Conditional Jump

Jump to different part of code if condition is true

	Condition	Description
`jmp`	`None`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`jl`	`(SF ^ OF)`	Less (Signed)
`jle`	`(SF ^ OF) | ZF`	Less or Equal (Signed)
`jg`	`~(SF ^ OF) & ~ZF`	Greater (Signed)
`jge`	`~(SF ^ OF)`	Greater or Equal (Signed)

`cmp a, b like computing a-b without setting destination`

Practicing Conditional Jumps

rdi	47
rsi	13

Consider each of the following segments of assembly code and indicate whether the jump will occur.

```
add rsi, rdi
je .L0
```

```
cmp rsi, rdi
jl .L0
```

```
sub rsi, rdi
jge .L0
```

```
test rdi, rdi
jne .L0
```

	Condition	Description
jmp	None	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
jl	(SF ^ OF)	Less (Signed)
jle	(SF ^ OF) | ZF	Less or Equal (Signed)
jg	~(SF ^ OF) & ~ZF	Greater (Signed)
jge	~(SF ^ OF)	Greater or Equal (Signed)

SF	Sign Flag (for signed)
ZF	Zero Flag
CF	Carry Flag (for unsigned)
OF	Overflow Flag (for signed)

Practicing Conditional Jumps

rdi	47
rsi	13

Consider each of the following segments of assembly code and indicate whether the jump will occur.

```
add rsi, rdi
je .L0
```

$13 + 47 \stackrel{?}{==} 0$

no jump

```
cmp rsi, rdi
jl .L0
```

$13 - 47 \stackrel{?}{<} 0$

jump

```
sub rsi, rdi
jge .L0
```

$13 - 47 \stackrel{?}{\geq} 0$

no jump

```
test rdi, rdi
jne .L0
```

$13 \,\&\, 13 \stackrel{?}{\neq} 0$

jump

	Condition	Description
jmp	None	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
jl	(SF ^ OF)	Less (Signed)
jle	(SF ^ OF) | ZF	Less or Equal (Signed)
jg	~(SF ^ OF) & ~ZF	Greater (Signed)
jge	~(SF ^ OF)	Greater or Equal (Signed)

SF	Sign Flag (for signed)
ZF	Zero Flag
CF	Carry Flag (for unsigned)
OF	Overflow Flag (for signed)

Conditional Branching

Register	Use
rdi	x
rsi	y
rax	result

What assembly instructions correspond with the highlighted code?

```
long absdiff(long x, long y){
  long result;

  if (x > y){
    result = x - y;
  } else {
    result = y - x;
  }

  return result;
}
```

```
absdiff:
   cmp     rdi, rsi
   jle     .L4
   mov     rax, rdi
   sub     rax, rsi
   ret

.L4  ; x-y <= 0
   mov     rax, rsi
   sub     rax, rdi
   ret
```

Address Computation Instruction

```
lea DEST, SRC ; load effective address
```

Computing an address without accessing the underlying data

```
p = &(x[i]);
```

Computing arithmetic expressions of the form x + k*y (k = 1, 2, 4, or 8)

```
y = x + i;
```

```
long m12(long x) {
  return x * 12;
}
```

Converted to ASM by compiler:

```
lea rax, [rdi + rdi*2]  ; rax = x + x * 2
sal rax, 2              ; rax = rax << 2
```

```
test:
  lea  rax, [rdi + rsi]
  add  rax, rdx
  cmp  rdi, -3
  jge  .L2
  cmp  rsi, rdx
  jge  .L3
  mov  rax, rdi
  imul rax, rsi
  ret
.L3:
  mov  rax, rsi
  imul rax, rdx
  ret
.L2
  cmp  rdi, 2
  jle  .L4
  mov  rax, rdi
  imul rax, rdx
.L4:
  rep
  ret
```

```
long test(long x, long y, long z){
  long val = __________;

  if(_______){

    if(_______){

      val = ______;

    } else {

      val = ______;

    }

  } else if (______){

    val = _______;

  }
  return val;
}
```

Register	Use
rdi	x
rsi	y
rdx	z
rax	val

```
test:
  lea   rax, [rdi + rsi]
  add   rax, rdx
  cmp   rdi, -3
  jge   .L2
  cmp   rsi, rdx
  jge   .L3
  mov   rax, rdi
  imul  rax, rsi
  ret
.L3:
  mov   rax, rsi
  imul  rax, rdx
  ret
.L2
  cmp   rdi, 2
  jle   .L4
  mov   rax, rdi
  imul  rax, rdx
.L4:
  rep
  ret
```

```
long test(long x, long y, long z){
  long val = x + y + z

  if(x < -3){

    if(y < z){

      val x * y;

    } else {

      val y * z;

    }

  } else if x > 2 {

    val x * z;

  }
  return val;
}
```

Register	Use
rdi	x
rsi	y
rdx	z
rax	val

Loops

Do-While, While, For

Do-while Loops

```
long bitcount(unsigned long x){
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

```
long bitcount(unsigned long x){
  long result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

```
   mov     rax, 0      ;  result = 0
.L2:                   ;  loop:
   mov     rdx, rdi    ;  t = x
   and     rdx, 1      ;  t = t & 0x1
   add     rax, rdx    ;  result += t
   shr     rdi, 1      ;  x >>= 1
   jne     .L2         ;  if (x) goto loop
   rep ret
```

Register	Use(s)
rdi	x
rax	result

While Loops

```
while (Condition) {
    Body
}
```

```
if (Condition){
    do {
        Body
    } while (Condition)
}
```

```
long bitcount(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Register	Use(s)
rdi	x
rax	result

```
  mov rax, 0
  jmp .L2

.L3:
  mov rdx, rdi
  and rdx, 1
  add rax, rdx
  shr 1, rdi

.L2:
  test rdi, rdi
  jne .L3
  rep ret
```

For loops

```
for (Init; Cond; Incr){
    Body
}
```

```c
long bitcount(unsigned long x) {
  long result;
  for (result = 0; x; x >>= 1)
    result += x & 0x1;
  return result;
}
```

```
  mov   rax, 0
  jmp   .L2

.L3:
  mov   rdx, rdi
  and   rdx, 1
  add   rax, rdx
  shr   rdi, 1

.L2:
  test   rdi, rdi
  jne   .L3
  rep
  ret
```

```
Init
while (Cond) {
    Body
    Incr
}
```

```
Init
if (Condition){
    do {
      Body
      Incr
    } while (Condition)
}
```

Register	Use(s)
rdi	Argument x
rax	result

Practice with Loops

Register	Use(s)
rdi	Argument val
rdx	Local i
rax	Local ret

```
func:
  mov rax, 0
  mov rdx, 0
  jmp L1

L0:
  add rax, rdx
  inc rdx

L1:
  cmp rdx, rdi
  jl L0
  ret
```

```
long func(long val){
  long ret = _______;
  long i;

  for(i = ____; __________; _____){

    ret = ________;

  }

  return ret;
}
```

Practice with Loops

Register	Use(s)
rdi	Argument val
rdx	Local i
rax	Local ret

```
func:
   mov rax, 0
   mov rdx, 0
   jmp L1

L0:
   add rax, rdx
   inc rdx

L1:
   cmp rdx, rdi
   jl L0
   ret
```

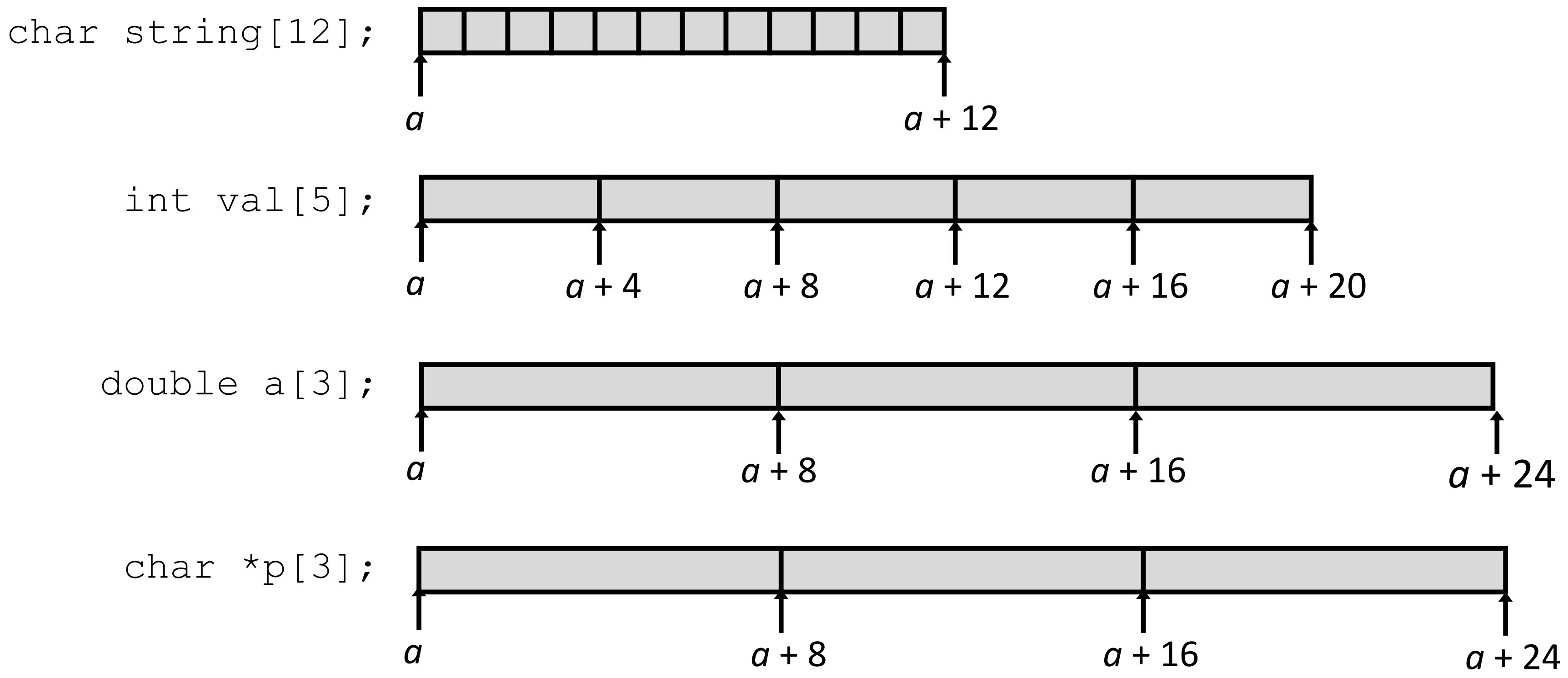
```
long func(long val){
  long ret = ___0___;
  long i;

  for(i = _0_; _i < val_; _i++_){

    ret = _ret + i_;

  }

  return ret;
}
```

Data

Arrays and Structs

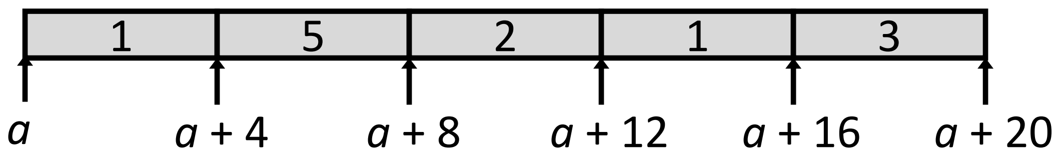
Array Allocation: `Type Name[Size];`

- Array of data type `Type` and size `Size`
- Contiguously allocated region of `Size * sizeof(Type)` bytes
- Identifier `Name` can be used as a pointer to array element 0



Practice with Array Access

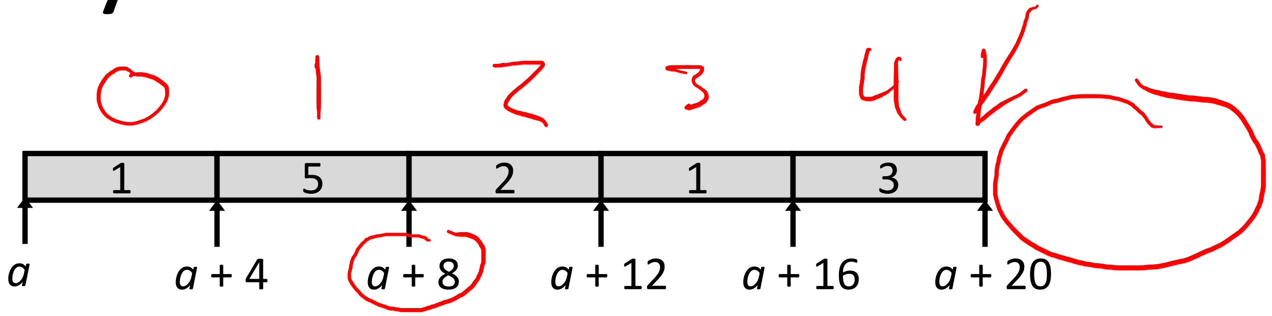
`int val[5];`



Code	Type	Value
`val[4]`		
`val`		
`val+1`		
`&val[2]`		
`val[5]`		
`*(val+1)`		
`val+i`		

Practice with Array Access

`int val[5];`



Code	Type	Value
`val[4]`	`int`	`3`
`val`	`int *`	`a`
`val+1`	`int *`	`a + 4`
`&val[2]`	`int *`	`a + 8`
`val[5]`	`int`	`??`
`*(val+1)`	`int`	`5`
`val+i`	`int *`	`a + 4 i`

val[1]

Practice Arrays and Loops

Variable	Register
z	
sum	
i	

```
array_loop:
  mov  esi, 0
  xor  eax, eax
  jmp  L2

L1:
  add  eax, [rdi + esi*4]
  inc  esi

L2:
  cmp  esi, 5
  jl   L1
  ret
```

```
int array_loop(int z[5]) {
  int sum = ______;

  int i;

  for(i = ___; i < ___; ___ )
    sum = _______;
  }

  return _______;
}
```

Why `esi` and `eax` instead of `rsi` and `rax`?
Why `rdi`?

Practice Arrays and Loops

Variable	Register
z	edi
sum	eax
i	esi

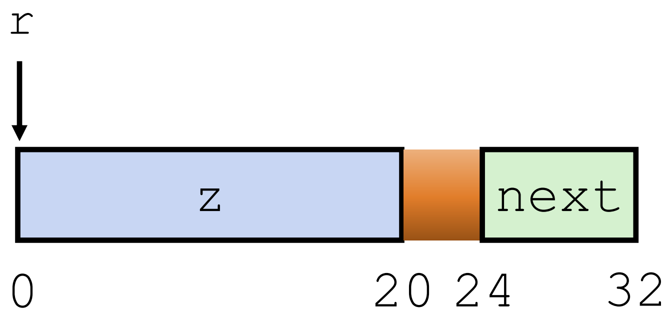
```
array_loop:
  mov   esi, 0
  xor   eax, eax
  jmp   L2

L1:
  add   eax, [edi + esi*4]
  inc   esi

L2:
  cmp   esi, 5
  jl    L1
  ret
```

```
int array_loop(int z[5]) {
  int sum = ___0___;

  int i;

  for(i = __0__; i < __5__; __i++__ )
    sum = __sum+z[i]__;
  }            sum + (*(z+i))

  return ___sum___;
}
```

Why esi and eax instead of rsi and rax?
Why rdi?

Structure Representation

```
struct rec {
  int z[5];
  struct rec *next;
};
```



Structure represented as block of memory

- Big enough to hold all the fields

Fields ordered according to declaration

- Even if another ordering could yield a more compact representation

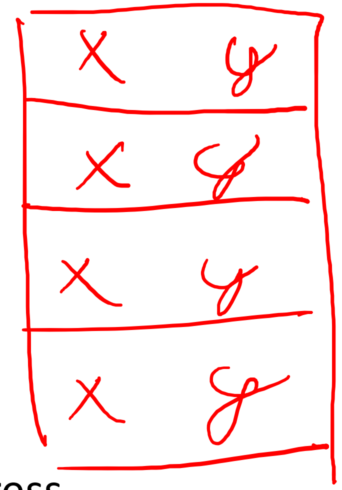
Compiler determines overall size + positions of fields

- Machine-level program has no understanding of structures and high-level types

```
struct Point {
        int xcoord;
        int ycoord;
};

struct Point points[10];
```

Variable	Register
points	ebx
i	eax
y	edx
p	esi

Memory Diagram



What is the corresponding assembly?

```
int y = points[i].ycoord;
```

Store the value found at the computed address.

```
mov edx, [ebx + 8*eax + 4]
```

What is the corresponding assembly?

```
int *p = &points[i].ycoord;
```

Store the computed address.

```
lea esi, [ebx + 8*eax + 4]
```

Unlike `add`, `lea` does not change flags.
Unlike `add`, `lea` effectively takes two or three operands.
Unlike `add`, `lea` can store its result in any register.

https://stackoverflow.com/questions/1658294/whats-the-purpose-of-the-lea-instruction
Thanks to I. J. Kennedy for the example.

Subroutines

Procedures

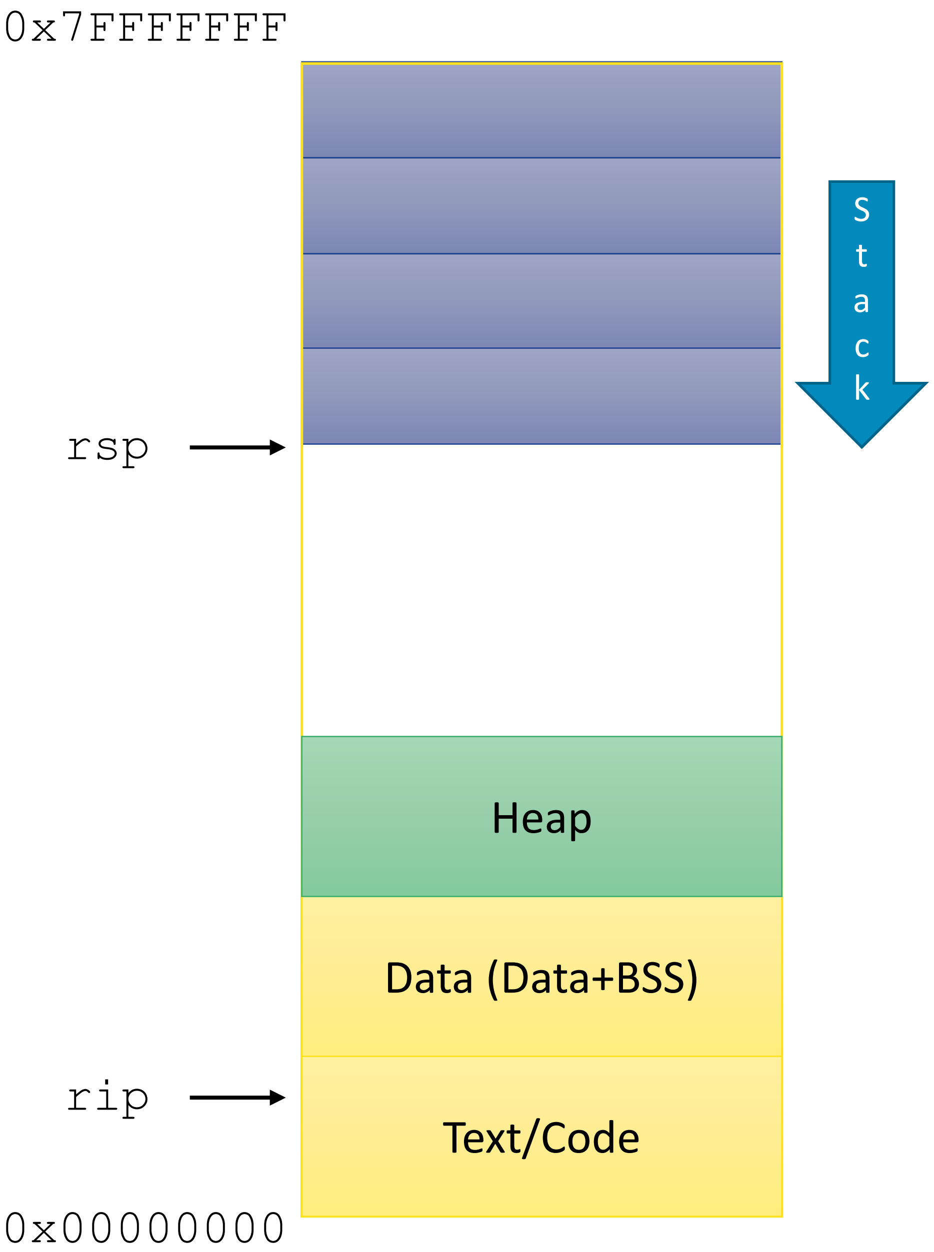
```
void P(int arg) {
    // …
    Q(x);
    // …
    return val;
}
```

Procedures, functions, methods, subroutines, handlers, etc.

- We need mechanisms for:
    - Passing Control: When procedure P calls procedure Q, program counter must be set to address of Q, when Q returns, program counter must be reset to instruction in P following procedure call
    - Passing Data: Must handle parameters and return values
    - Local memory: Q must be able to allocate (and deallocate) space for local variables

The Stack

- Traditionally the "top" of memory
- Grows "down"
- Provides storage for local variables
- `rsp` holds address of top element of stack



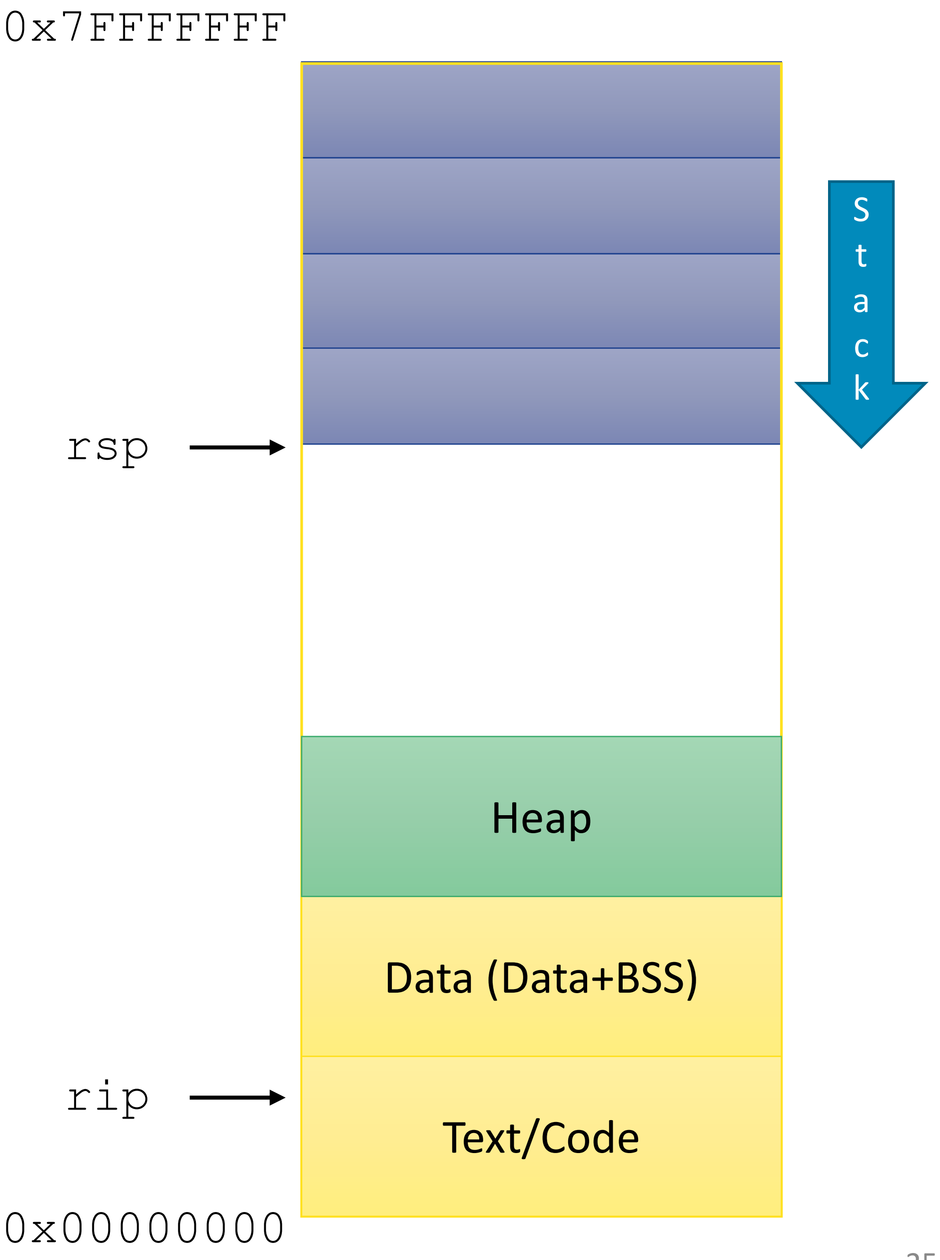
Modifying the Stack

push OP
- `sub rsp, 8` ; Depends on size
- `mov [rsp], OP`

pop DEST ; DEST can be M or R
- `mov DEST, [rsp]`
- `add rsp, 8` ; Depends on size

explicitly modify `rsp`
- `sub rsp, 4`
- `add rsp, 4`

modify memory values above `rsp`
- `mov [rsp + 4], 47`



Modifying the Stack

call ADDRESS

- sub rsp, 8
- mov [rsp], rip
- jmp ADDRESS

push rip

push OP

