Computer Structure and Language

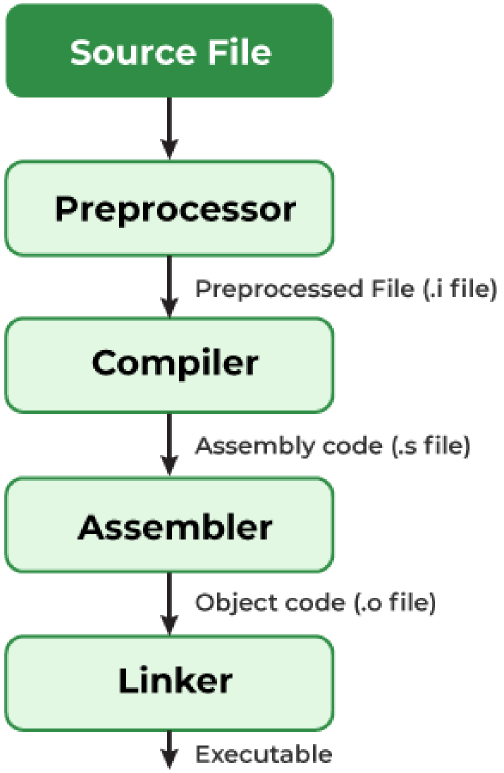
Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

Agenda

- From High level code to executable / compilation process
- Data Representation in Assembly / Registers, Memory & Variables
- Program Sections in Memory
- Implementing Logic in Assembly / What is different?
- Implementing If & Loop
- Talking About Stack
- Functions & Macros
- Writing Sample Programs

Compilation Process

- Compiling:
  - High Level Code to Assembly
- Assembling:
  - Assembly Code to Machine (Object) Code
  - No Address Translations
  - Some symbols remain unresolved.
  - Can not be executed.
- Linking:
  - Resolving symbols in object code & creating executables.



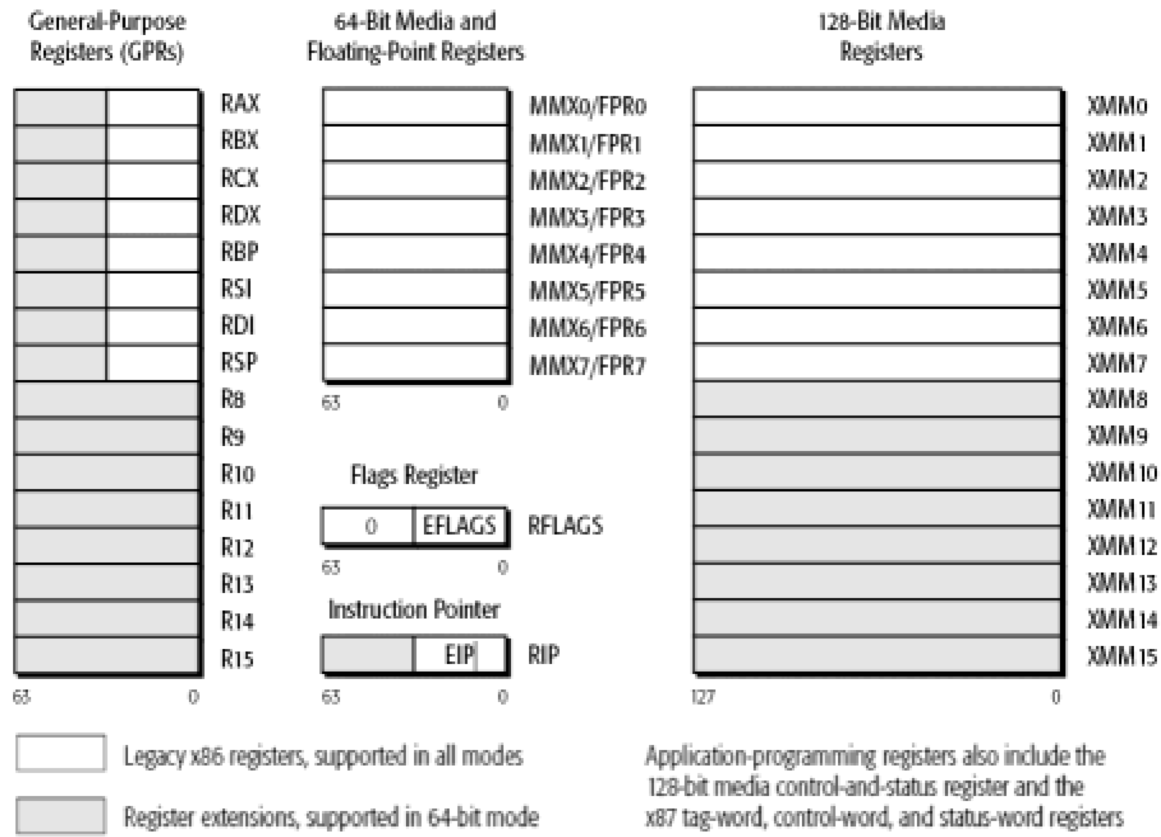
https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/

