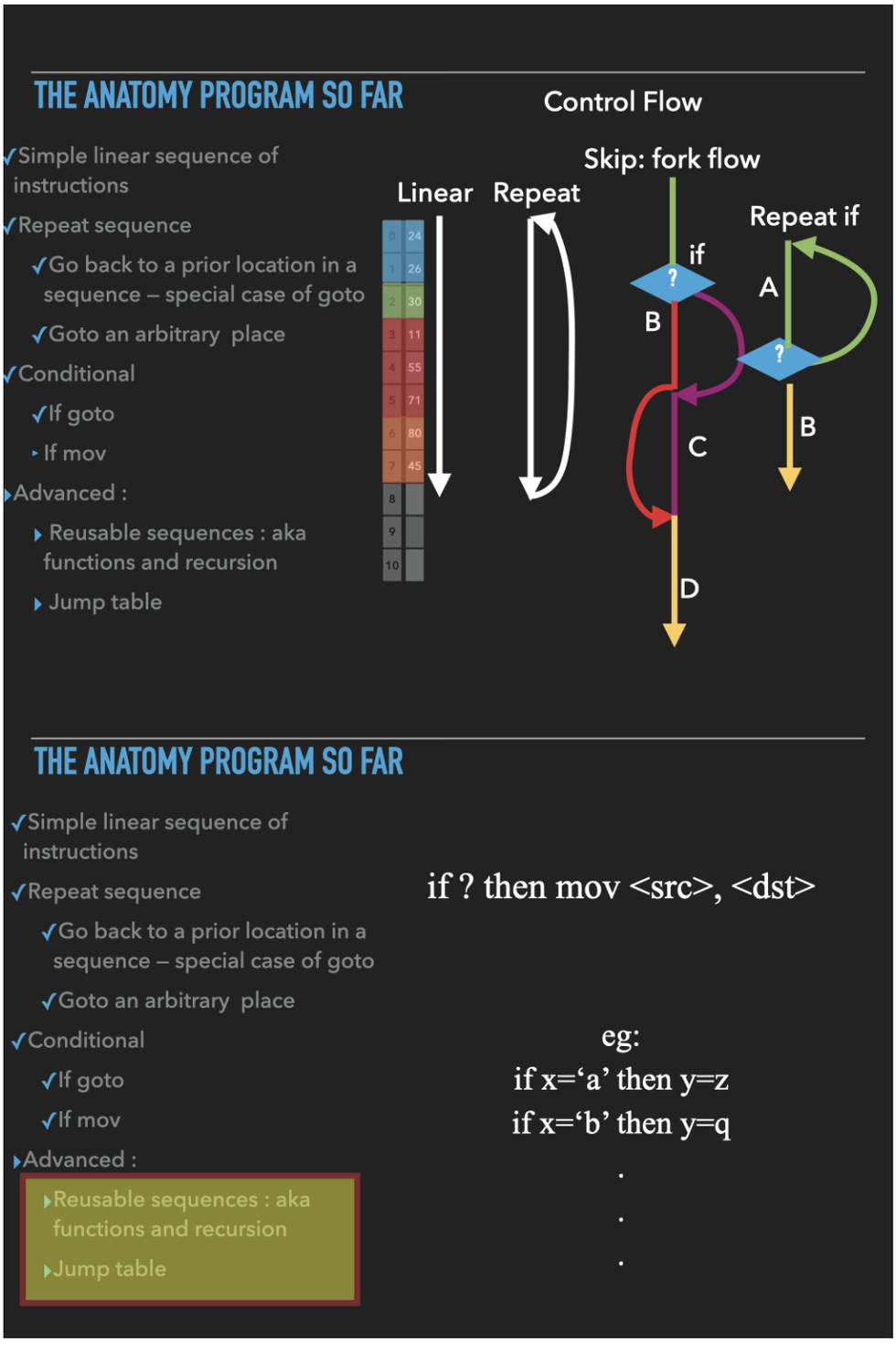
SLS Lecture 11 : Program Anatomy II : Functions

Contents

- create a directory `mkdir sum; cd sum`
- copy sumit and usesumit code examples
- add a `Makefile` to automate assembling and linking
  - we are going run the commands by hand this time to highlight the details
- add our `setup.gdb` and `usesumABC.gdb` to make working in gdb easier
- normally you would want to track everything in git



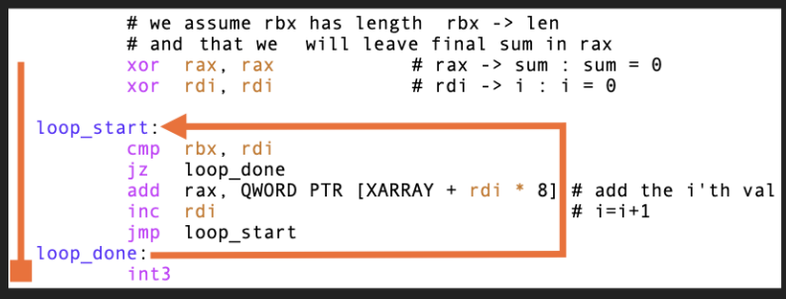
WHY IS JMP NOT GOOD ENOUGH?

OUR CODE:

```
        # we assume rbx has length  rbx -> len
        # and  that we  will leave final sum in rax
        xor  rax, rax            # rax -> sum : sum = 0
        xor  rdi, rdi            # rdi -> i : i = 0

loop_start:
        cmp  rbx, rdi
        jz   loop_done
        add  rax, QWORD PTR [XARRAY + rdi * 8] # add the i'th val
        inc  rdi                               # i=i+1
        jmp  loop_start
loop_done:
        int3
```

OUR CODE: VISUALIZING ITS CONTROL FLOW



```
        # we assume rbx has length  rbx -> len
        # and that we  will leave final sum in rax
        xor  rax, rax            # rax -> sum : sum = 0
        xor  rdi, rdi            # rdi -> i : i = 0

loop_start:
        cmp  rbx, rdi
        jz   loop_done
        add  rax, QWORD PTR [XARRAY + rdi * 8] # add the i'th val
        inc  rdi                               # i=i+1
        jmp  loop_start
loop_done:
        int3
```

OUR CODE: VISUALIZING ITS CONTROL FLOW

```
        # we assume rbx has length  rbx -> len
        # and that we  will leave final sum in rax
        xor  rax, rax            # rax -> sum : sum = 0
        xor  rdi, rdi            # rdi -> i : i = 0

loop_start:
        cmp  rbx, rdi
        jz   loop_done
        add  rax, QWORD PTR [XARRAY + rdi * 8] # add the i'th valsum
        inc  rdi                               # i=i+1
        jmp  loop_start
loop_done:
        int3
```

OUR CODE: LETS LOCATE IT IN MEMORY


```
        .intel_syntax noprefix
        .global sumIt                   # directive to let the linker
                                        # know that this symbolic label
                                        # is Global -- can be reference
          # code to sum data at XARRAY
          # we assume rbx has length  rbx -> len
          # and that we will leave final sum in rax
sumIt:                                     # location of this code is
                                           # sumIt .eg sumIt is 0x1022
          xor   rax, rax                # rax -> sum : sum = 0
          xor   rdi, rdi                # rdi -> i : i = 0

loop_start:
          cmp   rbx, rdi
          jz    loop_done
          add   rax, QWORD PTR [XARRAY + rdi * 8] # add the i'th val
          inc   rdi                                # i=i+1
          jmp   loop_start
loop_done:
          int3
```

OUR CODE: LET'S USE OUR SUM CODE

Ok lets now write a second sequence that sets things up and then uses our other sequence. Lets place write the second sequence in usesum.S

```
    .intel_syntax noprefix
    .data
    #  a place for us to store how much data is in the XARRAY
    #  initialized it to 0
XARRAY_LEN:
    .quad 0x0

    # reserve enough space for 1024 8 byte values
    #  third argument is alignment.... turns out cpu
    #  cpu is more efficient if things are located at address
    #  of a particular 'alignment (see intel manual)
    .comm XARRAY, 8*1024, 8
    .comm sum, 8, 8     # space to store final sum

    .text
    .global _start
_start:
    mov rbx, QWORD PTR [XARRAY_LEN]

    jmp sumIt
    mov QWORD PTR [sum], rax

    int3
```

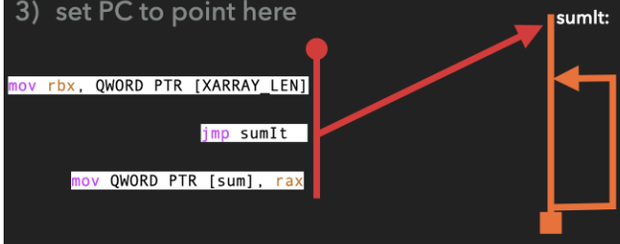
now we can assemble both and ask the linker to create a single binary image that has both sequences in them

```
as -g -a=sumit.o.lst sumit.S -o sumit.o
as -g -a=usesum.o.lst usesum.S -o usesum.o
lld -g -Map=usesum.map usesum.o sumit.o -o usesum
```

OUR CODE: HOW DO THINGS LOOK NOW?

We now have two sequences in memory that are connected by a jmp

1) load number at &XARRAY
2) set length in &XARRAY_LEN
3) set PC to point here

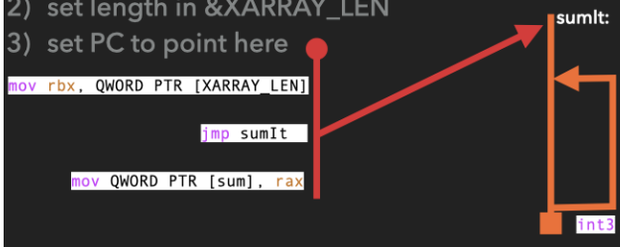


What are we missing?

OUR CODE: HOW DO THINGS LOOK NOW?

