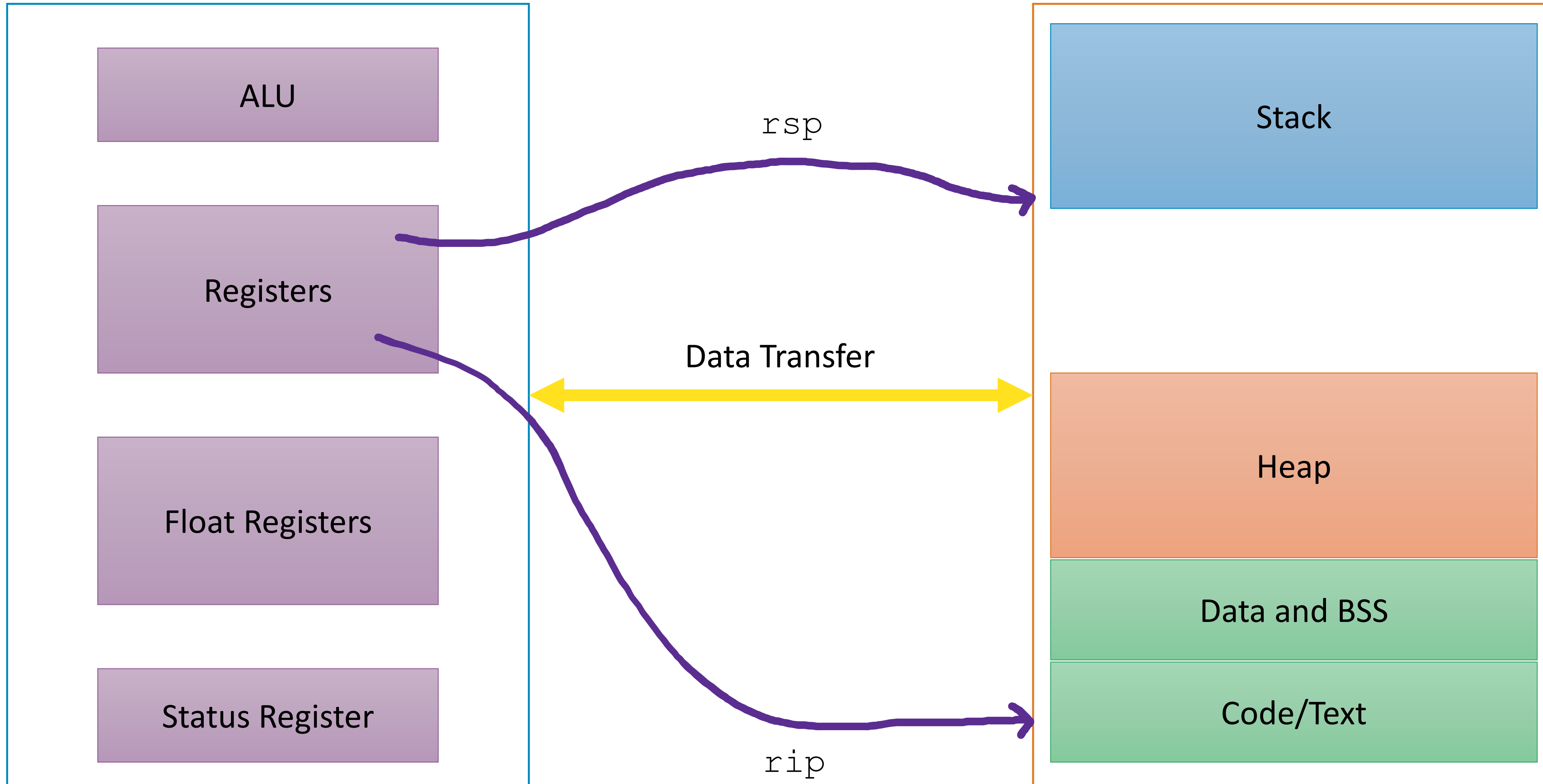
Drawing: Assembly

- Take three minutes to draw something related to “Assembly”

- Some ideas to remember
  - Source code
  - Assembly code
  - Object code
  - Registers
  - Operators
  - Operands


CPU
Memory
ALU
Registers
Float Registers
Status Register
rsp
Data Transfer
rip
Stack
Heap
Data and BSS
Code/Text

Checkpoint 2

0010
1010

Unsigned

8 4 2 1
0b1001

$1 \cdot 2^3$ $\quad$ $1 \cdot 2^0$

8 + 1 = 9

0b1001 >> 1

0b0100 = 4

8 4 2 1

Signed ← Negative

0b1001 = -7

0b0110 ← complement

+ 1

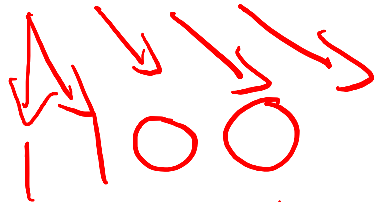
8 4 2 1
0b0111 ← 2's comp

4 + 2 + 1 = 7

signed

0b1001 = -7

0b1001 >> 1



0b1100 = -4

0b0011

\+ 1

8 4 2 1

0b0100 = 4

NaN → sign → does not matter

exponent → 11111111

mantissa → non-zero

$!!(x \wedge y)$

$\wedge$: returns 0 if y == x

returns # if y != x

case 1: 0

$!!(0) = !(1) = 0$

case 2: 1

$!!(-417) = !(0) = 1$

Assembly

Control Flow and Data

Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Compute arithmetic operations on data
  - Data from immediate values, registers, or memory

- Transfer control
  - Conditional branches
  - Unconditional jumps to/from procedures

Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Compute arithmetic operations on data
  - Data from immediate values, registers, or memory

- Transfer control
  - Conditional branches
  - Unconditional jumps to/from procedures

Processor State (partial)