DATA REPRESENTATION

Registers

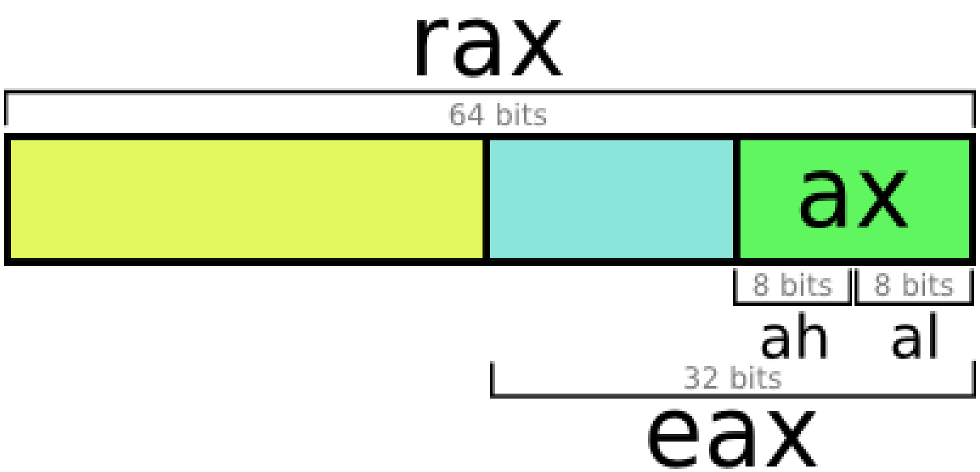
- High speed data stores built into the processor
- Limited in number, can not store all of the data required for the program
- Data has to move between them and memory
- Some of the registers have special purpose

Registers (x86_64)



https://pvs-studio.com/en/blog/posts/a0029/

Registers (x86_64)



https://nullprogram.com/blog/2015/05/15/

Memory – Data vs Address

```
segment .data

l1:      dd 1234
```

```
        mov eax, l1
        call print_int
        call print_nl

        mov eax, [l1]
        call print_int
        call print_nl
```

Memory – data segment

```
segment .data

l1: db 123                    ; one byte
l2: dw 1000                   ; one word - two bytes
l3: db 11010b
l4: db 12o
l6: dd 1A92h                  ; one double word - four bytes
l7: dd 0x1A92
l8: db 'A'
l9: db "AB"                   ; two bytes


; remember we specify size, not type!
```

Memory – data segment

```
segment .data

b:   db 1                     ; byte
w:   dw 1                     ; word - 2 bytes
d:   dd 1                     ; double word - 4 bytes
q:   dq 1                     ; quad word - 8 bytes
t:   dt 1                     ; 10 bytes

; (without initializing)
rb:  resb 4                   ; reserve 4 bytes
rw:  resw 2                   ; reserve 2 words
rd:  resd 5                   ; reserve 5 double words
rq:  resq 10                  ; reserver 10 quad words

id: dd 1, 2, 3, 4, 5, 6  ; 6 double words with values (1, 2, 4, 5, 6)
tb: times 9 db 1              ; 9 bytes all with value 1
```

Memory – address

```
; for specifying memory address
[reg1 + m*reg2 + offset]
; m can be 1, 2, 4 or 8
; offset has to be a literal
```

Memory – When size matters

```
        mov byte [l1], 5
        mov word [l2], 3
        inc dword [l3]
        add rax, qword [l4 + 4]

; We have to specify memory size for the operation
; To do this we use keywords: byte, word, dword, qword
```

Memory – Invalid Memory Operations

```
mov [l1], [l2]
add [l1], [l2]
sub [l1], [l2]
adc [l1], [l2]
sbb [l1], [l2]
cmp [l1], [l2]
and [l1], [l2]
or  [l1], [l2]
xor [l1], [l2]
; Only one operand can be Memory

mov [l1], 44
; Size is ambiguous
```

