Writing in Assembly

- Introduction
- x86 Data Types & Register Organization
- Selected Operations
- Put it to work: Hello World
- Addressing memory
- Stack and Function Calls

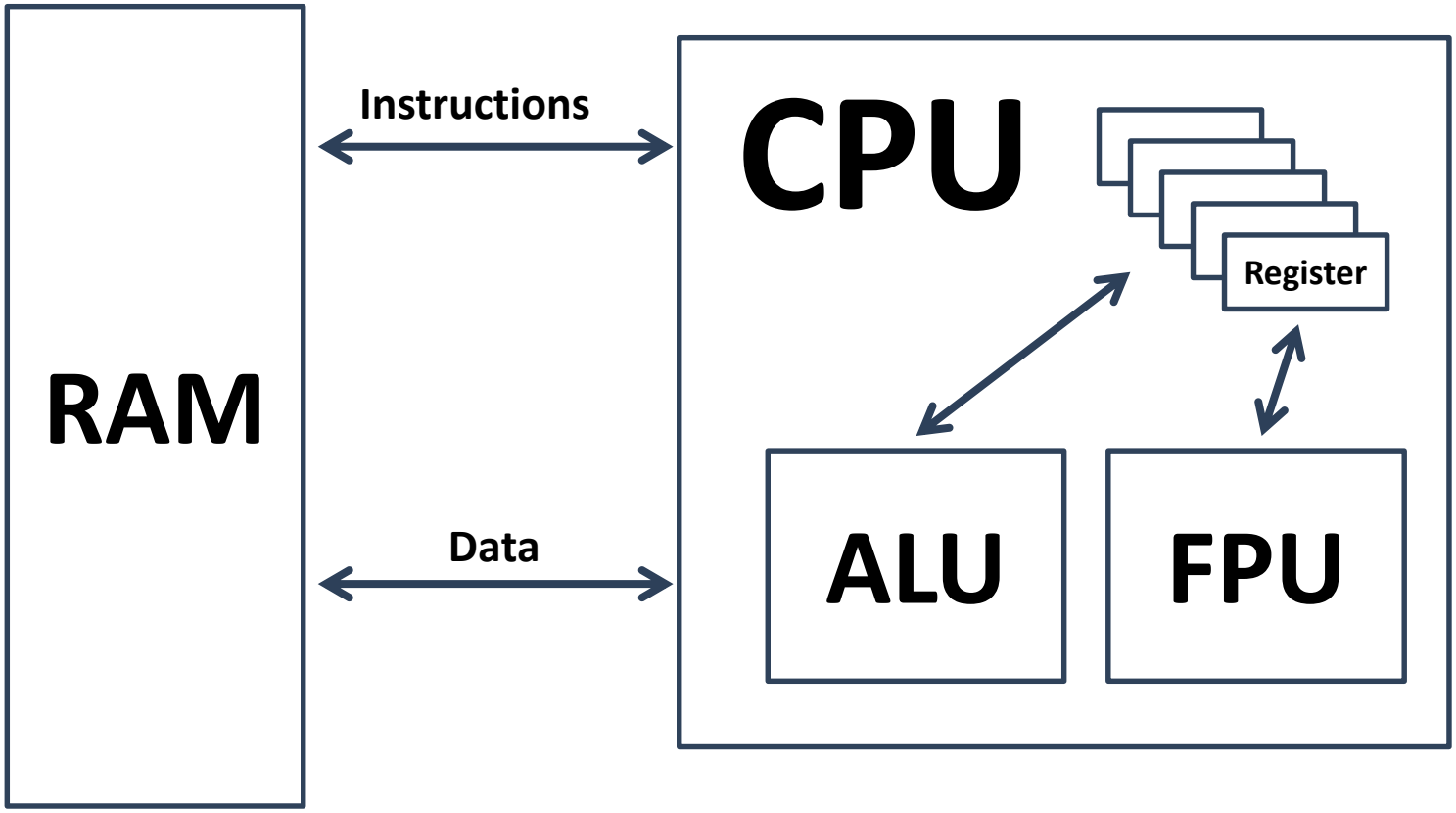
Introduction

- We will learn about the basics to get you started with writing code in assembler
- We will learn how machine code looks in general
- We will discuss some basic commands
- Don’t worry if you do not understand everything right now and today … you will have a lot of time in the lab to learn the practice

What is a Machine Instruction

- The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions*
- The collection of different instructions that the processor can execute is referred to as the processor's *instruction set*
- Each instruction must contain the information required by the processor for execution
- **What does the processor need in order to perform an Instruction?**

Very, very, very Simplified View



What does the CPU need to know?

- **Operation Code**: Specifies the operation to be performed
- **Source Operand**: The Operation may involve one or more inputs
- **Result Operand:** The operation may produce a result
- **Next Instruction:** Tells the processor where to fetch the next instruction

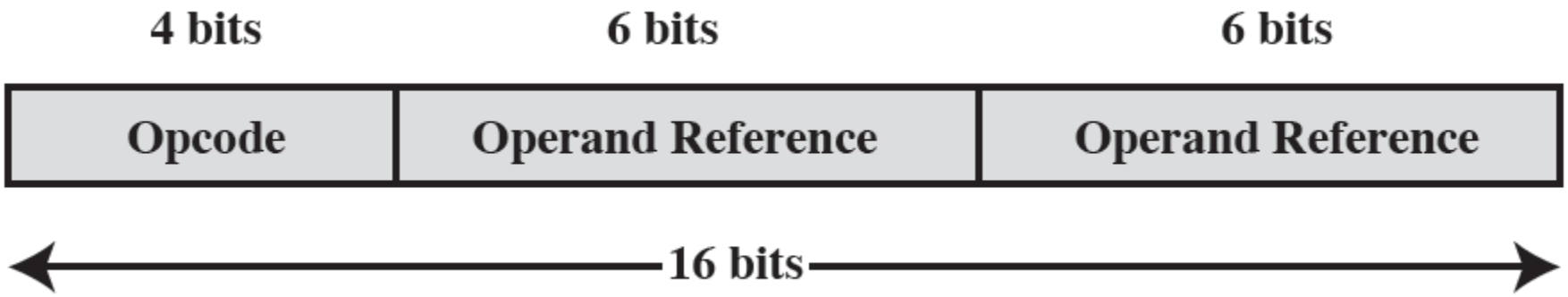
Source or Result Operands

- **Processor Register**: Every (almost) CPU has one or more registers. The name or number of the desired register must be given
- **Main Memory**: Main memory address must be given
- **Immediate:** That is a constant value which is directly stored in the instruction
- **I/O Device:** Must specify I/O module and device. With memory mapped I/O), this equals to a normal main memory operation

Instruction Types

- Each processor should have instructions of the following kind
- **Data Processing**: Arithmetic and Logic (boolean) instructions
- **Data Storage**: Movement of data into or out of registers and or memory locations
- **Data Movement**: I/O instructions
- **Control**: Test and branch instructions

Simple Instruction Format



Number of Addresses

- Ways of calculating $Y = \frac{A-B}{C+(D \cdot E)}$

Instruction		Comment
SUB	Y, A, B	Y ← A – B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y – B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction		Comment
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC – B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

(c) One-address instructions

Instruction Set Design

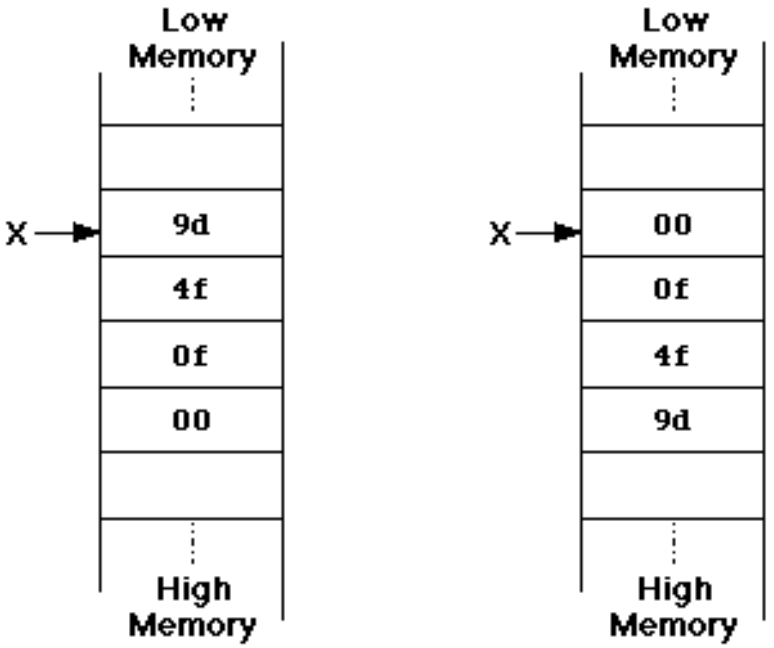
- **Operation Repertoire**:
  - How many and which operations to provide, and how complex operations should be
- **Data Types**:
  - The various types of data upon which operations are performed
- **Instruction Format**:
  - Instruction length (in bits), number of addresses, size of various fields, and so on
- **Registers**:
  - Number of processor registers that can be referenced by instructions, and their use
- **Addressing**:
  - The mode or modes by which the address of an operand is specified

Writing in Assembly

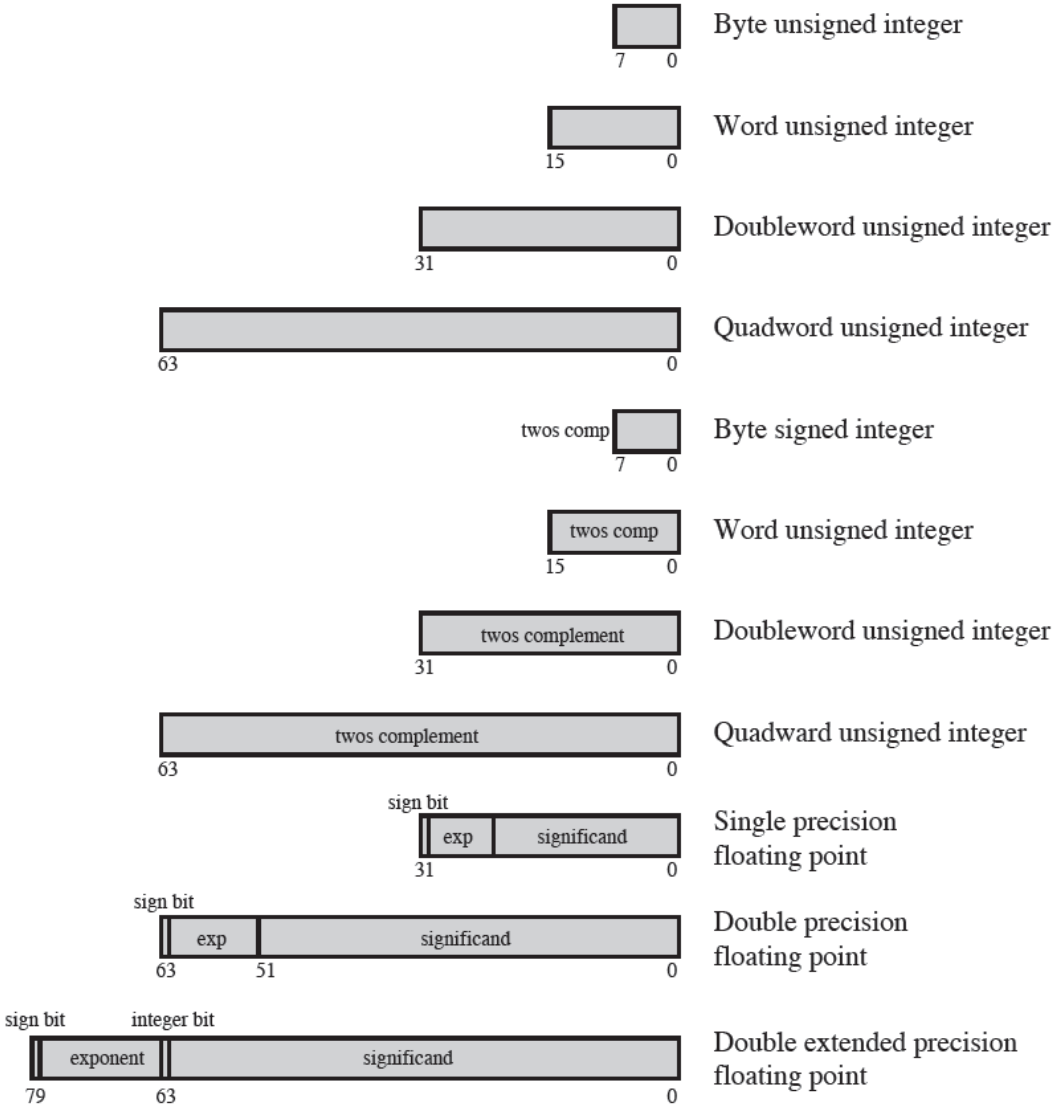
- Introduction
- x86 Data Types & Register Organization
- Selected Operations
- Put it to work: Hello World
- Addressing memory
- Stack and Function Calls

