UC-SLS Lecture 18 : Using C to write and organize opcode bytes - Functions

Contents

- create a directory `mkdir cfuncs; cd cfuncs`
- copy 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` to make working in gdb easier
- normally you would want to track everything in git

```
$ ls /home/jovyan/cfuncs
Makefile     main0.c   myadd.c  myfunc0.c  myfunc2.c  setup.gdb
callmyadd.c  main0e.c  myadd.h  myfunc1.c  myfunc3.c
$
```

18.1. Opcodes and C

When we write assembly code we are free to layout our opcodes and use registers in any way we like.

- We can place labels anywhere in our opcodes
- We can specify a jump to any arbitrary location
- While we can use processor support for passing return address via instructions, like `call` and `ret`, we are not required too
- We are not forced to use the registers in any particular way

18.1.1. C Standardizes how to organize and write opcodes

- Its all about standardizing how things are done
  - https://www.uclibc.org/docs/psABI-x86_64.pdf
  - this way code written by different people or tools can inter-operate
  - there are rules they can rely on
- "C" forces us to decompose and organize opcodes into "functions"
  - global label - single entry point
  - block of opcodes ending in a "return"
  - standardizes use of registers
  - standardizes use of stack
    - call frames - automatic storage of locals
  - separation into declaration (many) and definition (one)
  - compiler can get smart and optimize functions and variables away
    - in-line
    - dead-code elimination
    - register only variables

18.1.2. C Standardizes how to organize and write opcodes

Overall summary

- "C" forces us to decompose and organize opcodes into "functions"
- "C" functions:
  - Have a unique global label that identifies a single starting address for the function
    - the label is formed from the function's name which must conform to certain rules
  - Form a contiguous block of memory that does not overlap with another function or data
    - Therefore they have a clear size in bytes that spans their first opcode to the last
  - Are written so that there is a standard way for passing arguments to them
    - a fixed way for passing arguments eg. basics on x86_64
      - arg1 → `rdi`
      - arg2 → `rsi`
      - arg3 → `rdx`
      - arg4 → `r8`
      - arg5 → `r9`
      - rest are pushed on stack in reverse order (arg6 is last to be pushed)
    - a fixed way for passing a return value eg. basics on x86_64
      - return value → `rax`
    - details for cpu and OS are in specification documents eg:
      - https://www.uclibc.org/docs/psABI-x86_64.pdf
  - Execution from a function must end with a return to the next instruction after the call
    - eg on x86_64 `call` and `ret` are used
  - Support local variables that are automatically managed
    - eg. on x86_64 the processor stack is used
      - each call to a function creates a new stack frame
        - each frame represents a call to a function
          - the frame contains a version of the local variables for that call
            - thus each call has its own locals
            - when a function returns to its call the stack frame for the call is popped

        - thus support recursion
- Separates declaration from definition
    - declaration: only specify its name but does not define any opcodes
        - the function declaration is needed to generate the assembly code to call the function
        - given the rules above for how arguments are passed and a unique entry point
            - compiler can generate the assembly code for a call with the declaration
                - does not need the body
            - declarations are placed in "header" files.
            - definitions are placed in "c" files.
    - definition: repeats the declaration but include a body that defines the functions opcodes
        - there can only be one of these
- A compiler is allowed to "inline" a function if certain optimizations are enabled and criteria are met
    - there are times in which the overhead of calling a function is not worth it
        - it is better to just create a version of the opcodes in place where the call is being made
            - inlined

18.1.3. Let's start at the beginning

- we will use the compiler and our ability to read assembly code to learn how "C" works

18.1.3.1. Function Name → global label for its Entry point (start of its opcode)

C:

```
void myfunc(void) {}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc0.c -o
myfunc0.s
```

Assembly:

```
        .file   "myfunc0.c"
        .intel_syntax noprefix
        .text
        .globl  myfunc
        .type   myfunc, @function
myfunc:
        ret
        .size   myfunc, .-myfunc
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

- **function name "myfunc" introduces a global label `myfunc` in the text section**
- **return type prefixes function `void` is the "no" type**
    - **a `void` return type means that the function does not return anything**
- **parenthesis after function name `(` and `)` demarks parameter list**
    - **a `void` in parameter list means function takes no parameters**
- **`{` and `}` demarks body**
    - **a set of statements that will be converted into opcodes**
    - **implicitly every function has at least one statement**
        - `return`
            - **if not written the compiler assumes one**
            - **generates instructions to return to the caller**
                - **X86 :** `ret`

18.1.3.2. Calling a function

C:

```
__attribute__ ((noinline))
int funcA(void) {
  return 7;
}

int funcB(void) {
  return 3 + funcA();
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc1.c -o
myfunc1.s
```

Assembly

```
        .file   "myfunc1.c"
        .intel_syntax noprefix
        .text
        .globl  funcA
        .type   funcA, @function
funcA:
        mov     eax, 7
        ret
        .size   funcA, .-funcA
        .globl  funcB
        .type   funcB, @function
funcB:
        call    funcA
        add     eax, 3
        ret
        .size   funcB, .-funcB
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

- **avoid compiler for optimizing away noinline and use of return value**
- **exactly what we expected in assembly right?**

18.1.4. Interlude : _start vs main()

- **As we have seen the linker marks where execution should begin in our binary via the `_start` symbol**
- **However when we are writing 'C' we normally do not write raw assembly**
  - **all our code is in functions**
- **In the last lectures we wrote our own `_start` in an assembly file that called our C generated assembly**
  - **and linked it by hand avoid all the defaults**
- **However normally we don't do this.**
  - **the C compiler come with some startup code along with the standard C library of functions**
  - **this code is usually in an object files of the name `crt*.o`**
    - **The "c" runtime a bunch of code that runs before the code you write**
      - **runs setup code for you initializing c library and other aspects**
      - **when done calls `main` function passing in some standard parameters**
        - **`argc`, `argv` and on Unix `envp`**
  - **use `-v` to see it get added**
- **So lets write a main and use gcc to link it in the "normal" way**

C:

```c
int main() {
  myfunc();
  return 0;
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel main0e.c -o
main0e.s
main0e.c: In function 'main':
main0e.c:2:3: error: implicit declaration of function 'myfunc' [-Werror=implicit-
function-declaration]
    2 |   myfunc();
      |   ^~~~~~
cc1: all warnings being treated as errors
make[2]: *** [Makefile:34: main0e.s] Error 1
```

What's the fix?

18.1.4.1. Function Declarations vs Definitions

- **When C encounters code that calls another function**
  - **it cannot know how to generate the assembly for the call**
    - **unless it knows**
      - **name**
      - **arguments: type and order**
      - **and return value type**
  - **a function declaration is exactly this - just the signature of the function**
    - **does not generate any code itself just allows calls to be correctly created**
  - **a single definition either in the same file or another that will end up in a .o must exist**
  - **linker will stitch them together**

Add a declaration of `myfunc` that matches its defintion to the `main.c` file

Normally we would put this into a header file eg. myfunc0.h

- **this way any file in which we want to call `myfunc` in we would simply include the header**

C:

```c
void myfunc(void);

int main() {
  myfunc();
  return 0;
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel main0.c -o
main0.s
```

Assembly

```
        .file   "main0.c"
        .intel_syntax noprefix
        .text
        .section        .text.startup,"ax",@progbits
        .globl  main
        .type   main, @function
main:
        push    rax
        call    myfunc
        xor     eax, eax
        pop     rdx
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

```
gcc  -fno-pic -static -Os -g main0.s myfunc0.s -o main0
```

Lets add the verbose flag so that we can see what is reallly going on