```
sub rsp, 8
mov [rsp], OP
```

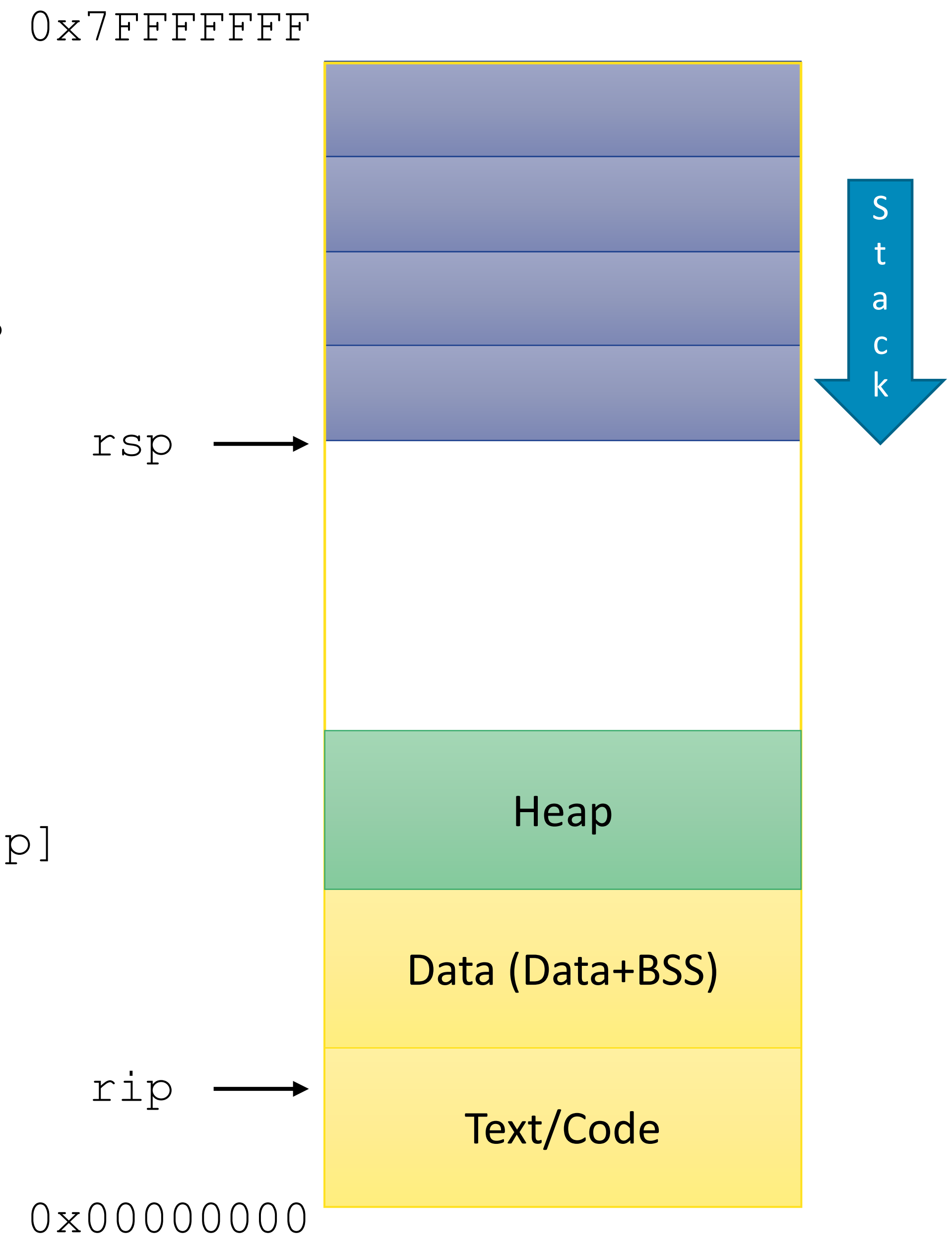
ret

- mov rip, [rsp]
- add rsp, 8

pop rip

pop DEST

```
mov DEST, [rsp]
add rsp, 8
```



Procedure Calls, Division of Labor

Caller

Before

- Save registers, if necessary
- Put arguments in place
- Make call

After

- Restore registers, if necessary
- Use result

Callee (Called procedure)

Preamble

- Save registers, if necessary
- Allocate space on stack

Exit code

- Put return value in place
- Restore registers, if necessary
- Deallocate space on stack
- Return

Stack Frames

```
int proc(int *p);

int example1(int x) {
   int a[4];
   a[3] = 10;
   return proc(a);
}
```

```
example1:
   sub    rsp, 16
   mov    [rsp + 12], 10
   mov    rdi, rsp
   call   0x400546 <proc>
   add    rsp, 16
   ret
```

16?
Return value?

Caller

Before

- Save registers, if necessary
- Put arguments in place
- Make call

After

- Restore registers, if necessary
- Use result

Callee

Preamble

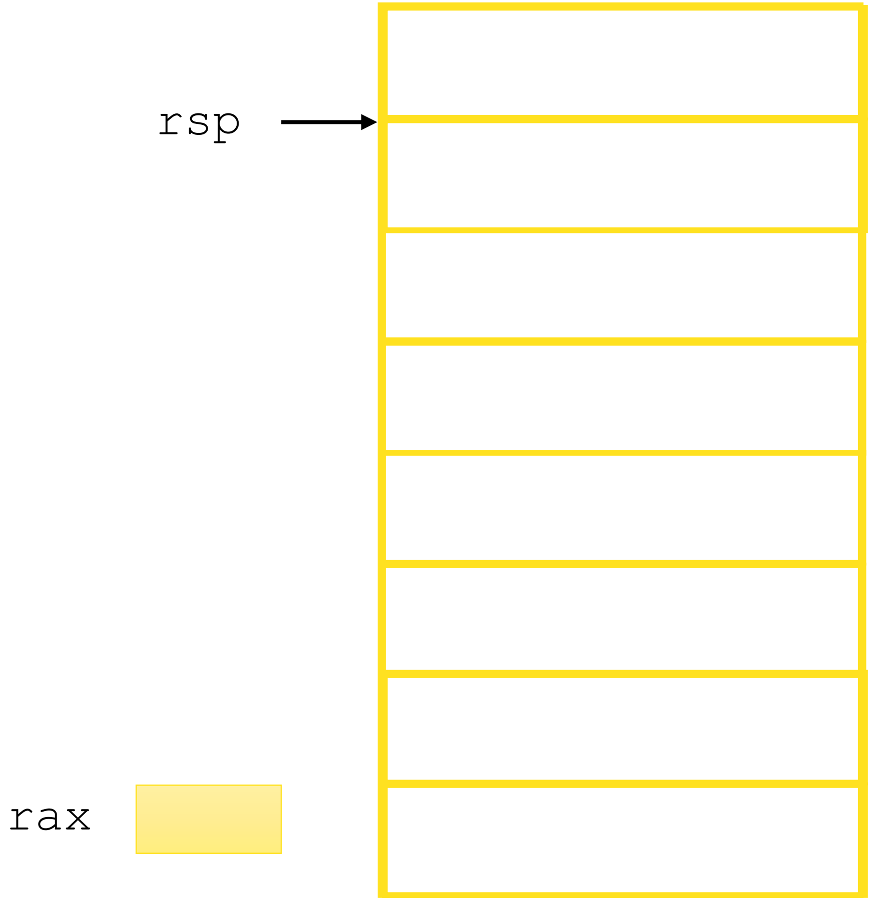
- Save registers, if necessary
- Allocate space on stack

Exit code

- Put return value in place
- Restore registers, if necessary
- Deallocate space on stack

Practice Modifying the Stack

```
0x400557 <fun>:
   400557: mov   [rsp + 16], 13
   40055a: ret

0x40055b <main>:
rip → 40055b: sub   rsp, 8
   40055f: push 47
   400560: call 0x400557 <fun>
   400565: pop   rax
   400566: add   rax, [rsp]
   40056a: add   rsp, 8
   40056e: ret
```

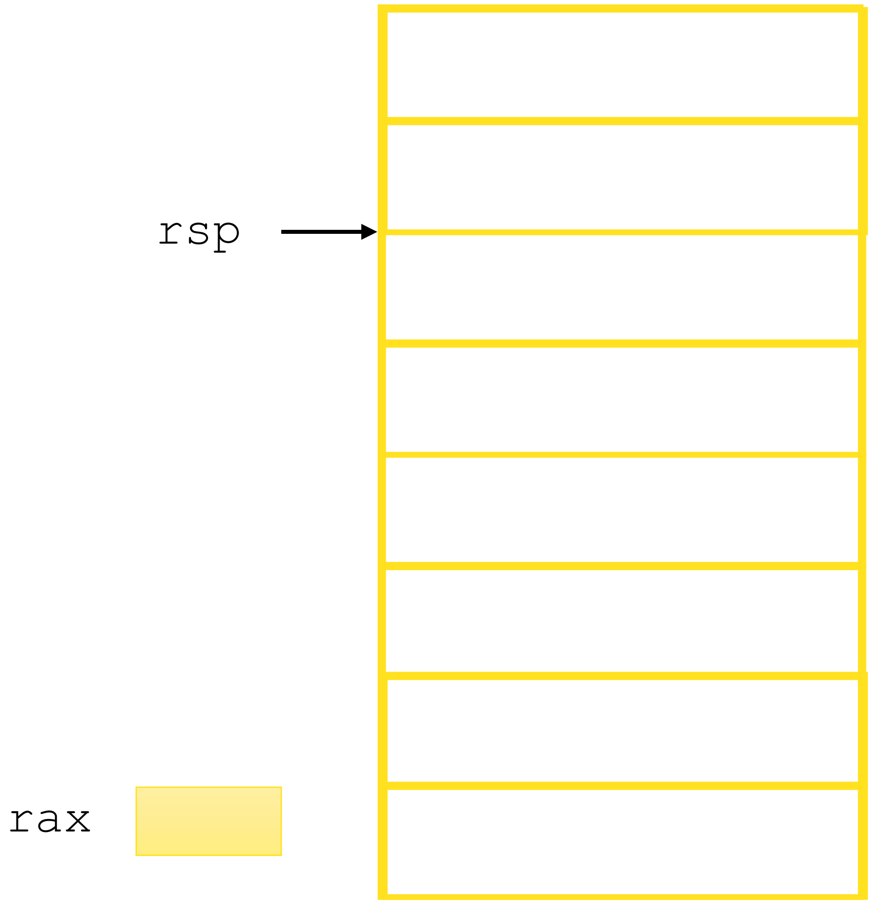


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

push OP
1. sub rsp, 8
2. mov [rsp], OP

```
0x400557 <fun>:
     400557: mov   [rsp + 16], 13
     40055a: ret

0x40055b <main>:
     40055b: sub   rsp, 8
rip → 40055f: push 47
     400560: call  0x400557 <fun>
     400565: pop   rax
     400566: add   rax, [rsp]
     40056a: add   rsp, 8
     40056e: ret
```

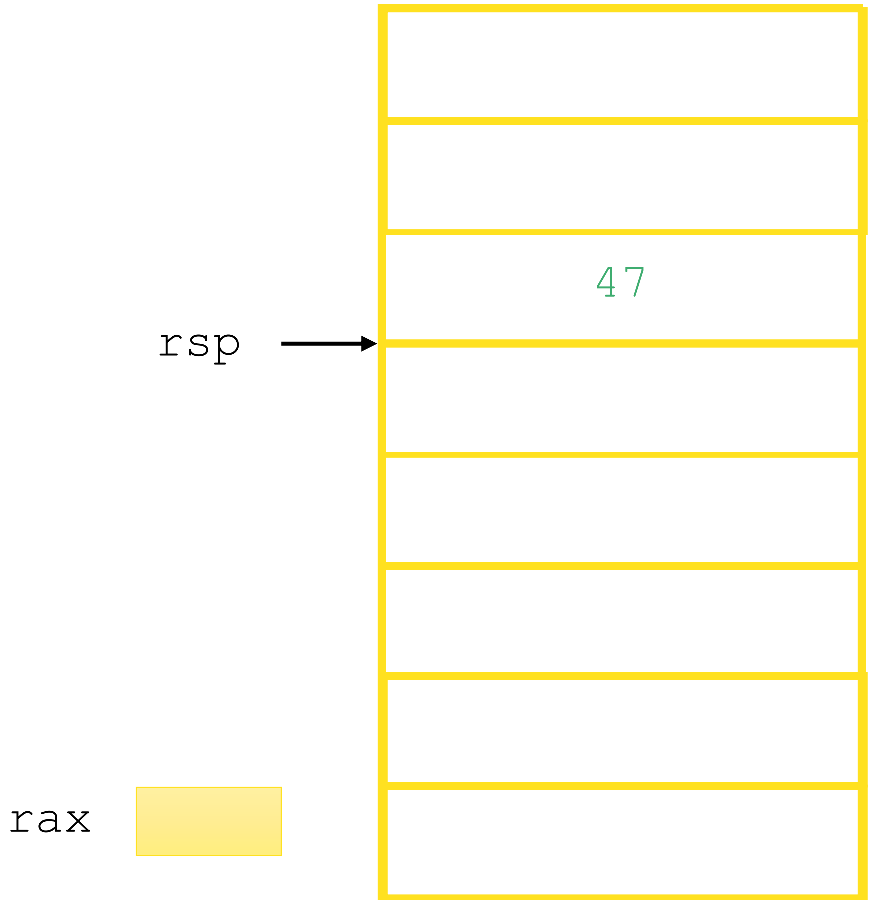


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

```
call ADDRESS
1. sub rsp, 8
2. mov [rsp], rip
3. jmp ADDRESS
```

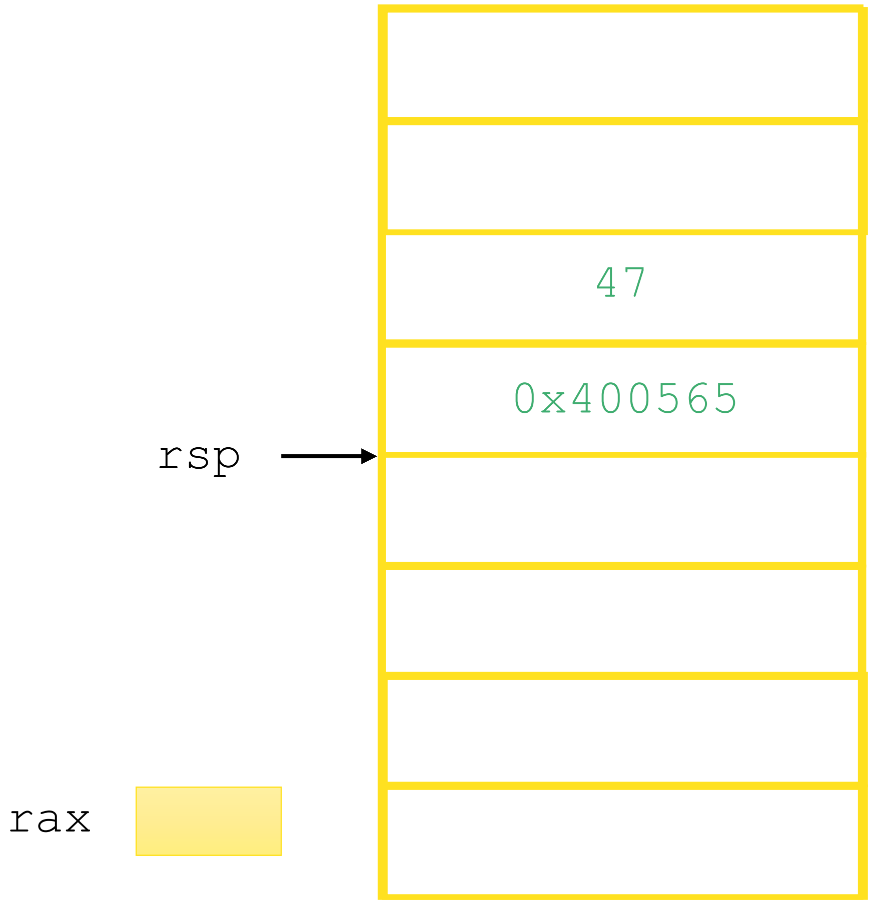
```
0x400557 <fun>:
   400557: mov   [rsp + 16], 13
   40055a: ret

0x40055b <main>:
   40055b: sub   rsp, 8
   40055f: push  47
rip → 400560: call  0x400557 <fun>
   400565: pop   rax
   400566: add   rax, [rsp]
   40056a: add   rsp, 8
   40056e: ret
```



What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

```
       0x400557 <fun>:
rip →    400557: mov  [rsp + 16], 13
         40055a: ret

       0x40055b <main>:
         40055b: sub  rsp, 8
         40055f: push 47
         400560: call 0x400557 <fun>
         400565: pop  rax
         400566: add  rax, [rsp]
         40056a: add  rsp, 8
         40056e: ret
```

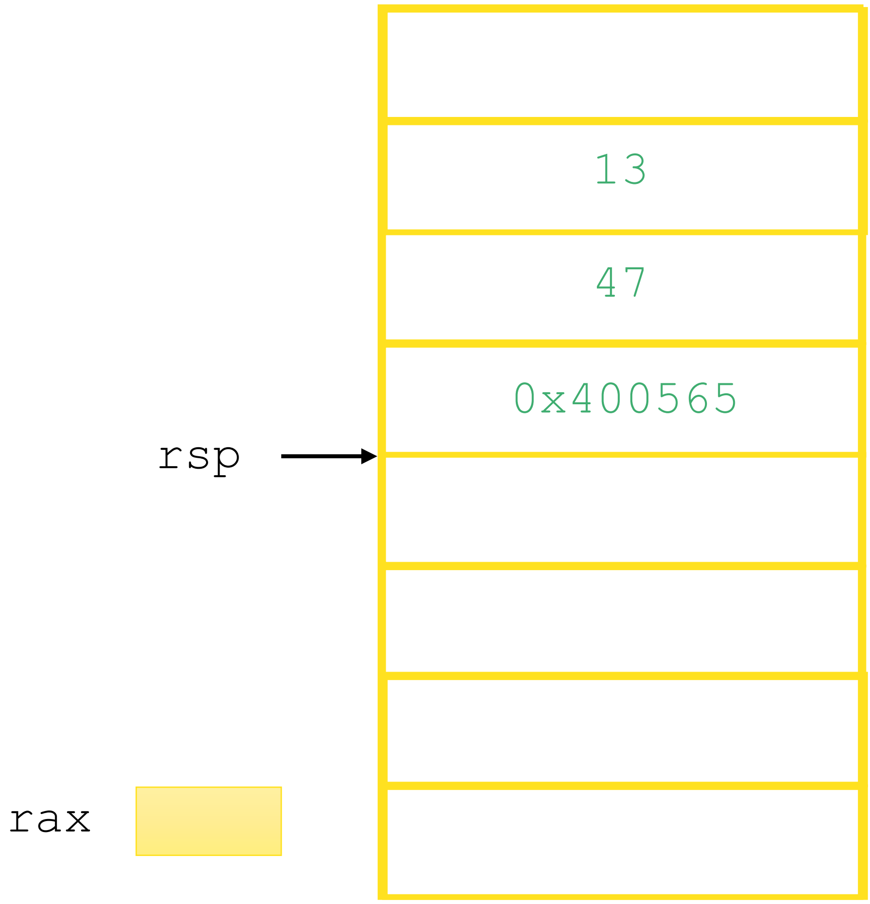


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

```
ret
1. mov rip, [rsp]
2. add rsp, 8
```

```
0x400557 <fun>:
         400557: mov  [rsp + 16], 13
rip →    40055a: ret

0x40055b <main>:
         40055b: sub  rsp, 8
         40055f: push 47
         400560: call 0x400557 <fun>
         400565: pop  rax
         400566: add  rax, [rsp]
         40056a: add  rsp, 8
         40056e: ret
```

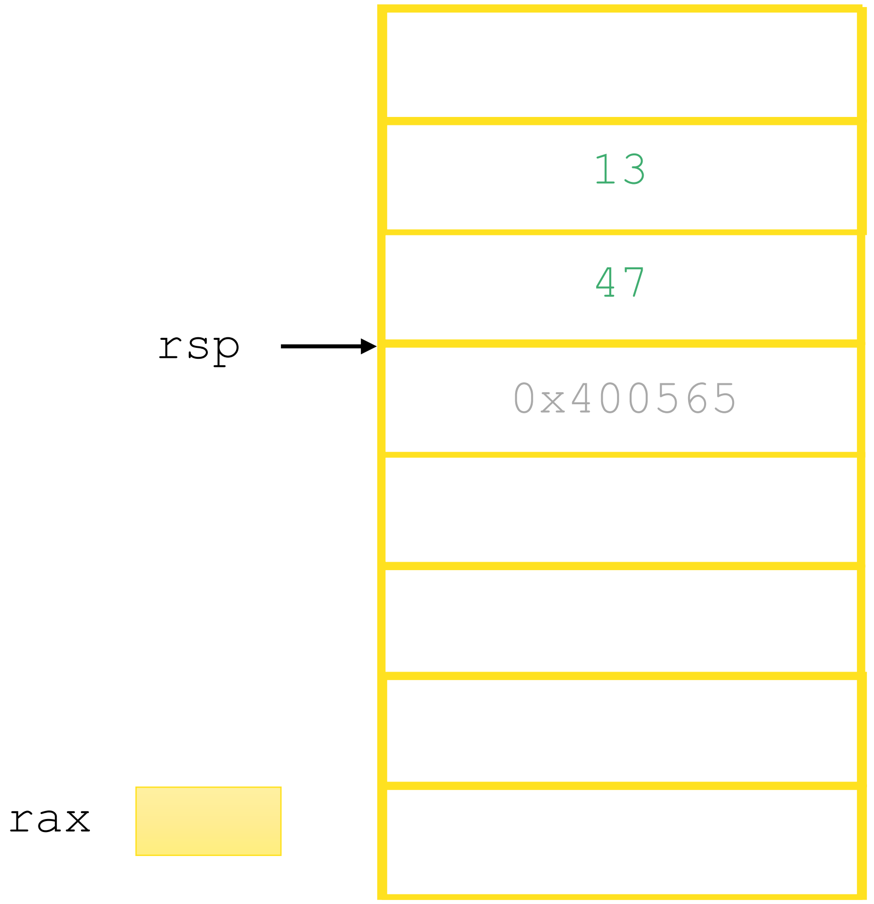


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

pop DEST
1. mov DEST, [rsp]
2. add rsp, 8

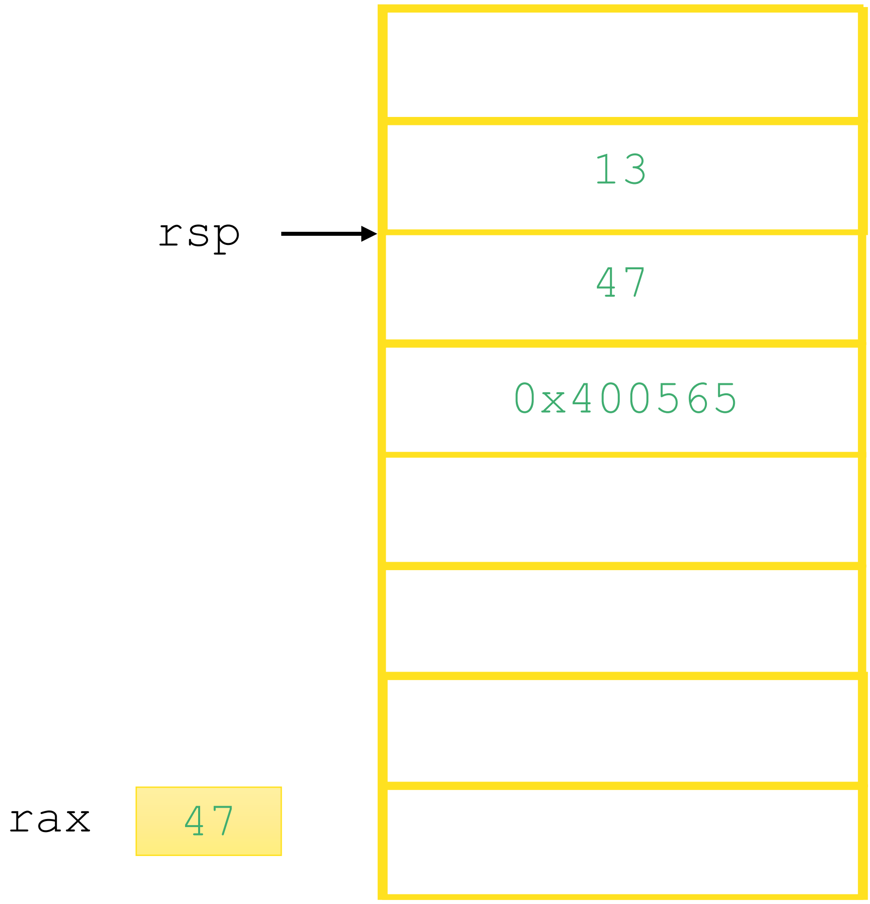
```
0x400557 <fun>:
   400557: mov   [rsp + 16], 13
   40055a: ret

0x40055b <main>:
   40055b: sub   rsp, 8
   40055f: push  47
   400560: call  0x400557 <fun>
rip →  400565: pop   rax
   400566: add   rax, [rsp]
   40056a: add   rsp, 8
   40056e: ret
```



What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

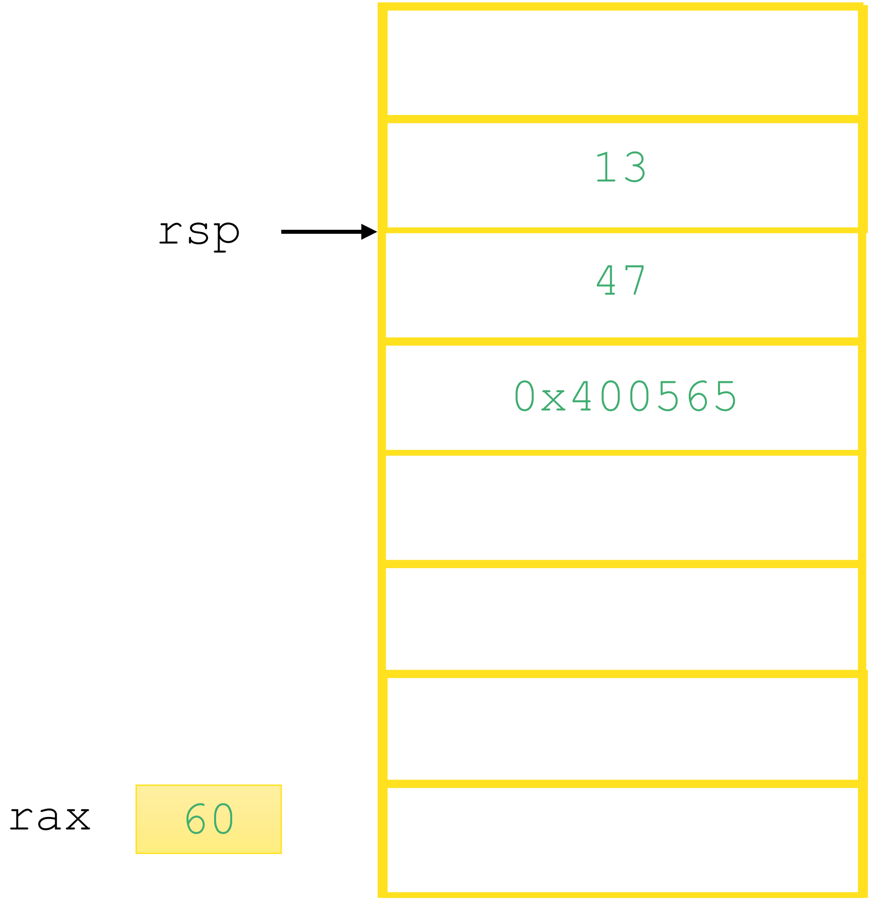
```
0x400557 <fun>:
   400557: mov   [rsp + 16], 13
   40055a: ret

