MACHINE LEVEL PROGRAMMING 3: PROCEDURES

CS 045

Computer Organization and Architecture

Prof. Donald J. Patterson

Adapted from Bryant and O'Hallaron,
Computer Systems:
A Programmer's Perspective, Third Edition

PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

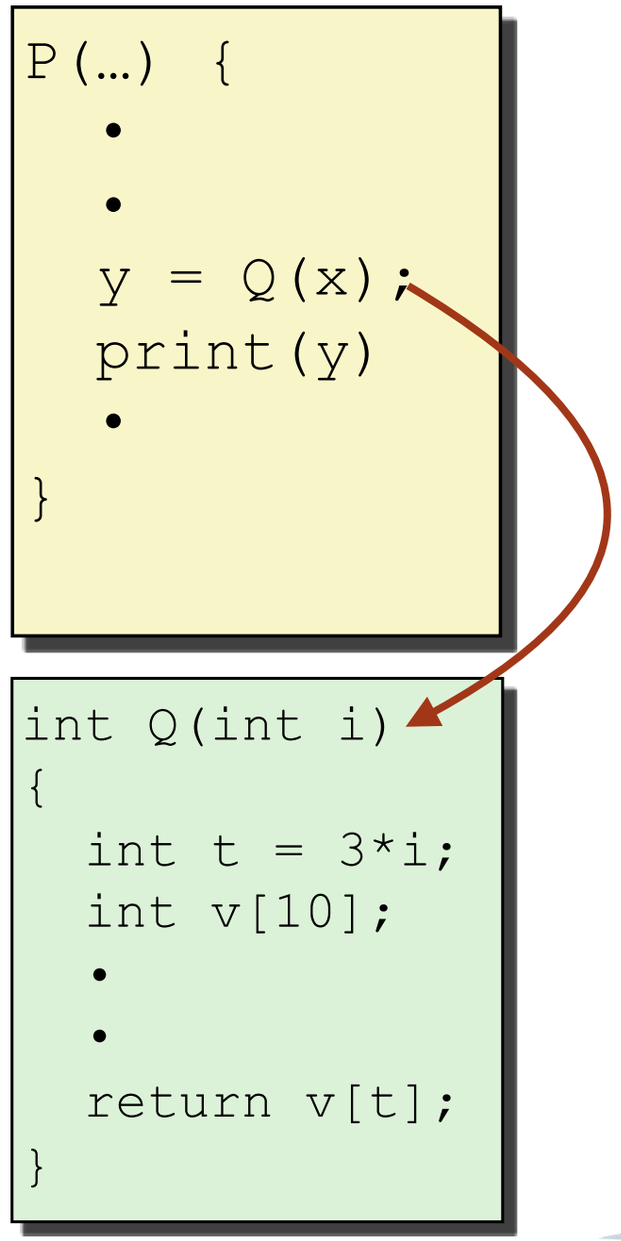
```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

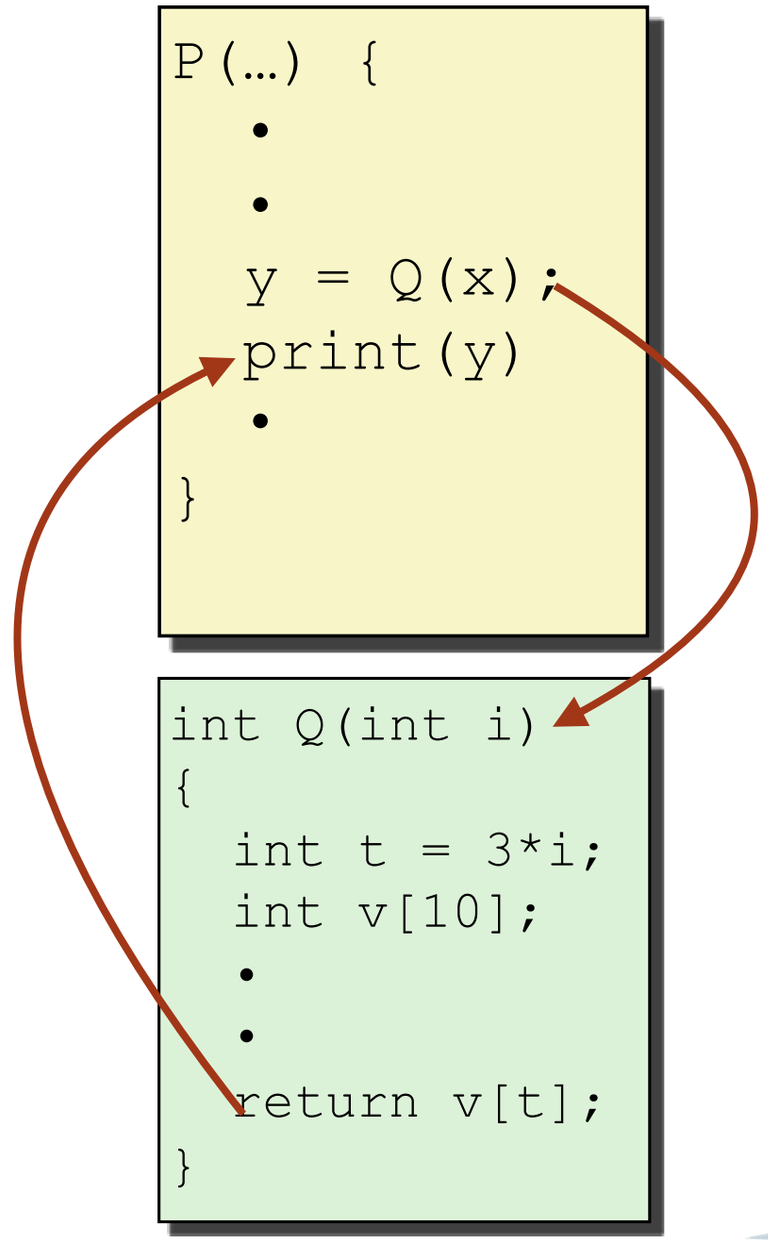
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

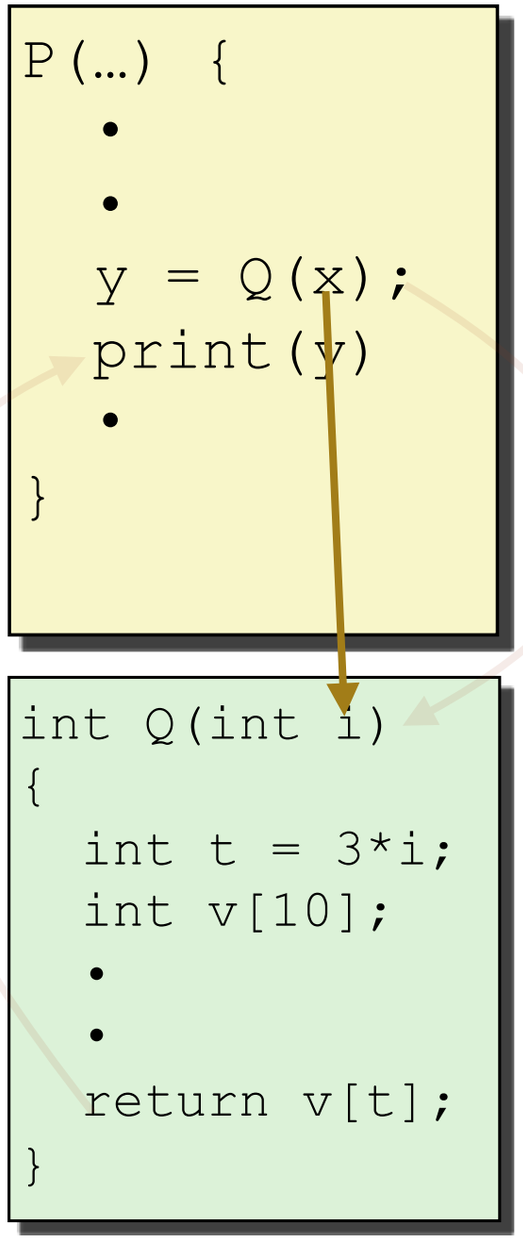
```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

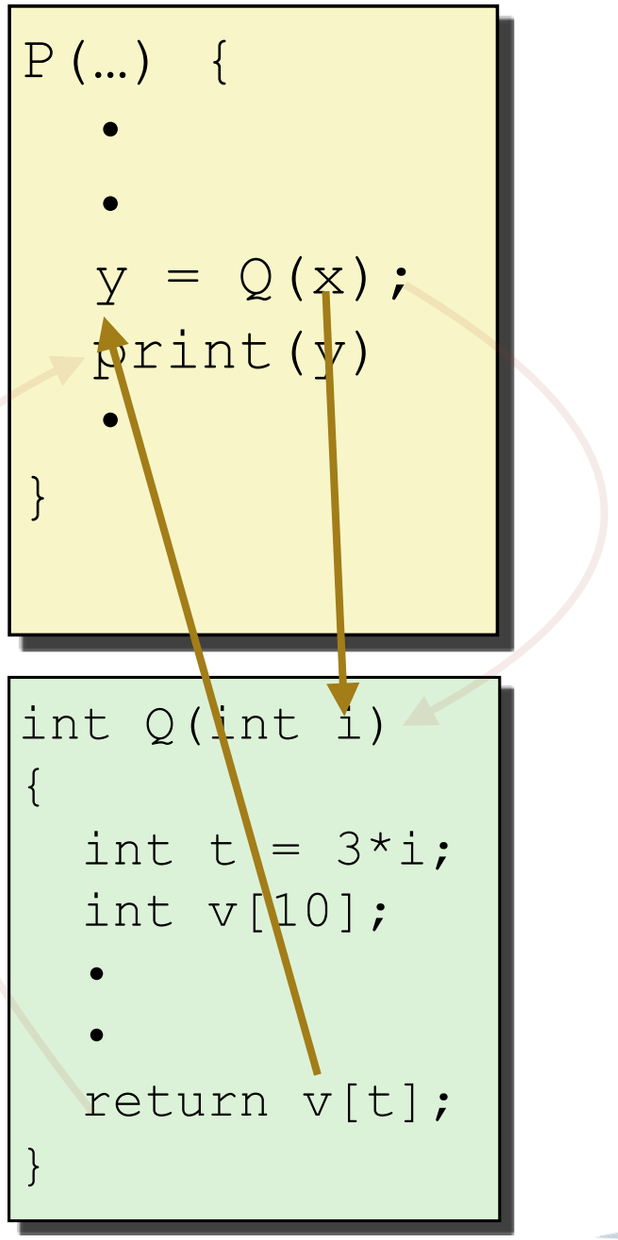
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

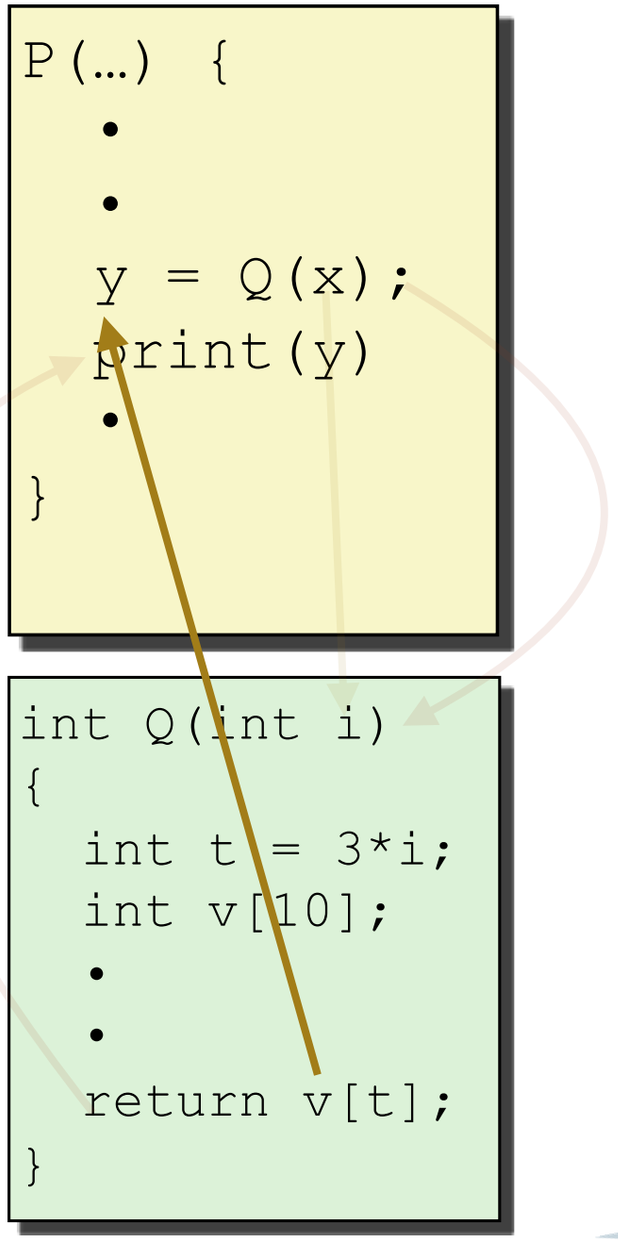
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

PROCEDURES

MECHANISMS IMPLEMENTED IN PROCEDURES

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

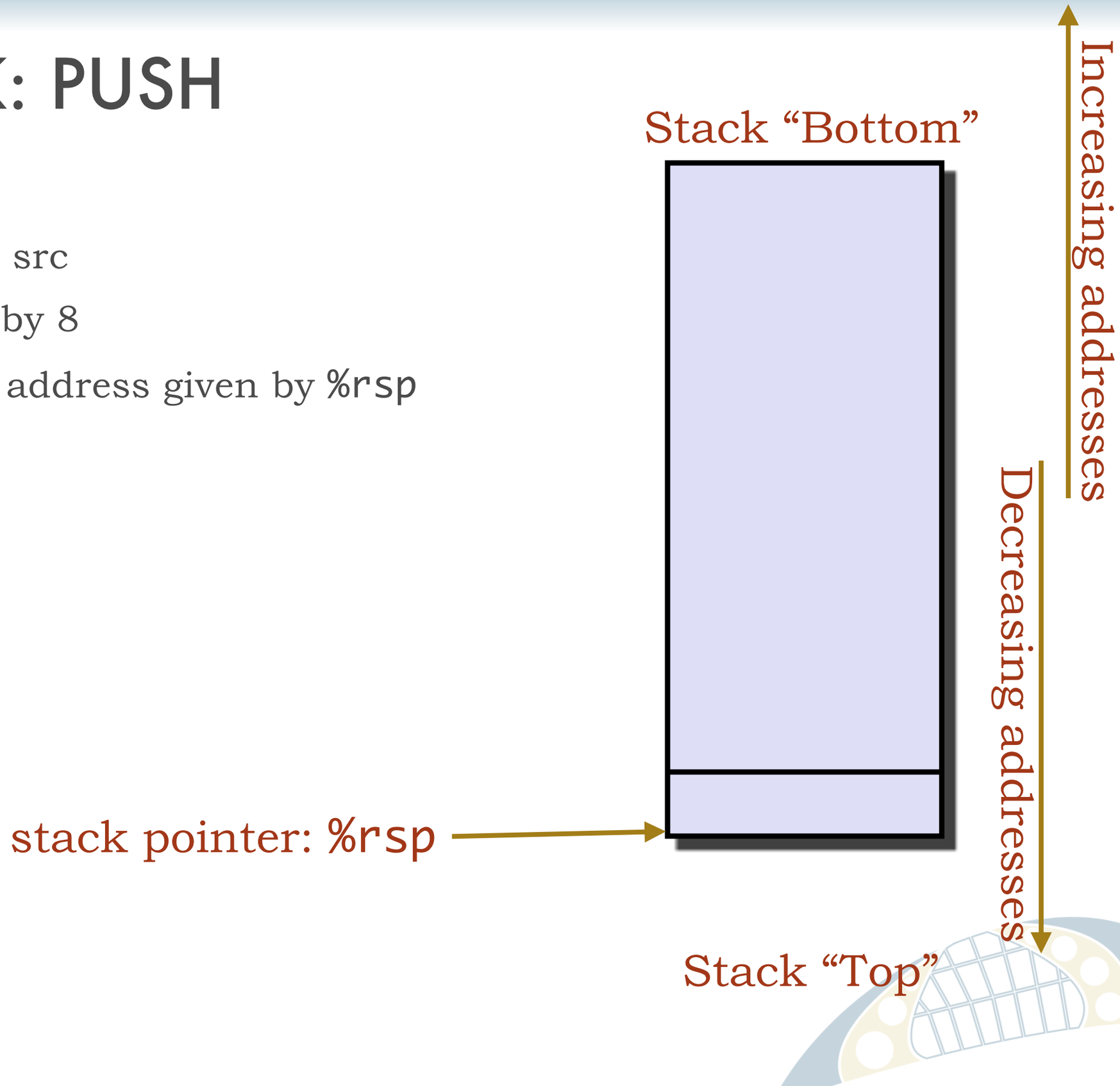
PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

PROCEDURES

X86-64 STACK: PUSH

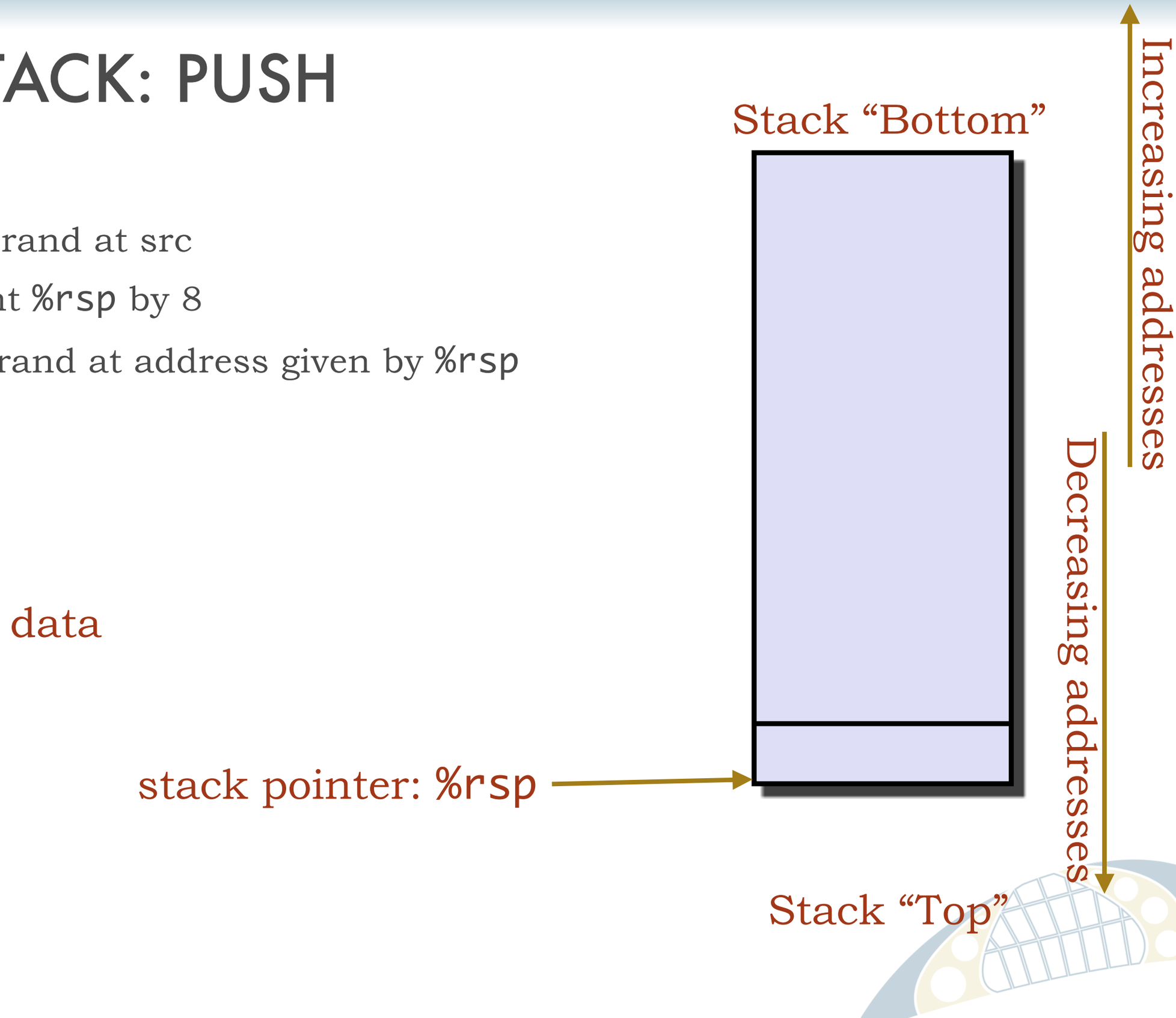
- `pushq src`
  - Fetch operand at src
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



PROCEDURES

X86-64 STACK: PUSH

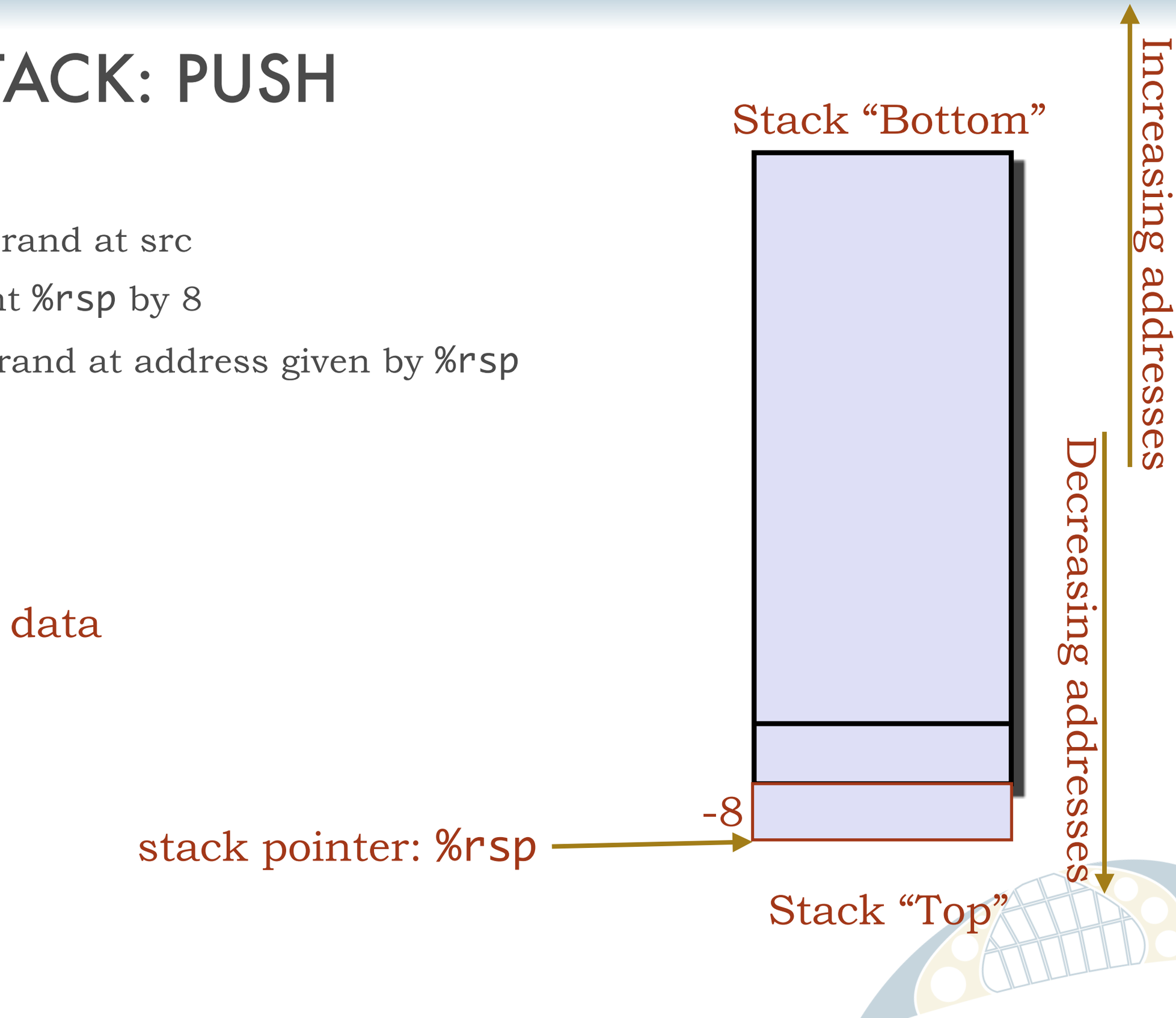
- `pushq src`
  - Fetch operand at src
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



