Calling Convention

Abhijit A M

The need for calling convention

An essential task of the compiler

- Generates object code (file) for given source code (file)
- Processors provide simple features
  - Registers, machine instructions (add, mov, jmp, call, etc.), imp registers like stack-pointer, etc; ability to do byte/word sized operations
  - No notion of data types, functions, variables, etc.
- But languages like C provide high level features
  - Data types, variables, functions, recursion, etc
- Compiler needs to map the features of C into processor's features, and then generate machine code
- In reality, the language designers design a language feature only after answering the question of conversion into machine code

The need for calling convention

Examples of some of the challenges before the compiler

- "call" + "ret" does not make a C-function call!
  - A "call" instruction in processor simply does this
    - Pushes IP(that is PC) on stack + Jumps to given address
  - This is not like calling a C-function !
  - Unsolved problem: How to handle parameters, return value?
- Processor does not understand data types!
  - Although it has instructions for byte, word sized data and can differentiate between integers and reals (mov, movw, addl, addf, etc.)

Compiler and Machine code generation

Example, code inside a function

```
int a, b, c;
c = a + b;
```

What kind of code is generated by compiler for this?

```
sub  12, <esp>  #normally local variables are located on stack, make space
mov <location of a in memory>, r1 #location is on stack, e,g. -4(esp)
mov <location of b in memory>, r2
add r1, r2 # result in r1
mov r1, <location of c in memory>
```

Compiler and Machine code generation

Across function calls

```
int f(int m, n) {
    int x = m, y = n;
    return g(x, y);
}
int x(int a) {
    return g (a, a+ 1);
}
int g(int p, int q) {
    p = p * q + p;
    return p
}
```

g() may be called from f() or from x()

Sequence of function calls can NOT be predicted by compiler

Compiler has to generate machine code for each function assuming nothing about the caller

Compiler and Machine code generation

Machine code generation for functions

Mapping C language features to existing machine code instructions.

Typical examples

a =100 ; ==> mov instruction

a = b+ c; ==> mov, add instructions

while(a < 5) { j++; } ==> mov, cmp, jlt, add, etc. Instruction

Where are the local variables in memory?

The only way to store them is on a stack.

Why?

Function calls

LIFO

- Last in First Out
- Must need a “stack” like feature to implement them

Processor Stack

- Processors provide a stack pointer
- %esp on x86
  - Instructions like push and pop are provided by hardware
  - They automatically increment/decrement the stack pointer. On x86 stack grows downwards (substract from esp!)
  - Unlike a “stack data type” data structure, this “stack” is simply implemented with only the esp (as the “top”). The entire memory can be treated as the “array”.

Function calls

System stack, compilers, Languages

- Processor provides us with esp (stack) and push/pop instructions.
- The esp pointer is initialized to a proper value at the time of fork-exec by the OS for each process. Then process runs.
- Knowing the above, compilers go ahead with generating machine code using the *esp*.
- This means, Langauges like C which provide for function calls, and recursion can only run on processors which support a system stack.

Convention needed

- How to use the stack for effective implementation of function calls ?

What goes on stack?

- Local variables
- Function Parameters
- Return address of instruction which called a function !

Activation Record

Local Vars + parameters + return address

When functions call each other

- One activation record is built on stack for each function call
- On function return, the record is destroyed

On x86

- ebp and esp pointers are used to denote the activation record.
- How? We will see soon. You may start exploring with “gcc -S” output assembly code.