Temporary data

- rax, …

Location of runtime stack

- rsp

Location of current code control point

- rip

Status of recent tests

- CF, ZF, SF, OF

Registers

rax (return val)	r8
rbx	r9
rcx (4th arg)	r10
rdx (3rd arg)	r11
rsi (2nd arg)	r12
rdi (1st arg)	r13
rsp (stack ptr)	r14
rbp	r15

rip — Instruction pointer

CF ZF SF OF — Condition codes

Unconditional Jump

A jump instruction can cause the execution to switch to a completely new position in the program (updates the program counter)

- `jmp Label`
- `jmp [Operand]`

`rip = an address of the next instruction to execute`

```
.L0:
  mov    rax, 0
  jmp    .L1
  mov    rdx, [rax]

.L1:
  mov    rcx, rax
```

```
jmp [rax]
```

Condition Codes

Single bit registers

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

- These "flags" are implicitly set (as 0 or 1) by arithmetic and comparison operations

- One exception: not set by `lea` instruction

Condition Codes: compare

Instruction `cmp` explicitly sets condition codes

```
cmp a, b ; computing a-b without storing the result
```

- `ZF` set if `(a-b) == 0`
- `SF` set if `(a-b) < 0` (as signed)
- `CF` set if carry out from most significant bit (used for unsigned comparisons)
- `OF` set if two’s-complement (signed) overflow

Condition Codes: test

Instruction `test` explicitly sets condition codes

```
testq a, b ; computing a&b without storing result
```

- `ZF` set when `a&b == 0`
- `SF` set when `a&b < 0`
- Test for zero: `testq rax, rax`

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
```

No

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

Jump

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

No

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

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)

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
```

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

no jump

rsi – rdi

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

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

jump

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

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

no jump

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

$13 \& 13 \overset{?}{\neq} 0$

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

```
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 pointer arithmetic without a memory reference

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

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

- `p = 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

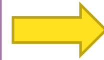
Loops

All use conditions and jumps

- 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    ;
   and      rdx, 1      ;  t = x & 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;
}
```

```
        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
```

Register	Use(s)
rdi	x
rax	result

For loops

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

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

Initial test can often be optimized away:
`for (j = 0; j < 99; j++)`

```
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
```

Register	Use(s)
rdi	Argument x
rax	result

Practice with Loops

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

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

L0:
  add rax, rdx
  inc rdx

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

```
long loop(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

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

L0:
  add rax, rdx
  inc rdx

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

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

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

    ret = ret + i;

  }

  return ret;
}
```