Memory – Extending

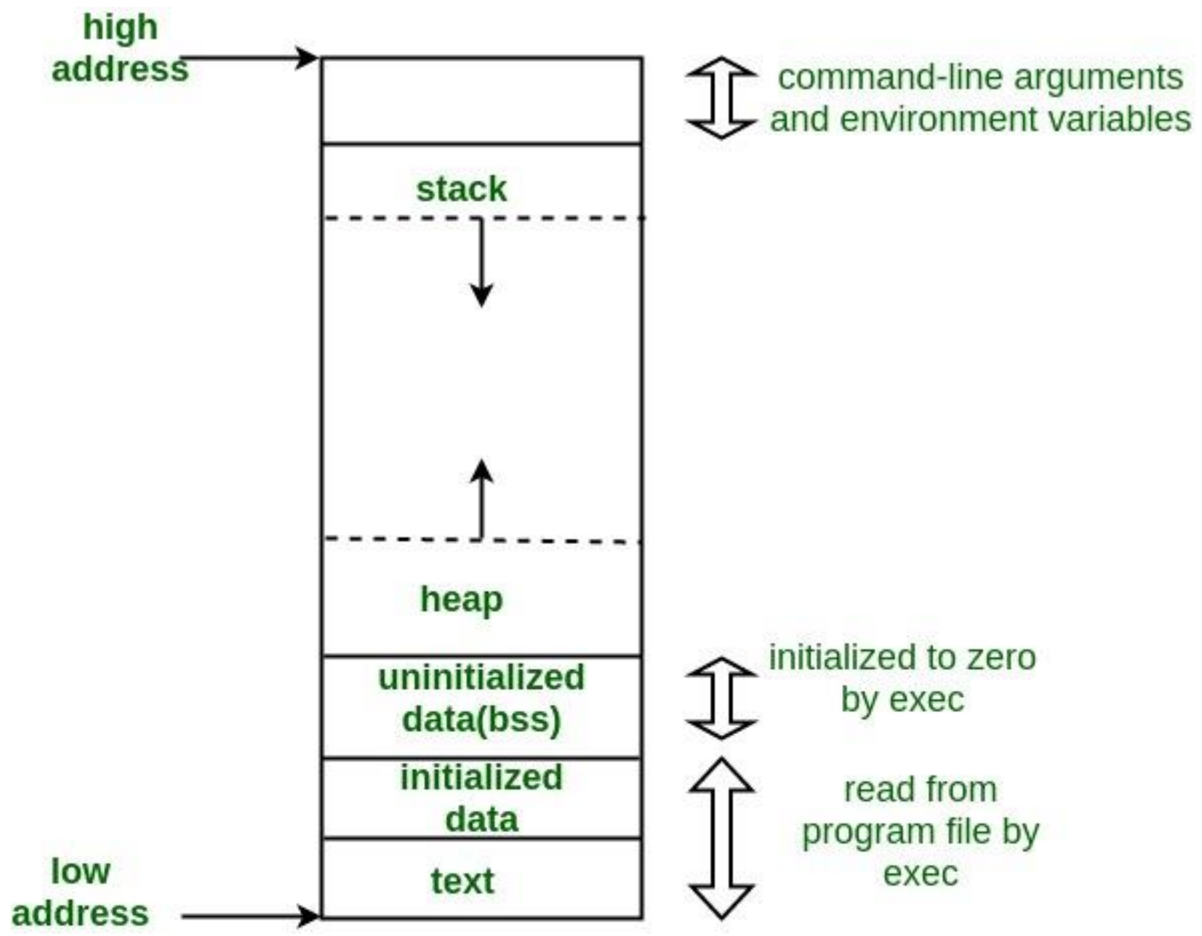
```
; Move Zero Extend
movzx rax, word [l1]
movzx rbx, bx
movzx rbx, cl

; Move Sign Extend
movsx rax, dword [l1]
movsx rbx, bx
movsx rbx, cl
```

PROGRAM SECTIONS

Program Sections

- Text (Instructions)
- Data (Global Variables)
- BSS (Global Variables)
- Stack (Return Address, Local Variables, Saved Registers)
- Heap

Program Sections



https://www.geeksforgeeks.org/memory-layout-of-c-program/

Program Sections

- Modern Programs Still have those five sections but, they are not stored in memory like this.
- You will learn about it in Operating Systems Course.

