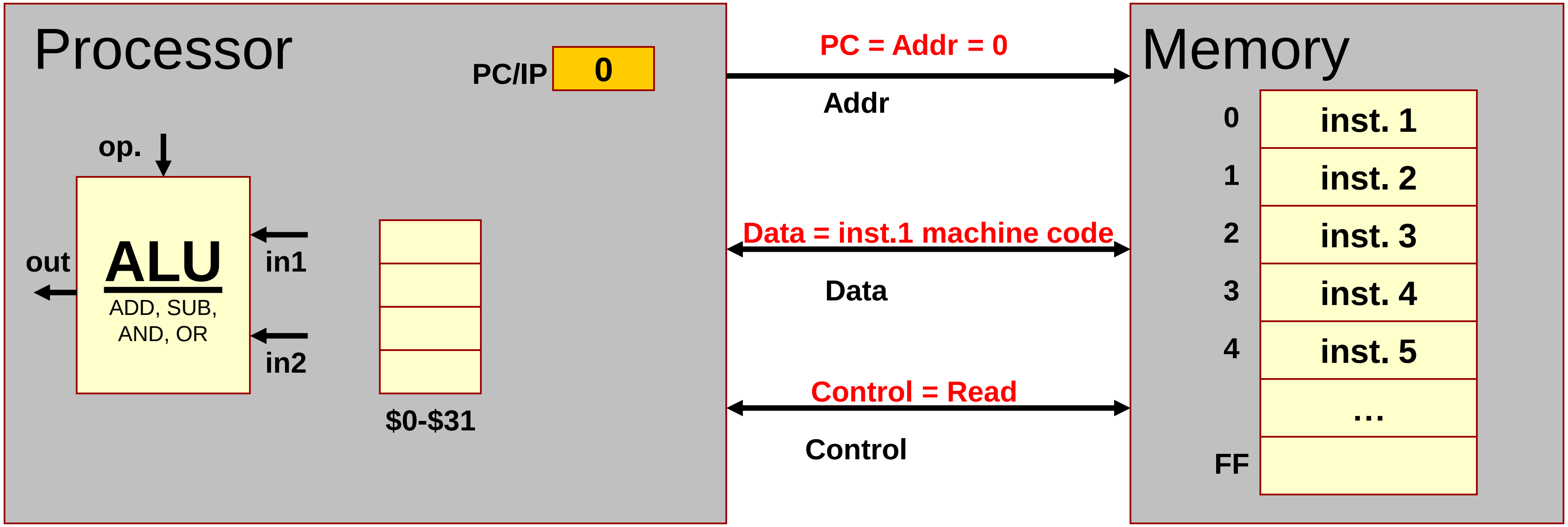
CS356 Unit 6

x86 Procedures

Basic Stack Frames

Review of Program Counter (Instruc. Pointer)

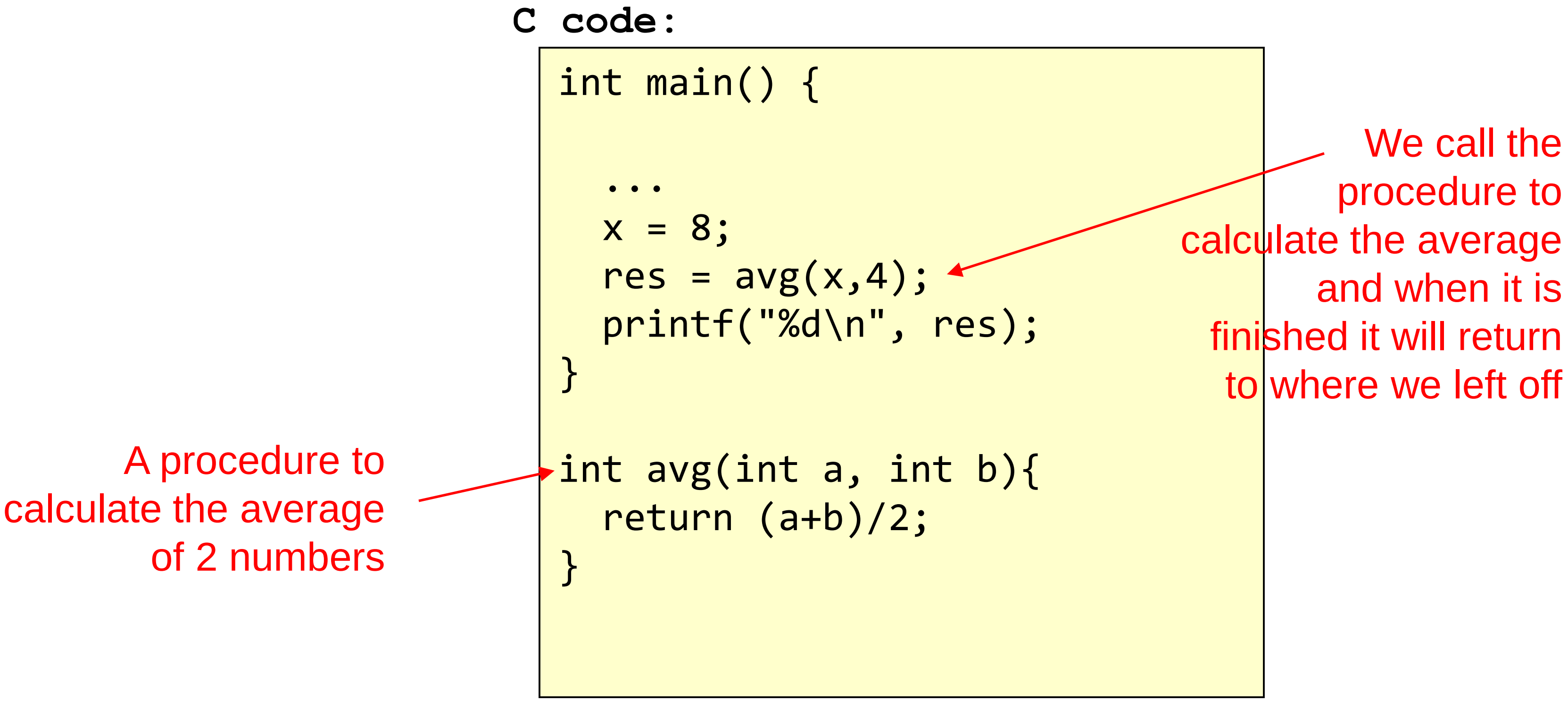
- PC/IP is used to fetch an instruction
  - PC/IP contains the address of the next instruction
  - The value in the PC/IP is placed on the address bus and the memory is told to read
  - The PC/IP is incremented, and the process is repeated for the next instruction



Procedures (Subroutines)

CS:APP 3.7.1

- Procedures (aka subroutines or functions) are reusable sections of code that we can call from some location, execute that procedure, and then **return to where we left off**



C code:

```
int main() {

   ...
   x = 8;
   res = avg(x,4);
   printf("%d\n", res);
}

int avg(int a, int b){
   return (a+b)/2;
}
```

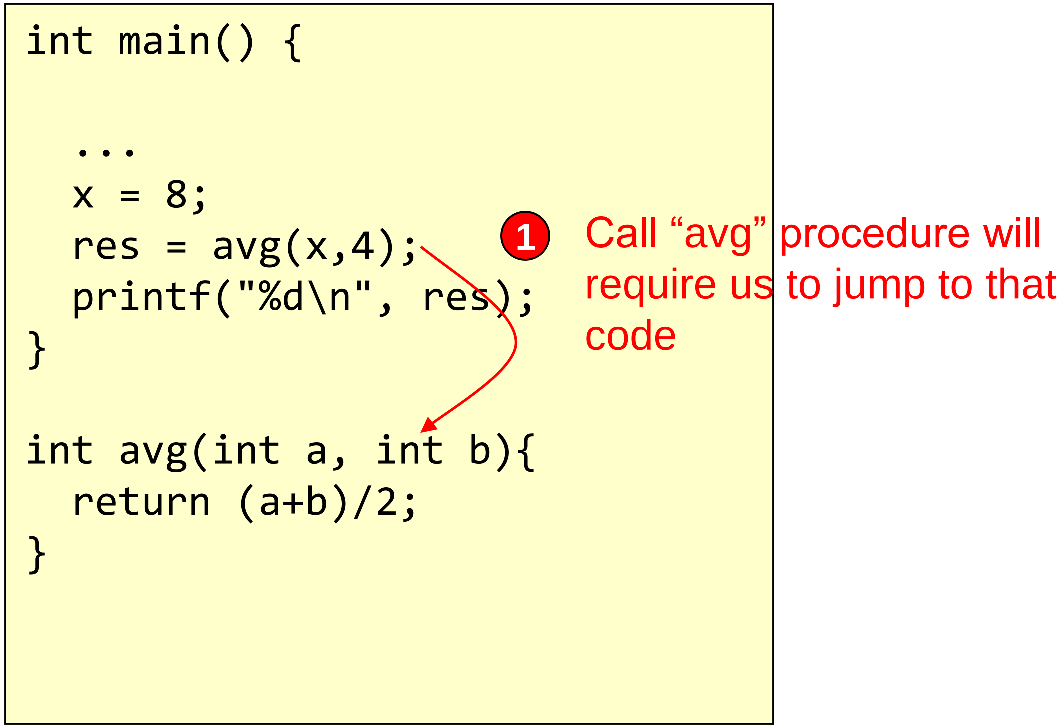
We call the procedure to calculate the average and when it is finished it will return to where we left off

A procedure to calculate the average of 2 numbers

Procedures

- Procedure calls are similar to 'jump' instructions where we go to a new location in the code

C code:



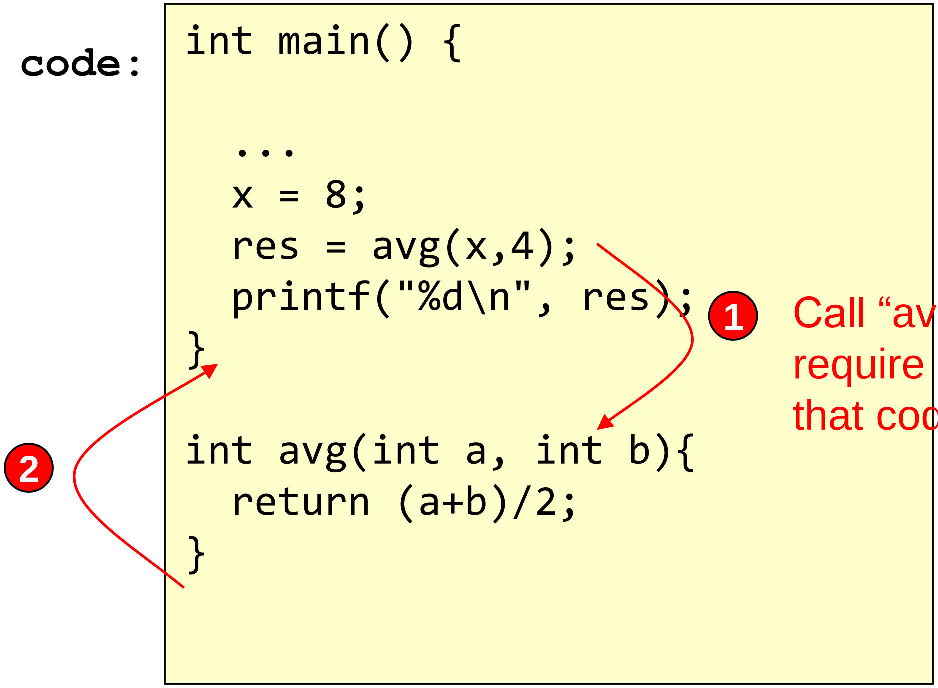
```
int main() {

  ...
  x = 8;
  res = avg(x,4);
  printf("%d\n", res);
}

int avg(int a, int b){
  return (a+b)/2;
}
```

