6.110 Computer Language Engineering

Recitation 4: x86 Introduction

March 1st, 2024

Weekly Updates ←

x86 Quickstart

Weekly updates

- Everyone should already have teams
- Weekly miniquiz and check-in released, due Thursday, March 7
- Project phase 2 is due **Friday, March 8**
- Quiz 1 is on **Friday, March 15**
  - Covers up to Codegen lectures
  - Practice material will be posted soon
  - Quiz review during re-lecture on March 13

Coming up soon... **Week 5**

Mon 3/4	Tue 3/5	Wed 3/6	Thu 3/7	Fri 3/8
Lecture Codegen	**Lecture**	**Lecture (?)**	**Lecture (?)**	**Recitation** CFGs, More Codegen
		Re-lecture Codegen	**Due:** Mini-quiz, weekly check-in	**Due:** **Project phase 2**

Favorite Whitespace

Weekly Updates

x86 Quickstart ←

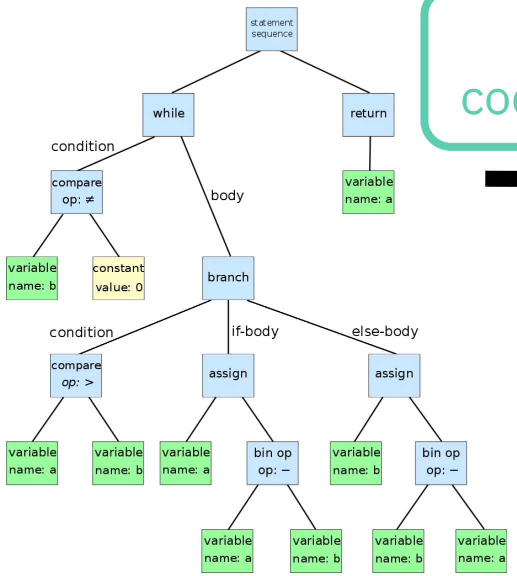
Project overview

```
import printf;

void main() {
…
```

Decaf source file

Phase 1. Does it have the right structure? (syntax)

Phase 2. Does it make sense? (semantics)



Internal representation

Phase 3
code generation

```
push %rbp
mov  %rsp, %rbp
…
```

x86-64 assembly

Why Now?

- For phase 2 you are asked to implement a high-level intermediate representation (IR) for semantic checking
- You may find that you will want to create an even lower-level IR, or a Control Flow Graph representation
- The **design of your IR** needs to be informed by the limitations of your code generator and x86 assembly