PROCEDURES

X86-64 STACK: PUSH

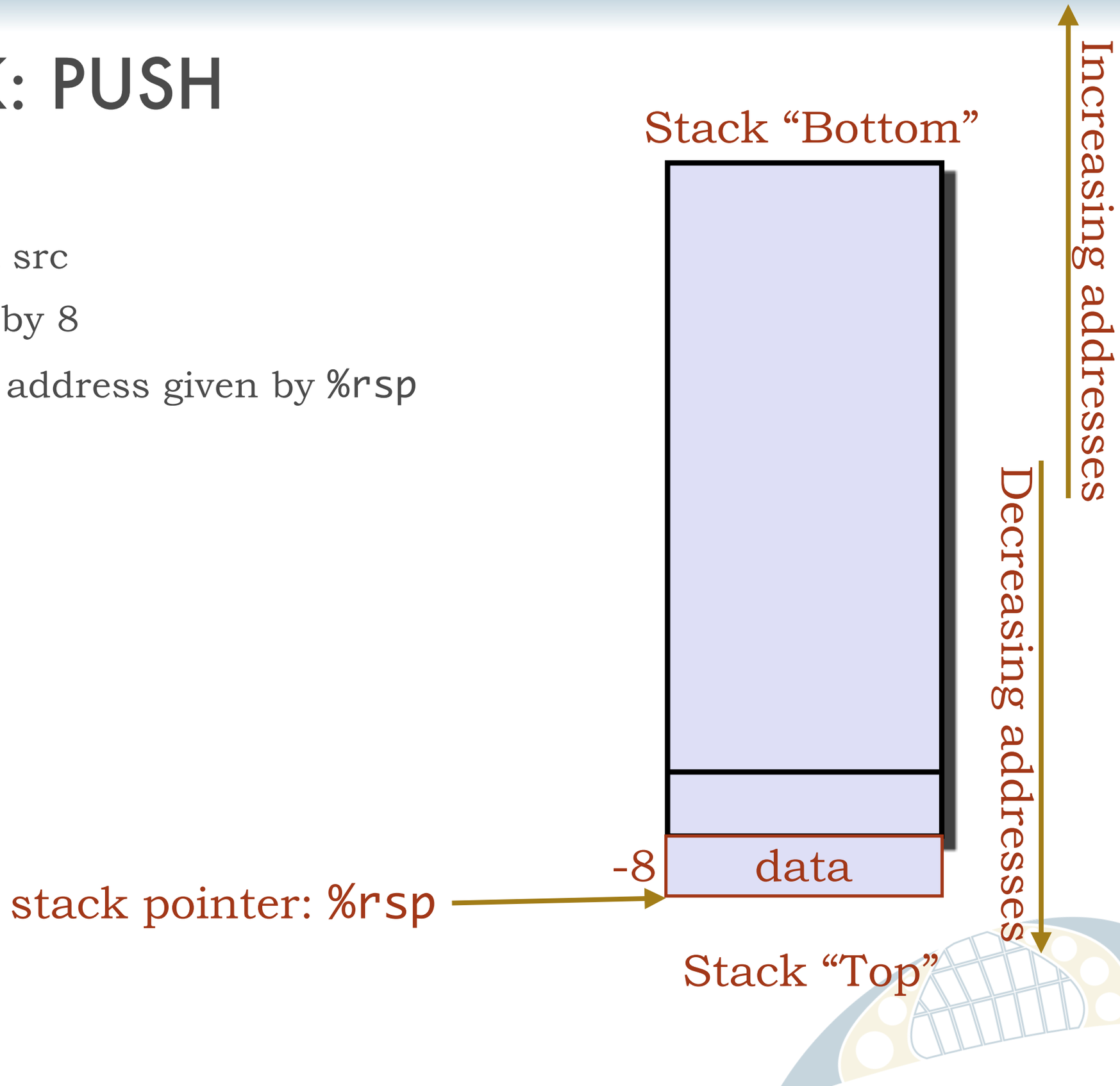
- `pushq src`
  - Fetch operand at src
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



PROCEDURES

X86-64 STACK: PUSH

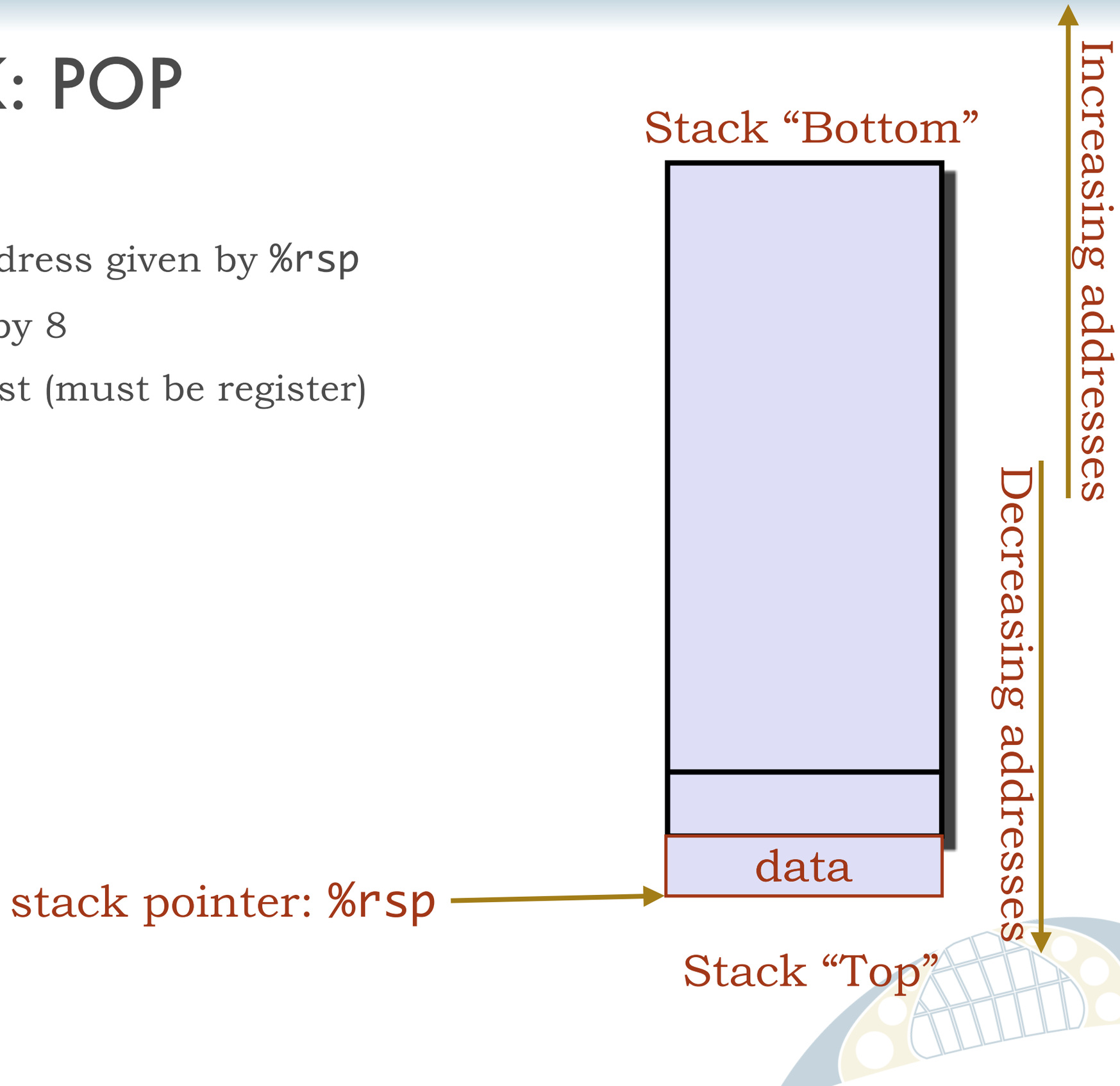
- `pushq src`
  - Fetch operand at src
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



PROCEDURES

X86-64 STACK: POP

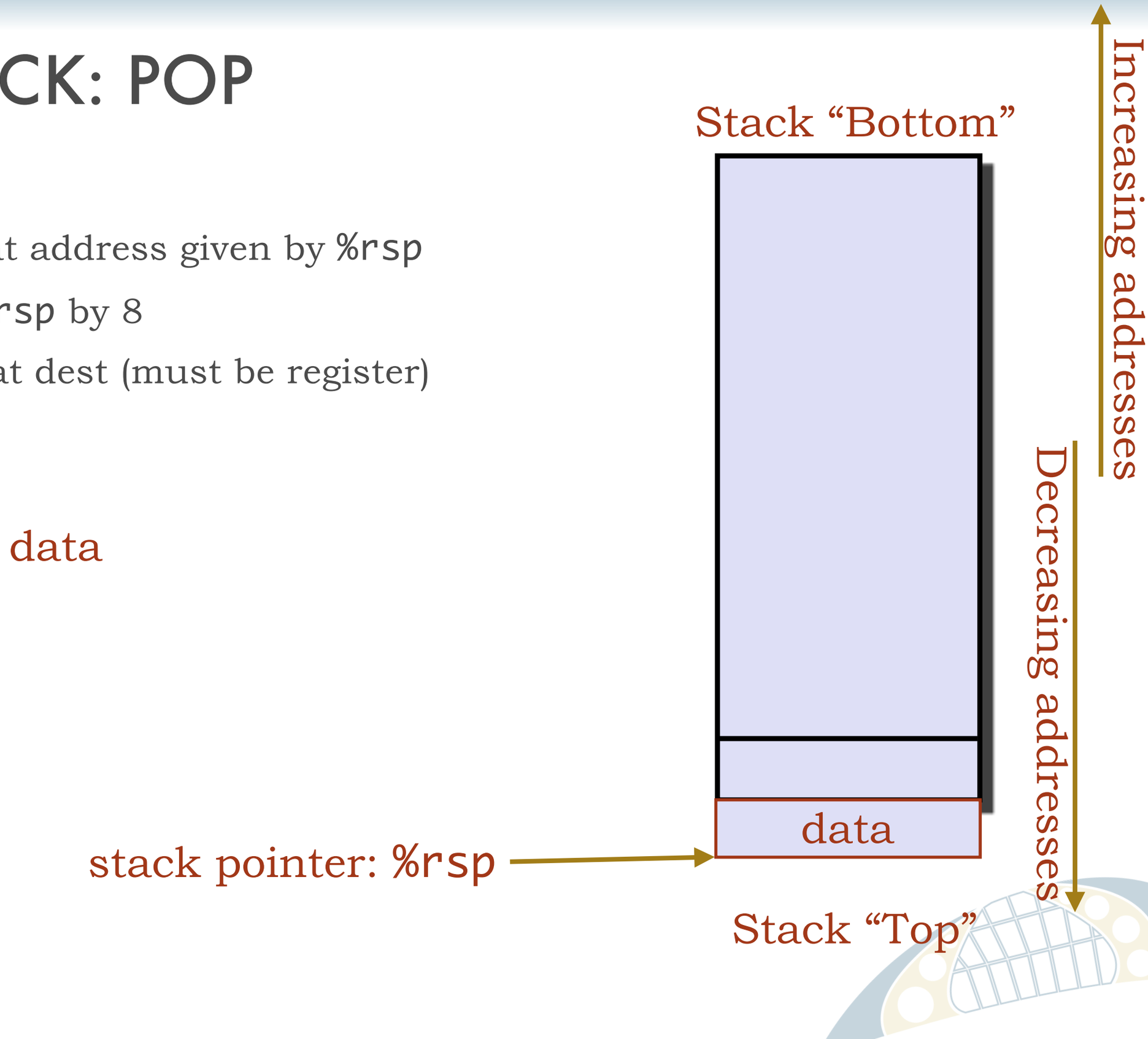
- `popq dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at dest (must be register)



PROCEDURES

X86-64 STACK: POP

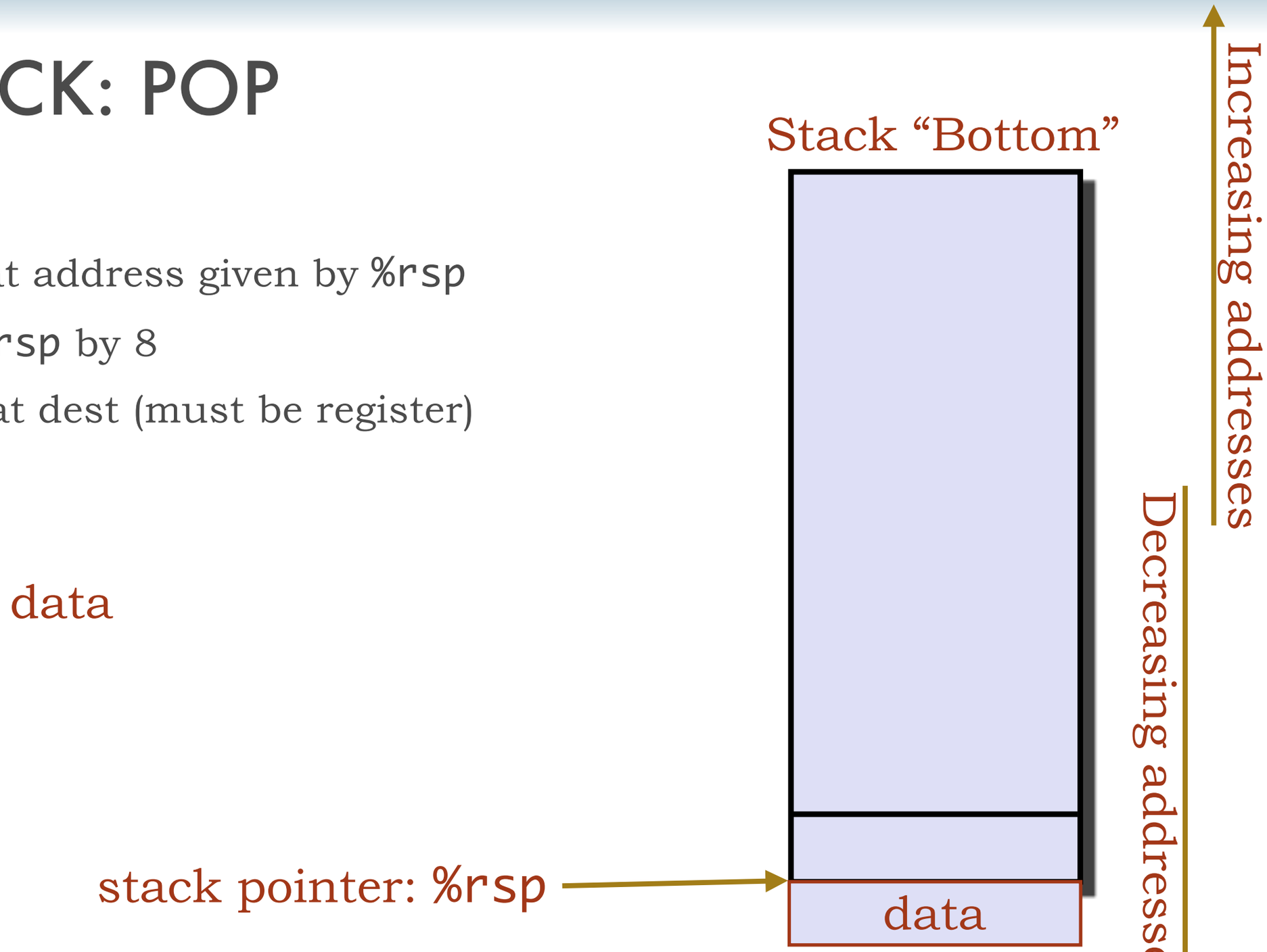
- `popq dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at dest (must be register)



PROCEDURES

X86-64 STACK: POP

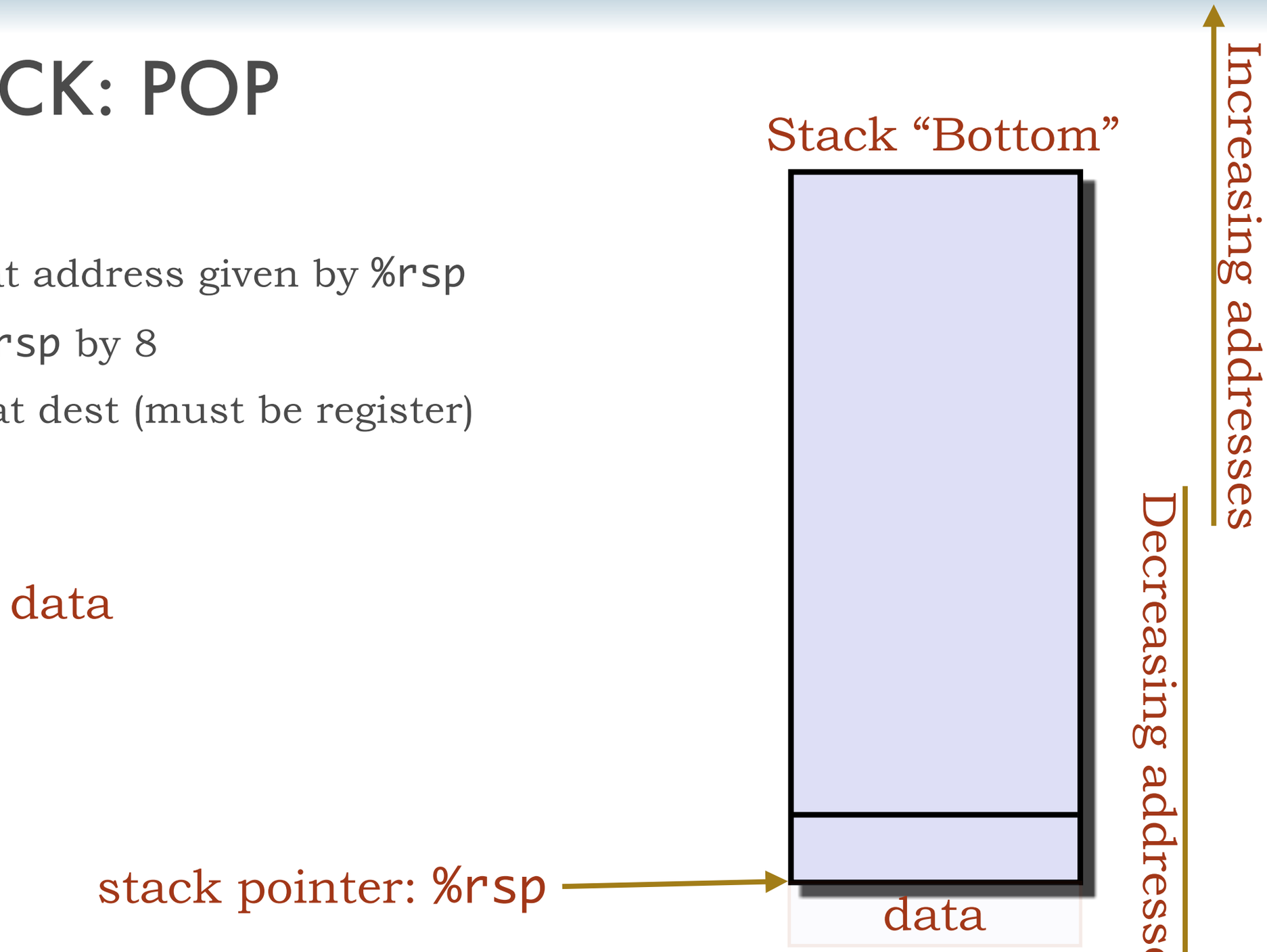
- `popq dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at dest (must be register)



PROCEDURES

X86-64 STACK: POP

- `popq dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at dest (must be register)



PROCEDURES

STACK EXAMPLE

```
pushq %r8
pushq %r9
popq %r9
popq %r8
```

PROCEDURES

STACK EXAMPLE

```
pushq %r8
pushq %r9
popq %r8
popq %r9
```

PROCEDURES

STACK EXAMPLE

```
pushq %r8
pushq %r9
pushq %r10
pushq %r11
popq %r8
popq %r11
popq %r10
popq %r9
```

PROCEDURES

STACK EXAMPLE

```
pushq %r8
pushq %r9
pushq %r10
pushq %r11
callq randomFunction
popq %r11
popq %r10
popq %r9
popq %r8
```

PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

PROCEDURES

CODE EXAMPLES

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx                  # Save %rbx
  400541: mov    %rdx,%rbx             # Save dest
  400544: callq  400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: retq                         # Return
```

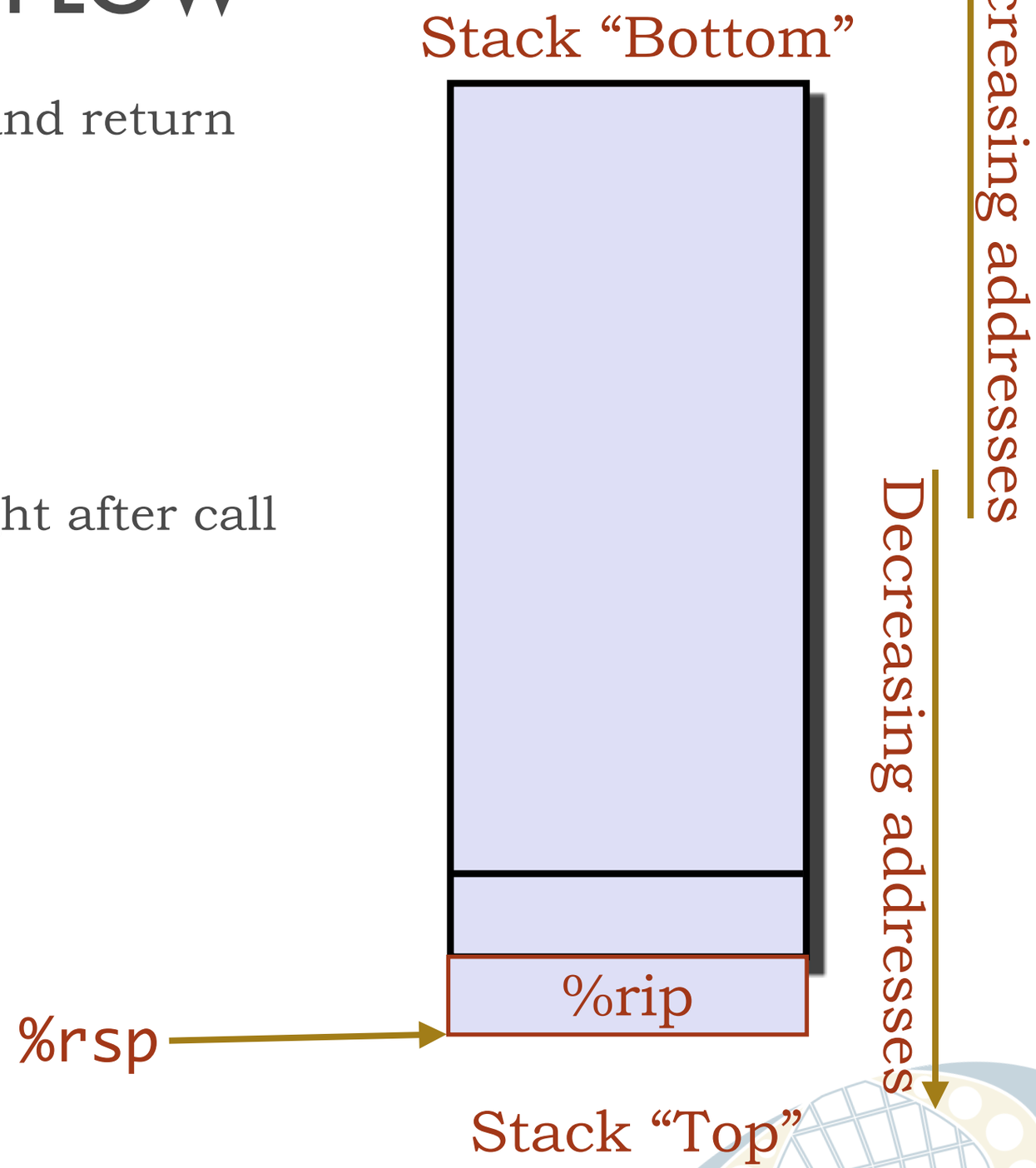
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax            # a
  400553:  imul   %rsi,%rax            # a * b
  400557:  retq                        # Return
```

PROCEDURES

PROCEDURE CONTROL FLOW

- Use stack to support procedure call and return
- Procedure call: `call label`
  - Push return address on stack
  - Jump to label
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address

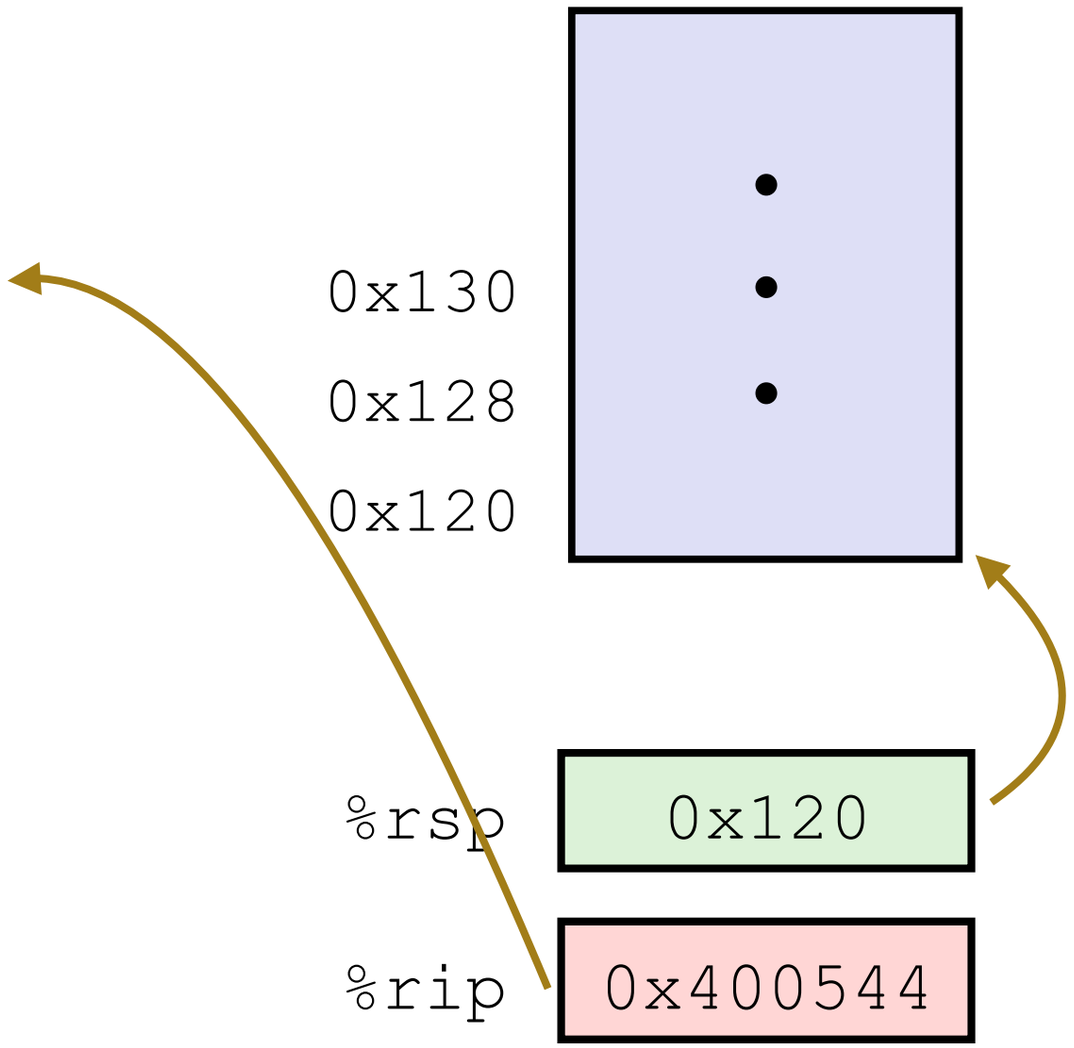


PROCEDURES

CONTROL FLOW EXAMPLE STEP #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

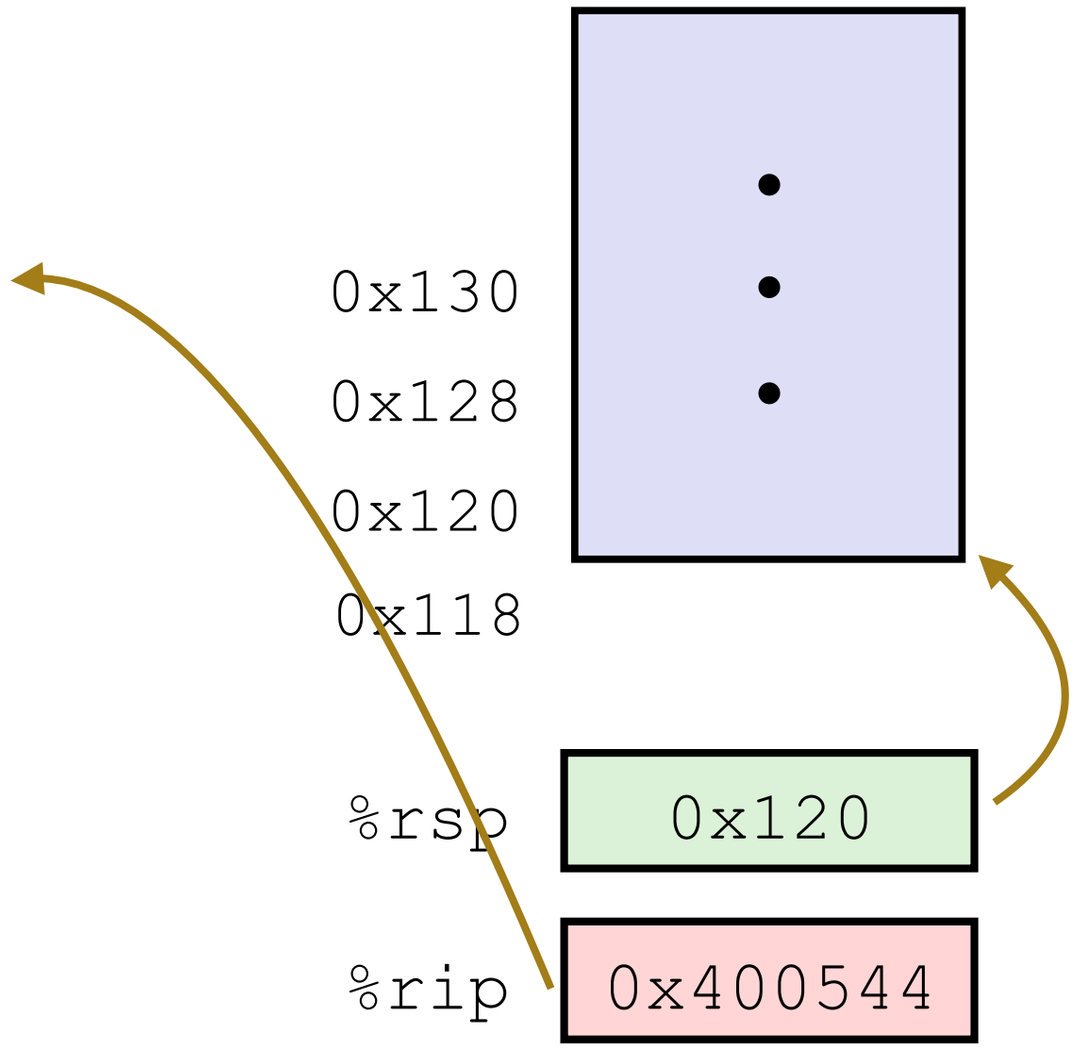


PROCEDURES

CONTROL FLOW EXAMPLE STEP #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

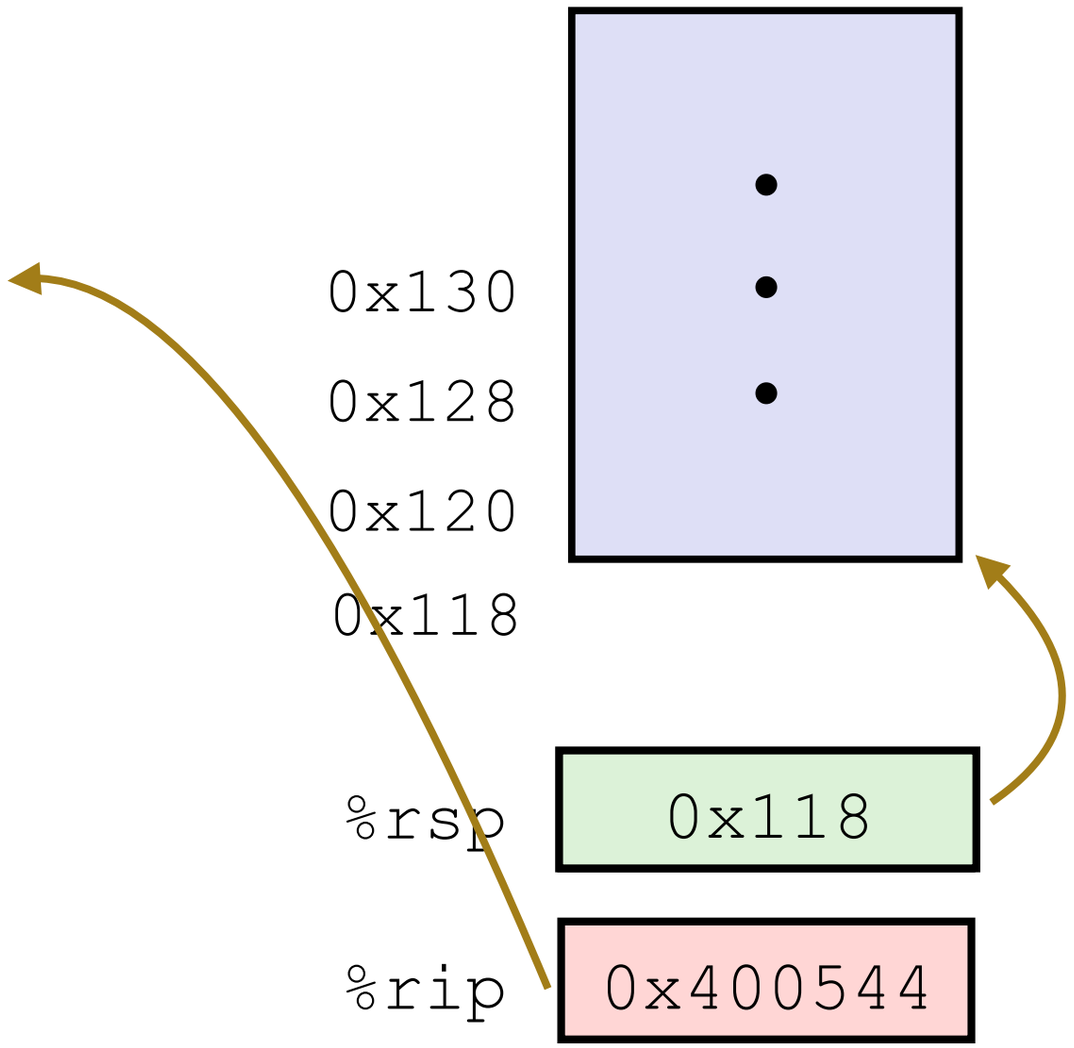


PROCEDURES

CONTROL FLOW EXAMPLE STEP #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

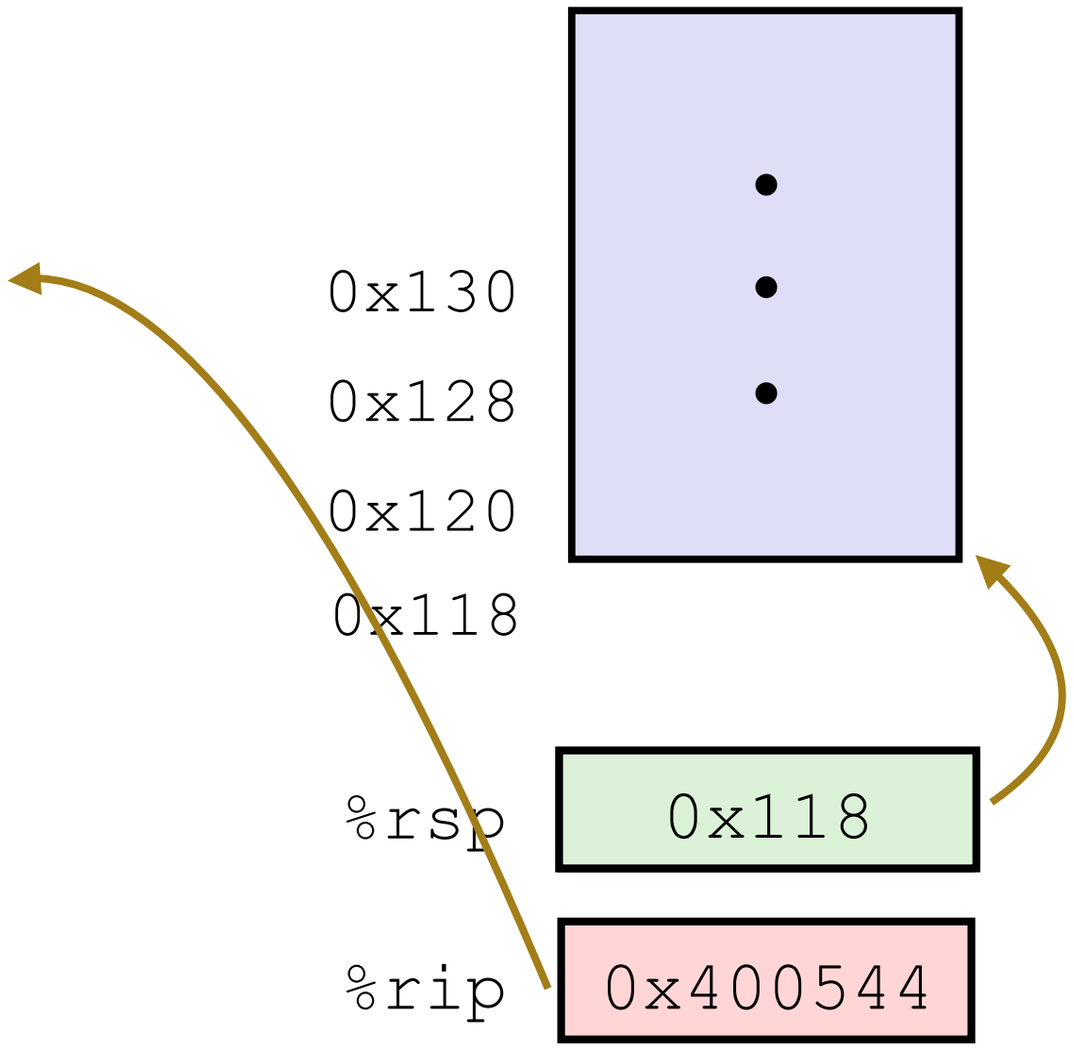


PROCEDURES

CONTROL FLOW EXAMPLE STEP #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

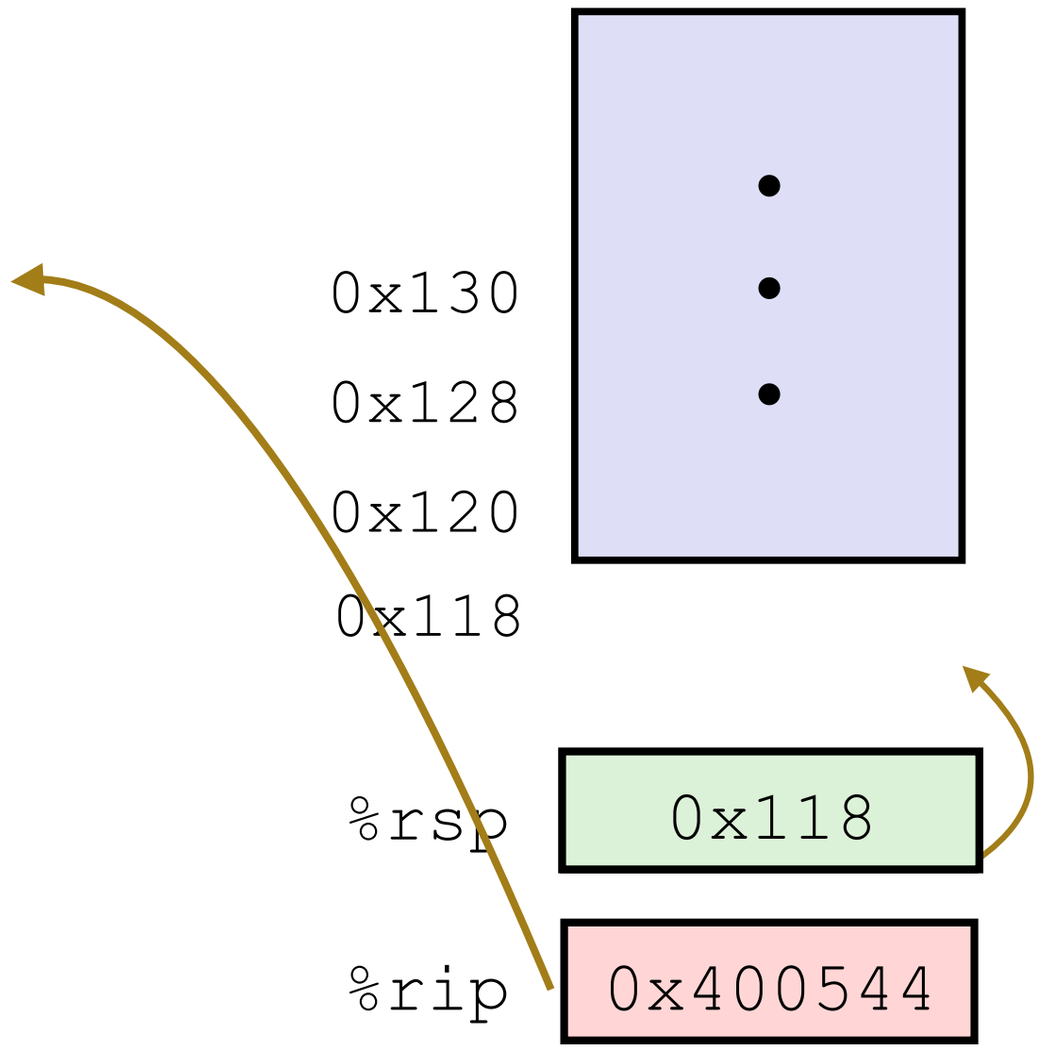


PROCEDURES

CONTROL FLOW EXAMPLE STEP #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

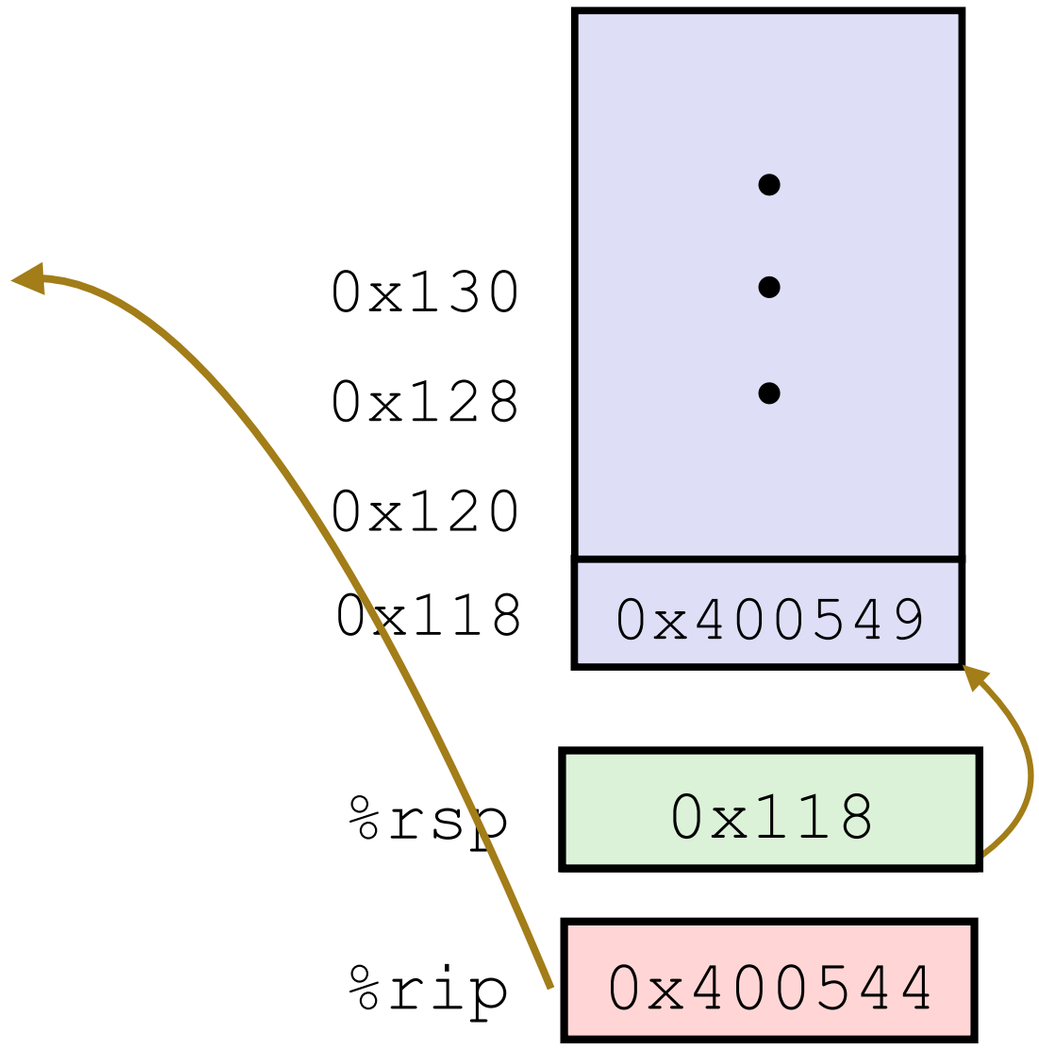
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