```
gcc -v -fno-pic -static -Os -g main0.s myfunc0.s -o main0
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-
1ubuntu1~20.04.1' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-
languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-
major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --
enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-
gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu
--enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-
abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --
enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-
objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-
abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --
enable-offload-targets=nvptx-none=/build/gcc-9-Av3uEd/gcc-9-9.4.0/debian/tmp-
nvptx/usr,hsa --without-cuda-driver --enable-checking=release --build=x86_64-
linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
COLLECT_GCC_OPTIONS='-v' '-fno-pic' '-static' '-Os' '-g' '-o' 'main0' '-
mtune=generic' '-march=x86-64'
 as --gdwarf2 -v --64 -o /tmp/ccHKeGxJ.o main0.s
GNU assembler version 2.34 (x86_64-linux-gnu) using BFD version (GNU Binutils for
Ubuntu) 2.34
COLLECT_GCC_OPTIONS='-v' '-fno-pic' '-static' '-Os' '-g' '-o' 'main0' '-
mtune=generic' '-march=x86-64'
 as --gdwarf2 -v --64 -o /tmp/ccMnEWaL.o myfunc0.s
GNU assembler version 2.34 (x86_64-linux-gnu) using BFD version (GNU Binutils for
Ubuntu) 2.34
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-
gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/9/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86_64-linux-
gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-fno-pic' '-static' '-Os' '-g' '-o' 'main0' '-
mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-linux-
gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper -
plugin-opt=-fresolution=/tmp/ccZKBpMK.res -plugin-opt=-pass-through=-lgcc -plugin-
opt=-pass-through=-lgcc_eh -plugin-opt=-pass-through=-lc --build-id -m elf_x86_64
--hash-style=gnu --as-needed -static -z relro -o main0 /usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/9/crtbeginT.o
-L/usr/lib/gcc/x86_64-linux-gnu/9 -L/usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib
-L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -
L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/9/../../.. /tmp/ccHKeGxJ.o
/tmp/ccMnEWaL.o --start-group -lgcc -lgcc_eh -lc --end-group /usr/lib/gcc/x86_64-
linux-gnu/9/crtend.o /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-
gnu/crtn.o
COLLECT_GCC_OPTIONS='-v' '-fno-pic' '-static' '-Os' '-g' '-o' 'main0' '-
mtune=generic' '-march=x86-64'
```

Much bigger than we might have expected

```
-rwxr-xr-x. 1 jovyan root 853K Sep 29 16:39 main0
```

Why so big?

1. **All the extra stuff**
2. **We have suppressed dynamic link/loading**

Normally we just take all this on faith ... but since we know how now let's look at it at least just this once at what all this stuff is

```
objdump -d main0 > main0.dis
```

Edit

18.1.5. Variables declared in body of function are function local

- **If the compiler needs memory for a local variable then it adds it to stack frame for call**

  C:

```c
__attribute__ ((noinline))
void myfunc2(long long *i)
{
  *i += 1;
}

long long myfunc(void) {
  long long i=(long long)&myfunc;
  myfunc2(&i);
  return i;
}
```

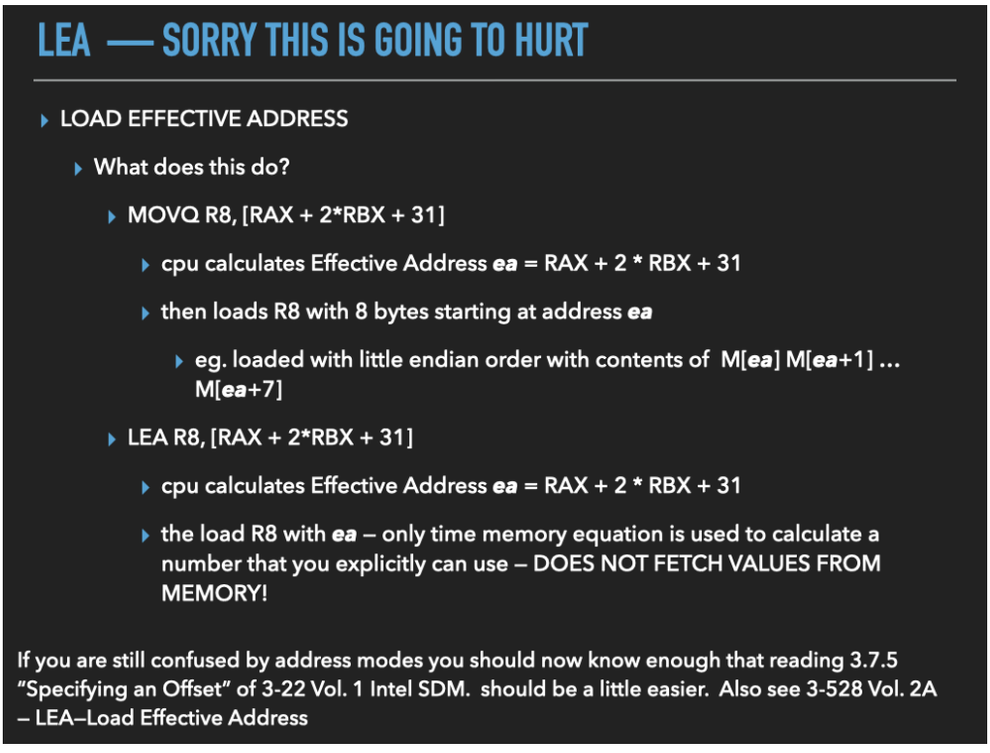
```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc2.c -o
myfunc2.s
```

Assembly

