CSO-1

X86 Assembly

Daniel G. Graham PhD

University of Virginia | ENGINEERING

Contents

WHY THE MOVE TO EDI

hello.c

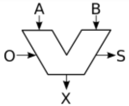
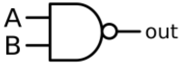
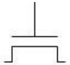
```c
/*
 *Written by Daniel Graham as a class demo
 */

#include <stdio.h>
int main(){
  puts("Hello World");
  return 0;
}
```

hello.s

```asm
  .text
  .file "hello.c"
  .globl  main                              # -- Begin function main
  .type main,@function
main:                                      # @main
  pushq %rax
  movl  $.L.str, %edi
  callq puts
  xorl  %eax, %eax
  popq  %rcx
  retq
.Lfunc_end0:
  .size main, .Lfunc_end0-main
  .cfi_endproc
                                           # -- End function
  .type .L.str,@object                     # @.str
  .section  .rodata.str1.1,"aMS",@progbits,1
.L.str:
  .asciz  "Hello World"
  .size .L.str, 12
```

NORMAL hello.c dgg6b 22%

Application Software	C
Operating system	Linux
Architecture	Risc-V
Micro Architecture 	Data path, Stages
Gates 	Nand, NOR, NOT ..
Devices 	Field Effect Transistors
Physics	Electrons

```
dgg6b@portal02:~/CS01/Assemble/lab$ clang -c stackExamplePart1.s -o stackExamplePart1.o
dgg6b@portal02:~/CS01/Assemble/lab$ ls
debugExample.o  registerExample.s    stackExamplePart1.s
debugExample.s  stackExamplePart1.o  stackExamplePart2.s
dgg6b@portal02:~/CS01/Assemble/lab$ objdump -D stackExamplePart1.o

stackExamplePart1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   6a 04                   push   $0x4
   2:   6a 05                   push   $0x5
   4:   58                      pop    %rax
   5:   5b                      pop    %rbx
   6:   48 01 c3                add    %rax,%rbx
   9:   53                      push   %rbx
dgg6b@portal02:~/CS01/Assemble/lab$
```

Notice the hex machine processes instructions just like our toy ISA

Also notice how the address in memory increases based on the size of the instruction

objdump – tool that allows us to inspect the object file

-D, --disassemble-all Display assembler contents of all sections

X86 OPCODE LOOKUP

http://ref.x86asm.net/coder32.html#x68

https://inst.eecs.berkeley.edu/~cs61c/fa18/img/riscvcard.pdf

WE'LL USE X86 AT&T SYNTAX AS OUR CASE STUDY FOR LOOKING AT

AN ASSEMBLY LANGUAGE

16 REGISTERS

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

OUR WORKING EXAMPLE

```
//callee
int add(int x, int y){
    int result = x + y;
    return result;
}

//caller
int main(){
    return add(2, 3);
}
```

The caller – The function that called another function

The callee – the function being called

```c
//callee
int add(int x, int y){
    int result = x + y;
    return result;
}

//caller
int main(){
    return add(2, 3);
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
    pop %rbp
    pop %rbx
    ret
main:
    movq $3, %rsi
    movq $2, %rdi
    call add(int, int)
    ret
```

Why not just push all the registers?

INSIGHT (ALSO EASIER FOR COMPILATION)

Well instead of pushing everything on the stack. Why don't set some registers as caller saved so that callee can use the registers without having to push them?

C LANGUAGE CALLING CONVENTION

The calling convention is broken into two sets of rules.

1. The first set of rules is employed by the caller of the subroutine (function)
2. The second set of rules is observed by the writer of the subroutine/function (the “callee”)

16 REGISTERS

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

REGISTERS (STACK POINTER)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

THE CALLER

```
//caller
int main(){
    return add(2, 3);
}
```

CALLER RULES

Rule 1. The caller should save the content of the register that is designated as the caller saved register

REGISTERS (CALLER SAVED)

%rax	Return value	%r8	Arguments #5
%rbx	Callee saved	%r9	Arguments #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

CALLER RULES

Rule 1. The caller should save the content of the register that is designated as the caller saved register