CONTROL FLOW EXAMPLE STEP #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

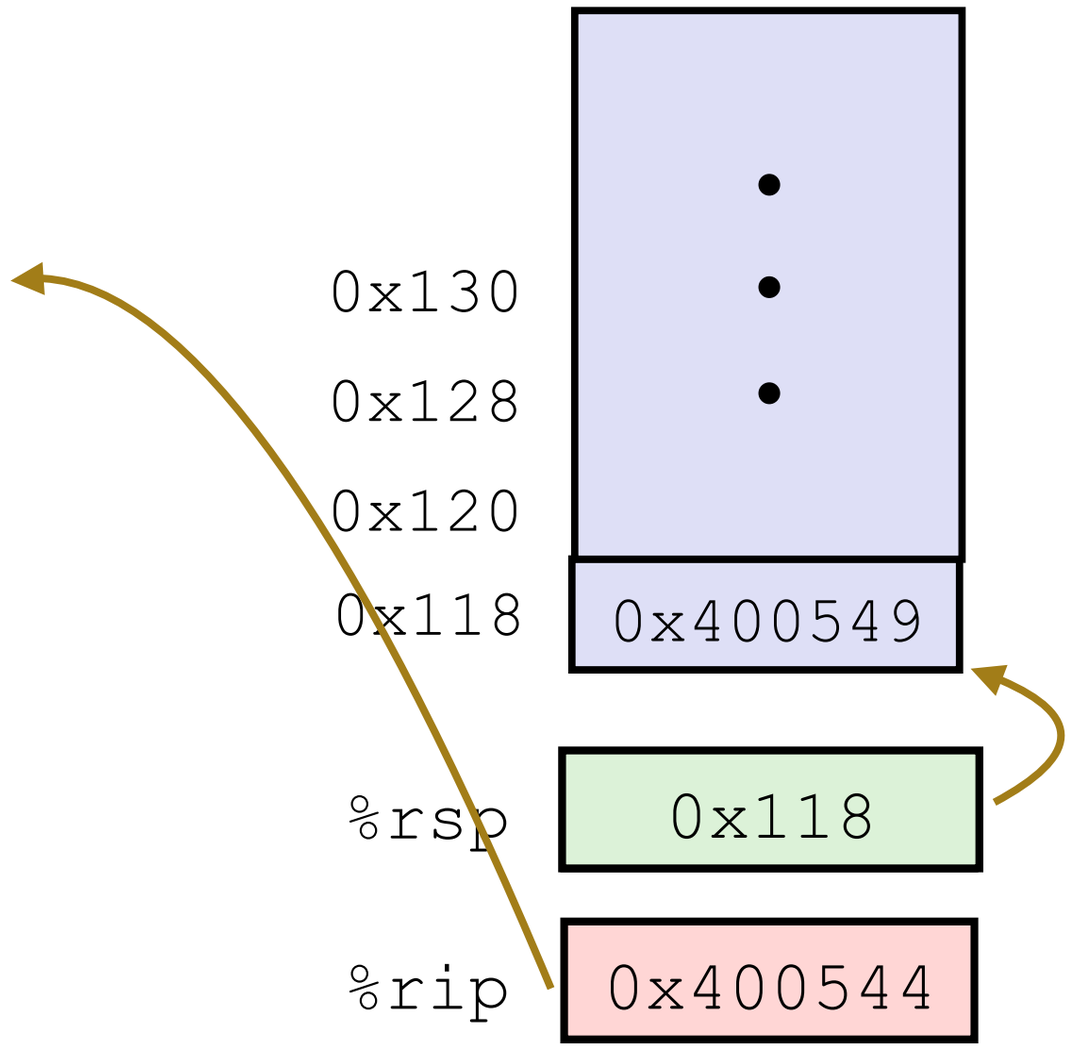


PROCEDURES

CONTROL FLOW EXAMPLE STEP #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

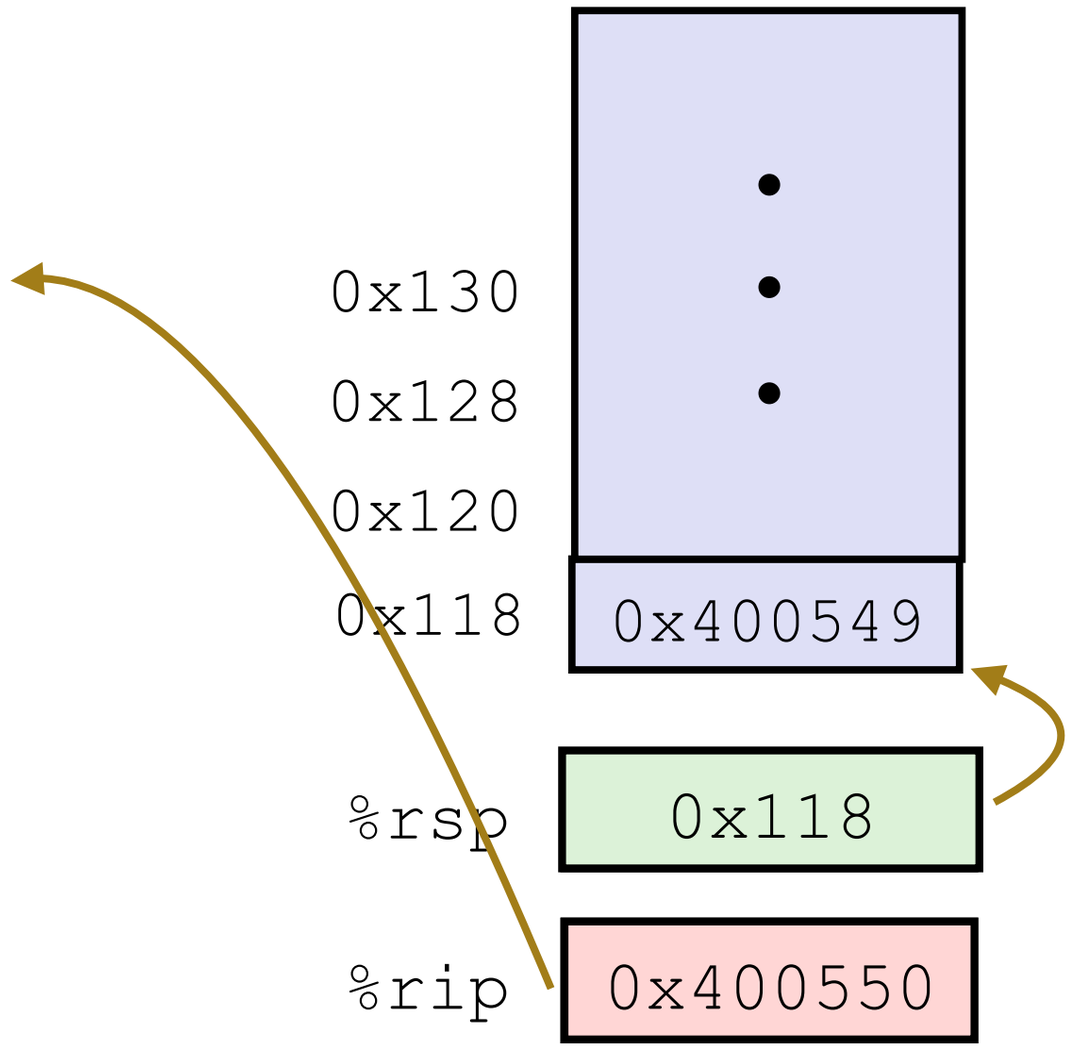


PROCEDURES

CONTROL FLOW EXAMPLE STEP #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

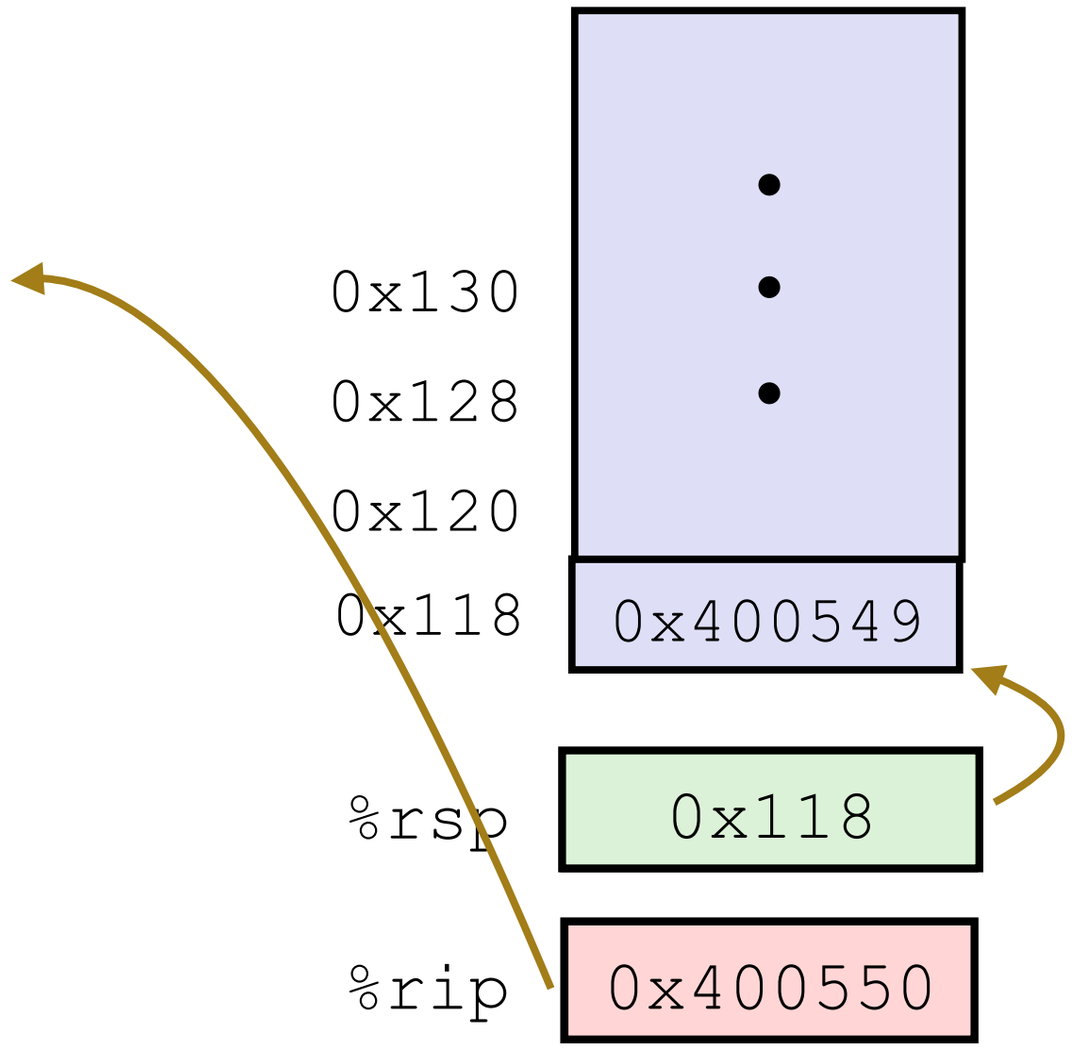


PROCEDURES

CONTROL FLOW EXAMPLE STEP #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

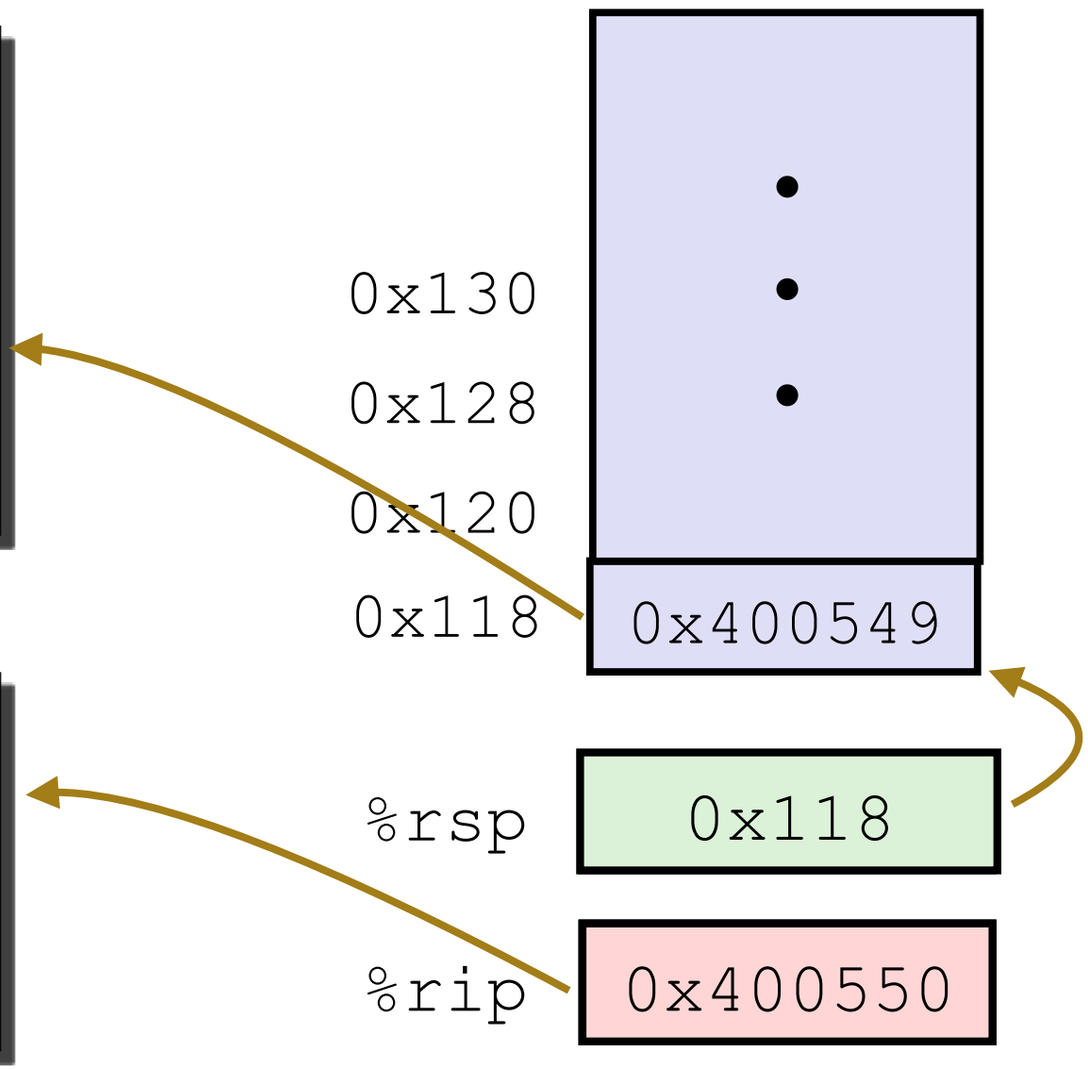


PROCEDURES

CONTROL FLOW EXAMPLE STEP #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

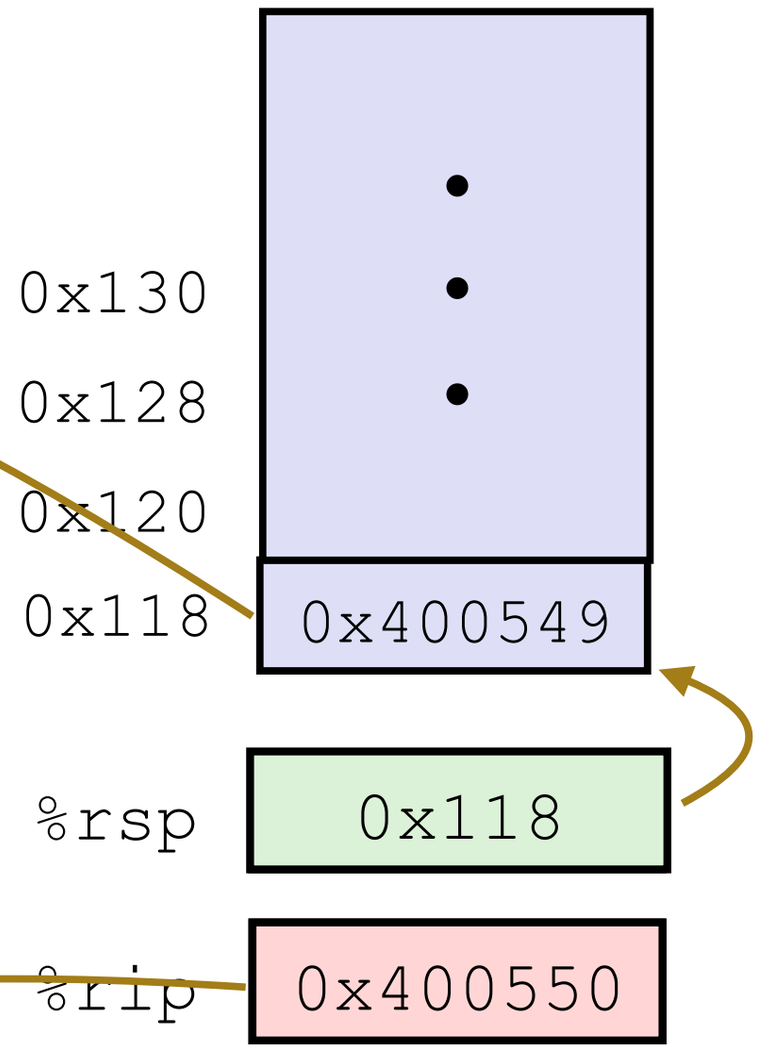


PROCEDURES

CONTROL FLOW EXAMPLE STEP #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

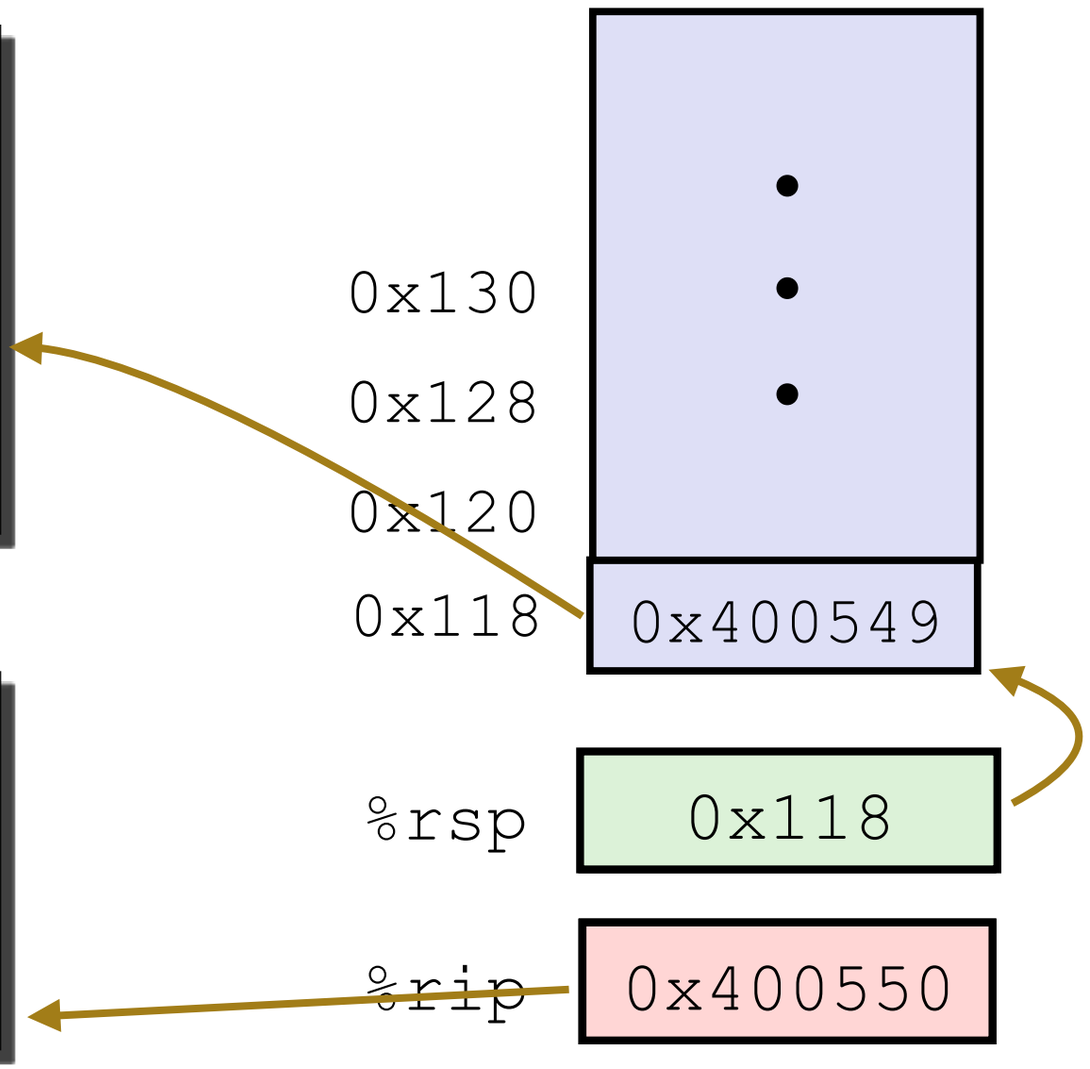


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

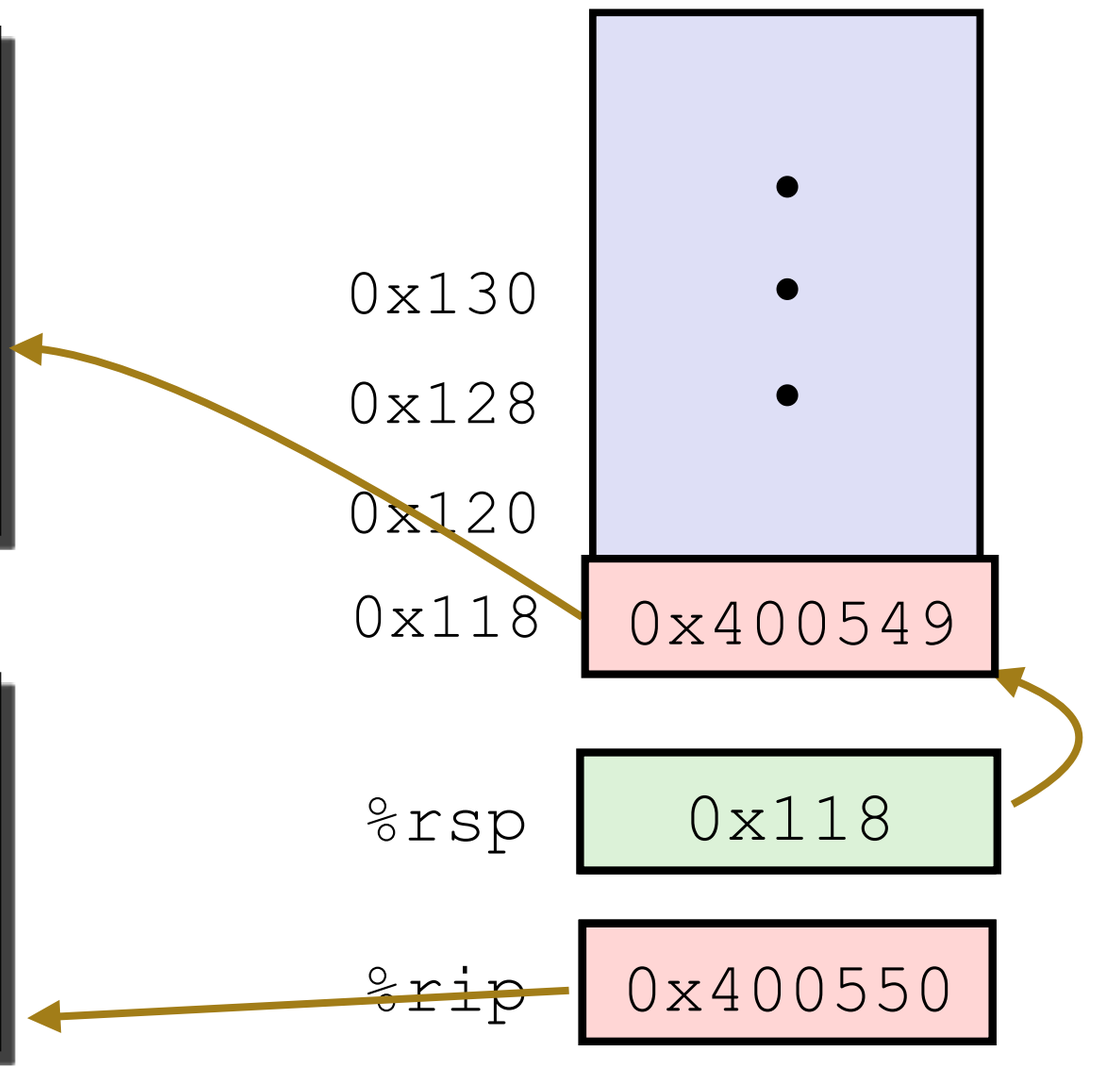


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

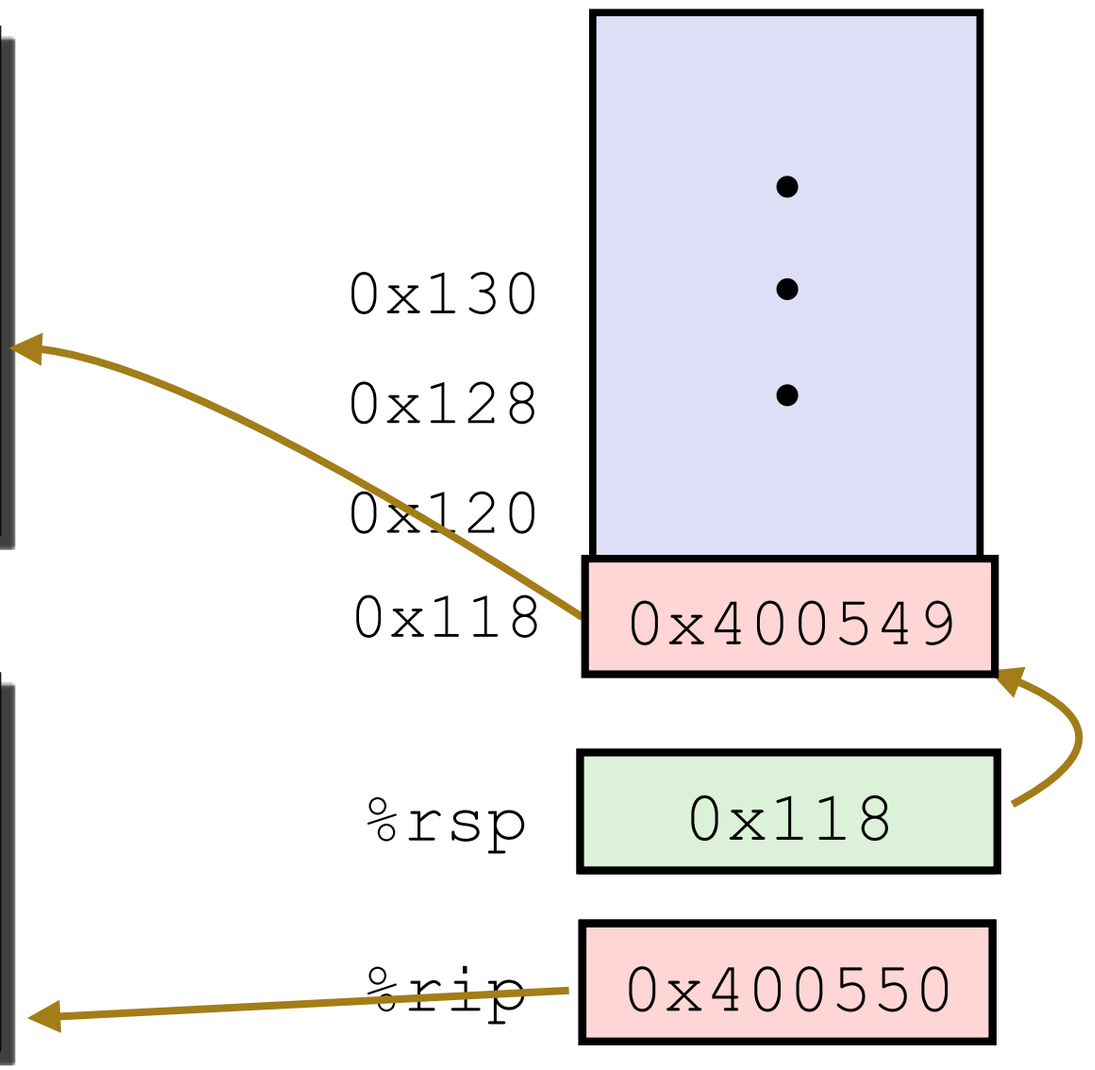


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

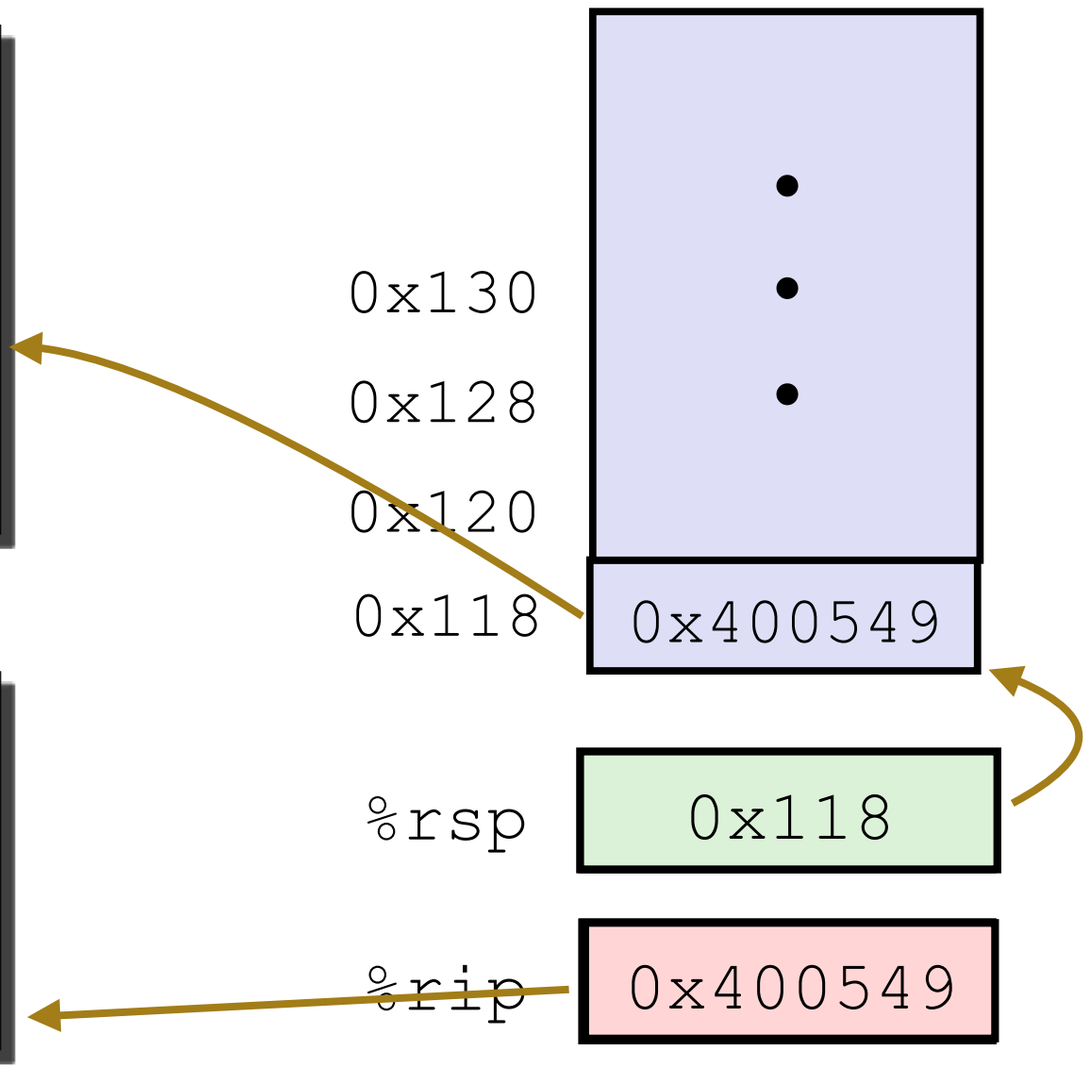


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

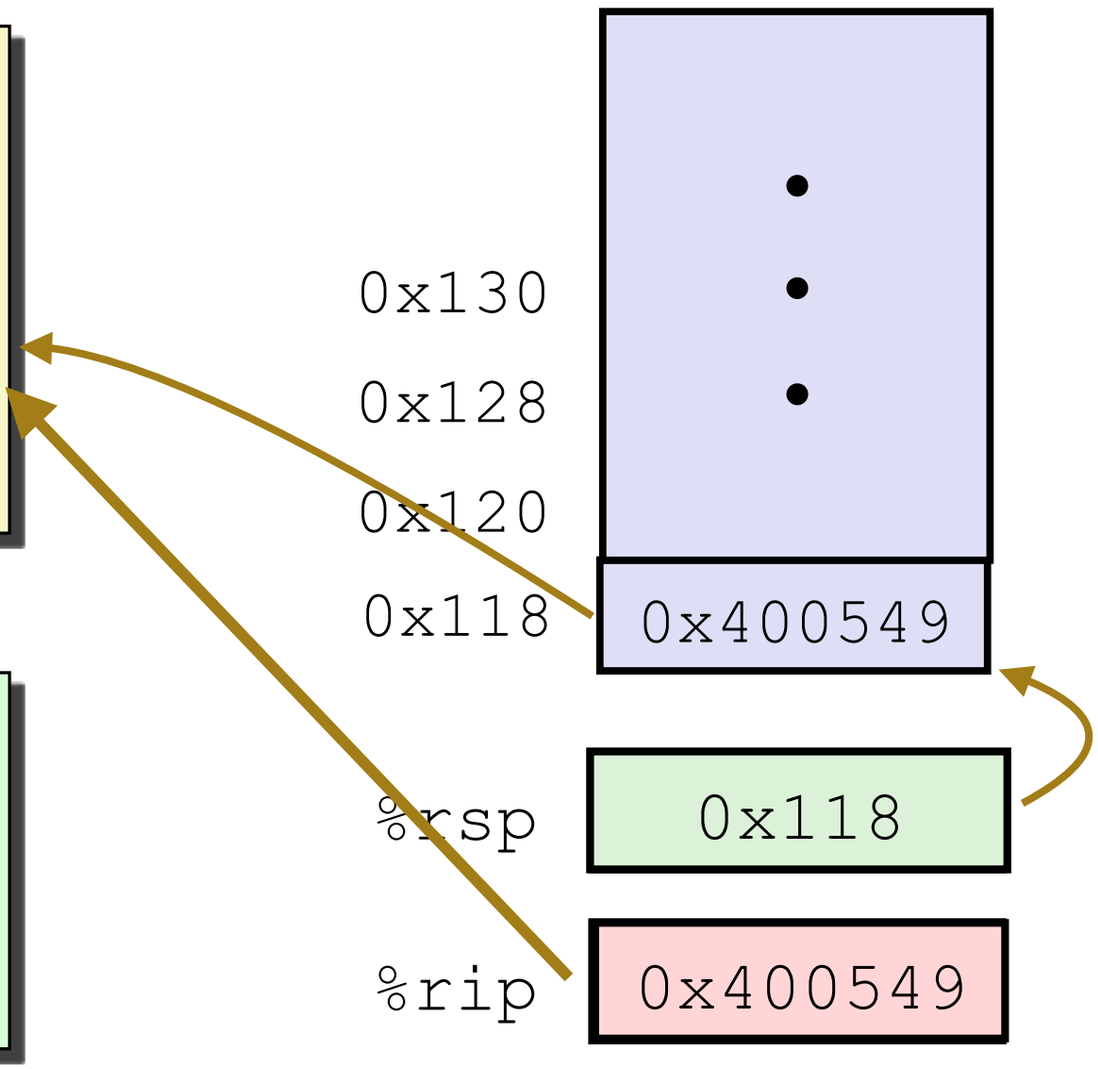


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

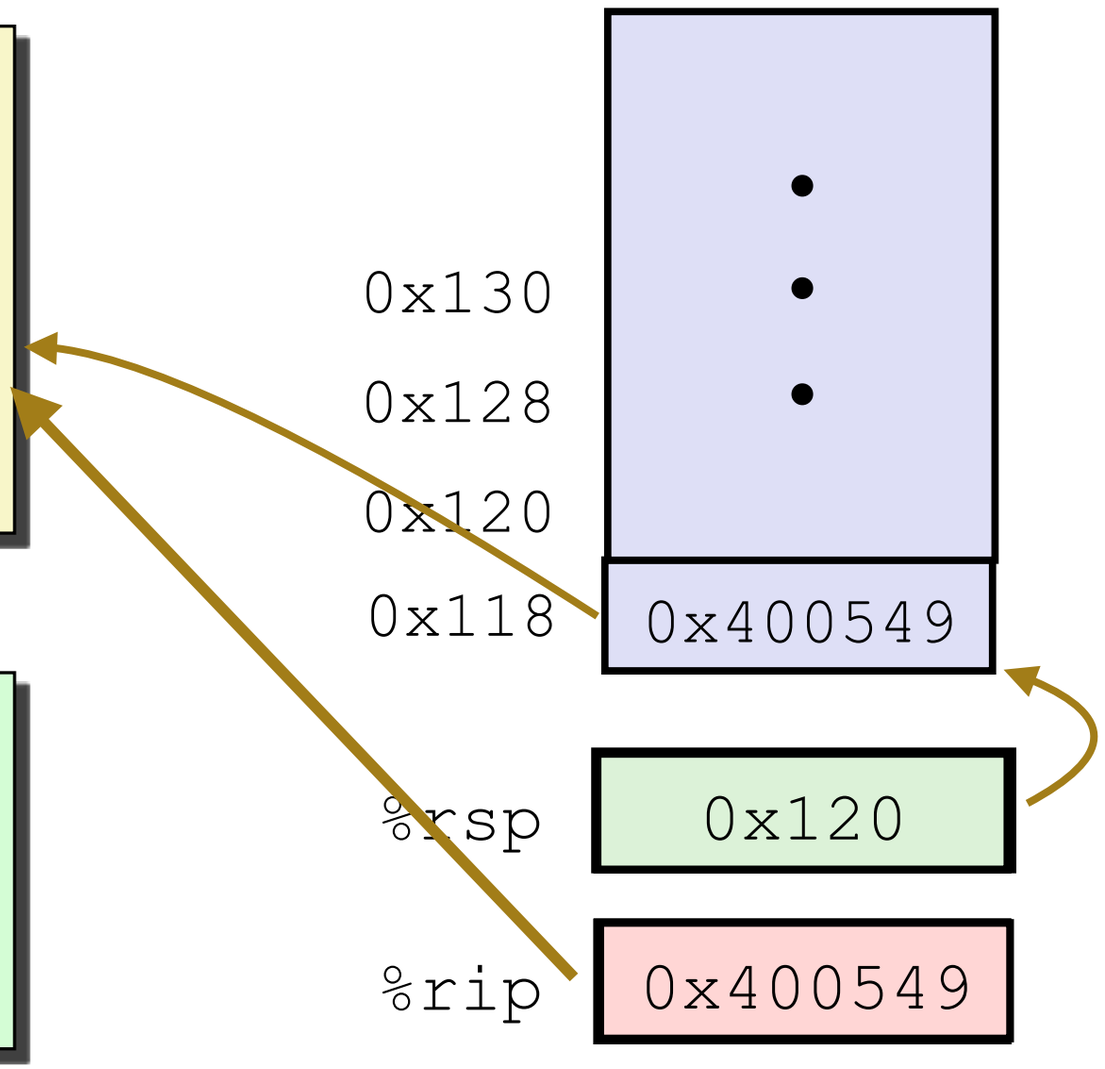


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

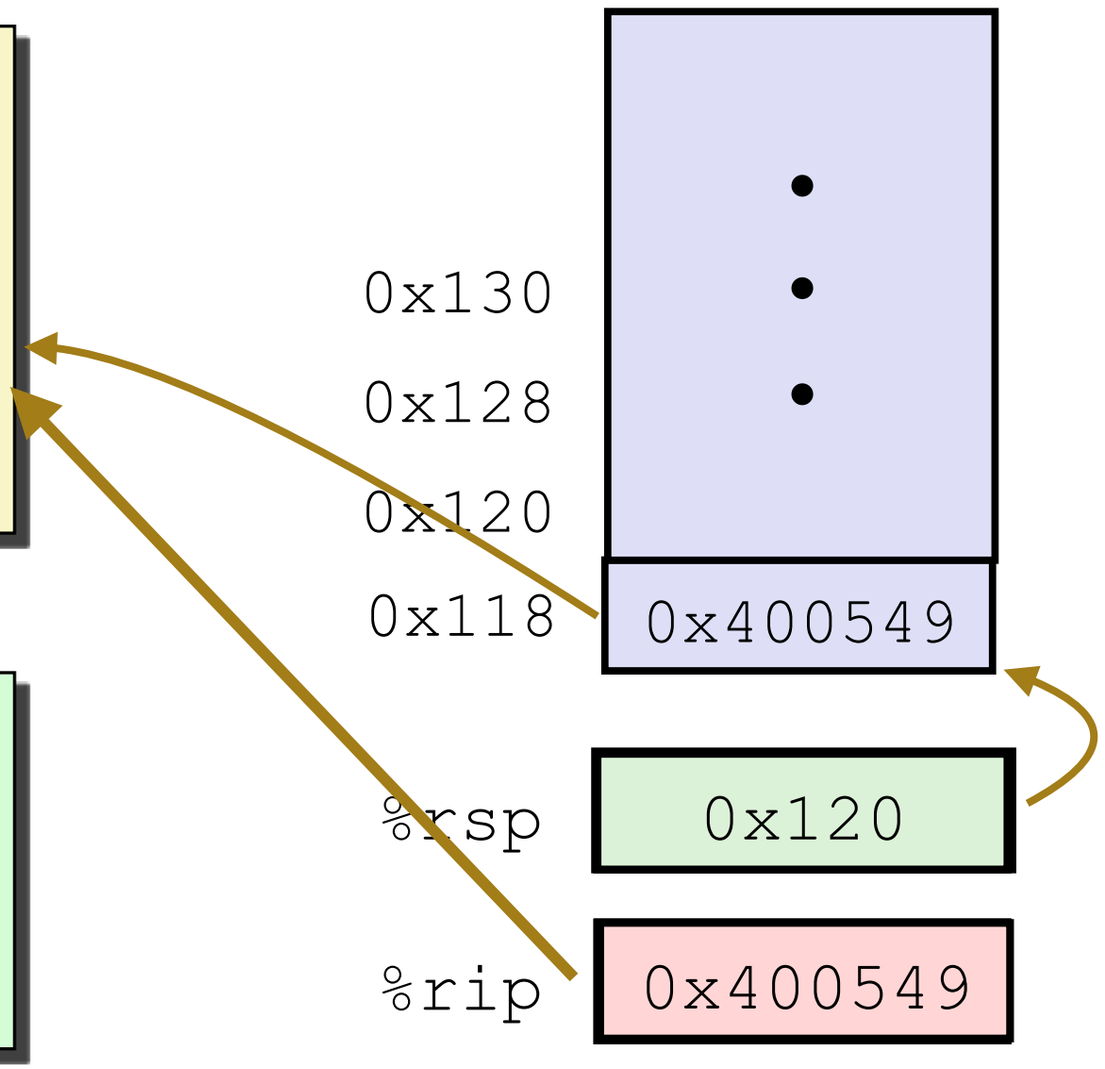


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

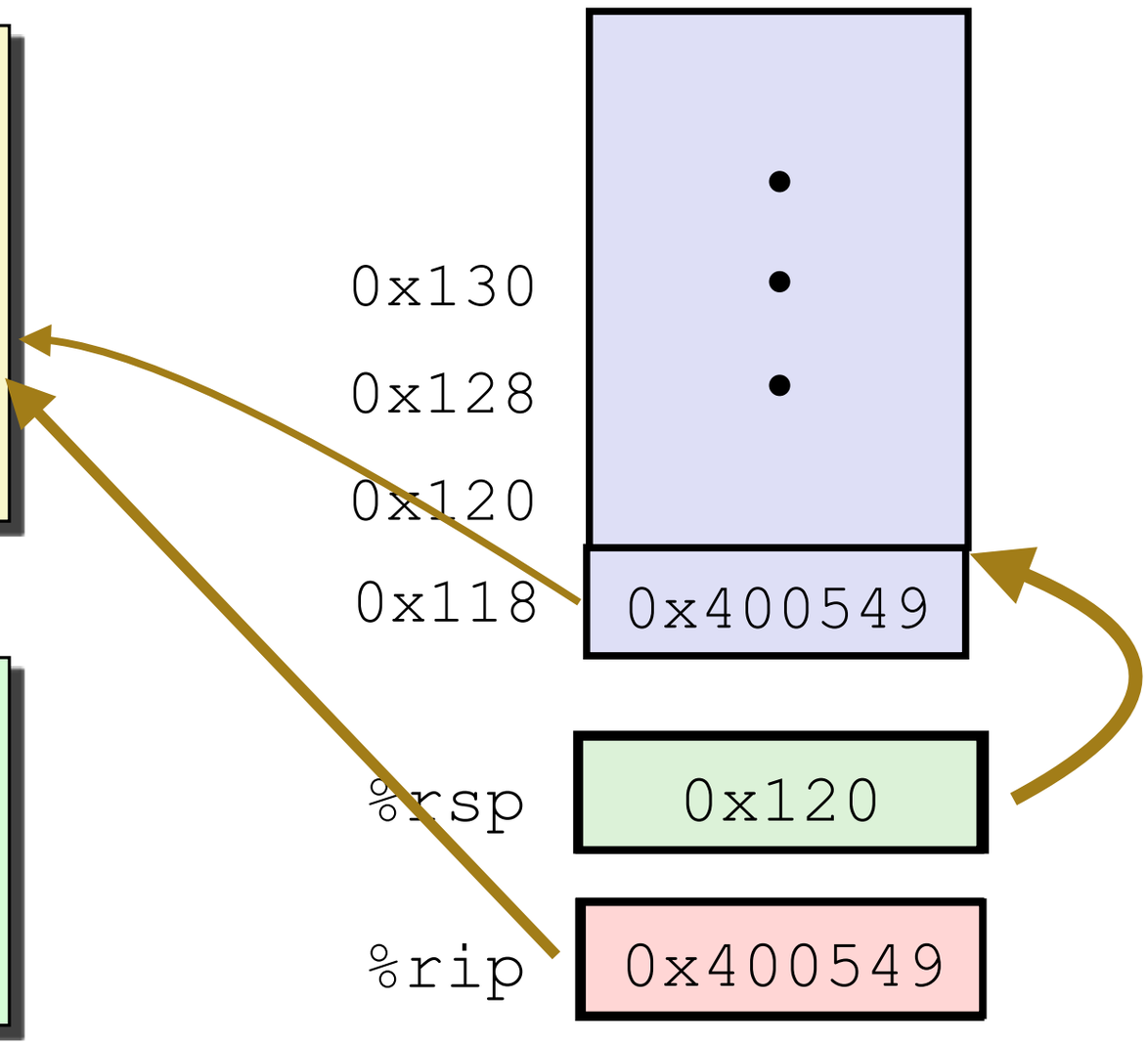


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

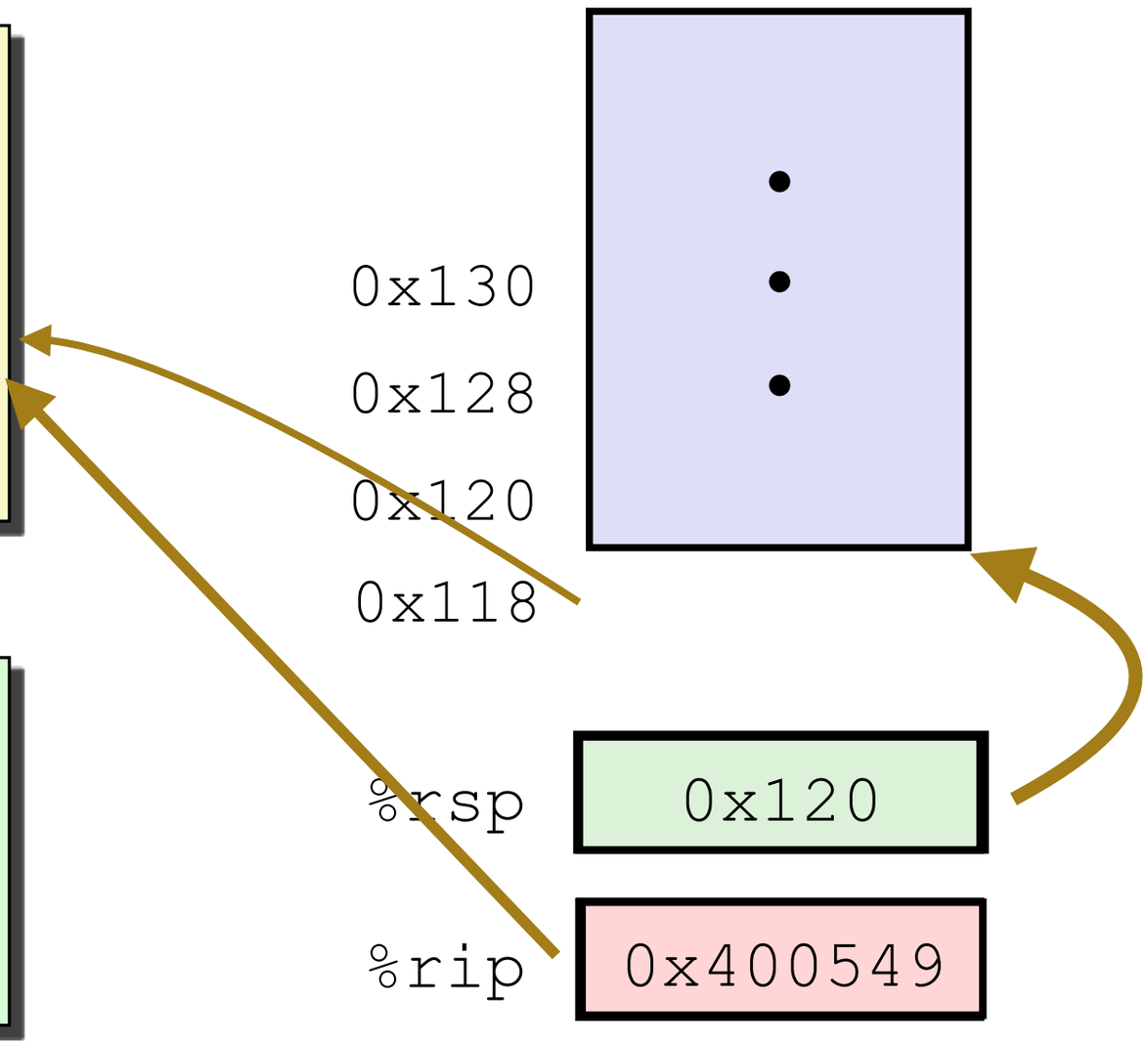


PROCEDURES

CONTROL FLOW EXAMPLE STEP #5

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