Subroutines

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

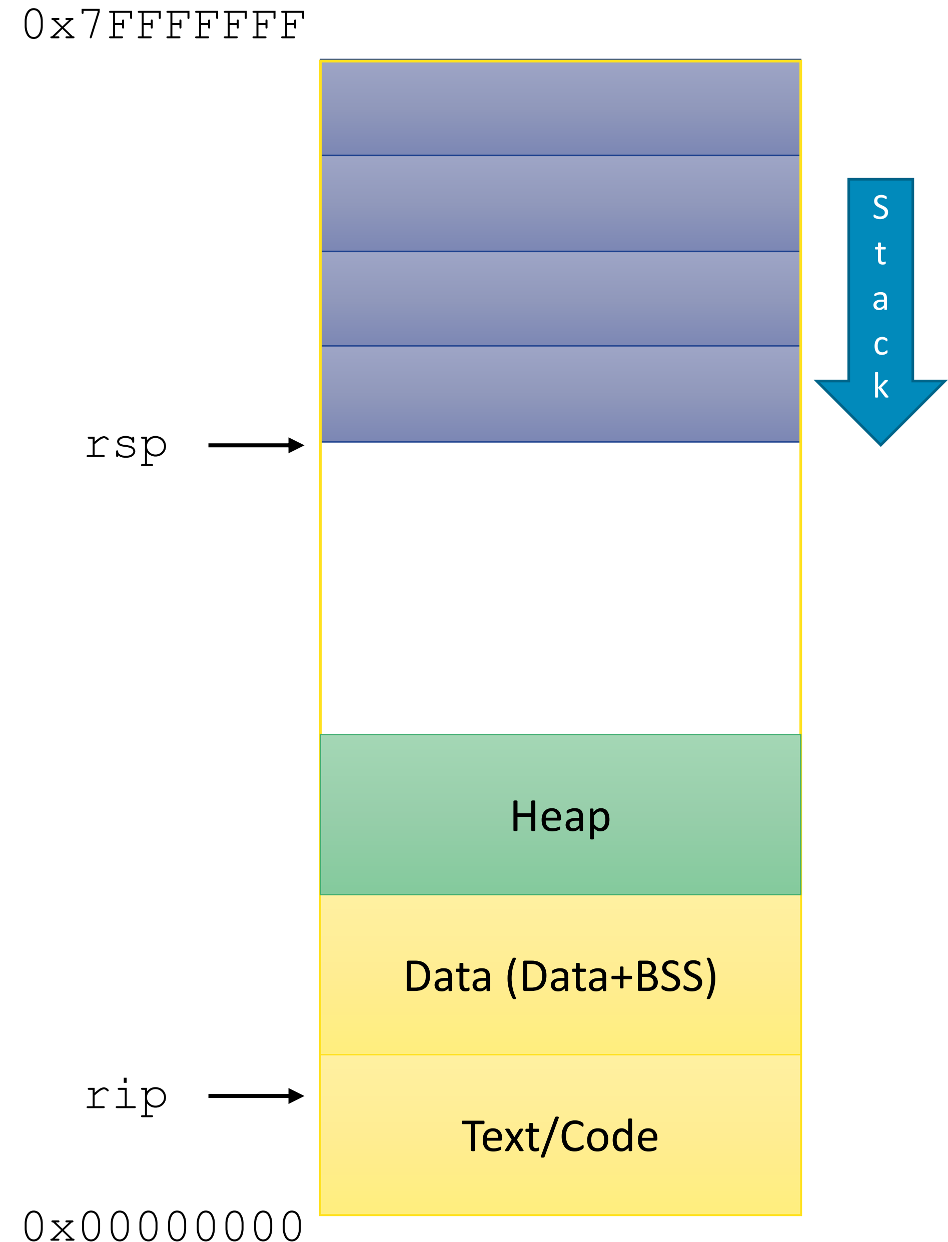
Procedures

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

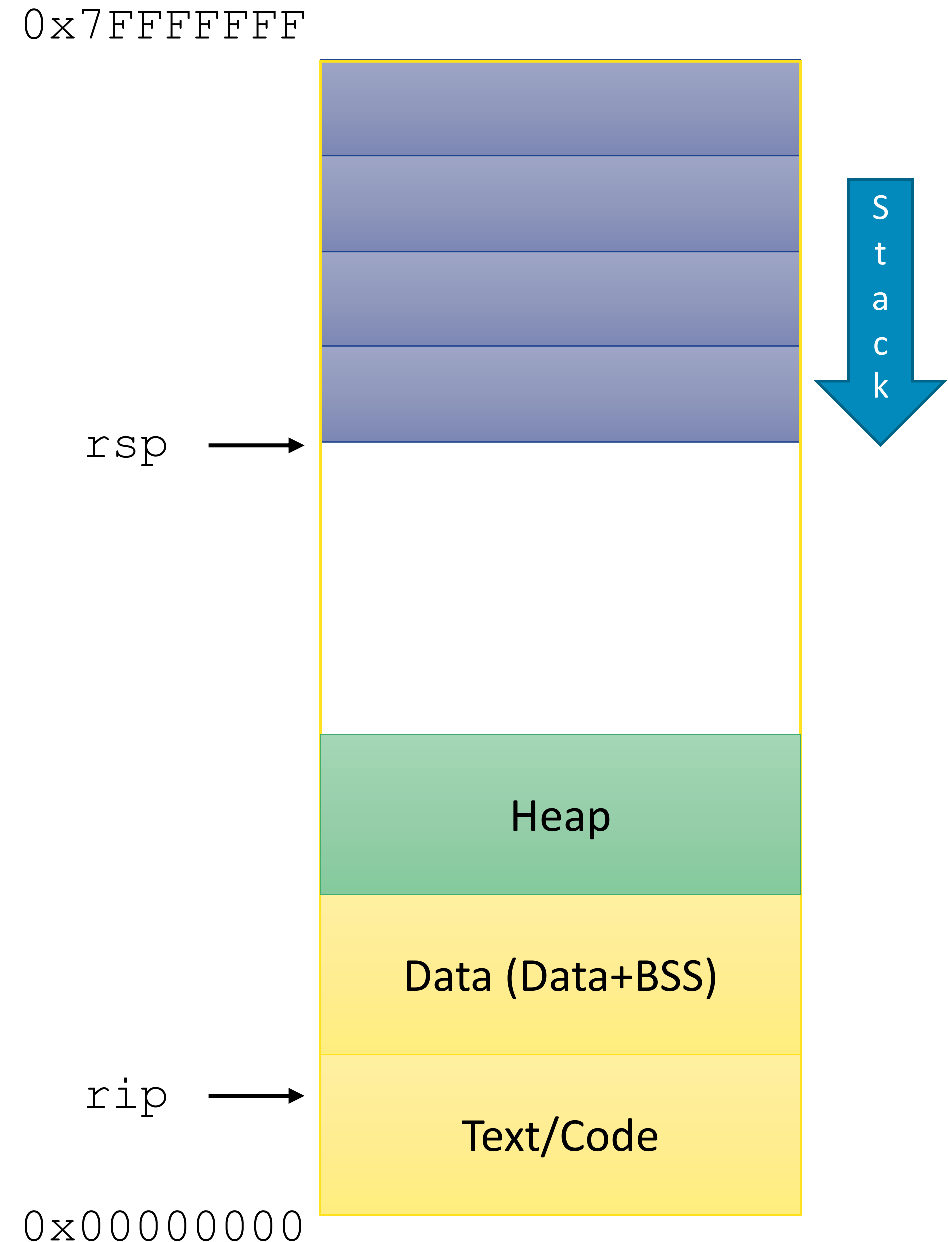
```
push SRC
• sub rsp, 8     ; Depends on size
• mov [rsp], SRC

pop DEST         ; (M or R)
• mov DEST, [rsp]
• add rsp, 8     ; Depends on size
```