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

WHY THE MOVE TO EDI

hello.c

```c
/*
 *Written by Daniel Graham as a class demo
 */

#include <stdio.h>
int main(){
  puts("Hello World");
  return 0;
}
```

hello.s

```
  .text
  .file "hello.c"
  .globl  main                    # -- Begin function main
  .type main,@function
main:                             # @main
  pushq %rax
  movl  $.L.str, %edi
  callq puts
  xorl  %eax, %eax
  popq  %rcx
  retq
.Lfunc_end0:
  .size main, .Lfunc_end0-main
  .cfi_endproc
                                  # -- End function
  .type .L.str,@object            # @.str
  .section  .rodata.str1.1,"aMS",@progbits,1
.L.str:
  .asciz  "Hello World"
  .size .L.str, 12
```

NORMAL hello.c dgg6b 22%

REGISTERS (CALLER SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

THE CALLER

```
//caller
int main(){
    return add(2, 3);
}
```

```
main:
    movq $3, %rsi
    movq $2, %rdi
```

CALLER RULES

Rule 1. The caller should save the content of the register that is designated as the caller saved register

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

Rule 3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Run the subroutine instruction.

THE CALLER

```c
//caller
int main(){
    return add(2, 3);
}
```

```
main:
    movq $3, %rsi
    movq $2, %rdi
    call add(int, int)
    ret
```

Rule 1. The caller should save the content of the register that is designated as the caller saved register

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

Rule 3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Run the subroutine instruction

Rule 4. After the subroutine returns, (i.e. immediately following the call instruction) the caller must remove any additional parameters (beyond the six stored in registers) from the stack. This restores the stack to its state before the call was performed

Rule 5. The caller can expect to find the subroutine's return value in the register RAX.

Rule 6. The caller restores the contents of caller-saved registers (r10, r11, and any in the parameter passing registers) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

REGISTERS (CALLER SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

THE CALLER

```c
//caller
int main(){
    return add(2, 3);
}
```

```asm
main:
    movq $3, %rsi
    movq $2, %rdi
    call add(int, int)
    ret
```

RSI and RDI are caller saved. Why didn't the compiler bother to push and pop them from the stack?

CALLEE

```
//callee
int add(int x, int y){
    int result = x + y;
    return result;
}
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

EXAMPLE OF ALLOCATING LOCAL VARIABLES

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
--snip--
```

Why doesn't the compiler have to allocate any space result variable?
(This return about where the return is stored)

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

REGISTERS (CALLEE SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    --snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack.

Run the subroutine instruction .

UNIVERSITY of VIRGINIA | ENGINEERING

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    --snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the subroutine instructions

Rule 3. When the function is done, the return value for the function should be placed in RAX

SAVING REGISTERS

```c
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
--snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the subroutine instruction

Rule 3. When the function is done, the return value for the function should be placed in RAX

Rule 4. The function must restore the old values of any callee-saved registers (RBX, RBP, and R12 through R15) that were modified. The registers should be popped in the inverse order that they were pushed.

REGISTERS (CALLEE SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
    pop %rbp
    pop %rbx
    --snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the subroutine instruction .

Rule 3. When the function is done, the return value for the function should be placed in RAX

Rule 4. The function must restore the old values of any callee-saved registers (RBX, RBP, and R12 through R15) that were modified. The registers should be popped in the inverse order that they were pushed.

Rule 5. Next, we deallocate local variables. By subtracting from RSP

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the subroutine instruction

Rule 3. When the function is done, the return value for the function should be placed in RAX

Rule 4. The function must restore the old values of any callee-saved registers (RBX, RBP, and R12 through R15) that were modified. The registers should be popped in the inverse order that they were pushed.

Rule 5. Next, we deallocate local variables. By subtracting from RSP

Rule 6. Execute the `ret` instruction.

SAVING REGISTERS

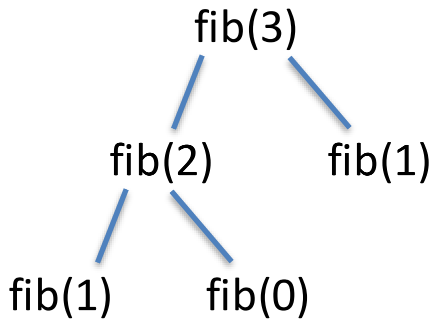
```c
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
     push %rbx
     push %rbp
     movq %rdi, %rbx
     movq %rsi, %rbp
     addq %rbx, %rbp
     movq %rbp, %rax
     pop %rbp
     pop %rbx
     ret
```

FIBONACCI RECURSIVE