```
        .file   "myfunc2.c"
        .intel_syntax noprefix
        .text
        .globl  myfunc2
        .type   myfunc2, @function
myfunc2:
        inc     QWORD PTR [rdi]
        ret
        .size   myfunc2, .-myfunc2
        .globl  myfunc
        .type   myfunc, @function
myfunc:
        sub     rsp, 16
        mov     QWORD PTR [rsp+8], OFFSET FLAT:myfunc
        lea     rdi, [rsp+8]
        call    myfunc2
        mov     rax, QWORD PTR [rsp+8]
        add     rsp, 16
        ret
        .size   myfunc, .-myfunc
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

- **must play some games to get the compiler to create a local with such simple functions**
- **notice what this code is doing?**
- `sub rsp, 16` **what is this?**
- **what is** `rsp + 8`
- **Local variables have not fixed location in memory!**
- **PS this code is dangerous and not something you would normally do**
  - **but C lets you cheat if you want too!**

18.1.6. LEA



18.1.7. Passing arguments and simple return value

C:

```c
__attribute__ ((noinline))
int func2(int x, int y)
{
  return x + y;
}

int func1(int x)
{
  return func2(x,2);
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc3.c -o
myfunc3.s
```

Assembly

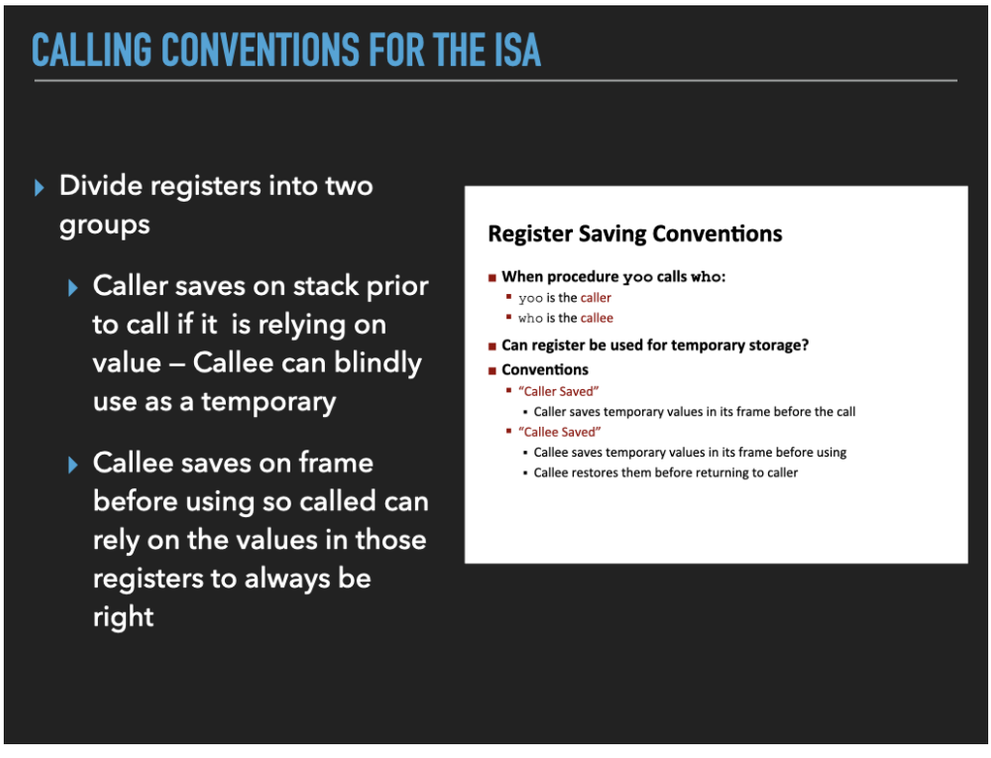