1 Call “avg” procedure will require us to jump to that code

Normal Jumps vs. Procedures

- Difference between normal jumps and procedure calls is that with procedures we have to return to where we left off
- We need to **leave a link** to the return location before we jump to the procedure…if we wait until we get to the function its too late

C code:



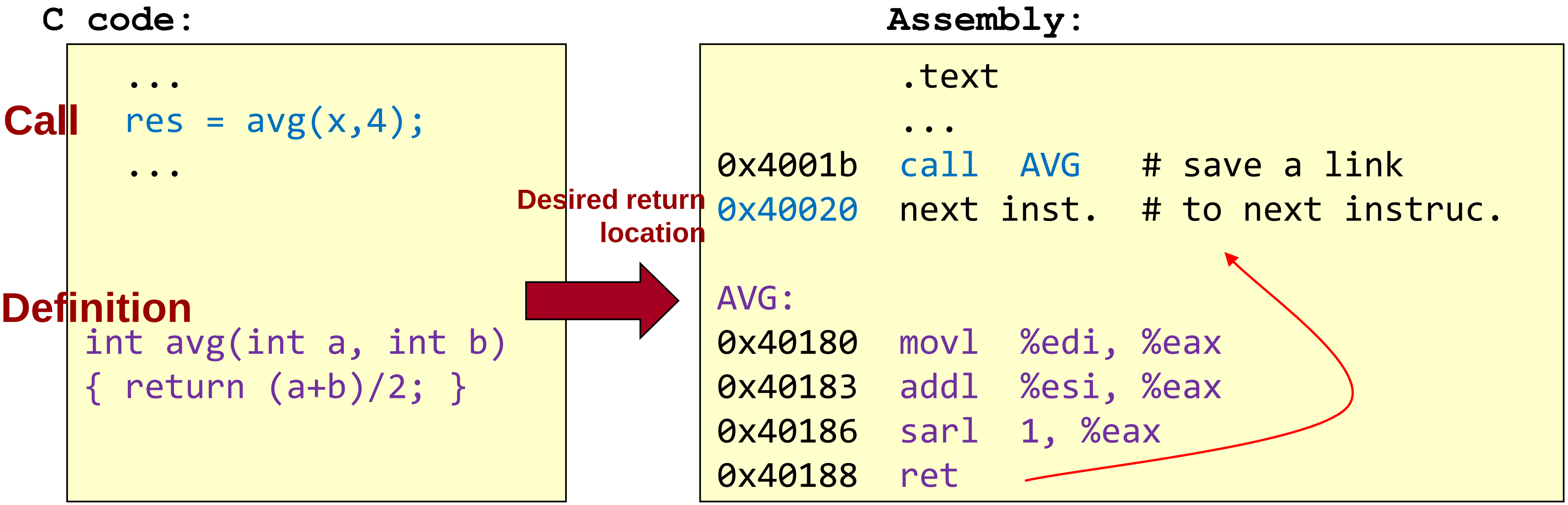
```
int main() {

   ...
   x = 8;
   res = avg(x,4);
   printf("%d\n", res);
}

int avg(int a, int b){
   return (a+b)/2;
}
```

1 Call “avg” procedure will require us to jump to that code

2 After procedure completes, return to the statement in the main code where we left off

Implementing Procedures

- To implement procedures in assembly we need to be able to:
  - Jump to the procedure code, leaving a "return link" (i.e. return address) to know where to return
  - Find the return address and go back to that location



C code:

```
   ...
   res = avg(x,4);
   ...
```

Call

Definition

```
int avg(int a, int b)
{ return (a+b)/2; }
```

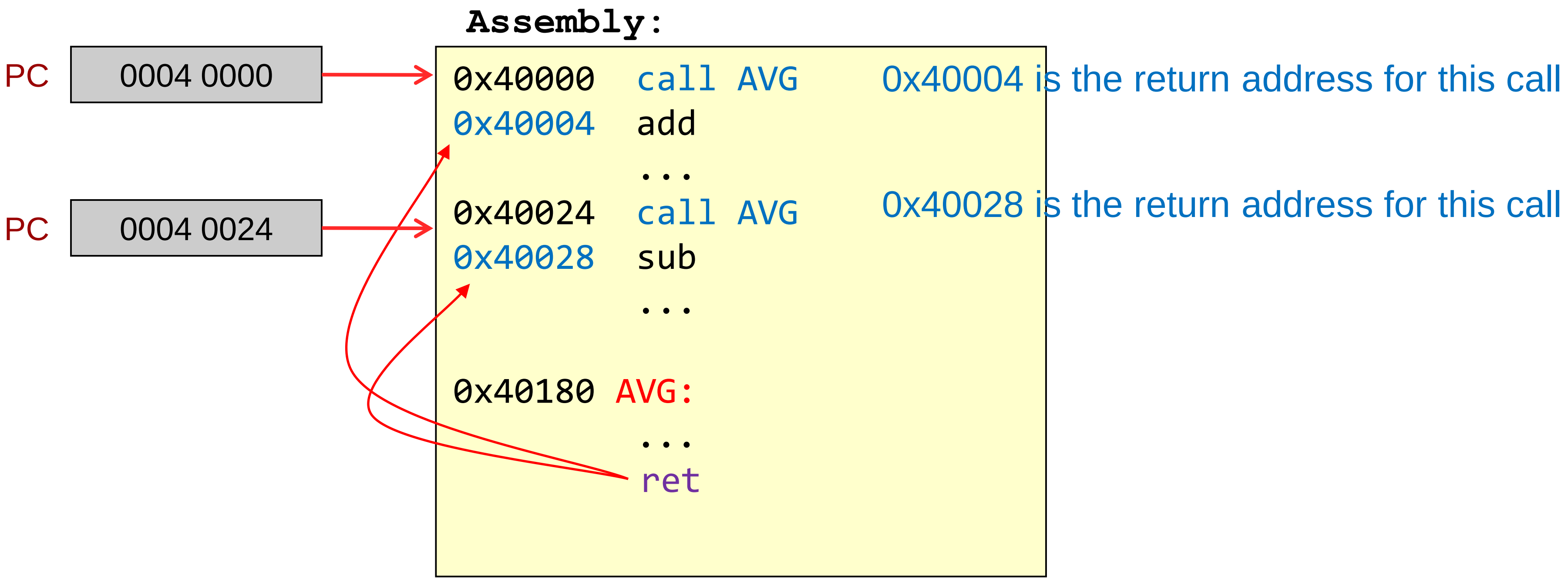
Desired return location

Assembly:

```
              .text
              ...
0x4001b   call   AVG    # save a link
0x40020   next inst.    # to next instruc.

AVG:
0x40180   movl   %edi, %eax
0x40183   addl   %esi, %eax
0x40186   sarl   1, %eax
0x40188   ret
```

Return Addresses

- When calling a procedure, the address to jump to is ALWAYS the same
- The location where a procedure returns will vary
  - Always the address of the instruction after the 'call'



Assembly:

```
0x40000   call AVG
0x40004   add
          ...
0x40024   call AVG
0x40028   sub
          ...

0x40180 AVG:
          ...
          ret
```

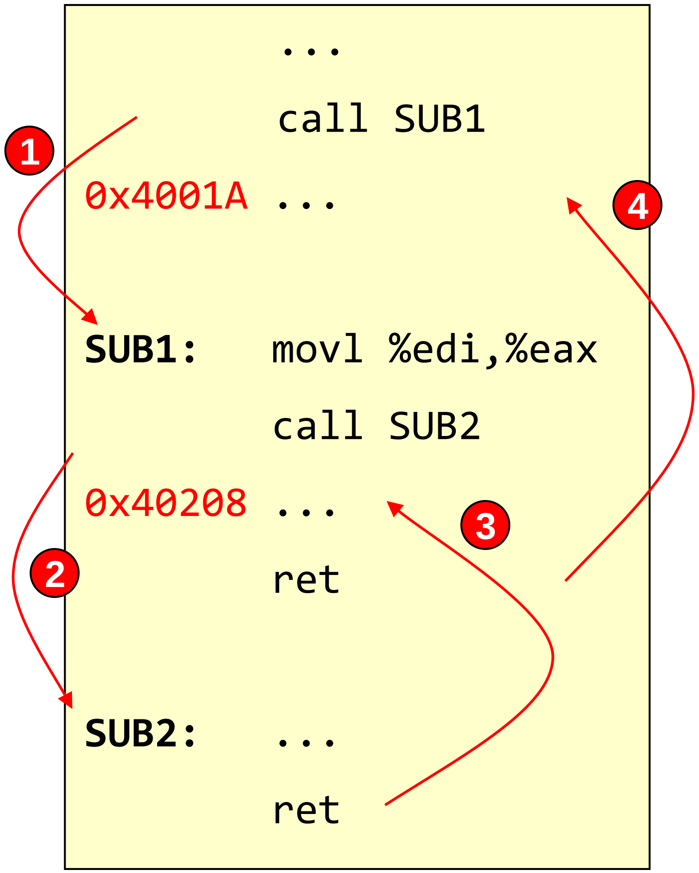
0x40004 is the return address for this call

0x40028 is the return address for this call

Return Addresses

- A further (very common) complication is nested procedure calls
  - One procedure calls another
- Example: Main routine calls SUB1 which calls SUB2
- Must store both return addresses but where?
  - Registers? No…very limited number
  - Memory? Yes…usually enough memory for deep levels of nesting

Assembly:

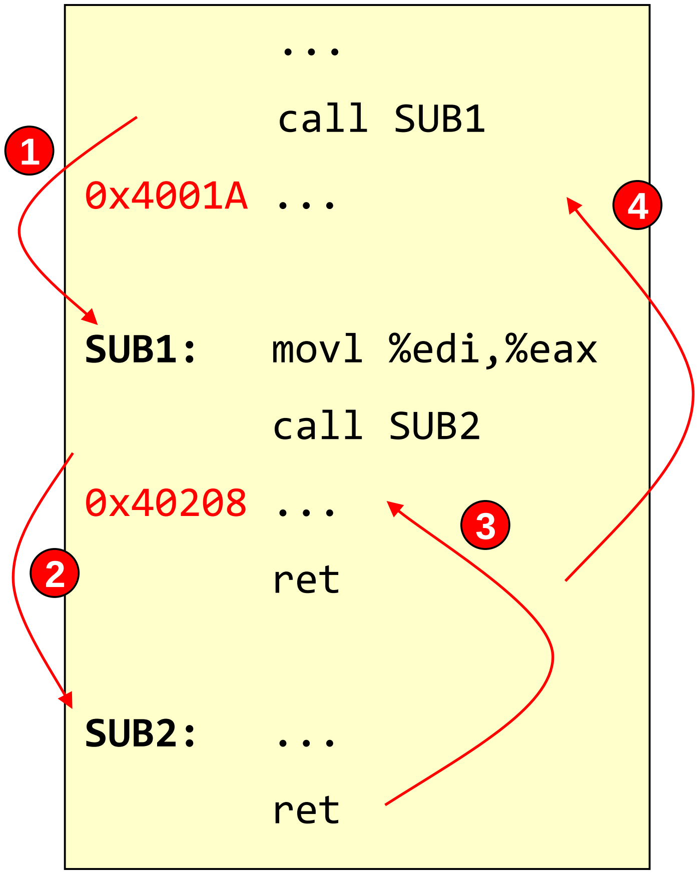


```
                 ...
                 call  SUB1
0x4001A          ...

SUB1:            movl  %edi,%eax
                 call  SUB2
0x40208          ...
                 ret

SUB2:            ...
                 ret
```

Return Addresses and Stacks

- Note: Return addresses will be accessed in reverse order as they are stored
  - 0x40208 is the second RA to be stored but should be the first one used to return
- A stack structure is appropriate!
- The system stack will be a place where we can store
  - Return addresses and other saved register values
  - Local variables of a function
  - Arguments for procedures

Assembly:

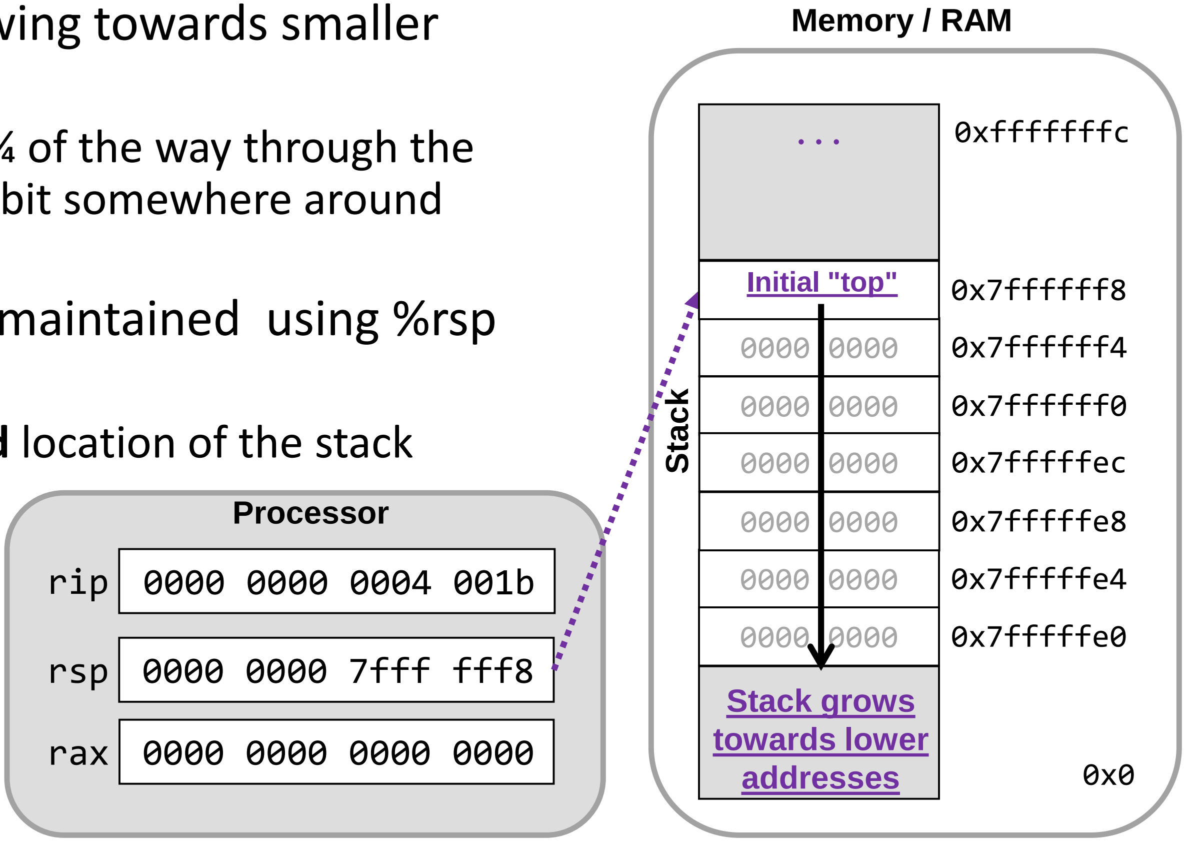


```
           ...
           call SUB1
0x4001A    ...

SUB1:      movl %edi,%eax
           call SUB2
0x40208    ...
           ret

SUB2:      ...
           ret
```

System Stack

- Stack is a data structure where data is accessed in reverse order as it is stored (a.k.a. LIFO = Last-in First-out)
- Use a stack to store the return addresses and other data
- System stack defined as growing towards smaller addresses
  - Usually starts around ½ to ¾ of the way through the address space (i.e. for a 32-bit somewhere around 0x7ffff… or 0xbffff…)
- Top of stack is accessed and maintained using %rsp (stack pointer) register
  - %rsp points at top **occupied** location of the stack

Stack Pointer
Always points to top occupied element of the stack



Processor

rip	0000 0000 0004 001b
rsp	0000 0000 7fff fff8
rax	0000 0000 0000 0000

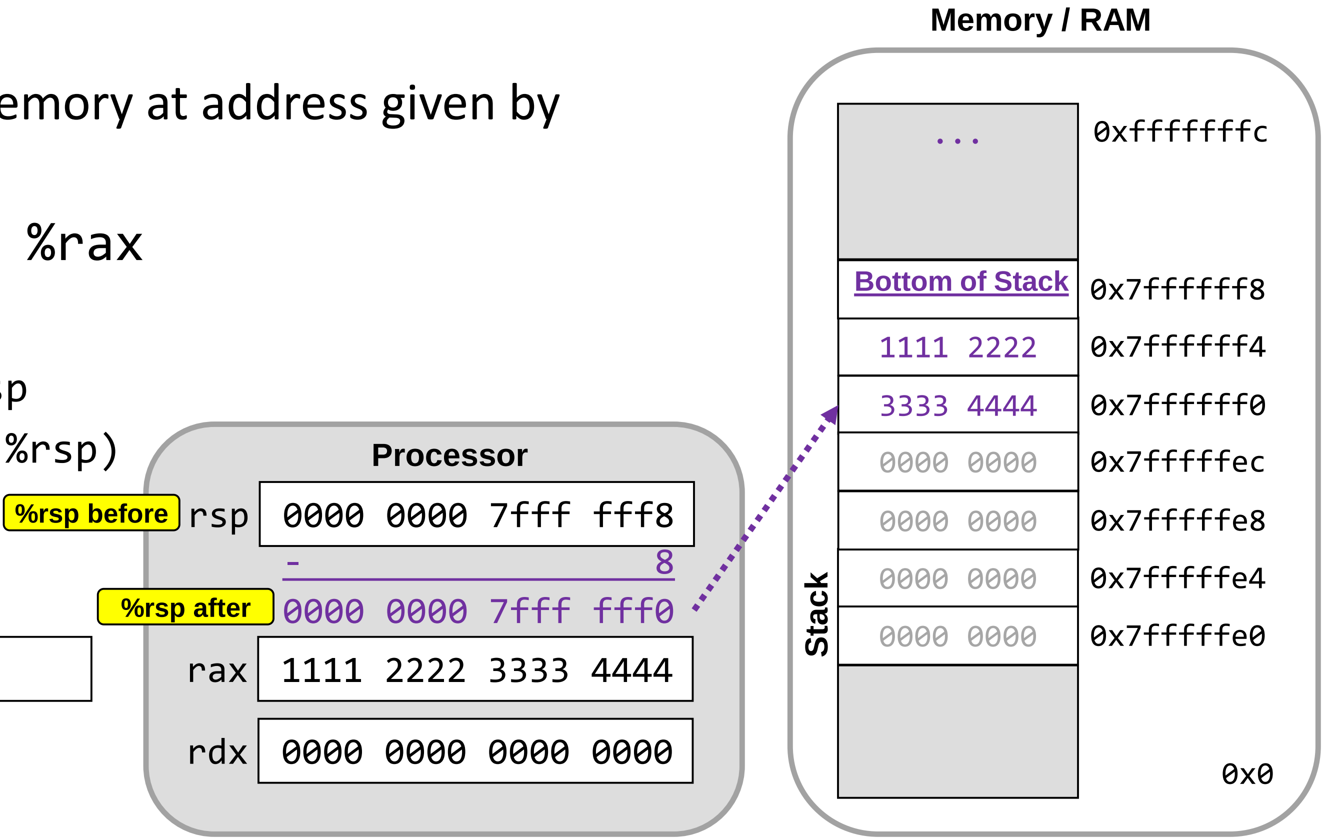
Memory / RAM

. . .	0xfffffffc
Initial "top"	0x7ffffff8
0000 0000	0x7ffffff4
0000 0000	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
Stack grows towards lower addresses	0x0

Push Operation and Instruction

- Push operation adds data to system stack
- Format: `push[w,q,l] %reg`
  - Decrements %rsp by 2, 4, or 8 (depending on [w,q,l]
  - Write %reg to memory at address given by %rsp
  - Example: `pushq %rax`
  - Equivalent:
    - `subq $8, %rsp`
    - `movq %rax, (%rsp)`

`pushq %rax`



Processor

%rsp before rsp	0000 0000 7fff fff8
	- 8
%rsp after	0000 0000 7fff fff0
rax	1111 2222 3333 4444
rdx	0000 0000 0000 0000

Memory / RAM

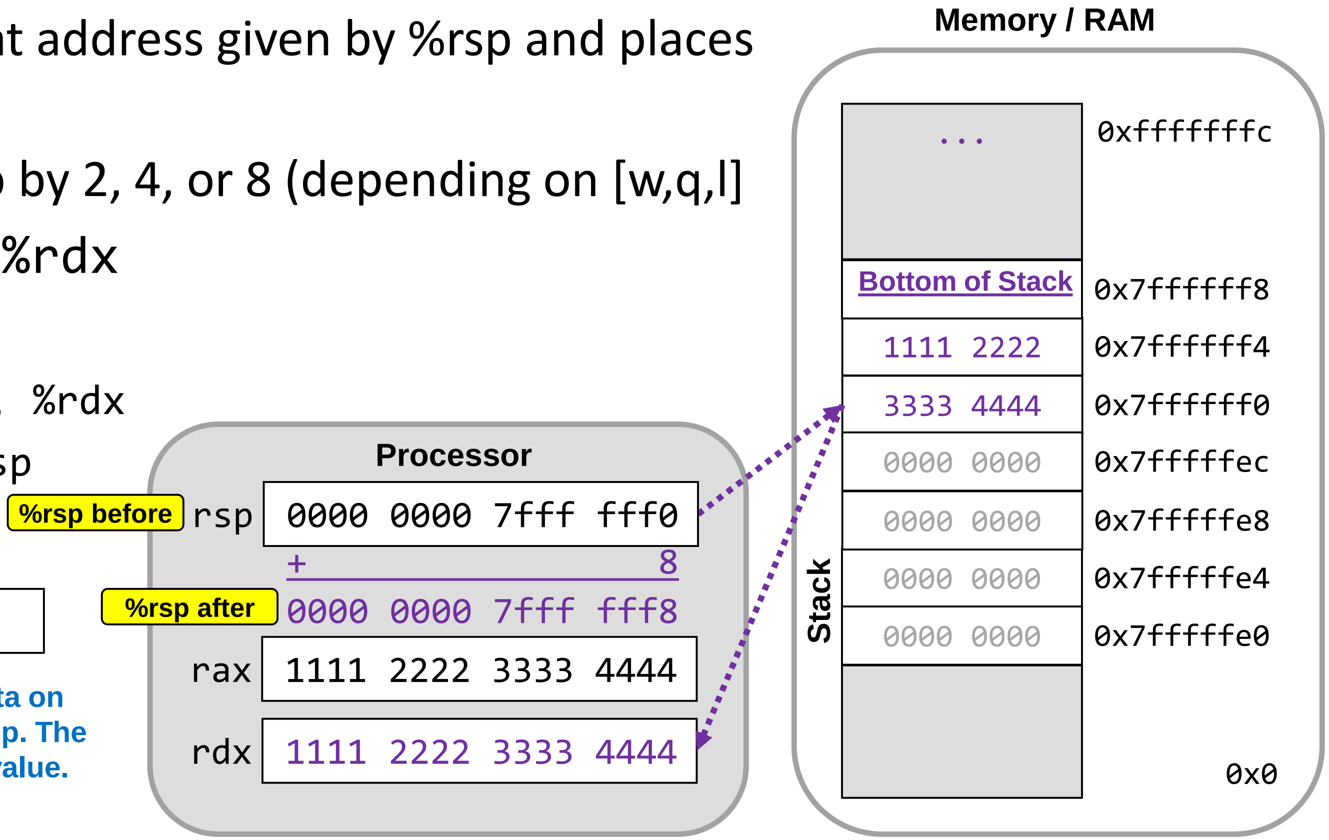
Stack	
...	0xfffffffc
Bottom of Stack	0x7ffffff8
1111 2222	0x7ffffff4
3333 4444	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
	0x0

Pop Operation and Instruction

- Pop operation removes data from system stack
- Format: `pop[w,q,l] %reg`
  - Reads memory at address given by %rsp and places value into %reg
  - Increments %rsp by 2, 4, or 8 (depending on [w,q,l])
  - Example: `popq %rdx`
  - Equivalent:
    - `movq (%rsp), %rdx`
    - `addq $8, %rsp`

`popq %rdx`

Note: pop does not erase the data on the stack, it simply moves the %rsp. The next push will overwrite the old value.