X86 instructions

leave

Equivalent to

```
mov   %ebp, %esp     # esp = ebp
pop   %ebp
```

ret

Equivalent to

```
pop %ecx
Jmp %ecx
```

call x

Equivalent to

```
push %eip
jmp  x
```

X86 instructions

endbr64

Normally a NOP

Let’s see some examples now

Let’s compile using

gcc -S

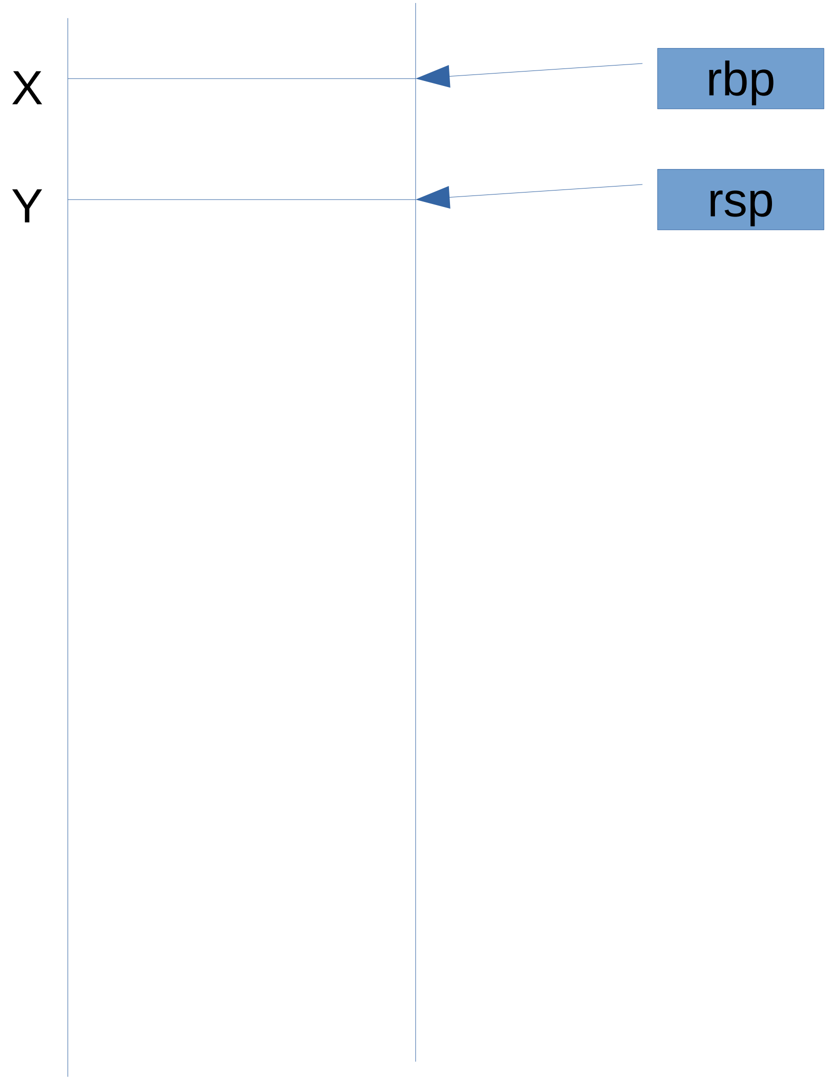
See code and understand

simple.c and simple.s

```c
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

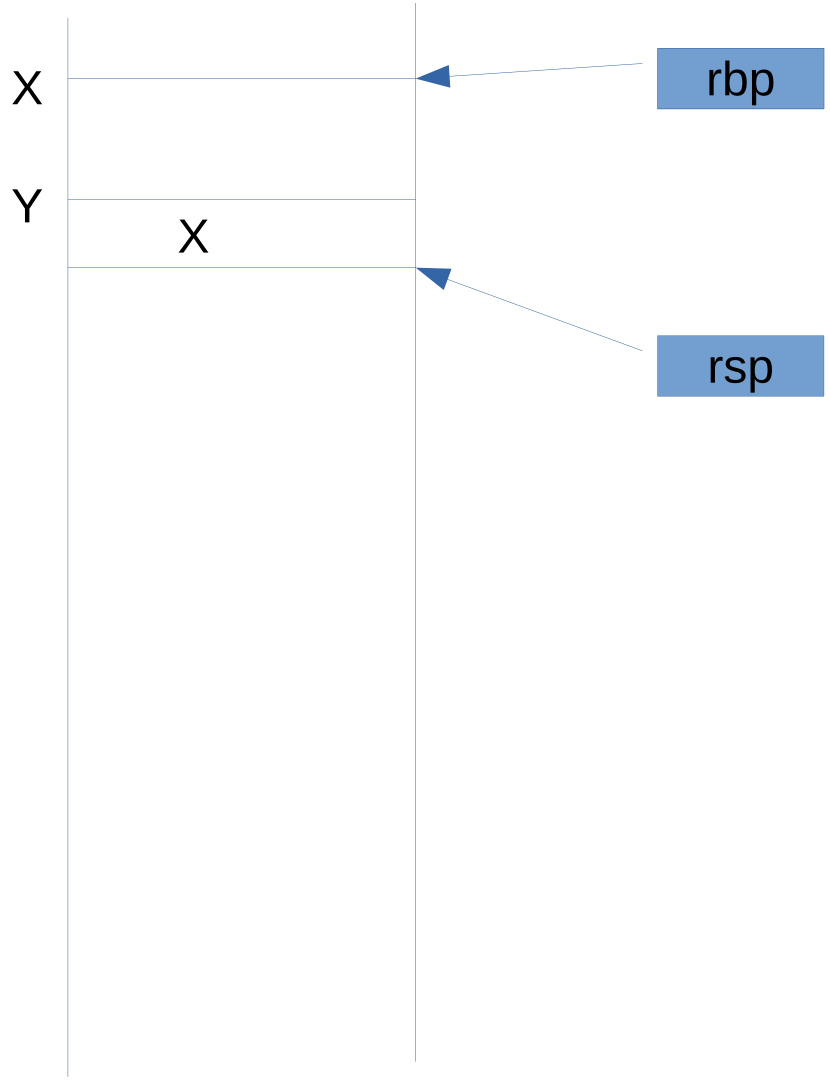
```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```