explicitly modify `rsp`

```
• sub rsp, 4
• add rsp, 4
```

modify memory above `rsp`

```
• mov [rsp + 4], 47
```



Modifying the Stack

```
call ADDRESS
```

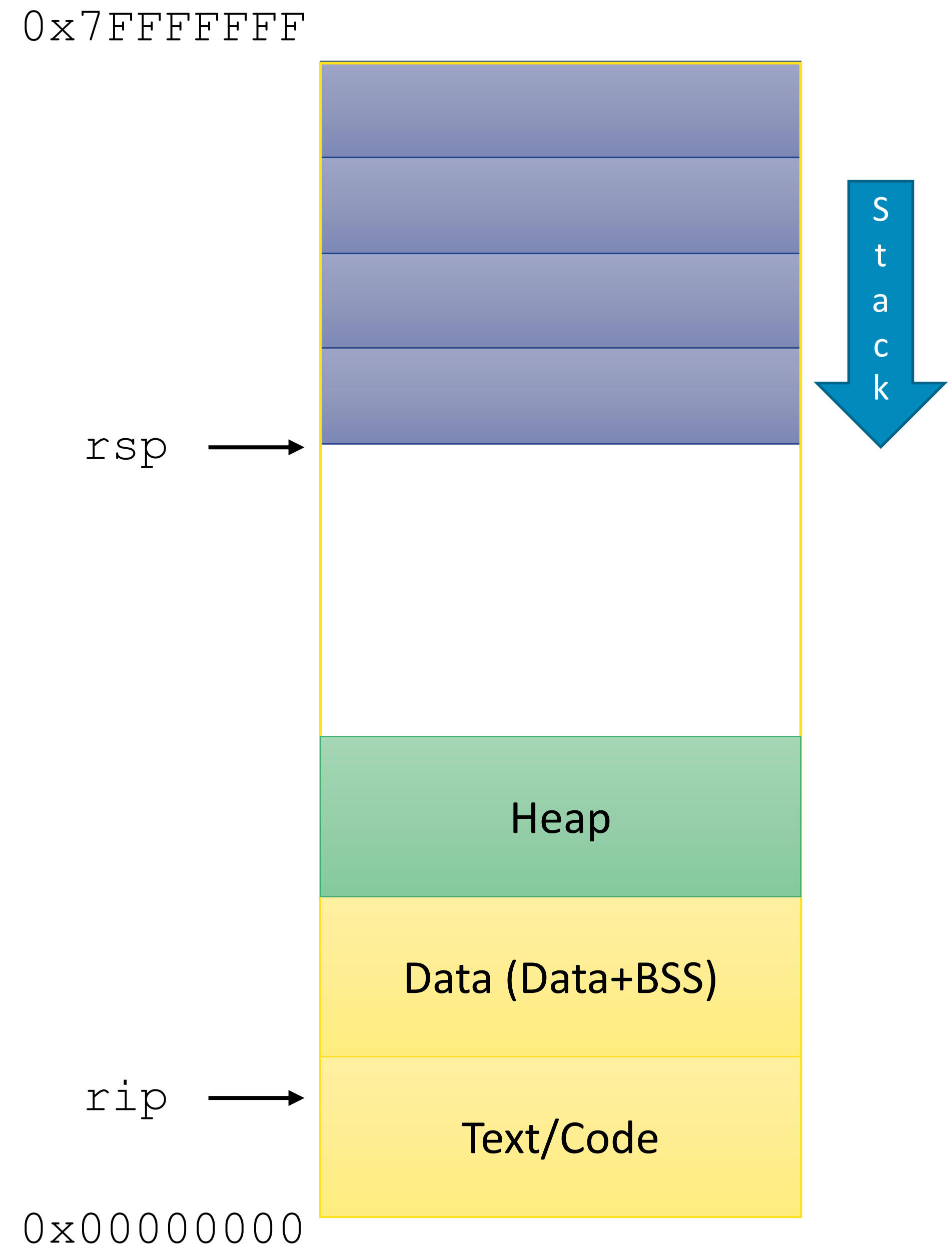
- `sub rsp, 8`
- `mov [rsp], rip`
- `jmp ADDRESS`

`push rip`

```
ret
```

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

`pop rip`



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
```

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

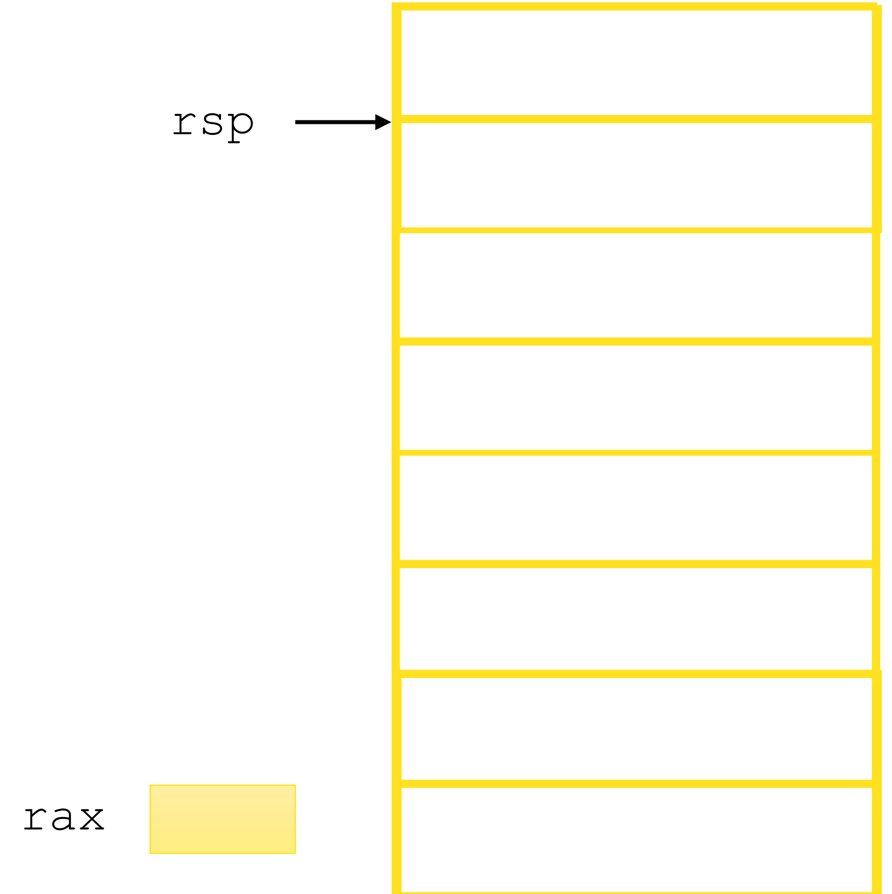
Exit code

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

16?
Return value?

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
```

