Computer Structure and Language

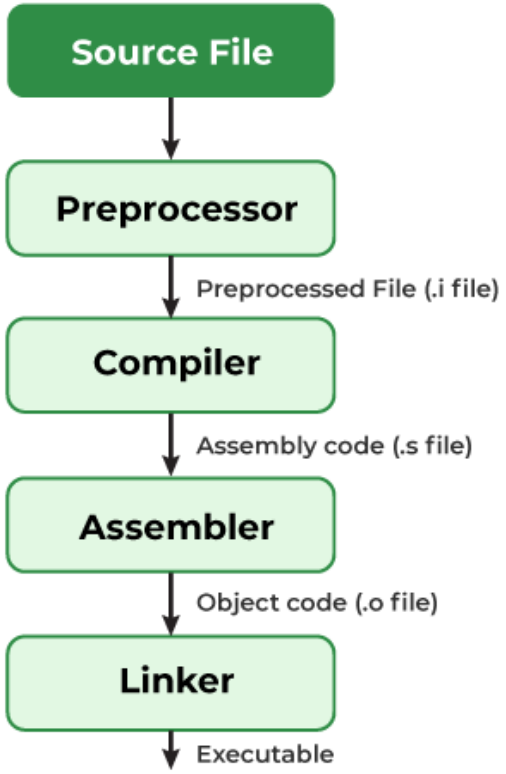
Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

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.



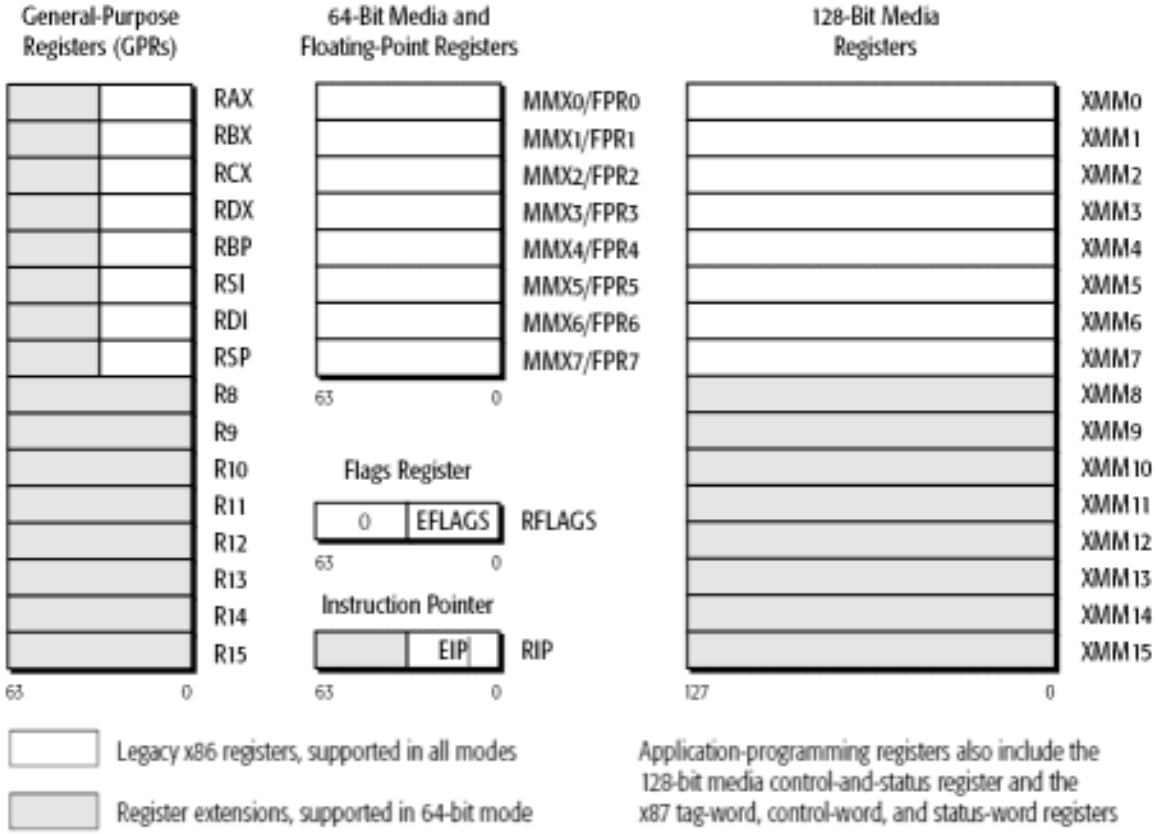
https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/