```
        .file   "myfunc3.c"
        .intel_syntax noprefix
        .text
        .globl  func2
        .type   func2, @function
func2:
        lea     eax, [rdi+rsi]
        ret
        .size   func2, .-func2
        .globl  func1
        .type   func1, @function
func1:
        mov     esi, 2
        jmp     func2
        .size   func1, .-func1
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

- **What is `lea` being used for here?**
- **Why `eax`, `rdi` and `rsi` (`esi`) being used the way they are?**

LEA

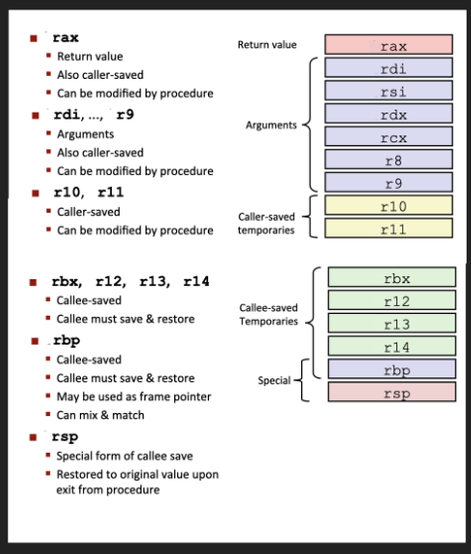
- **The compiler is smart and figure out that it can use `lea` for arbitrary math that does not require flags update**

Why these Register?



CALLING CONVENTIONS FOR THE ISA

- Divide registers into two groups
  - Caller saves on stack prior to call if it is relying on value – Callee can blindly use as a temporary
  - Callee saves on frame before using so called can rely on the values in those registers to always be right
- Arguments:
  - **Diane Sips Delicious Coffee 8 out of 9 times**

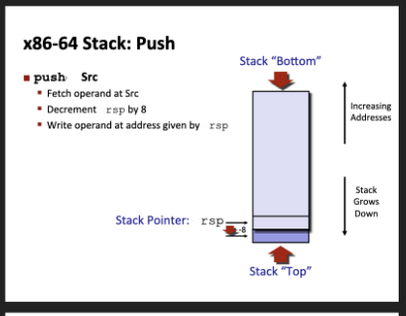


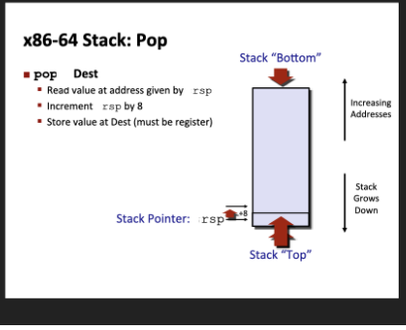
But remember the real truth or the rules for any given CPU and OS is in a standard somewhere