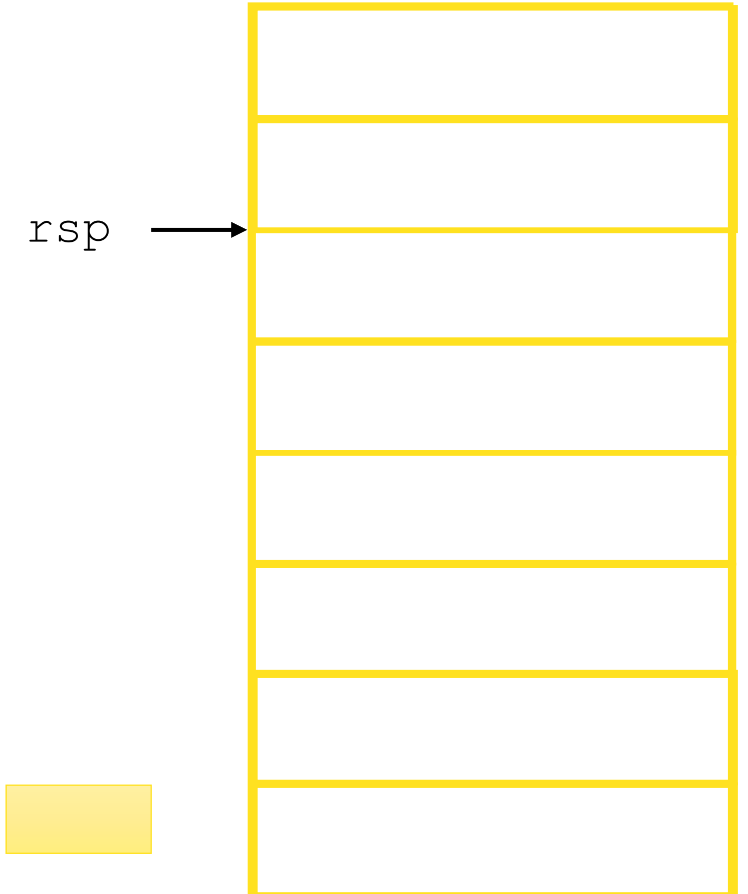


What is the value in rax immediately before main returns?
What is the value in rsp immediately before main returns?

```
push SRC
1. sub rsp, 8
2. mov [rsp], SRC
```

Practice Modifying the Stack

```
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
```



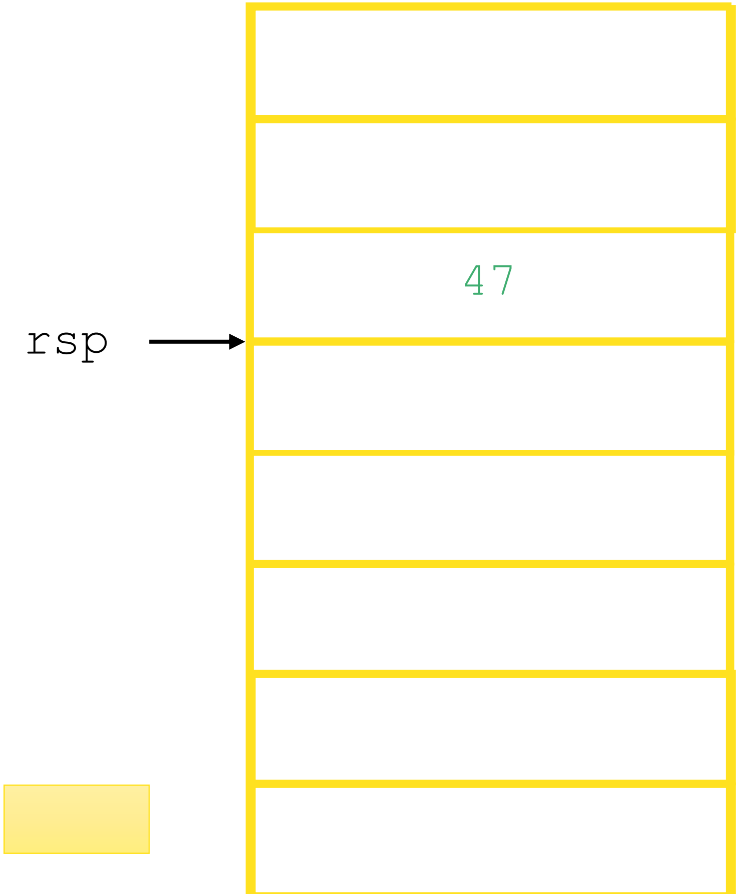
rax

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
```

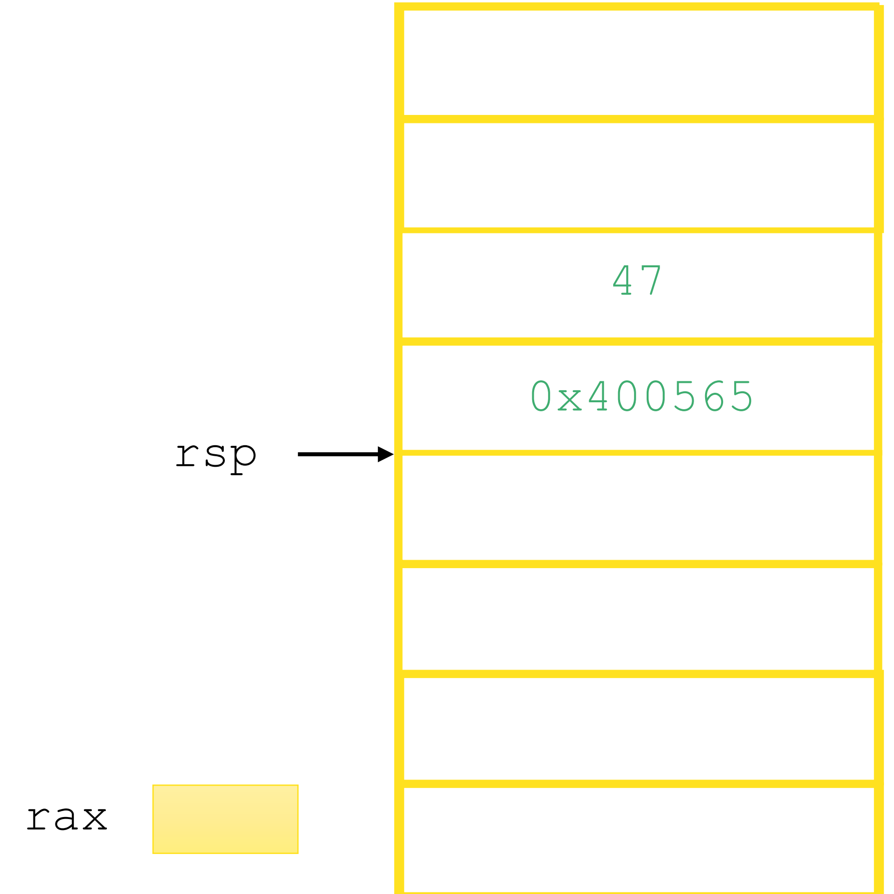
```
0x400557 <fun>:
  400557: mov  [rsp + 16], 13
  40055a: ret