DATA REPRESENTATION

Registers

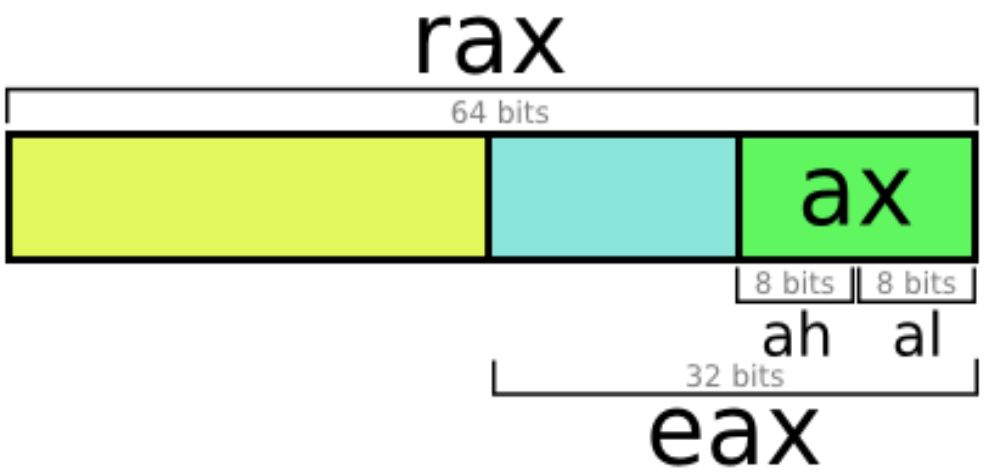
- High speed data stores built into the processor
- Limited in number, can not store all of the data required for the program
- Data has to move between them and memory
- Some of the registers have special purpose