0x40055b <main>:
   40055b: sub   rsp, 8
   40055f: push 47
   400560: call 0x400557 <fun>
   400565: pop   rax
rip → 400566: add   rax, [rsp]
   40056a: add   rsp, 8
   40056e: ret
```



What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

```
0x400557 <fun>:
   400557: mov  [rsp + 16], 13
   40055a: ret

0x40055b <main>:
   40055b: sub  rsp, 8
   40055f: push 47
   400560: call 0x400557 <fun>
   400565: pop  rax
   400566: add  rax, [rsp]
rip → 40056a: add  rsp, 8
   40056e: ret
```

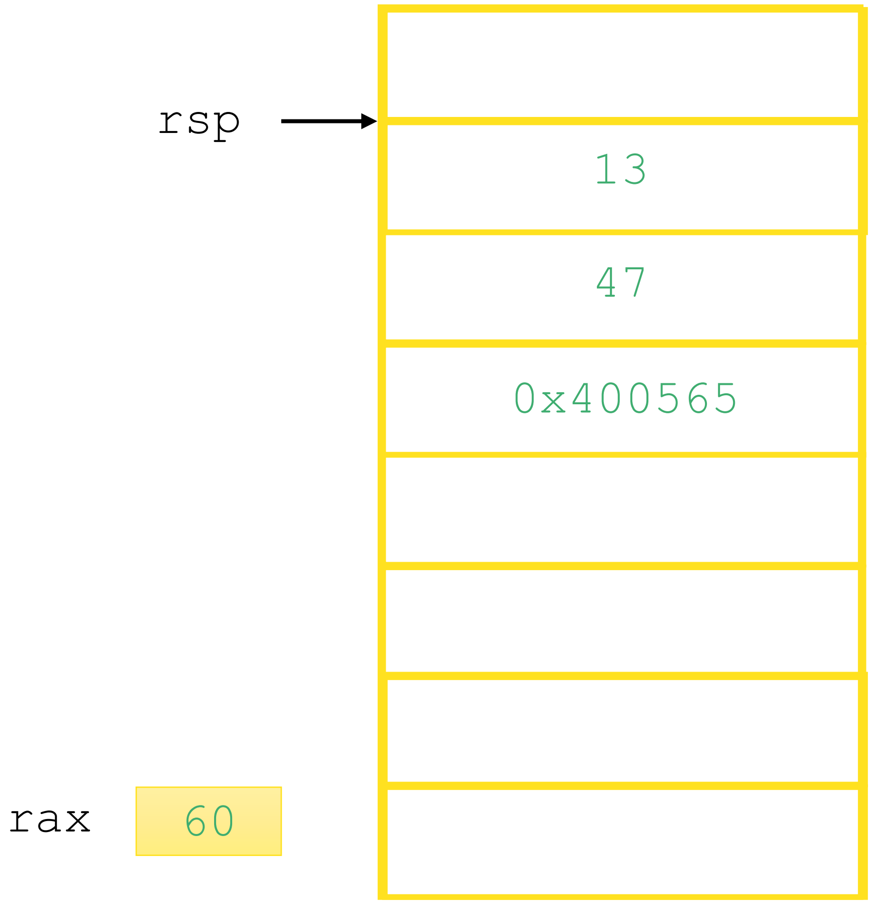


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

Practice Modifying the Stack

```
ret
1. mov rip, [rsp]
2. add rsp, 8
```

```
0x400557 <fun>:
   400557: mov  [rsp + 16], 13
   40055a: ret

0x40055b <main>:
   40055b: sub  rsp, 8
   40055f: push 47
   400560: call 0x400557 <fun>
   400565: pop  rax
   400566: add  rax, [rsp]
   40056a: add  rsp, 8
rip → 40056e: ret
```

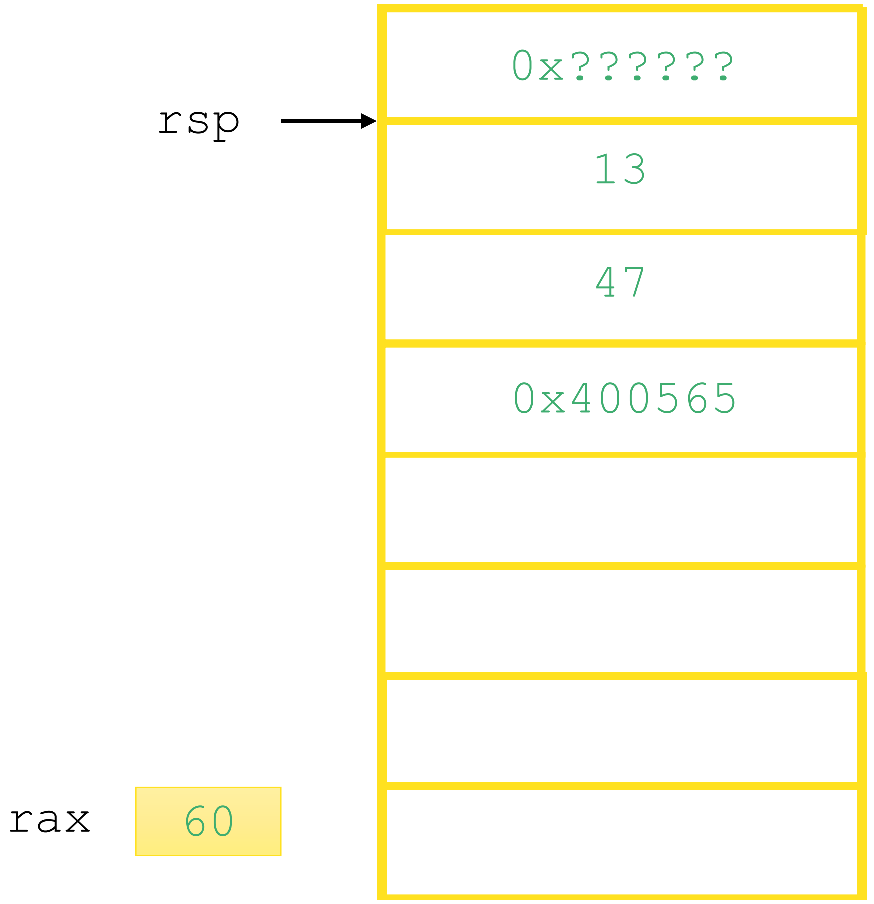


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

ret
1. mov rip, [rsp]
2. add rsp, 8

Practice Modifying the Stack

```
0x400557 <fun>:
   400557: mov  [rsp + 16], 13
   40055a: ret

0x40055b <main>:
   40055b: sub  rsp, 8
   40055f: push 47
   400560: call 0x400557 <fun>
   400565: pop  rax
   400566: add  rax, [rsp]
   40056a: add  rsp, 8
rip → 40056e: ret
```

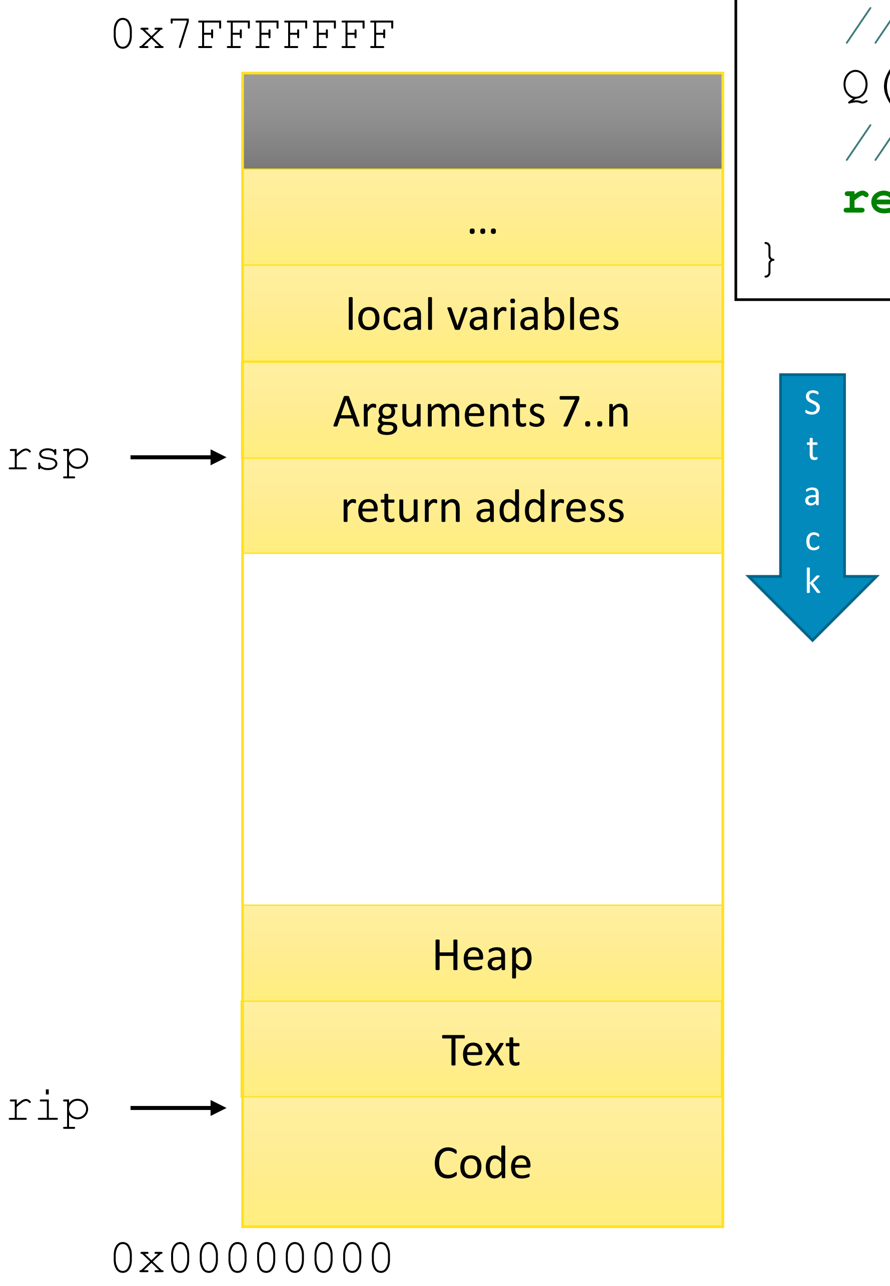


What is the value in `rax` immediately before main returns?
What is the value in `rsp` immediately before main returns?

- Every procedure call has its own stack frame

- Procedure `P` (caller) uses registers and stack space to provide arguments to `Q` (callee)

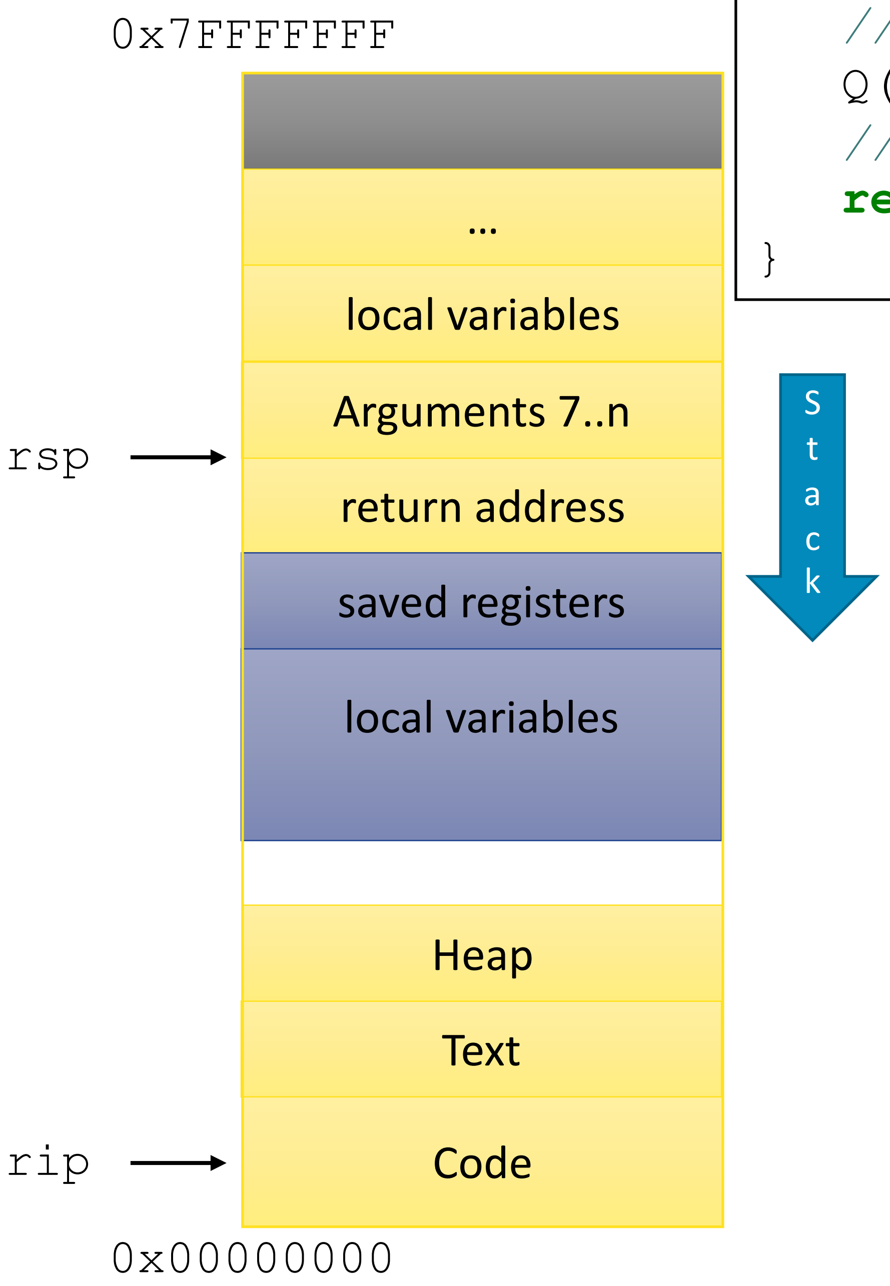
- `call Q`
  - Pushes return address (current `rip`) onto stack
  - Sets `rip` to first instruction of `Q`



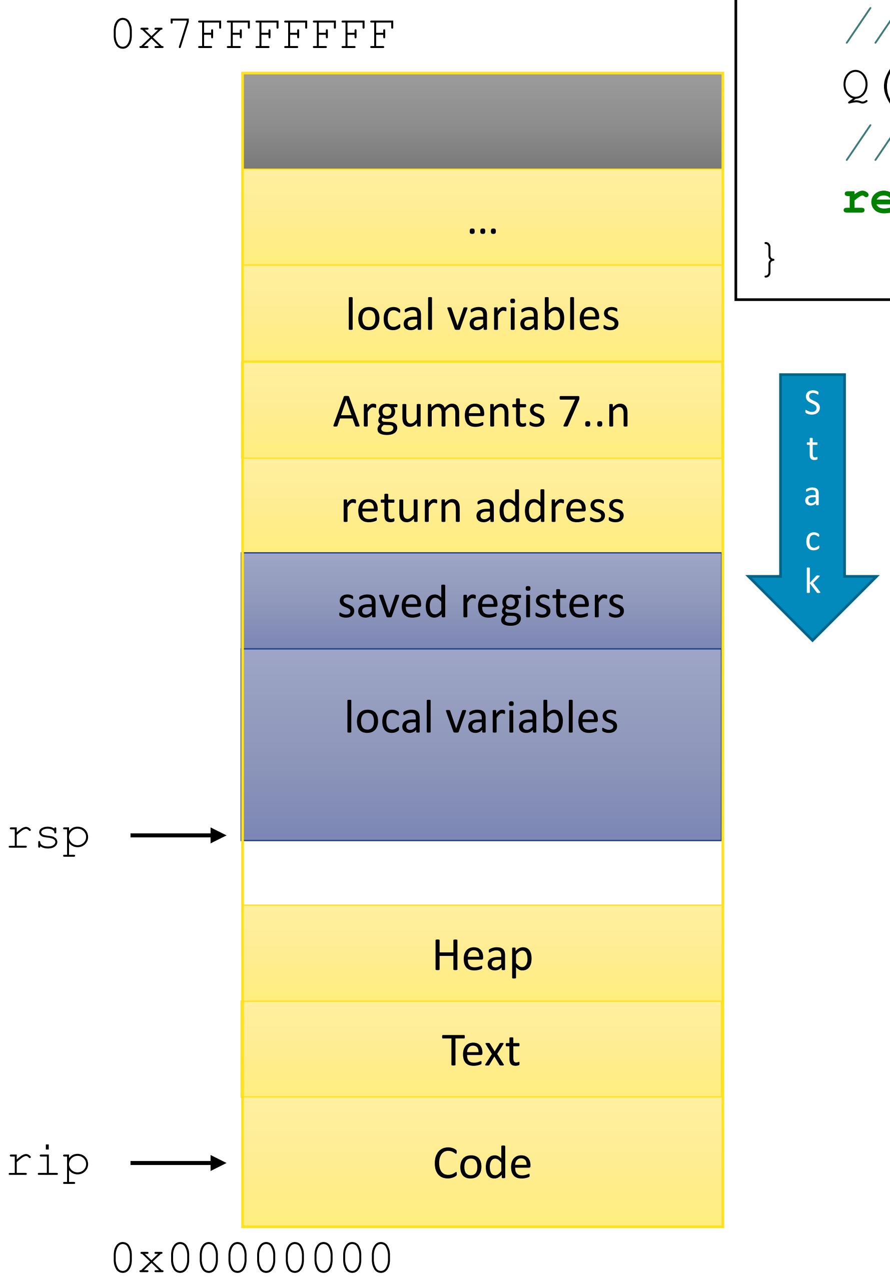

```
void P(int arg) {
    // ...
    Q(x);
    // ...
    return val;
}
```

- Every procedure call has its own stack frame
- Procedure `P` (caller) uses registers and stack space to provide arguments to `Q` (callee)
- `call Q`
  - Pushes return address (current `rip`) onto stack
  - Sets `rip` to first instruction of `Q`
- `Q`'s new stack frame is reserved by subtracting from `rsp`

```
void P(int arg) {
    // …
    Q(x);
    // …
    return val;
}
```

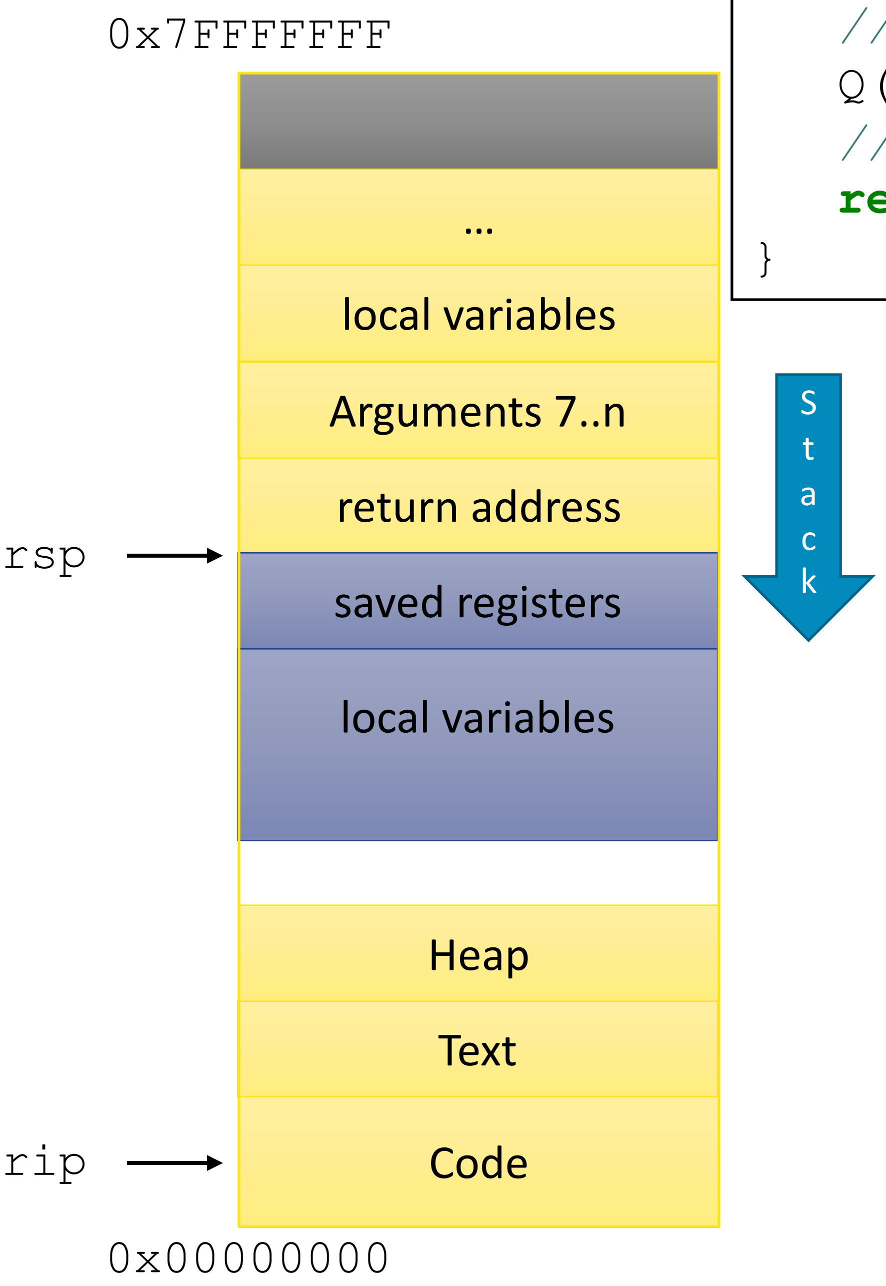


- Every procedure call has its own stack frame
- Procedure `P` (caller) uses registers and stack space to provide arguments to `Q` (callee)
- `call Q`
  - Pushes return address (current `rip`) onto stack
  - Sets `rip` to first instruction of `Q`
- `Q`'s new stack frame is reserved by subtracting from `rsp`
- `Q` uses `rax` register to store the return value
- `Q` releases its stack frame by adding to `rsp`



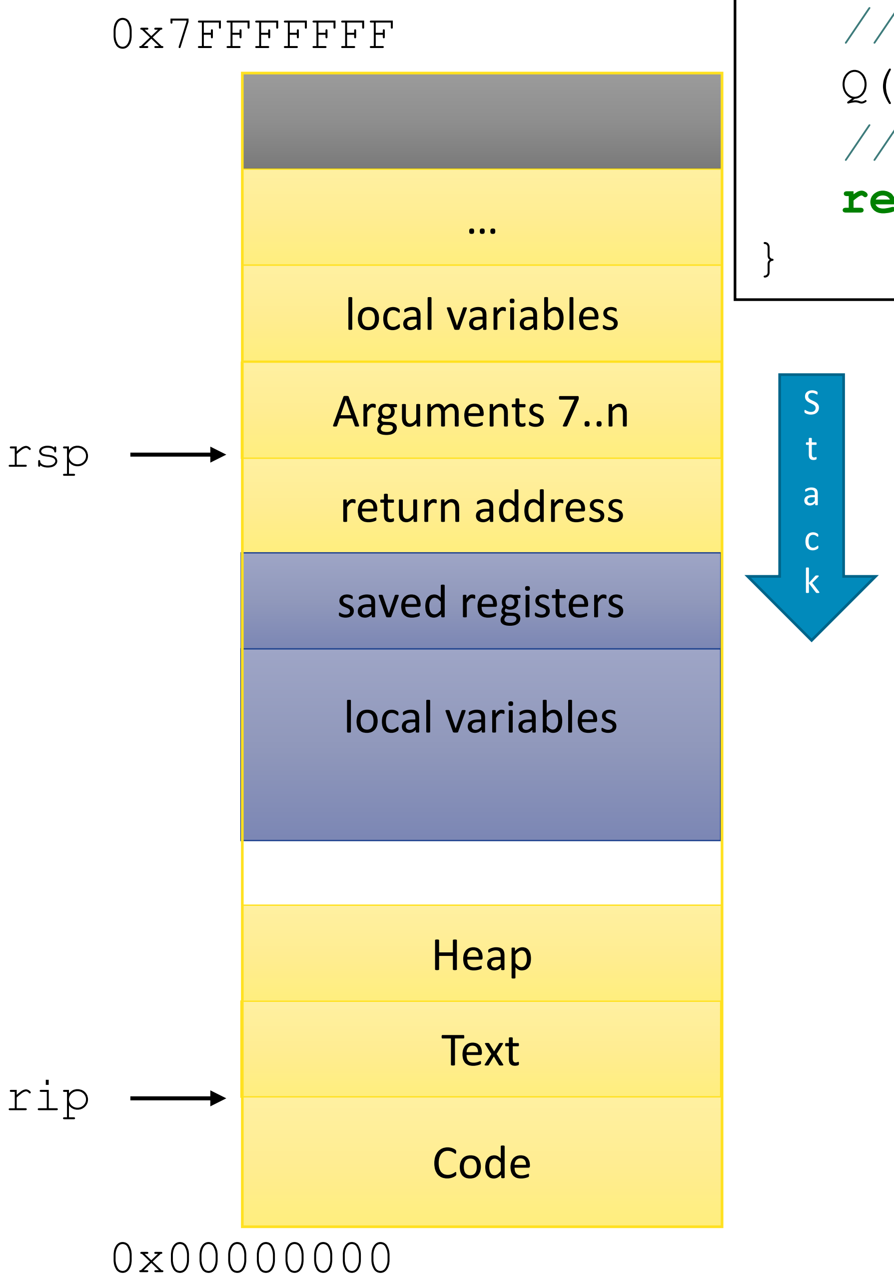