```
int fib(int n){
    if (n == 0){
        return 0;
    }
     if (n == 1){
        return 1;
     }
     return fib(n-1) + fib(n-2);
}
```

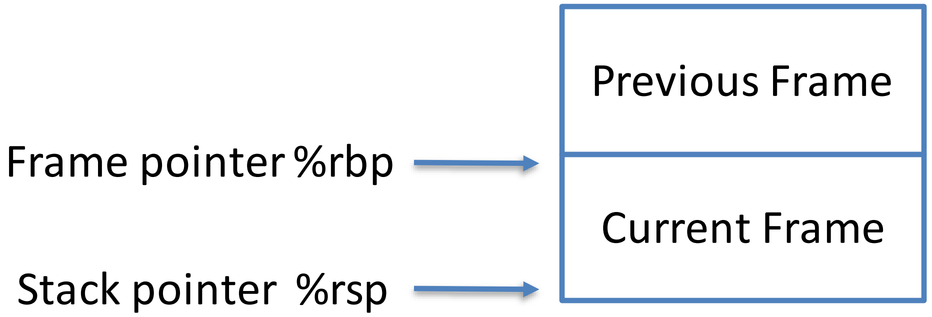
Let's think about the call tree for fib (3)



STACK FRAMES

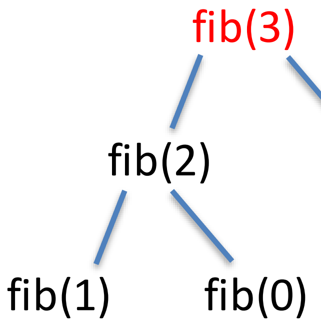
Contains:

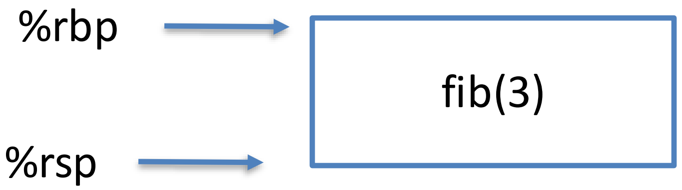
1. Local storage of variables (optional)
2. Temporary space (optional)
3. return address



FIBONACCI RECURSIVE

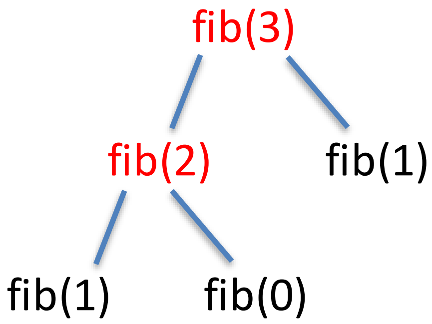
Let's think about the call tree for fib (3)

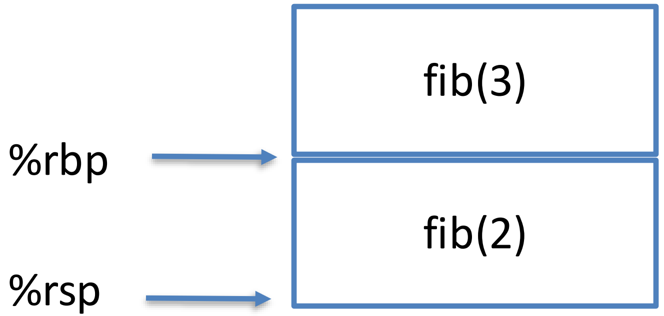




FIBONACCI RECURSIVE

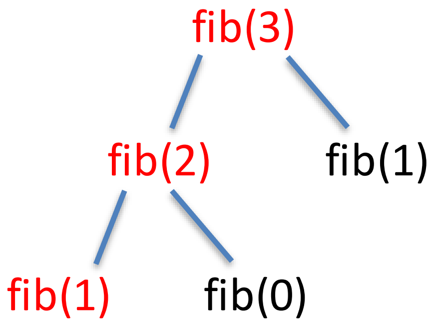