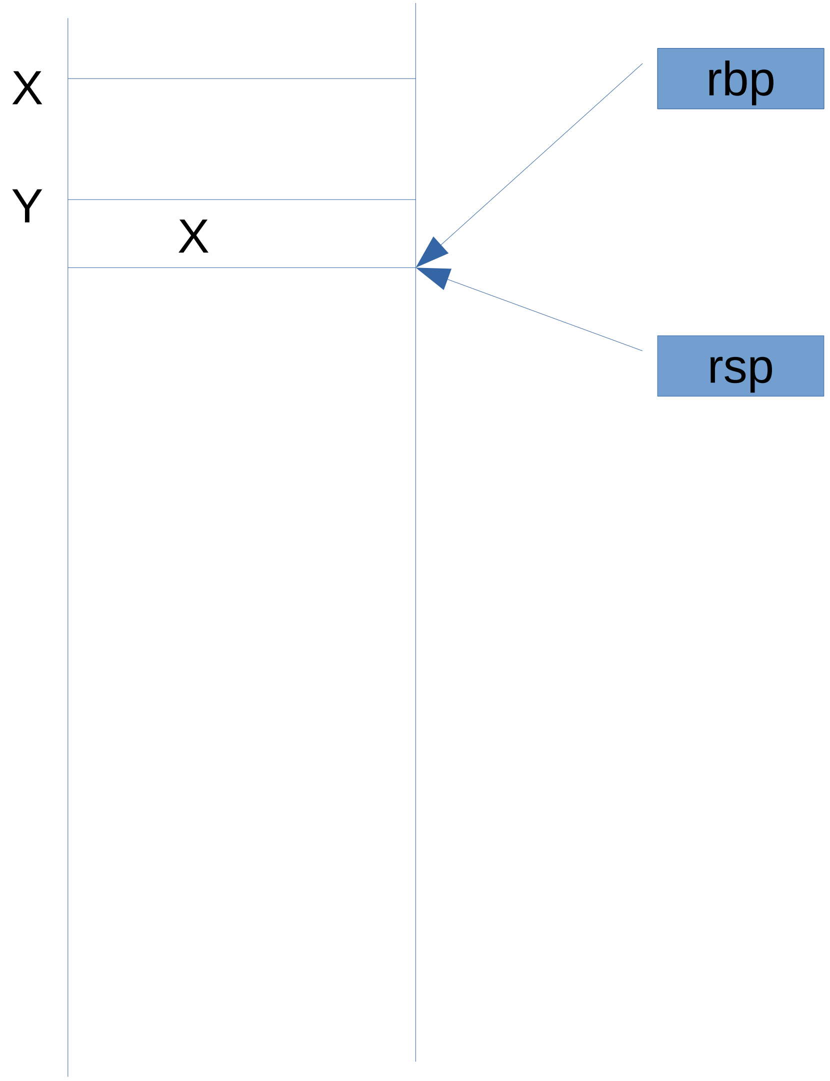

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



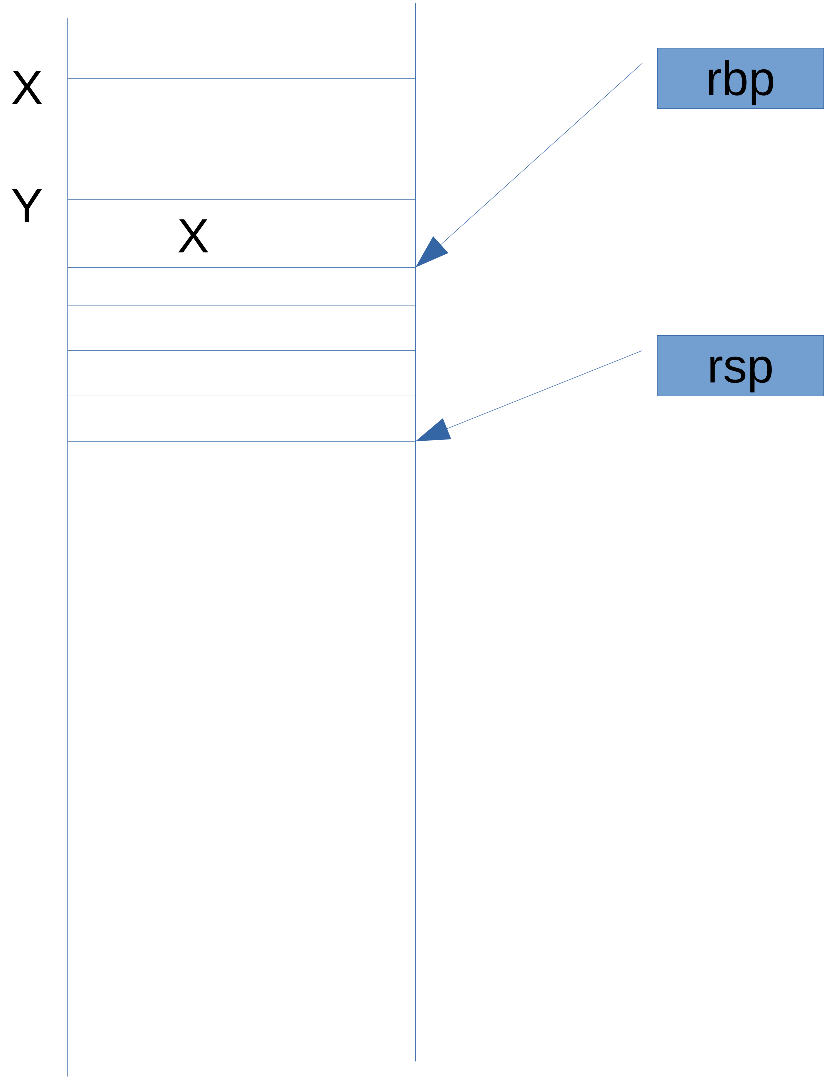