```
void P(int arg) {
    // …
    Q(x);
    // …
    return val;
}
```

- Every procedure call has its own stack frame
- Procedure `P` (caller) uses registers and stack space to provide arguments to `Q` (callee)
- `call Q`
  - Pushes return address (current `rip`) onto stack
  - Sets `rip` to first instruction of `Q`
- `Q`'s new stack frame is reserved by subtracting from `rsp`
- `Q` uses `rax` register to store the return value
- `Q` releases its stack frame by adding to `rsp`
- `ret`
  - Pops return address from stack into `rip`




```
void P(int arg) {
    // …
    Q(x);
    // …
    return val;
}
```

- Every procedure call has its own stack frame
- Procedure `P` (caller) uses registers and stack space to provide arguments to `Q` (callee)
- `call Q`
  - Pushes return address (current `rip`) onto stack
  - Sets `rip` to first instruction of `Q`
- `Q`'s new stack frame is reserved by subtracting from `rsp`
- `Q` uses `rax` register to store the return value
- `Q` releases its stack frame by adding to `rsp`
- `ret`
  - Pops return address from stack into `rip`




```
void P(int arg) {
    // ...
    Q(x);
    // ...
    return val;
}
```

Procedure call with many arguments

```c
int func1(int x1, int x2, int x3,
          int x4, int x5, int x6,
          int x7, int x8){
  int l1 = x1+x2;
  int l2 = x3+x4;
  int l3 = x5+x6;
  int l4 = x7+x8;
  int l5 = 4;
  int l6 = 13;
  int l7 = 47;
  int l8 = l1 + l2 + l3 + l4 + l5
              + l6 + l7;
  return l8;
}

int main(int argc, char *argv[]){
  int x = func1(1,2,3,4,5,6,7,8);
  return x;
}
```

```asm
main:
  mov   edi, 1
  mov   esi, 2
  mov   edx, 3
  mov   ecx, 4
  mov   r8d, 5
  mov   r9d, 6
  push  8
  push  7
  call  func1
  add   rsp, 16
  ret
```

```asm
func1:
  add   edi, esi
  add   edx, ecx
  add   edi, edx
  add   r8d, r9d
  add   edi, r8d
  add   edi, DWORD PTR [rsp+8]
  add   edi, DWORD PTR [rsp+16]
  lea   eax, [rdi+64]
  ret
```

Size Directives

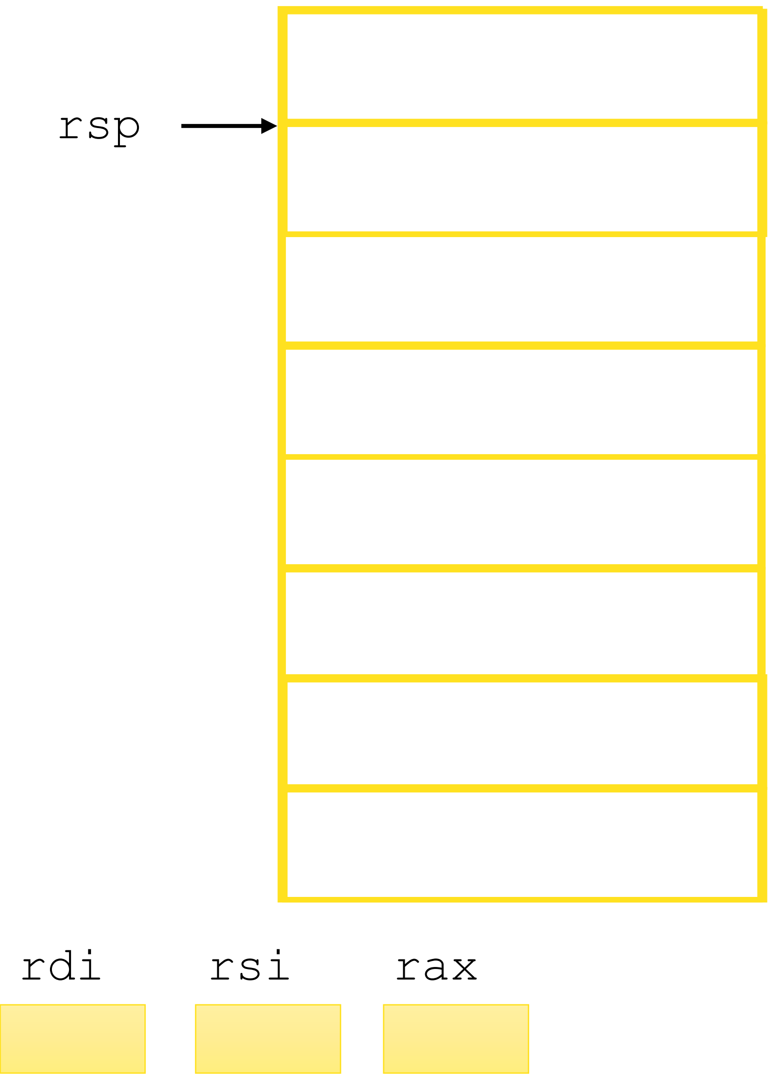
- The assembler can mostly infer the size of a piece of data
- When it can't, you need to specifically tell it

```
; Move 2 into the single byte at the address stored in rbx.
mov BYTE PTR [rbx], 2

; Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in rbx.
mov WORD PTR [rbx], 2

; Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in rbx.
mov DWORD PTR [rbx], 2

; Move the 64-bit integer representation of 2 into the 8 bytes starting at the address in rbx.
mov QWORD PTR [rbx], 2
```

What value gets returned by main?

Practice with Value Passing

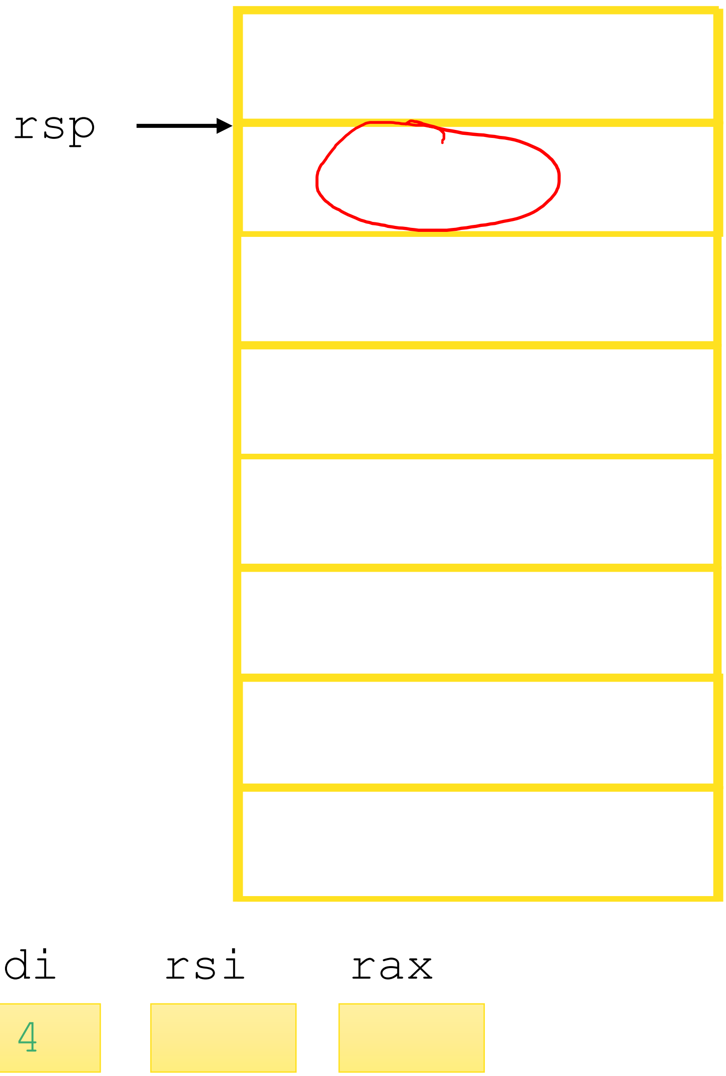
```
0x400540 <last>:
  400540: mov  rax, rdi
  400543: imul rax, rsi
  400547: ret

0x400548 <first>:
  400548: lea  rsi, [rdi + 0x1]
  40054c: sub  rdi, 0x1
  400550: call 0x400540 <last>
  400555: ret

0x400556 <main>:
rsp → 400560: mov  rdi, 4
  400563: call 0x400548 <first>
  400568: add  rax, 0x13
  40056c: ret
```



What value gets returned by main?

Practice with Value Passing

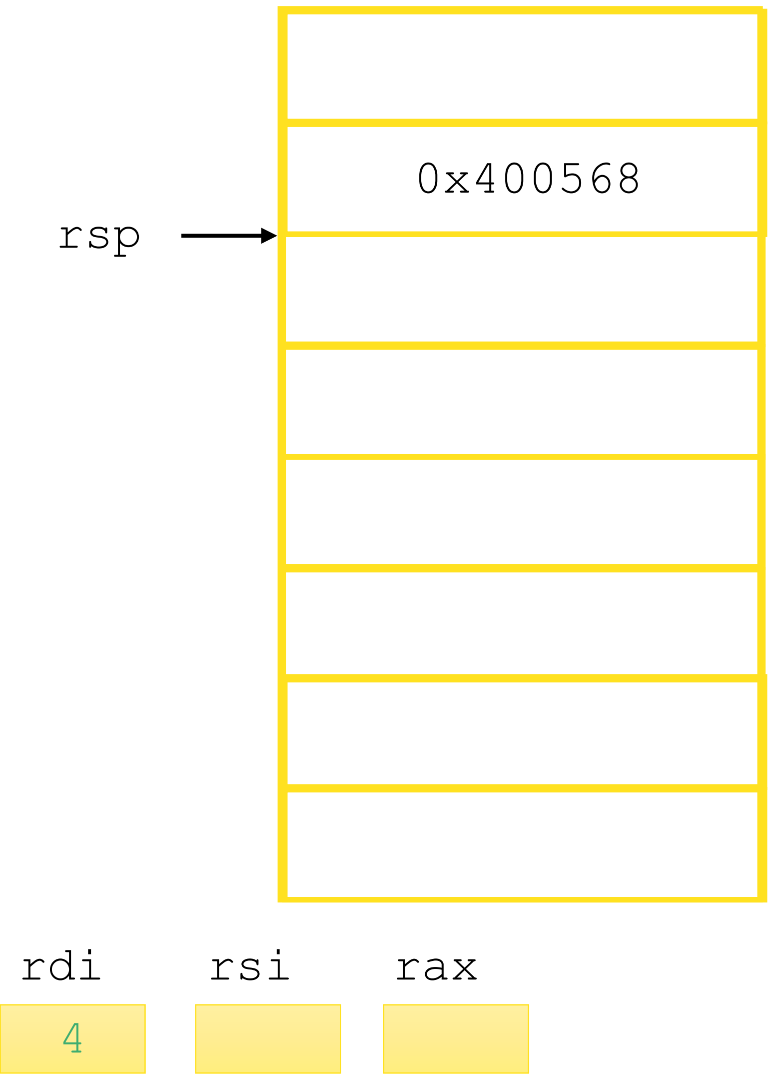
```
0x400540 <last>:
  400540: mov  rax, rdi
  400543: imul rax, rsi
  400547: ret

0x400548 <first>:
  400548: lea  rsi, [rdi + 0x1]
  40054c: sub  rdi, 0x1
  400550: call 0x400540 <last>
  400555: ret

0x400556 <main>:
  400560: mov  rdi, 4
rsp → 400563: call 0x400548 <first>
  400568: add  rax, 0x13
  40056c: ret
```



What value gets returned by main?

Practice with Value Passing

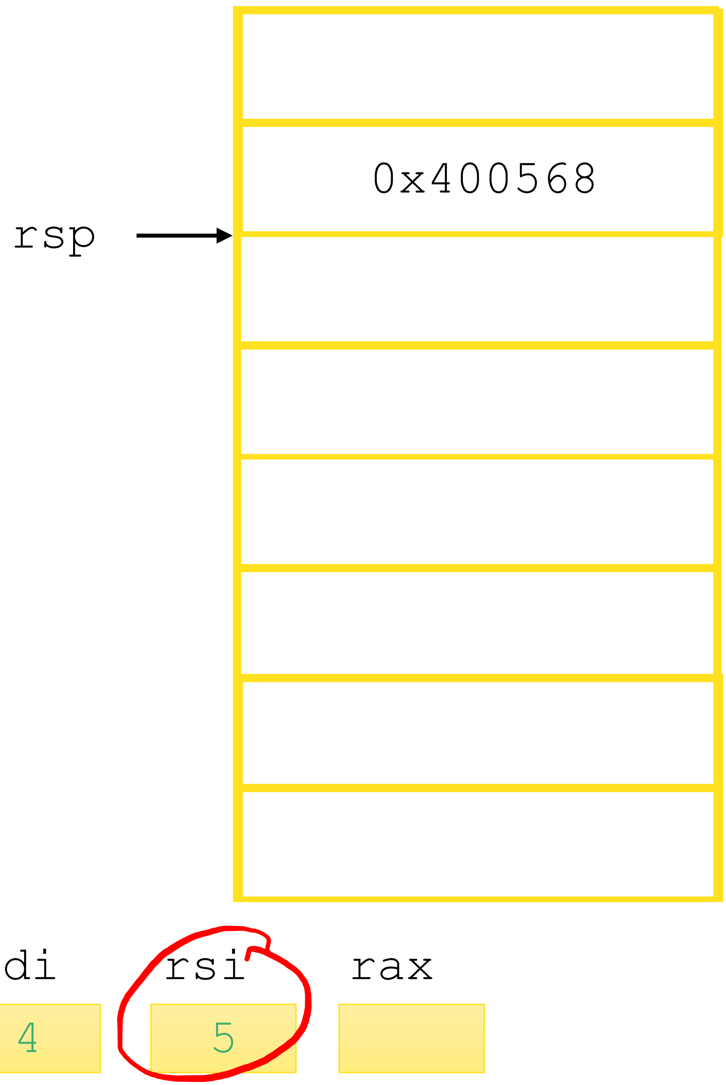
```
0x400540 <last>:
  400540: mov  rax, rdi
  400543: imul rax, rsi
  400547: ret

0x400548 <first>:
rsp → 400548: lea  rsi, [rdi + 0x1]
  40054c: sub  rdi, 0x1
  400550: call 0x400540 <last>
  400555: ret

0x400556 <main>:
  400560: mov  rdi, 4
  400563: call 0x400548 <first>
  400568: add  rax, 0x13
  40056c: ret
```



What value gets returned by main?

Practice with Value Passing

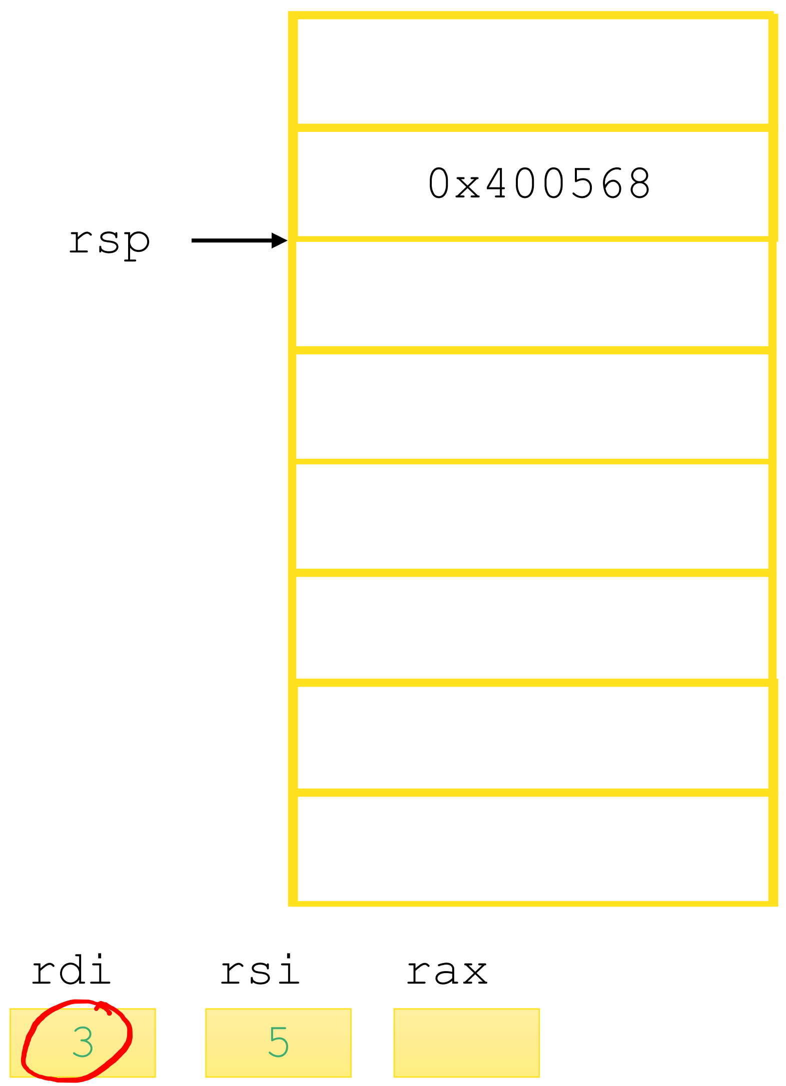
```
    0x400540 <last>:
      400540: mov   rax, rdi
      400543: imul  rax, rsi
      400547: ret

    0x400548 <first>:
      400548: lea   rsi, [rdi + 0x1]
rsp → 40054c: sub   rdi, 0x1
      400550: call  0x400540 <last>
      400555: ret

    0x400556 <main>:
      400560: mov   rdi, 4
      400563: call  0x400548 <first>
      400568: add   rax, 0x13
      40056c: ret
```



What value gets returned by main?

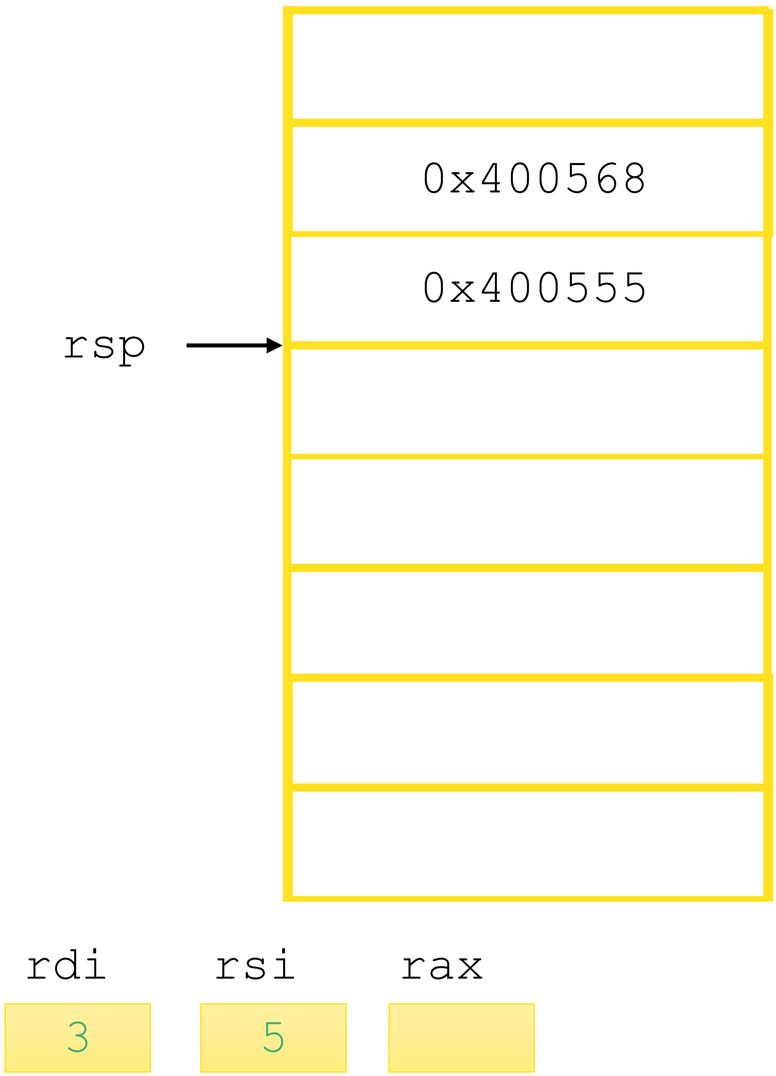
Practice with Value Passing

```
0x400540 <last>:
  400540: mov   rax, rdi
  400543: imul  rax, rsi
  400547: ret

0x400548 <first>:
  400548: lea   rsi, [rdi + 0x1]
  40054c: sub   rdi, 0x1
rsp → 400550: call  0x400540 <last>
  400555: ret

0x400556 <main>:
  400560: mov   rdi, 4
  400563: call  0x400548 <first>
  400568: add   rax, 0x13
  40056c: ret
```



Practice with Value Passing

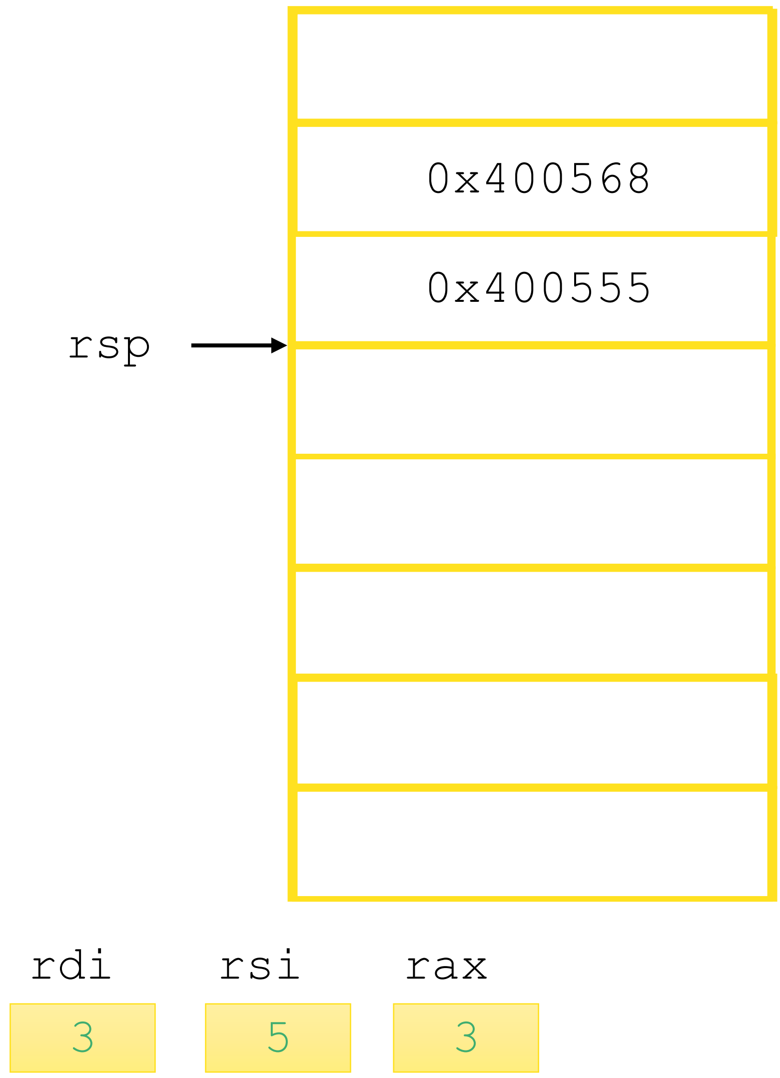
```
        0x400540 <last>:
rsp ──▶   400540: mov   rax, rdi
          400543: imul  rax, rsi
          400547: ret

        0x400548 <first>:
          400548: lea   rsi, [rdi + 0x1]
          40054c: sub   rdi, 0x1
          400550: call  0x400540 <last>
          400555: ret

        0x400556 <main>:
          400560: mov   rdi, 4
          400563: call  0x400548 <first>
          400568: add   rax, 0x13
          40056c: ret
```



What value gets returned by main?

Practice with Value Passing

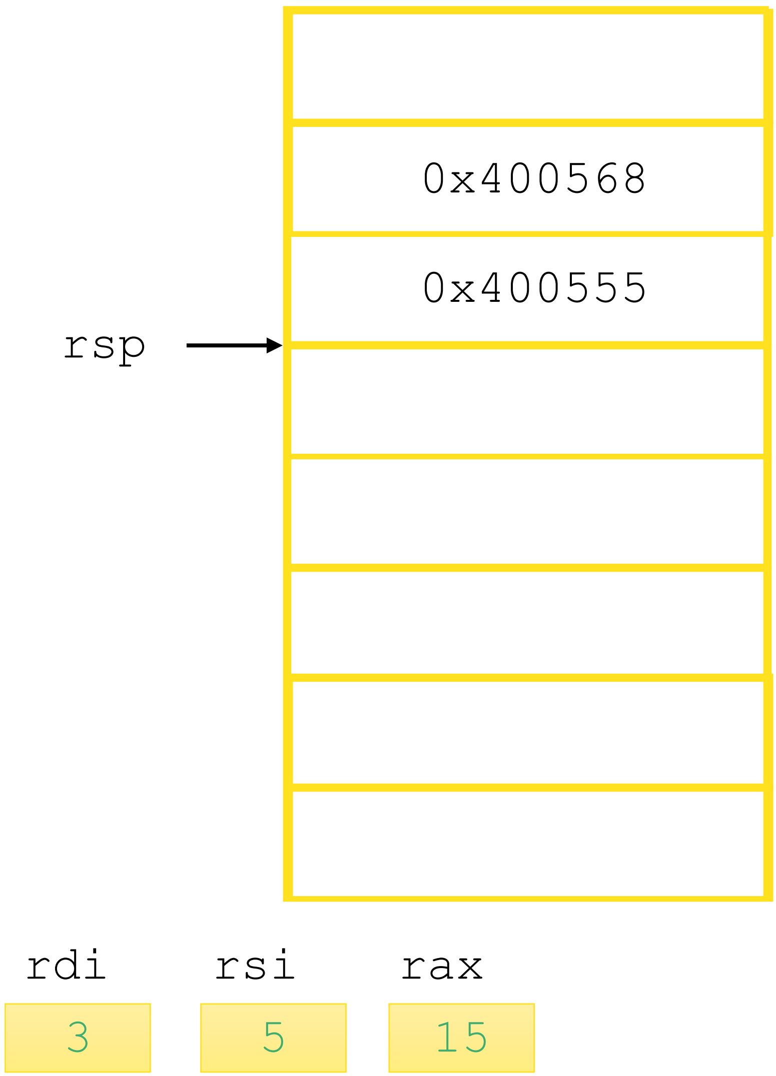
```
0x400540 <last>:
  400540: mov  rax, rdi
rsp → 400543: imul rax, rsi
  400547: ret