PROCEDURE CONTROL FLOW SUMMARY

- A procedure call is just a jump with some support for returning
  - when a `call label` is executed, the chip
    - pushes the next instruction location on the stack
      - decrease the stack pointer, %rsp, by 8
      - move the next instruction location into (%rsp)
    - jumps to `label`
      - moves the location of label into %rip

- when a `ret` is executed, the chip
  - pops the top of the stack into %rip
    - move (%rsp) into %rip
    - subtract 8 from %rsp

PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

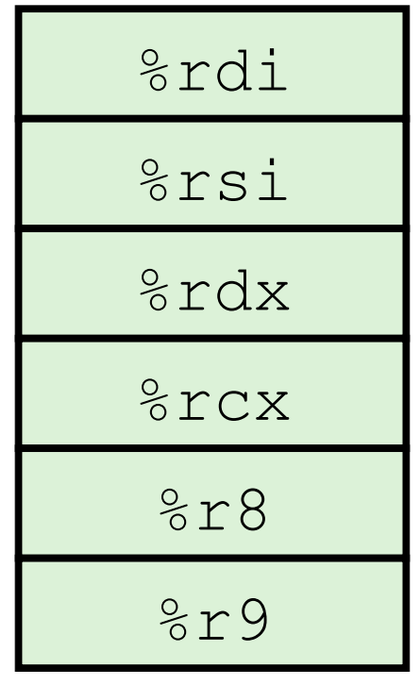
PROCEDURES

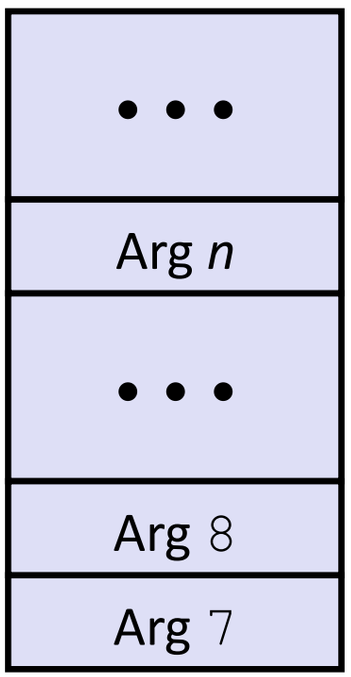
- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

PROCEDURES

PROCEDURE DATA FLOW

- Where do we put the parameters for a function?
  - The first six parameters are placed in registers

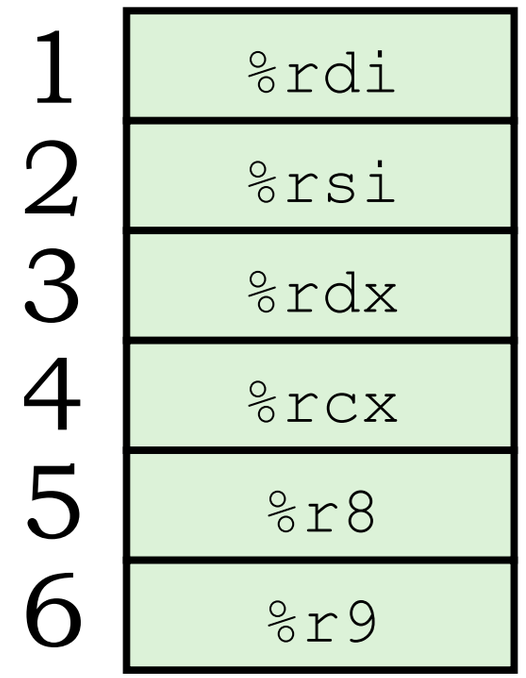


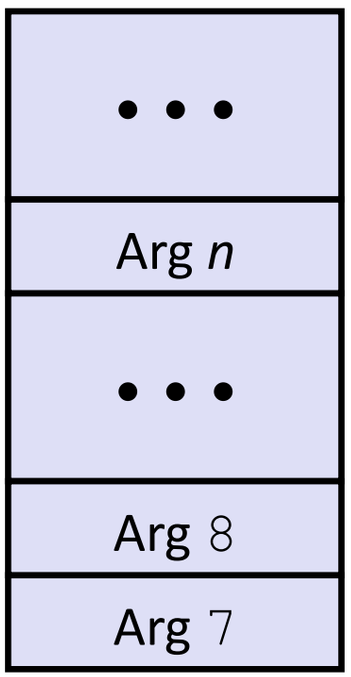


PROCEDURES

PROCEDURE DATA FLOW

- Where do we put the parameters for a function?
  - The first six parameters are placed in registers



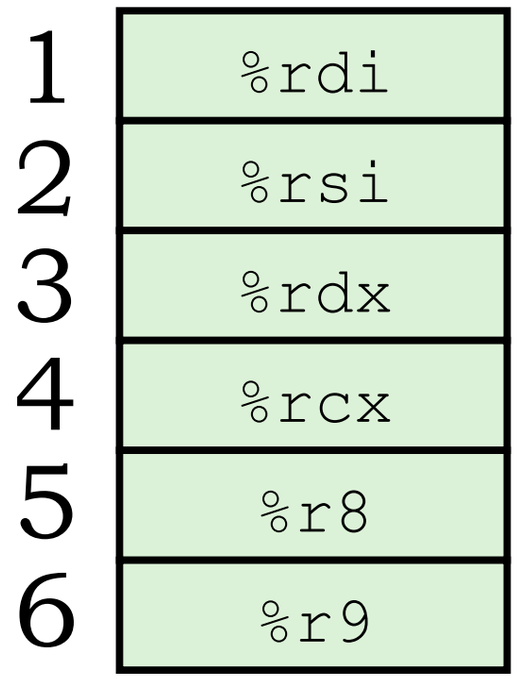


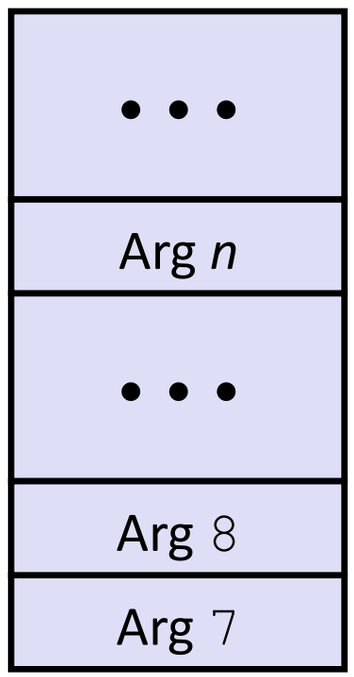
PROCEDURES

PROCEDURE DATA FLOW

- Where do we put the parameters for a function?
  - The first six parameters are placed in registers

- Where do we put the return value from a function?
  - into %rax



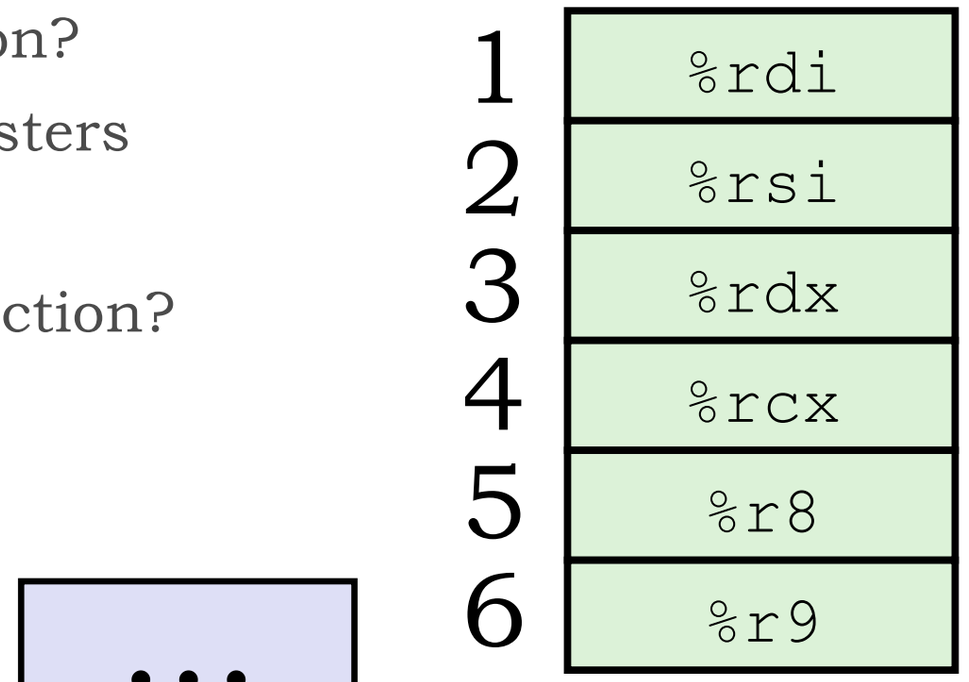


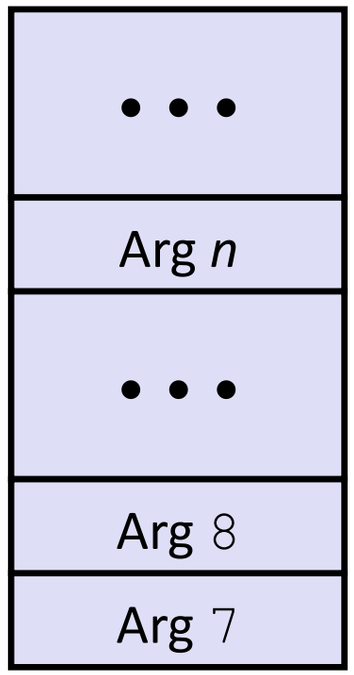
PROCEDURES

PROCEDURE DATA FLOW

- Where do we put the parameters for a function?
  - The first six parameters are placed in registers

- Where do we put the return value from a function?
  - into `%rax`