Processor

%rsp before rsp	0000 0000 7fff fff0
	+ 8
%rsp after	0000 0000 7fff fff8
rax	1111 2222 3333 4444
rdx	1111 2222 3333 4444

Memory / RAM

Stack	Address
...	0xfffffffc
Bottom of Stack	0x7ffffff8
1111 2222	0x7ffffff4
3333 4444	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
	0x0

Jumping to a Procedure

CS:APP 3.7.2

- Format:
  - `call label`
  - `call *operand [e.g. call (%rax)]`
- Operations:
  - Pushes the address of next instruction (i.e. return address (RA)) onto the stack
    - Implicitly performs `subq $8, (%rsp)` and `movq %rip, (%rsp)`
  - Updates the PC to go to the start of the desired procedure [i.e. PC = addr]
    - addr is the address you want to branch to (*Usually specified as a label*)

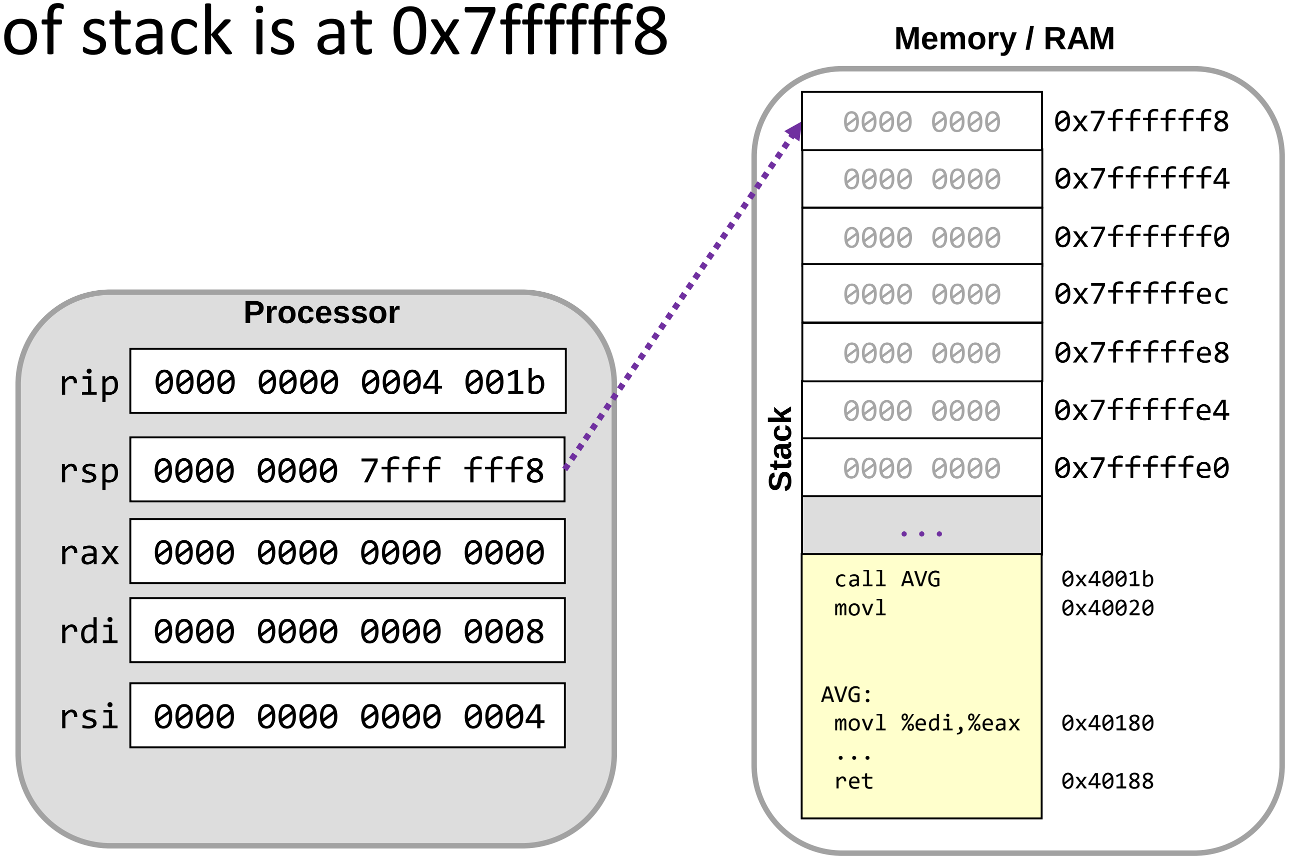
Returning From a Procedure

- Format:
  - `ret`
- Operations:
  - Pops the return address from the stack into %rip [i.e. PC = return-address]
  - Implicitly performs `movq (%rsp), %rip` and `addq $8, %rsp`

Procedure Call Sequence 1a

- Initial conditions
  - About to execute the 'call' instruction
  - Current top of stack is at 0x7ffffff8

```
 ...
 call AVG
 movl %eax,(%rbp)
 ...

AVG:
 movl %edi,%eax
 ...
 ret
```



Processor

rip	0000 0000 0004 001b
rsp	0000 0000 7fff fff8
rax	0000 0000 0000 0000
rdi	0000 0000 0000 0008
rsi	0000 0000 0000 0004

Memory / RAM

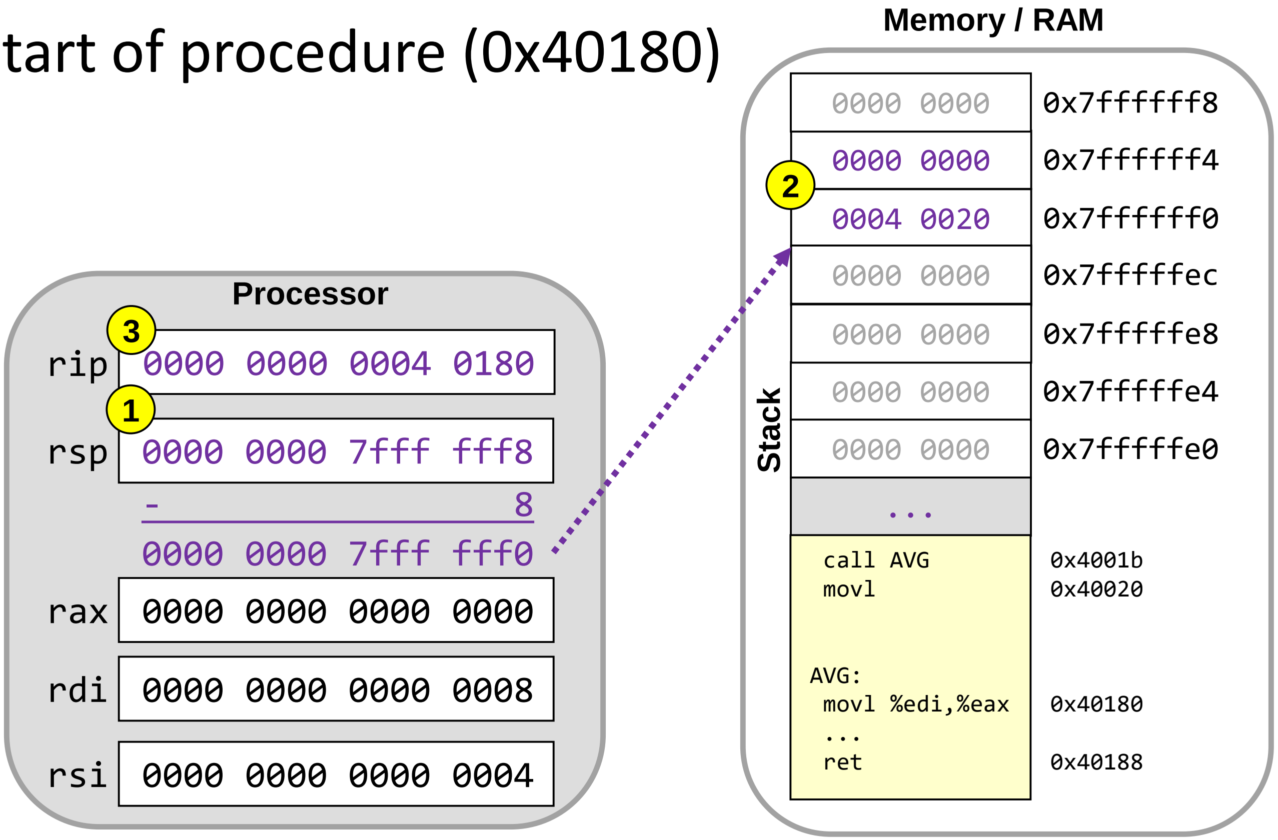
Stack

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0000 0000	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
...	
call AVG	0x4001b
movl	0x40020
AVG:	
movl %edi,%eax	0x40180
...	
ret	0x40188

Procedure Call Sequence 1b

- call Operation (i.e. push return address) & jump
  - Decrement stack pointer ($rsp) and push RA (0x40020) onto stack (as 64-bit address)
  - Update PC to start of procedure (0x40180)

```
  ...
  call AVG
  movl %eax,(%rbp)
  ...

AVG:
  movl %edi,%eax
  ...
  ret
```



Processor

3 rip 0000 0000 0004 0180

1 rsp 0000 0000 7fff fff8

- 8

0000 0000 7fff fff0

rax 0000 0000 0000 0000

rdi 0000 0000 0000 0008

rsi 0000 0000 0000 0004

Memory / RAM

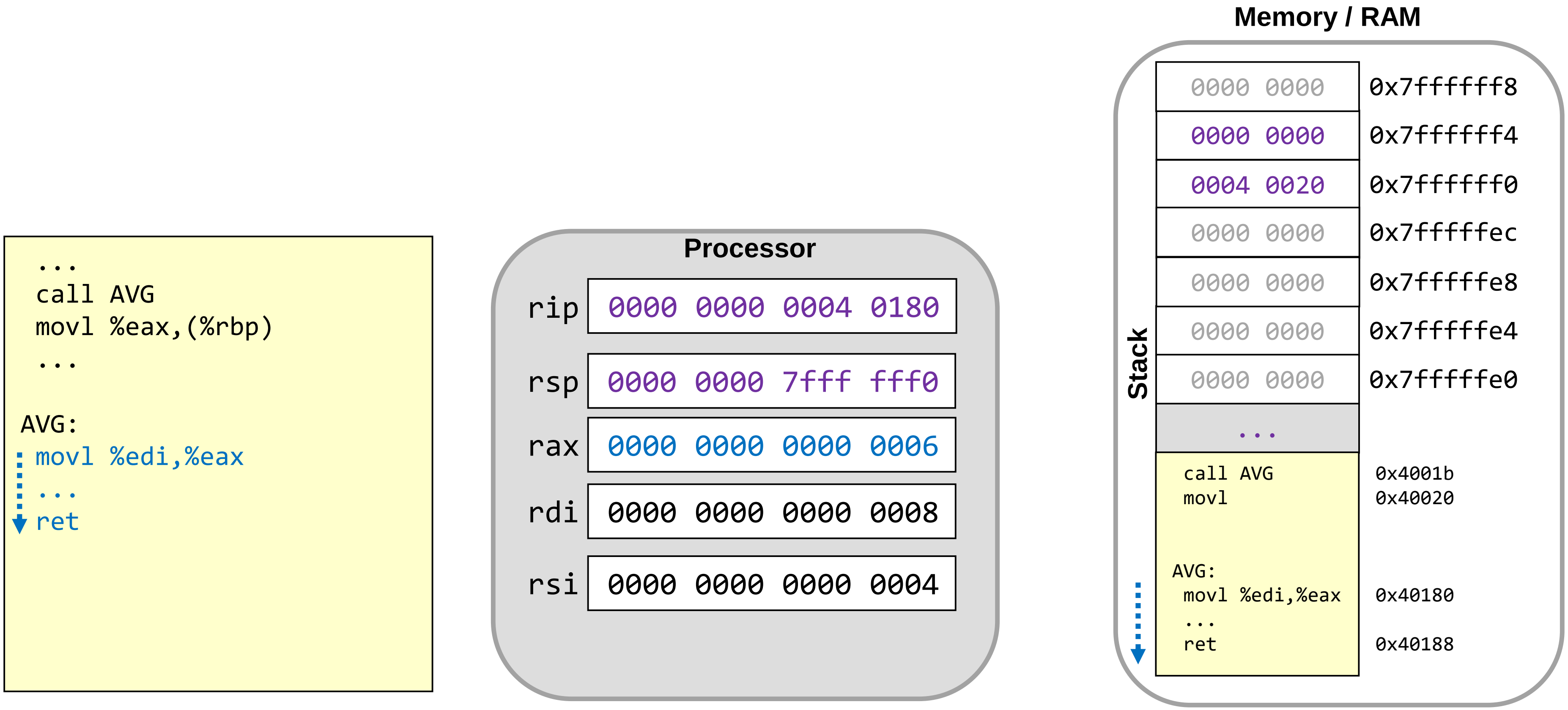
Stack

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 0020	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
...	
call AVG	0x4001b
movl	0x40020
AVG:	
movl %edi,%eax	0x40180
...	
ret	0x40188