General Information

- Supports bytes (8), words (16), **doublewords** (32), **quadwords** (64) and double quadwords (128)
- Words need not to be aligned in Memory
  - Nevertheless, they should because of performance
- Intel uses **Little Endian**
  - The least significant byte is is stored in the lowest address
  - Example: The integer 1003421_{10} ($000f4f9d_{16}$) stored as little endian (left) and big endian (right)
- Wide range of numeric data types



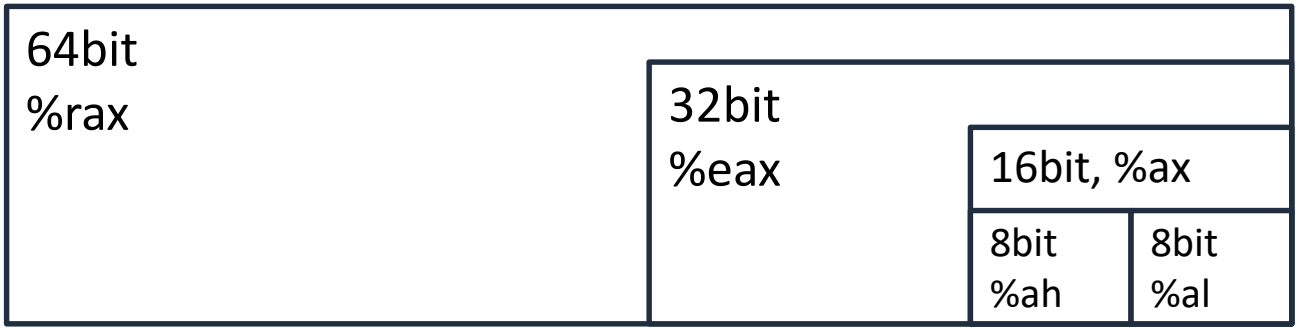
Numeric Dataformats



The Register Structure of x86_64

Name	(historic) Meaning
	General Purpose Registers
RAX	Accumulator register. Used in arithmetic operations.
RCX	Counter register. Used in shift/rotate instructions and loops.
RDX	Data register. Used in arithmetic operations and I/O operations.
RBX	Base register. Used as a pointer to data.
RSP	**Stack Pointer register. Pointer to the top of the stack.**
RBP	**Stack Base Pointer register. Used to point to the base of the stack.**
RSI	Source Index register. Used as a pointer to a source in stream ops.
RDI	Destination Index register. Used as a pointer to a destination.
R8-R15	Introduced with the 64 Bit extensions
	Other
EFLAGS	Status word
SS, CS ..	Segment Registers
RIP	Instruction Pointer

Why those strange names? History!



- The counterparts for the new R8-R15 registers are:
- R8D
- R8W
- R8L

- There is no equivalent for %ah

Writing in Assembly

- Introduction
- x86 Data Types & Register Organization
- Selected Operations
- Put it to work: Hello World
- Addressing memory
- Stack and Function Calls

What we are using

- We are using the GNU assembler
- We are using AT&T syntax

```
op    source,target
```

- Read the instructions from left to right
- For most instructions you should specify the register length:
  - b = byte (8 bit)
  - s = short (16 bit integer) or single (32-bit floating point)
  - w = word (16 bit)
  - l = long (32 bit integer or 64-bit floating point)
  - q = quad (64 bit)

The AT&T syntax

- Registers are indicated with a “%”:

```
%rax
```

- Comments are started with “#”:

```
#Blah
```

- Immediates start with a “$”:

```
$5
```

- Hex values are indicated by “0x”:

```
$0x123
```

Data Transfer Instructions

- **Move**

```
mov    src, dest
```

  - **src**: Immediate, register or memory
  - **dest**: Register or memory
  - Example: **movq $0x5,%rax**

- **Move and extend**

```
movz    src, dest
```

  - **src**: Register or memory
  - **dest**: Register
  - Example: **movzwl %ax,%ebx**
  - **movs** extends with the sign

Data Transfer Instructions

- **Data swap**

  `xchg src, dest`

  - **`src`**: Register or memory
  - **`dest`**: Register or memory
  - Example: **`xchg %rax,%r8`**
  - There exists also a **`cmpxchg`** instruction

- **Stack operations**

  `push/pop src/dest`

  - **`src`**: Immediate, register or memory
  - **`dest`**: Register or memory
  - Example: **`pushq %rax`**

Arithmetic Operations

- **Addition/Subtraction**

```
add/sub   src, dest
```

  - **`src`**: Immediate, register or memory
  - **`dest`**: Register or memory
  - Example: **`addq  %rax,%r8`**
  - Corresponds to: **`dest = dest +/- src`**

- **Unsigned multiplication**

```
mul   arg
```

  - **`arg`**: Immediate, register or memory
  - Multiplies **`%rax`** with **`arg`**. Stores the result in **`%rdx:%rax`**
  - Example: **`mulq  %r8`**
  - There exists also a **`imul`** instruction which performs a signed multiplication

Arithmetic Operations

- **Unsigned division**

```
div   arg
```

  - **`arg`**: Immediate, register or memory
  - Devides **`%rdx:%rax`** by **`arg`**. (Make sure **`%rdx`** is set correctly)
  - Result: **`%rax`**; Remainder **`%rdx`**
  - There exists also a **`idiv`** instruction which performs an signed division

- **Special signed multiplication**

```
imul   src,dest
imul   src1,src2,dest
```

  - Corresponds to: **`dest = dest * src`**
  - Corresponds to: **`dest = src1 * src2`**

Control Flow

- **Comparison Instructions**

```
cmp   arg2, arg1
```

  - **arg1**: Register or Memory
  - **arg2**: Immediate, register or memory
  - Performs temp = arg1 – arg2
  - This operation causes flags to be set. Among them:
    - SF (signed flag) according to the sign of temp
    - ZF (zero flag) if ZF == 0

```
test   arg2, arg1
```

  - **arg1**: Register or Memory
  - **arg2**: Immediate, register or memory
  - Performs temp = arg1 && arg2

Control Flow

- **Unconditional Jump**

  `jmp loc`

  - Loads the corresponding address into instruction counter

- **Conditional Jumps**

  `jXX loc`

  - Normally, a conditional jump follows after a cmp instruction
  - **`je`** (Jump on equality)
  - **`jne`** (Jump on inequality)
  - **`jg / jge`** (Jump if grater / greater equal)
  - **`jl / jle`** (Jump if lower / lower equal)
  - …

Logic & Shift Instructions

- **Logic Instructions**

```
and/or/xor   src, dest
```

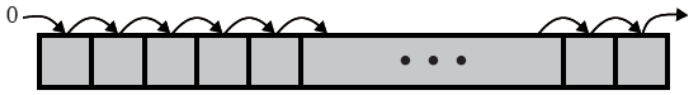
  - **`src`**: Immediate, register or memory
  - **`dest`**: Register or Memory
  - Example: **`xor %rdx,%rdx`**

- **Shift and Rotate Instructions**

```
op   src, dest
```

  - **`src`**: Immediate, register or memory
  - **`dest`**: Register or Memory
  - Maipulates **`dest`** by **`src`** bits
  - **`op`**: see next slide

Different Shifts



(a) Logical right shift

shr

(b) Logical left shift

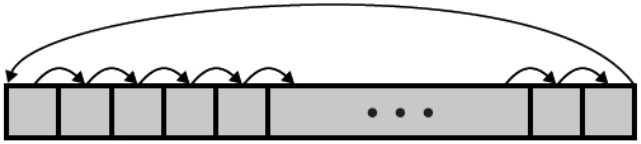
shl

(c) Arithmetic right shift

sar

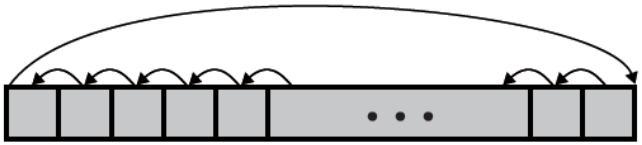
(d) Arithmetic left shift

sal



(e) Right rotate

ror



(f) Left rotate

rol

Writing in Assembly

- Introduction
- x86 Data Types & Register Organization
- Selected Operations
- Put it to work: Hello World
- Addressing memory
- Stack and Function Calls

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

- Assembly programs are separated into sections
- In **`.data`**, you normally define variables, reserve space for them, etc
- In the section **`.text`** we put the actual program

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

- Here, we define a variable of type .ascii
- The assembler takes care of reserving the space and initializing everything

- Other types are:
  - **.double**
  - **.float**
  - **.byte**
  - **.int**
  - …

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

- This is required for the linker
- You are basically telling it that you program starts at the line indicated with **`_start:`**

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

- Okay, our first instruction:
- This means: We move **1** into register **rax**
- Remember:
  - Regsiters are prefixed with a **%**
  - Constants with **$**

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

- We are asking the linux kernel for help:
  - The number of the syscall is in rax
  - Arguments are in registers
  - Additional arguments can be on the stack (we will come to that later)
  - The return value is in eax

  - The called routine is expected to preserve rsp,rbp, rbx, r12, r13, r14, and r15 **but may trample any other registers**.

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

Okay, our system call:

- **%rax=1**
  - Syscall for write
- **%rdi=1**
  - file descriptor
  - 1 is always stdout
- **%rsi**
  - Address of the buffer
  - We let the assembler give calculate the address of our label
- **%rdx = 13**
  - Number of bytes to write

Lets Start with Hello World

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
        mov $1, %rax
        mov $1, %rdi
        mov $hello,%rsi
        mov $13,%rdx
        syscall

        mov $60, %rax
        mov $0, %rdi
        syscall
```

And the next syscall:

- **%rax=60**
  - syscall exit
- **%rdi=0**
  - exit code

How to Run?