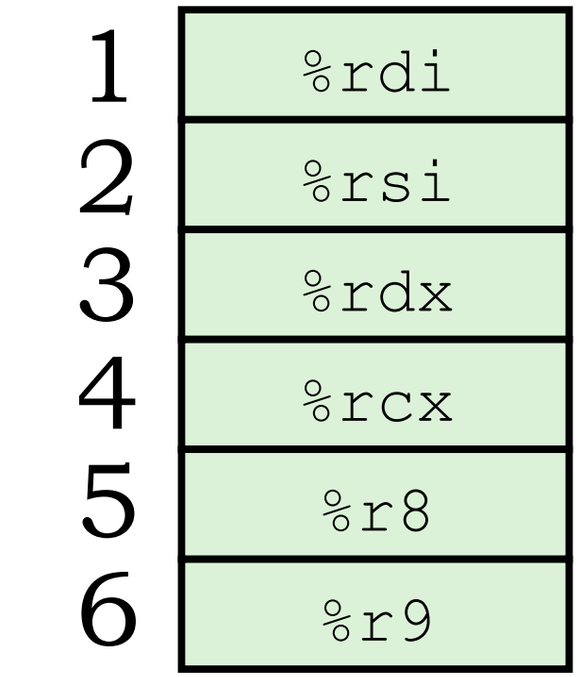
PROCEDURES

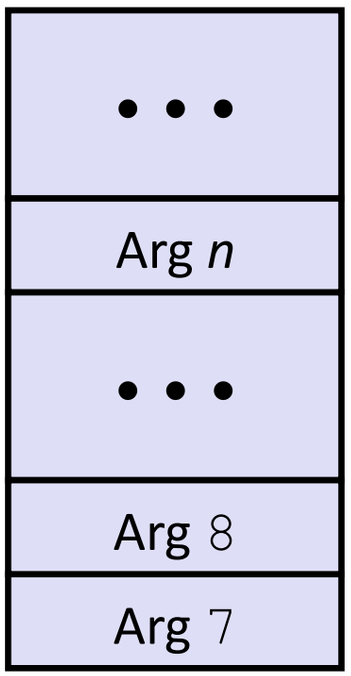
PROCEDURE DATA FLOW

- Where do we put the parameters for a function?
  - The first six parameters are placed in registers

- Where do we put the return value from a function?
  - into `%rax`

- What if we have more than 6 parameters?
  - the rest go onto the stack
    - pushing the last one on first
    - so the 7th comes off first





PROCEDURES

PROCEDURE DATA FLOW EXAMPLE

```
void multstore
  (long x, long y, long *dest)
{
     long t = mult2(x, y);
     *dest = t;
}
```

```
long mult2
   (long a, long b)
{
   long s = a * b;
   return s;
}
```

PROCEDURES

PROCEDURE DATA FLOW EXAMPLE

```
void multstore
  (long x, long y, long *dest)
{
     long t = mult2(x, y);
     *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  •••
```

```
long mult2
   (long a, long b)
{
   long s = a * b;
   return s;
}
```

PROCEDURES

PROCEDURE DATA FLOW EXAMPLE

```
void multstore
  (long x, long y, long *dest)
{
     long t = mult2(x, y);
     *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  •••
```

```
long mult2
   (long a, long b)
{
   long s = a * b;
   return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax         # a
  400553:  imul   %rsi,%rax         # a * b
  # s in %rax
  400557:  retq                     # Return
```

PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

PROCEDURES

- STACK
- PASSING CONTROL
- PASSING DATA
- MANAGING LOCAL DATA
- RECURSION

PROCEDURES

STACK-BASED LANGUAGES

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be “Reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in Frames
  - state for single procedure instantiation