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
```

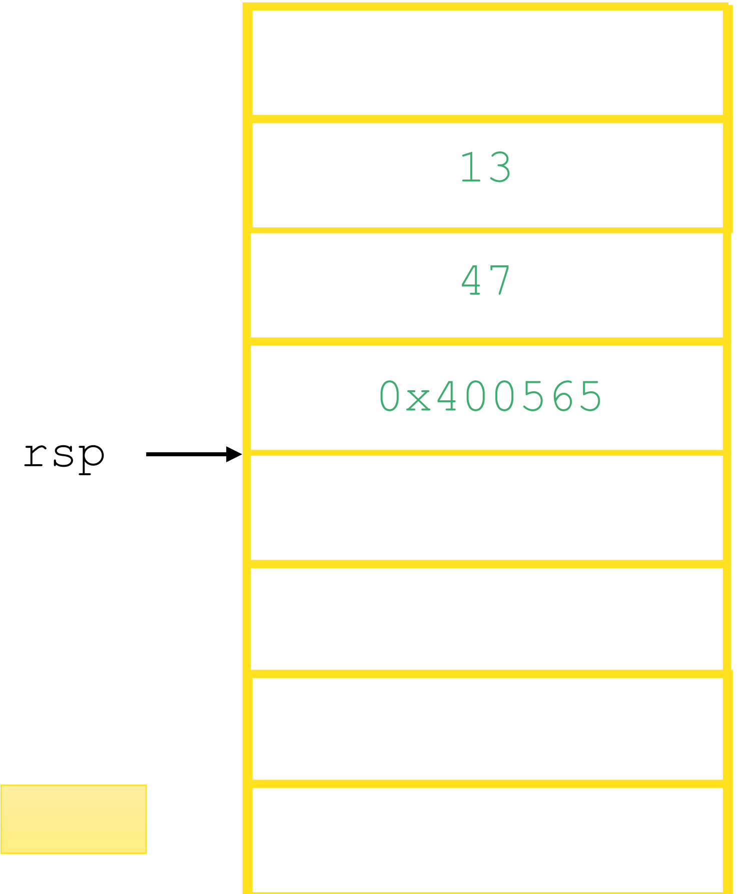


What is the value in rax immediately before main returns?
What is the value in rsp immediately before main returns?

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
```



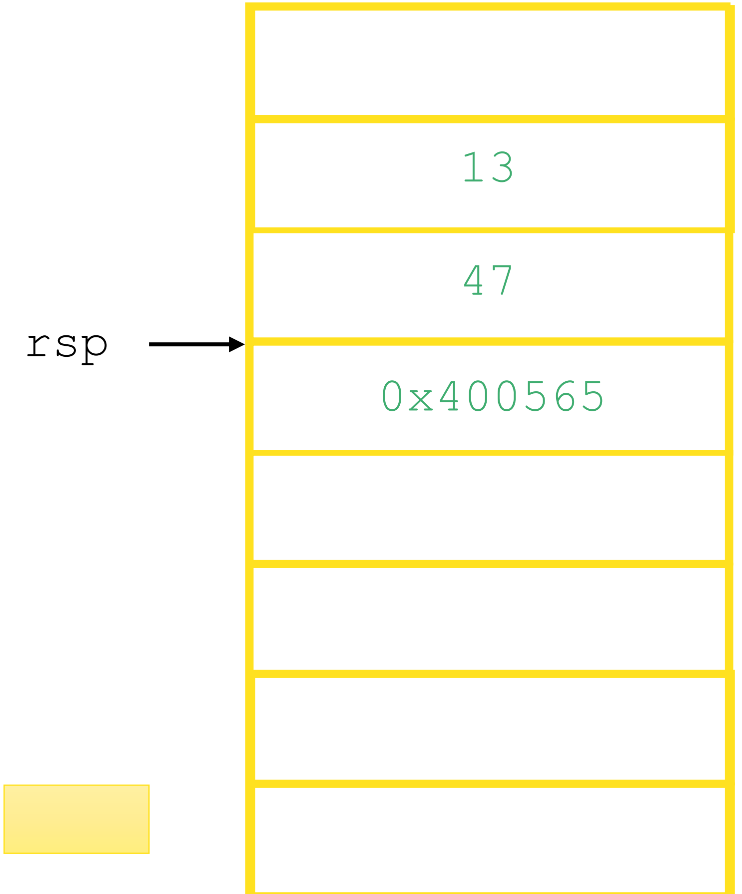
What is the value in rax immediately before main returns?
What is the value in rsp immediately before main returns?