https://www.uclibc.org/docs/psABI-x86_64.pdf

- **3.2.3 Parameter Passing p17**
- **Figure 3.4: Register Usage p21**

FUNCTIONS AND MEMORY FOR LOCAL VARIABLES — THE STACK AND FRAMES

- Using call and ret already make use of the stack so that we can transfer control and return back to and from a function
- C more formally defines how to use the stack for functions





FUNCTIONS AND MEMORY FOR LOCAL VARIABLES — THE STACK AND FRAMES

- Using call and ret already make use of the stack so that we can transfer control and return back to and from a function
- C more formally defines how to use the stack for functions – Stack oriented languages

Stack-Based Languages

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

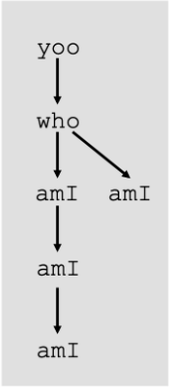
Call Chain Example

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

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

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

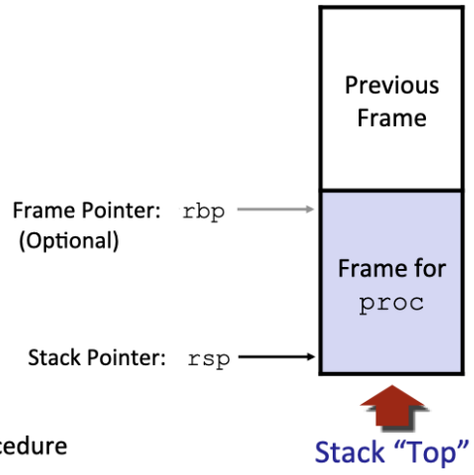
Example Call Chain



Procedure `amI()` is recursive

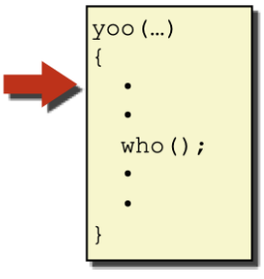
Stack Frames

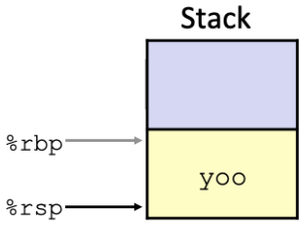
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)



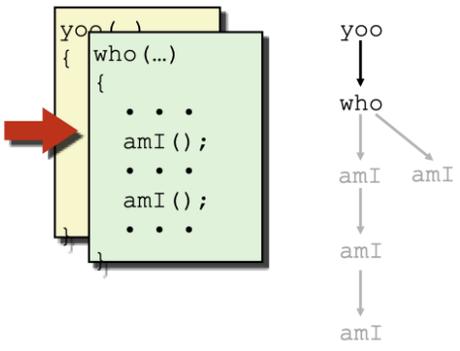
- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction

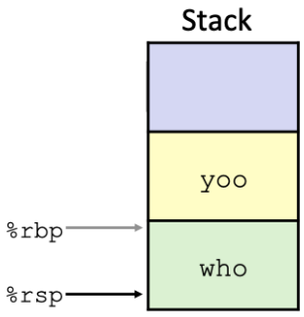
Example



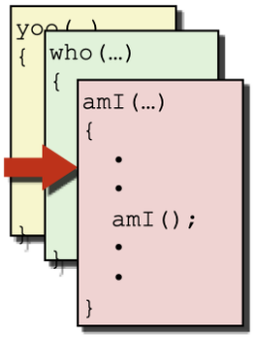


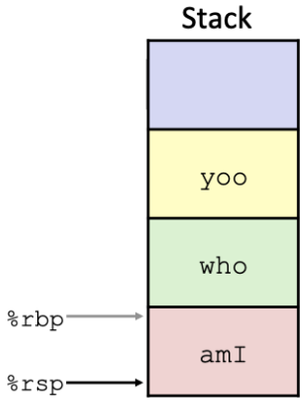
Example



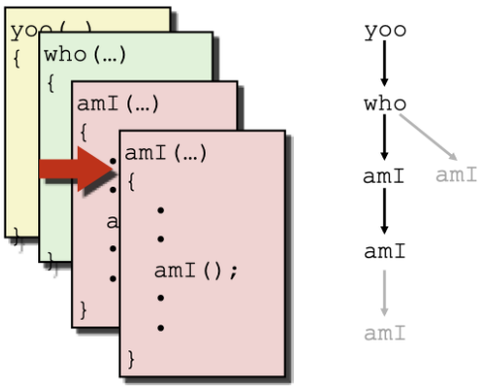


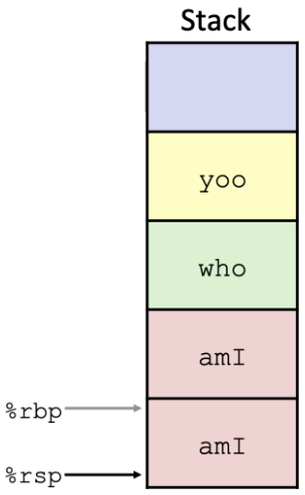
Example



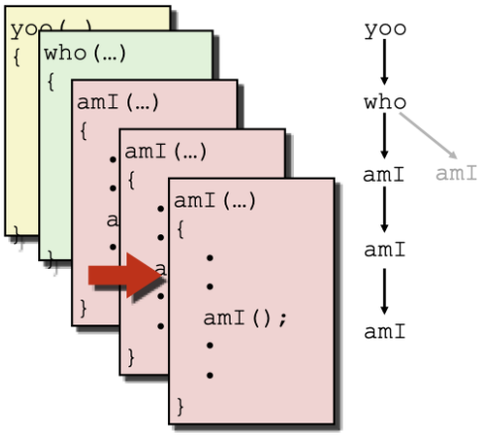


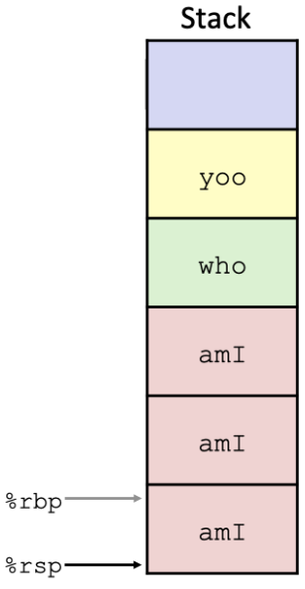
Example



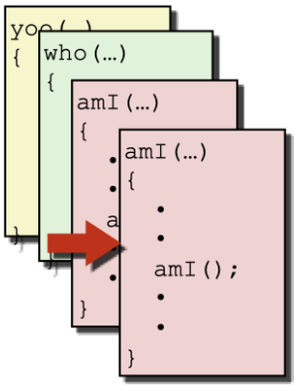


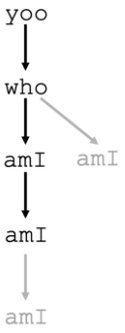
Example

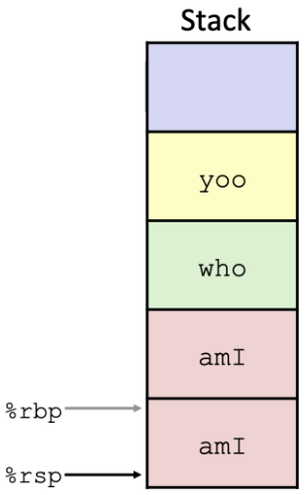




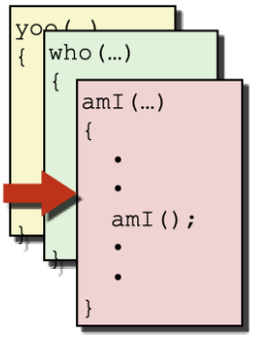
Example

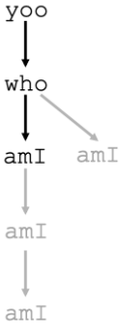


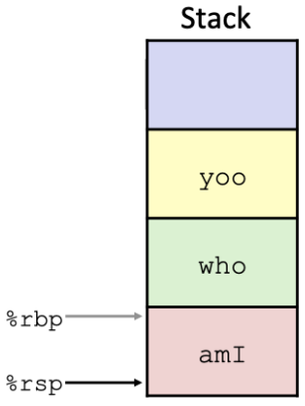




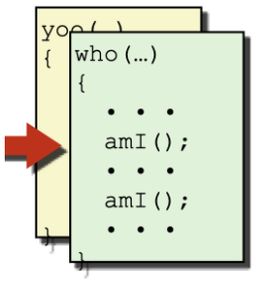
Example

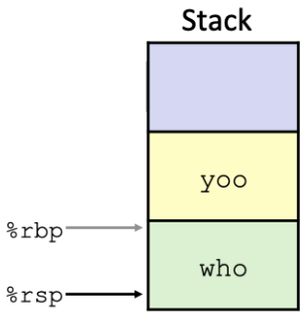




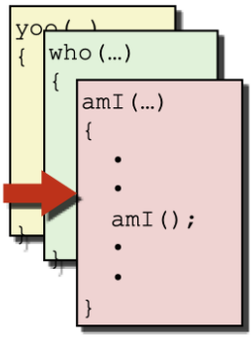


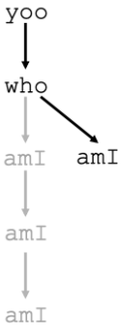
Example

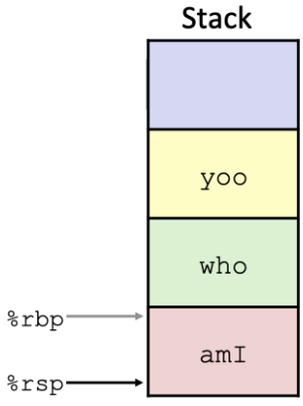




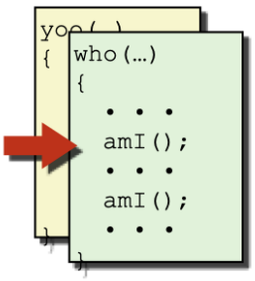
Example

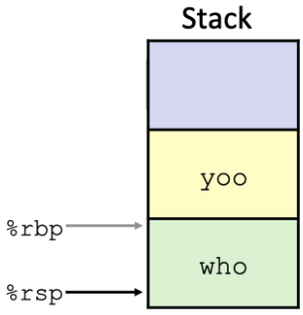




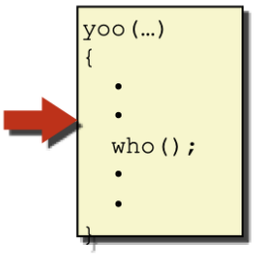


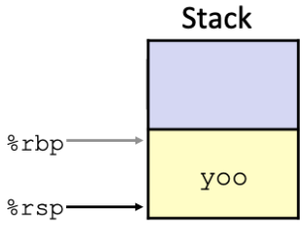
Example





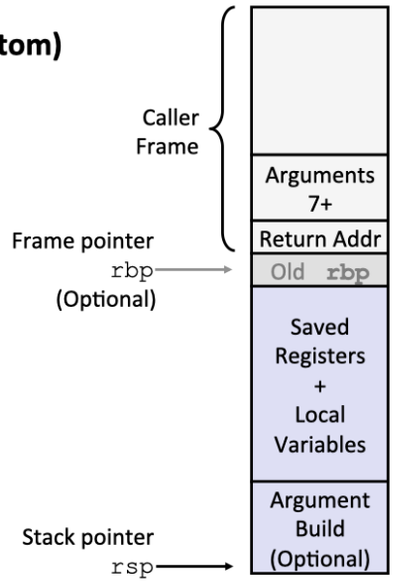
Example





x86-64/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



18.2. An example: Trace this code and visualize the stack

C: myadd.c