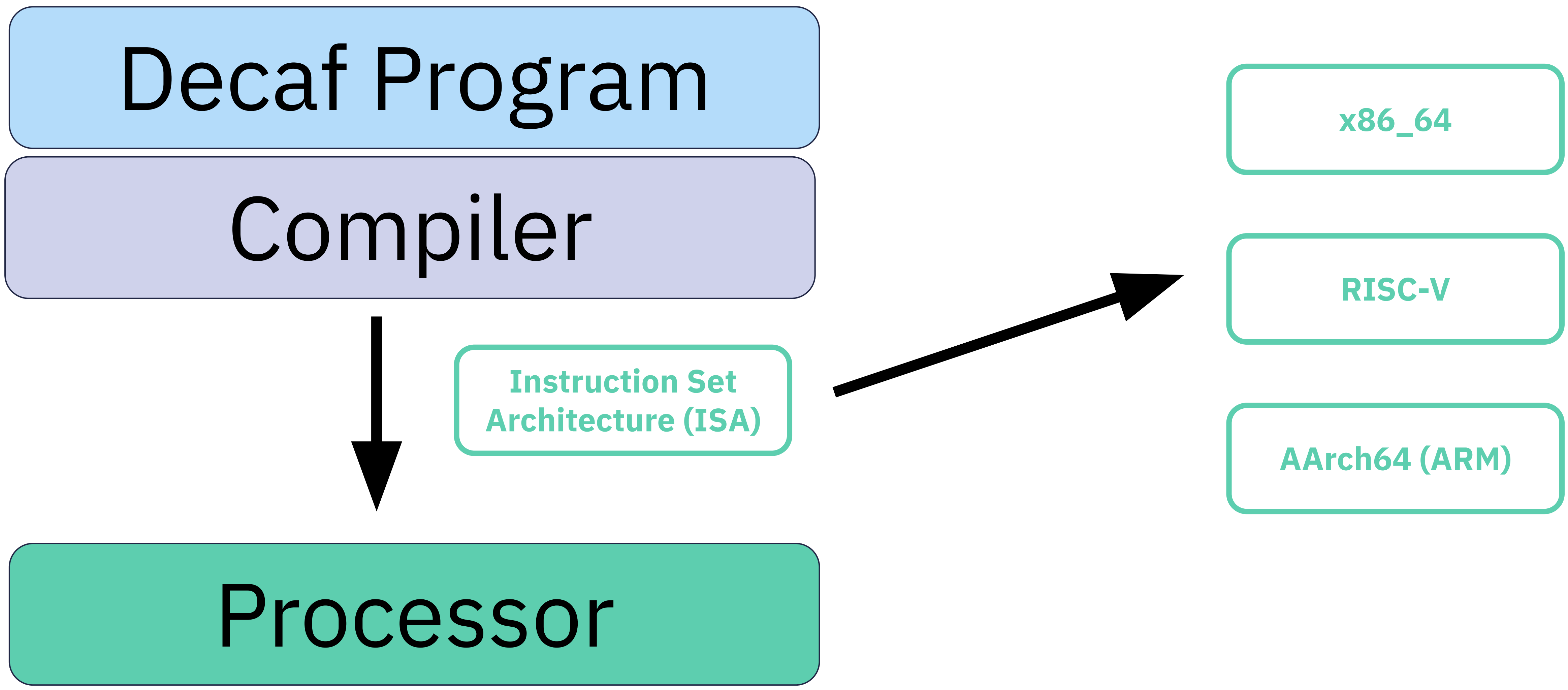
x86 Assembly

- Low-level programming language used to communicate with hardware
- Can do (mostly) what you want, but there's no safety net
- A reminder that a processor a *digital* circuit, and hardware is required to perform operations (6.004)

x86 ISA

- An ISA is the set of instructions that software can issue to hardware **implementations** (such as a CPU)
- Standardized by various hardware manufacturers
- x86 is **old**, the first version (16-bit) was created in **1978**
- Widely adopted despite its age

x86 Assembly



Coming from RISC-V…

- x86 is considered a **CISC** (complex instruction set computer) ISA

- It has **considerably** more instruction complexity/diversity than **RISC** (reduced instruction set computer) ISAs

- As such, you will be able to perform more complex operations in one assembly instruction than in RISC-V

Another note about x86

- x86 has two syntax styles
- Intel syntax and AT&T/NASM syntax
- We’ll use AT&T syntax because it is the default in Linux
  - The chief difference is the ordering of the operands
  - Keep this in mind if you consult Intel manuals
    (flip the operands in your head)

Decaf	**AT&T Syntax**	**Intel Syntax**
int a = 100; int b = a;	movq $100, -8(%rbp) movq -8(%rbp), -16(%rbp)	mov [rbp-8], 100 mov [rbp-16], [rbp-8]

A Tour Through x86

- In general, instructions have the following format:

instr	**ret**
instr argument	**call** function
instr src, dst	**addq** $10, %rax
instr aux, src, dst	**imulq** $2, %rcx, %rdx

Registers

- 16 registers available, some with “special” uses!
- %rax, %rdx used in arithmetic operations
- %rbp, %rsp (base pointer, stack pointer)
- 10 are caller-save, 6 are callee-save

Registers

- Can be further operated on as a 32 bit register, 16 bit register, or two 8 bit registers
- Integers in Decaf are 64-bit, we will usually use the entire register but GCC/Clang may optimize register usage

<table>
<tr><td>64-bit</td><td colspan="4">rax</td></tr>
<tr><td>32-bit</td><td></td><td colspan="3">eax</td></tr>
<tr><td>16-bit</td><td colspan="2"></td><td colspan="2">ax</td></tr>
<tr><td>8-bit</td><td colspan="2"></td><td>ah</td><td>al</td></tr>
</table>

Calling Conventions

- Some registers are *caller-save* registers, which means that you must save them before a **call** instruction
- Other registers are *callee-save* registers, which means that you do *not* need to save them before a **call** instruction
- Useful to optimize how you allocate registers (phase 5), but for now, focus on a working compiler

Arguments

- First 6 arguments are passed in **registers**
- %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Any further arguments are passed on the stack
- This is a convention. You are the compiler-writer, do what you want (except when calling external functions)

Let’s Talk Instructions

Broadly, there are a few categories of instructions:

- Data transfer
- Control flow
- Arithmetic/Logic/Shift

Data Transfer

In general, you can transfer data:

- Between two registers (fastest)
- From an immediate to a memory location or register
- Between a register and a memory location (slowest)

Note that virtually **no instructions allow memory locations as both operands**

Examples

```
movq    $1, %rax             (move 1 to rax)
movq    %rax, %rcx           (move from rax to rcx)
movq    -8(%rbp), %rax       (move from rbp-8 to rax)
movq    %rax, -8(%rsp)       (move from rax to rbp-8)

movq    -8(%rbp), -16(%rbp) (illegal)
```

Performance Considerations

There are multiple places to store variables and data:

- Globally (slowest)
- On the stack
- In registers (fastest)

Considerations for your IR

- How will you represent variables/arrays? How will you assign them to be global or on the stack?
  - Eventually, you will want certain variables in registers (phase 5), how will you handle this?
- How will you represent constants? On the stack or globally?

Control Flow

Differs significantly from RISC-V

- There are no instructions that do a compare and jump in one instruction:
  - **jmp** - unconditional jump
  - **je/jl/jle/jg/jge/jne** - examples of conditional jumps
- You must execute the **cmpq** instruction to set a special "flag" register
- This flag register determines the behavior of the various jump instructions

Flag Register

- A 32 bit EFLAGS register is used to store state about the CPU (and the result of certain math operations)
- Many of these are used for determining whether jumps, conditional moves, conditional set bytes will execute
- **Most arithmetic or logic instructions will clobber (reset) these flag registers**