0x400548 <first>:
  400548: lea  rsi, [rdi + 0x1]
  40054c: sub  rdi, 0x1
  400550: call 0x400540 <last>
  400555: ret

0x400556 <main>:
  400560: mov  rdi, 4
  400563: call 0x400548 <first>
  400568: add  rax, 0x13
  40056c: ret
```



What value gets returned by main?

Practice with Value Passing

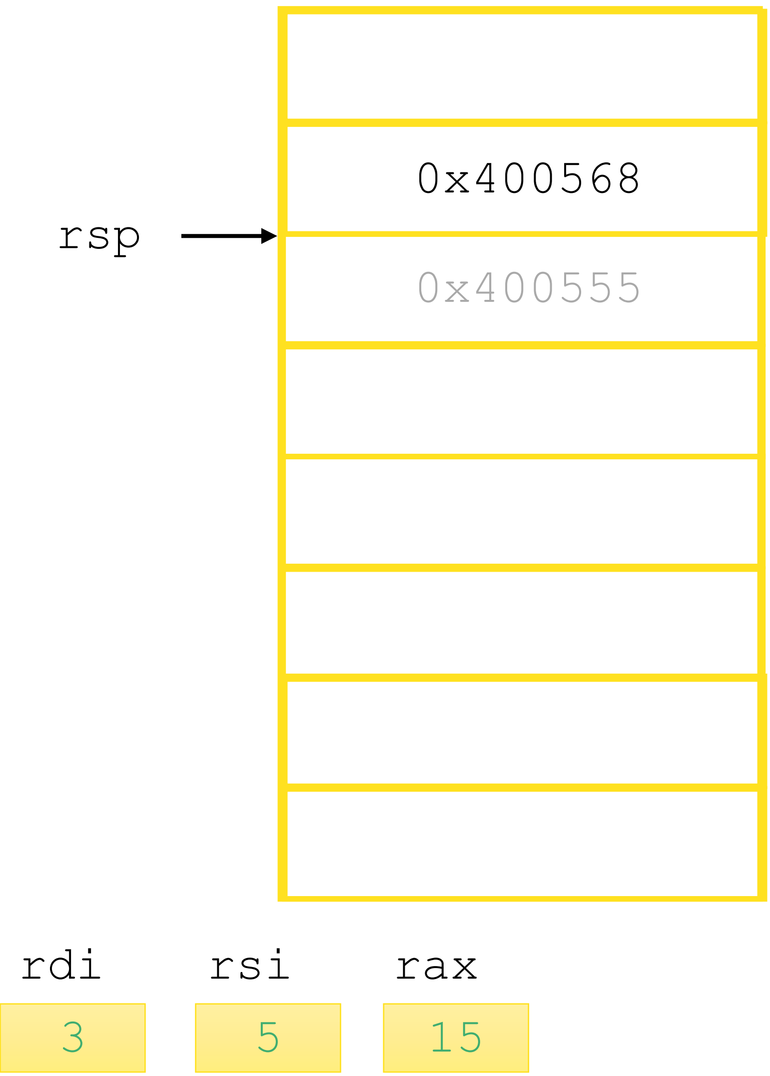
```
0x400540 <last>:
        400540: mov  rax, rdi
        400543: imul rax, rsi
rsp  →  400547: ret

0x400548 <first>:
        400548: lea  rsi, [rdi + 0x1]
        40054c: sub  rdi, 0x1
        400550: call 0x400540 <last>
        400555: ret

0x400556 <main>:
        400560: mov  rdi, 4
        400563: call 0x400548 <first>
        400568: add  rax, 0x13
        40056c: ret
```



What value gets returned by main?

Practice with Value Passing

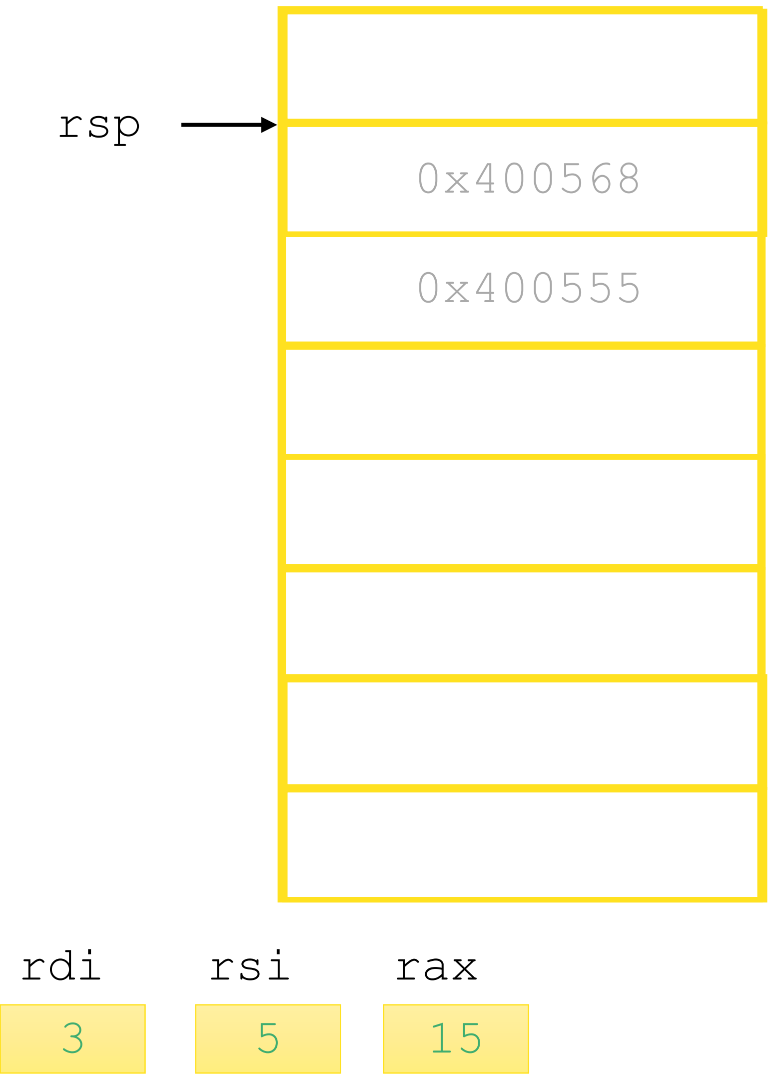
```
0x400540 <last>:
   400540: mov  rax, rdi
   400543: imul rax, rsi
   400547: ret

0x400548 <first>:
   400548: lea  rsi, [rdi + 0x1]
   40054c: sub  rdi, 0x1
   400550: call 0x400540 <last>
rsp → 400555: ret

0x400556 <main>:
   400560: mov  rdi, 4
   400563: call 0x400548 <first>
   400568: add  rax, 0x13
   40056c: ret
```



What value gets returned by main?

Practice with Value Passing

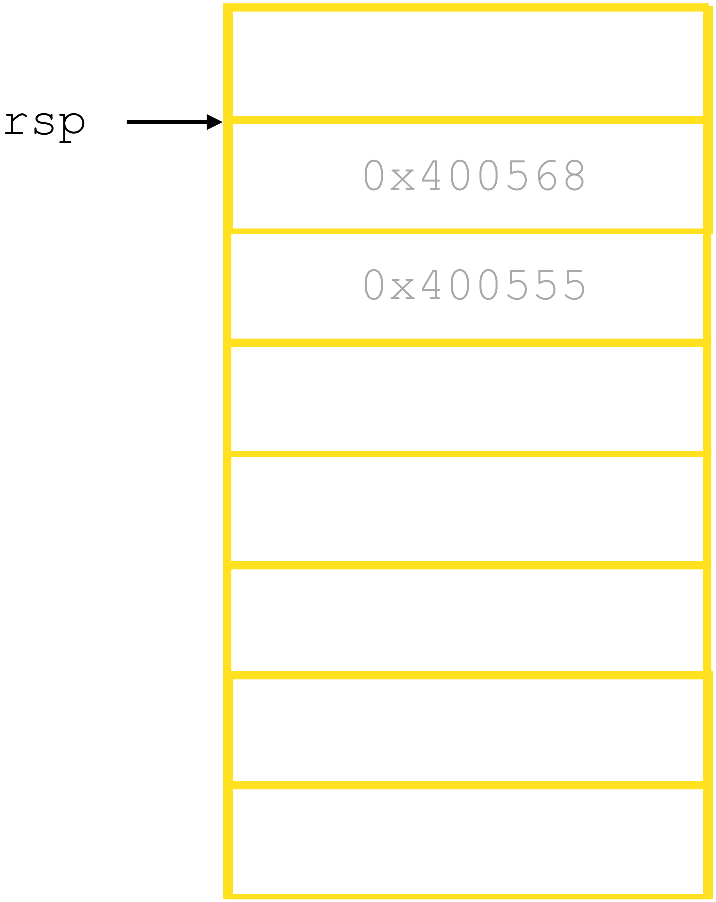
```
0x400540 <last>:
  400540: mov   rax, rdi
  400543: imul  rax, rsi
  400547: ret

0x400548 <first>:
  400548: lea   rsi, [rdi + 0x1]
  40054c: sub   rdi, 0x1
  400550: call  0x400540 <last>
  400555: ret

0x400556 <main>:
  400560: mov   rdi, 4
  400563: call  0x400548 <first>
rsp → 400568: add   rax, 0x13
  40056c: ret
```



What value gets returned by main?

Practice with Value Passing

```
0x400540 <last>:
  400540: mov   rax, rdi
  400543: imul  rax, rsi
  400547: ret

0x400548 <first>:
  400548: lea   rsi, [rdi + 0x1]
  40054c: sub   rdi, 0x1
  400550: call  0x400540 <last>
  400555: ret

0x400556 <main>:
  400560: mov   rdi, 4
  400563: call  0x400548 <first>
  400568: add   rax, 0x13
rsp → 40056c: ret
```



rdi	rsi	rax
3	5	34

Recursion is handled without special consideration

Stack frames mean that each procedure call has private storage

- Saved registers & local variables
- Saved return pointer

Register saving conventions prevent one procedure call from corrupting another's data

- Unless the C code explicitly does so (more later!)

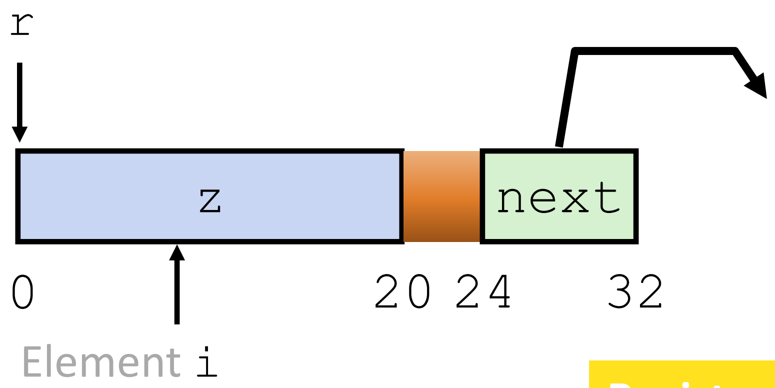
Stack discipline follows call / return pattern

- If P calls Q, then Q returns before P
- Last-In, First-Out

Also works for mutual recursion

- P calls Q; Q calls P

Practice with Structs



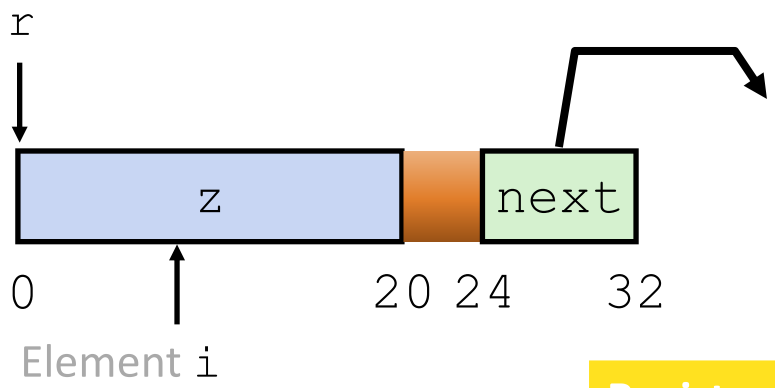
Register	Value
rdi	**p**

```
struct rec {
  int z[5];
  struct rec *next;
};
```

```
exercise:
  mov     rax, rdi
  test    rax, rax
  je      L1
  mov     rdi, [rax + 24]
  test    rdi, rdi
  je      L2
  push    rax
  call    exercise
  add     rsp, 8
L2:
  ret
L1:
  xor     rax, rax
  ret
```

```
struct rec *exercise(struct rec *p) {

}
```

Practice with Structs



Register	Value
rdi	**p**

```
struct rec {
  int z[5];
  struct rec *next;
};
```

```
exercise:
  mov     rax, rdi
  test    rax, rax
  je      L1
  mov     rdi, [rax + 24]
  test    rdi, rdi
  je      L2
  push    rax
  call    exercise
  add     rsp, 8
L2:
  ret
L1:
  xor     rax, rax
  ret
```

```
struct rec *exercise(struct rec *p) {

    struct rec * ret = p;

    if (ret == NULL)
        return NULL;

    p = p->next;

    if (p == NULL)
        return ret;

    return exercise(p);

}
```

Practice Arrays and Loops

Variable	Register
z	rdi
sum	rax
i	rsi

```
array_loop:
  mov   esi, 0
  xor   eax, eax
  jmp   L2

L1:
  add   eax, [rdi + esi*4]
  inc   esi

L2:
  cmp   esi, 5
  jl    L1
  ret
```

```
int array_loop(int z[5]) {
  int sum = __0__;

  int i;

  for(i = _0_; i < _5_; _i++_ )
    sum = _sum+z[i]_;
  }         sum + (*(z+i))

  return _sum_;
}
```

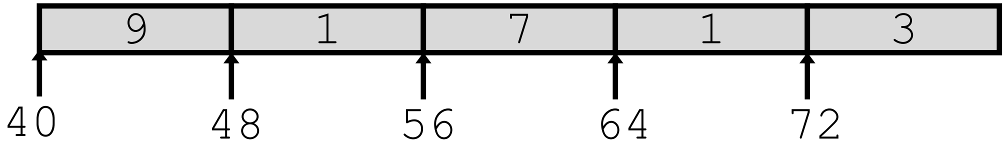
Array Recursion

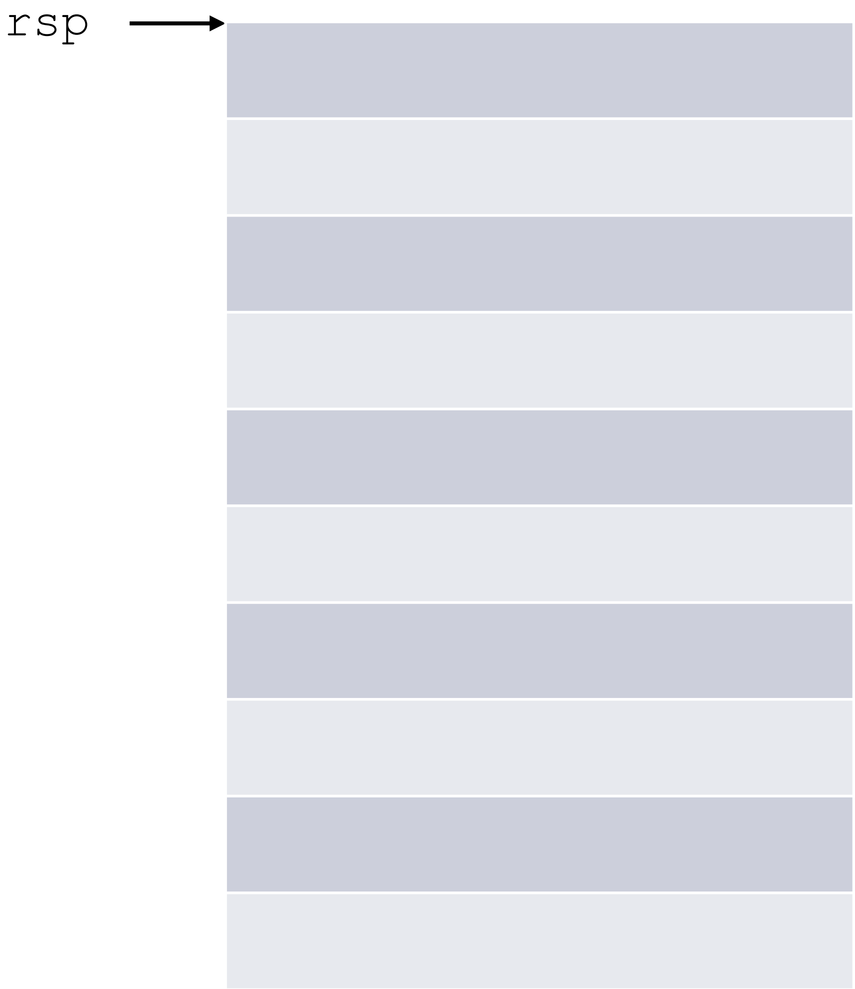
```
array_loop:
  mov   esi, 0
  xor   eax, eax
  jmp   L2

L1:
  add   eax, [rdi + esi*4]
  inc   esi

L2:
  cmp   esi, 5
  jl    L1
  ret
```

```
array_r:
  xor     eax, eax
  cmp     rsi, 5
  jge     L2
  push    rbx
  mov     ebx, [rdi + esi*4]
  inc     esi
  call    array_r
  add     eax, ebx
  pop     rbx

L2:
  ret
```

Array Recursion

Memory address in decimal:



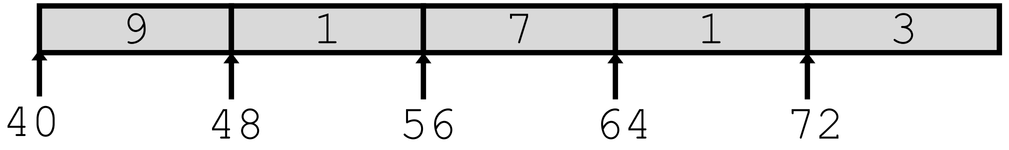


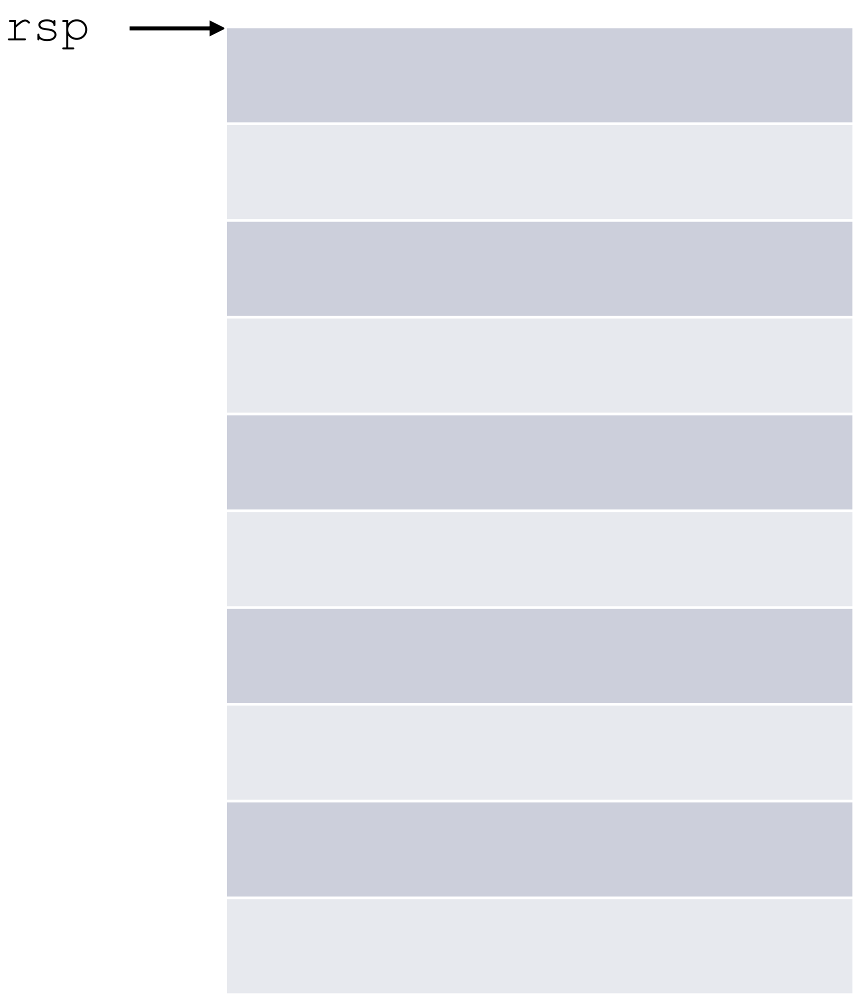

```
long array_r(long z[5]) {
array_r:
  xor     rax, rax
  cmp     rsi, 5  ; Set SF?
  jge     L2        ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx

L2:
  ret
```

rdi	rsi	rax	rbx
?	?	?	?

Needs initialized prior to calling `array_r`.

Array Recursion



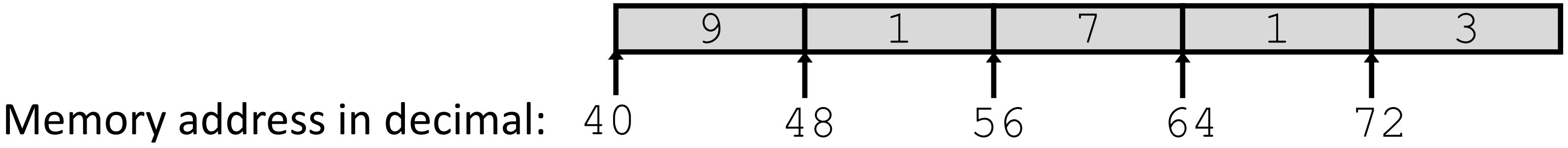


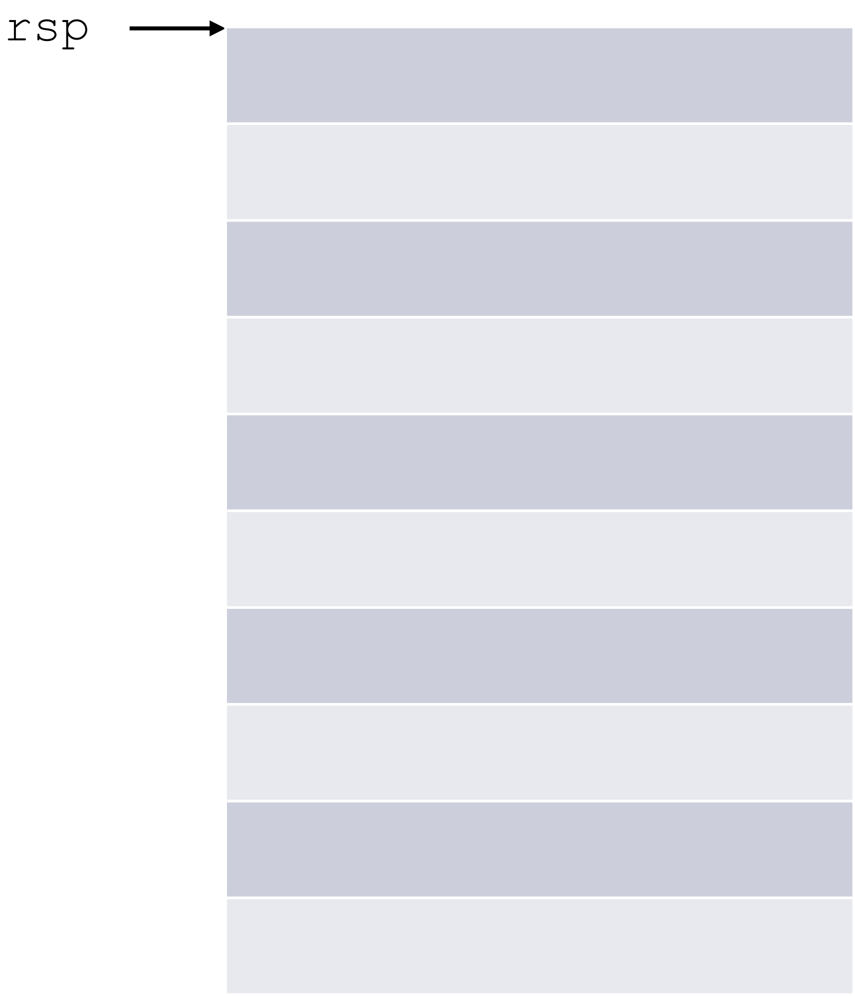

```
long array_r(long z[5]) {
array_r:
rip →  xor     rax, rax
       cmp     rsi, 5   ; Set SF?
       jge     L2       ; jump if ~SF
       push    rbx
       mov     rbx, [rdi + rsi*8]
       inc     rsi
       call    array_r
       add     rax, rbx
       pop     rbx

L2:
       ret
```

rdi	rsi	rax	rbx
40	0	X	X

Needs initialized prior to calling `array_r`.

Array Recursion






```
long array_r(long z[5]) {
```

rip → (at `cmp`)

```
array_r:
  xor     rax, rax
  cmp     rsi, 5   ; Set SF?
  jge     L2       ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	0	0	X

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:

40	48	56	64	72
9	1	7	1	3

rip → push rbx

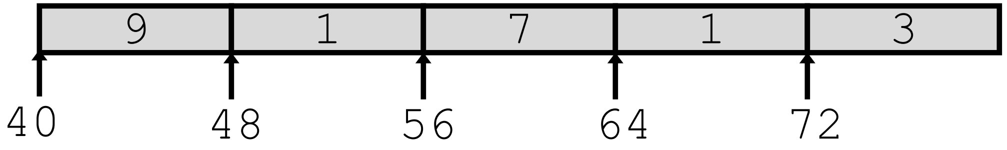
```
long array_r(long z[5]) {
array_r:
   xor     rax, rax
   cmp     rsi, 5   ; Set SF?
   jge     L2       ; jump if ~SF
   push    rbx
   mov     rbx, [rdi + rsi*8]
   inc     rsi
   call    array_r
   add     rax, rbx
   pop     rbx

L2:
   ret
```