LOGIC

What is different from High Level Languages?

- Variables
- If & Loop
- Function Calling, Struct, High-Level Programming

Implementing If

- If is implemented using conditional jumps (branches) and non conditional jumps in all assembly languages
- In amd64 assembly branches depend on side effect of previous instructions
- Implementing if, else, or, and
  - You might want to use logical not of the original condition

Implementing If

```
        ; if (rax > rbx)
        ;   inc rax
        ; inc rbx

        cmp rax, rbx
        jle end_if
if_cond:    inc rax
end_if: inc rbx
```

Implementing If, Else

```
          ; if (rax > bax)
          ;   inc rbx
          ; else
          ;   inc rax
          ; dec rcx

           cmp rax, rbx
           jle else_cond
if_cond:       inc rbx
           jmp end_if
else_cond:     inc rax
end_if:    dec rcx
```

Implementing If (&&)

```
          ; if (rax > bax && rbx > rcx)
          ;    inc rbx
          ; dec rcx

           cmp rax, rbx
           jle end_if
           cmp rbx, rcx
           jle end_if
if_cond:       inc rbx
end_if:    dec rcx
```

Implementing If (||)

```
          ; if (rax > bax || rbx > rcx)
          ;   inc rbx
          ; dec rcx

           cmp rax, rbx
           jg  if_cond
           cmp rbx, rcx
           jg if_cond
           jmp end_if
if_cond:       inc rbx
end_if:    dec rcx
```

Implementing Loop

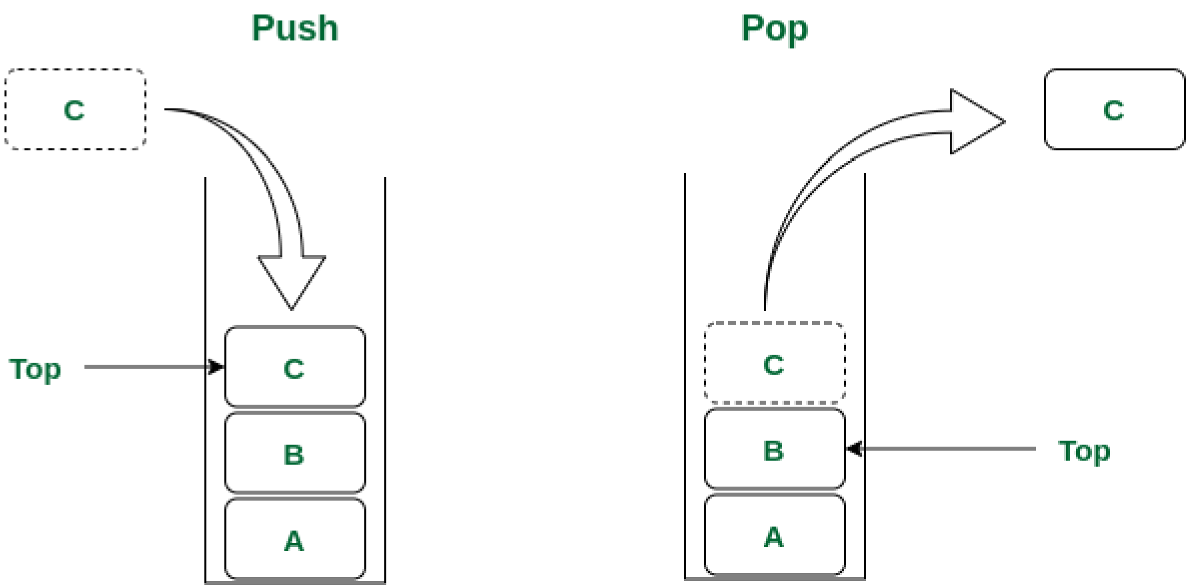
- While loop is the fundamental loop, all others can be created using while
- While Loop: If + Jump

Implementing Loop

```
           ; while (rax > bax)
           ;   inc rbx
           ; dec rcx

loop_cond: cmp rax, rbx
           jle  end_loop
in_loop:       inc rbx
           jmp  loop_cond
end_loop:  dec rcx
```