2

Procedure Call Sequence 1c

- Execute the code for the procedure
- Return value should be in %rax/%eax



```
  ...
  call AVG
  movl %eax,(%rbp)
  ...

AVG:
  movl %edi,%eax
  ...
  ret
```

Processor

rip	0000 0000 0004 0180
rsp	0000 0000 7fff fff0
rax	0000 0000 0000 0006
rdi	0000 0000 0000 0008
rsi	0000 0000 0000 0004

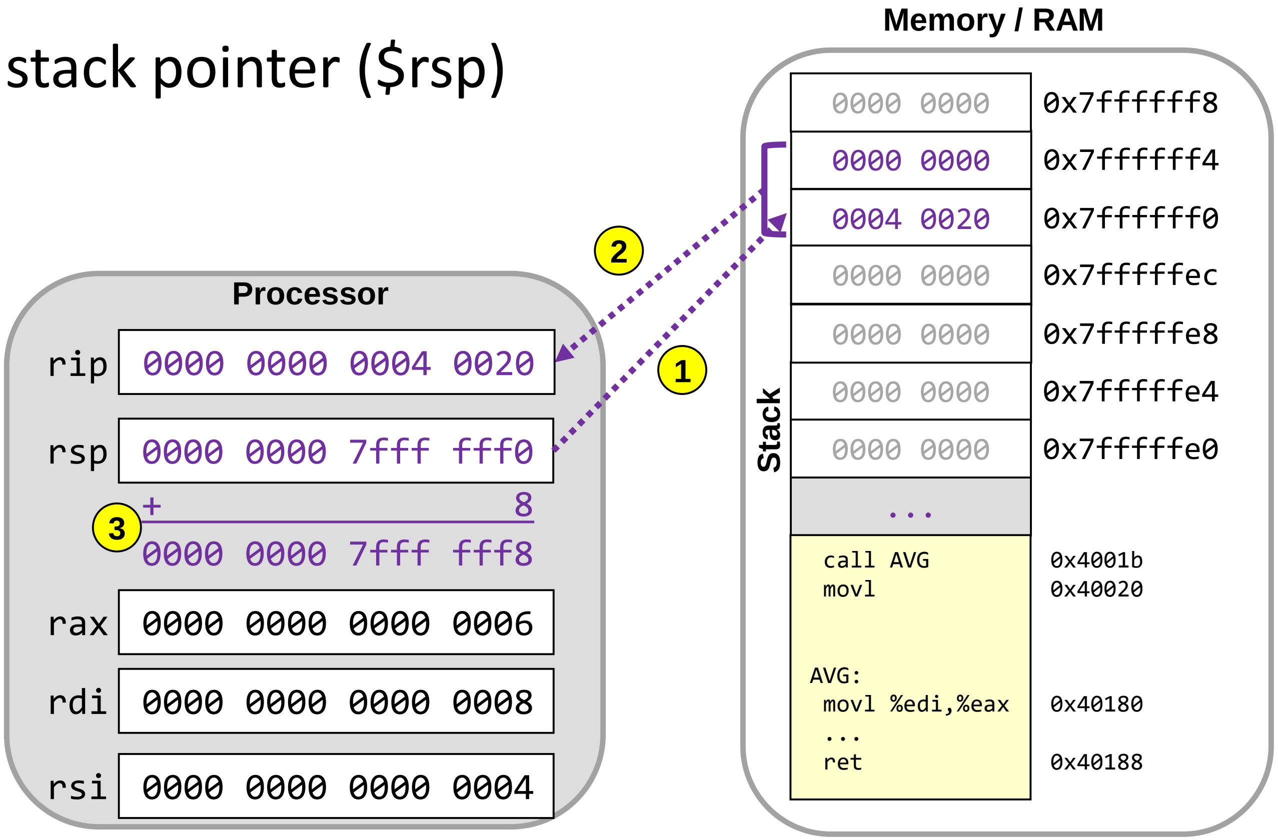
Memory / RAM

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 0020	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
...	
call AVG	0x4001b
movl	0x40020
AVG:	
movl %edi,%eax	0x40180
...	
ret	0x40188

Procedure Call Sequence 1d

- ret Operation (i.e. pop return address)
  - Retrieve RA (0x40020) from stack
  - Put it in the PC
  - Increment the stack pointer ($rsp)

```
  ...
  call AVG
  movl %eax,(%rbp)
  ...

AVG:
  movl %edi,%eax
  ...
  ret
```



Processor

rip 0000 0000 0004 0020

rsp 0000 0000 7fff fff0

+ 8

0000 0000 7fff fff8

rax 0000 0000 0000 0006

rdi 0000 0000 0000 0008

rsi 0000 0000 0000 0004

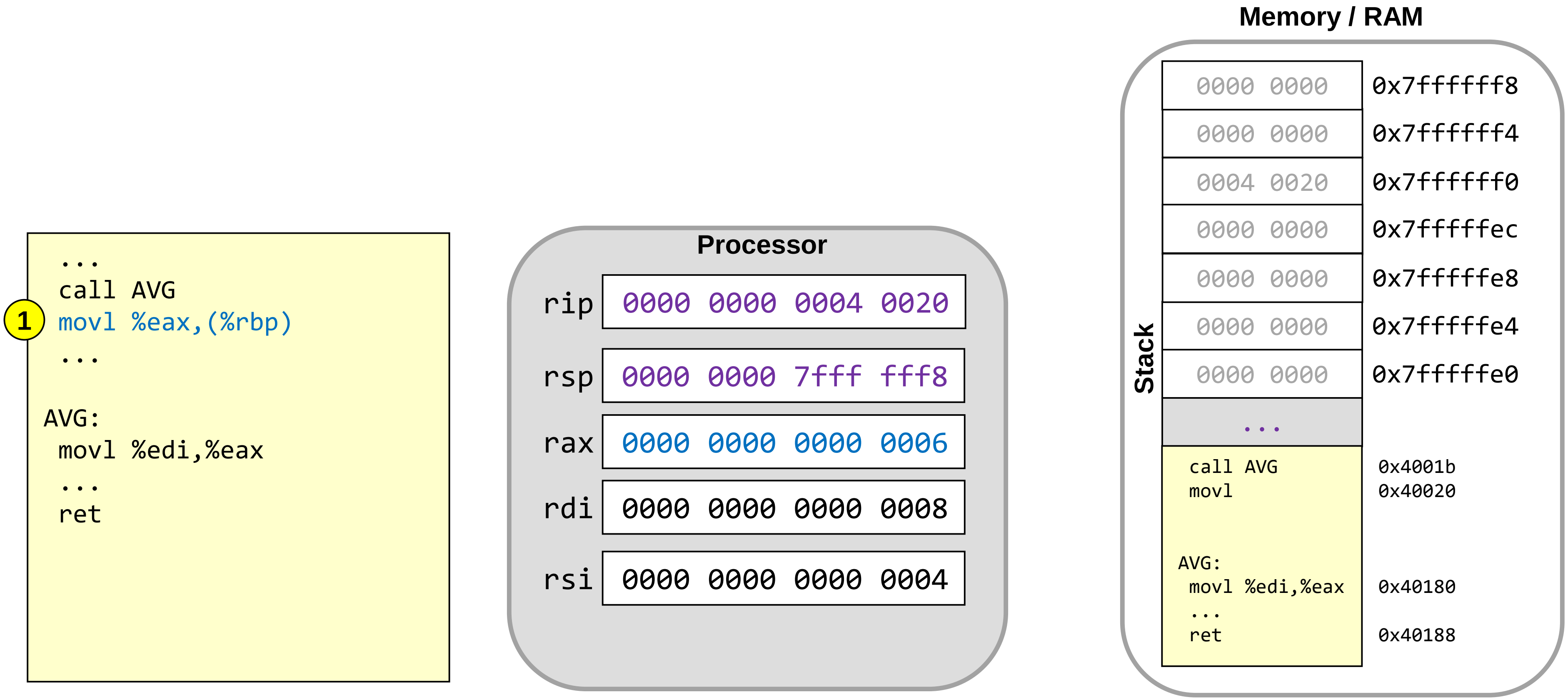
Memory / RAM

Stack

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 0020	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
...	
call AVG	0x4001b
movl	0x40020
AVG:	
movl %edi,%eax	0x40180
...	
ret	0x40188

Procedure Call Sequence 1e

- Execution resumes after the procedure call



```
  ...
  call AVG
1 movl %eax,(%rbp)
  ...

AVG:
  movl %edi,%eax
  ...
  ret
```

Processor

rip	0000 0000 0004 0020
rsp	0000 0000 7fff fff8
rax	0000 0000 0000 0006
rdi	0000 0000 0000 0008
rsi	0000 0000 0000 0004