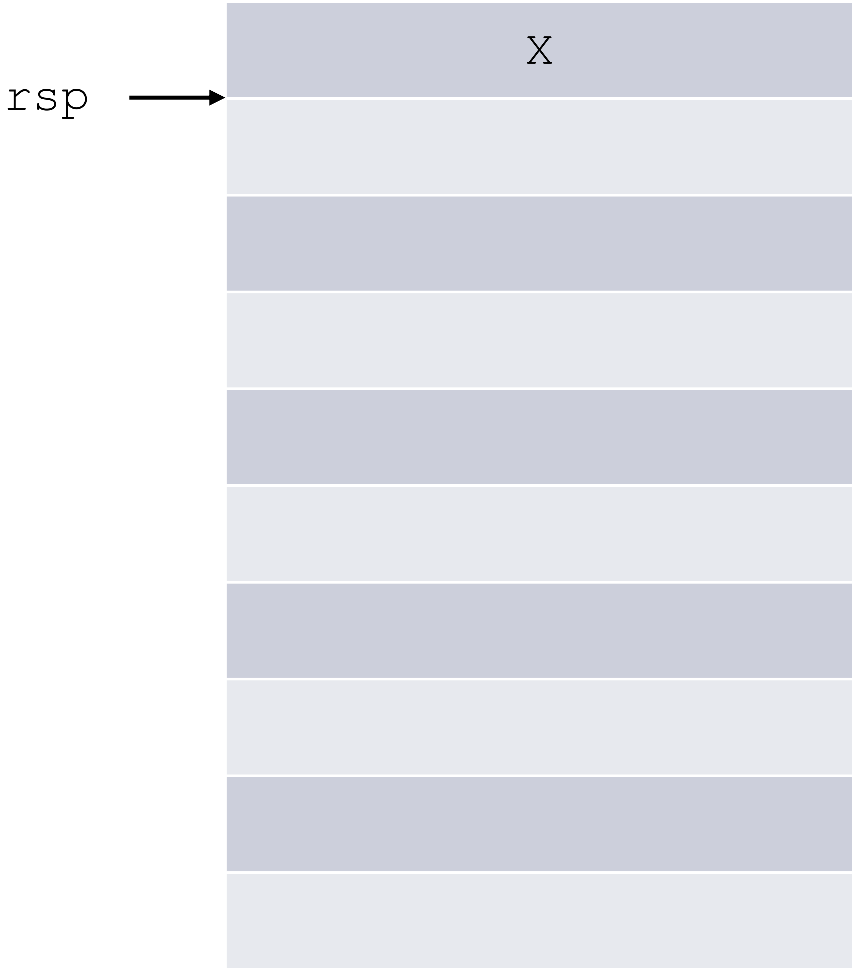
rdi	rsi	rax	rbx
40	0	0	X

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:






```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5  ; Set SF?
  jge    L2      ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]     <- rip
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

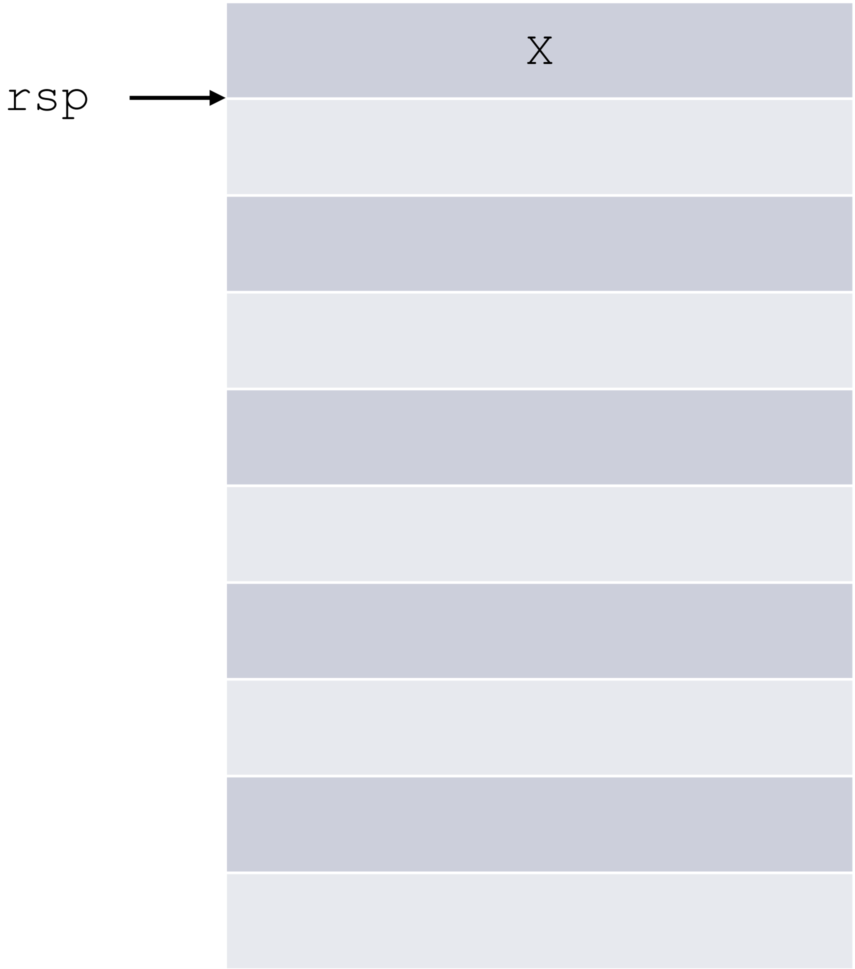
rdi	rsi	rax	rbx
40	0	0	X

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:

40	48	56	64	72
9	1	7	1	3

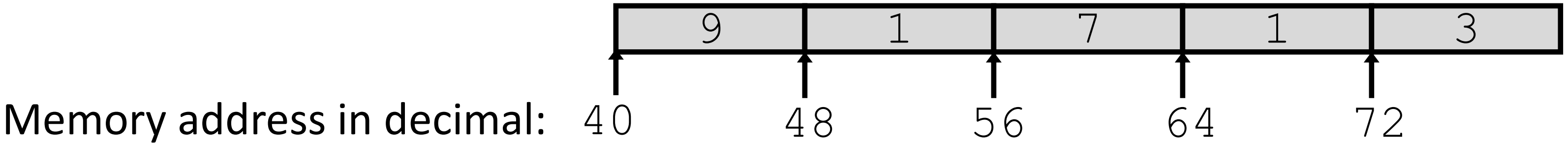



```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5  ; Set SF?
  jge    L2      ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi          <- rip
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	0	0	9

Needs initialized prior to calling `array_r`.

Array Recursion



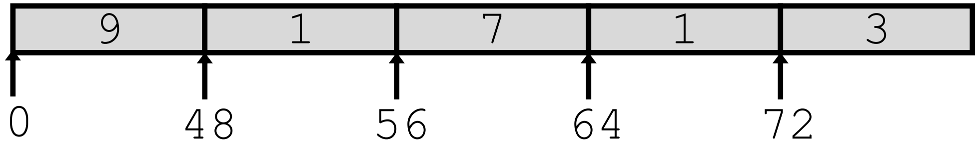

```
long array_r(long z[5]) {
array_r:
   xor     rax, rax
   cmp     rsi, 5  ; Set SF?
   jge     L2      ; jump if ~SF
   push    rbx
   mov     rbx, [rdi + rsi*8]
   inc     rsi
   call    array_r
   add     rax, rbx
   pop     rbx

L2:
   ret
```

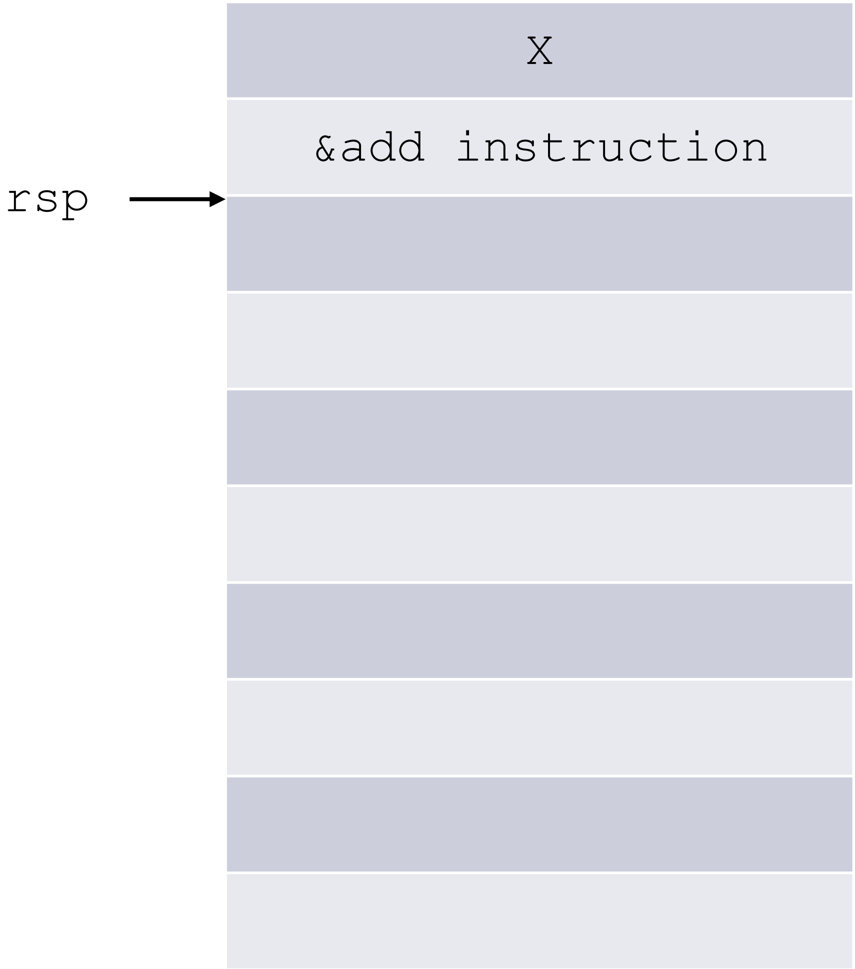
rdi	rsi	rax	rbx
40	1	0	9

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:





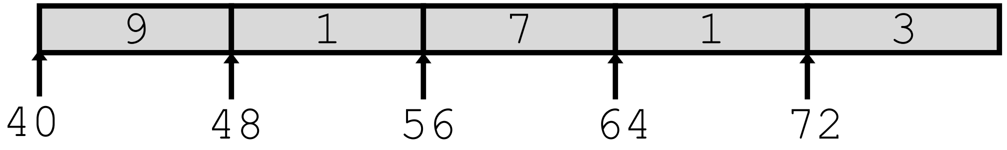
rip → xor rax, rax

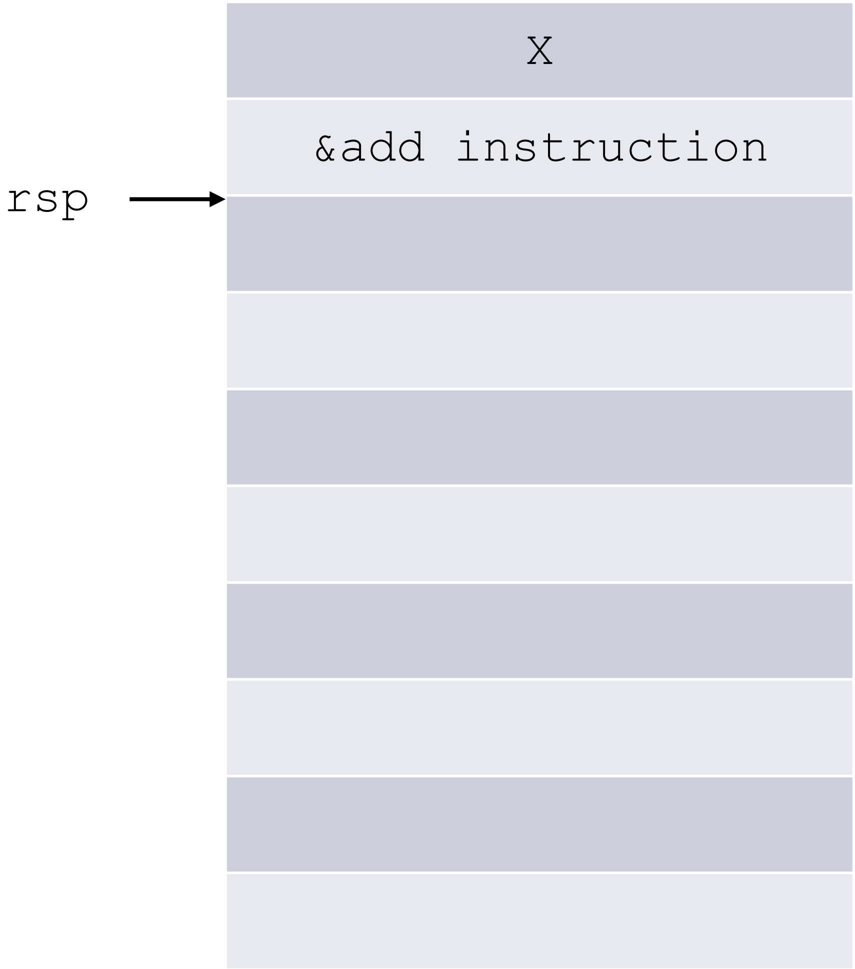
```
long array_r(long z[5]) {
array_r:
    xor     rax, rax
    cmp     rsi, 5      ; Set SF?
    jge     L2          ; jump if ~SF
    push    rbx
    mov     rbx, [rdi + rsi*8]
    inc     rsi
    call    array_r
    add     rax, rbx
    pop     rbx

L2:
    ret
```

rdi	rsi	rax	rbx
40	1	0	9

Needs initialized prior to calling `array_r`.

Array Recursion



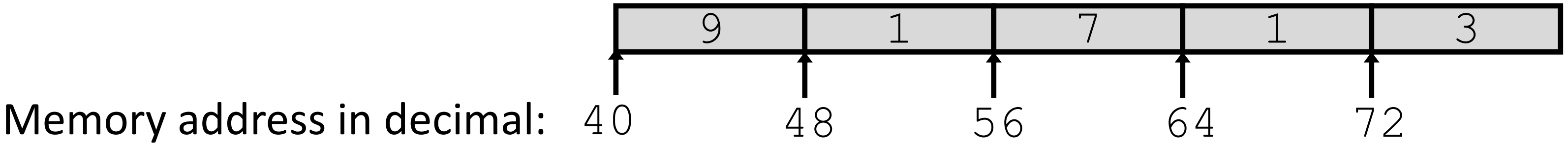


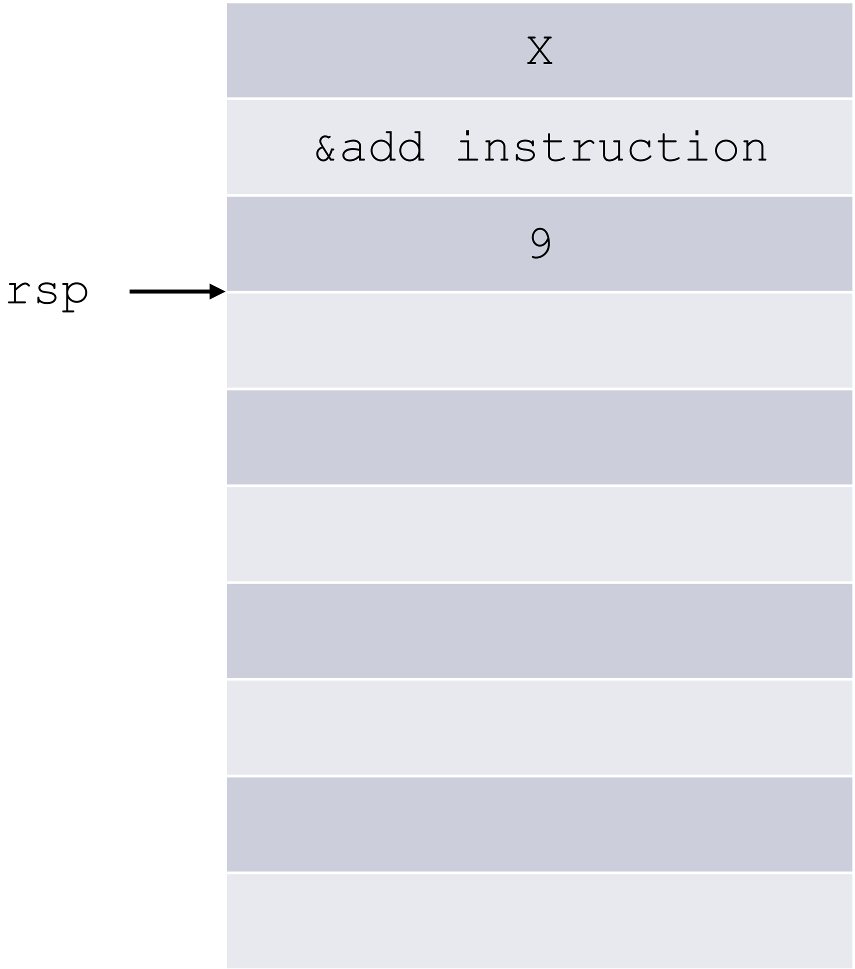

```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5   ; Set SF?
  jge    L2       ; jump if ~SF
  push   rbx          <- rip
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	1	0	9

Needs initialized prior to calling array_r.

Array Recursion






```
long array_r(long z[5]) {
```

rip →

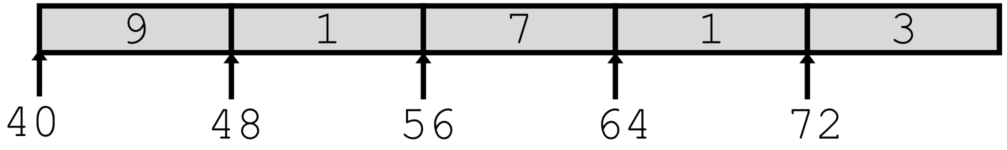
```
array_r:
  xor     rax, rax
  cmp     rsi, 5   ; Set SF?
  jge     L2       ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx

L2:
  ret
```

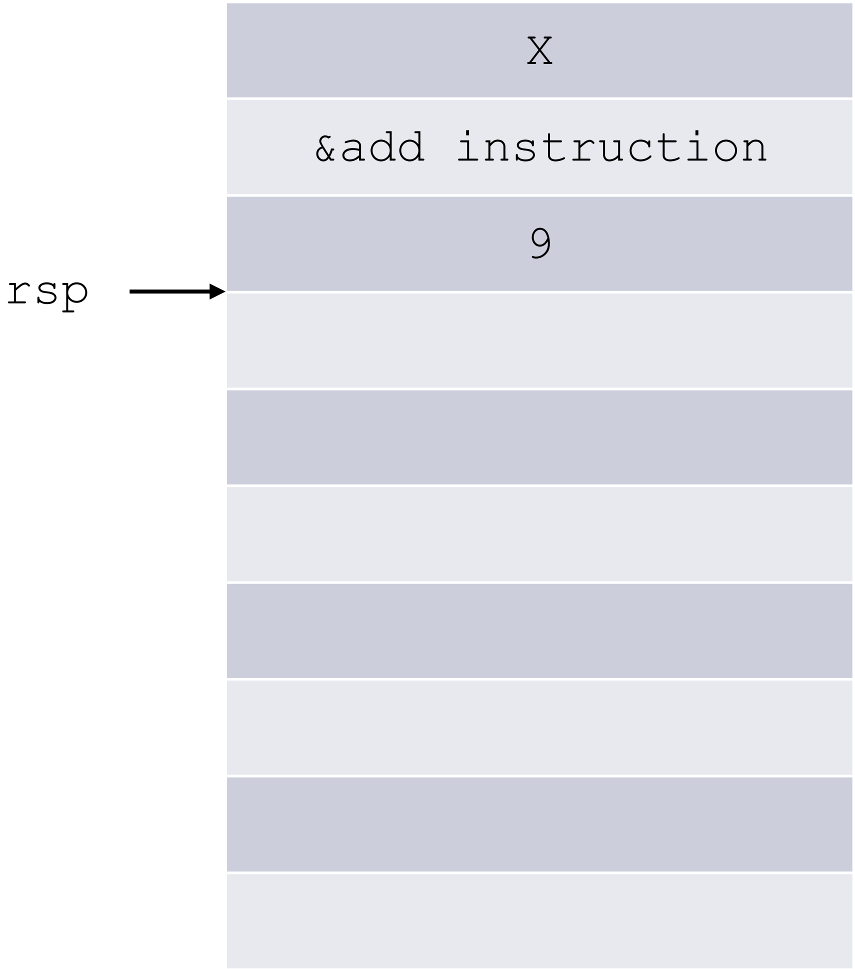
rdi	rsi	rax	rbx
40	1	0	9

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:





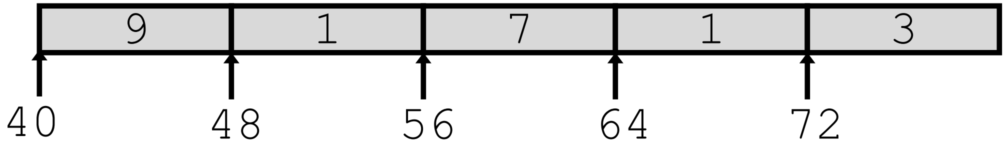

```
long array_r(long z[5]) {
array_r:
  xor     rax, rax
  cmp     rsi, 5  ; Set SF?
  jge     L2      ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx

L2:
  ret
```

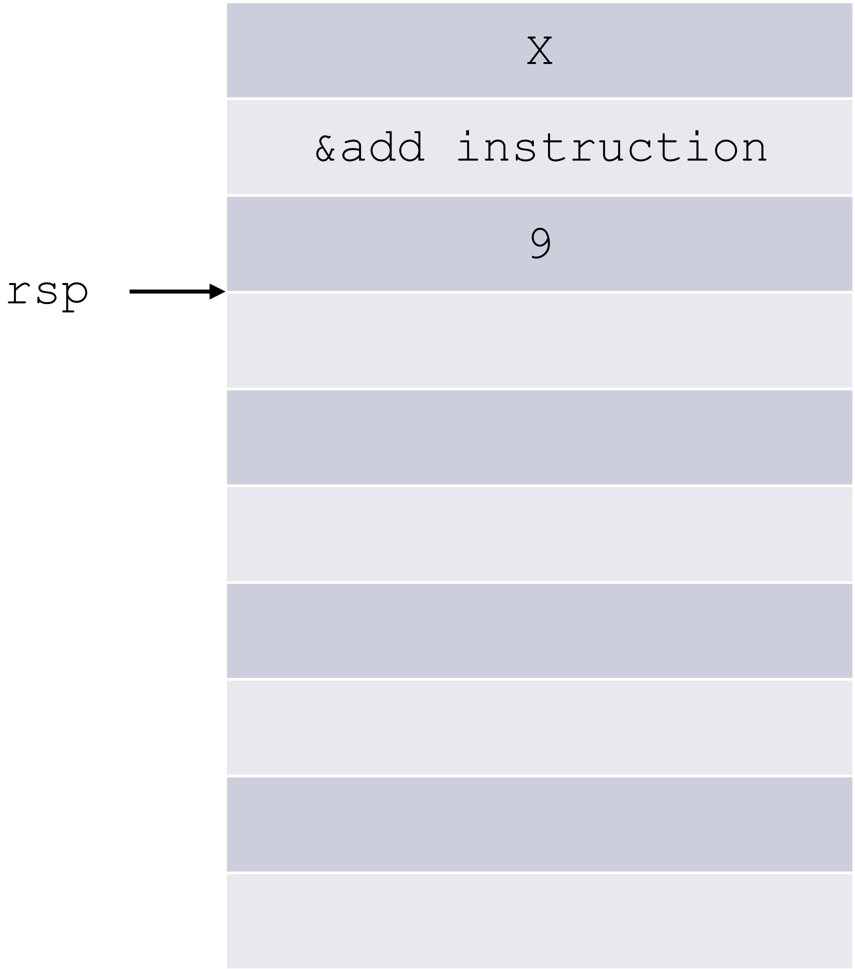
rip → inc rsi

rdi	rsi	rax	rbx
40	1	0	1

Needs initialized prior to calling `array_r`.

Array Recursion





long array_r(long z[5]) {

```
array_r:
  xor    rax, rax
  cmp    rsi, 5   ; Set SF?
  jge    L2       ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

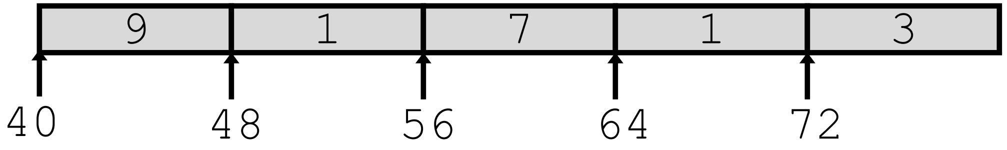
rip → call array_r

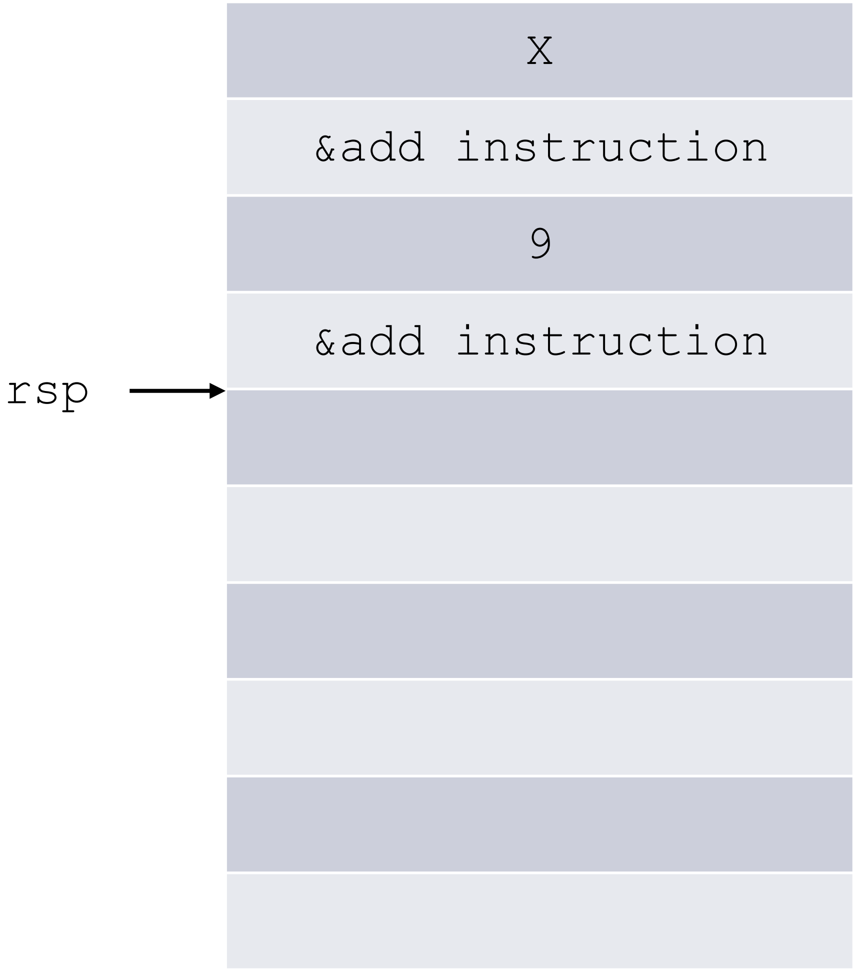
rdi	rsi	rax	rbx
40	2	0	1

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:



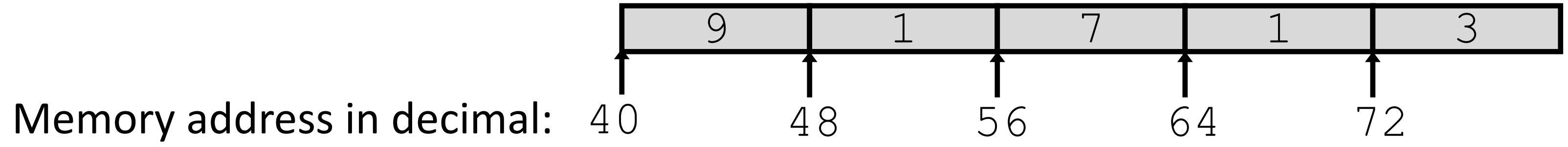


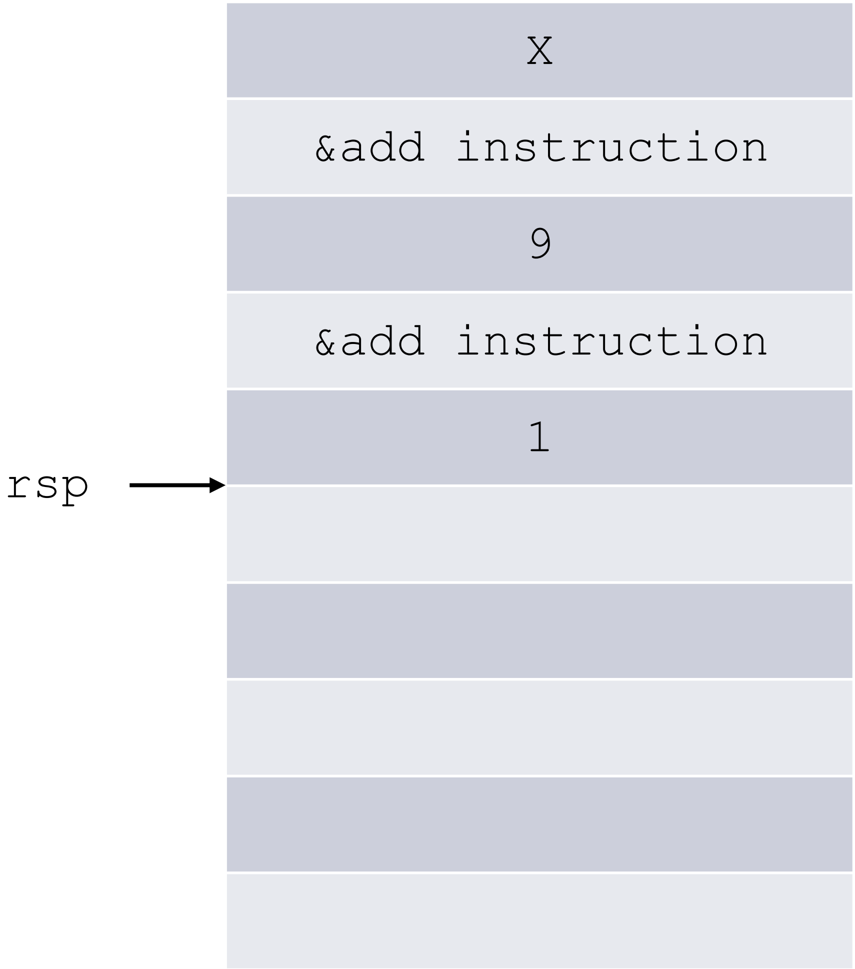

```
long array_r(long z[5]) {
array_r:
rip →  xor     rax, rax
       cmp     rsi, 5   ; Set SF?
       jge     L2       ; jump if ~SF
       push    rbx
       mov     rbx, [rdi + rsi*8]
       inc     rsi
       call    array_r
       add     rax, rbx
       pop     rbx

L2:
       ret
```

rdi	rsi	rax	rbx
40	2	0	1

Needs initialized prior to calling `array_r`.

Array Recursion





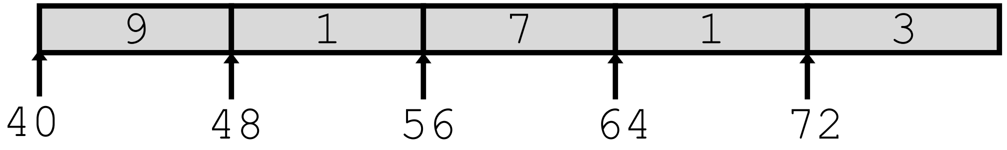

```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5    ; Set SF?
  jge    L2        ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r          <- rip
  add    rax, rbx
  pop    rbx

L2:
  ret
```

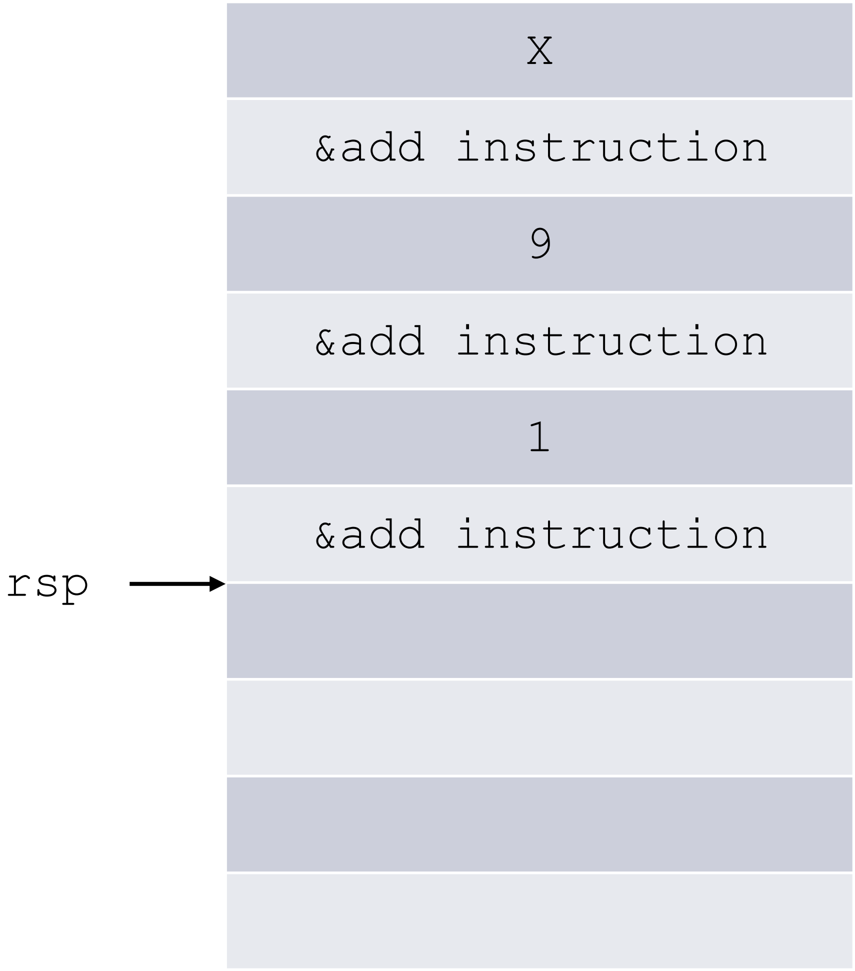
rdi	rsi	rax	rbx
40	3	0	7

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:



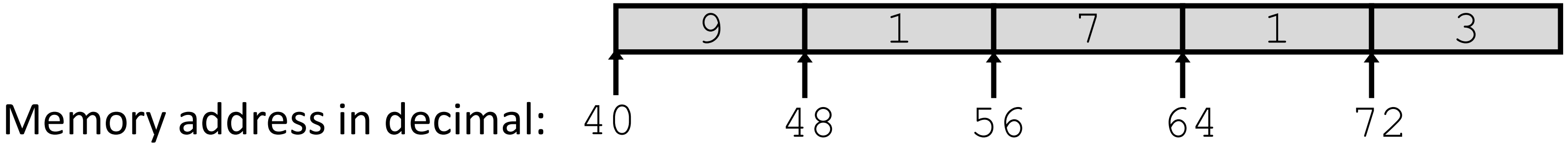


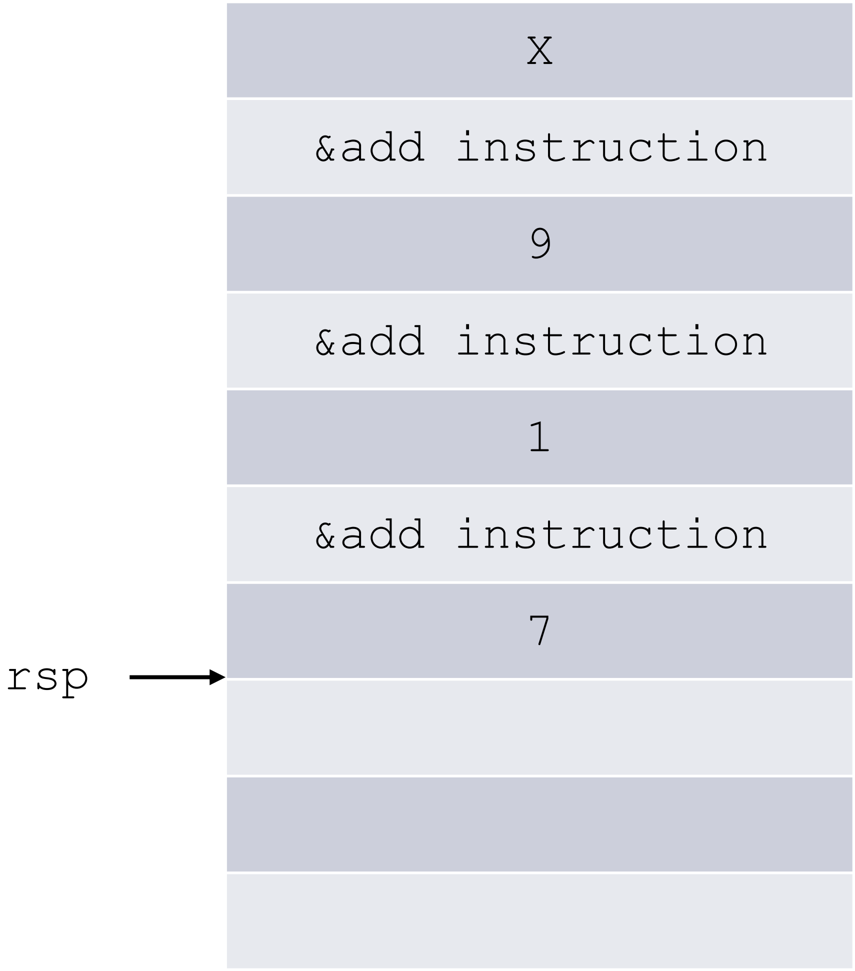

```
long array_r(long z[5]) {
array_r:
rip →  xor     rax, rax
       cmp     rsi, 5   ; Set SF?
       jge     L2       ; jump if ~SF
       push    rbx
       mov     rbx, [rdi + rsi*8]
       inc     rsi
       call    array_r
       add     rax, rbx
       pop     rbx

L2:
       ret
```

rdi	rsi	rax	rbx
40	3	0	7

Needs initialized prior to calling `array_r`.

Array Recursion






```
long array_r(long z[5]) {
array_r:
  xor     rax, rax
  cmp     rsi, 5   ; Set SF?
  jge     L2       ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r      <- rip
  add     rax, rbx
  pop     rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	4	0	1

Needs initialized prior to calling `array_r`.

Array Recursion

9	1	7	1	3

Memory address in decimal: 40 48 56 64 72

X
&add instruction
9
&add instruction
1
&add instruction
7
&add instruction

rsp →

```
long array_r(long z[5]) {
array_r:
    xor     rax, rax
    cmp     rsi, 5   ; Set SF?
    jge     L2       ; jump if ~SF
    push    rbx
    mov     rbx, [rdi + rsi*8]
    inc     rsi
    call    array_r
    add     rax, rbx
    pop     rbx

L2:
    ret
```

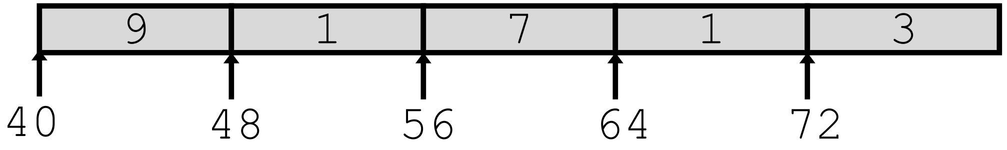
rip → xor

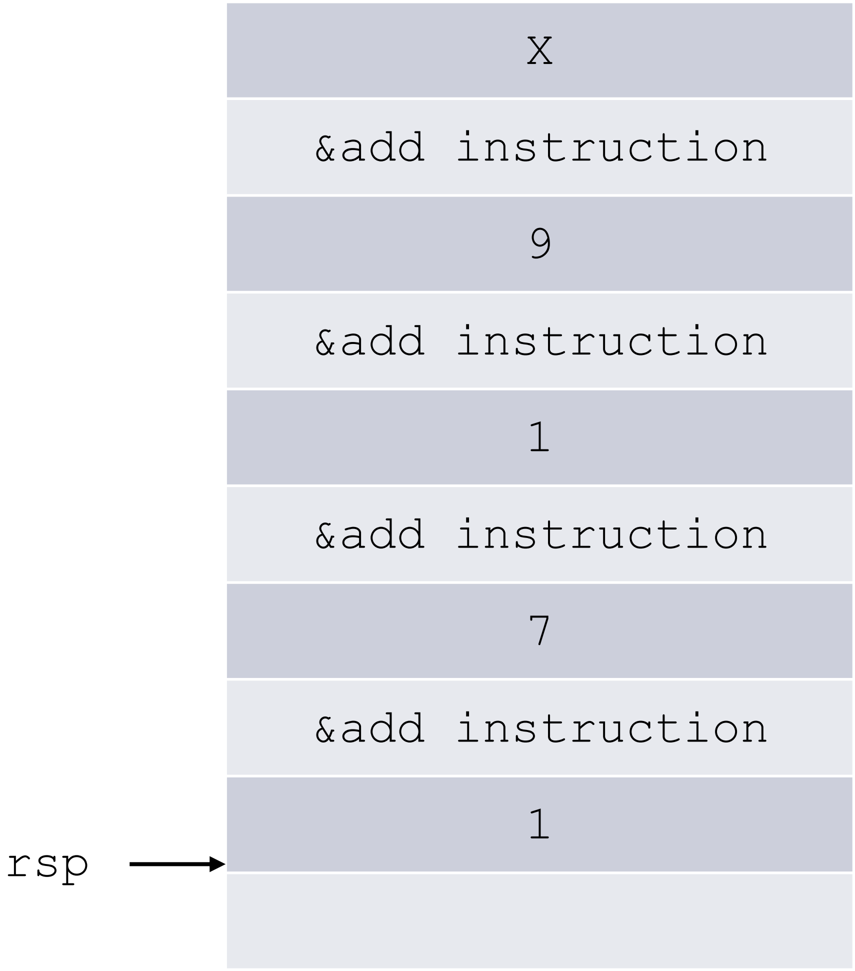
rdi	rsi	rax	rbx
40	4	0	1

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:






```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5     ; Set SF?
  jge    L2         ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

rip → call array_r

rdi	rsi	rax	rbx
40	5	0	3

Needs initialized prior to calling `array_r`.

Array Recursion

9	1	7	1	3

Memory address in decimal: 40 48 56 64 72

X
&add instruction
9
&add instruction
1
&add instruction
7
&add instruction
1
&add instruction

rsp →

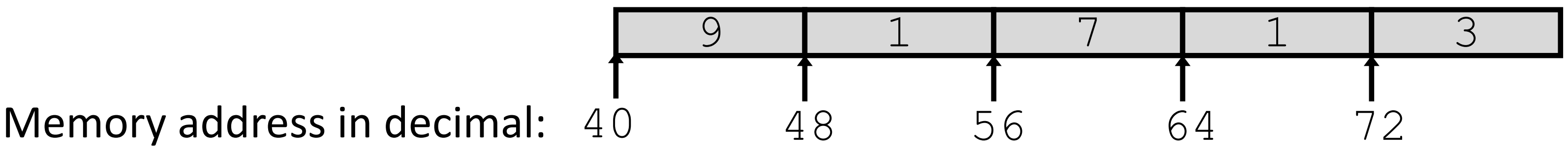
```
long array_r(long z[5]) {
```

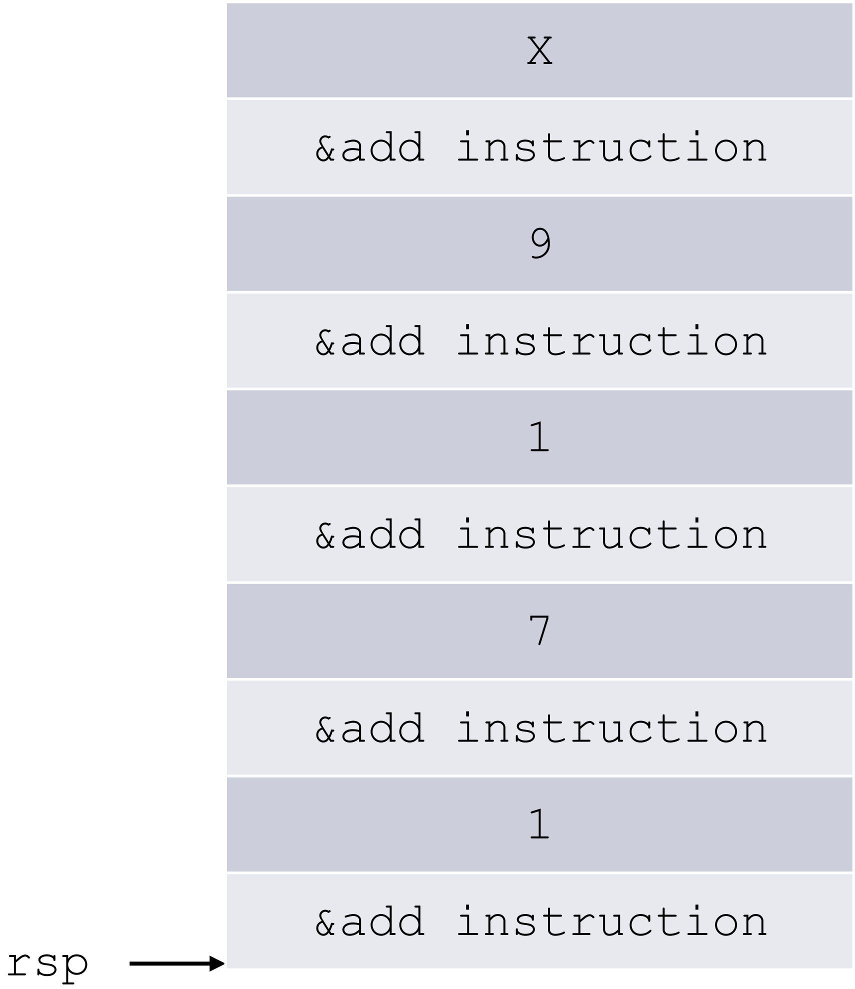
```
array_r:
rip →  xor     rax, rax
       cmp     rsi, 5      ; Set SF?
       jge     L2          ; jump if ~SF
       push    rbx
       mov     rbx, [rdi + rsi*8]
       inc     rsi
       call    array_r
       add     rax, rbx
       pop     rbx

L2:
       ret
```

rdi	rsi	rax	rbx
40	5	0	3

Needs initialized prior to calling `array_r`.

Array Recursion






```
long array_r(long z[5]) {
```

rip → cmp

```
array_r:
  xor     rax, rax
  cmp     rsi, 5   ; Set SF?
  jge     L2       ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	5	0	3

Needs initialized prior to calling `array_r`.

Array Recursion

9	1	7	1	3

Memory address in decimal: 40 48 56 64 72

```
long array_r(long z[5]) {
```

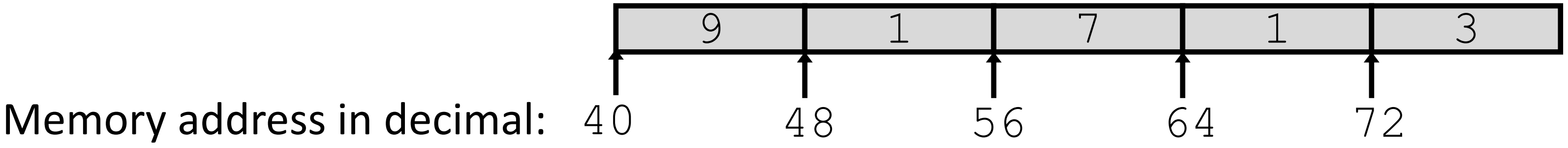
Stack
X
&add instruction
9
&add instruction
1
&add instruction
7
&add instruction
1
&add instruction ← rsp

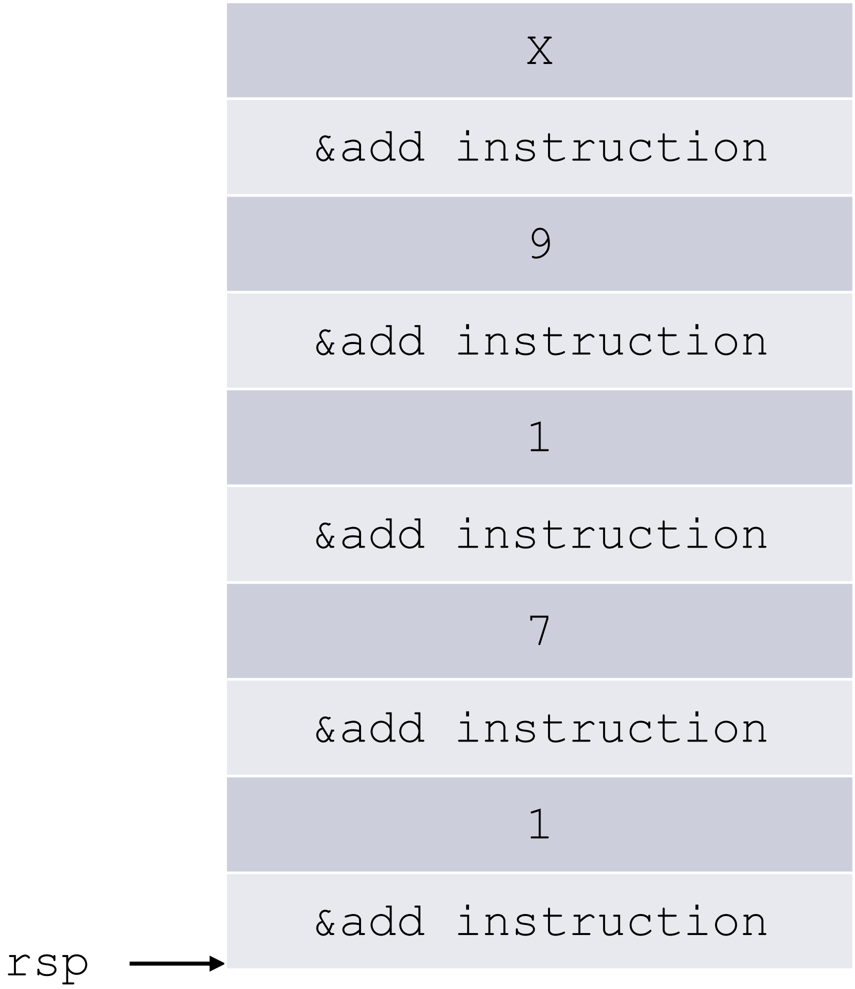
```
array_r:
  xor    rax, rax
  cmp    rsi, 5    ; Set SF?
  jge    L2        ; jump if ~SF      ← rip
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	5	0	3

Needs initialized prior to calling `array_r`.

Array Recursion






```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5    ; Set SF?
  jge    L2        ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

rip → ret

rdi	rsi	rax	rbx
40	5	0	3

Needs initialized prior to calling `array_r`.

Array Recursion

9	1	7	1	3

Memory address in decimal: 40 48 56 64 72

X
&add instruction
9
&add instruction
1
&add instruction
7
&add instruction
1 ← rsp
&add instruction

```
long array_r(long z[5]) {
```

```
array_r:
  xor     rax, rax
  cmp     rsi, 5   ; Set SF?
  jge     L2       ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx     ← rip
  pop     rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	5	0	3

Needs initialized prior to calling `array_r`.