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

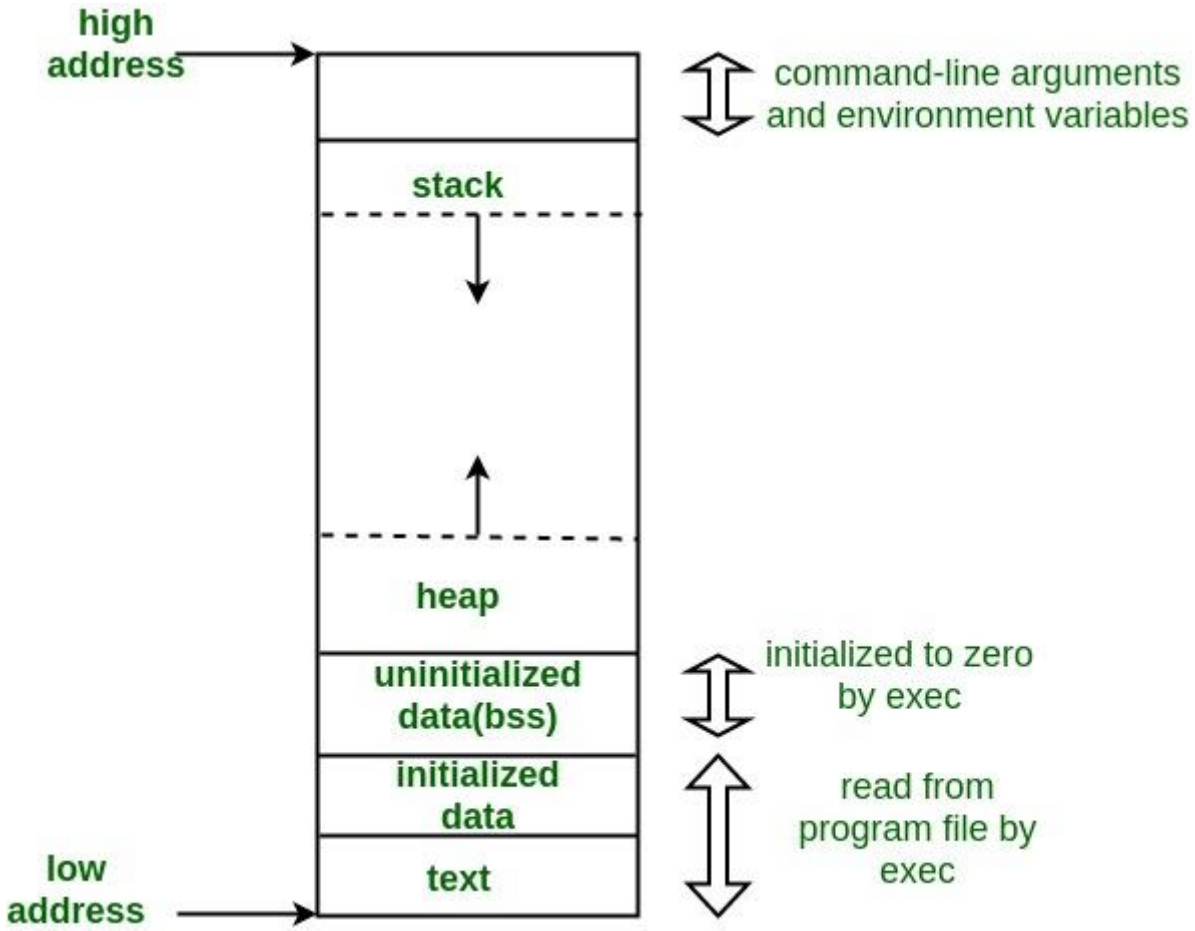
```
; Move Zero Extend
movzx rax, word [l1]
movzx rbx, bx
movzx rbx, cl

; 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

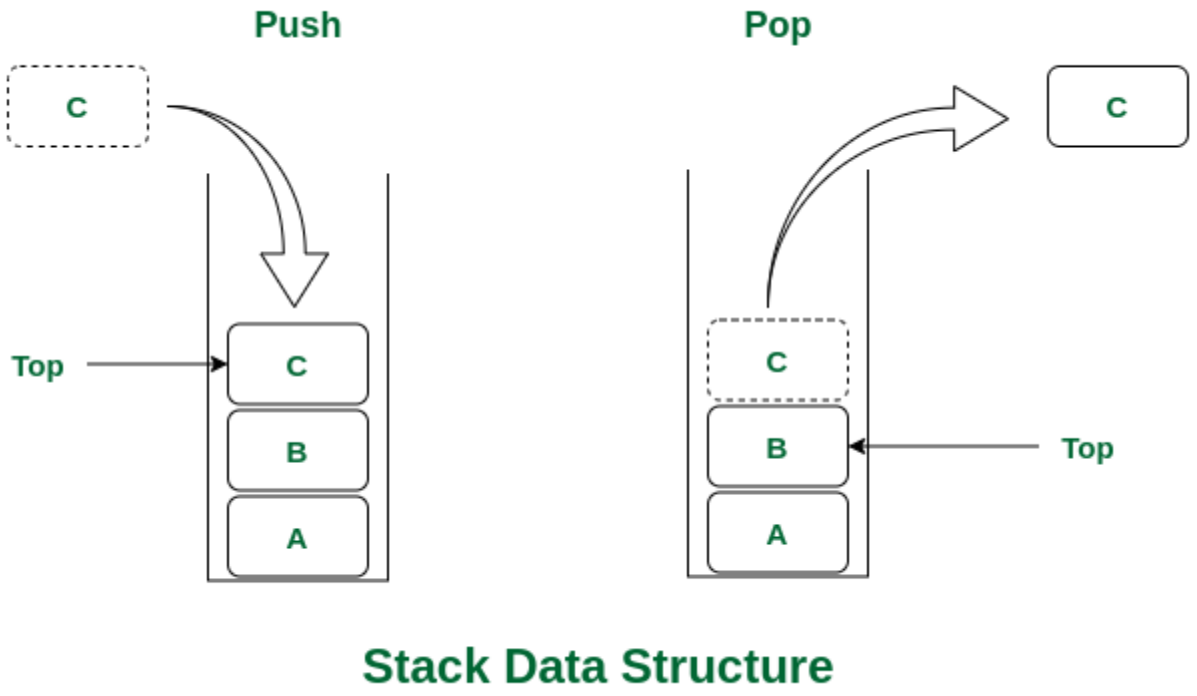
- While loop is the fundamental loop, all others can be created using while
- While Loop: If + Jump

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

END OF SLIDES