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



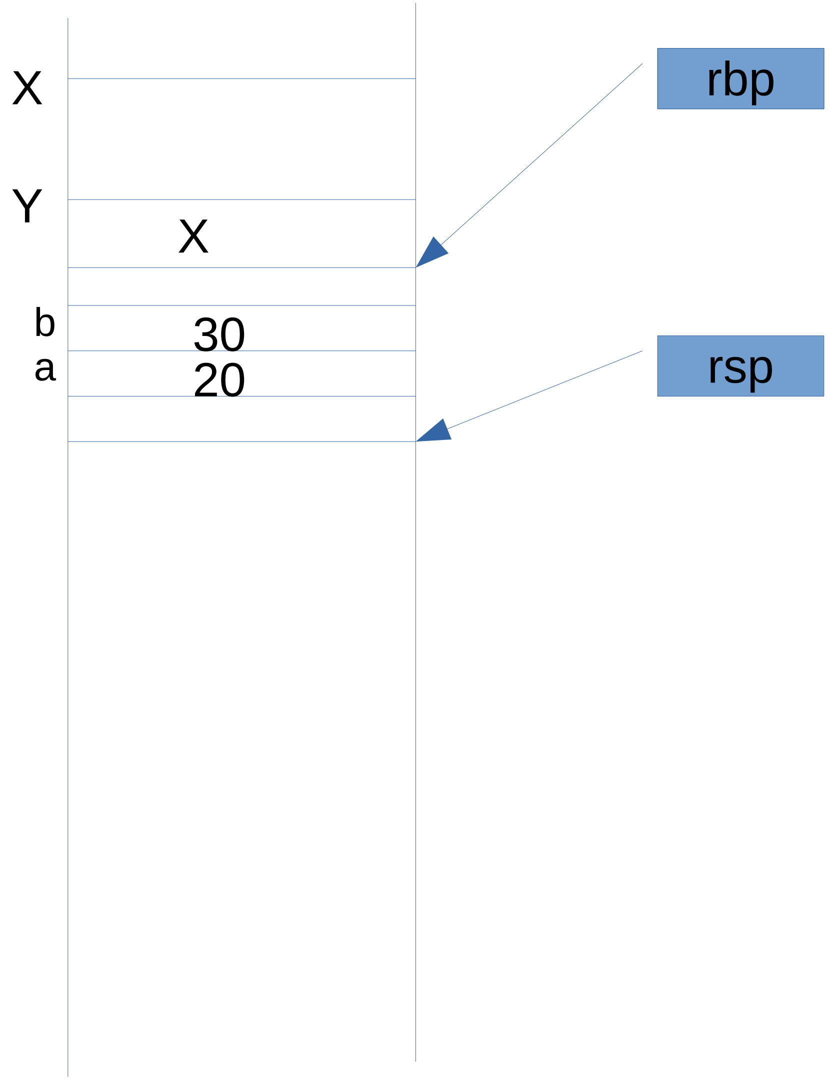

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



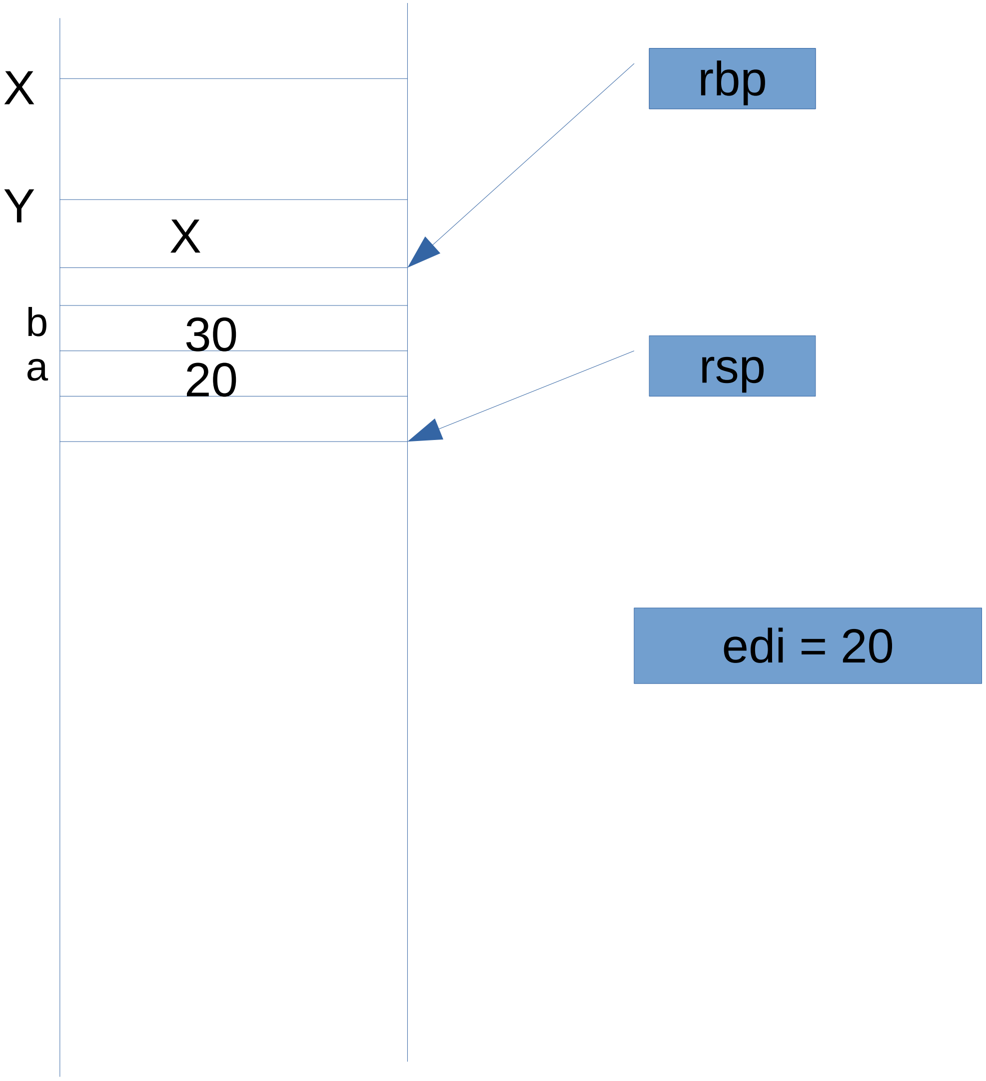