We now have two sequences in memory that are connected by a jmp

1) load number at &XARRAY
2) set length in &XARRAY_LEN
3) set PC to point here

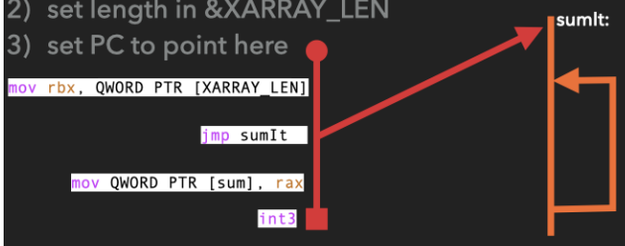


Certainly we don't want the stop here.

OUR CODE: HOW DO THINGS LOOK NOW?

We now have two sequences in memory that are connected by a jmp

1) load number at &XARRAY
2) set length in &XARRAY_LEN
3) set PC to point here

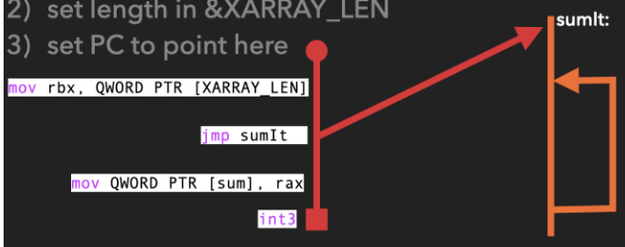


Rather we want to stop here

OUR CODE: HOW DO THINGS LOOK NOW?

We now have two sequences in memory that are connected by a jmp

1) load number at &XARRAY
2) set length in &XARRAY_LEN
3) set PC to point here

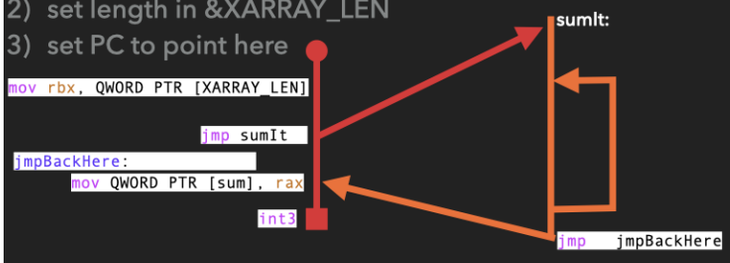


Now what?

OUR CODE: HOW DO THINGS LOOK NOW?

We now have two sequences in memory that are connected by a jmp

1) load number at &XARRAY
2) set length in &XARRAY_LEN
3) set PC to point here

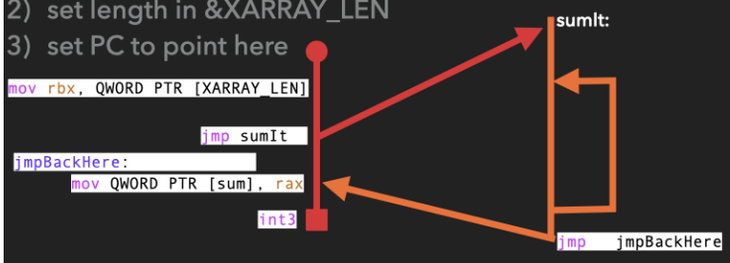


we need a connection back!

OUR CODE: HOW DO THINGS LOOK NOW?

We now have two sequences in memory that are connected by a jmp

1) load number at &XARRAY
2) set length in &XARRAY_LEN
3) set PC to point here

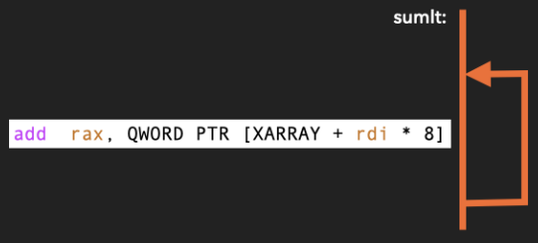


we need a connection back!

WHY MIGHT THIS APPROACH CAUSE US PROBLEMS DOWN THE ROAD?

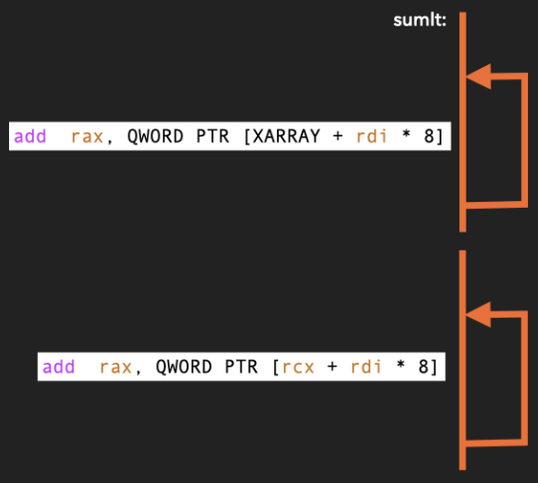
LETS MAKE OUR CODE A LITTLE MORE GENERIC SO WE CAN USE IT TO SUM TWO DIFFERENT ARRAYS

OUR CODE: REMOVE HARDCODE ARRAY LOCATION



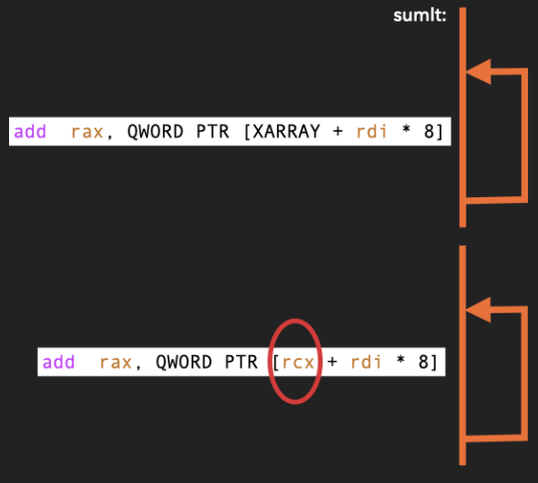
Right now this line hardcodes where we process data from.

OUR CODE: REMOVE HARDCODE ARRAY LOCATION



If we assume that the caller passes in the location of the array in rcx then we can make our code sum any array by this change.

OUR CODE: REMOVE HARDCODE ARRAY LOCATION



If we assume that the caller passes in the location of the array in rcx then we can make our code sum any array by this change.

ADDRESSING MODES:

THE CPU'S DIFFERENT WAYS OF IDENTIFYING AN MEMORY ADDRESS

- When writing assembly code we need to say what memory address we want to either read or write data form (eg mov in intel speak).
- Processor can calculate the address in several was as we have seen.
- Most generally there is an equation (and variants of it) that it support
- The syntax of the assembler allows us to write this equation when we specify an operand for an instruction mnemonic

INTEL ASSEMBLER SYNTAX FOR SPECIFYING AN INTEL MEMORY ADDRESS EQUATION

Address mode equations:

Most General Form:

[Rb + Ri * S + D]

- **Rb: Base Register (any of the 16 GPRS)**
- **Ri: Index Register (any of the 16 GPRS)**
- **S: A value of 1, 2, 4 or 8**
- **D: 1, 2 or 4 byte number**
  - **Label or constant**

Various Special forms in which some terms are skipped

[D]

[Rb]

[Rb + Ri]

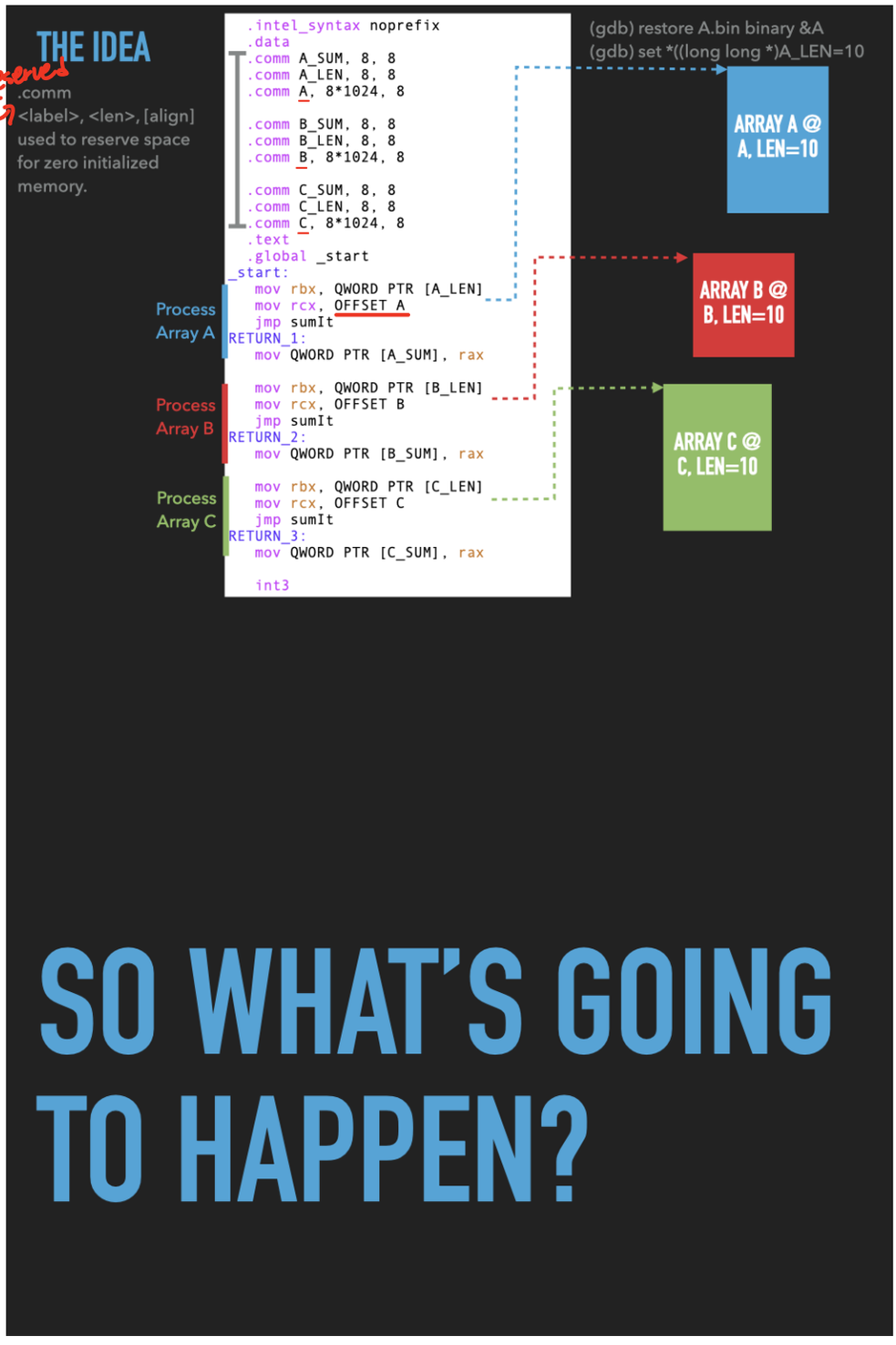
[Rb + Ri + D]