```
> as hello_world.asm -o hello_world.o
> ld hello_world.o -o hello_world
> ./hello_world
Hello World!
>
```

Other Syscalls

- **read:**

arg:	**%rax**	0
	%rdi	file descriptor (0 is stdin)
	%rsi	ptr to input buffer
	%rdx	buffer size
ret:	**%rax**	length of read bytes (pointer advanced accordingly)

- **open:**

arg:	**%rax**	2
	%rdi	ptr to filename string
	%rsi	file access bits
	%rdx	mode
ret:	**%rax**	file descriptor

➢ http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64 for an overview of all syscalls

Other Syscalls

- **close:**

arg:	**%rax**	3
	%rdi	file descriptor
ret:	**%rax**	--

- http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64 for an overview of all syscalls

Control Flow: Unconditional Jump

```
        ...
        jmp exit
        ...

exit:
        mov $60, %rax
        mov $0, %rdi
        syscall
```

- The instruction `jmp` unconditionally jumps to the given address
- The label `exit` gets replaced from the assembler with the actual address

Control Flow: Conditional Jumps

```
        ...
        cmp %rax,%rbx
        je exit
        ...

exit:
        mov $60, %rax
        mov $0, %rdi
        syscall
```

- The instruction `cmp val1,val2` compares two values and sets status bits

- We then can decide depending on these bits what to do:
  - **`je <adr>`** Jump if equal
  - **`jne <adr>`** Jump if not equal
  - **`jg <adr>`** Jump if `val2` > `val1`
  - **`jge <adr>`** Jump if `val2` ≥ `val1`
  - **`jl <adr>`** Jump if `val2` < `val1`
  - **`jle <adr>`** Jump if `val2` ≤ `val1`

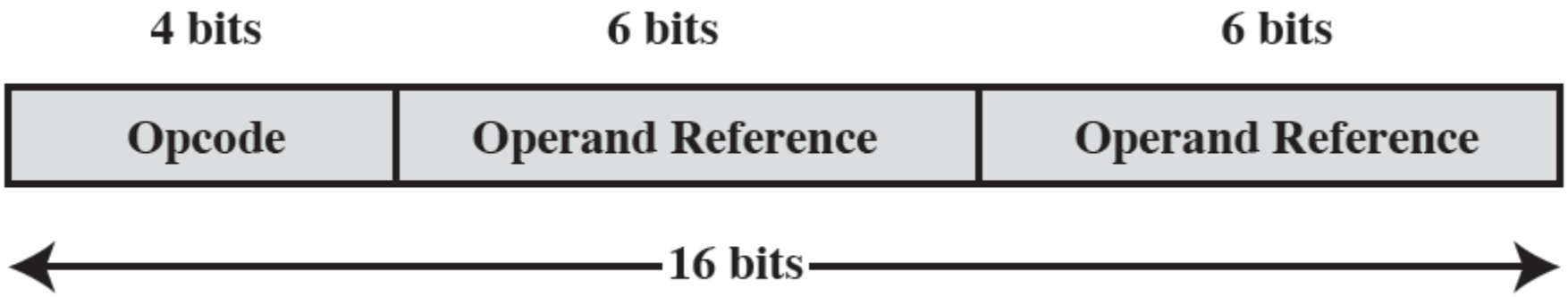
Writing in Assembly

- Introduction
- x86 Data Types & Register Organization
- Selected Operations
- Put it to work: Hello World
- Addressing memory
- Stack and Function Calls

Basic Addressing Principles

- The address field(s) in a typical instruction format are relatively small, too small to address the entire memory
- To overcome this limit, a variety of addressing techniques has been employed.
- Generally they are some trade-off between address range and/or addressing flexibility and complexity.

A very Simple Instruction Format

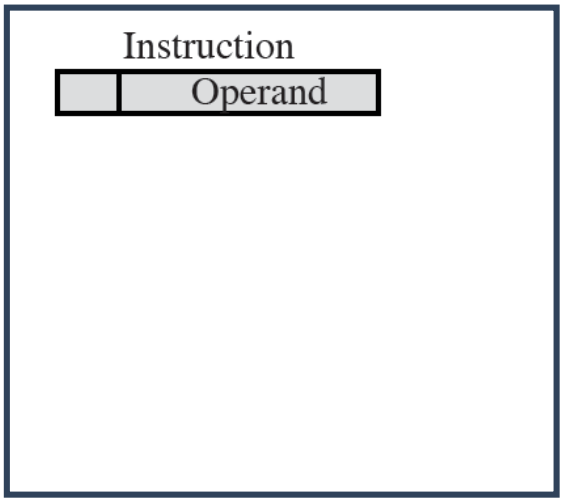


Basic Addressing Principles

- Typical Addressing Modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - Stack
- Here:
  - A = contents of an address field in the instruction
  - R = contents of a register
  - (X) = contents of memory location X
  - EA = Effective Address

Basic Addressing Principles

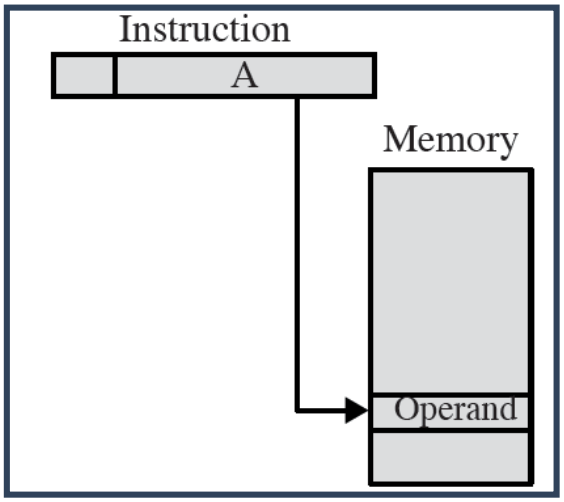
- Typical Addressing Modes are:
  - **Immediate**
    - **Operand = A**
    - Simplest form of addressing (actually it is not really addressing at all)
    - Limited in Size
    - Is used, e.g., to load constants into a registers
  - Direct
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - Stack



- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Basic Addressing Principles

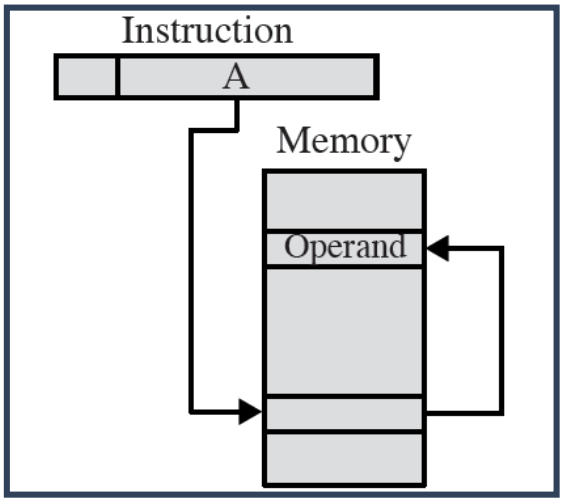
- Typical Addressing Modes are:
  - Immediate
  - **Direct**
    - **EA = A**
    - The address field of the instruction contains the effective address
    - Very common in older computers
    - Limited address size
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - Stack



- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Basic Addressing Principles

- Typical Addressing Modes are:
  - Immediate
  - Direct
  - **Indirect**
    - **EA = (A)**
    - The address field of the instructions refers to a word in the memory
    - This field contains the actual address
    - Addressable Space is 2^K different 2^N addresses (K being the width of the address field, N the width of a memory word)

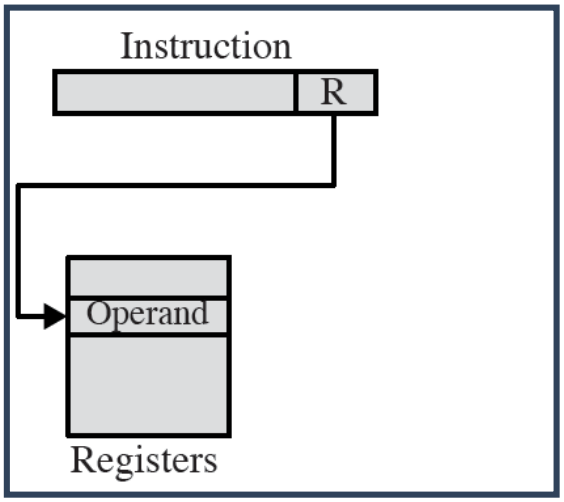


  - Register
  - Register indirect
  - Displacement
  - Stack

- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Basic Addressing Principles

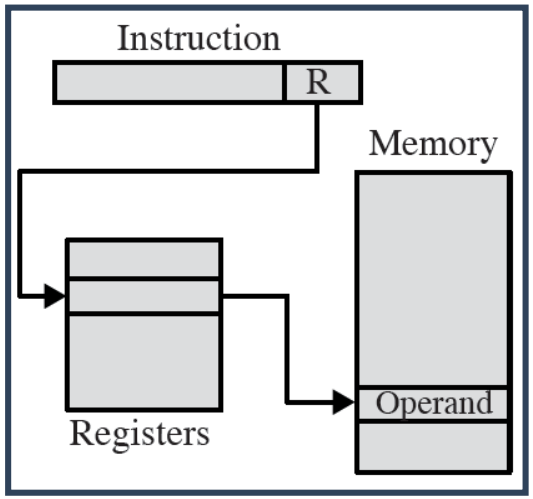
- Typical Addressing Modes are:
  - Immediate
  - Direct
  - Indirect
  - **Register**
    - **EA = R**
    - Same as direct addressing, but this time referring to a register
    - e.g. if the address field has the value 5, the effective address is R5
    - extremely fast but very limited
  - Register indirect
  - Displacement
  - Stack



- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Basic Addressing Principles

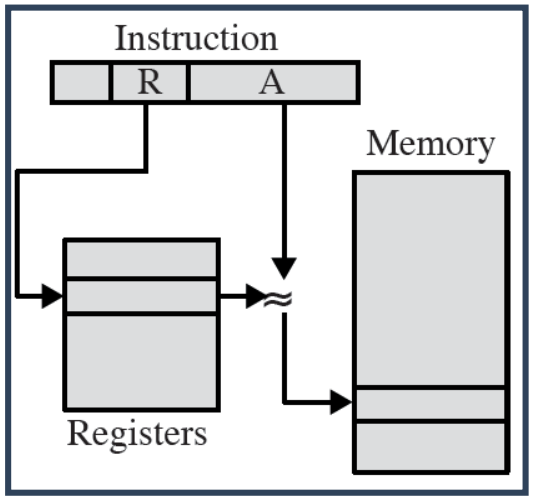
- Typical Addressing Modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - **Register indirect**
    - **EA = (R)**
    - Same as indirect addressing, but this time the EA is stored in a register
    - faster than indirect addressing, as only one memory operation is required
  - Displacement
  - Stack



- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Basic Addressing Principles

- Typical Addressing Modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register indirect
  - **Displacement**
    - **EA = A + (R)**
    - Very powerful, but complicated
    - The instruction must support at least two address fields
    - There are several common displacement addressing modes