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



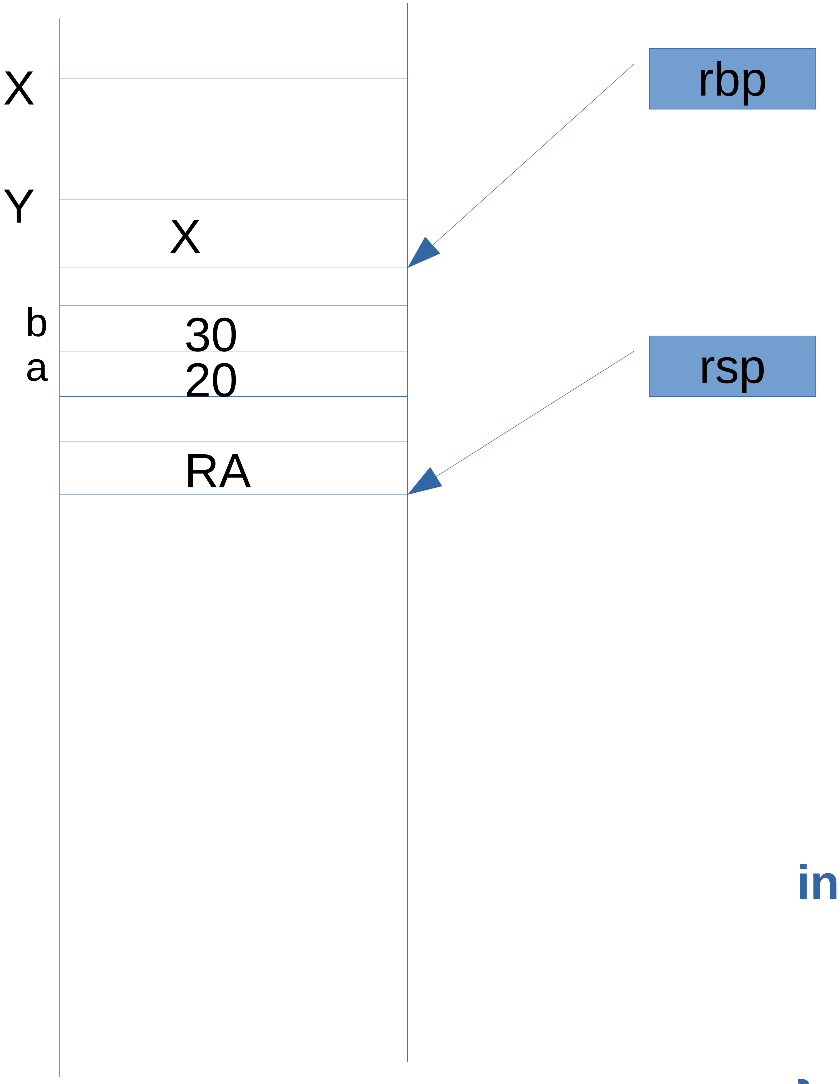

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