```c
long myadd(long *x_ptr, long val)
{
  long x = *x_ptr;
  long y = x + val;
  *x_ptr = y;
  return x;
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myadd.c -o
myadd.s
```

Assembly

```
        .file   "myadd.c"
        .intel_syntax noprefix
        .text
        .globl  myadd
        .type   myadd, @function
myadd:
        mov     rax, QWORD PTR [rdi]
        add     rsi, rax
        mov     QWORD PTR [rdi], rsi
        ret
        .size   myadd, .-myadd
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

C: myadd.h

```c
#ifndef __MY_ADD_H__
#define __MY_ADD_H__

long myadd(long *x_ptr, long val);

#endif // __MY_ADD_H__
```

C: callmyadd.c

```c
#include "myadd.h"

long call_myadd(void) {
  long a = 15214;
  long b = myadd(&a, 5001);
  return a+b;
}

long
main() {
  return call_myadd();
}
```

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel callmyadd.c -o
callmyadd.s
```

Assembly

```
        .file   "callmyadd.c"
        .intel_syntax noprefix
        .text
        .globl  call_myadd
        .type   call_myadd, @function
call_myadd:
        sub     rsp, 24
        mov     esi, 5001
        mov     QWORD PTR [rsp+8], 15214
        lea     rdi, [rsp+8]
        call    myadd
        add     rax, QWORD PTR [rsp+8]
        add     rsp, 24
        ret
        .size   call_myadd, .-call_myadd
        .section        .text.startup,"ax",@progbits
        .globl  main
        .type   main, @function
main:
        jmp     call_myadd
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

By Jonathan Appavoo