[Rb + Ri * S]

Remember we prefix all of these with size of access we intent eg. one of

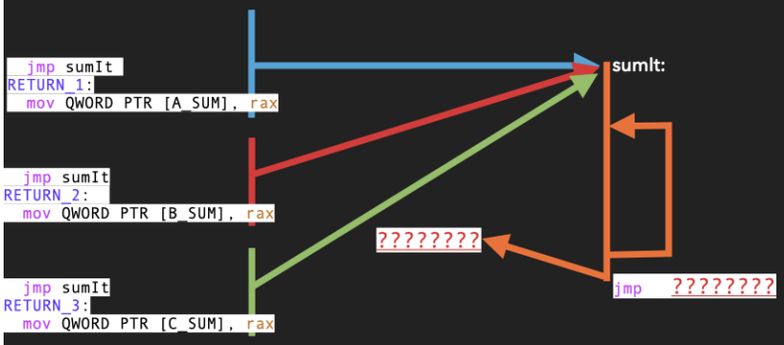
QWORD PTR, DWORD PTR, WORD PTR, BYTE PTR

OK LET'S USE OUR NEW POWERFUL CODE SEQUENCE



SO WHAT'S GOING TO HAPPEN?

HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

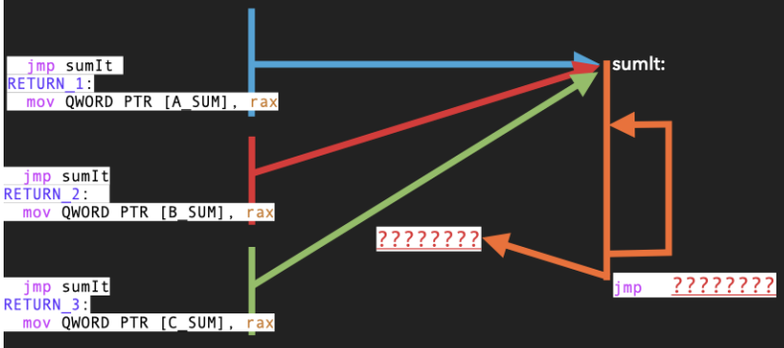


HOW CAN WE SOLVE THIS PROBLEM

TRY 1

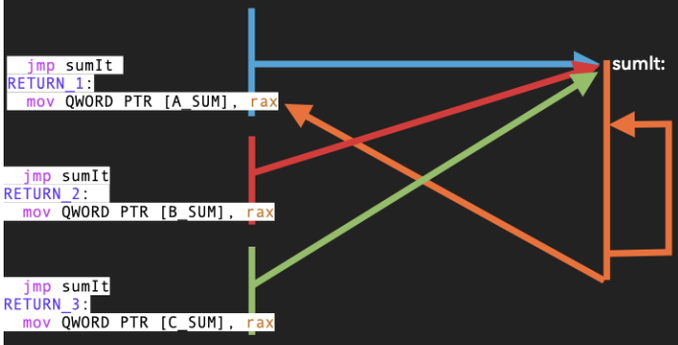
HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

- What would we like?



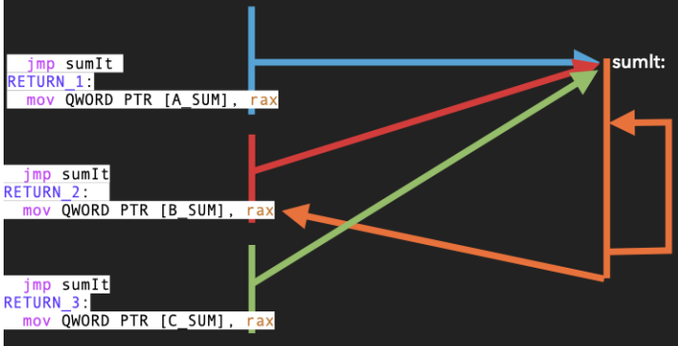
HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

- It should jump back to the address of the instruction after the jmp that started it!



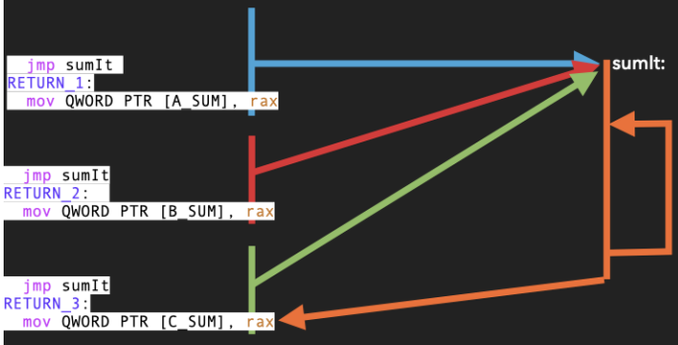
HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

- It should jump back to the address of the instruction after the jmp that started it!



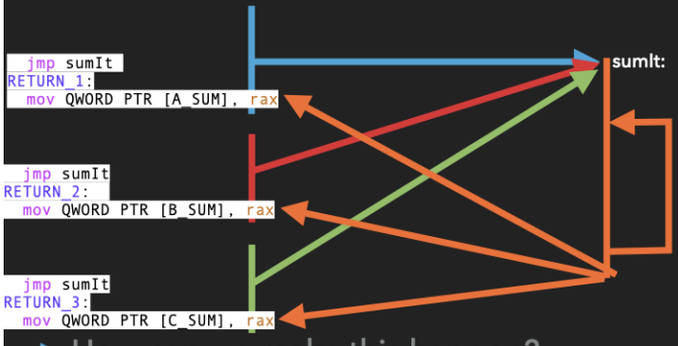
HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

- It should jump back to the address of the instruction after the jmp that started it!



HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

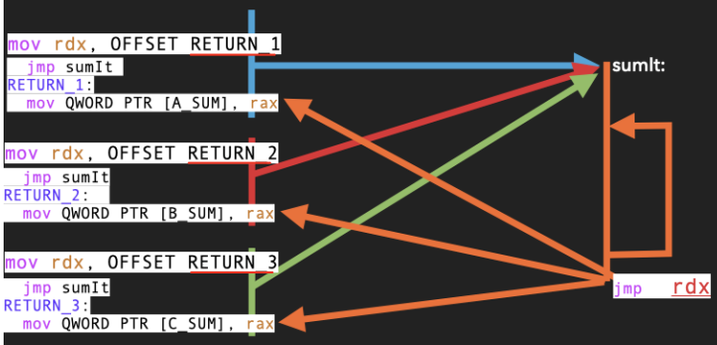
- It should jump back to the address of the instruction after the jmp that started it!



- How can we make this happen?

HOW DO WE MODIFY SUMIT? WHAT ADDRESS DO WE JUMP BACK TOO?

- It should jump back to the address of the instruction after the jmp that started it!

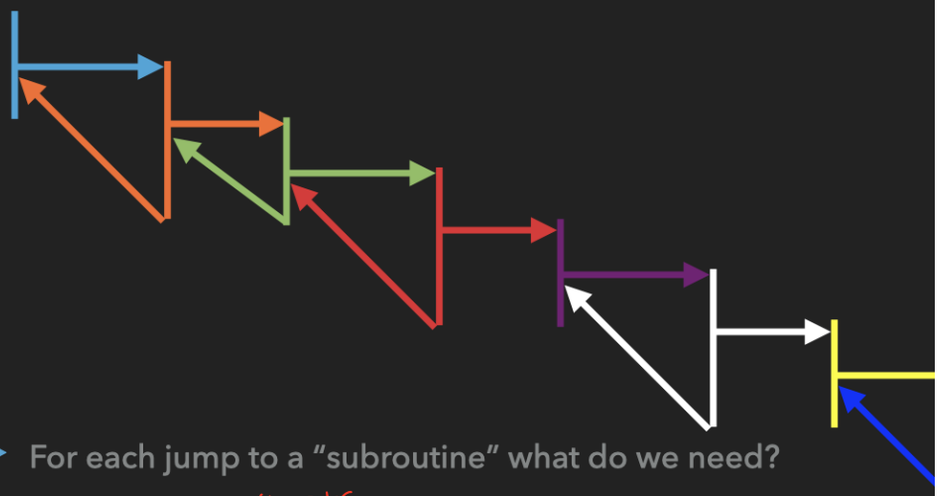


How can we make this happen?

- Maybe would could pass the address to return to as an argument in a register

YES THIS WORKS!
BUT . . .