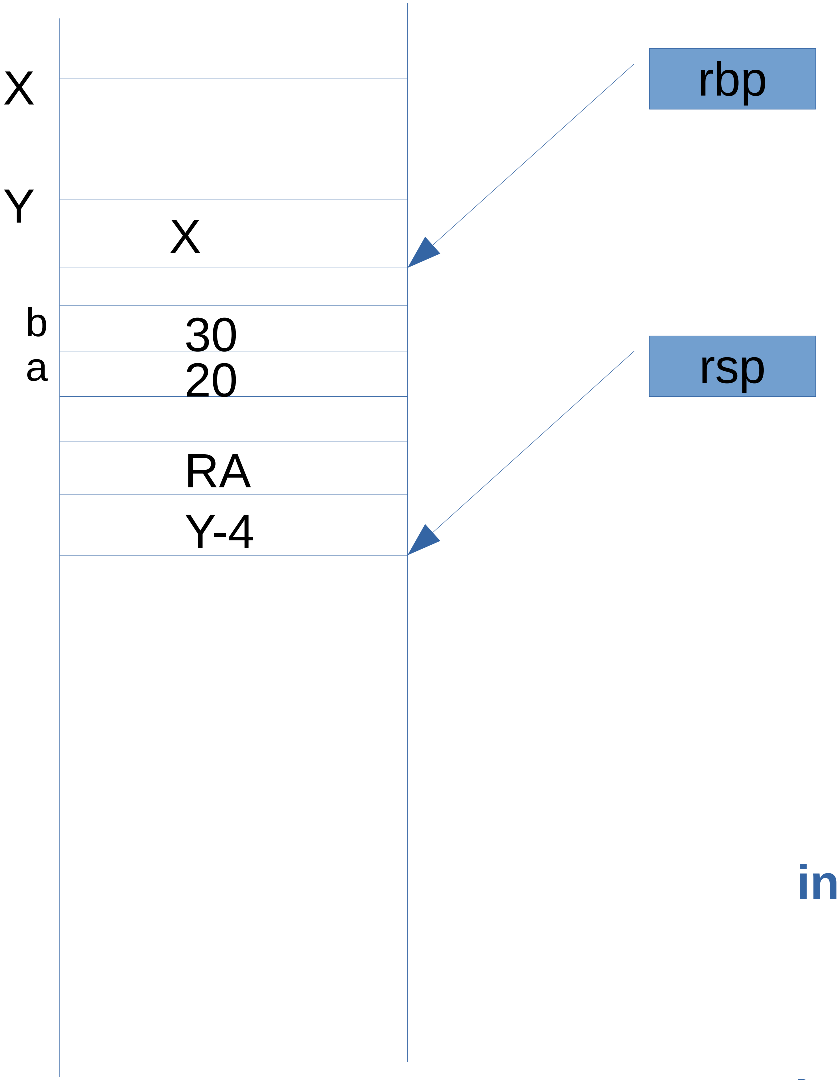

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



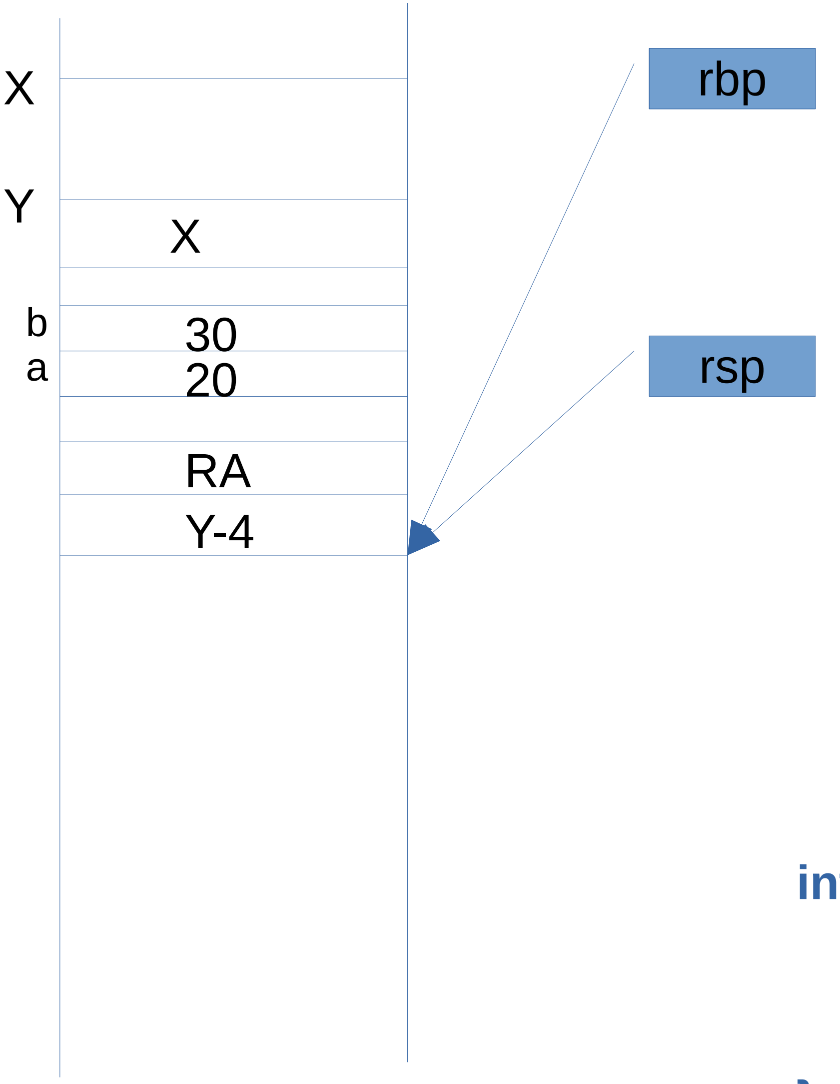