STACK

Stack



Stack Data Structure

https://www.geeksforgeeks.org/stack-meaning-in-dsa/

Stack

- Remember stack grows in reverse order in x86
  - Stack size increases when stack pointer (rsp) decreases
- Stack is used to store function return address (Next PC)
  - **call** instruction automatically pushes next PC on top of stack
  - **ret** instructions automatically pops top value of the stack and jumps to it
- Stack is used to store and later load values of registers
  - Using **push** and **pop** instructions
- Stack is used to store local variables
  - By manipulating stack pointer (rsp)
- Stack is used to pass parameters to functions

MACROS & FUNCTIONS

Macros

- Pieces of code that were written before and are added to program
- Like copy and pasting
- Like #define functions in C
- Macros are replaced in place

Functions

- When calling a function program jumps to a different address (call) and later return to it’s original execution path (ret)
- Each programming language implements them slightly different from others (calling conventions)
- Functions could be located in different memory locations from the calling program (shared libraries)
- Functions often declare local variables on top of the stack

Functions

```
swap_function:
        push rcx
        push rdx
        mov rcx, qword [rax]
        mov rdx, qword [rbx]
        mov qword [rax], rdx
        mov qword [rbx], rcx
        pop rdx
        pop rcx
        ret

mov rax, l1
mov rbx, l2
call swap_function
```

System Calls

- Requests to the operating system
- Execution stops, OS performs requested operation, then return to the program and execution continues

CALLING CONVENTIONS

Calling Conventions

- Conventions used between high level programming languages to call functions and get results from them
- Differ from language to language
- Differ from ISA to ISA

CDECL (C calling convention) for x86_64

- Input is given in this way
  - First six parameters (except floating point parameters) are given in registers (in the following order):
    - rdi, rsi, rdx, rcx, r8, r9
  - First eight floating points parameters are given in XMM0 to XMM7
  - Excess parameters will be pushed to stack in reverse order
  - Number of vector inputs is given in al (rax)
- Callee rules
  - Callee save registers rbp, rbx, r12~15
  - Callee puts output in rax or xmm0 (in case of float)
- Caller rules
  - Caller clears parameters pushed to stack

- https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf
- https://en.wikipedia.org/wiki/X86_calling_conventions

Calling convention for IBM s390x

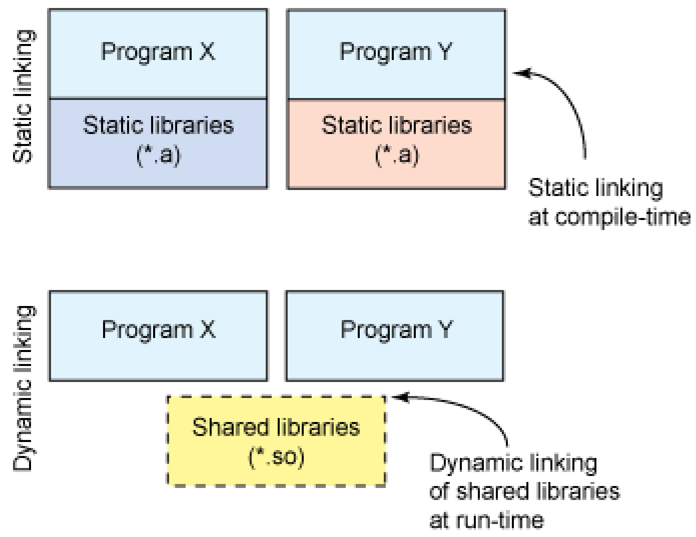
- Input is given in this way:
  - General registers r2 to r6 are used for integer values.
  - Floating point registers f0 and f2 are used for floating point values.
  - rest of the arguments are passed on the stack 96 bytes above the initial stack pointer. (lowest address for first, highest for last)
  - long long are passed in two consecutive general registers if the next available register is smaller than 6. If the upper 32 bits would end in general register 6 then this register is skipped and the whole 64 bit value is passed on the stack.
  - by reference. If needed, the called function makes a copy of the value.
  - Caller clears stack after function call
- Output will be put in r2 or (r2:r3)
- Registers r6~r13, r15 and f4~f6 are saved by called function, rest are volatile
- Registers r12~r15 have special purpose
- https://refspecs.linuxbase.org/ELF/zSeries/lzsabi0_s390.html#AEN414
- https://en.wikipedia.org/wiki/Calling_convention#IBM_System/360_and_successors
- https://legacy.redhat.com/pub/redhat/linux/7.1/es/os/s390x/doc/lzsabi0.pdf