Flag Register

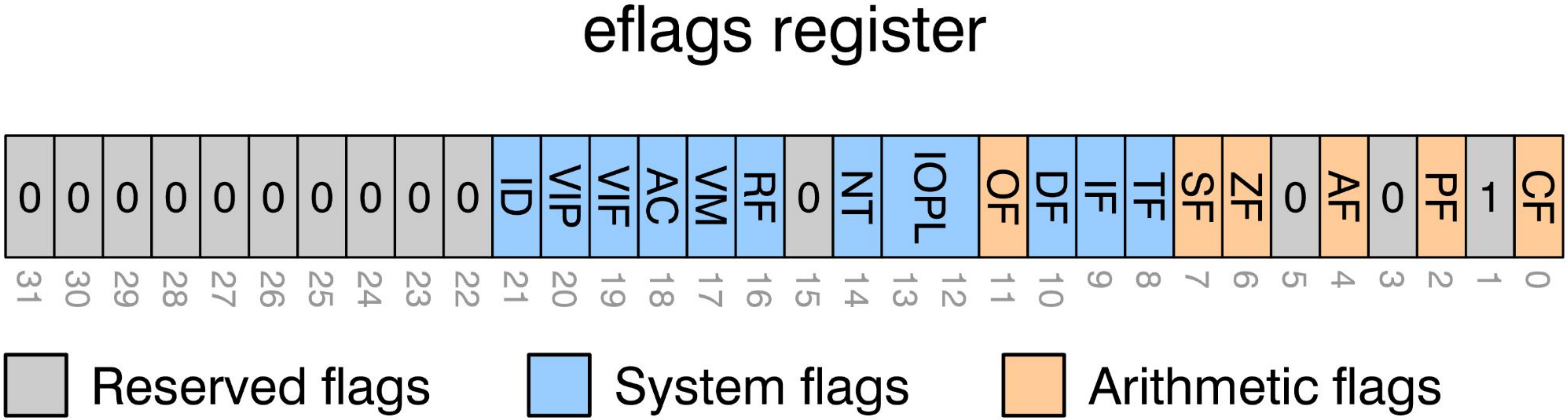


Image Credit: Source

Example

```
int a = 1;
int b = 0;
if (a < 5) {
  b = 1;
} else {
  b = 2;
}
```

Example

```c
int a = 1;
int b = 0;
if (a < 5) {
  b = 1;
} else {
  b = 2;
}
```

```
movq      $1, %rax
movq      $0, %rcx
cmpq      $5, %rax
jl        _if_body
jge       _else_body
_if_body:
  movq    $1, %rcx
  jmp     _exit
_else_body:
  movq    $2, %rcx
_exit:
  (...)
```

Example

```c
int a = 1;
int b = 0;
if (a < 5) {
  b = 1;
} else {
  b = 2;
}
```

```asm
movq        $1, %rax
movq        $0, %rcx
cmpq        $5, %rax
jl          _if_body
jge         _else_body
_if_body:
    movq    $1, %rcx
    jmp     _exit
_else_body:
    movq    $2, %rcx
_exit:
    (...)
```

Example

```c
int a = 1;
int b = 0;
if (a < 5) {
  b = 1;
} else {
  b = 2;
}
```

```asm
movq        $1, %rax
movq        $0, %rcx
cmpq        $5, %rax
jl          _if_body
jge         _else_body
_if_body:
  movq      $1, %rcx
  jmp       _exit
_else_body:
  movq      $2, %rcx
_exit:
  (...)
```

Example

```
int a = 1;
int b = 0;
if (a < 5) {
  b = 1;
} else {
  b = 2;
}
```

```
movq      $1, %rax
movq      $0, %rcx
cmpq      $5, %rax
~~jl        _if_body~~
jge       _else_body
~~_if_body:~~
  movq    $1, %rcx
  jmp     _exit
_else_body:
  movq    $2, %rcx
_exit:
  (...)
```

Considerations for your IR

- How might you structure your IR so that you can accommodate this **cmp** before **jump** requirement?

- How might you take advantage of fallthrough? Think about how you would make blocks easy to move around in your representation

  Hint: This will also be important for phase 4!

Doing Math

- Think like a circuit designer (6.004)
  - How might you accomplish math operations?

- Which operations are expensive?

  addq - **1 cycle**

  subq - **1 cycle**

  imulq - **3 cycles**

  idivq - **42-95 cycles (yikes!)**

  (on Skylake-X CPUs)

Doing Math

- Some operations **require** operands to be placed in specific registers
- **idivq** takes the 128-bit value stored in `rdx:rax`, divides it by the argument register
- The quotient is placed in `rax`, the remainder in `rdx`
  - This destroys whatever was stored there!
- Most math clobbers flag registers

Doing Math

- Some multiplications and divisions can be made cheap if they are by powers of 2
  - **shl/sar** only takes 1 cycle!
- You cannot perform complex operations like 1 + (2 * 3)
  - Must linearize the operation (more about this next recitation)

$t_1 = 2 * 3$
$t_2 = 1 + t_1$

Considerations for your IR

- How might you design a low-level IR that can be easily translated to x86 assembly?

- How might you represent complex operations?