Let's think about the call tree for fib (3)

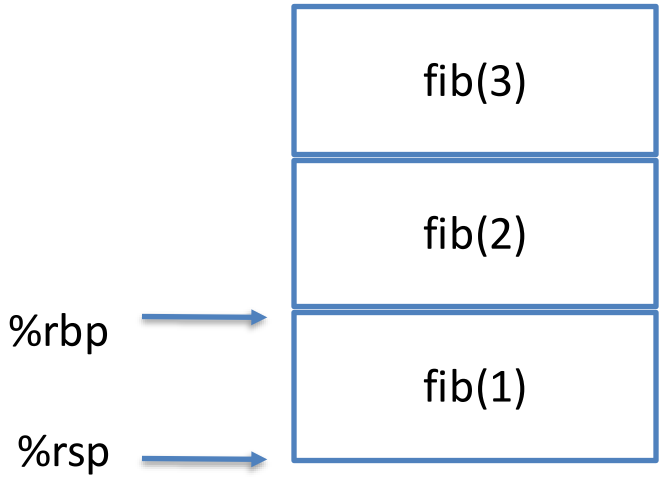




FIBONACCI RECURSIVE

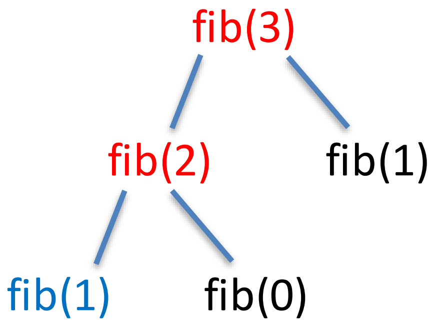
Let's think about the call tree for fib (3)

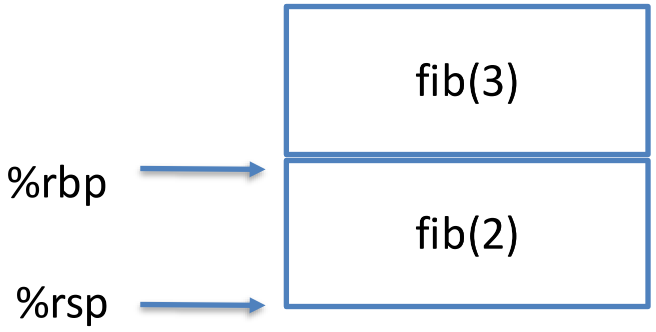




FIBONACCI RECURSIVE

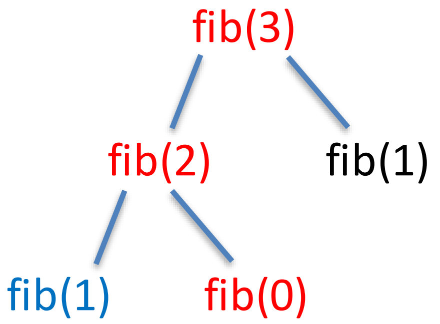
Let's think about the call tree for fib (3)

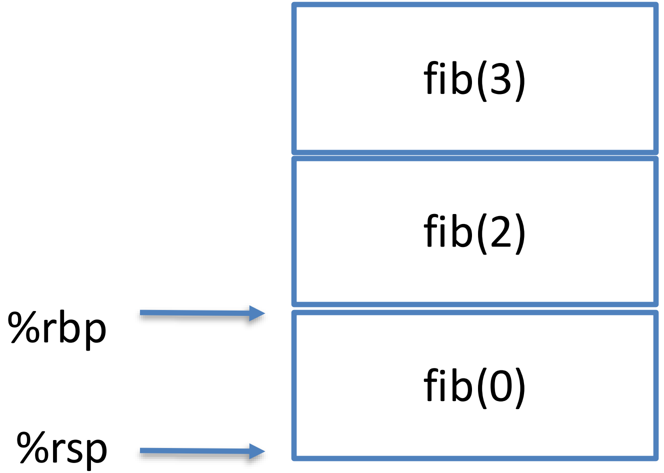




FIBONACCI RECURSIVE

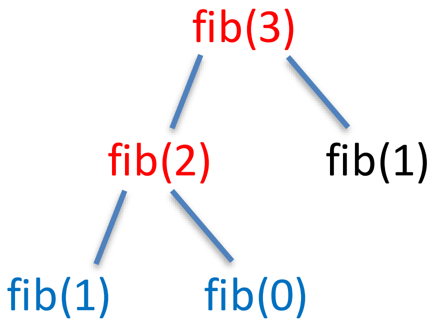
Let's think about the call tree for fib (3)

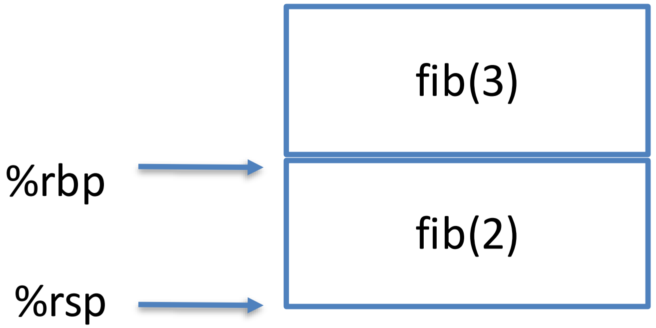




FIBONACCI RECURSIVE

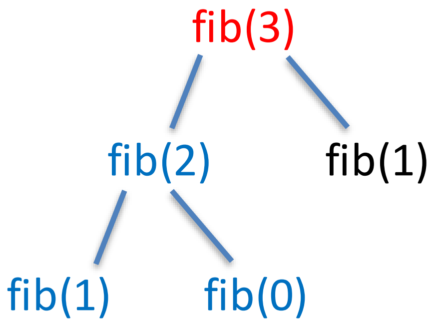
Let's think about the call tree for fib (3)

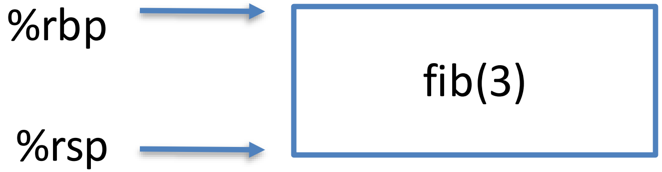