SHARED LIBRARIES

Shared Libraries

- Shared Libraries are often used by many programs (e.g. GLIBC)
- Shared libraries are compiled, linked and loaded in memory in a special manner (you will learn more about it in operating systems course)
- Shared libraries are only loaded once into the memory and all programs use that one instance
- Therefore calling them requires few considerations (we focus on dynamic linking, you can learn about dynamic loading yourself)

Shared Libraries



https://developer.ibm.com/tutorials/l-dynamic-libraries/

Calling Shared Libraries with C calling convention

- Firstly, there is a part of calling convention we didn't mention before.
  - Stack pointer must be aligned by 16 (be a multiple of 16) before calling a function
  - It's not always necessary, specially if you don't call shared libraries in your function
  - Failing to comply with this will typically lead to segmentation fault
  - Compiler always complies with this, even if it's not necessary (since compiler might not have access to function implementation at compile time, there is no other way to be sure)
- Program must use relative addressing mode
- PLT should be used to resolve address (at runtime)

Calling Shared Libraries with C calling convention (Sample)

```
bits    64
default rel

segment .data
    print_format: db "Hello world! %d", 0xA, 0
    scan_format: db "%d", 0

segment .text

global main
extern printf
extern scanf
```

Calling Shared Libraries with C calling convention (Sample)

```
main:
    sub rsp, 8

    lea     rdi, [scan_format]
    lea     rsi, [rsp]
    mov     al, 1
    call    scanf wrt ..plt
    lea     rdi, [print_format]
    mov     rsi, [rsp]
    mov     al, 1
    call    printf wrt ..plt

    mov rax, 0
    add rsp, 8

    ret
```

Calling Shared Libraries with C calling convention (Sample)

```
amir@DESKTOP-2C4UD0U:~/x86_call$ nasm -f elf64 call.asm; gcc call.o -o call
amir@DESKTOP-2C4UD0U:~/x86_call$ ./call
1234
Hello world! 1234
amir@DESKTOP-2C4UD0U:~/x86_call$
```

INLINE ASSEMBLY

Inline assembly

- You can use assembly in middle of your C program
- Remember gcc uses at&t syntax by default. So either write your assembly code in at&t syntax, mark syntaxes when they change or tell gcc to use intel syntax
  - the .intel_syntax and .att_syntax directives change assembly syntax in middle of program

- https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
- https://gcc.gnu.org/onlinedocs/gcc/extensions-to-the-c-language-family/how-to-use-inline-assembly-language-in-c-code.html
- https://en.cppreference.com/w/c/language/asm

Inline assembly - Format

```
asm ( AssemblerTemplate)

asm asm-qualifiers ( AssemblerTemplate
             : OutputOperands
             [ : InputOperands
             [ : Clobbers ] ])

asm asm-qualifiers ( AssemblerTemplate
                 : OutputOperands
                 : InputOperands
                 : Clobbers
                 : GotoLabels) – Only in case
of goto qualifier
```

Inline assembly

- Your assembly section is copied directly in output assembly file
  - No Checks
  - No modifications
- Therefore you should fully disclose clobbers
- Rest of your code might be displaced (as a result of compiler optimization)
- Volatile Qualifier Prevents compiler optimization (only w.r.t. your assembly section)
- With Volatile your assembly code stays where it is with regards to rest of your code

Inline assembly – Parameters

Registers	
a	rax
b	rbx
c	rcx
d	rdx
S	rsi
D	rdi
r	Register
f	float register

- In case memory changes (other than outputs) “memory” must be included in clobbers too
- In clobbers we use full names

Inline assembly – Sample

```
asm ("xchgl rax, rbx"
     : "=a" (x), "=b" (y)
     : "a" (x), "b" (y)
     : );
```

```
asm ("xchgl $0, $1"
     : "=r" (x), "=r" (y)
     : "r" (x), "r" (y)
     : );
```

Inline assembly – Sample

```
char msg[] = "Hello, World!\n";
int length = strlen(msg);
asm ("mov eax, 4;" // system call 4: sys_write
     "mov ebx, 1;" // file handle 1: stdout
     "int 0x80;" // syscall
     : : "c" (msg), "d" (length) : "eax", "ebx");
```

INTRINSICS

Intrinsics

- Some programming languages have special function calls (called intrinsics) for instructions that compiler wouldn't use (e.g. SIMD instructions)
- See this link for example. (Intel intrinsics for C)
  - https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

END OF SLIDES