Representing Functions

- Functions are “fake” in assembly - only labels/jumps
- What does it mean to “allocate” space on the stack?
- Byte alignment

Example

```
int add_2(int a) {
   int b = 2;
   return a + b;
}
```

```c
int add_2(int a) {
    int b = 2;
    return a + b;
}
```

```asm
add_2:
    push    %rbp
    movq    %rsp, %rbp
    subq    $8, %rsp

    movq    $2, -8(%rbp)
```

```c
int add_2(int a) {
    int b = 2;
    return a + b;
}
```

```asm
add_2:
    push    %rbp
    movq    %rsp, %rbp
    subq    $8, %rsp

    movq    $2, -8(%rbp)

    addq    -8(%rbp), %rdi
    movq    %rdi, %rax
```

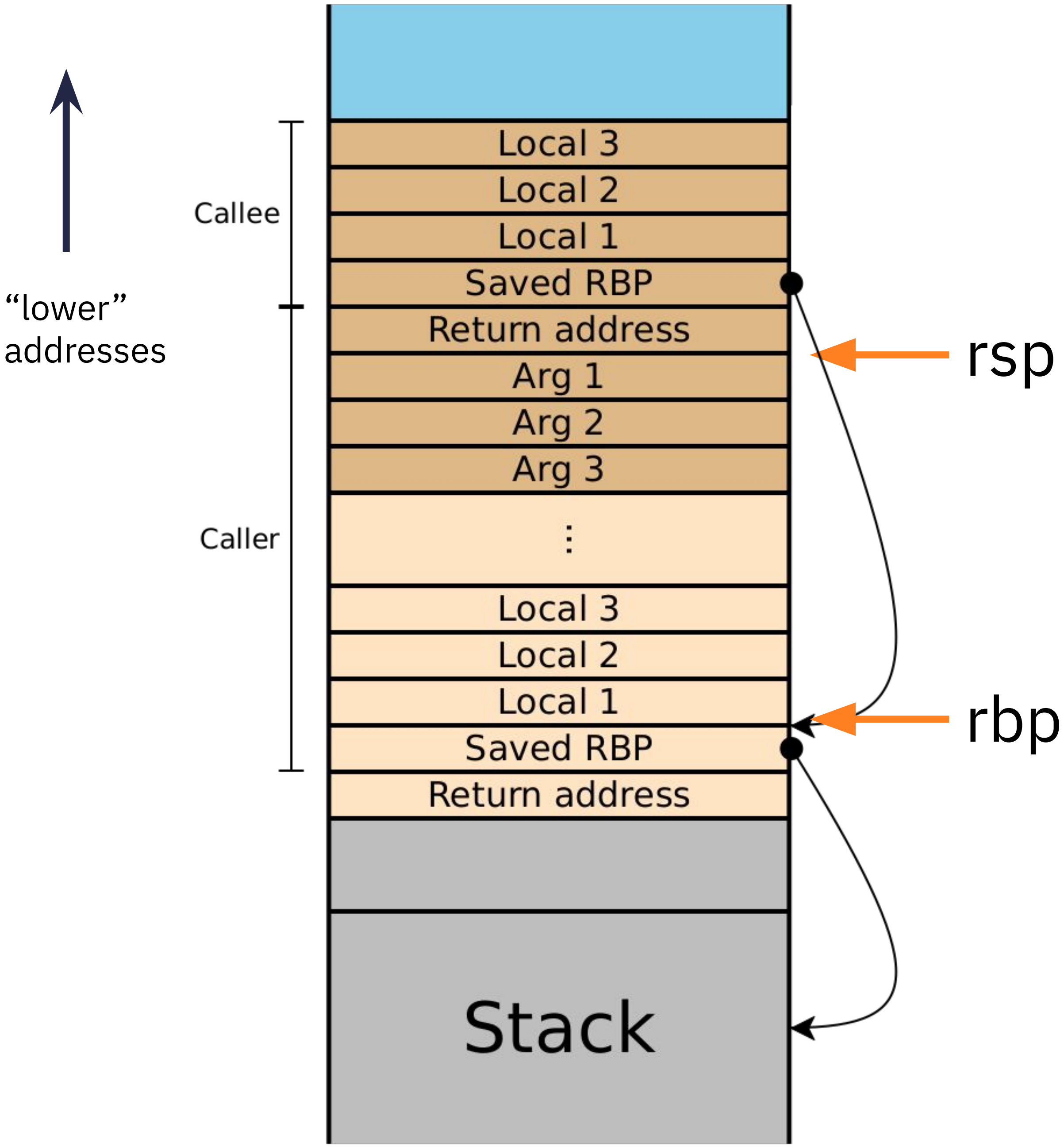
```c
void add_2(int a) {
    int b = 2;
    return a + b;
}
```

```
add_2:
    push    %rbp
    movq    %rsp, %rbp
    subq    $8, %rsp

    movq    $2, -8(%rbp)

    addq    -8(%rbp), %rdi
    movq    %rdi, %rax

    addq    $8, %rsp
    movq    %rbp, %rsp
    pop     %rbp
    ret
```

```c
void add_2(int a) {
    int b = 2;
    return a + b;
}

void main(){
    int c = 0;
    c = add_2(10);
}
```

```
add_2:
    push    %rbp
    movq    %rsp, %rbp
    subq    $8, %rsp

    movq    $2, -8(%rbp)

    addq    -8(%rbp), %rdi
    movq    %rdi, %rax

    addq    $8, %rsp
    movq    %rbp, %rsp
    pop     %rbp
    ret
```

```
main:
    push    %rbp
    movq    %rsp, %rbp
    subq    $8, %rsp

    movq    $0, -8(%rbp)

    movq    $10, %rdi
    call    add_2
    movq    %rax, -8(%rbp)

    addq    $8, %rsp
    movq    %rbp, %rsp
    pop     %rbp
    ret
```