  - Stack



- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Displacement Addressing

- **Relative Addressing**
  - Also called PC-relative addressing
  - The PC (program counter) forms the basis of the calculation
  - The effective address is a displacement relative to address of the instruction
  - Exploits the concept of locality
  - Saves address bits in the instruction if most memory references are relatively near to the instruction being executed

- **Base-register Addressing**
- **Indexing**

- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Displacement Addressing

- **Relative Addressing**
- **Base-register Addressing**
  - The referenced register contains a main memory address and the address field contains a displacement from that address
  - The register reference may be explicit or implicit
  - Exploits the locality of memory references
  - Convenient means of implementing segmentation
  - In some implementations a single segment base register is employed and is used implicitly
  - In others the programmer may choose a register to hold the base address of a segment and the instruction must reference it explicitly
- **Indexing**

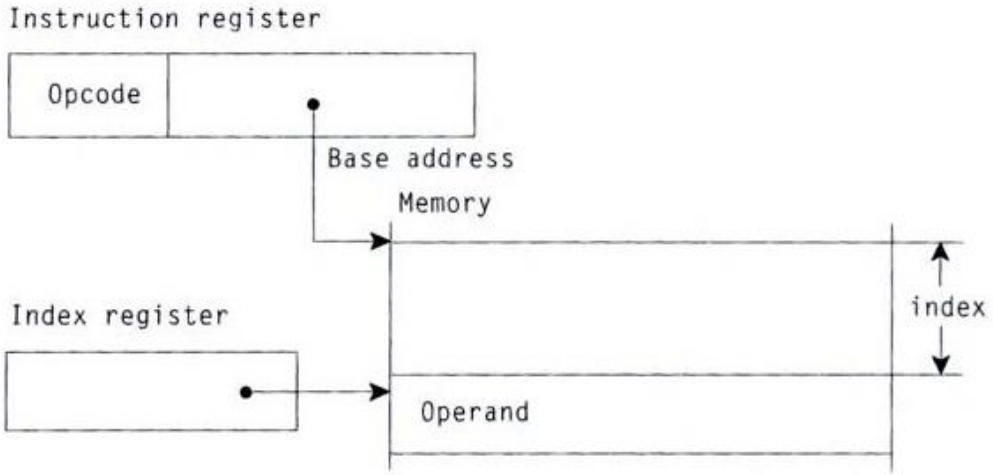
- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Displacement Addressing

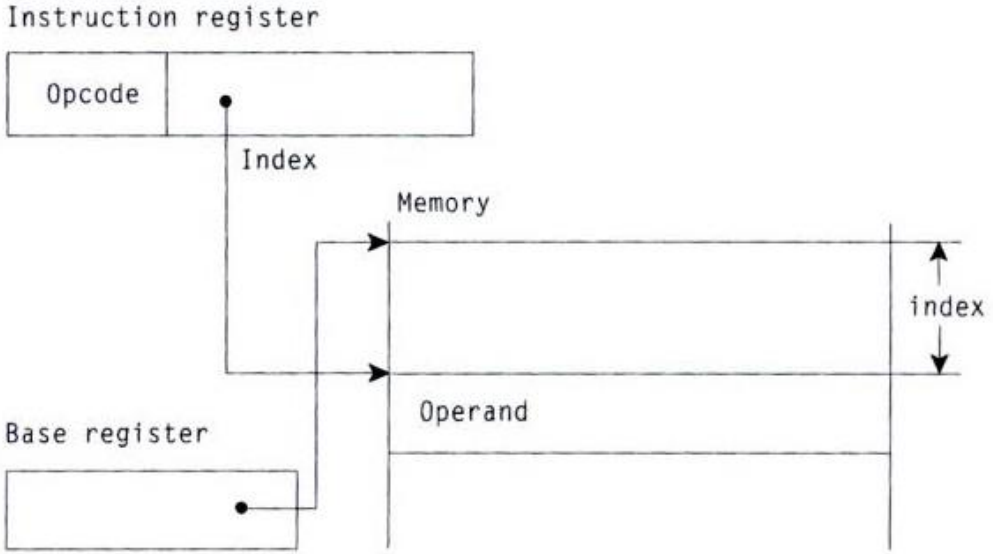
- **Relative Addressing**
- **Base-register Addressing**
- **Indexing**
  - The address field references a main memory address and the referenced register contains a positive displacement from that address
  - The method of calculating the EA is the same as for base-register addressing
  - An important use is to provide an efficient mechanism for performing iterative operations

- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Difference Between Indexing and Base Register



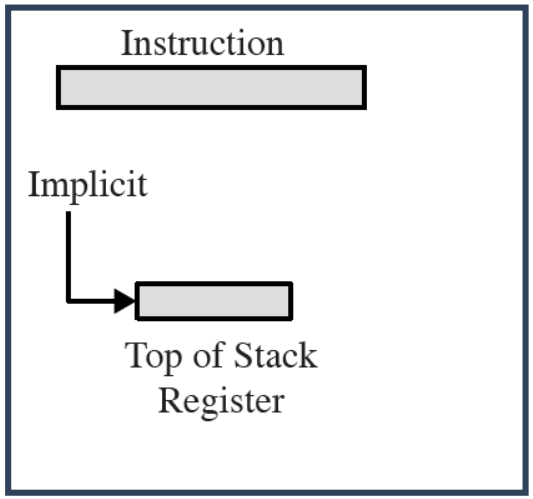
(a)



- John Hsu: **Computer Architecture: Software Aspects, Coding, and Hardware**, CRC Press, 2001.

Basic Addressing Principles

- Typical Addressing Modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - **Stack**

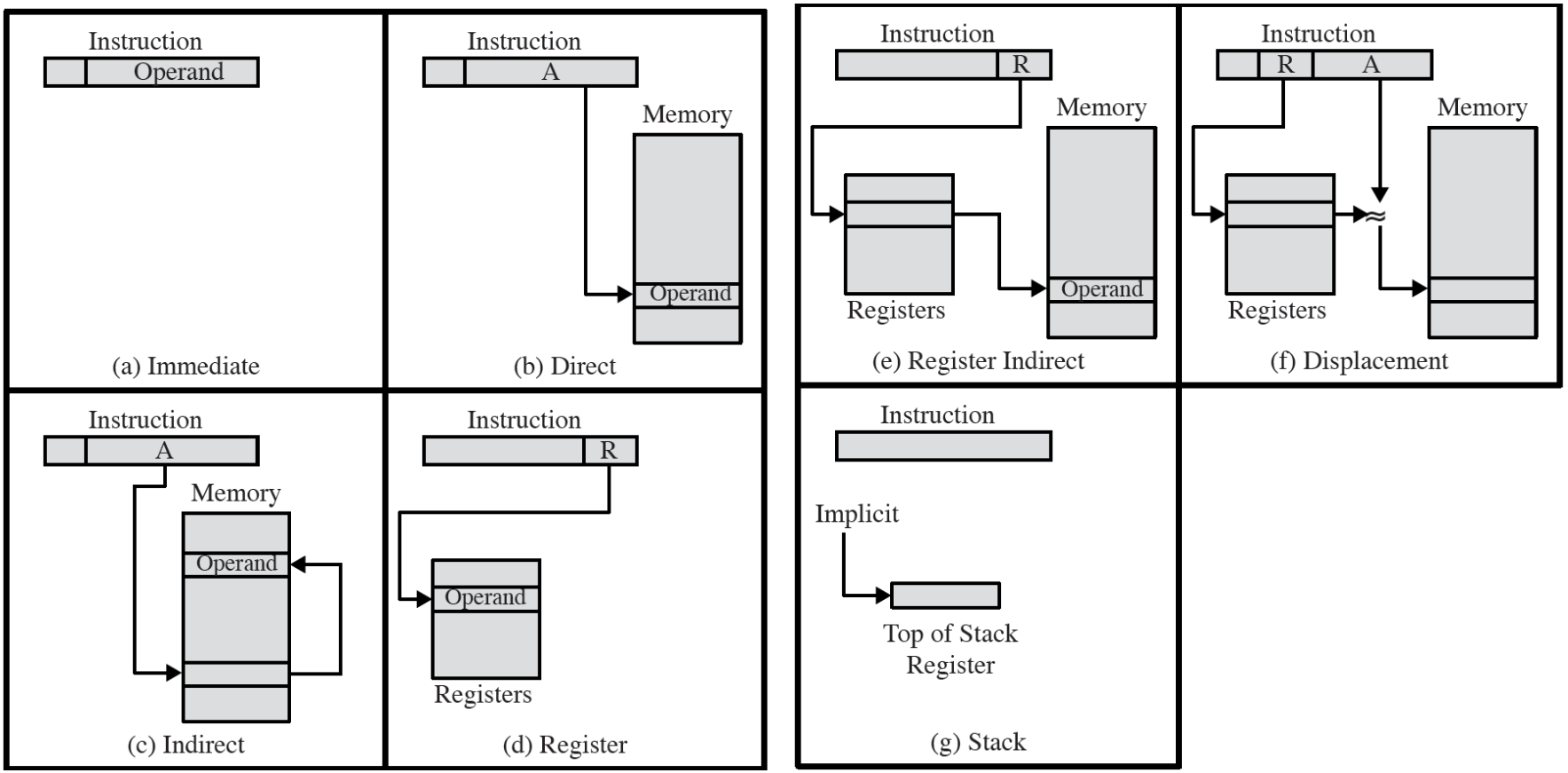


- **A = contents of an address field in the instruction**
- **R = contents of a register**
- **(X) = contents of memory location X**
- **EA = Effective Address**

Stack Addressing

- A stack is a linear array of locations
  - Sometimes referred to as a pushdown list or last-in-first-out queue
  - The stack grows from the high address to the lower address
- The Stack Pointer
  - Most machines have a stack pointer register (e.g., `%rsp`)
  - Thus references to stack locations in fact register indirect addresses
  - Is a form of implied addressing
  - Many machines also have a stack base pointer register (e.g., `%rbp`)
- Operated with **push** and **pop**
- Essential for supporting functions (later in this lecture)

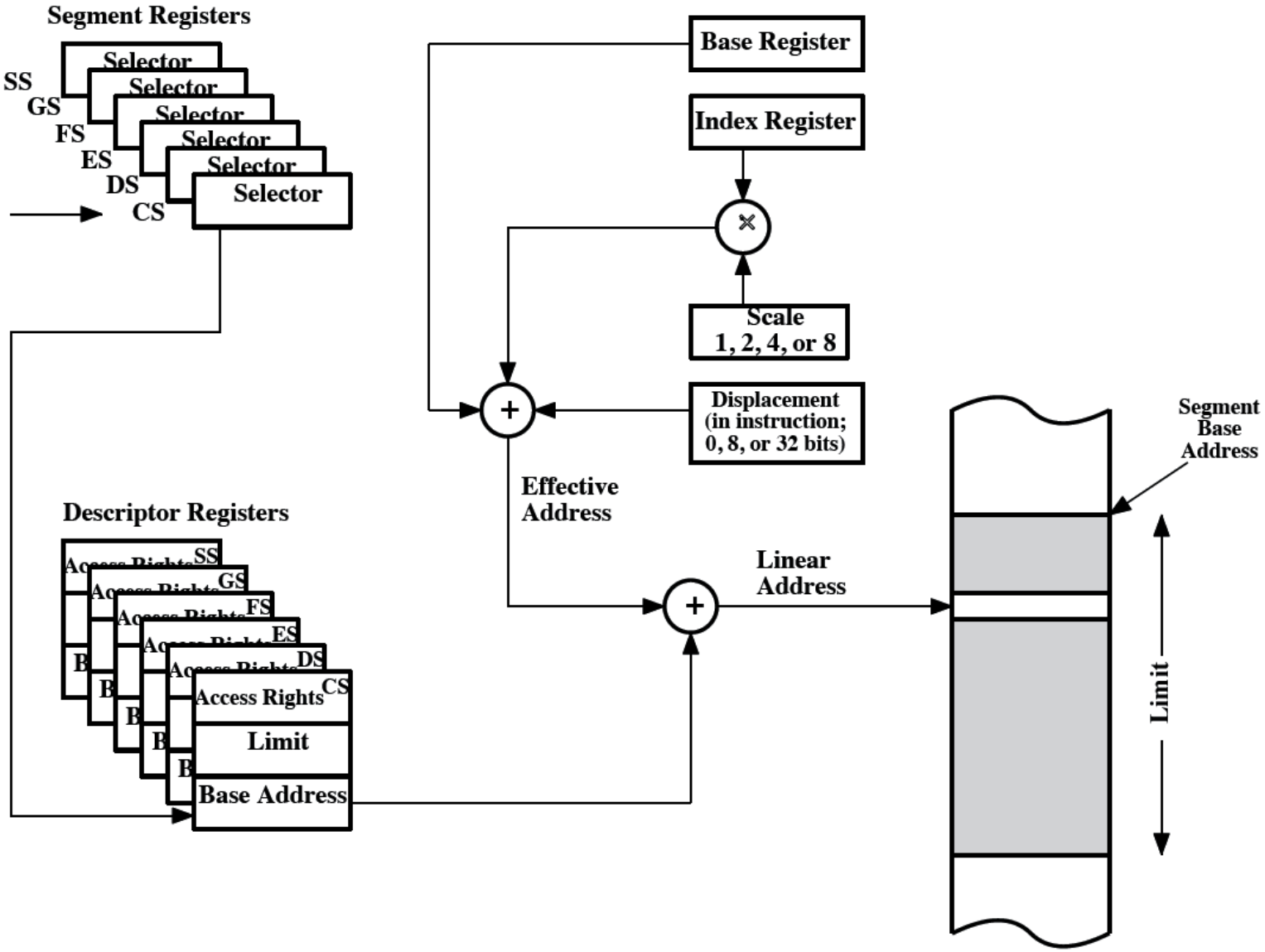
Basic Addressing Principles



Virtual Addresses

- Most modern CPUs support virtual addresses for paging etc.
- This adds further indirections to the address calculation
- This is transparent to the programmer
- The programmer produces a **effective address**
- The MMU of the CPU then calculates the actual address

x86 Addressing Modes



x86 Addressing Modes

Mode	Algorithm
Immediate	Operand = A
Register Operand	adr = R
Displacement	adr = (SR) + A
Base	adr = (SR) + (B)
Base with Displacement	adr = (SR) + (B) + A
Scaled Index with Displacement	adr = (SR) + (I) x S + A
Base with Index and Displacement	adr = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	adr = (SR) + (I) x S + (B) + A
Relative	adr = (PC) + A

(X) = Content of register X
A = Content of the address field in the instruction
S = Scaling factor
SR = Segment Register
PC = Program counter
R = Register
B = Base register
I = Index register

How Does It Look in Assembly

- We look at the GNU Assembler:

```
displacement(base register, offset register,
                scalar multiplier)
```

- This is equivalent to

```
[base register + displacement + offset
    register * scalar multiplier]
```

Some Examples

- `movl -4(%ebp, %edx, 4), %eax`
  - # Full example: load *(ebp - 4 + (edx * 4)) into eax
- `movl -4(%ebp), %eax`
  - # Typical example: load a stack variable into eax
- `movl (%ecx), %edx`
  - # No offset: copy the target of a pointer into a register
- `leal 8(,%eax,4), %eax`
  - # Arithmetic: multiply eax by 4 and add 8
- `leal (%eax,%eax,2), %eax`
  - # Arithmetic: multiply eax by 2 and add eax (i.e. multiply by 3)

- `lea`: Load effective address. Often exploited by compilers to help the scheduler as this does not set any flags.

Writing in Assembly

- Introduction
- x86 Data Types & Register Organization
- Selected Operations
- Put it to work: Hello World
- Addressing memory
- Stack and Function Calls

What Should Functions Do?

- Key Features:
- Calling and returning
- Passing parameters
- Storing local variables
- Handling registers without interference
- Return values

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

How not to do it?