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>
  400565: pop  rax
  400566: add  rax, [rsp]
  40056a: add  rsp, 8
  40056e: ret
```

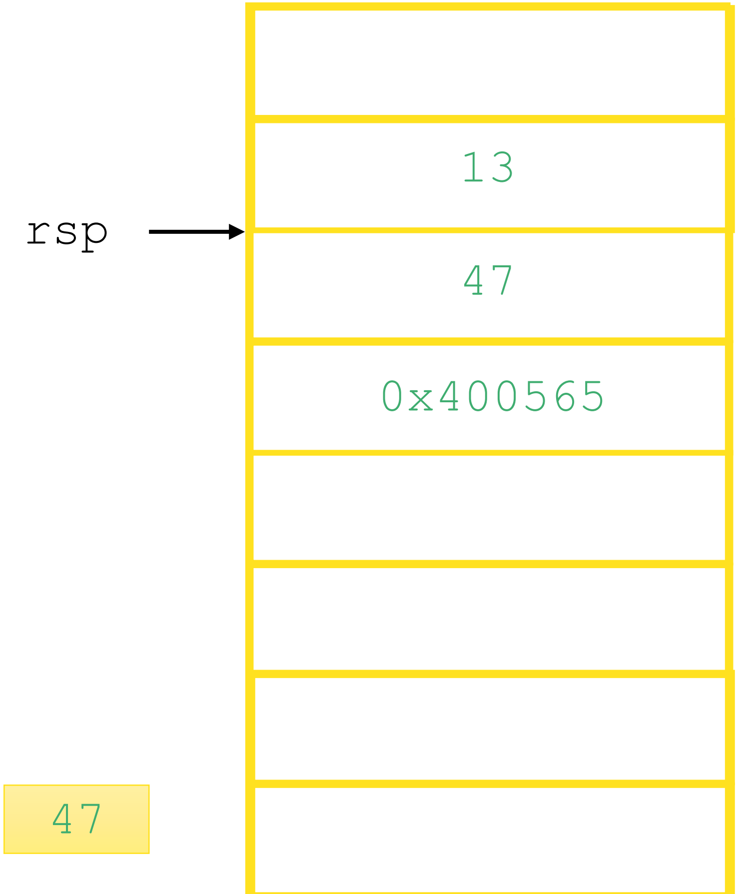
rip → 400565



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

Practice Modifying the Stack

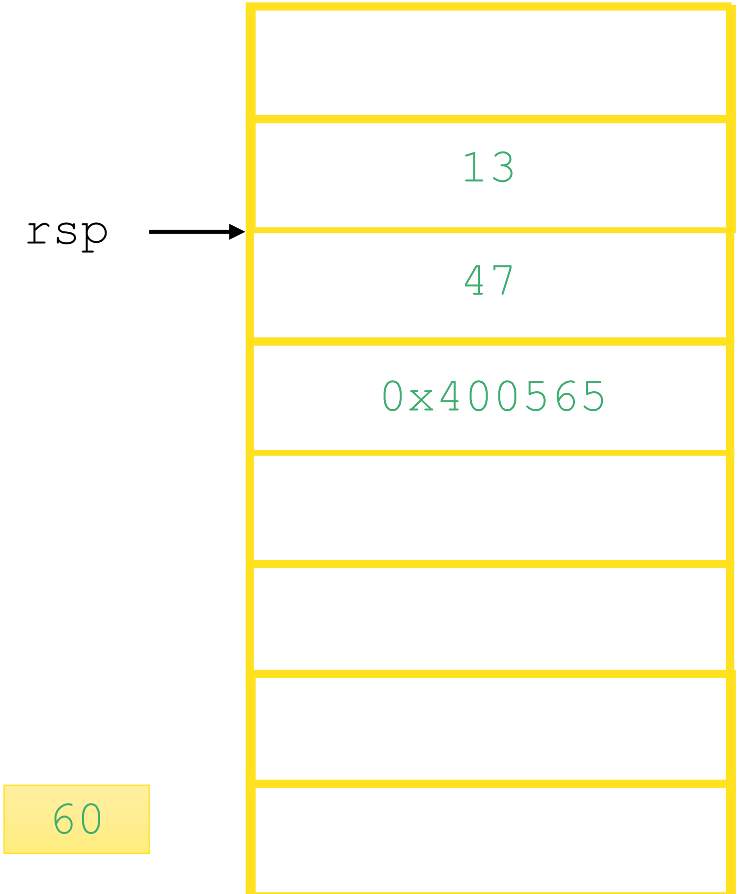
```
0x400557 <fun>:
   400557: mov  [rsp + 16], 13
   40055a: ret