call add_1

1. The return address is pushed (%rip)
2. %rip is set to address of procedure

```
add_2:
```

push %rbp

1. %rsp = %rsp - 8
2. copies %rbp to address in %rsp

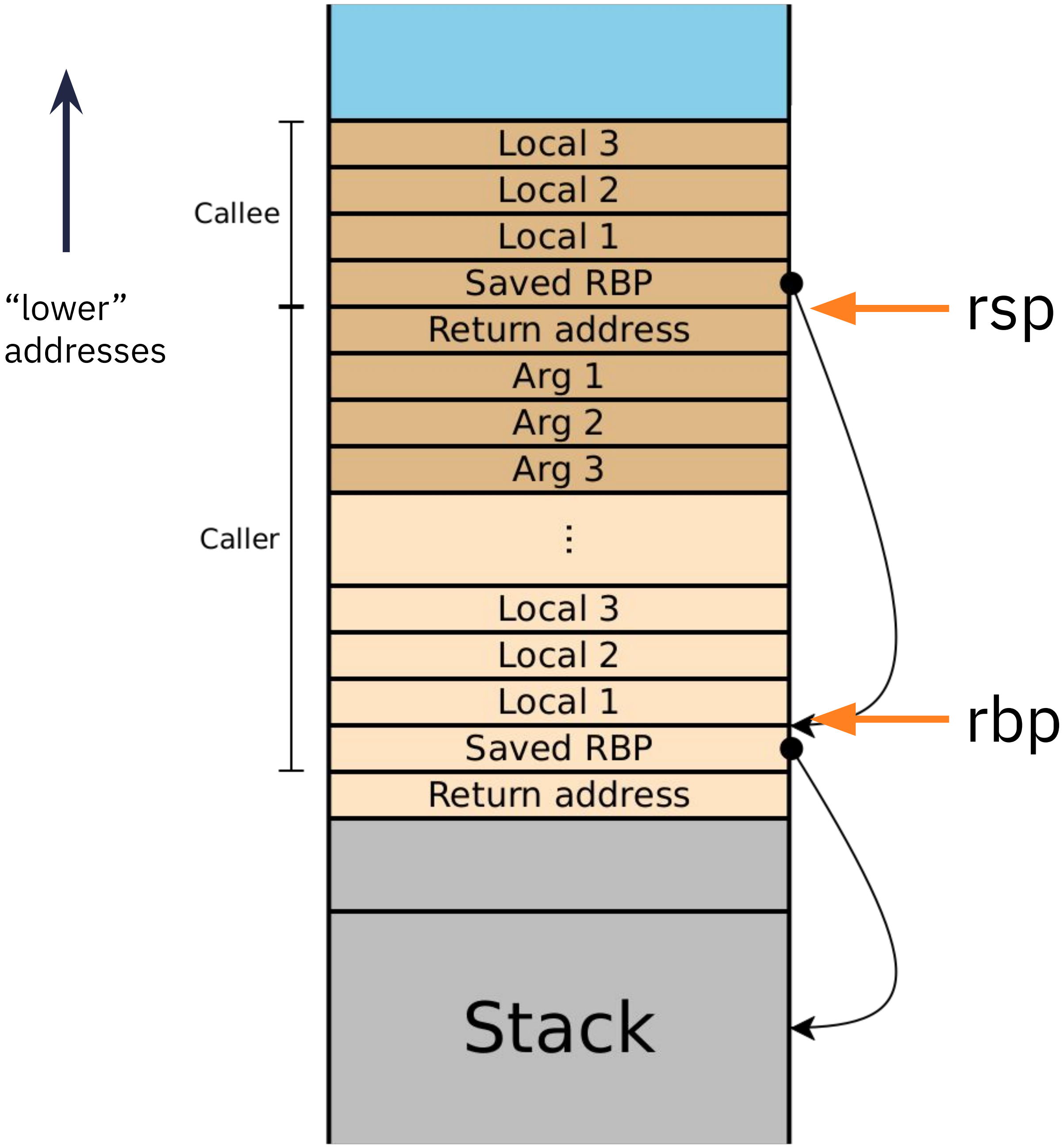
movq %rsp, %rbp

1. make %rsp equal %rbp

subq $24, %rsp

1. allocates 24 bytes

Image Credit: NEU



call add_1

1. The return address is pushed (%rip)
2. %rip is set to address of procedure

`add_2:`

push %rbp

1. %rsp = %rsp - 8
2. copies %rbp to address in %rsp

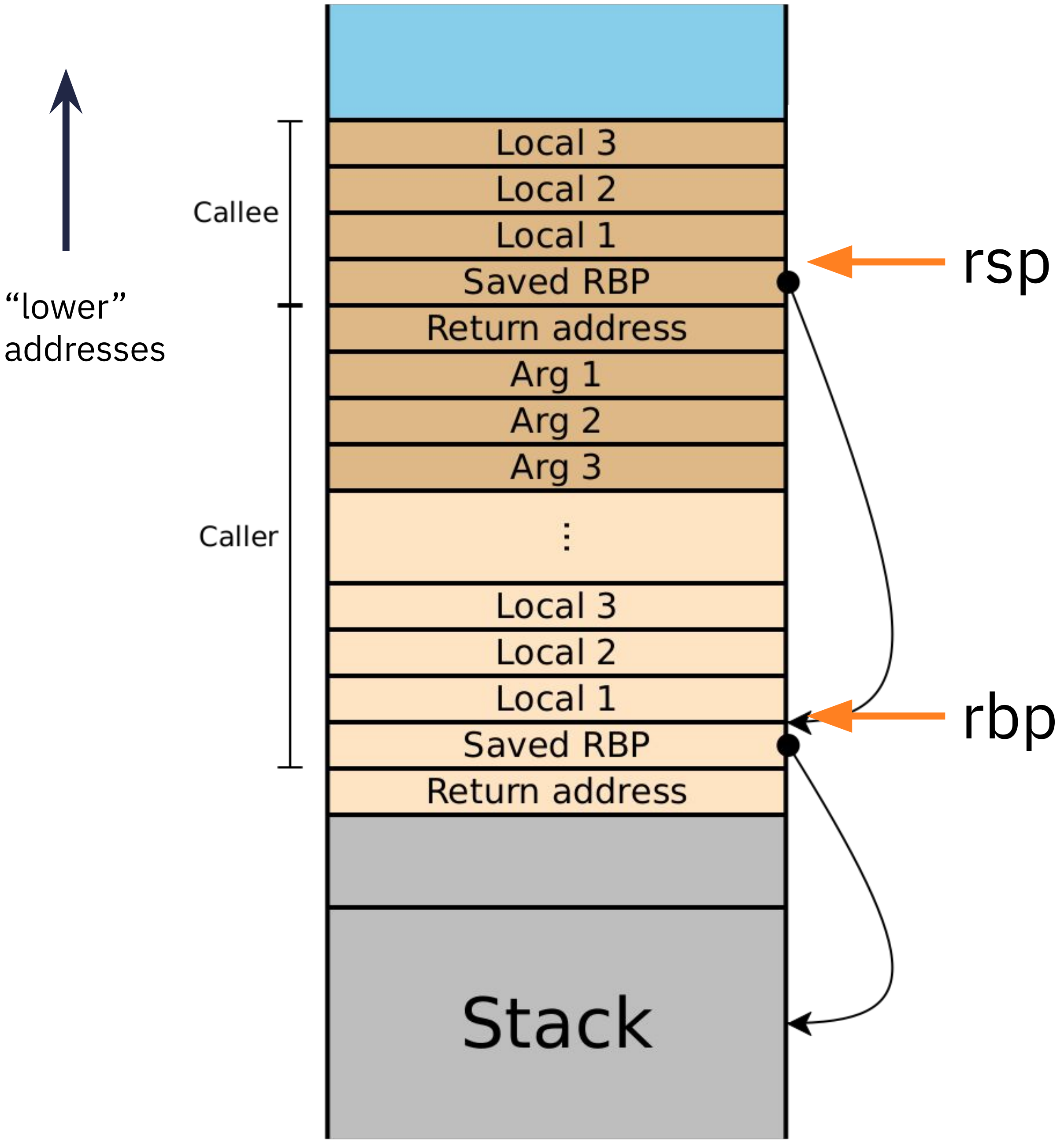
movq %rsp, %rbp

1. make %rsp equal %rbp

subq $24, %rsp

1. allocates 24 bytes

Image Credit: NEU



call add_1

1. The return address is pushed (%rip)
2. %rip is set to address of procedure

```
add_2:
```

push %rbp

1. %rsp = %rsp - 8
2. copies %rbp to address in %rsp

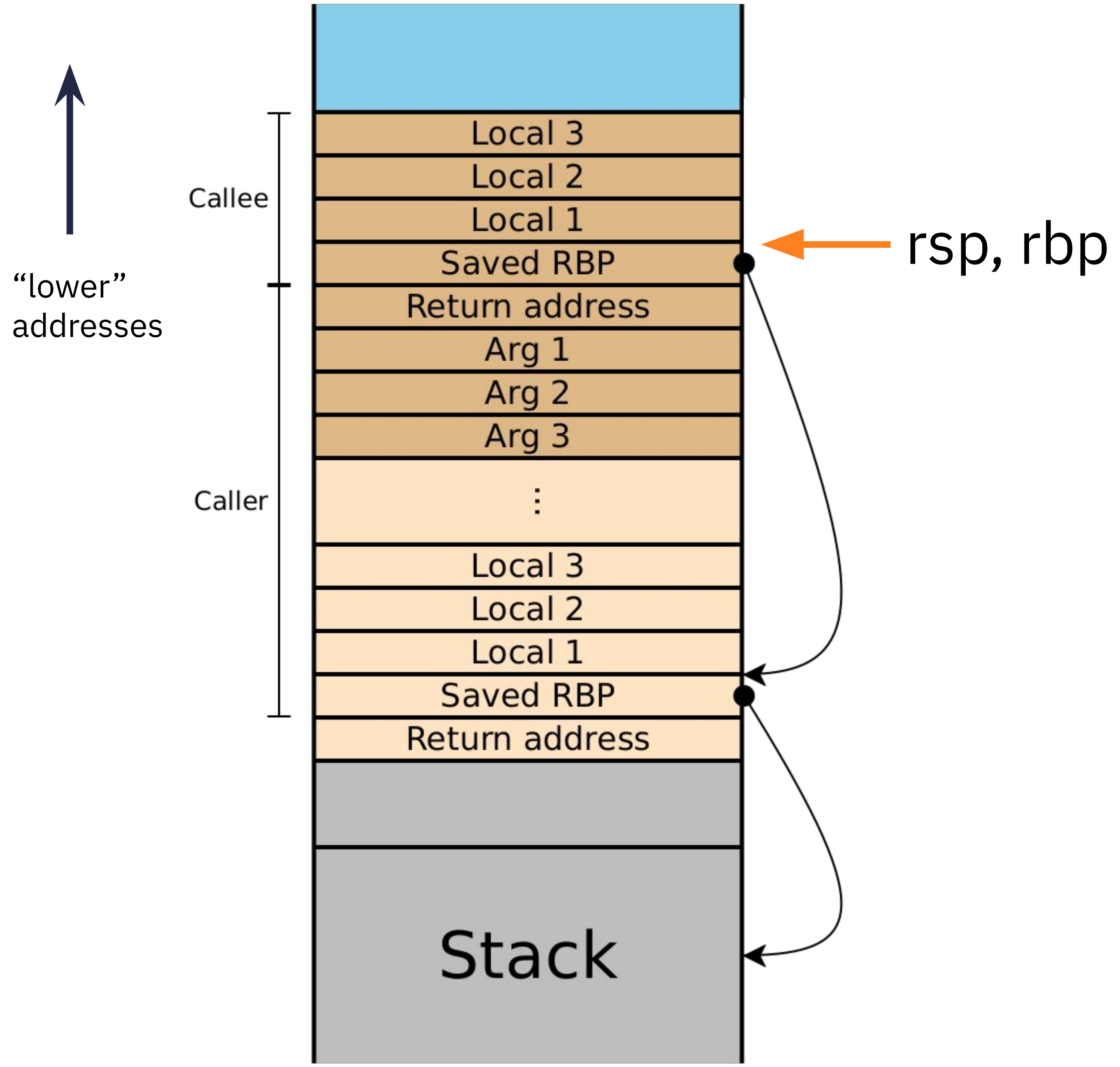
movq %rsp, %rbp

1. make %rsp equal %rbp

subq $24, %rsp

1. allocates 24 bytes

Image Credit: NEU




```
call    add_1
```

1. The return address is pushed (%rip)
2. %rip is set to address of procedure

```
add_2:

    push    %rbp
```

1. %rsp = %rsp - 8
2. copies %rbp to address in %rsp

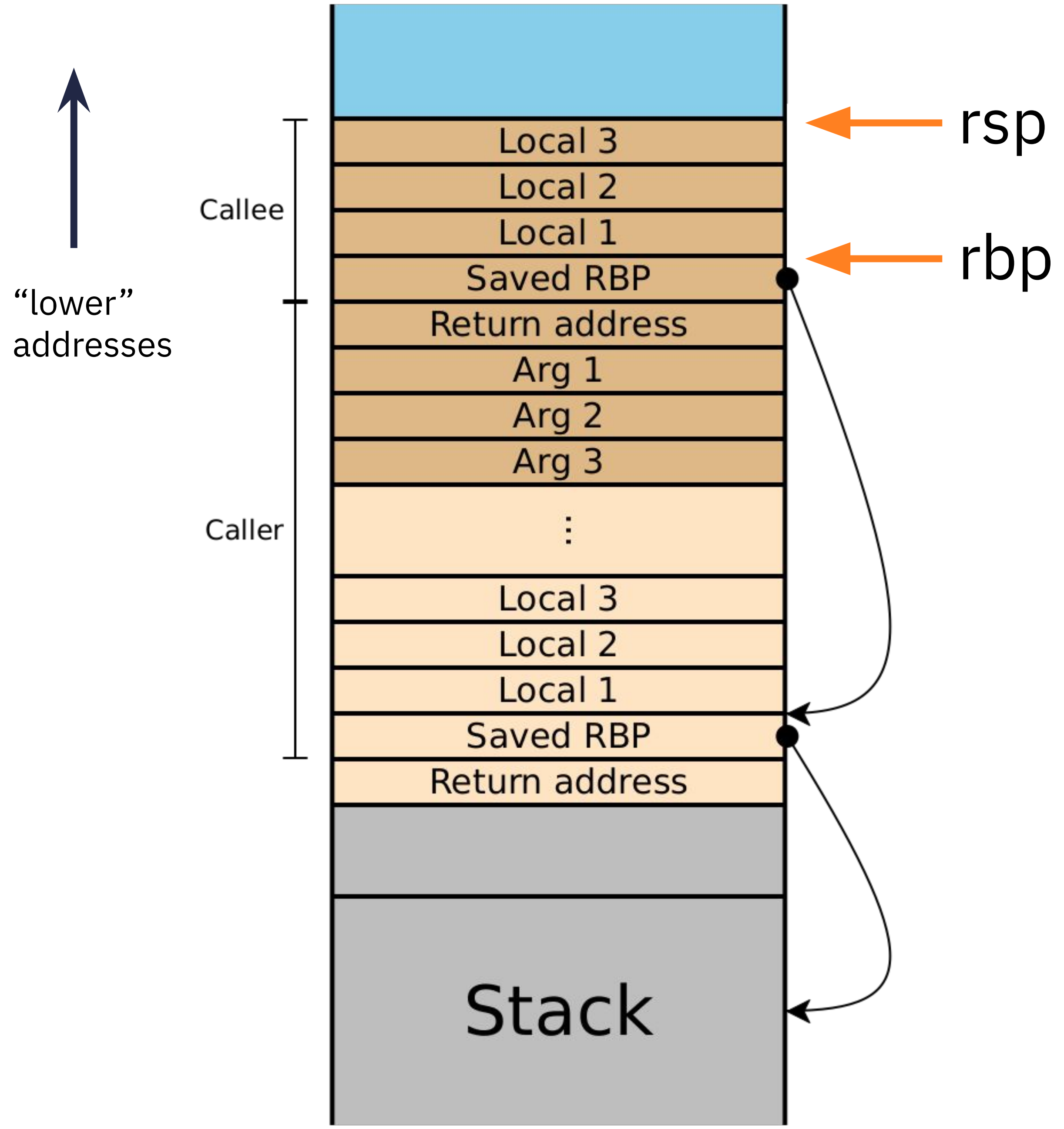
```
    movq    %rsp, %rbp
```

1. make %rsp equal %rbp

```
    subq    $24, %rsp
```

1. allocates 24 bytes

Image Credit: NEU




```
call    add_1
```