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



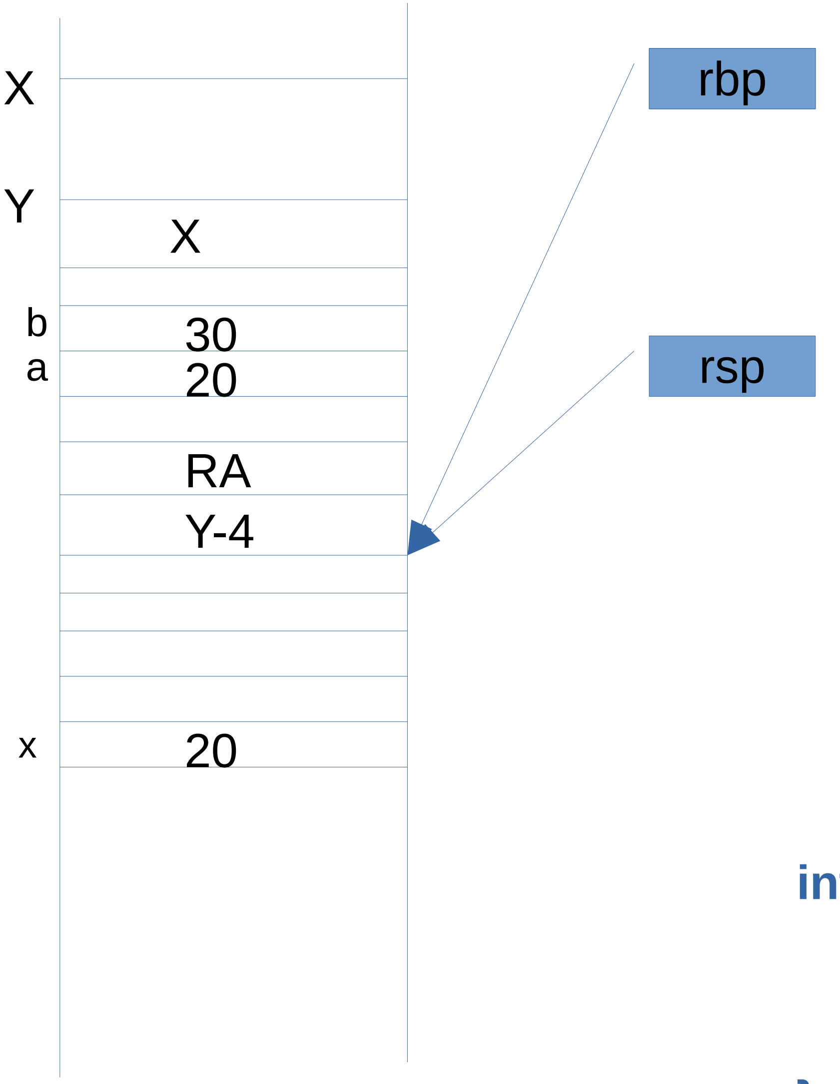

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



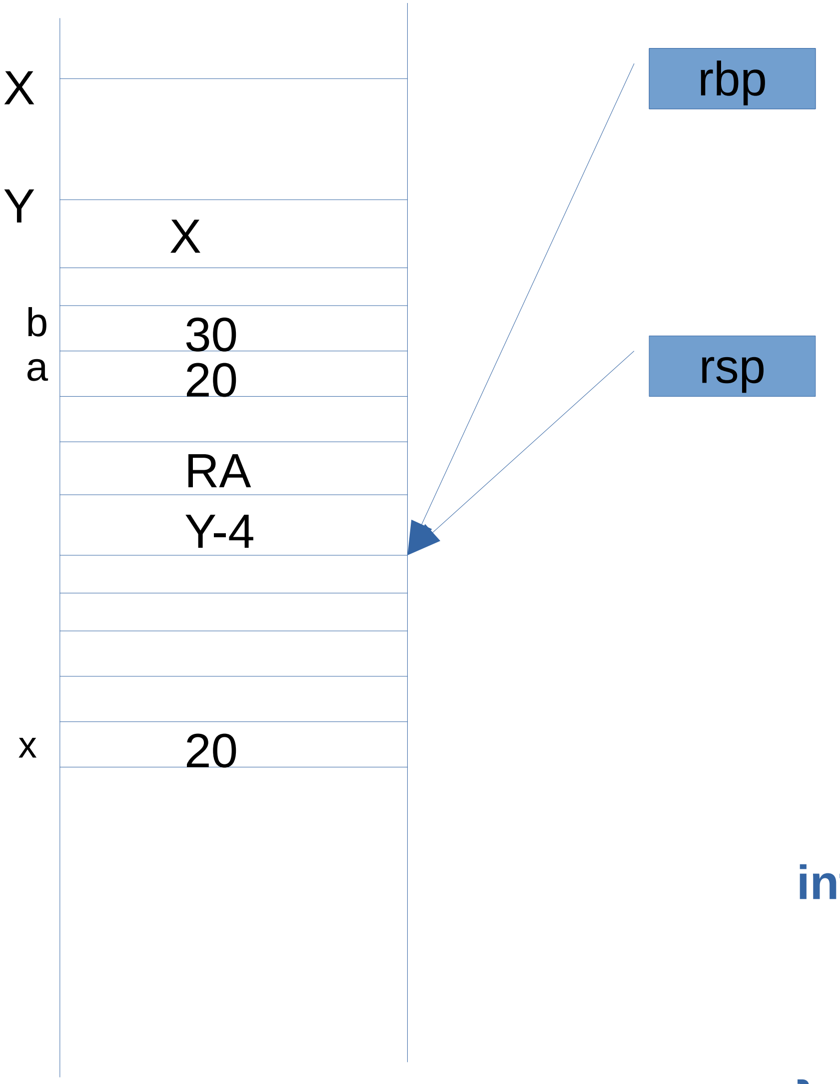