PROCEDURES

CALL CHAIN EXAMPLE

Example Call Chain

```
yoo(…)
{
   •
   •
   who();
   •
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

PROCEDURES

CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain

yoo

amI() is recursive

PROCEDURES

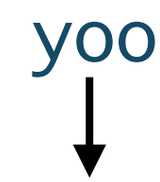
CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain



amI() is recursive

PROCEDURES

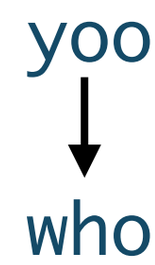
CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain



amI() is recursive

PROCEDURES

CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain

amI() is recursive

PROCEDURES

CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain

yoo
↓
who
↓
amI

amI() is recursive

PROCEDURES

CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain

PROCEDURES

CALL CHAIN EXAMPLE

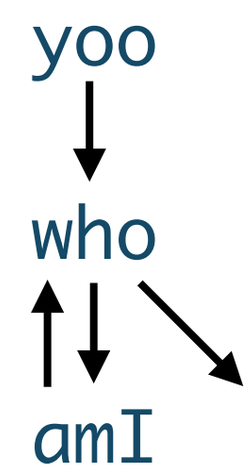
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

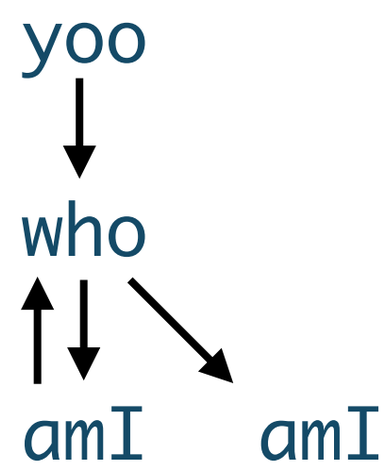
CALL CHAIN EXAMPLE

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain



amI() is recursive

PROCEDURES

CALL CHAIN EXAMPLE

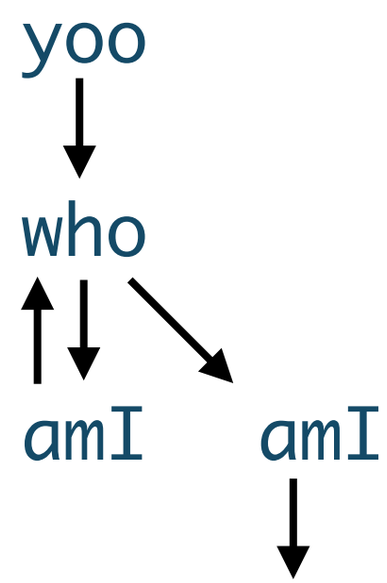
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

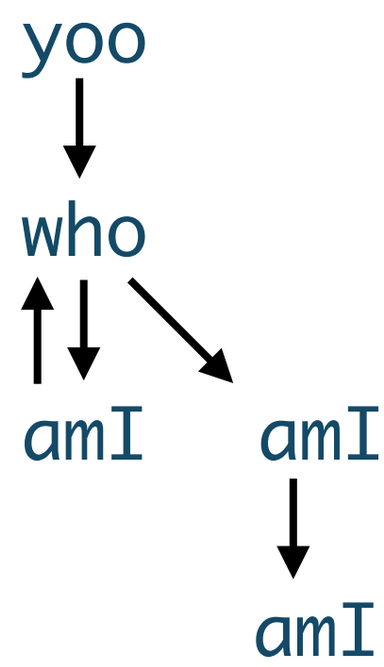
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

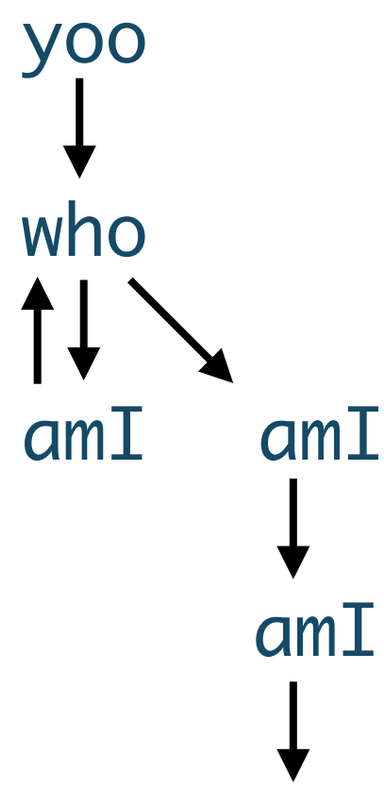
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

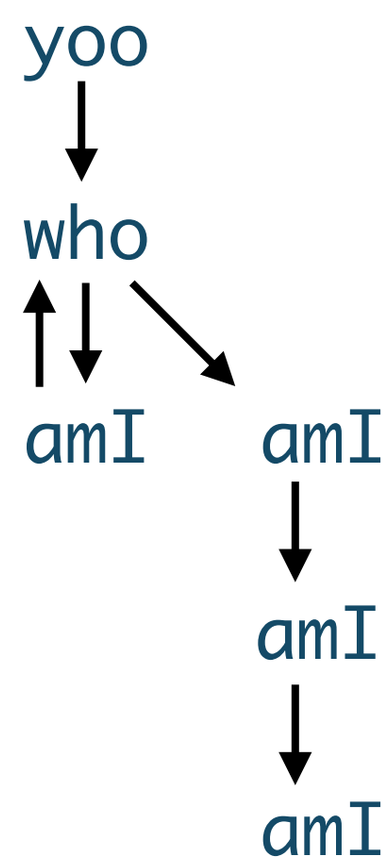
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

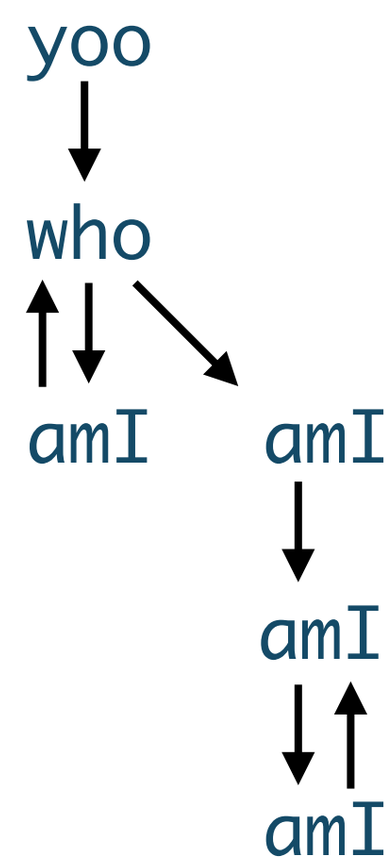
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

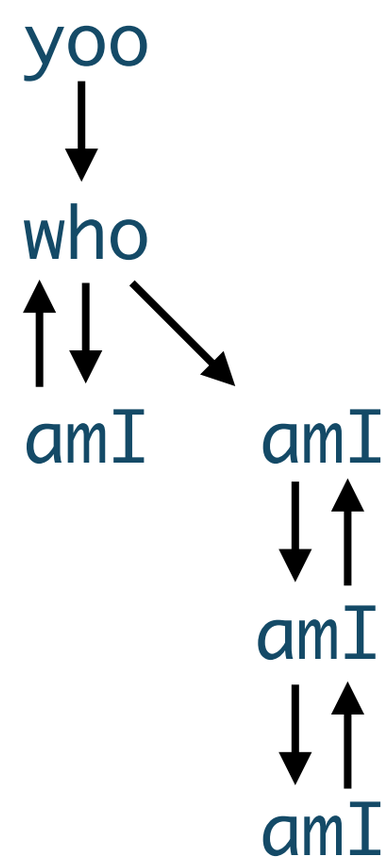
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

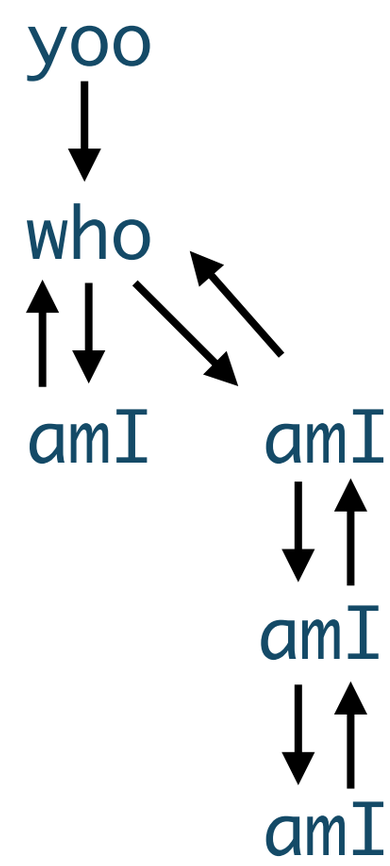
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

Example Call Chain



PROCEDURES

CALL CHAIN EXAMPLE

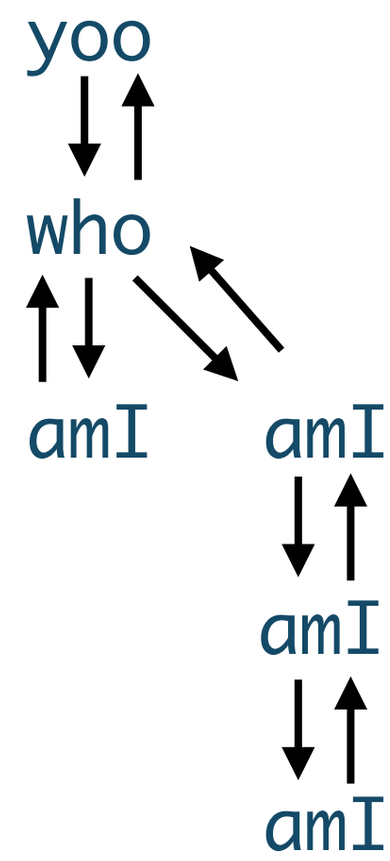
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

amI() is recursive

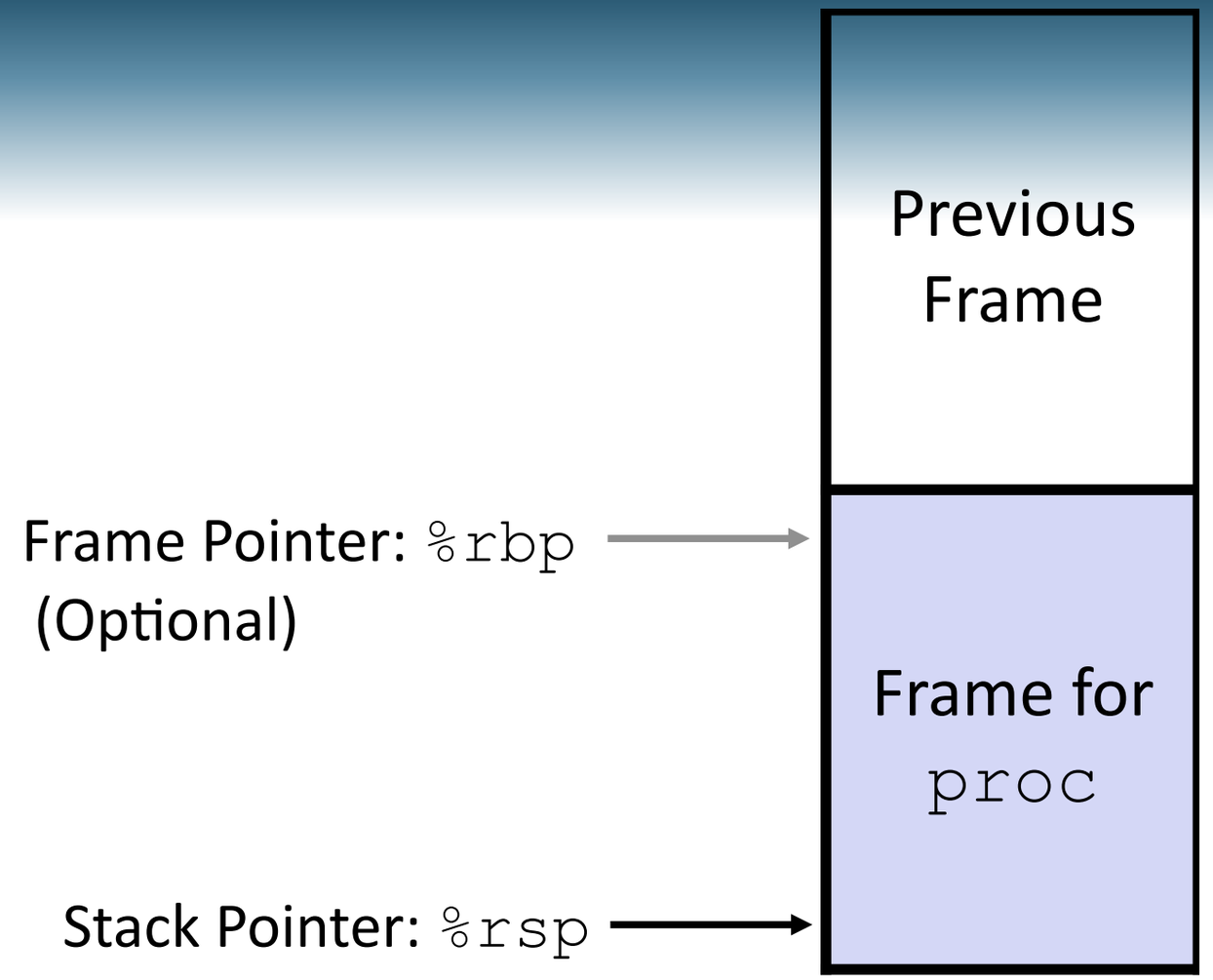
Example Call Chain



PROCEDURES

STACK FRAMES