Memory / RAM

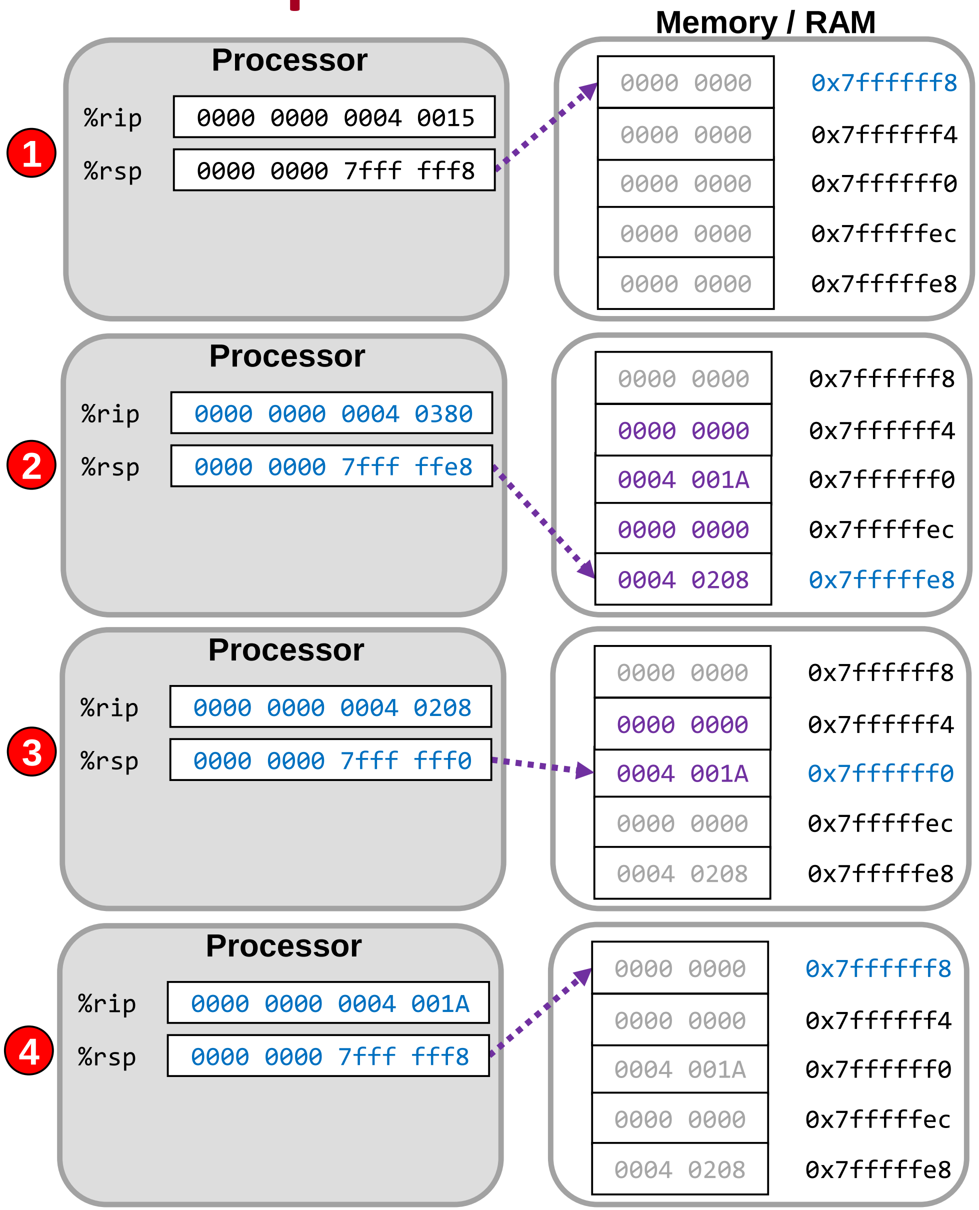
Stack

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 0020	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8
0000 0000	0x7fffffe4
0000 0000	0x7fffffe0
...	
call AVG	0x4001b
movl	0x40020
AVG:	
movl %edi,%eax	0x40180
...	
ret	0x40188

Procedure Call Sequence 2

- Show the values of the stack, %rsp, and %rip at the various timestamps for the following code

```
                ...
1  0x40015  call SUB1
4  0x4001A  ...

   0x40200
   SUB1:    movl %edi,%eax
            call SUB2
3  0x40208  ...
            ret

   0x40380
2  SUB2:    ...
            ret
```



Memory / RAM

1 Processor

%rip 0000 0000 0004 0015

%rsp 0000 0000 7fff fff8

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0000 0000	0x7ffffff0
0000 0000	0x7fffffec
0000 0000	0x7fffffe8

2 Processor

%rip 0000 0000 0004 0380

%rsp 0000 0000 7fff ffe8

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 001A	0x7ffffff0
0000 0000	0x7fffffec
0004 0208	0x7fffffe8

3 Processor

%rip 0000 0000 0004 0208

%rsp 0000 0000 7fff fff0

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 001A	0x7ffffff0
0000 0000	0x7fffffec
0004 0208	0x7fffffe8

4 Processor

%rip 0000 0000 0004 001A

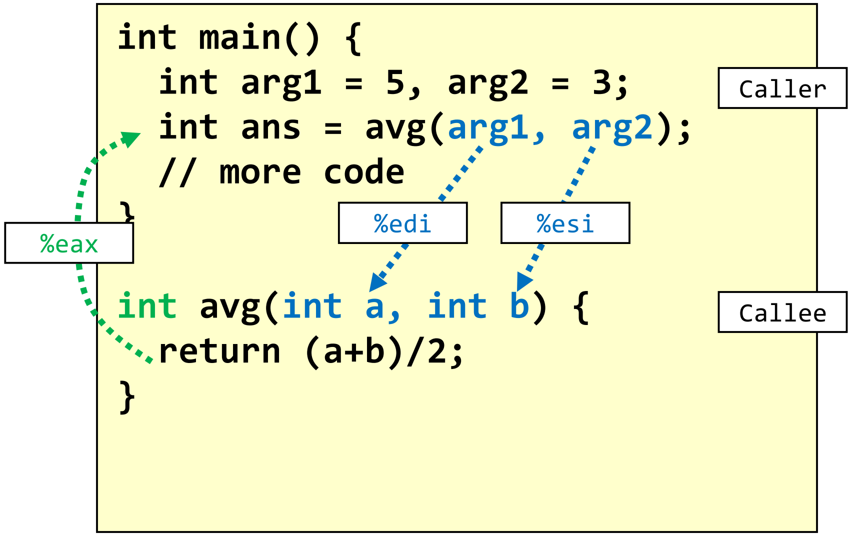
%rsp 0000 0000 7fff fff8

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0004 001A	0x7ffffff0
0000 0000	0x7fffffec
0004 0208	0x7fffffe8

Arguments and Return Values

CS:APP 3.7.3

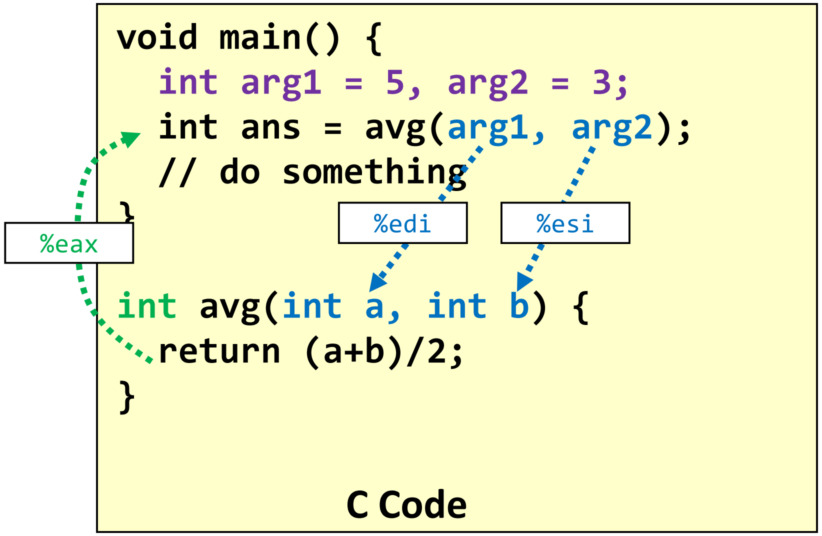
- Most procedure calls pass arguments/parameters to the procedure and it often produces return values
- To implement this, there must be locations agreed upon by caller and callee for where this information will be found
- x86-64 convention is to use certain registers for this task (**see table**)



```
int main() {
  int arg1 = 5, arg2 = 3;
  int ans = avg(arg1, arg2);
  // more code
}

int avg(int a, int b) {
  return (a+b)/2;
}
```

1st Argument	%rdi
2nd Argument	%rsi
3rd Argument	%rdx
4th Argument	%rcx
5th Argument	%r8
6th Argument	%r9
Additional arguments	Pass on stack
Return value	%rax

Passing Arguments and Return Values

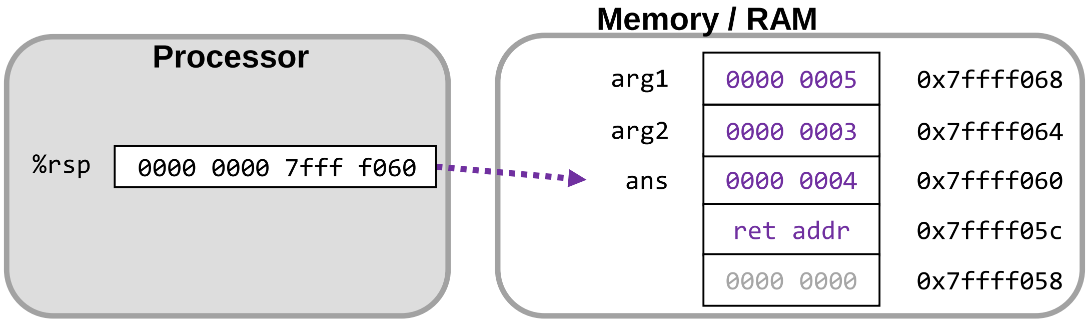


```
void main() {
  int arg1 = 5, arg2 = 3;
  int ans = avg(arg1, arg2);
  // do something
}

int avg(int a, int b) {
  return (a+b)/2;
}
```

C Code

```
      .text
      movl  $5, 8(%rsp)
      movl  $3, 4(%rbp)
      movl  8(%rsp), %edi
      movl  4(%rsp), %esi
      call  AVG
      movl  %eax, (%rsp)

AVG:
      movl  %edi, %eax
      addl  %esi, %eax
      sarl  1, %eax
      ret
```

Assembly



Processor

%rsp 0000 0000 7fff f060

Memory / RAM

arg1	0000 0005	0x7ffff068
arg2	0000 0003	0x7ffff064
ans	0000 0004	0x7ffff060
	ret addr	0x7ffff05c
	0000 0000	0x7ffff058

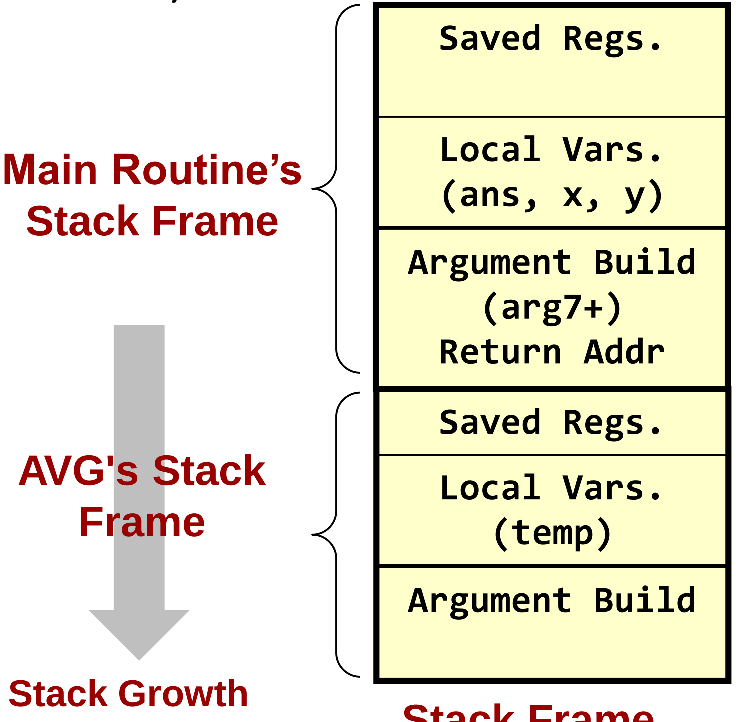
Compiler Handling of Procedures

- When coding in an high level language & using a compiler, certain conventions are followed that may lead to heavier usage of the stack
  - We have to be careful not to **overwrite** registers that have useful data
- High level languages (HLL) use the stack:
  - to save register values including the return address
  - for storage of local variables declared in the procedure
  - to pass arguments to a procedure
- Compilers usually put data on the stack in a certain order, which we call a stack frame

Stack Frames

- Frame = ***Def:*** All data on stack belonging to a procedure / function
  - Space for saved registers
  - Space for local variables (those declared in a function)
  - Space for arguments