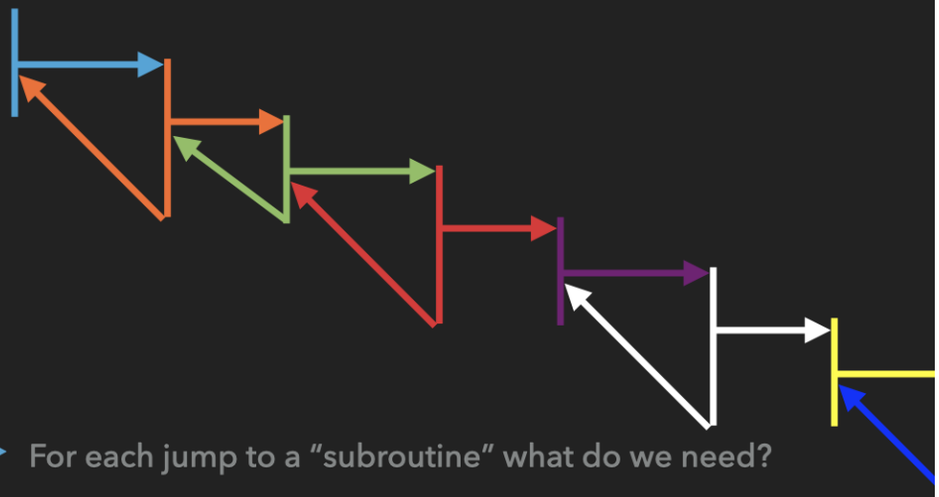
WHAT WOULD HAPPEN IF WE HAD THIS KIND OF STRUCTURE



- For each jump to a "subroutine" what do we need?

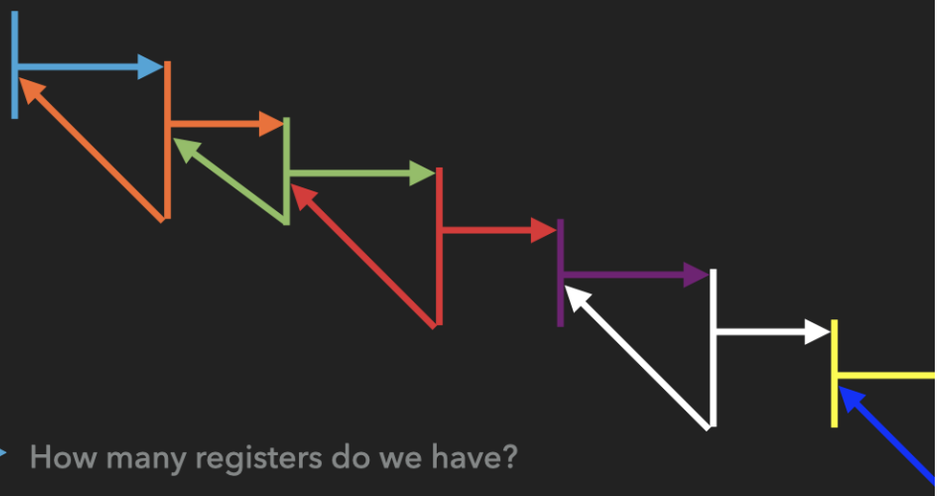
short term it works

WHAT WOULD HAPPEN IF WE HAD THIS KIND OF STRUCTURE



- For each jump to a "subroutine" what do we need?
  - A return label and a register to hold the return address

WHAT WOULD HAPPEN IF WE HAD THIS KIND OF STRUCTURE

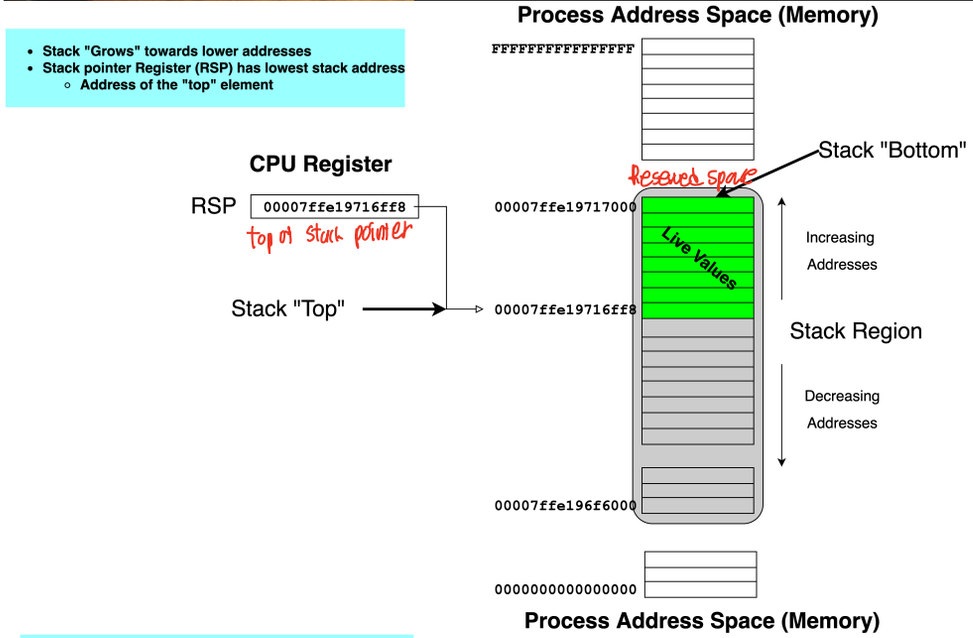


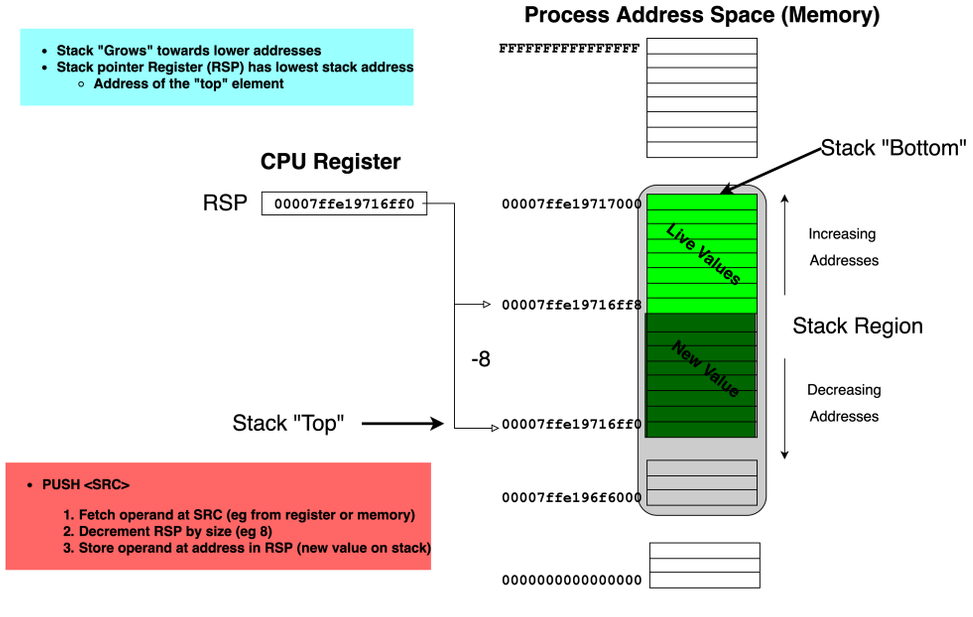
- How many registers do we have?

WHAT WOULD HAPPEN IF WE HAD THIS KIND OF STRUCTURE

- How many registers do we have? Remember all the real work we do requires registers we can't tie them all up holding return addresses :-(

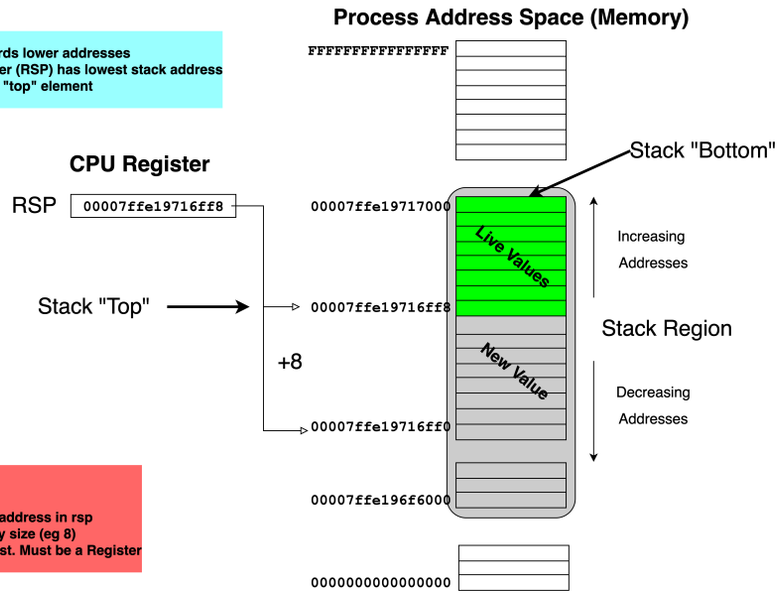
THE TYPICAL SOLUTION — "THE" STACK





- PUSH <SRC>
  1. Fetch operand at SRC (eg from register or memory)
  2. Decrement RSP by size (eg 8)
  3. Store operand at address in RSP (new value on stack)

- Stack "Grows" towards lower addresses
- Stack pointer Register (RSP) has lowest stack address
  - Address of the "top" element



- POP <DEST Register>
  1. Read value from address in rsp
  2. Increment RSP by size (eg 8)
  3. Store value in Dest. Must be a Register

TEXT

THE STACK IS AN AREA OF MEMORY WE CAN ALWAYS USE TO STORE AND WORK WITH DATA TEMPORARILY

- What are some ways we can make use of it
  - push register values on it so that we can then use the registers in any way we like
    - when done we can restore the old values by pop'ing
    - Our job to make sure we are discipled about the order
  - IMPORTANT SPECIAL CASE: store return address
  - Organizer the stacks that we can have place for temporary variables when we run out of registers

STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```

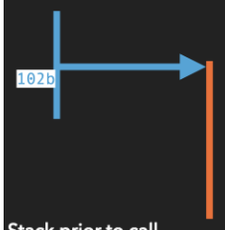
Stack prior to call

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```



Stack prior to call

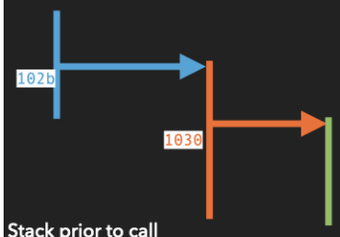
```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```


STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```

Stack prior to call

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```


```
(gdb) x/3xg $sp
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```

STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```

Stack prior to call

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

```
(gdb) x/3xg $sp
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```

```
(gdb) x/4xg $sp
0xffff9fe8:     0x0000000000001018      0x0000000000001030
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```


STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```


Stack prior to call

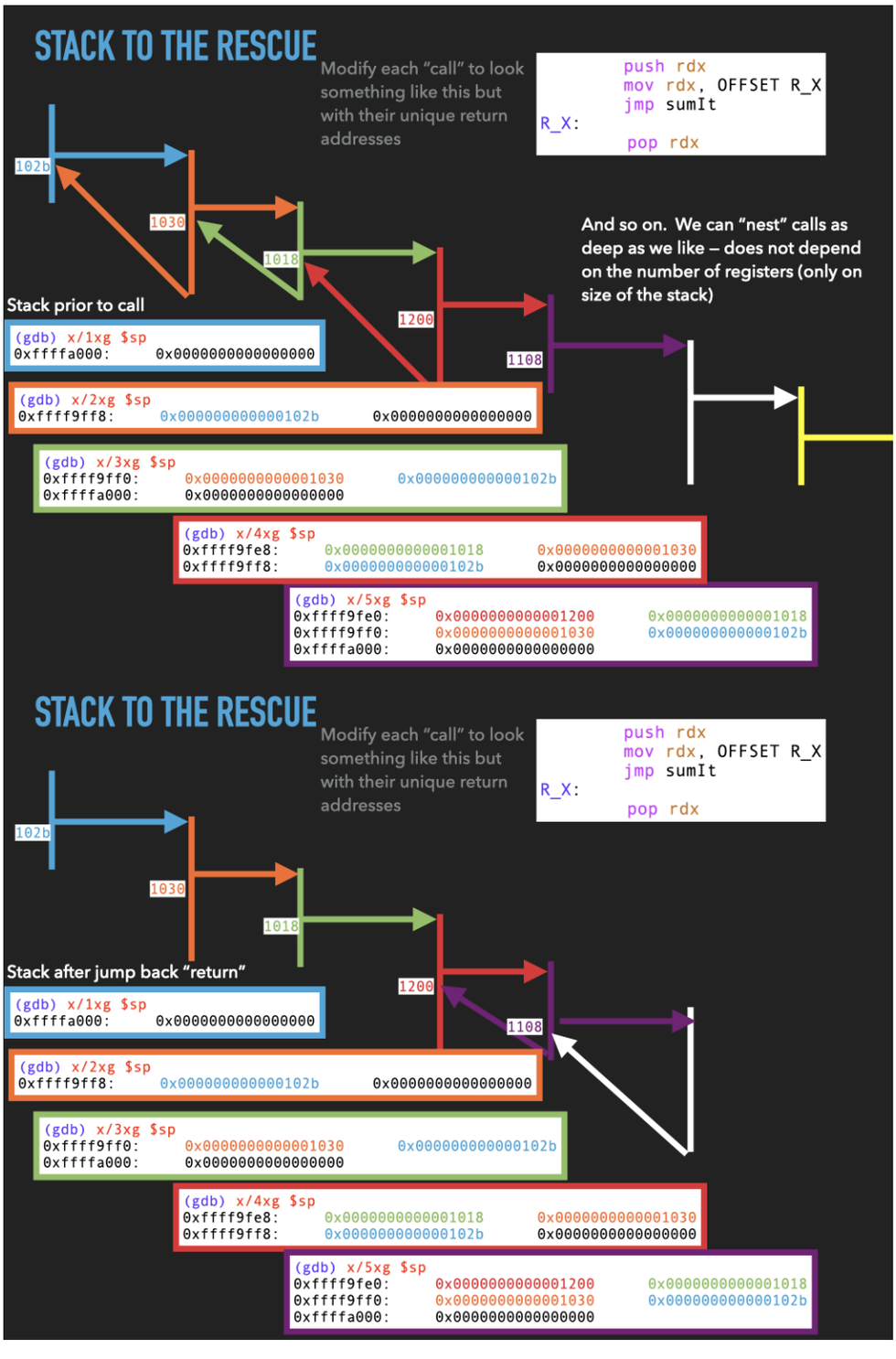
```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

```
(gdb) x/3xg $sp
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```

```
(gdb) x/4xg $sp
0xffff9fe8:     0x0000000000001018      0x0000000000001030
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

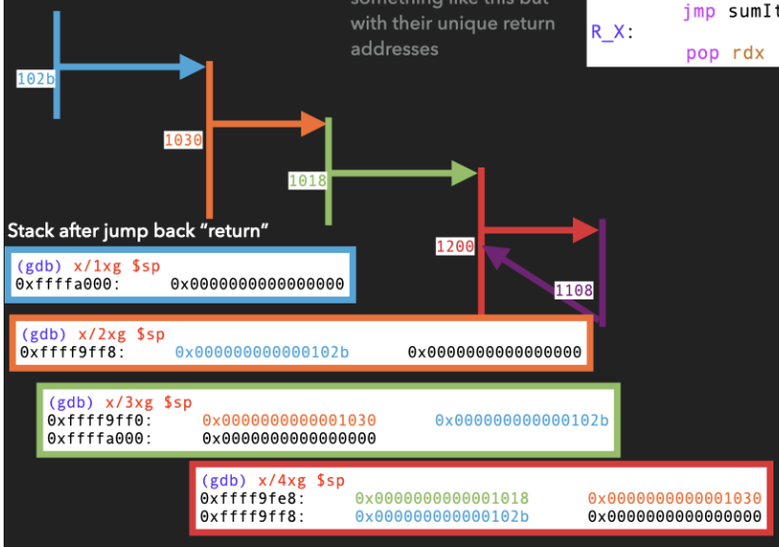

```
(gdb) x/5xg $sp
0xffff9fe0:     0x0000000000001200      0x0000000000001018
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```


STACK TO THE RESCUE
Modify each "call" to look something like this but with their unique return addresses
push rdx
mov rdx, OFFSET R_X
jmp sumIt
R_X:
pop rdx
102b
1030
1018
1200
1108
And so on. We can "nest" calls as deep as we like – does not depend on the number of registers (only on size of the stack)
Stack prior to call
(gdb) x/1xg $sp
0xffffa000: 0x0000000000000000
(gdb) x/2xg $sp
0xffff9ff8: 0x000000000000102b 0x0000000000000000
(gdb) x/3xg $sp
0xffff9ff0: 0x0000000000001030 0x000000000000102b
0xffffa000: 0x0000000000000000
(gdb) x/4xg $sp
0xffff9fe8: 0x0000000000001018 0x0000000000001030
0xffff9ff8: 0x000000000000102b 0x0000000000000000
(gdb) x/5xg $sp
0xffff9fe0: 0x0000000000001200 0x0000000000001018
0xffff9ff0: 0x0000000000001030 0x000000000000102b
0xffffa000: 0x0000000000000000
STACK TO THE RESCUE
Modify each "call" to look something like this but with their unique return addresses
push rdx
mov rdx, OFFSET R_X
jmp sumIt
R_X:
pop rdx
102b
1030
1018
1200
1108
Stack after jump back "return"
(gdb) x/1xg $sp
0xffffa000: 0x0000000000000000
(gdb) x/2xg $sp
0xffff9ff8: 0x000000000000102b 0x0000000000000000
(gdb) x/3xg $sp
0xffff9ff0: 0x0000000000001030 0x000000000000102b
0xffffa000: 0x0000000000000000
(gdb) x/4xg $sp
0xffff9fe8: 0x0000000000001018 0x0000000000001030
0xffff9ff8: 0x000000000000102b 0x0000000000000000
(gdb) x/5xg $sp
0xffff9fe0: 0x0000000000001200 0x0000000000001018
0xffff9ff0: 0x0000000000001030 0x000000000000102b
0xffffa000: 0x0000000000000000

STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```



Stack after jump back "return"

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

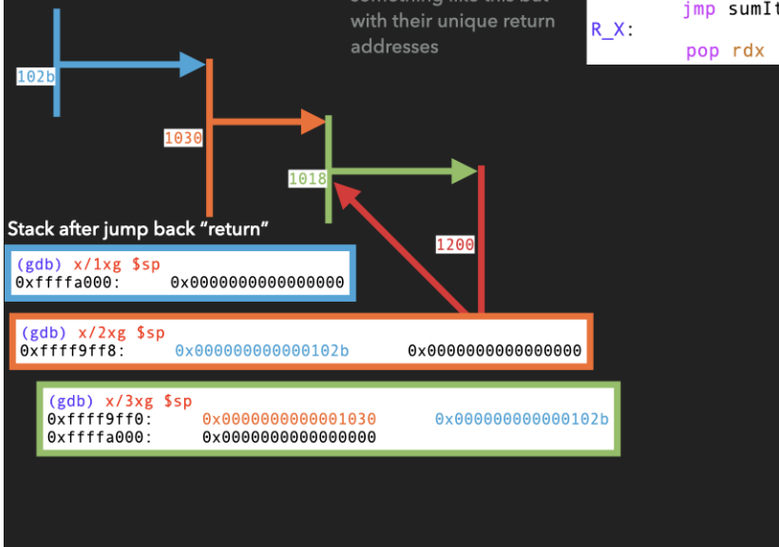
```
(gdb) x/3xg $sp
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```

```
(gdb) x/4xg $sp
0xffff9fe8:     0x0000000000001018      0x0000000000001030
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```


STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```

Stack after jump back "return"