```
P:      ...
        jmp foo
retp:   ...

foo:
        ...
        jmp retp
```

- This clearly doesn't work
- You can have several points were you want to call this function
- How to distinguish between the different callers?

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

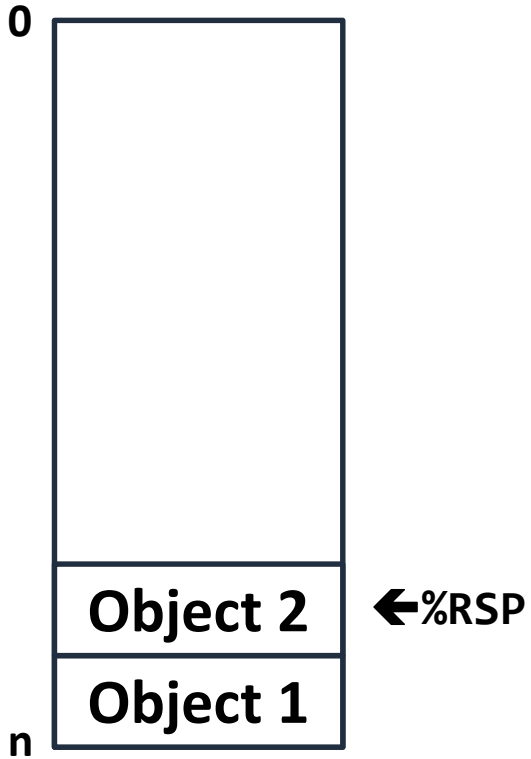
How not to do it?

```
P:      ...
        mov $ret1,%rax
        jmp foo
ret1:   ...

Q:      ...
        mov $ret2,%rax
        jmp foo
ret2:   ...