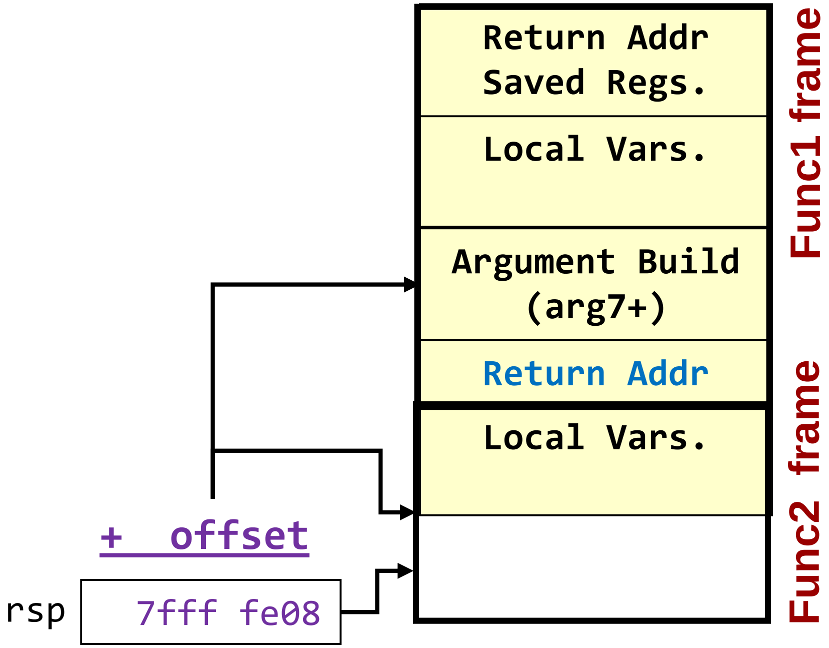
```
void main() {
  int ans, x, y;
  ans = avg(x, y);
  ...
}
int avg(int a, int b) {
  int temp=1;  // local vars
  ...
}
```



Stack Frame Organization

Accessing Values on the Stack

- Stack pointer (%rsp) is usually used to access only the top value on the stack
- To access arguments and local variables, we need to access values buried in the stack
  - We can simply use an offset from %rsp [e.g. 8(%rsp)]



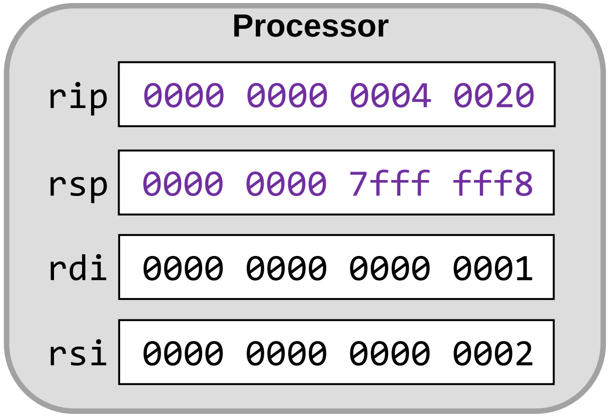
To access parameters we could try to use some displacement [i.e. d($sp)]

Many Arguments Examples

- Examine the following C code and corresponding assembly
- Assume initially %rsp = 0x7ffffff8
- Note how the 7th and 8th arguments are passed via the stack

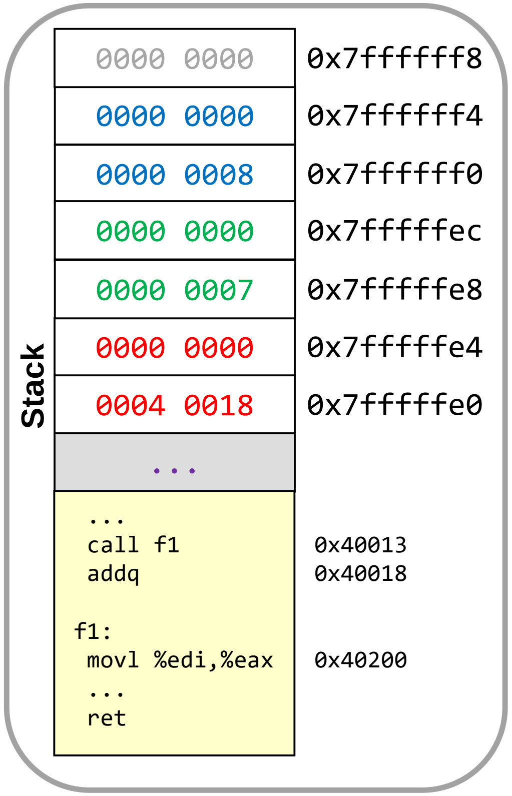
```
caller:
    pushq   $8
    pushq   $7
    movl    $6, %r9d
    movl    $5, %r8d
    movl    $4, %ecx
    movl    $3, %edx
    movl    $2, %esi
    movl    $1, %edi
    call    f1
    addq    $16, %rsp
    ret
f1:   # 0x40200
1   addl    %edi, %esi
    addl    %esi, %edx
    addl    %edx, %ecx
    addl    %ecx, %r8d
    addl    %r8d, %r9d
    movl    %r9d, %eax
    addl    8(%rsp), %eax
    addl    16(%rsp), %eax
    ret
```

```
int caller()
{
  int sum = f1(1, 2, 3, 4, 5, 6, 7, 8);
  return sum;
}

int f1(int a1, int a2, int a3, int a4,
       int a5, int a6, int a7, int a8)
{
  return a1+a2+a3+a4+a5+a6+a7+a8;
}
```


Processor

rip	0000 0000 0004 0020
rsp	0000 0000 7fff fff8
rdi	0000 0000 0000 0001
rsi	0000 0000 0000 0002

Memory / RAM


Stack

0000 0000	0x7ffffff8
0000 0000	0x7ffffff4
0000 0008	0x7ffffff0
0000 0000	0x7fffffec
0000 0007	0x7fffffe8
0000 0000	0x7fffffe4
0004 0018	0x7fffffe0
...	

```
 ...
 call f1          0x40013
 addq             0x40018

f1:
 movl %edi,%eax   0x40200
 ...
 ret
```

Local Variables

CS:APP 3.7.4

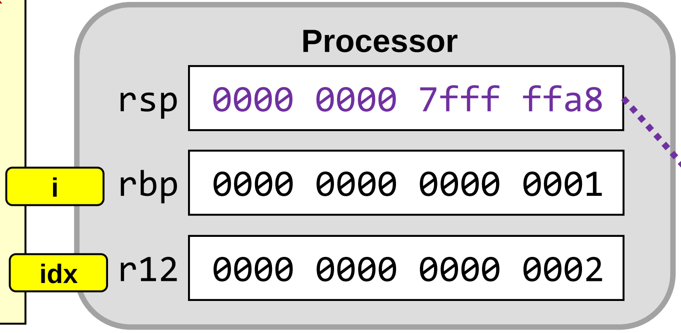
- For simple integer/pointers the compiler can optimize code by using a register rather than allocating the variable on the stack
- Local variables need to be allocated on the stack if:
  - No free registers (too many locals)
  - The & operator is used and thus we need to be able to generate an address
  - Arrays or structs are used

Local Variables Example

```
f2:  ⓪ pushq   %r12
        pushq   %rbp
        pushq   %rbx
     ① subq    $0x30, %rsp
        movl    %edi, %r12d
        movq    %fs:0x28, %rax
        movq    %rax, 0x28(%rsp)
        xorl    %eax, %eax
     ② leaq    0xc(%rsp), %rdi
        call    getInt
     ③ movl    $0, %ebx
        jmp     .L4
.L6:    movslq  %ebx, %rbp
     ⑤ leaq    0x10(%rsp,%rbp,4), %rdi
        call    getInt
     ⑥ movl    0x10(%rsp,%rbp,4), %eax
        cmpl    0xc(%rsp), %eax
        jge     .L5
     ⑦ movl    %eax, 0xc(%rsp)
.L5:    addl    $1, %ebx
.L4: ④ cmpl    $3, %ebx
        jle     .L6
     ⑧ movslq  %r12d, %r12
        movl    0xc(%rsp), %eax
        addl    0x10(%rsp,%r12,4), %eax
     ⑨ movq    0x28(%rsp), %rdx
        xorq    %fs:0x28, %rdx
        je      .L7
        call    __stack_chk_fail
.L7:    addq    $0x30, %rsp
        popq    %rbx
        popq    %rbp
        popq    %r12
        ret
```

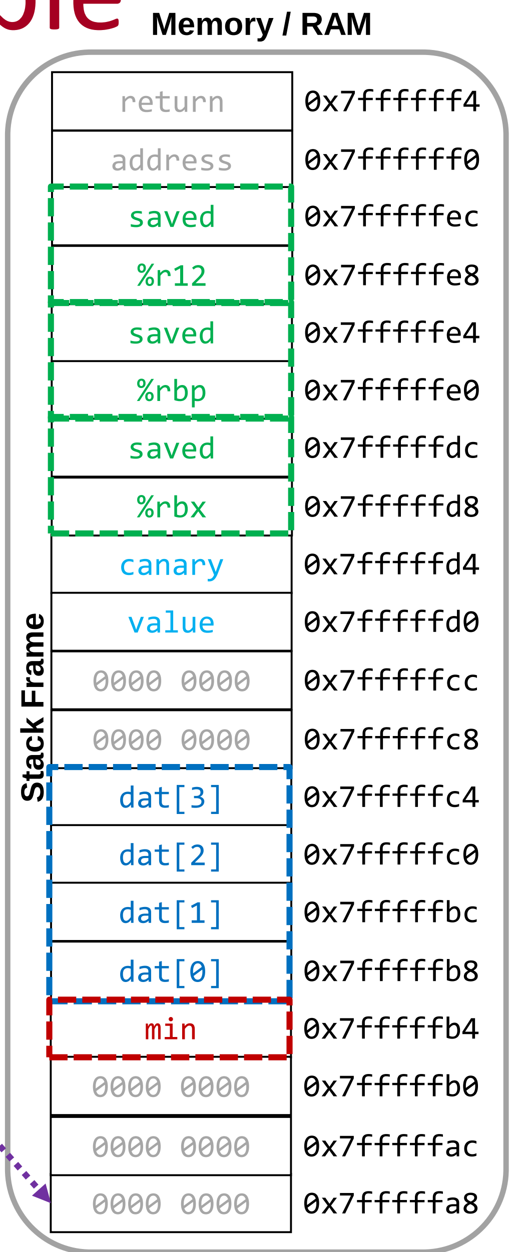
```
 void getInt(int* ptr);
 int f2(int idx)
⓪{
 ①int dat[4], min;
 ②getInt(&min); ④
 ③for(int i=0; i < 4; i++){
 ⑤  getInt(&dat[i]); ⑦
 ⑥  if(dat[i] < min) min = dat[i];
    }
 ⑧return dat[idx] + min;
⑨}
```

- %rdi = %r12 = idx
- %rbp = %ebx = int i
- Notice %rdi must be reused from idx to the arguments for getInt(), thus the use of %r12 to hold idx