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

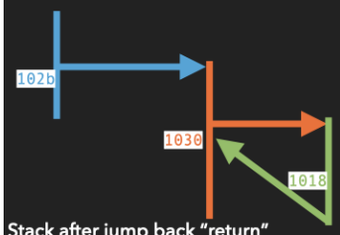
```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

```
(gdb) x/3xg $sp
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```

STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```



Stack after jump back "return"

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

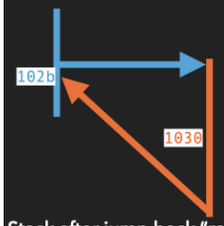
```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

```
(gdb) x/3xg $sp
0xffff9ff0:     0x0000000000001030      0x000000000000102b
0xffffa000:     0x0000000000000000
```


STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```

Stack after jump back "return"

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

```
(gdb) x/2xg $sp
0xffff9ff8:     0x000000000000102b      0x0000000000000000
```

STACK TO THE RESCUE

Modify each "call" to look something like this but with their unique return addresses

```
        push rdx
        mov rdx, OFFSET R_X
        jmp sumIt
R_X:
        pop rdx
```

```
102b
```

Stack after jump back "return"

```
(gdb) x/1xg $sp
0xffffa000:     0x0000000000000000
```

After we jump back and "pop" rdx and the stack pointer are restored. Everything is automatically back the way it was – as if neither were changed

Of course the memory area of where the stack pointer was still has the values but they are not considered "alive" – only memory below the stack pointer is considered valid.

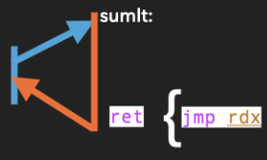
It is our job to maintain this "Stack Discipline"

INTEL HAS ADDITIONAL INSTRUCTIONS TO MAKE THIS EVEN EASIER

- A Pair of instructions – CALL and RET that implement "procedure" calls using the stack – no need for a register or label

```
    push rdx
    mov rdx, OFFSET R_1
    jmp sumIt
R_1:
    pop rdx
    mov QWORD PTR [A_SUM], rax
```

`call sumIt`

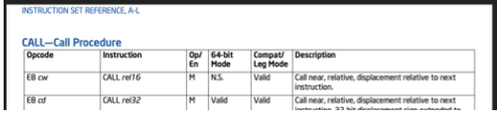


CALL

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

RET

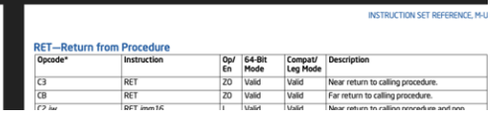
Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the **return is made to the instruction that follows the CALL instruction** … the return address is popped…



INSTRUCTION SET REFERENCE, A-L

CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 *cw*	CALL *rel16*	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 *cd*	CALL *rel32*	M	Valid	Valid	Call near, relative, displacement relative to next



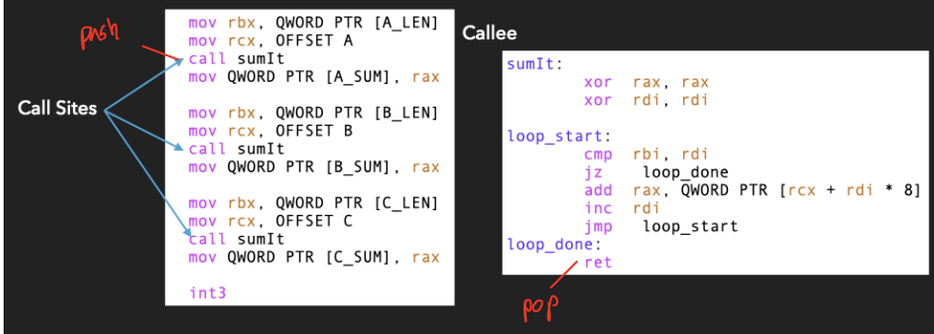
INSTRUCTION SET REFERENCE, M-U

RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 *iw*	RET *imm16*	I	Valid	Valid	Near return to calling procedure and pop

INTEL PROCEDURE CALL —

- CALL's push address of instruction following call on to the stack. RET pops top value and jumps to that address (automatically going back to most recent call site)
- An active call effectively is represented as an "entry", also called a FRAME, on the stack
- For a given call the call site is the CALLER and the target is the CALLEE
- Naturally supports nesting of procedure calls as a tree and recursion.



WHAT ELSE CAN WE USE THE STACK FOR?

you have to watch from the first video

like they stack up jokes

Basically it's making guest tv shows

SPILL

When writing assembly code and we run out of registers or not sure if it is safe to use a register because some other part of the code might have a "live" value in it – a value the code expects to be there (eg current value of i) we can "spill" the contents of the register on the stack and use the register as we like ands then restore the register original value from the stack. This way we won't break anything else.

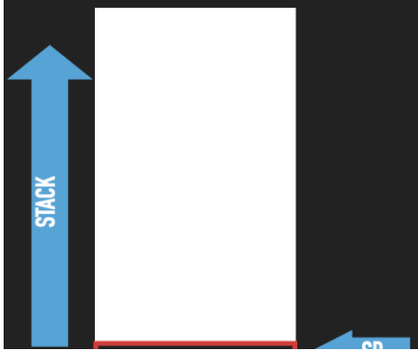
BUT MUST BE CAREFUL TO RESTORE BEFORE WHEN WE ARE DONE

```
sumIt:
        xor  rax, rax              # rax -> sum : sum = 0
        push rdi                   # spill current value of rdi before we
                                   # modify it -- now we can safely use it for our own use
        xor  rdi, rdi              # rdi -> i : i = 0

loop_start:
        cmp  rdi, rbx
        jz   loop_done
        add  rax, QWORD PTR [rcx + rdi * 8] # add the i'th value to the sum
        inc  rdi                   # i=i+1
        jmp  loop_start
loop_done:
        pop rdi                      # restore rdi back to its original value
        ret                          # return address should now be top value on the stack
```

LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

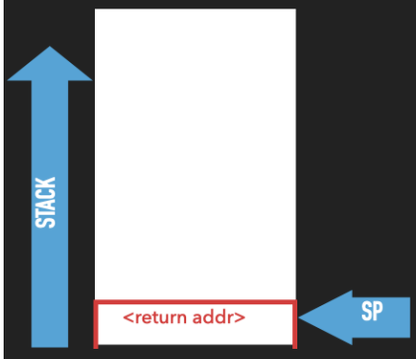
It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!



LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  rbx, DATA
call numUniqAscii
```

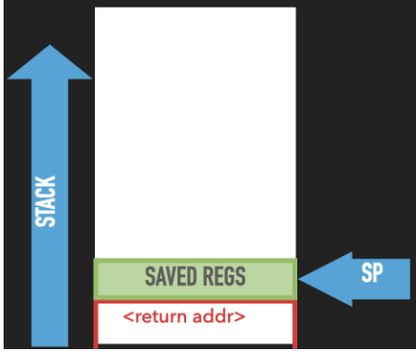


LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  rbx, DATA
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
```



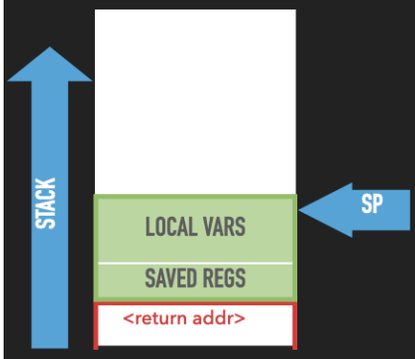
LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
```

Stack point



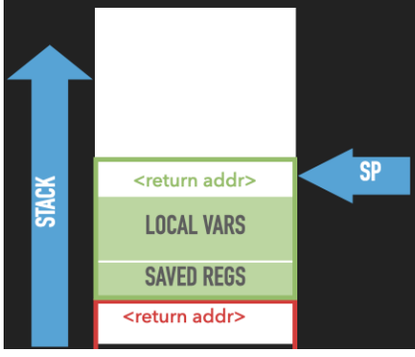
LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
```


```
        .
        .
        .
call countOnes
```

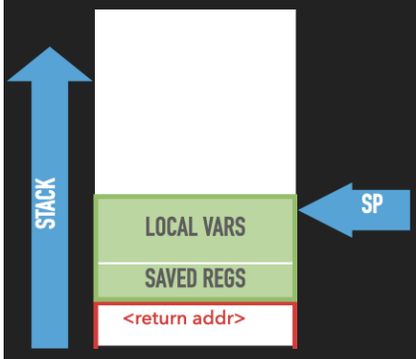


LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
        .
        .
        .
call countOnes
```



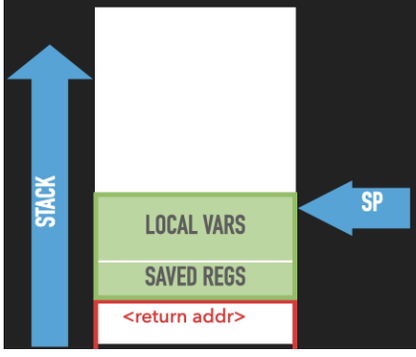
LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
        .
        .
        .
call countOnes
```


```
add  rsp, 256
```

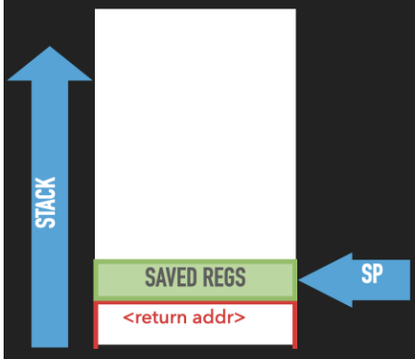


LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
        .
        .
        .
call countOnes
add  rsp, 256
```



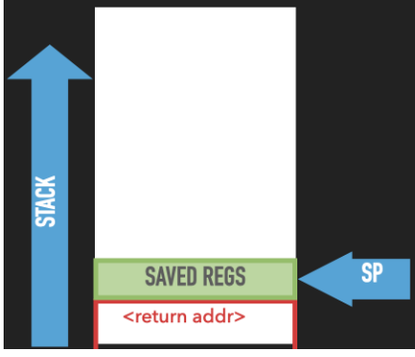
LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
        .
        .
        .
call countOnes
add  rsp, 256
```

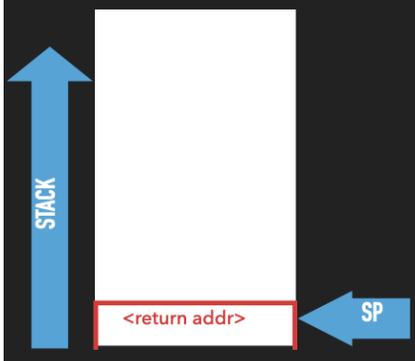

```
pop  r9
pop  r8
```



LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

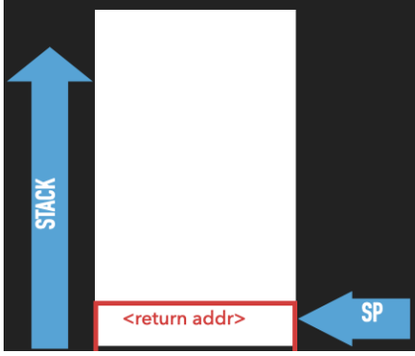



```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
        .
        .
        .
call countOnes
add  rsp, 256
pop  r9
pop  r8
```


LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!