foo:
        ...
        jmp *%rax
```

- This seems to work
- Btw. **`jmp *%rax`** is a special form of jmp
- But what about recursive functions and nested functions?

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

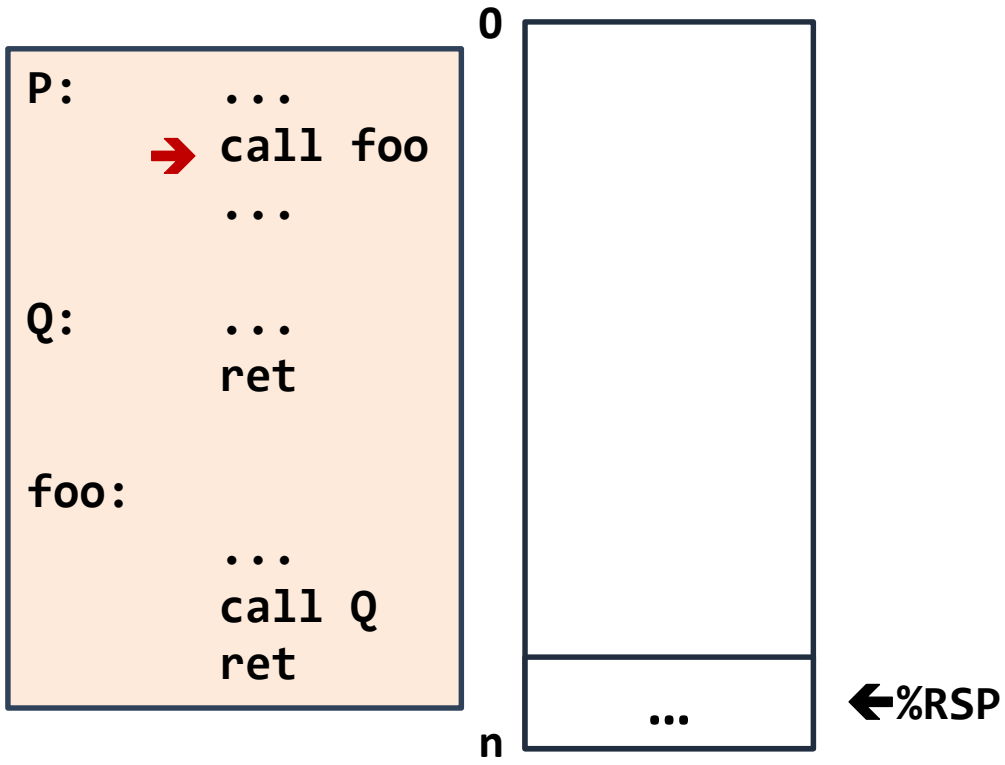
We Will use the Stack



- The stack grows from the high address to the lower address
- The stack pointer **`%rsp`** points always to the top of the stack
- push: put value on stack and decrease stack pointer
- pop: retrieve element from stack and increase stack pointer

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

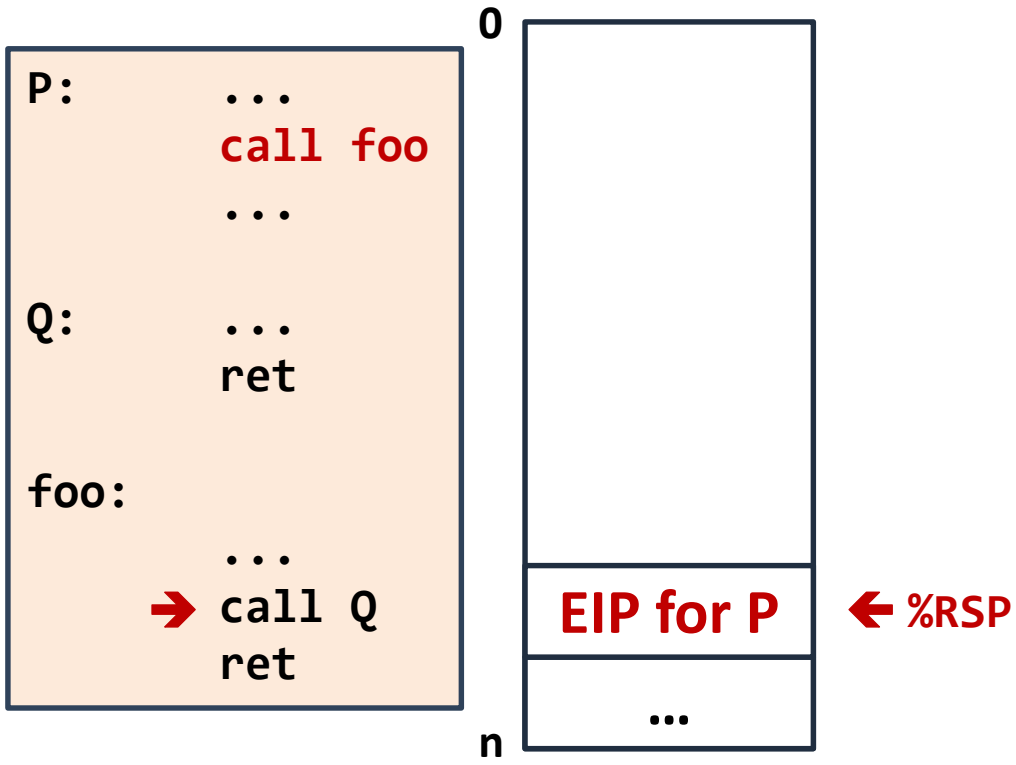
Using the Stack



- Now, we use call and ret in order to jump to functions and return back.

- Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

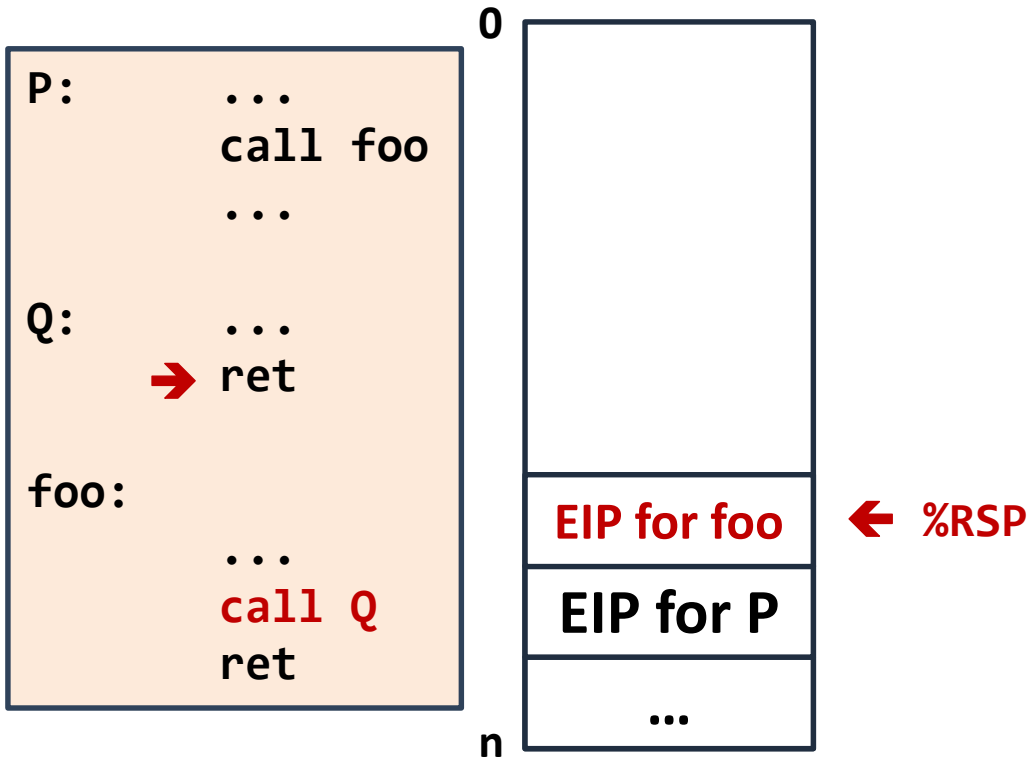
Using the Stack



- Now, we use call and ret in order to jump to functions and return back.
- Call automatically puts the return address on the stack

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

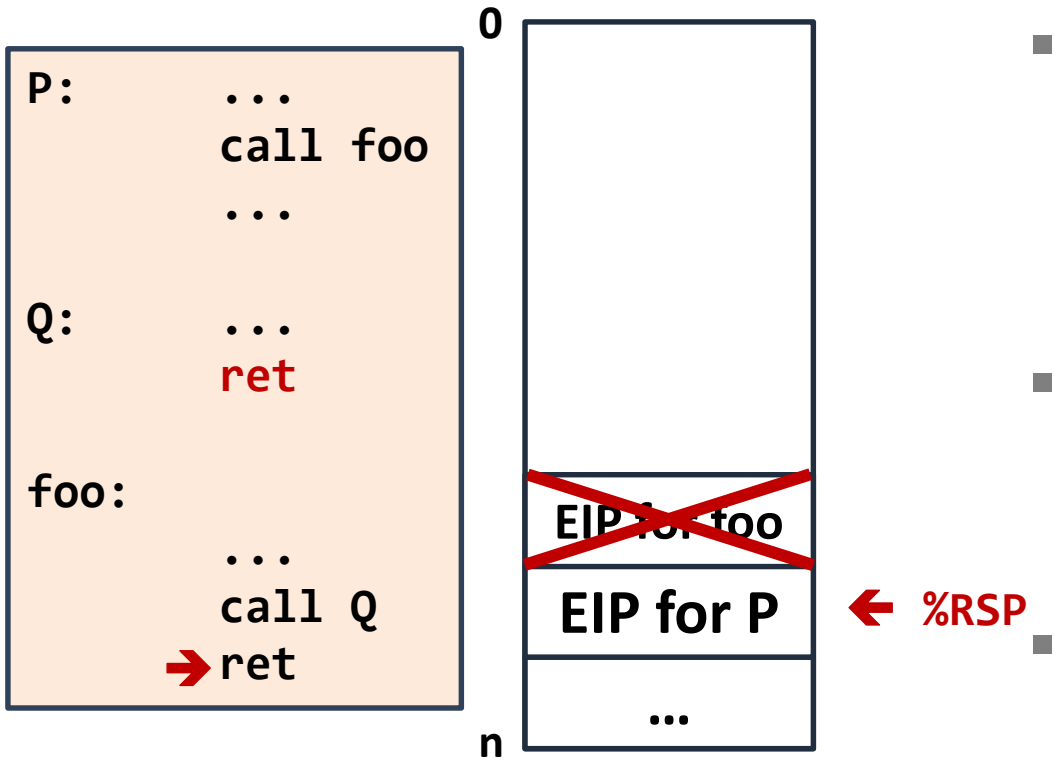
Using the Stack



- Now, we use call and ret in order to jump to functions and return back.
- Call automatically puts the return address on the stack

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

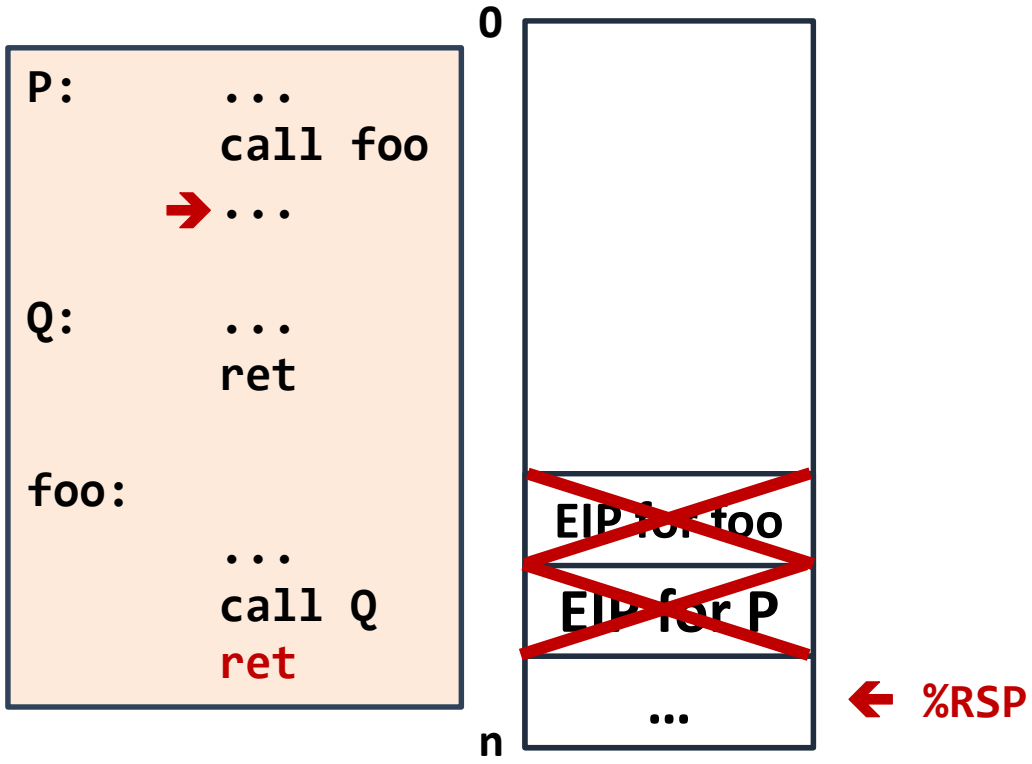
Using the Stack



- Now, we use call and ret in order to jump to functions and return back.
- Call automatically puts the return address on the stack
- Return reads the return address from the stack and jumps accordingly

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

Using the Stack



- Now, we use call and ret in order to jump to functions and return back.
- Call automatically puts the return address on the stack
- Return reads the return address from the stack and jumps accordingly

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

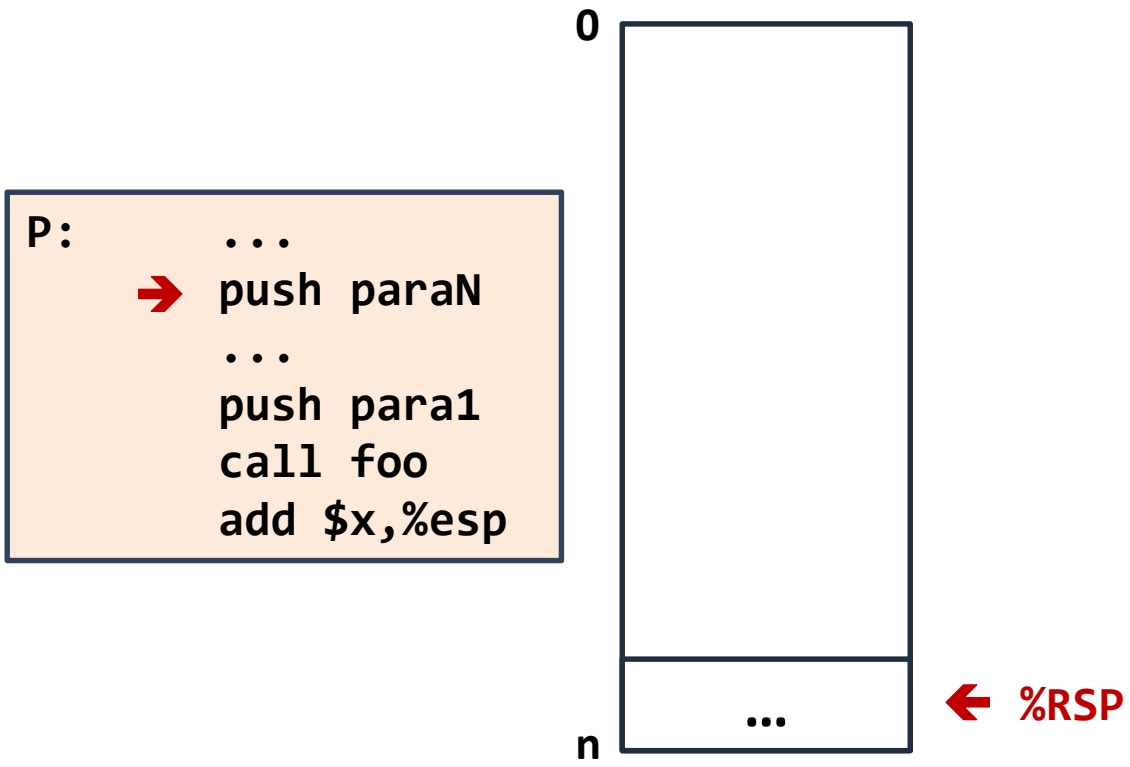
Passing Parameters

- We could simply use registers
- This is done, but there need conventions be followed
  - Which registers values will be destroyed
  - Which will be conserved
- And what is if we have more parameters than registers?
- Best method: we use the stack again!

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

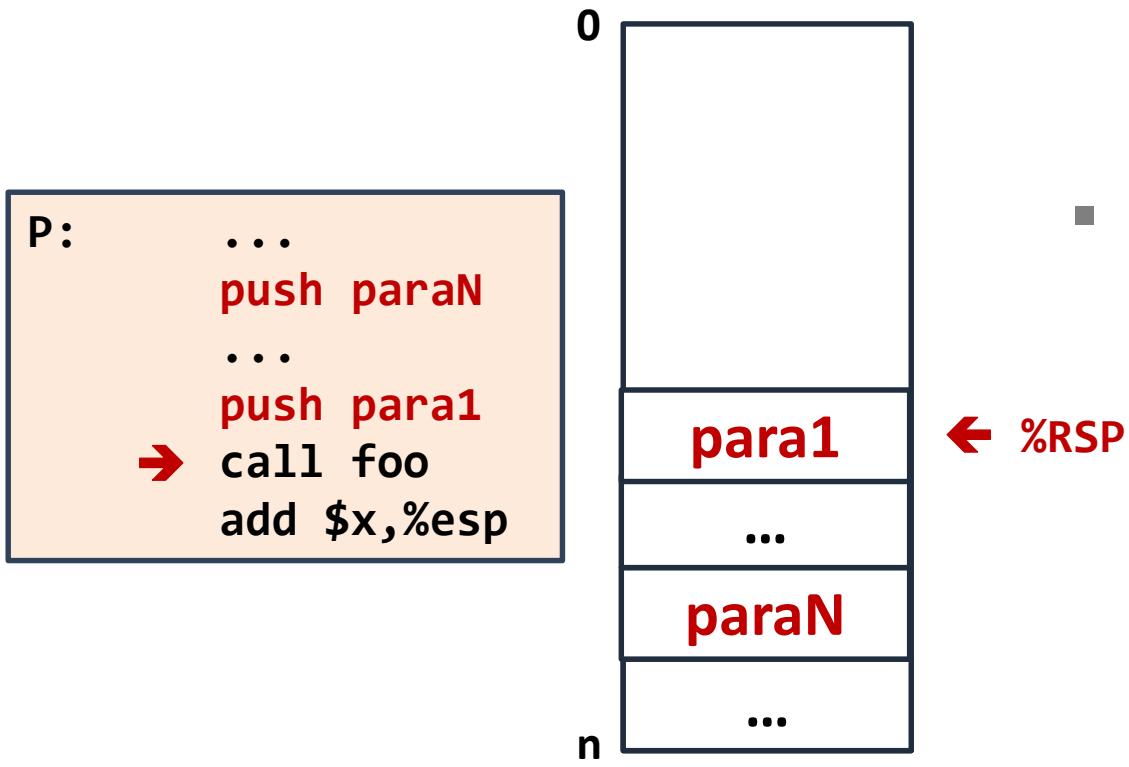
Using the Stack

- We start by pushing the parameters on the stack



- Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

Using the Stack

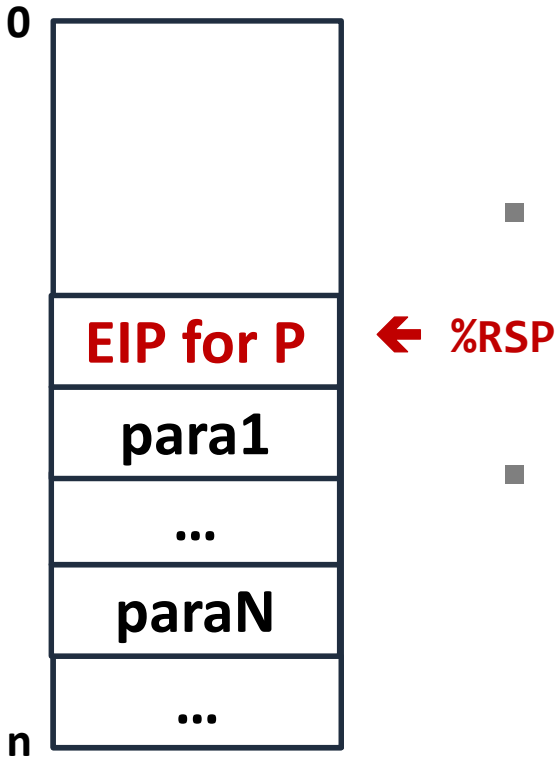


- We start by pushing the parameters on the stack
- We begin with the last parameter

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

Using the Stack

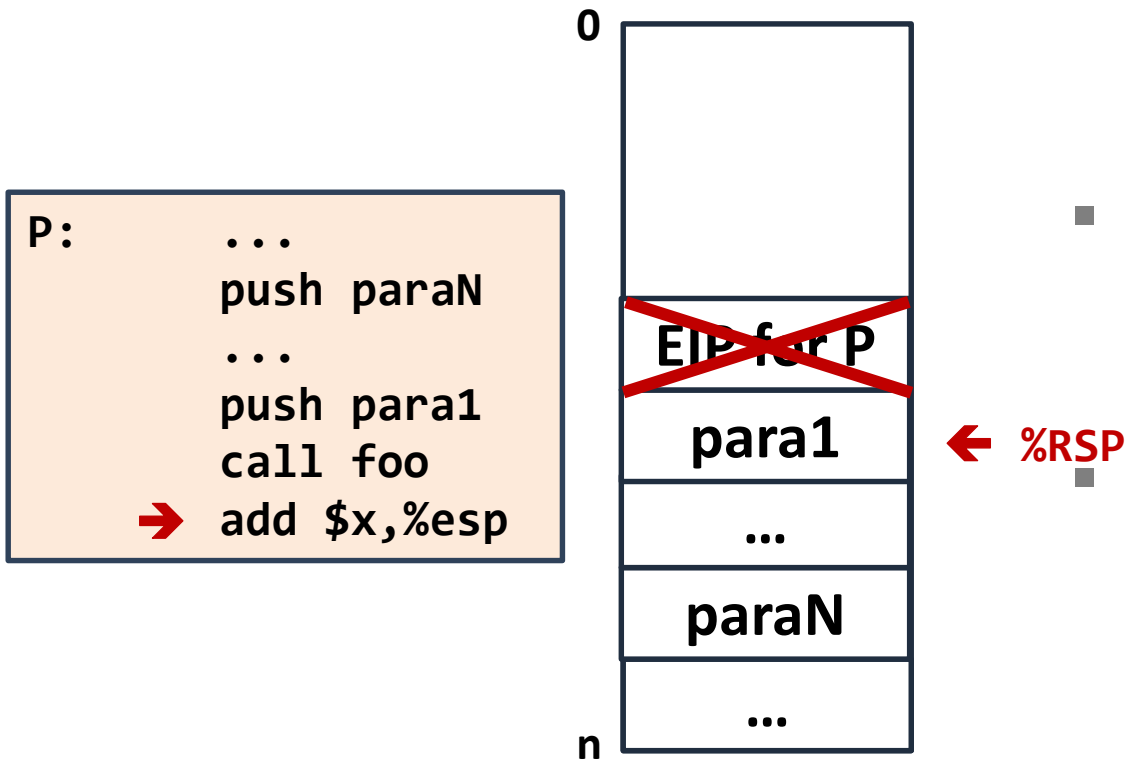
```
P:      ...
        push paraN
        ...
        push para1
        call foo
      ➔ add $x,%esp
```



- We start by pushing the parameters on the stack
- We begin with the last parameter
- We call the function

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

Using the Stack

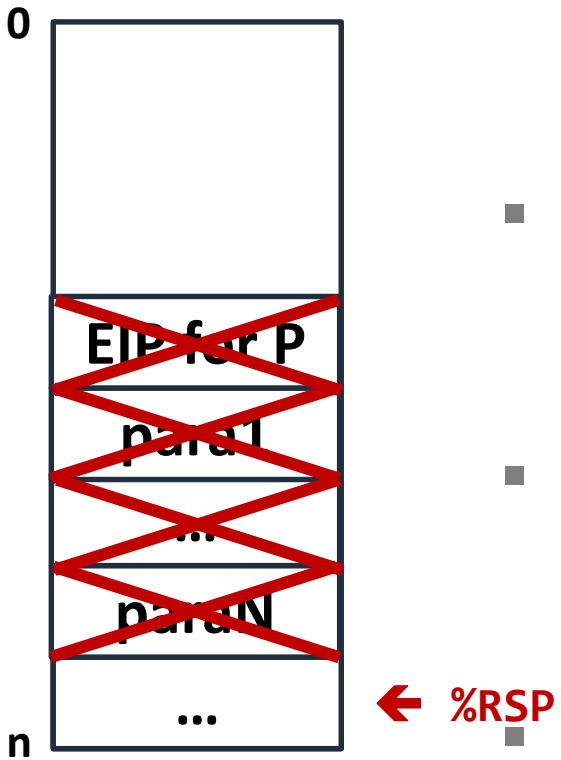


- We start by pushing the parameters on the stack
- We begin with the last parameter
- We call the function and return eventually

➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

Using the Stack