Array Recursion

9	1	7	1	3

Memory address in decimal: 40 48 56 64 72

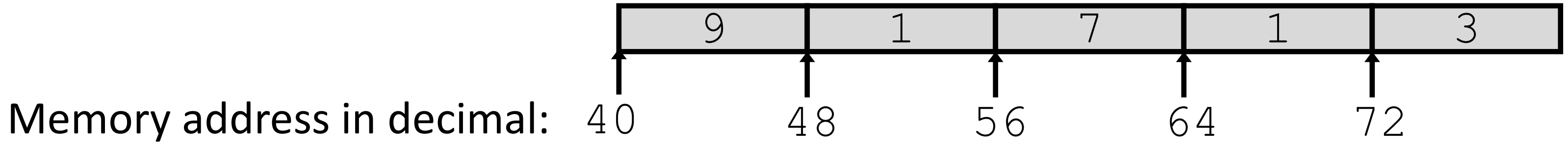
Stack
X
&add instruction
9
&add instruction
1
&add instruction
7
&add instruction
1 ← rsp
&add instruction

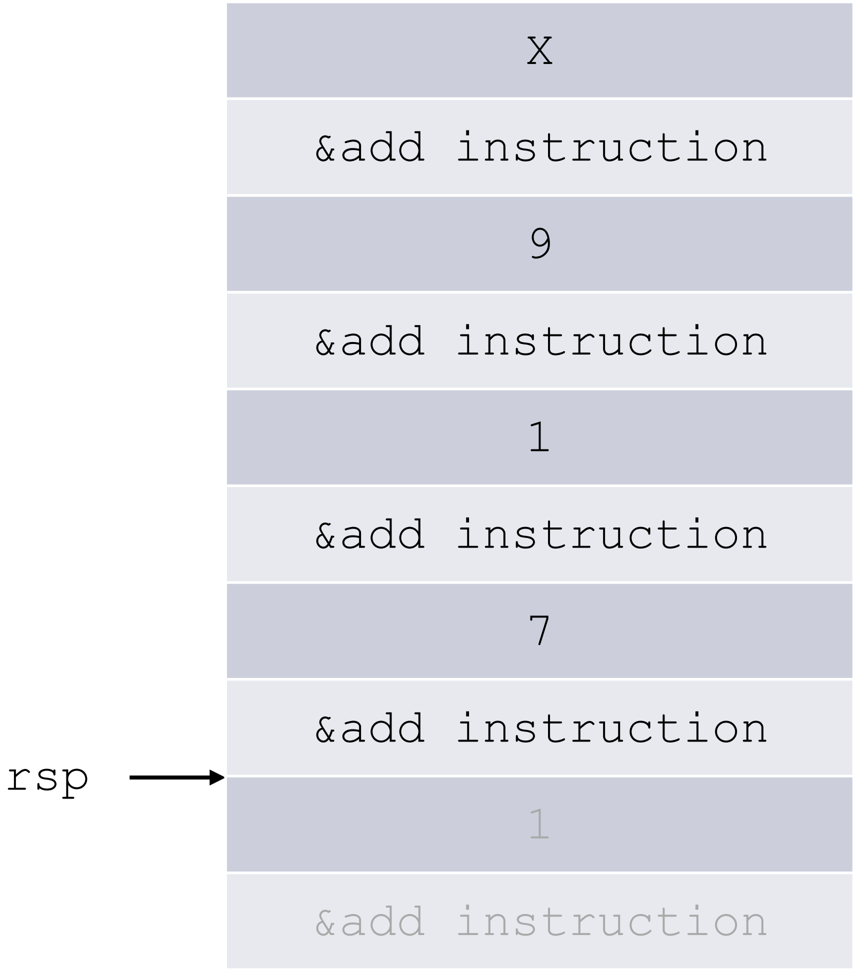
```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5   ; Set SF?
  jge    L2       ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx        ← rip

L2:
  ret
```

rdi	rsi	rax	rbx
40	5	3	3

Needs initialized prior to calling `array_r`.

Array Recursion



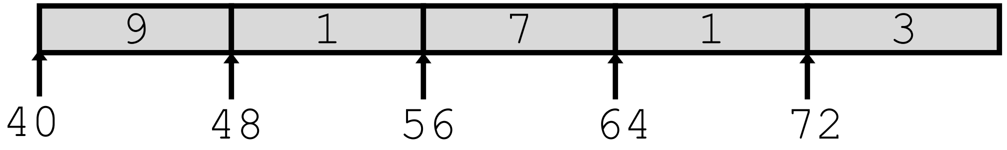


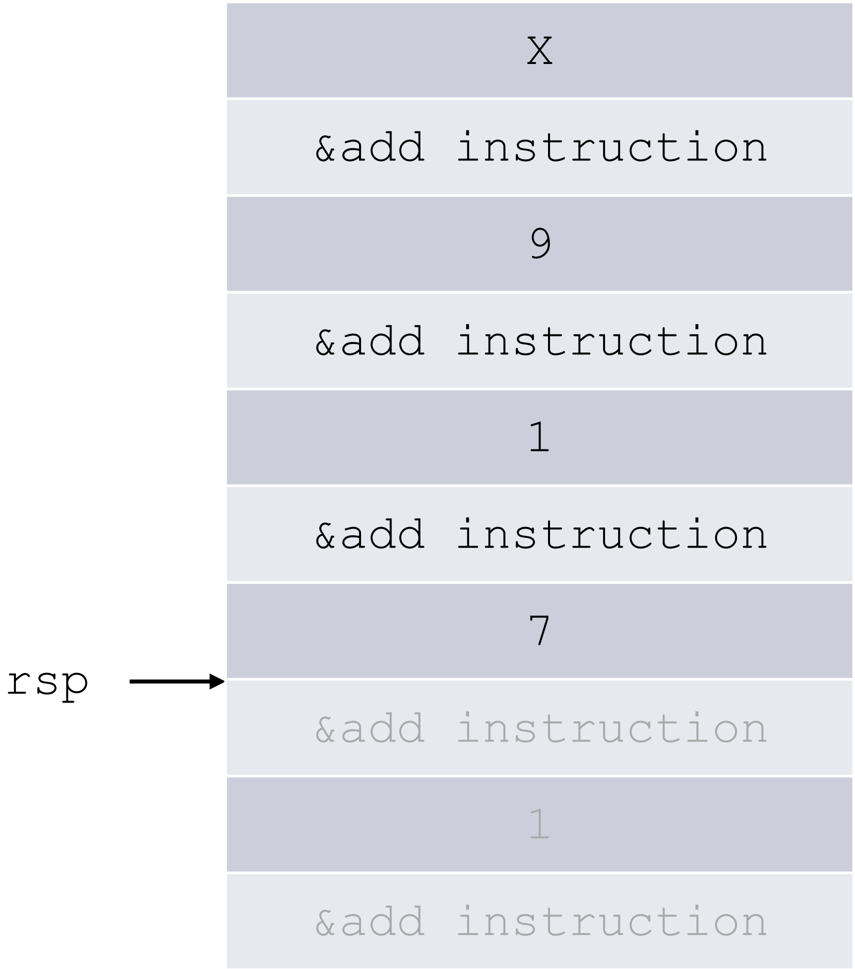

```
long array_r(long z[5]) {
array_r:
    xor     rax, rax
    cmp     rsi, 5   ; Set SF?
    jge     L2       ; jump if ~SF
    push    rbx
    mov     rbx, [rdi + rsi*8]
    inc     rsi
    call    array_r
    add     rax, rbx
    pop     rbx

L2:
rip →  ret
```

rdi	rsi	rax	rbx
40	5	3	1

Needs initialized prior to calling `array_r`.

Array Recursion





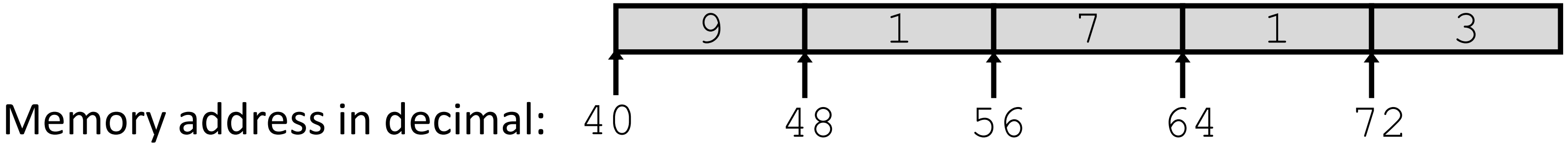

```
long array_r(long z[5]) {
array_r:
  xor    rax, rax
  cmp    rsi, 5   ; Set SF?
  jge    L2       ; jump if ~SF
  push   rbx
  mov    rbx, [rdi + rsi*8]
  inc    rsi
  call   array_r
  add    rax, rbx
  pop    rbx

L2:
  ret
```

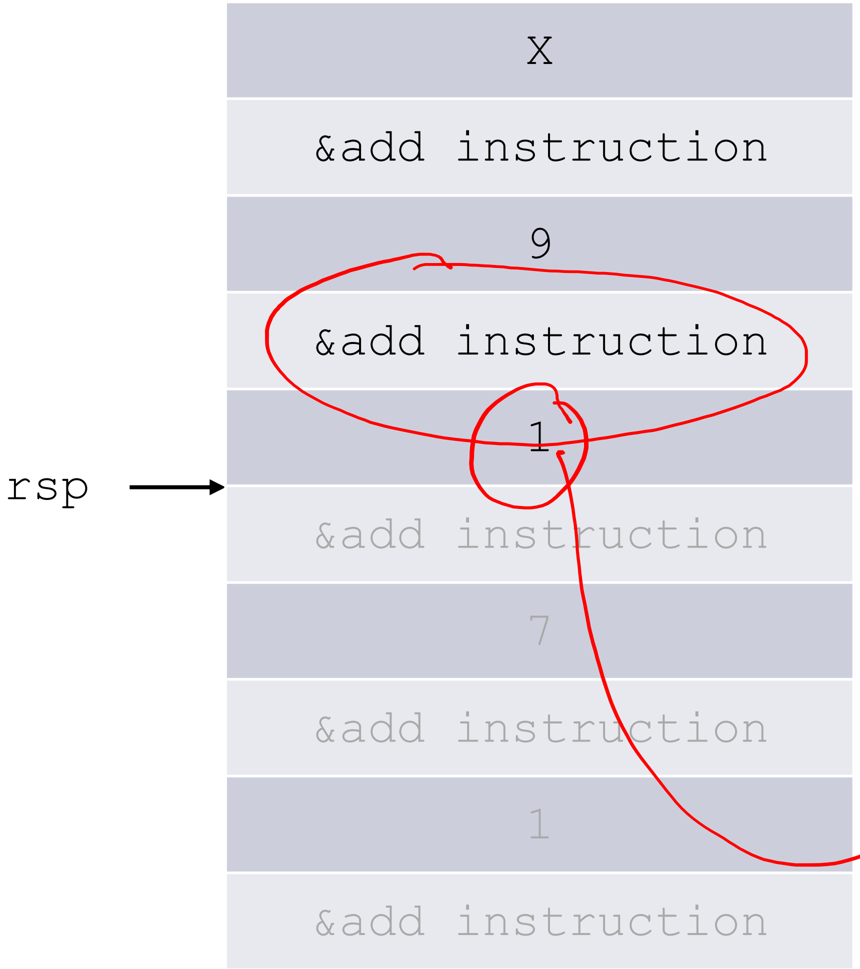
rip → add rax, rbx

rdi	rsi	rax	rbx
40	5	3	1

Needs initialized prior to calling `array_r`.

Array Recursion





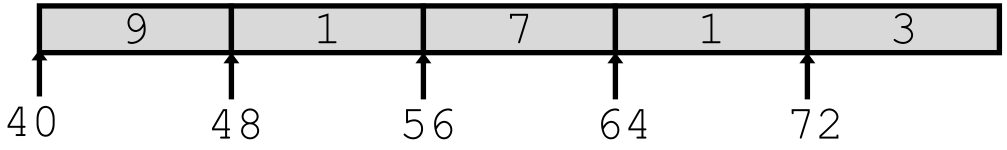

```
long array_r(long z[5]) {
array_r:
    xor     rax, rax
    cmp     rsi, 5      ; Set SF?
    jge     L2          ; jump if ~SF
    push    rbx
    mov     rbx, [rdi + rsi*8]
    inc     rsi
    call    array_r
    add     rax, rbx    <- rip
    pop     rbx

L2:
    ret
```

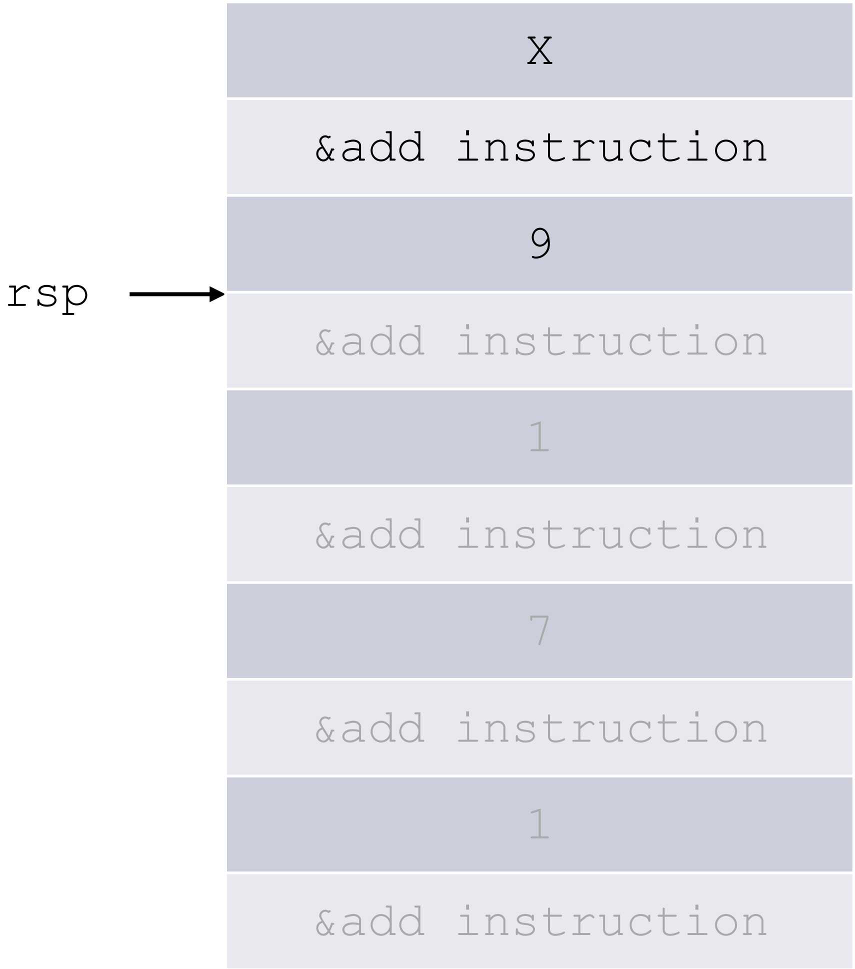
rdi	rsi	rax	rbx
40	5	4	7

Needs initialized prior to calling `array_r`.

Array Recursion

Memory address in decimal:



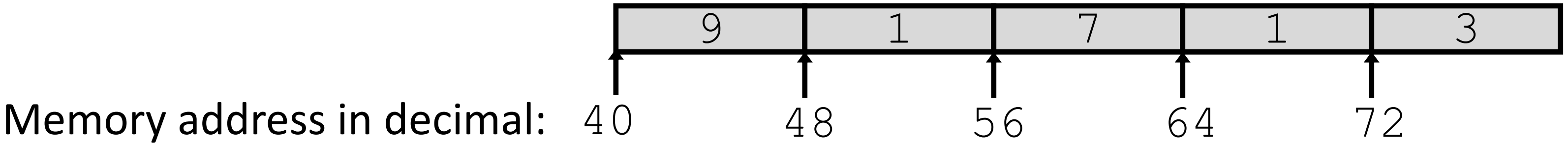


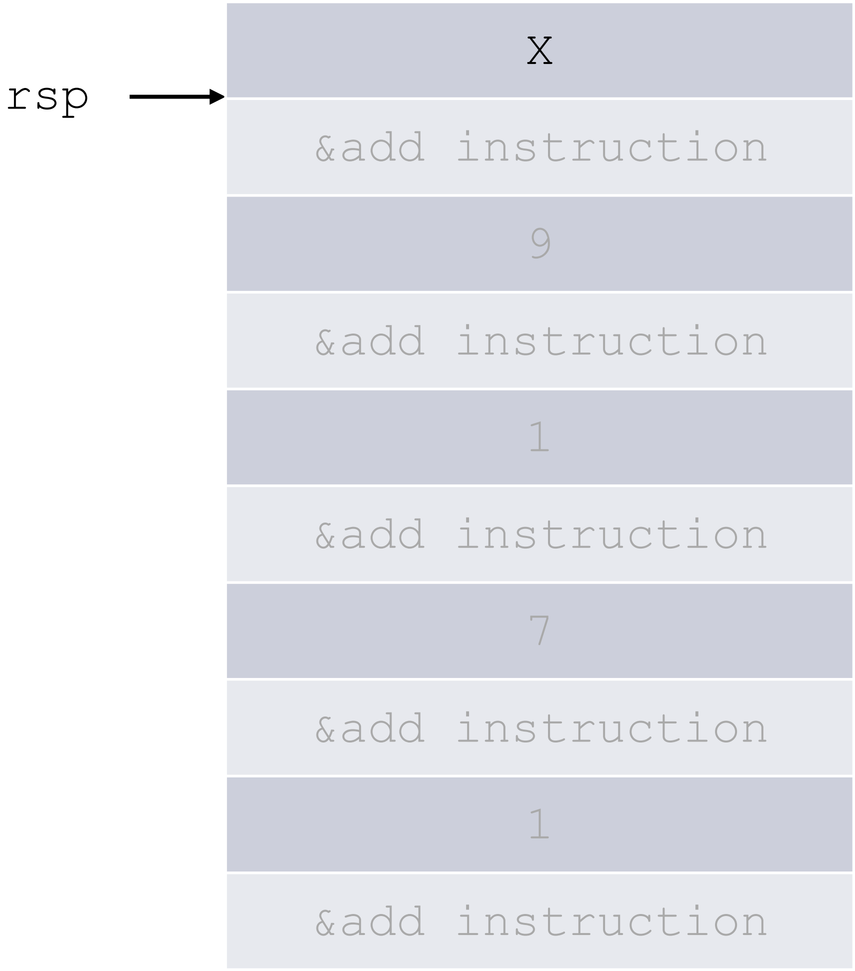

```
long array_r(long z[5]) {
array_r:
  xor     rax, rax
  cmp     rsi, 5    ; Set SF?
  jge     L2        ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx      <- rip
  pop     rbx

L2:
  ret
```

rdi	rsi	rax	rbx
40	5	11	1

Needs initialized prior to calling `array_r`.

Array Recursion






```
long array_r(long z[5]) {
```

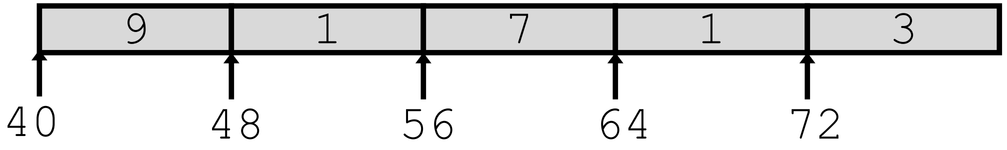
```
array_r:
  xor     rax, rax
  cmp     rsi, 5   ; Set SF?
  jge     L2       ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx

L2:
  ret
```

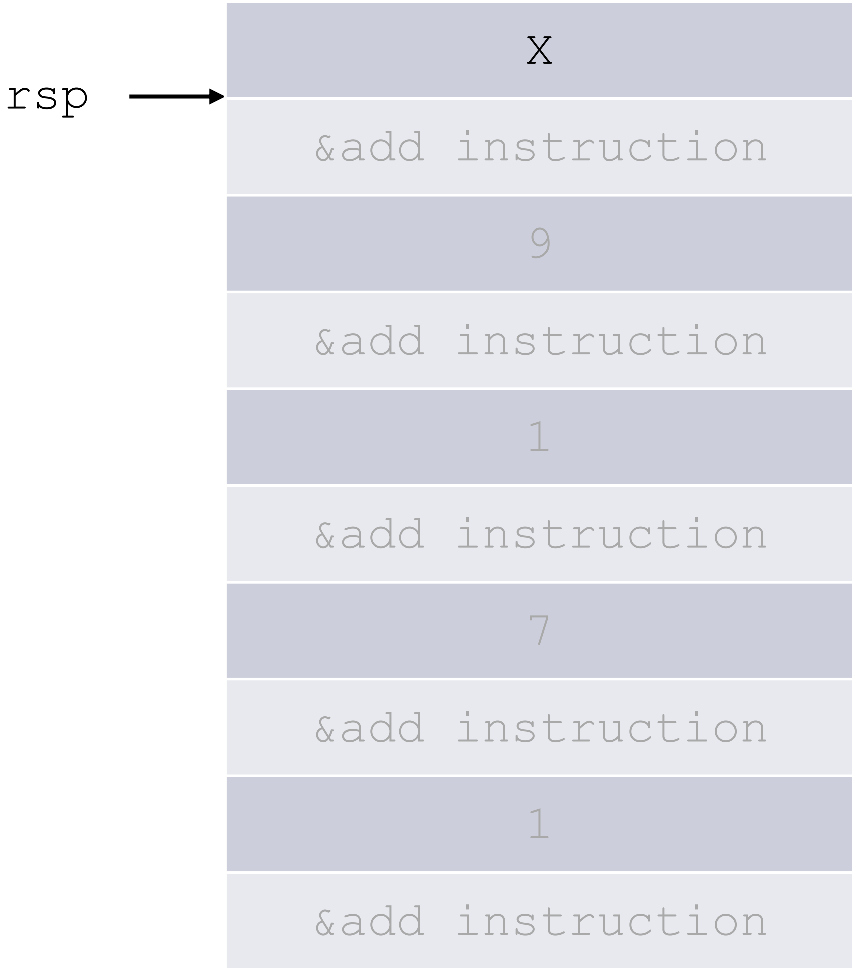
rip → add rax, rbx

rdi	rsi	rax	rbx
40	5	12	9

Needs initialized prior to calling `array_r`.

Array Recursion



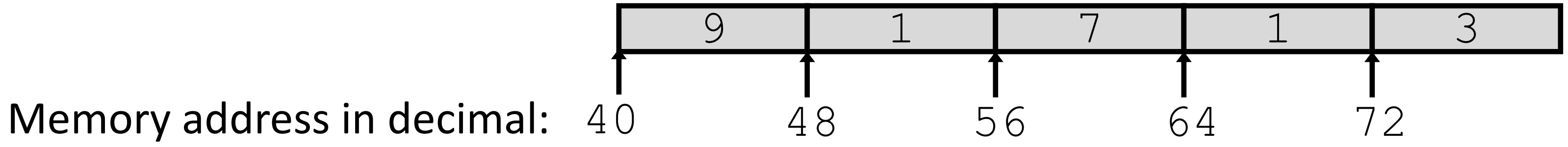


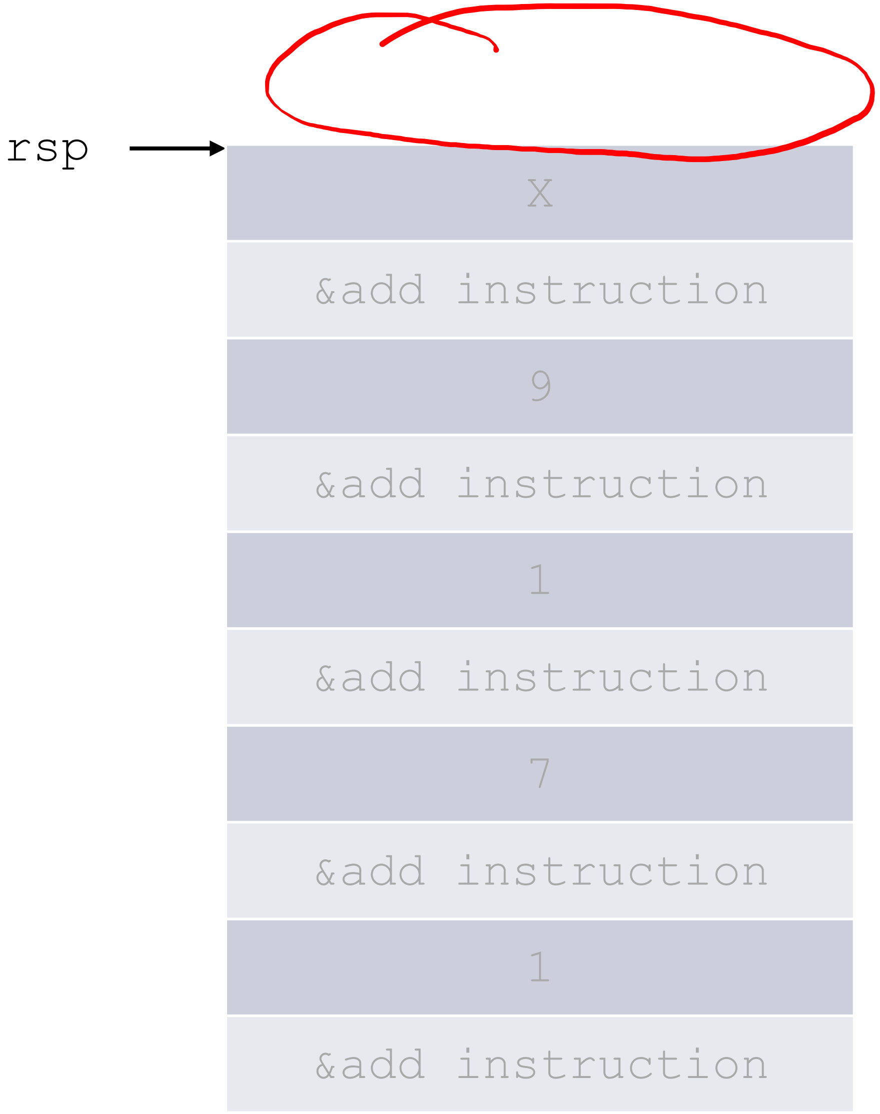

```
long array_r(long z[5]) {
array_r:
   xor    rax, rax
   cmp    rsi, 5   ; Set SF?
   jge    L2       ; jump if ~SF
   push   rbx
   mov    rbx, [rdi + rsi*8]
   inc    rsi
   call   array_r
   add    rax, rbx
   pop    rbx          <- rip

L2:
   ret
```

rdi	rsi	rax	rbx
40	5	21	9

Needs initialized prior to calling `array_r`.

Array Recursion






```
long array_r(long z[5]) {

array_r:
  xor     rax, rax
  cmp     rsi, 5  ; Set SF?
  jge     L2      ; jump if ~SF
  push    rbx
  mov     rbx, [rdi + rsi*8]
  inc     rsi
  call    array_r
  add     rax, rbx
  pop     rbx      <- rip

L2:
  ret
```

rdi	rsi	rax	rbx
40	5	21	X

Needs initialized prior to calling `array_r`.