1. The return address is pushed (%rip)
2. %rip is set to address of procedure

```
add_2:
    push    %rbp
```

1. %rsp = %rsp - 8
2. copies %rbp to address in %rsp

```
    movq    %rsp, %rbp
```

1. make %rsp equal %rbp

```
    subq    $24, %rsp
```

1. allocates 24 bytes

Image Credit: NEU

Byte Alignment

- When calling external functions, your stack must be 16-byte aligned

- What to do when they are misaligned?
  - Push a `$0` to the stack
  - You can optimize for this (phase 5)

High-Level IR

- You will probably want at least two versions of your IR (high and low level)

- High-level IR
  - Needed if you used a hacked grammar or a parser generator to get the proper structure
  - Not needed if your parser is hand-written and well-designed

- Perform your semantics check at this stage (phase 2)

Low-Level IR

- This is closer to the assembly. Commonly in the form of a control flow graph for ease of performing optimizations
- Have to consider all of the things we mentioned this recitation!
- Based on x86, make informed decisions on how this low-level IR can be structured

Low-Level IR

- Start early! This is a representation that is farther from the original Decaf.
- Don’t be afraid to refactor! You will make lots of changes to this representation as you implement phases 3/4/5
- Design your high-level IR to be easy to convert (do not try to generate low-level IR based on an parse tree alone)

Phase 2/3

- At the end of phase 2, you will be fairly close to a working compiler
- If you start thinking about your low-level IR representation during phase 2, phase 3 will be much easier to implement
- For groups that want to attempt SSA, which we will cover in a coming recitation, this is a good time to start thinking about it

Latency Tables / x86 Reference

Felix Cloutier x86 Reference

Agner Fog Latency Table

Appendix: Example Program

```
import printf;

void main() {
   int i;
   for (i = 0; i < 30; i++) {
      printf(“%d\n”, i);
   }
   return;
}
```

```
import printf;

void main() {
    int i;
    for (i = 0; i < 30; i++) {
        printf("%d\n", i);
    }
    return;
}
```

```
.globl main
print_str:
    .string "%d\n"
    .align 16
main:
```

```
import printf;

void main() {
    int i;
    for (i = 0; i < 30; i++) {
        printf("%d\n", i);
    }
    return;
}
```

```
.globl main
print_str:
    .string "%d\n"
    .align 16
main:
    pushq     %rbp
    movq      %rsp, %rbp
    subq      $16, %rsp
```

```
import printf;

void main() {
    int i;
    for (i = 0; i < 30; i++) {
        printf("%d\n", i);
    }
    return;
}
```

```
.globl main
print_str:
    .string "%d\n"
    .align 16
main:
    pushq     %rbp