```
P:      ...
        push paraN
        ...
        push para1
        call foo
        add $x,%esp
```



- We start by pushing the parameters on the stack
- We begin with the last parameter
- We call the function and return eventually
- Clean up the stack
  - x=#Parameters*wordlength

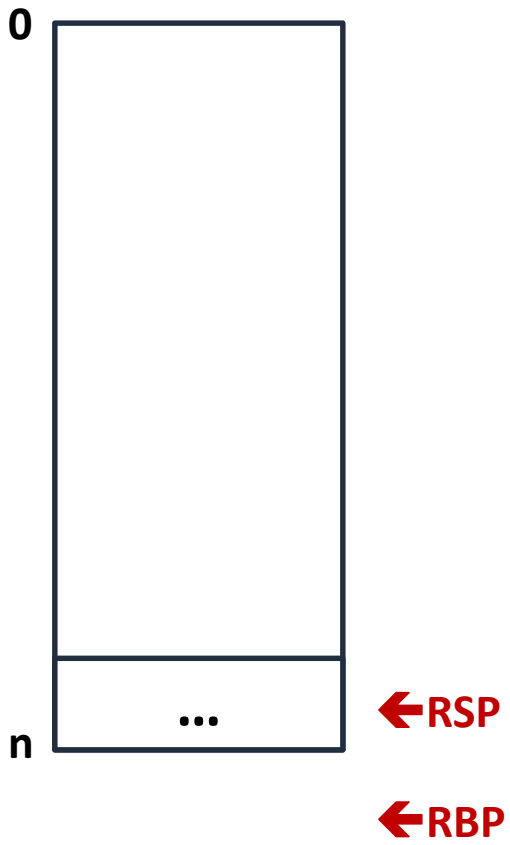
➢ Jennifer Rexford, Lecture Introduction to Programming Systems, Princeton University

Is That All?

- We still have some Problems left
  - We want to access the variables in the called function
  - We want to use the stack in the function as well
- Introduction of the Stack Base Pointer (**`%rbp`**)
- Always points to the base of the stack for that function
- We save the old value of **`%rbp`** at the start of the function and restore it at the end
- Variables are accessed using the **`%rbp`**

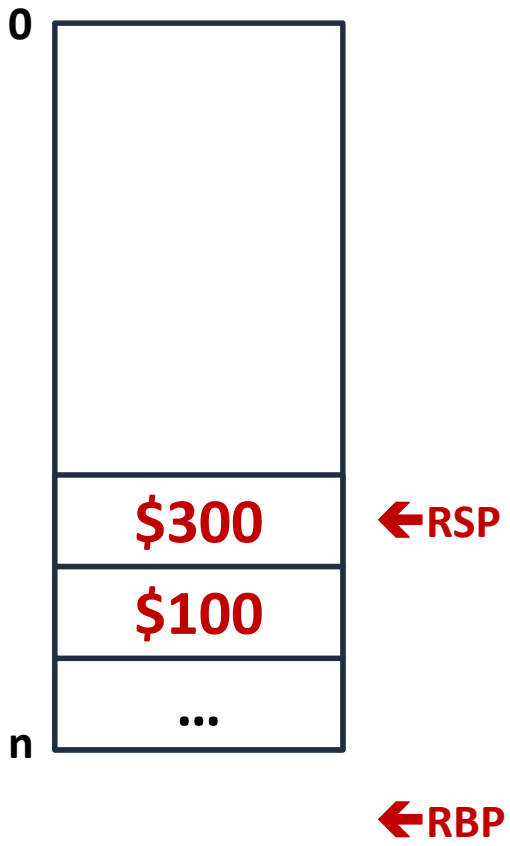
Using the Stack

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



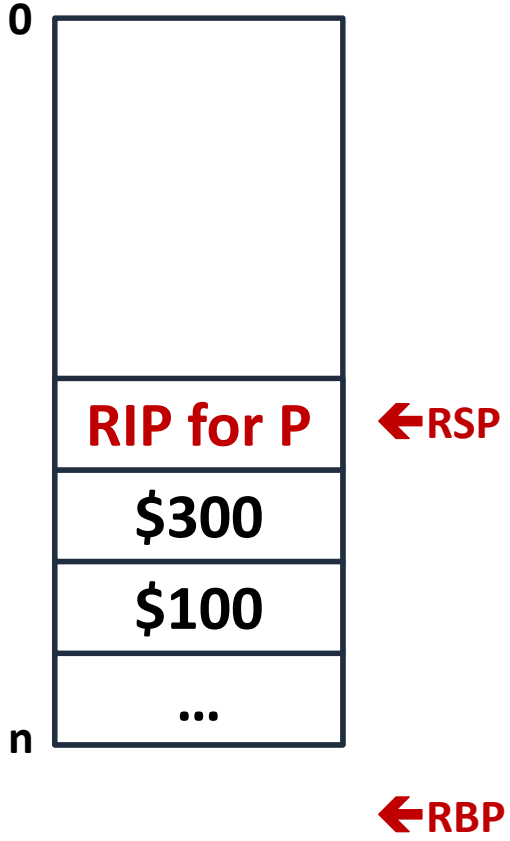
Using the Stack

```
P:      ...
        push $100
        push $300
      ➔ call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



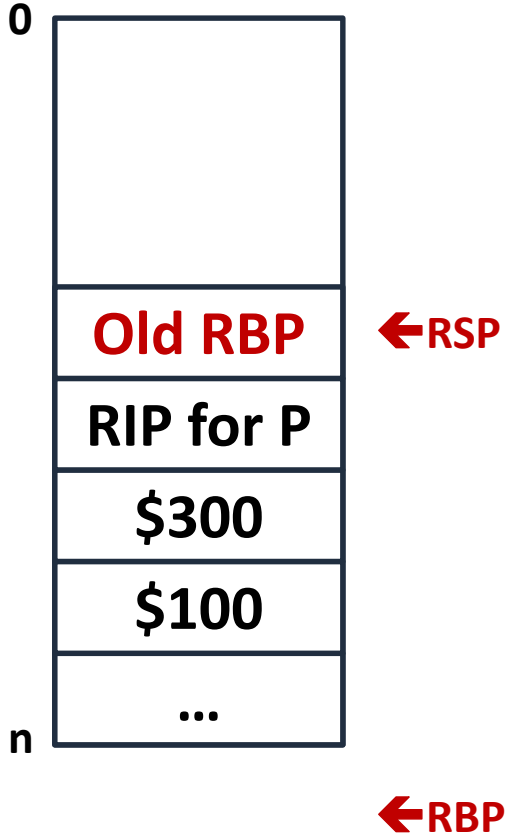
Using the Stack

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



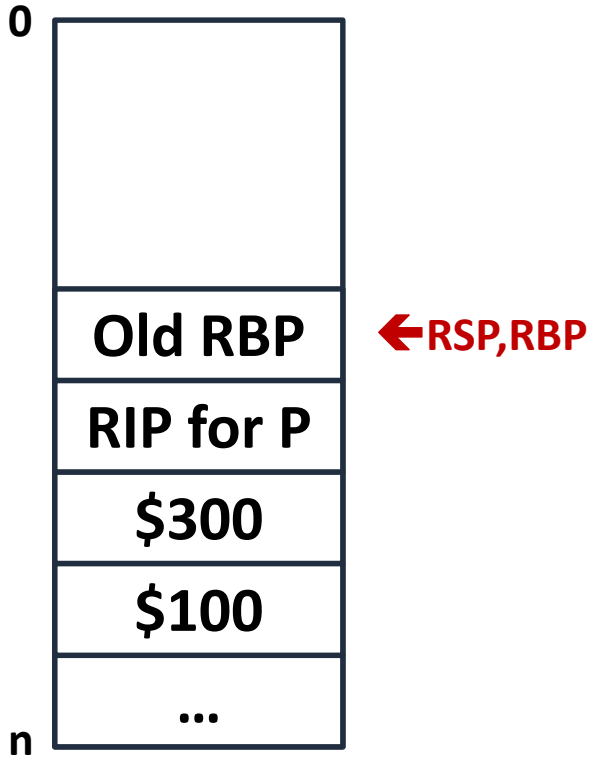
Using the Stack

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
     ➔  mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