```
mov  DATA, rbx
call numUniqAscii
```

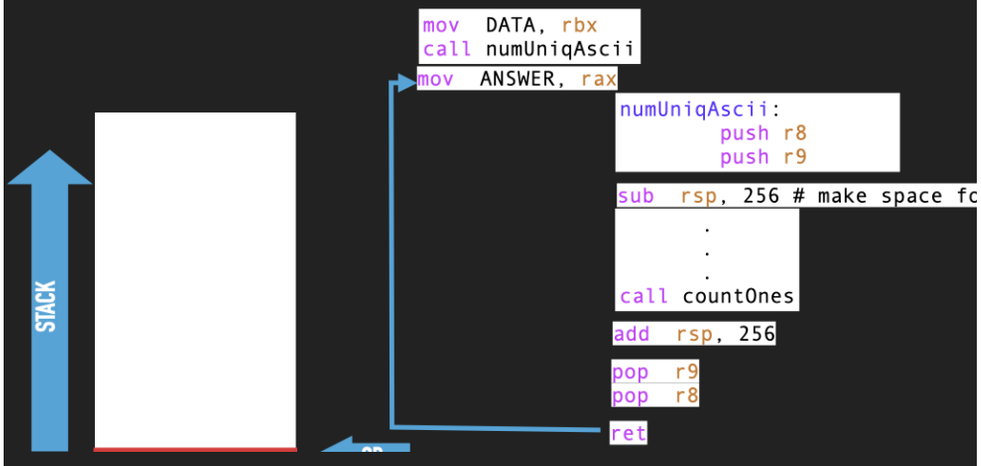




```
numUniqAscii:
        push r8
        push r9
sub  rsp, 256 # make space fo
        .
        .
        .
call countOnes
add  rsp, 256
pop  r9
pop  r8
```

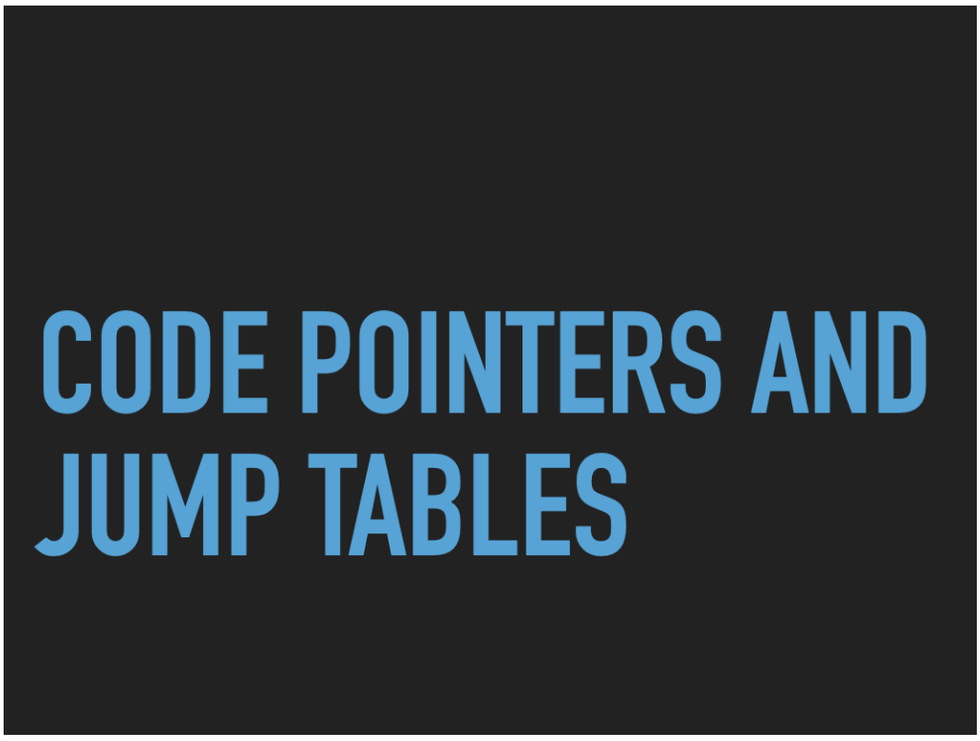

```
ret
```

LOCAL VARIABLES — A WAY TO USE THE STACK TO MAKE PROCEDURES MORE POWERFUL

It is really useful to have some memory that every call to a procedure can use as space for variables that automatically becomes available when the call starts and is freed when the call is done. STACK to the rescue again!



RECURSION : STACK MAGIC!



11.1. Examples used in previous slides

11.1.1. `sumit` as per last lecture

Hardcode `int3` stops us from saving our sum as we would like

CODE: asm - sumit.s : Version 1

```
        .intel_syntax noprefix

        .section .text

        # tell linker that sumIt symbol can be referenced in other files
        .global sumIt

        # code to sum data at XARRAY
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
sumIt:                          # label that marks where this code begins
        xor  rax, rax           # rax -> sum : sum = 0
        xor  rdi, rdi           # rdi -> i : i = 0

        # code to sum data at XARRAY
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
loop_start:
        cmp  rbx, rdi                           # rbx - rdi
        jz   loop_done                          # if above is zero (they are equal) jump
        add  rax, QWORD PTR [XARRAY + rdi * 8] # add the i'th value to the sum
        inc  rdi                                # i=i+1
        jmp   loop_start                        # go back to the start of the loop
loop_done:
        int3                                    # don't know where to go next
                                                # give up and "trap" to the debugger
```

Corresponding: usesum

rax: sum
rdi : i
rbx : array len

CODE: asm - sumit.s : Version 1

```
        .intel_syntax noprefix

        .section .data
        #  a place for us to store how much data is in the XARRAY
        #  initalized it to 0
XARRAY_LEN:
        .quad 0x0
        # reserve enough space for 1024 8 byte values
        #  third argument is alignment.... turns out cpu
        #  cpu is more efficient if things are located at address
        #  of a particular 'alignment (see intel manual)
        .comm XARRAY, 8*1024, 8
        .comm sum, 8, 8         # space to store final sum

        .section .text
        .global _start
_start:
        mov rbx, QWORD PTR [XARRAY_LEN]

        jmp sumIt
        mov QWORD PTR [sum], rax

        int3
```

sum of all array #

see last lecture notes to in terms of how to use `gdb` to run and play with `usesum`

11.1.2. Made `sumit` more generic but ...

1. **Make the address of the array a parameter passed to `sumit`.**
2. **Changed `sumit` code to use this instead of `XARRAY`**

3. Now have `usesum` leave space for three arrays and have three jumps to `sumit`
4. remove `int3` from `sumit` and replace it with a `jmp` back to the address after the first jump.
5. BUT WE HAVE PROBLEMS IN `sumit`:

- how do we `jump` back to the right spot?

CODE: asm - sumit2.s : Version 2

```
        .intel_syntax noprefix

        .section .text

        # tell linker that sumIt symbol can be referenced in other files
        .global sumIt

        # code to sum data in array who's address is in rcx
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
sumIt:                          # label that marks where this code begins
        xor  rax, rax           # rax -> sum : sum = 0
        xor  rdi, rdi           # rdi -> i : i = 0

        # code to sum data at value in rcx
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
loop_start:
        cmp  rbx, rdi                       # rbx - rdi
        jz   loop_done                      # if above is zero (they are equal) jump
        add  rax, QWORD PTR [rcx + rdi * 8] # add the i'th value to the sum
        inc  rdi                            # i=i+1
        jmp   loop_start                    # go back to the start of the loop
loop_done:
        jmp RETURN_1 ← int3           # jump back to hard coded location
```

Corresponding: usesum

CODE: asm - usesum2.s : Version 2

```
        .intel_syntax noprefix
        .data
        .comm A_SUM, 8, 8
        .comm A_LEN, 8, 8
        .comm A, 8*1024, 8

        .comm B_SUM, 8, 8
        .comm B_LEN, 8, 8
        .comm B, 8*1024, 8

        .comm C_SUM, 8, 8
        .comm C_LEN, 8, 8
        .comm C, 8*1024, 8

        .text
        .global _start
_start:
        mov rbx, QWORD PTR [A_LEN]
        mov rcx, OFFSET A
        jmp sumIt
RETURN_1:
        mov QWORD PTR [A_SUM], rax

        mov rbx, QWORD PTR [B_LEN]
        mov rcx, OFFSET B
        jmp sumIt
RETURN_2:
        mov QWORD PTR [B_SUM], rax

        mov rbx, QWORD PTR [C_LEN]
        mov rcx, OFFSET C
        jmp sumIt
RETURN_3:
        mov QWORD PTR [C_SUM], rax
        int3

        .global RETURN_1
        .global RETURN_2
        .global RETURN_3
```

GDB commands to use