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

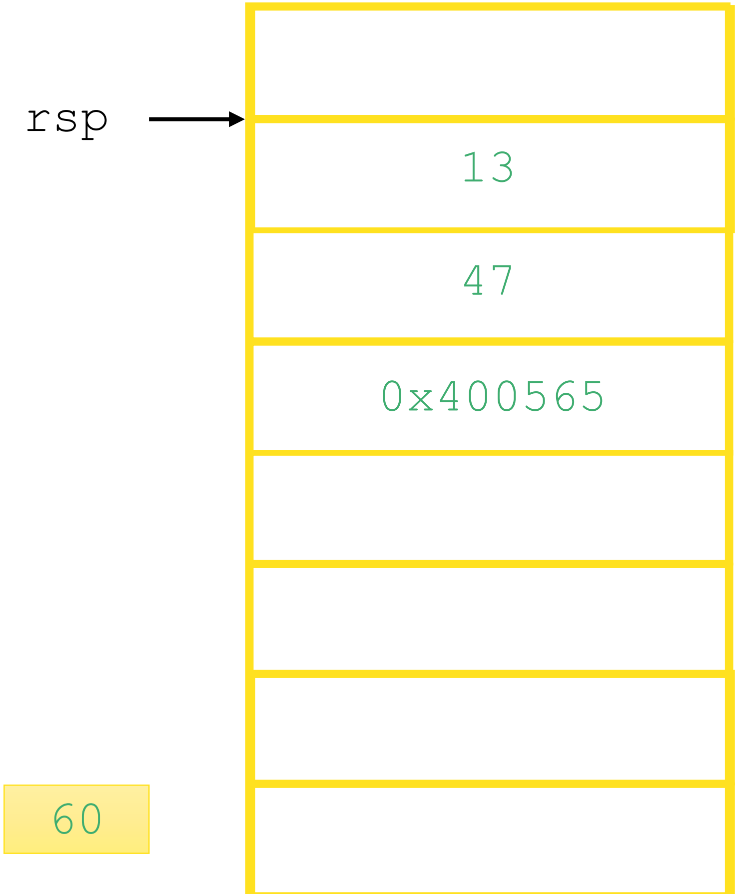
```
0x400557 <fun>:
   400557: mov  [rsp + 16], 13
   40055a: ret

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

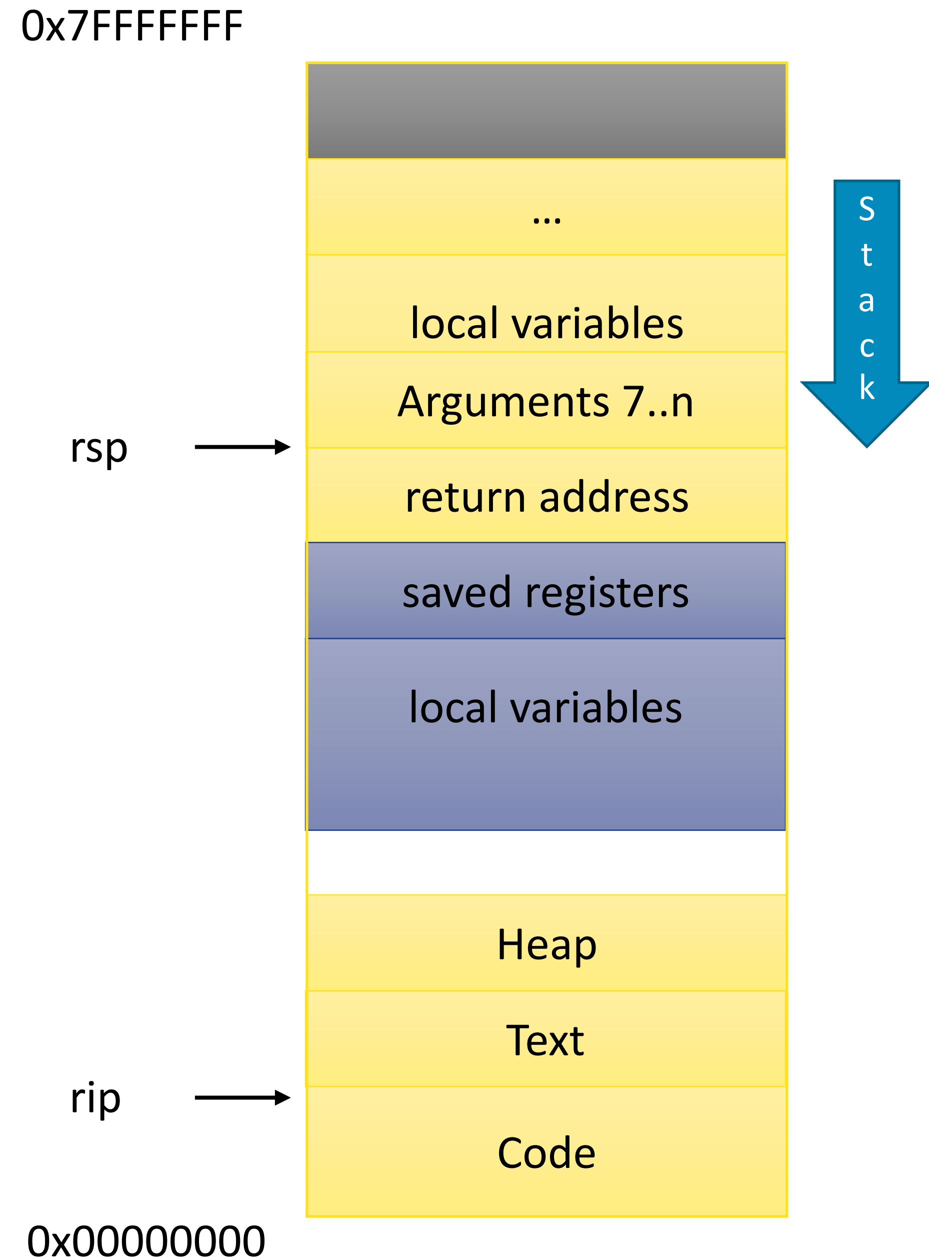
```
0x400557 <fun>:
   400557: mov  [rsp + 16], 13
   40055a: ret

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?

Stack Frames

- Each function called gets a stack frame
- Passing data:
  - calling procedure P uses registers (and stack) to provide parameters to Q.
  - Q uses register rax for return value
- Passing control:
  - call <proc>
    - Pushes return address (current rip) onto stack
    - Sets rip to first instruction of proc
  - ret
    - Pops return address from stack and places it in rip
- Local storage:
  - allocate space on the stack by decrementing stack pointer, deallocate by incrementing



Procedure Call Example: 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;
}
```

```
func1:
  addl    edi, esi
  addl    ecx, edx
  addl    r9d, r8d
  movl    16(rsp), eax
  addl    8(rsp), eax
```

```
main:
        movl    $1, edi
        movl    $2, esi
        movl    $3, edx
        movl    $4, ecx
        movl    $5, r8d
        movl    $6, r9d
        pushq   $8
        pushq   $7
        callq   _function1
        addq    $16, rsp
        retq
```

Exercise 2: Value Passing

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

0x400548 <first>:
  400548: lea 0x1(rdi),rsi
  40054c: sub $0x1, rdi
  400550: callq 400540 <last>
  400555: rep; ret

0x400556 <main>:
  400560: mov $4, rdi
  400563: callq 400548 <first>
  400568: addq $0x13, rax
```

What value gets returned by main?

rdi	rsi	rax	rip
			0x400560

Exercise 2: Value Passing

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

0x400548 <first>:
  400548: lea 0x1(rdi),rsi
  40054c: sub $0x1, rdi
  400550: callq 400540 <last>
  400555: rep; ret

0x400556 <main>:
  400560: mov $4, rdi
  400563: callq 400548 <first>
  400568: addq $0x13, rax
```

rdi	rsi	rax	rip
3	5	34	0x40056c

What value gets returned by main?

Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function 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

Recursive Function

```
/* Recursive bitcount */
long bitcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + bitcount_r(x >> 1);
}
```

```
bitcount_r:
  testq   rdi, rdi
  je      .L3
  pushq   rbx
  movq    rdi, rbx
  andl    $1, ebx
  shrq    rdi
  call    bitcount_r
  addq    rbx, rax
  popq    rbx
  ret
.L3: # Base Case
  movl    $0, eax
  ret
```

What is in the stack frame?