908 (Red Hat 4.6.1-9)"
        .section    .note.GNU-stack,"",@progbits
```

dude, where’s my function?

Looks like a perfect exam question!

gcc -O4 -S

```
	.file	"test.c"
	.text
	.p2align 4,,15
	.globl	foo
	.type	foo, @function
foo:
.LFB0:
	.cfi_startproc
	testl	%esi, %esi
	jle .L1
	movq	%rdi, %rcx
	movq	%rdi, %rdx
	andl	$15, %ecx
	shrq	$2, %rcx
	negq	%rcx
	andl	$3, %ecx
	cmpl	%esi, %ecx
	cmova	%esi, %ecx
	xorl	%eax, %eax
	testl	%ecx, %ecx
	mov %ecx, %r11d
	je	.L3
	.p2align 4,,10
	.p2align 3
.L4:
	movl	%eax, (%rdx)
	addl	$1, %eax
	addq	$4, %rdx
	cmpl	%ecx, %eax
	jb	.L4
	cmpl	%ecx, %esi
	je	.L13
.L3:
	movl	%esi, %r10d
	subl	%ecx, %r10d
	movl	%r10d, %r8d
	shrl	$2, %r8d
	leal	0(,%r8,4), %r9d
	testl	%r9d, %r9d
	je	.L5
	leal	1(%rax), %edx
	movl	%eax, -24(%rsp)
	leaq	(%rdi,%r11,4), %rcx
	movl	%edx, -20(%rsp)
	leal	2(%rax), %edx
	movd	-20(%rsp), %xmm2
	movl	%edx, -16(%rsp)
	leal	3(%rax), %edx
	movd	-16(%rsp), %xmm1
	movl	%edx, -12(%rsp)
	xorl	%edx, %edx
	movd	-12(%rsp), %xmm0
	 punpckldq	%xmm0, %xmm1
	movd	-24(%rsp), %xmm0
	punpckldq	%xmm2, %xmm0
	movdqa	.LC0(%rip), %xmm2
	punpcklqdq	%xmm1, %xmm0
	jmp .L6
	.p2align 4,,10
	.p2align 3
.L9:
	movdqa	%xmm1, %xmm0
.L6:
	movdqa	%xmm0, %xmm1
	addl	$1, %edx
	movdqa	%xmm0, (%rcx)
	addq	$16, %rcx
	cmpl	%r8d, %edx
	paddd	%xmm2, %xmm1
	jb	.L9
	addl	%r9d, %eax
	cmpl	%r9d, %r10d
	je	.L1
	.L5:
	movslq	%eax, %rdx
	leaq	(%rdi,%rdx,4), %r
	.p2align 4,,10
	.p2align 3
.L7:
	movl	%eax, (%rdx)
	addl	$1, %eax
	addq	$4, %rdx
	cmpl	%eax, %esi
	jg	.L7
.L1:
	rep
	ret
.L13:
	ret
	.cfi_endproc
.LFE0:
	.size	foo, .-foo
	.section	.text.startup,"
	.p2align 4,,15
	.globl	main
	.type	main, @function
main:
```

gcc -O4 -S

```
        .section    .text.startup,"ax",@progbits
    .p2align 4,,15
    .globl  main
    .type   main, @function
main:
.LFB1:
    .cfi_startproc
    xorl    %eax, %eax
    ret
    .cfi_endproc
```

Lessons

- Writing highly optimized assembly is hard work.
- Modern compilers are very good at it, often times better than humans
- Modern compilers also do many things that in 99.9% of the time are good for you, but in 0.1% of the time are not. Among the ones that will be the least helpful for your -S usage:
  - Dead code elimination
  - Loop unrolling
  - Loop invariant code motion
  - Code motion in general (instruction scheduling)
  - Function inlining

gcc -S

```
    .file   "test.c"
    .text
    .globl  foo
    .type   foo, @function
foo:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -24(%rbp)
    movl    %esi, -28(%rbp)
    movl    $0, -4(%rbp)
    jmp .L2
.L3:
    movl    -4(%rbp), %eax
    cltq
    salq    $2, %rax
    addq    -24(%rbp), %rax
    movl    -4(%rbp), %edx
    movl    %edx, (%rax)
    addl    $1, -4(%rbp)
.L2:
    movl    -4(%rbp), %eax
    cmpl    -28(%rbp), %eax
    jl  .L3
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
```

```
main:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $528, %rsp
    movl    %edi, -516(%rbp)
    movq    %rsi, -528(%rbp)
    leaq    -512(%rbp), %rax
    movl    $128, %esi
    movq    %rax, %rdi
    call    foo
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   main, .-main
    .ident  "GCC: (GNU) 4.6.1 20110908 (Red H
    .section    .note.GNU-stack,"",@progbits
```

gcc -S

```
	.file	"test.c"
	.text
	.globl	foo
	.type	foo, @function
foo:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -24(%rbp)
	movl	%esi, -28(%rbp)
	movl	$0, -4(%rbp)
	jmp	.L2
.L3:
	movl	-4(%rbp), %eax
	cltq
	salq	$2, %rax
	addq	-24(%rbp), %rax
	movl	-4(%rbp), %edx
	movl	%edx, (%rax)
	addl	$1, -4(%rbp)
.L2:
	movl	-4(%rbp), %eax
	cmpl	-28(%rbp), %eax
	jl	.L3
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
```

```
main:
.LFB1:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$528, %rsp
	movl	%edi, -516(%rbp)
	movq	%rsi, -528(%rbp)
	leaq	-512(%rbp), %rax
	movl	$128, %esi
	movq	%rax, %rdi
	call	foo
	movl	$0, %eax
	leave
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE1:
	.size	main, .-main
	.ident	"GCC: (GNU) 4.6.1 20110908 (Red H
	.section	.note.GNU-stack,"",@progbits
```

gcc -O1 -S

just about right for human understanding

```
    .file   "test.c"
    .text
    .globl  foo
    .type   foo, @function
foo:
.LFB0:
    .cfi_startproc
    testl   %esi, %esi
    jle .L1
    movl    $0, %eax
.L3:
    movl    %eax, (%rdi,%rax,4)
    addq    $1, %rax
    cmpl    %eax, %esi
    jg  .L3
.L1:
    rep
    ret
    .cfi_endproc
.LFE0:
    .size   foo, .-foo
    .globl  main
    .type   main, @function
main:
```

```
    .text
    .globl  foo
foo:
    testl   %esi, %esi
    jle .L1
    movl    $0, %eax
.L3:
    movl    %eax, (%rdi,%rax,4
    addq    $1, %rax
    cmpl    %eax, %esi
    jg  .L3
.L1:
    rep
    ret
    .globl  main
main:
    subq    $512, %rsp
    movl    $128, %esi
    movq    %rsp, %rdi
    call    foo
    movl    $0, %eax
    addq    $512, %rsp
    ret
```

gcc -O1 -S

```
    .text
    .globl  foo
foo:
    testl   %esi, %esi
    jle .L1
    movl    $0, %eax
.L3:
    movl    %eax, (%rdi,%rax,4)
    addq    $1, %rax
    cmpl    %eax, %esi
    jg  .L3
.L1:
    rep
    ret
    .globl  main
main:
    subq    $512, %rsp
    movl    $128, %esi
    movq    %rsp, %rdi
    call    foo
    movl    $0, %eax
    addq    $512, %rsp
    ret
```

huh? rep ret?