FIBONACCI RECURSIVE

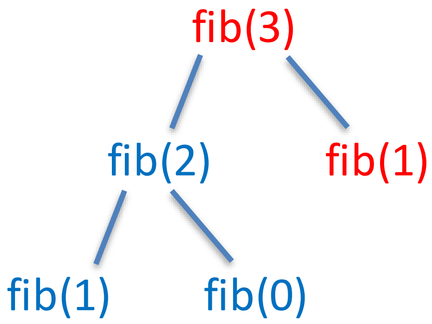
Let's think about the call tree for fib (3)

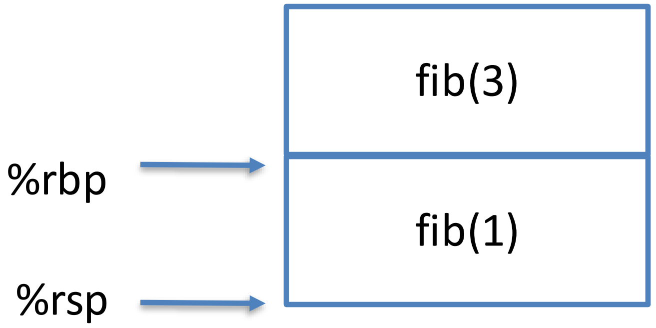




FIBONACCI RECURSIVE

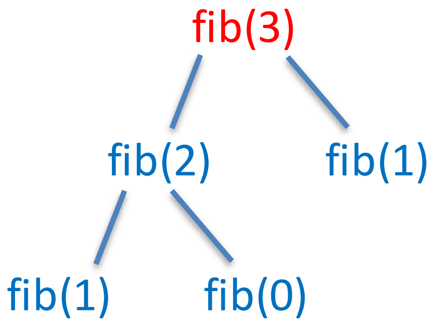
Let's think about the call tree for fib (3)

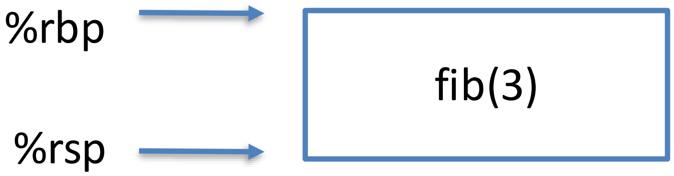




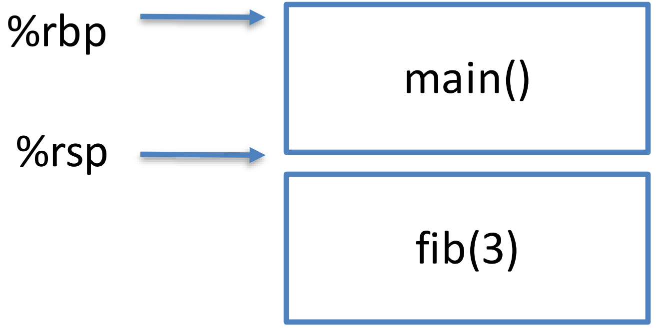
FIBONACCI RECURSIVE

Let's think about the call tree for fib (3)

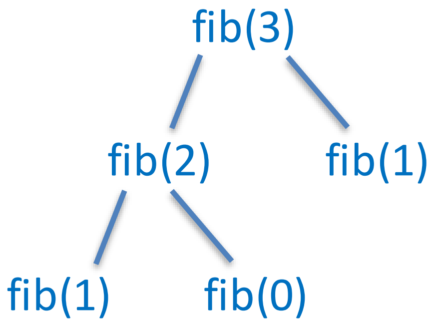




FIBONACCI RECURSIVE



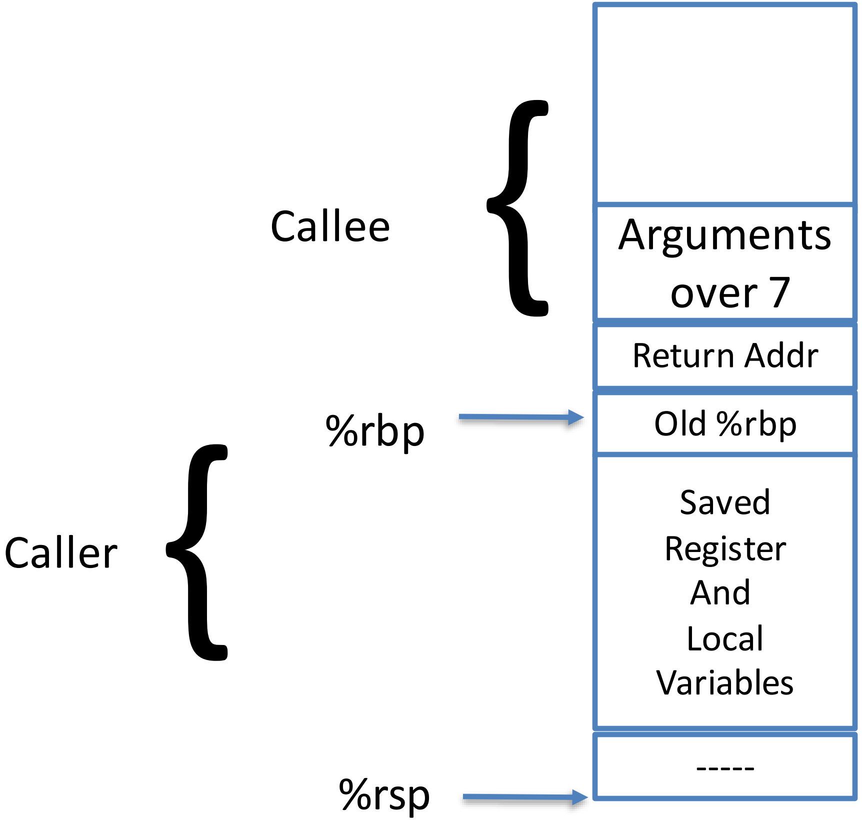
Let's think about the call tree for fib (3)



DETAIL LOOK AT THE FRAME

%rbp is optional

- You'll see when we look at the optimized examples



CALLEE'S PROLOGUE AND EPILOGUE:

Sometimes you will see the following callee prologue and epilogue added the beginning and end of the function

push rbp; *at the start of the callee (prologue)* **mov** rbp, rsp …
pop rbp; *just before the ending ' ret ' (epilogue)*

This code is unnecessary and is a hold-over from the 32-bit calling convention. You can tell the compiler to not include this code by invoking it with the `-fomit-frame-pointer` flag.

NEXT TIME

Swap Example with Mov instruction

Swap Example with lea (load effective address) instruction.

Later:

- jmp instruction and condition codes (Building loops)
- switch statements.