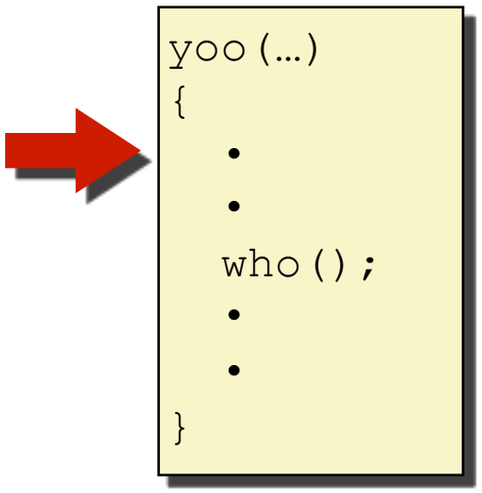
- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

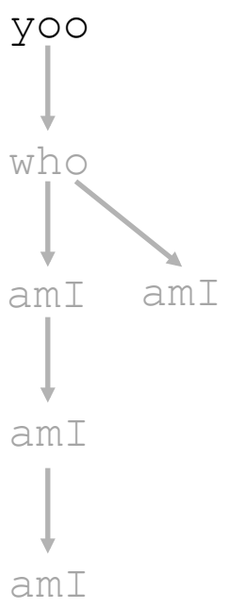


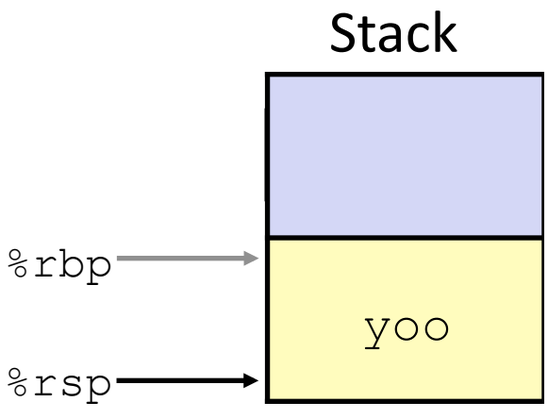
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by `ret` instruction

PROCEDURES

EXAMPLE

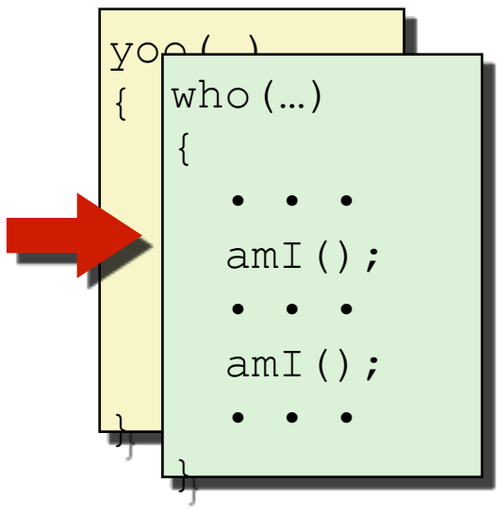


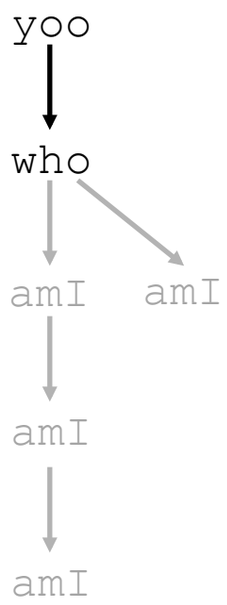


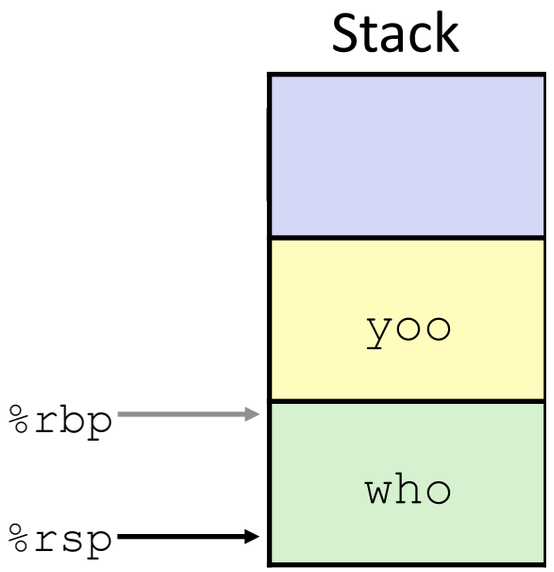


PROCEDURES

EXAMPLE

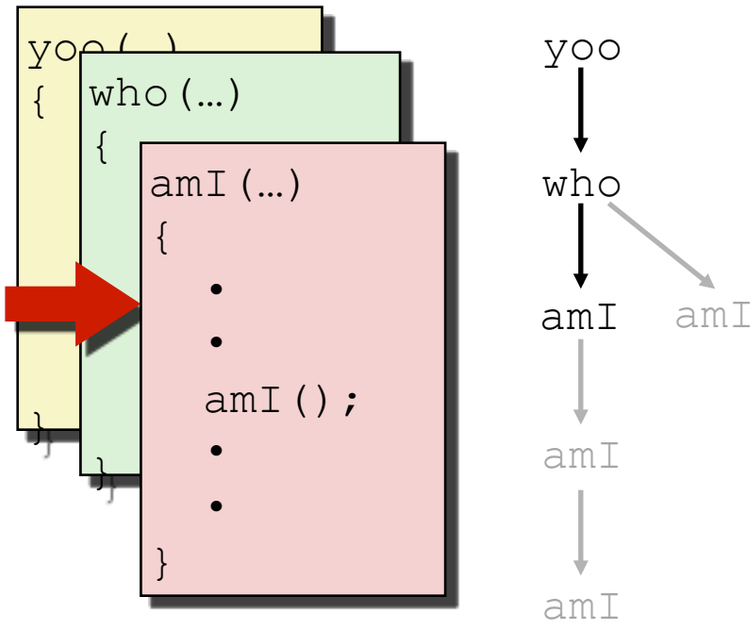


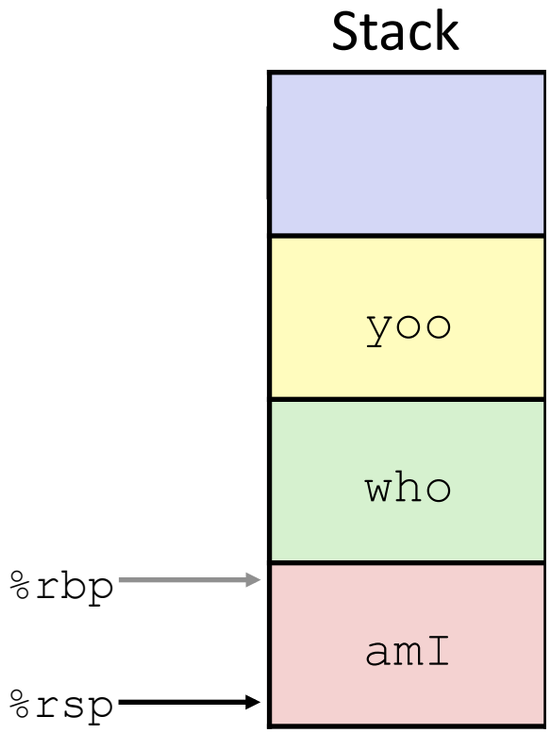




PROCEDURES

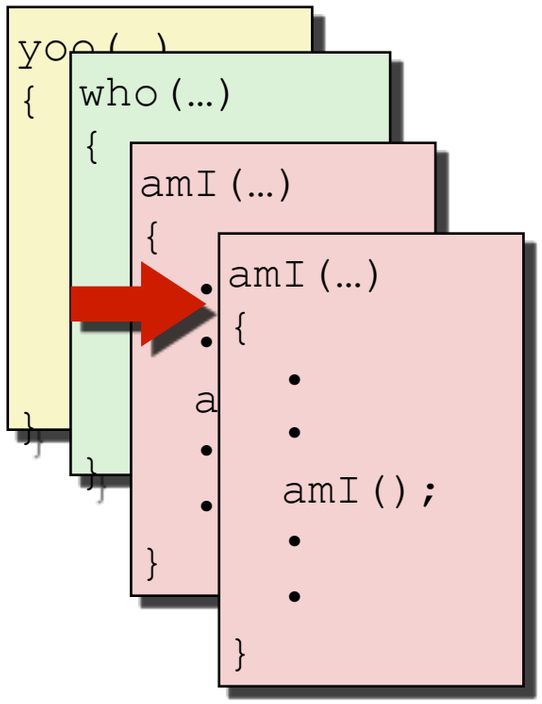
EXAMPLE

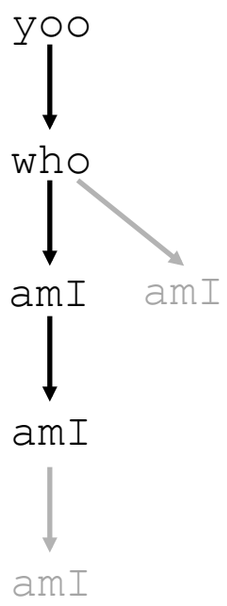


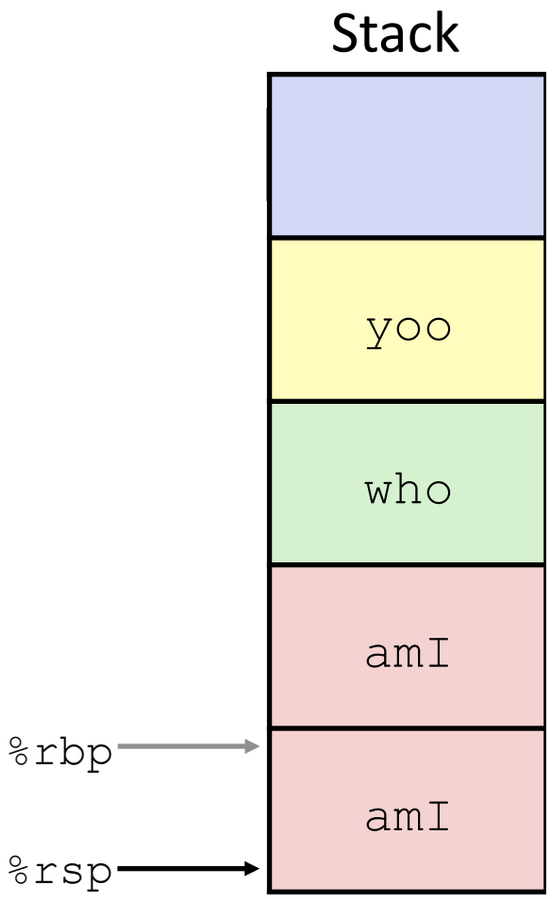


PROCEDURES

EXAMPLE

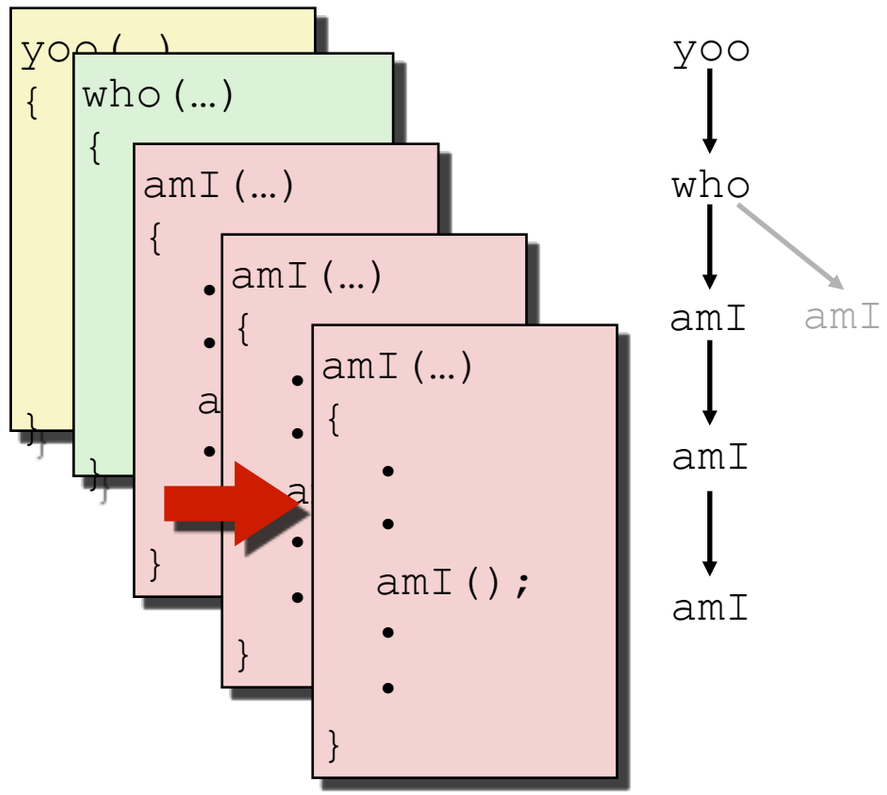


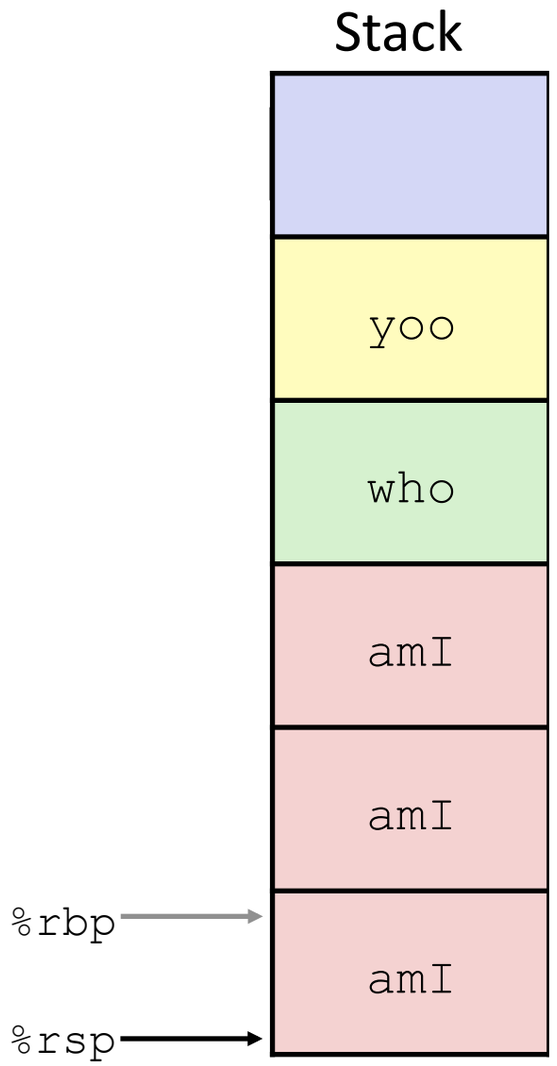




PROCEDURES

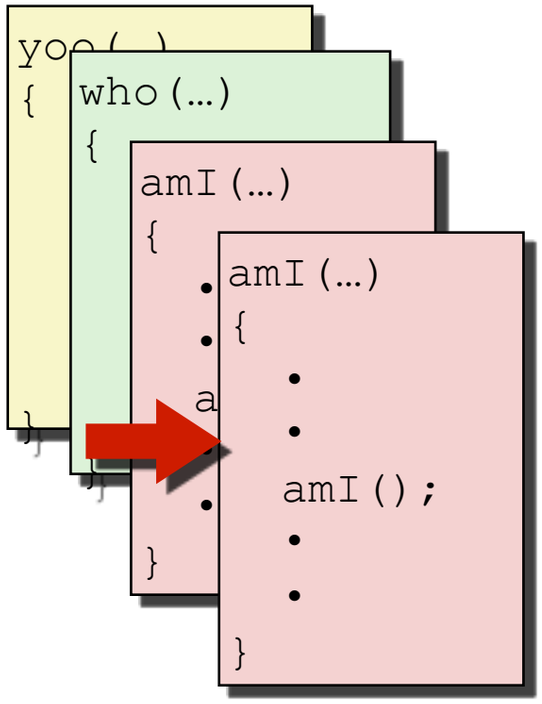
EXAMPLE

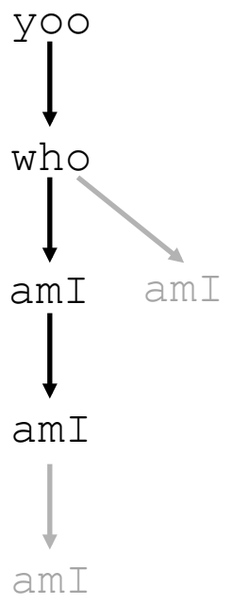


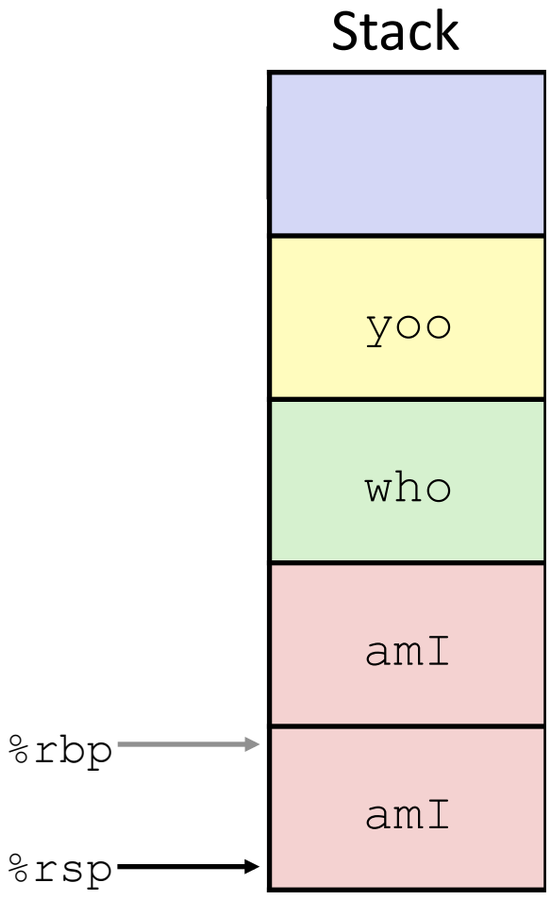


PROCEDURES

EXAMPLE

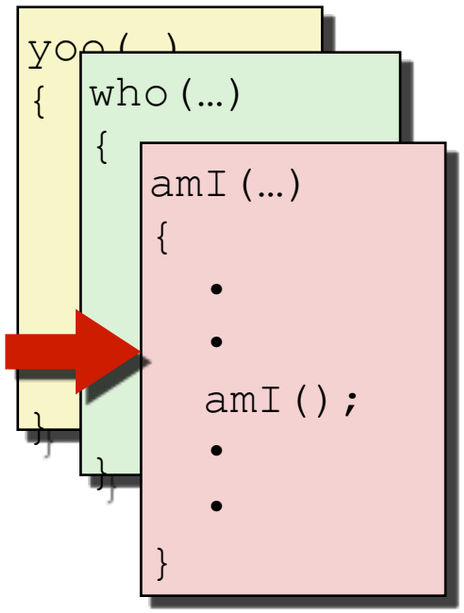


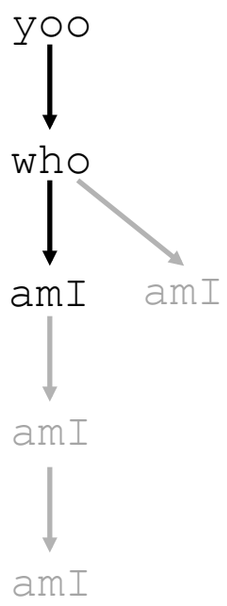


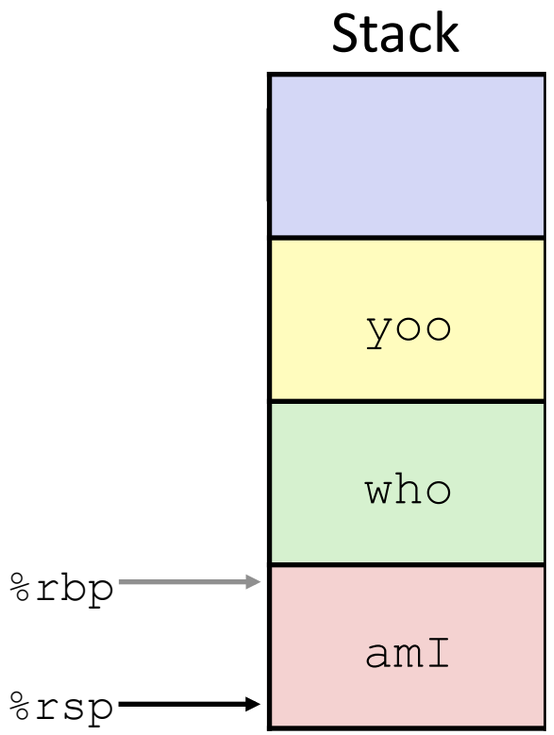


PROCEDURES

EXAMPLE

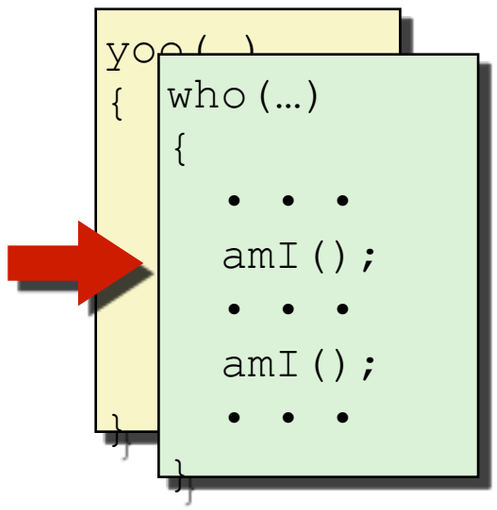


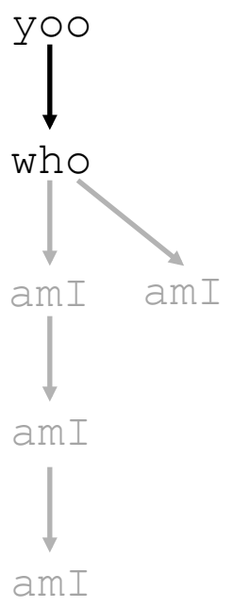


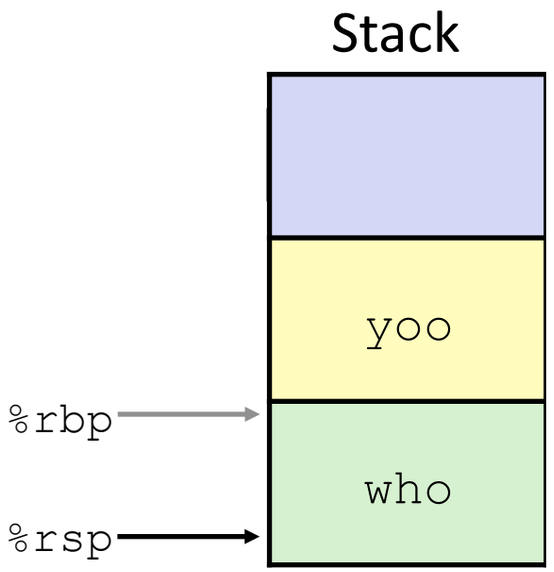


PROCEDURES

EXAMPLE

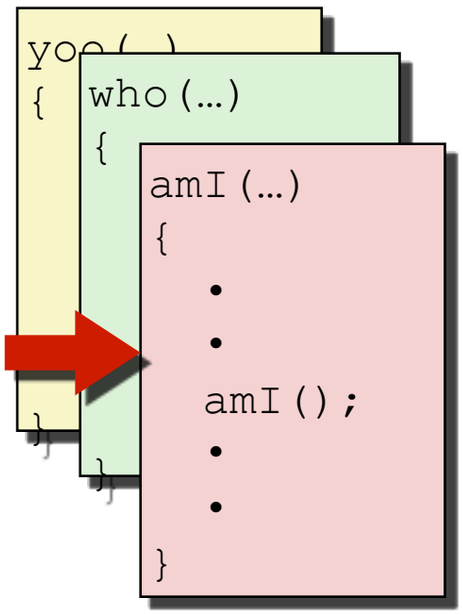


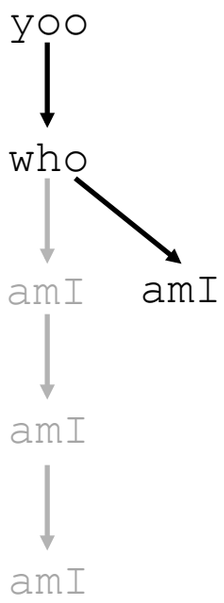


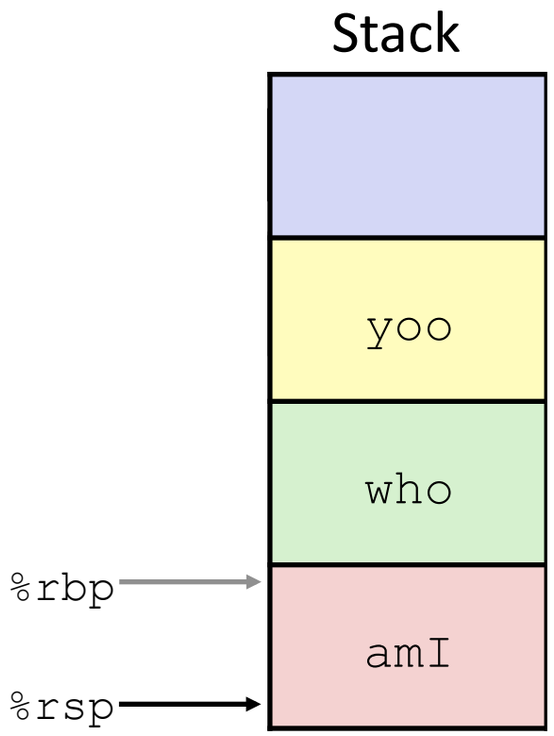


PROCEDURES

EXAMPLE

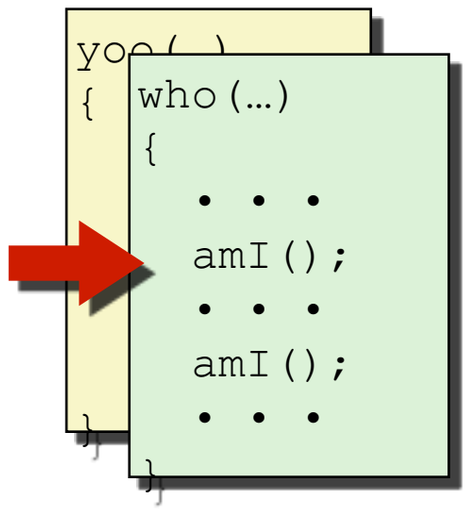


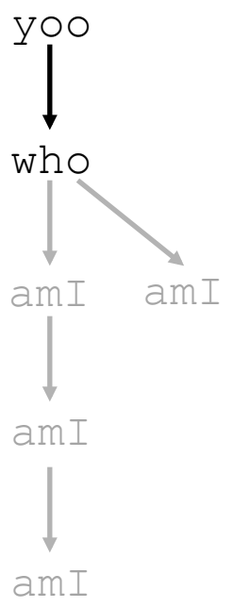


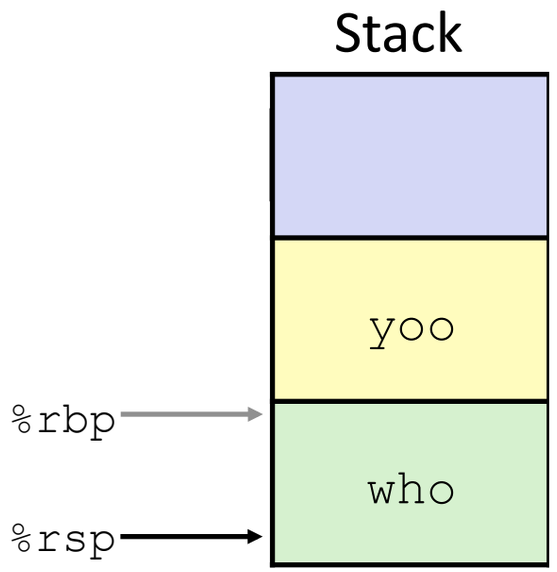


PROCEDURES

EXAMPLE

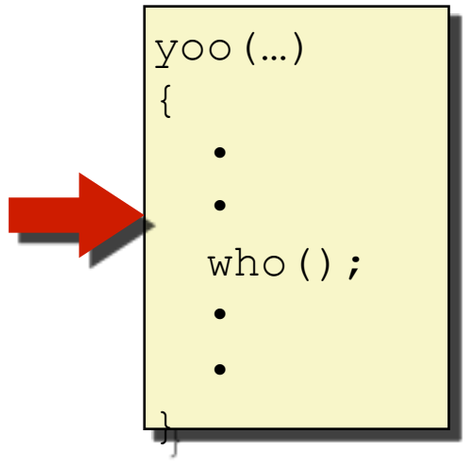


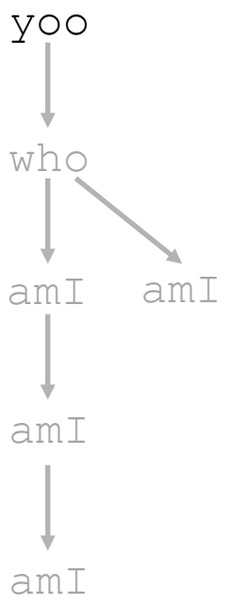


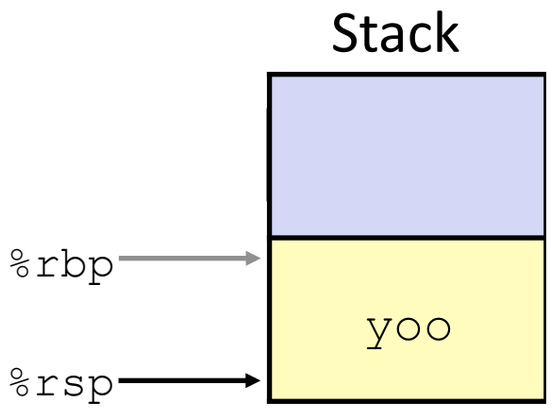


PROCEDURES

EXAMPLE





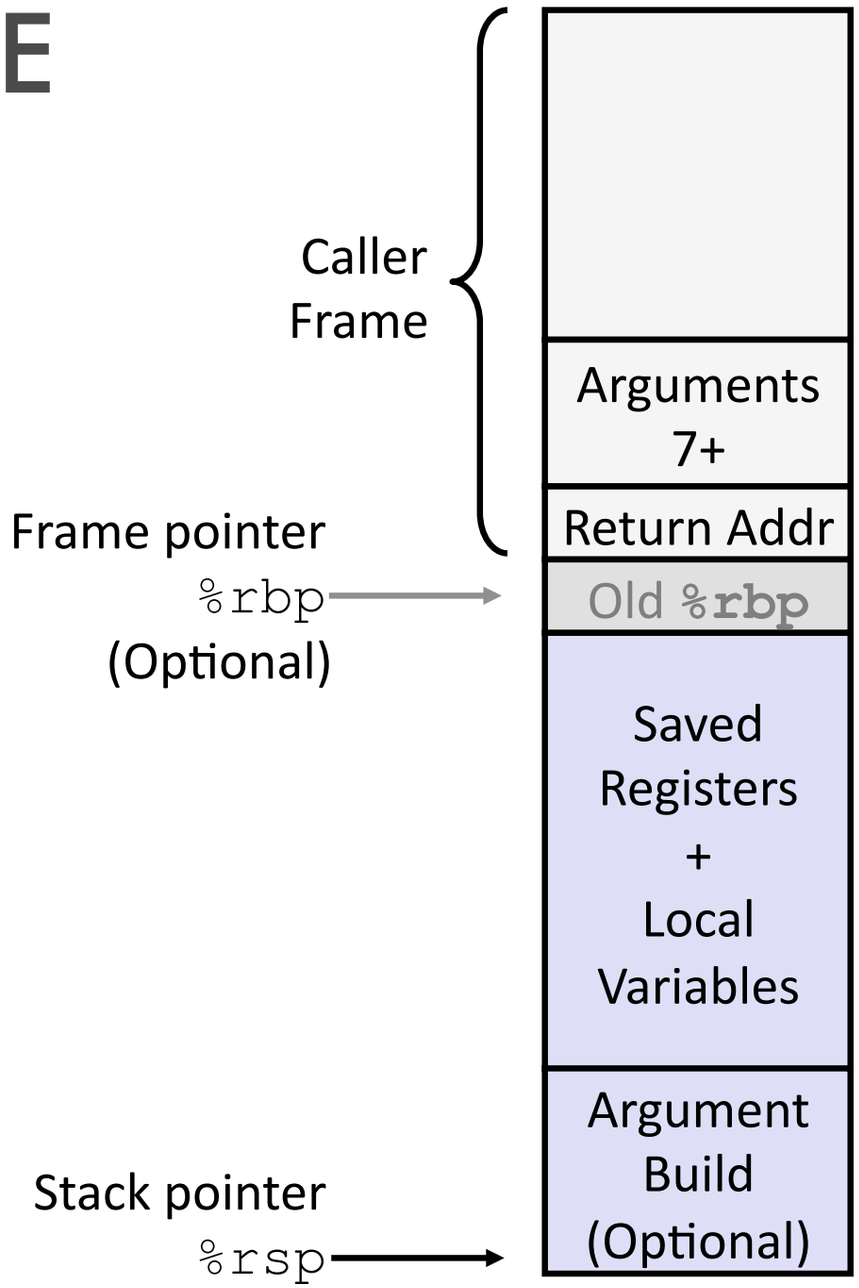


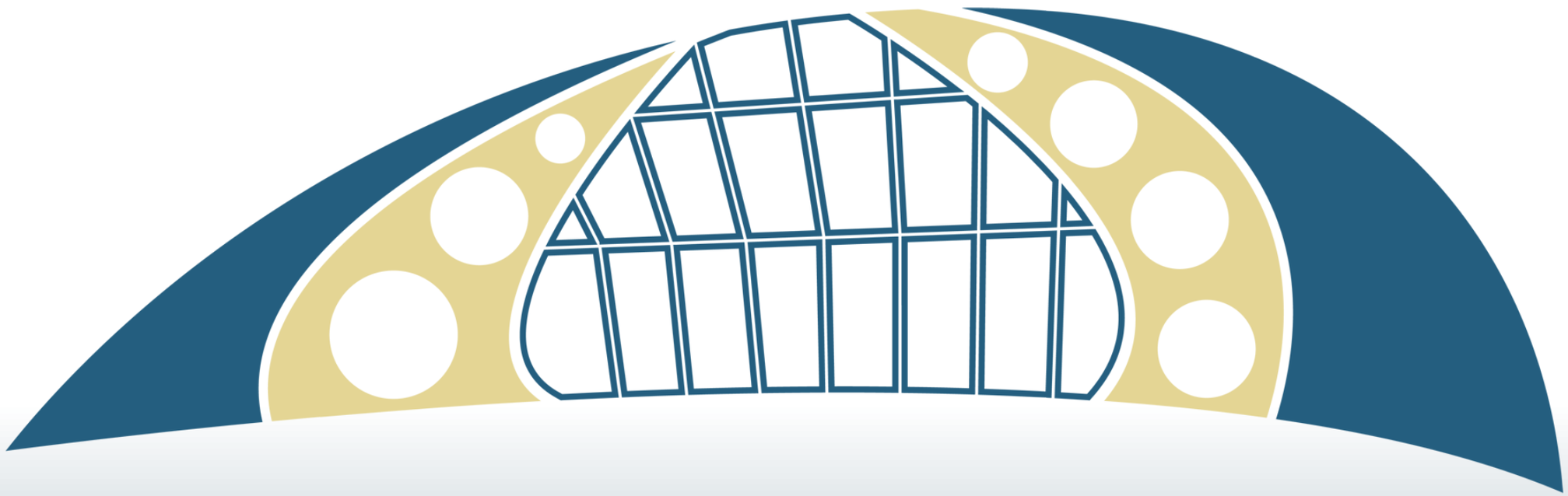
PROCEDURES

X86-64/LINUX STACK FRAME

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- **Caller Stack Frame**
  - Return address
    - Pushed by call instruction
  - Arguments for this call





WESTMONT INSPIRED

COMPUTING LAB

WORK IT OUT