Processor

	rsp	0000 0000 7fff ffa8
i	rbp	0000 0000 0000 0001
idx	r12	0000 0000 0000 0002



Memory / RAM

Stack Frame

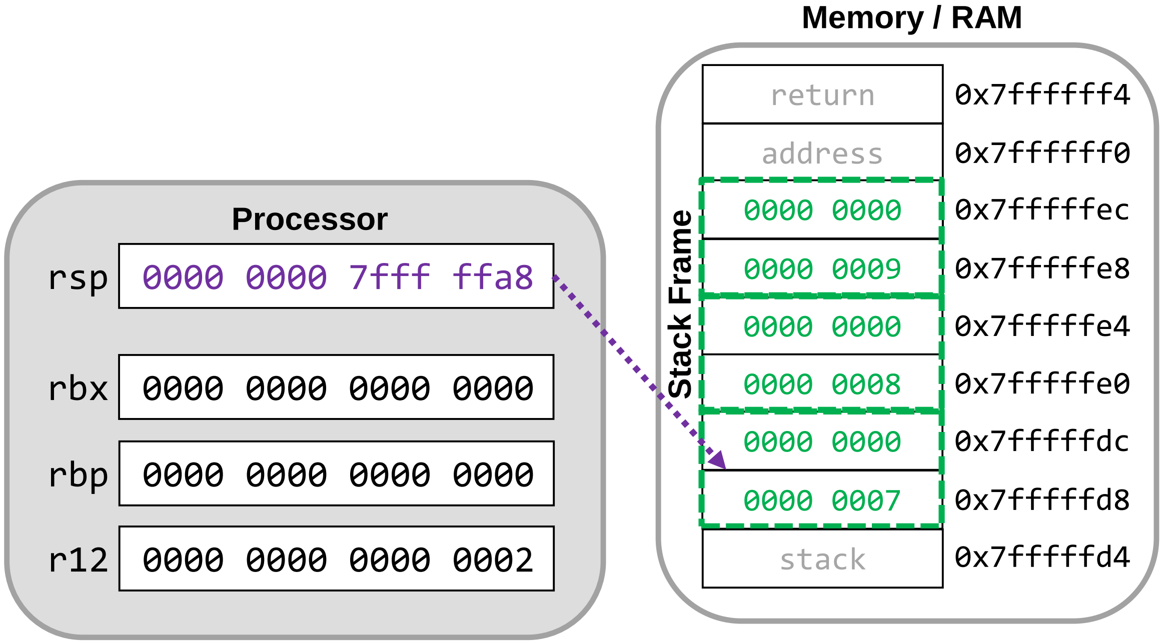
return	0x7ffffff4
address	0x7ffffff0
saved	0x7fffffec
%r12	0x7fffffe8
saved	0x7fffffe4
%rbp	0x7fffffe0
saved	0x7fffffdc
%rbx	0x7fffffd8
canary	0x7fffffd4
value	0x7fffffd0
0000 0000	0x7fffffcc
0000 0000	0x7fffffc8
dat[3]	0x7fffffc4
dat[2]	0x7fffffc0
dat[1]	0x7fffffbc
dat[0]	0x7fffffb8
min	0x7fffffb4
0000 0000	0x7fffffb0
0000 0000	0x7fffffac
0000 0000	0x7fffffa8

Saved Register Problem

CS:APP 3.7.5

- Procedures are generally compiled separately
- The compiler will use registers for some temporaries and local variables
- What could go wrong?

```
f2:     pushq   %r12  ┐
        pushq   %rbp  ├  Why are these
        pushq   %rbx  ┘    needed?
        subq    $0x30, %rsp
        movl    %edi, %r12d
        ...
        movl    $0, %ebx
        ...
        movslq  %ebx, %rbp
        leaq    0x10(%rsp,%rbp,4), %rdi
        ...
        popq    %rbx
        popq    %rbp
        popq    %r12
        ret

f1:     ...
        movl    $7, %ebx
        movl    $8, %ebp
        movq    $9, %r12
        movl    $2, %rdi
        call    f2
        ...
        add     %ebx, %ebp
        subq    $1, %r12
        ...
```



Processor

rsp	0000 0000 7fff ffa8
rbx	0000 0000 0000 0000
rbp	0000 0000 0000 0000
r12	0000 0000 0000 0002

Memory / RAM

return	0x7ffffff4
address	0x7ffffff0
0000 0000	0x7fffffec
0000 0009	0x7fffffe8
0000 0000	0x7fffffe4
0000 0008	0x7fffffe0
0000 0000	0x7fffffdc
0000 0007	0x7fffffd8
stack	0x7fffffd4

Saved Register Problem

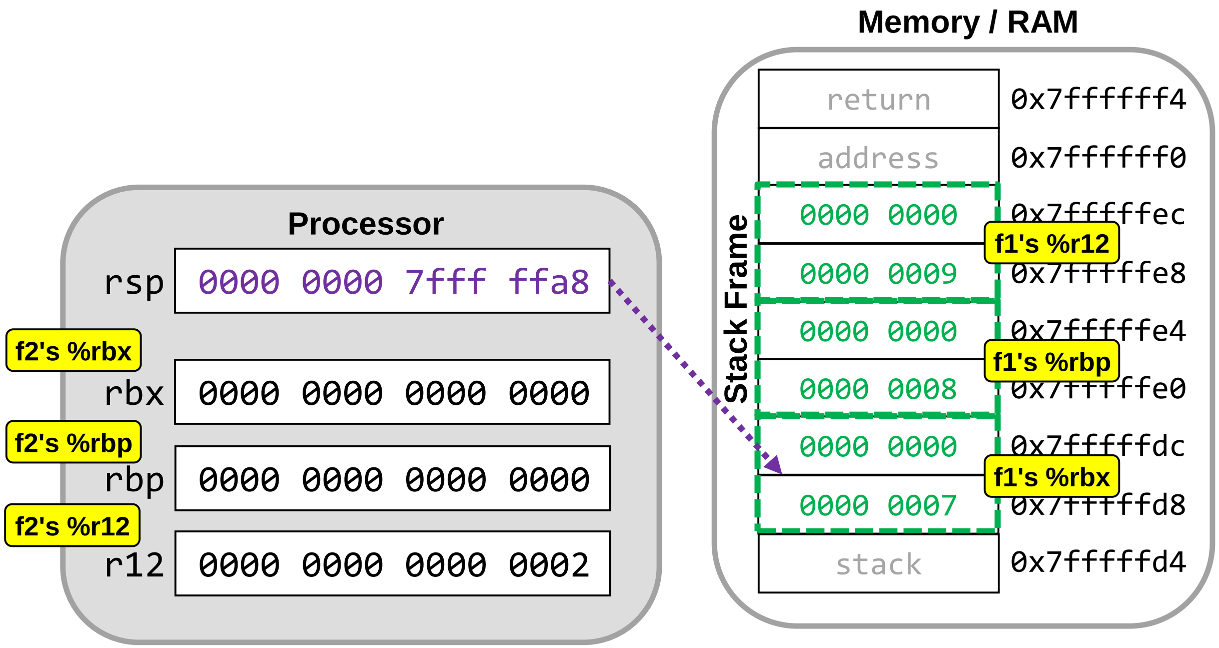
- One procedure might overwrite a register value needed by the caller
- If f1() had values in %rbx, %rbp, and %r12 before calling f2() and then needed those values upon return, f2() may accidentally overwrite them

```
f2:     pushq   %r12
        pushq   %rbp
        pushq   %rbx
        subq    $0x30, %rsp
        movl    %edi, %r12d
        ...
        movl    $0, %ebx
        ...
        movslq  %ebx, %rbp
        leaq    0x10(%rsp,%rbp,4), %rdi
        ...
        popq    %rbx
        popq    %rbp
        popq    %r12
        ret

f1:     ...
        movl    $7, %ebx
        movl    $8, %ebp
        movq    $9, %r12
        movl    $2, %rdi
        call    f2
        ...
        add     %ebx, %ebp
        subq    $1, %r12
        ...
```

Why are these needed?

Solution: Save/restore registers to/from the stack before overwriting it

- Which ones? Any register?



Caller & Callee-Saved Convention

- Having to always play it safe and save a register to the stack before using it can decrease performance
- To increase performance, a standard is set to indicate which registers must be preserved (callee-saved) and which ones can be overwritten freely (caller-saved)
  - Callee Saved: Push values before overwriting them; restore before returning
  - Caller Saved: Push if the register value is needed after the function call; callee can freely overwrite; caller will restore upon return

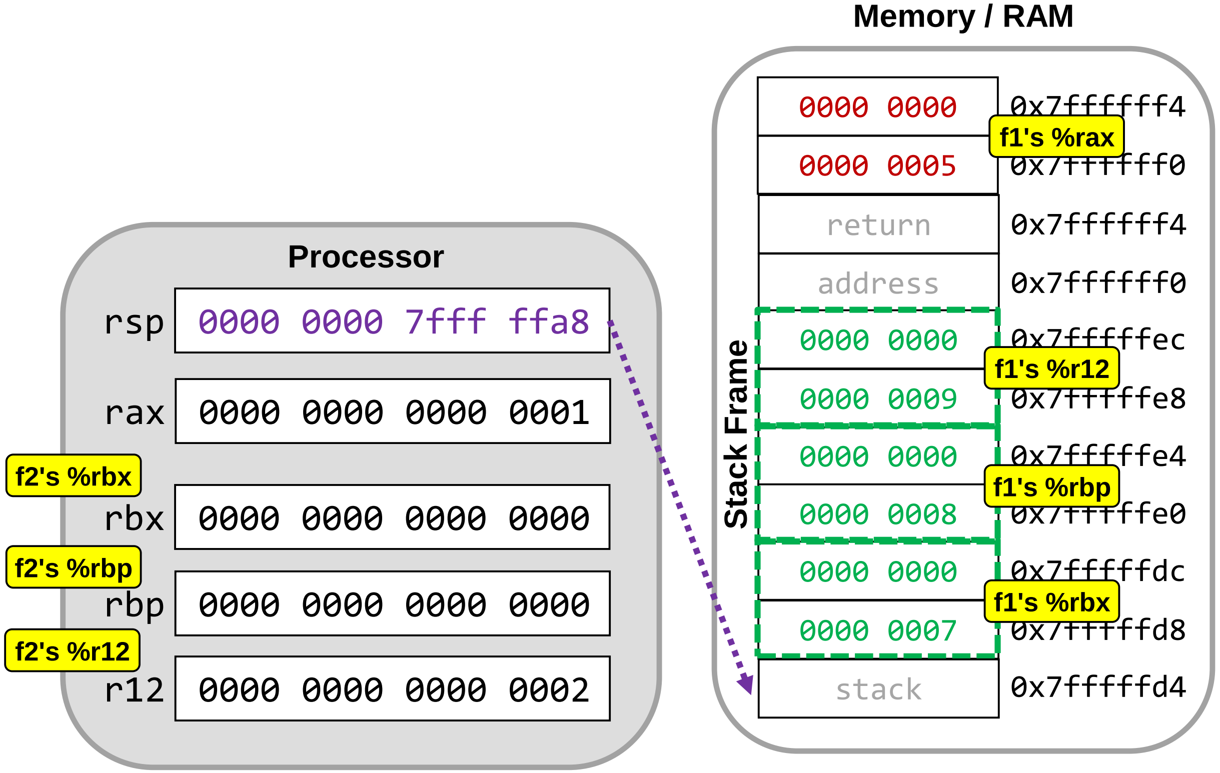
Callee-saved (Callee must ensure the value is not modified)	%rbp, %rbx, %r12-%r15, %rsp*
Caller-saved (Caller must save the value if it wants to preserve it across a function call)	All other registers

*%rsp need not be saved to the stack but should have the same value upon return as it did when the call was made

Caller vs. Callee Saved

- One procedure might overwrite a register value needed by the caller
- If f1() had values in %rbx, %rbp, and %r12 before calling f2() and then needed those values upon return, f2() may accidentally overwrite them

```
f2:     pushq   %r12   ┐
        pushq   %rbp   ├  Callee Saved
        pushq   %rbx   ┘
        subq    $0x30, %rsp
        movl    %edi, %r12d
        movl    $0, %ebx
        movl    $1, %eax
        movslq  %ebx, %rbp
        leaq    0x10(%rsp,%rbp,4), %rdi
        popq    %rbx
        popq    %rbp
        popq    %r12
        ret

f1:     ...
        movl    $7, %ebx
        movl    $8, %ebp
        movq    $9, %r12
        movq    $5, %rax ┐
        push    %rax     ┘ Caller Saved
        movl    $2, %rdi
        call    f2
        pop     %rax
        add     %ebx, %ebp
        subq    $1, %r12
        ...
```



Summary

- To support subroutines we need to save the return address on the stack
  - call and ret perform this implicitly
- There must be agreed upon locations where arguments and return values can be communicated
- The stack is a common memory location to allocate space for saved values and local variables