    movq      %rsp, %rbp
    subq      $16, %rsp
    movq      $0, -8(%rbp)
loop_body:
    leaq      print_str(%rip), %rdi
    movq      -8(%rbp), %rsi
    movq      $0, %rax
    call      printf
```

```
import printf;

void main() {
    int i;
    for (i = 0; i < 30; i++) {
        printf(“%d\n”, i);
    }
    return;
}
```

```
.globl main
print_str:
    .string “%d\n”
    .align 16
main:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movq     $0, -8(%rbp)
loop_body:
    leaq     print_str(%rip), %rdi
    movq     -8(%rbp), %rsi
    movq     $0, %rax
    call     printf
    addq     $1, -8(%rbp)
    cmpq     $30, -8(%rbp)
    jl       loop_body
```

```
import printf;

void main() {
    int i;
    for (i = 0; i < 30; i++) {
        printf("%d\n", i);
    }
    return;
}
```

```
.globl main
print_str:
        .string "%d\n"
        .align 16
main:
        pushq       %rbp
        movq        %rsp, %rbp
        subq        $16, %rsp
        movq        $0, -8(%rbp)
loop_body:
        leaq        print_str(%rip), %rdi
        movq        -8(%rbp), %rsi
        movq        $0, %rax
        call        printf
        addq        $1, -8(%rbp)
        cmpq        $30, -8(%rbp)
        jl          loop_body
addq $16, %rsp
movq %rbp, %rsp
popq %rbp
ret
```

Questions?

Good luck on phase 2!

Godbolt Example (time permitting)