Using the Stack

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
      ➔ sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

      ➔ mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```

Making Space for Local Variables

- We just made some space for local variables (in this case for 2) by decreasing the stack pointer
- After this we can use the stack as we like, push and pop elements
- The local variables can easily be accessed by using the base pointer:
  - Variable 1: `mov -8(%rbp),%rax`
  - Variable 2: `mov -16(%rbp),%rax`

Making Space for Local Variables

- We just made some space for local variables (in this case for 2) by decreasing the stack pointer

- After this we can use the stack as we like, push and pop elements

- The local variables can easily be accessed by using the base pointer:
  - Variable 1: **`mov -8(%rbp),%rax`**
  - Variable 2: **`mov -16(%rbp),%rax`**

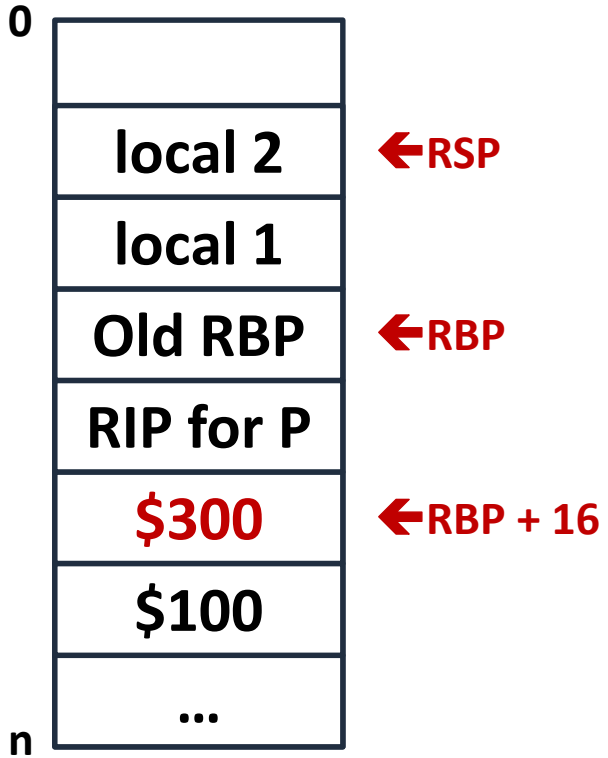
- Same thing for the parameters:
  - Parameter 1: **`mov 16(%rbp),%rax`**
  - Parameter 2: **`mov 24(%rbp),%rax`**

Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

      ➔ mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```

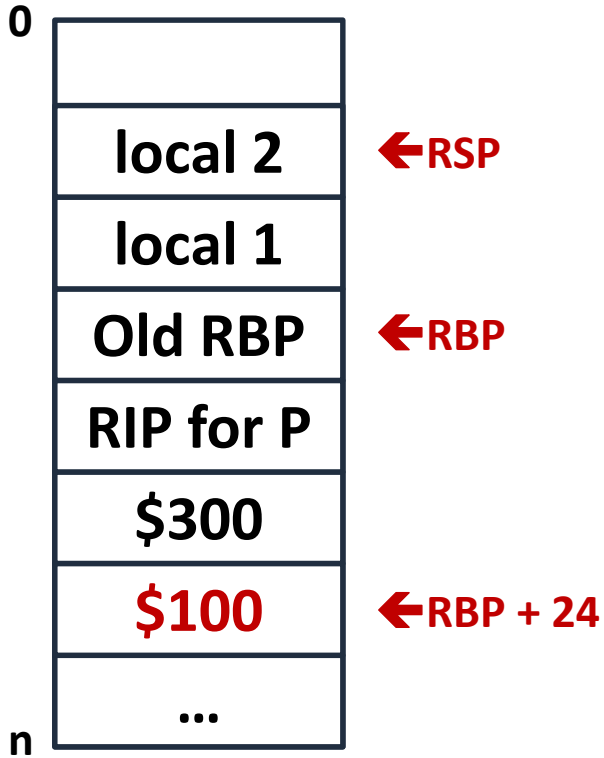
Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
      ➔ add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



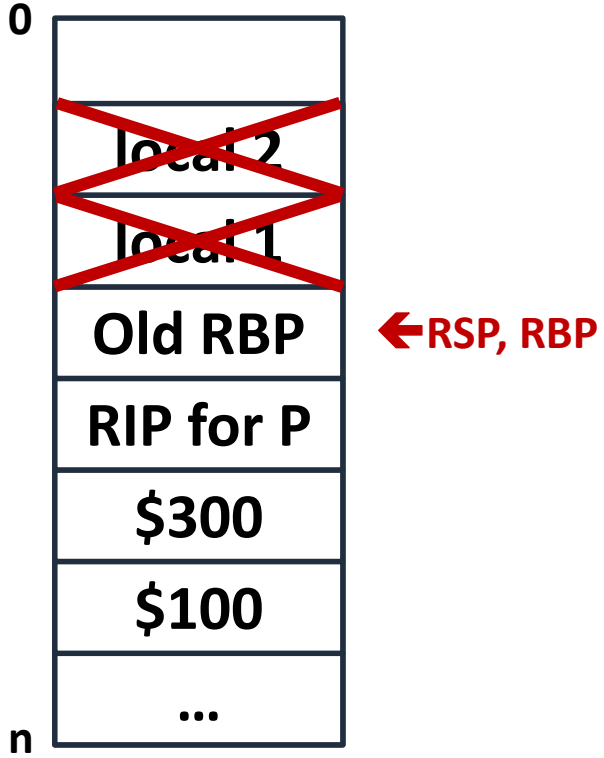
Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

      ➔ mov %rbp,%rsp
        pop %rbp
        ret
```



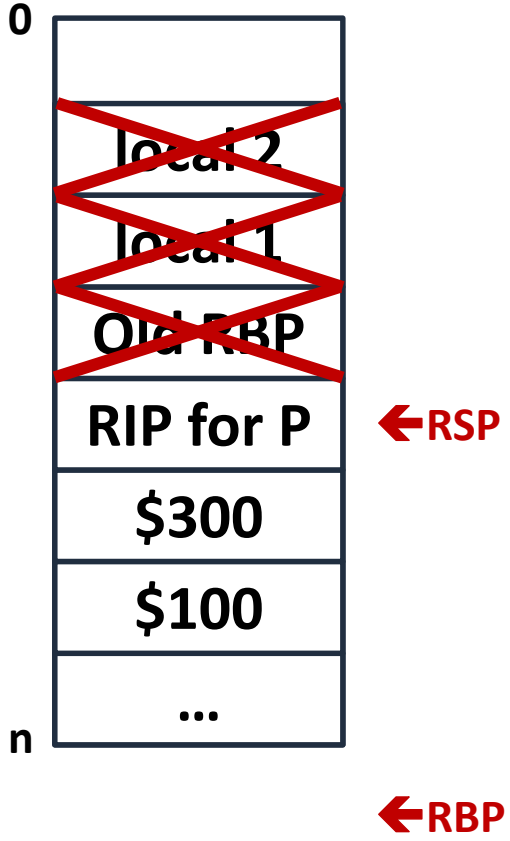
Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



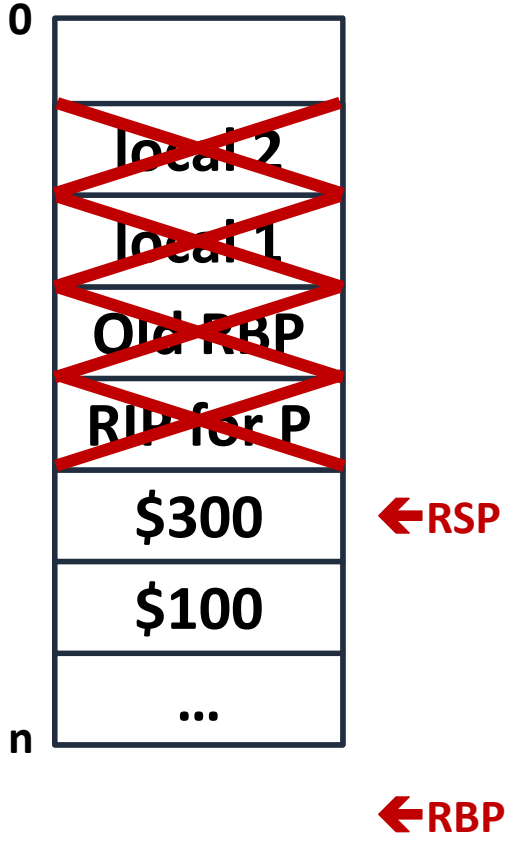
Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
      ➔ ret
```



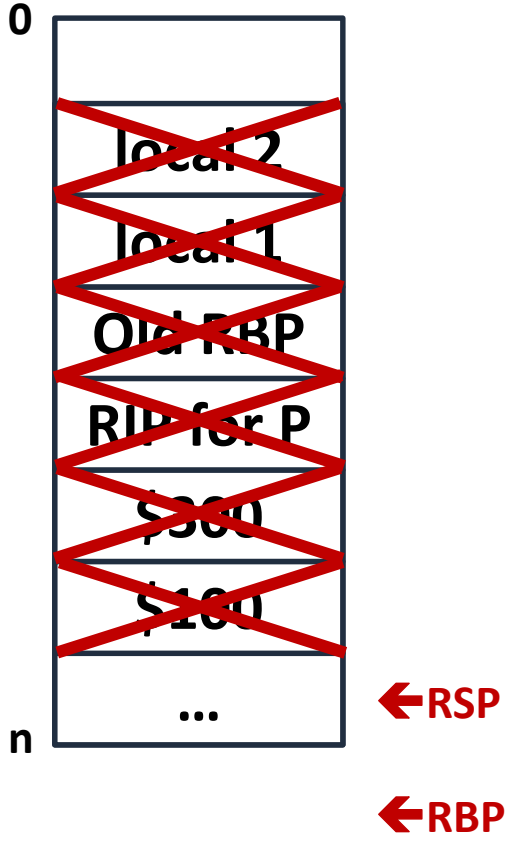
Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
      ➔ add $16,%rsp
        ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



Making Space for Local Variables

```
P:      ...
        push $100
        push $300
        call foo
        add $16,%rsp
     ➔  ...

foo:    push %rbp
        mov %rsp,%rbp
        sub $16,%rsp

        mov 16(%rbp),%rax
        add 24(%rbp),%rax

        mov %rbp,%rsp
        pop %rbp
        ret
```



Prolog and Epilog

```
push %rbp
mov %rsp,%rbp
sub $x,%rsp
```

- Is called the **prolog**. Making space for local variables is optional

```
mov %rbp,%rsp
pop %rbp
ret
```

- Is called the **epilog**.