```
b _start
run
restore A.bin binary &A
set *((long long *)&A_LEN)=10
restore B.bin binary &B
set *((long long *)&B_LEN)=10
restore C.bin binary &C
set *((long long *)&C_LEN)=10

display /d { (long long)A_SUM, (long long)B_SUM, (long long)C_SUM }
```

Now you can single step or continue as you like

11.1.3. Version 3: Fix our problem by passing in the return address in a register. In our case we use 'rdx' and jump back via this register.

Note this approach requires one register for each call to hold the return address. This has problems if we were to nest calls.

CODE: asm - sumit3.s : Version 3 use rdx to store return address

```
        .intel_syntax noprefix

        .section .text

        # tell linker that sumIt symbol can be referenced in other files
        .global sumIt

        # code to sum data in array who's address is in rcx
        # we assume rdx has the address to jump back too
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
sumIt:                          # label that marks where this code begins
        xor  rax, rax           # rax -> sum : sum = 0
        xor  rdi, rdi           # rdi -> i : i = 0

        # code to sum data at value in rcx
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
loop_start:
        cmp  rbx, rdi                       # rbx - rdi
        jz   loop_done                      # if above is zero (they are equal) jump
        add  rax, QWORD PTR [rcx + rdi * 8] # add the i'th value to the sum
        inc  rdi                            # i=i+1
        jmp   loop_start                    # go back to the start of the loop
loop_done:
        jmp rdx                             # jump back to location held in rdx
```

Corresponding: usesum

CODE: asm - usesum3.s : Version 3

```
        .intel_syntax noprefix
        .data
        .comm A_SUM, 8, 8
        .comm A_LEN, 8, 8
        .comm A, 8*1024, 8

        .comm B_SUM, 8, 8
        .comm B_LEN, 8, 8
        .comm B, 8*1024, 8

        .comm C_SUM, 8, 8
        .comm C_LEN, 8, 8
        .comm C, 8*1024, 8

        .text
        .global _start
_start:
        mov rbx, QWORD PTR [A_LEN]
        mov rcx, OFFSET A
        mov rdx, OFFSET RETURN_1
        jmp sumIt
RETURN_1:
        mov QWORD PTR [A_SUM], rax

        mov rbx, QWORD PTR [B_LEN]
        mov rcx, OFFSET B
        mov rdx, OFFSET RETURN_2
        jmp sumIt
RETURN_2:
        mov QWORD PTR [B_SUM], rax

        mov rbx, QWORD PTR [C_LEN]
        mov rcx, OFFSET C
        mov rdx, OFFSET RETURN_3
        jmp sumIt
RETURN_3:
        mov QWORD PTR [C_SUM], rax
        int3

        .global RETURN_1
        .global RETURN_2
        .global RETURN_3
```

11.1.4. Version 4: Use stack to spill `rdx` before making any call.

Although we don't use it this would allow us to make nested calls assuming we always used the same register to pass the return address in. In our case `rdx`. This approach effectively uses the stack to spill `rdx` before we use it to make a call. In this way the caller is using the stack to hang on to where it needs to return before making a new call.

CODE: asm - sumit3.s : Version 4 no change use version 3 use stack

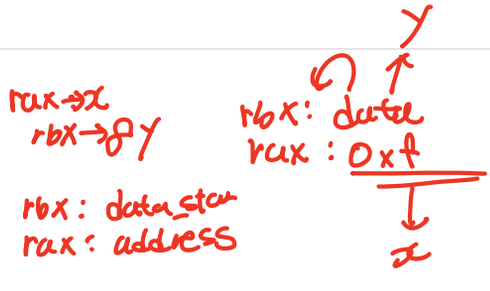
```
        .intel_syntax noprefix

        .section .text

        # tell linker that sumIt symbol can be referenced in other files
        .global sumIt

        # code to sum data in array who's address is in rcx
        # we assume rdx has the address to jump back too
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
sumIt:                          # label that marks where this code begins
        xor  rax, rax           # rax -> sum : sum = 0
        xor  rdi, rdi           # rdi -> i : i = 0

        # code to sum data at value in rcx
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
loop_start:
        cmp  rbx, rdi                       # rbx - rdi
        jz   loop_done                      # if above is zero (they are equal) jump
        add  rax, QWORD PTR [rcx + rdi * 8] # add the i'th value to the sum
        inc  rdi                            # i=i+1
        jmp   loop_start                    # go back to the start of the loop
loop_done:
        jmp rdx                             # jump back to location held in rdx
```



Corresponding: usesum

CODE: asm - usesum4.s : Version 4

```
        .intel_syntax noprefix
        .data
        .comm A_SUM, 8, 8
        .comm A_LEN, 8, 8
        .comm A, 8*1024, 8

        .comm B_SUM, 8, 8
        .comm B_LEN, 8, 8
        .comm B, 8*1024, 8

        .comm C_SUM, 8, 8
        .comm C_LEN, 8, 8
        .comm C, 8*1024, 8

        .text
        .global _start
_start:
        mov rbx, QWORD PTR [A_LEN]
        mov rcx, OFFSET A
        push rdx
        mov rdx, OFFSET RETURN_1
        jmp sumIt
RETURN_1:
        pop rdx
        mov QWORD PTR [A_SUM], rax

        mov rbx, QWORD PTR [B_LEN]
        mov rcx, OFFSET B
        push rdx
        mov rdx, OFFSET RETURN_2
        jmp sumIt
RETURN_2:
        pop rdx
        mov QWORD PTR [B_SUM], rax

        mov rbx, QWORD PTR [C_LEN]
        mov rcx, OFFSET C
        push rdx
        mov rdx, OFFSET RETURN_3
        jmp sumIt
RETURN_3:
        pop rdx
        mov QWORD PTR [C_SUM], rax
        int3

        .global RETURN_1
        .global RETURN_2
        .global RETURN_3
```

11.1.5. Version 5: Use `call` and `ret` instructions to turn our code into an "real" X86 function

CODE: asm - sumit5.s : Version 5

```
        .intel_syntax noprefix

        .section .text

        # tell linker that sumIt symbol can be referenced in other files
        .global sumIt

        # code to sum data in array who's address is in rcx
        # we assume we were started with call so return address on stack
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
sumIt:                          # label that marks where this code begins
        xor  rax, rax           # rax -> sum : sum = 0
        xor  rdi, rdi           # rdi -> i : i = 0

        # code to sum data at value in rcx
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
loop_start:
        cmp  rbx, rdi                       # rbx - rdi
        jz   loop_done                      # if above is zero (they are equal) jump
        add  rax, QWORD PTR [rcx + rdi * 8] # add the i'th value to the sum
        inc  rdi                            # i=i+1
        jmp   loop_start                    # go back to the start of the loop
loop_done:
        ret                                 # use return to pop value off the stack
                                            # and jump to that location
```

Corresponding: usesum

CODE: asm - usesum5.s : Version 5

```
        .intel_syntax noprefix
        .data
        .comm A_SUM, 8, 8
        .comm A_LEN, 8, 8
        .comm A, 8*1024, 8

        .comm B_SUM, 8, 8
        .comm B_LEN, 8, 8
        .comm B, 8*1024, 8

        .comm C_SUM, 8, 8
        .comm C_LEN, 8, 8
        .comm C, 8*1024, 8

        .text
        .global _start
_start:
        mov rbx, QWORD PTR [A_LEN]
        mov rcx, OFFSET A
        call sumIt
        mov QWORD PTR [A_SUM], rax

        mov rbx, QWORD PTR [B_LEN]
        mov rcx, OFFSET B
        call sumIt
        mov QWORD PTR [B_SUM], rax

        mov rbx, QWORD PTR [C_LEN]
        mov rcx, OFFSET C
        call sumIt
        mov QWORD PTR [C_SUM], rax
        int3
```

11.1.6. Version 6 improving code by “spilling” `rdi` in `sumIt`

CODE: asm - sumit6.s : Version 6

```
        .intel_syntax noprefix

        .section .text

        # tell linker that sumIt symbol can be referenced in other files
        .global sumIt

        # code to sum data in array who's address is in rcx
        # we assume we were started with call so return address on stack
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
sumIt:                          # label that marks where this code begins
        xor  rax, rax           # rax -> sum : sum = 0
        push rdi                # spill current value of rdi before we use it
        xor  rdi, rdi           # rdi -> i : i = 0

        # code to sum data in array who's address is in rcx
        # we assume we were started with call so return address on stack
        # we assume rbx has length  rbx -> len
        # and that we will leave final sum in rax
loop_start:
        cmp  rbx, rdi                       # rbx - rdi
        jz   loop_done                      # if above is zero (they are equal) jump
        add  rax, QWORD PTR [rcx + rdi * 8] # add the i'th value to the sum
        inc  rdi                            # i=i+1
        jmp  loop_start                     # go back to the start of the loop
loop_done:
        pop  rdi                            # restore rdi back to its original value
        ret                                 # use return to pop value off the stack
                                            # and jump to that location
```

Corresponding: usesum

CODE: asm - usesum5.s : Version 6 no change to usesum use Version 5

```
        .intel_syntax noprefix
        .data
        .comm A_SUM, 8, 8
        .comm A_LEN, 8, 8
        .comm A, 8*1024, 8

        .comm B_SUM, 8, 8
        .comm B_LEN, 8, 8
        .comm B, 8*1024, 8

        .comm C_SUM, 8, 8
        .comm C_LEN, 8, 8
        .comm C, 8*1024, 8

        .text
        .global _start
_start:
        mov rbx, QWORD PTR [A_LEN]
        mov rcx, OFFSET A
        call sumIt
        mov QWORD PTR [A_SUM], rax

        mov rbx, QWORD PTR [B_LEN]
        mov rcx, OFFSET B
        call sumIt
        mov QWORD PTR [B_SUM], rax

        mov rbx, QWORD PTR [C_LEN]
        mov rcx, OFFSET C
        call sumIt
        mov QWORD PTR [C_SUM], rax
        int3
```