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

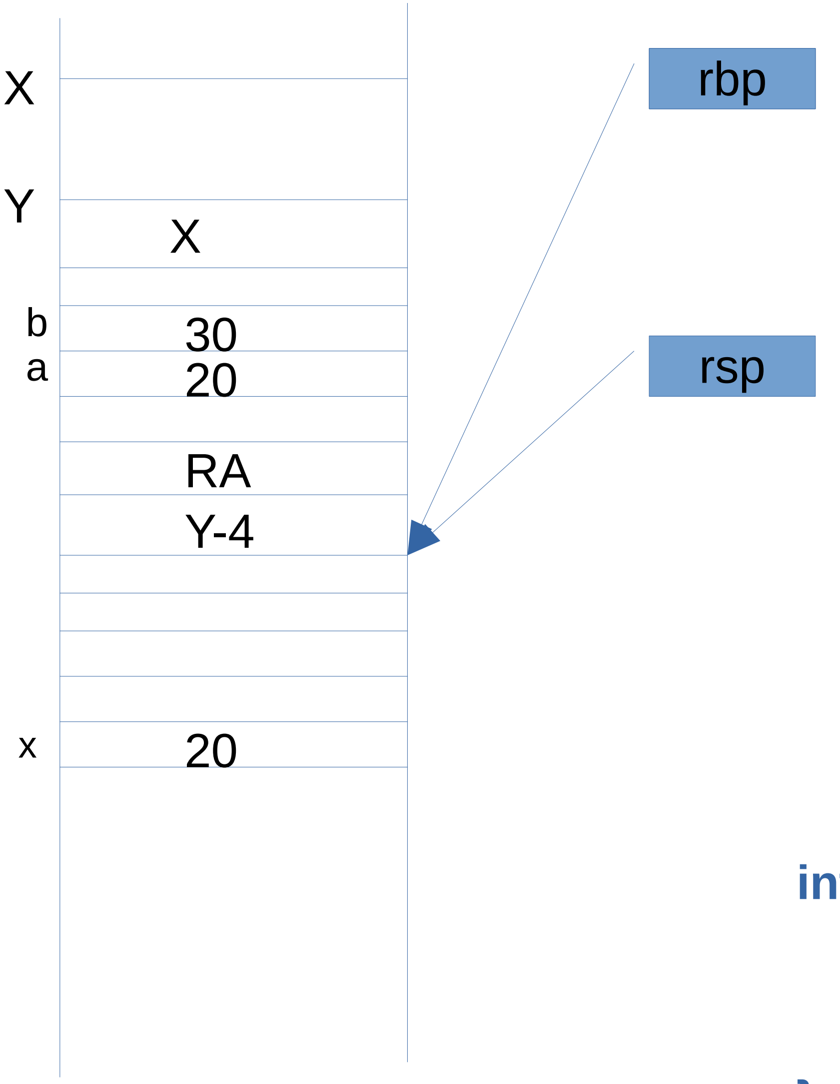
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

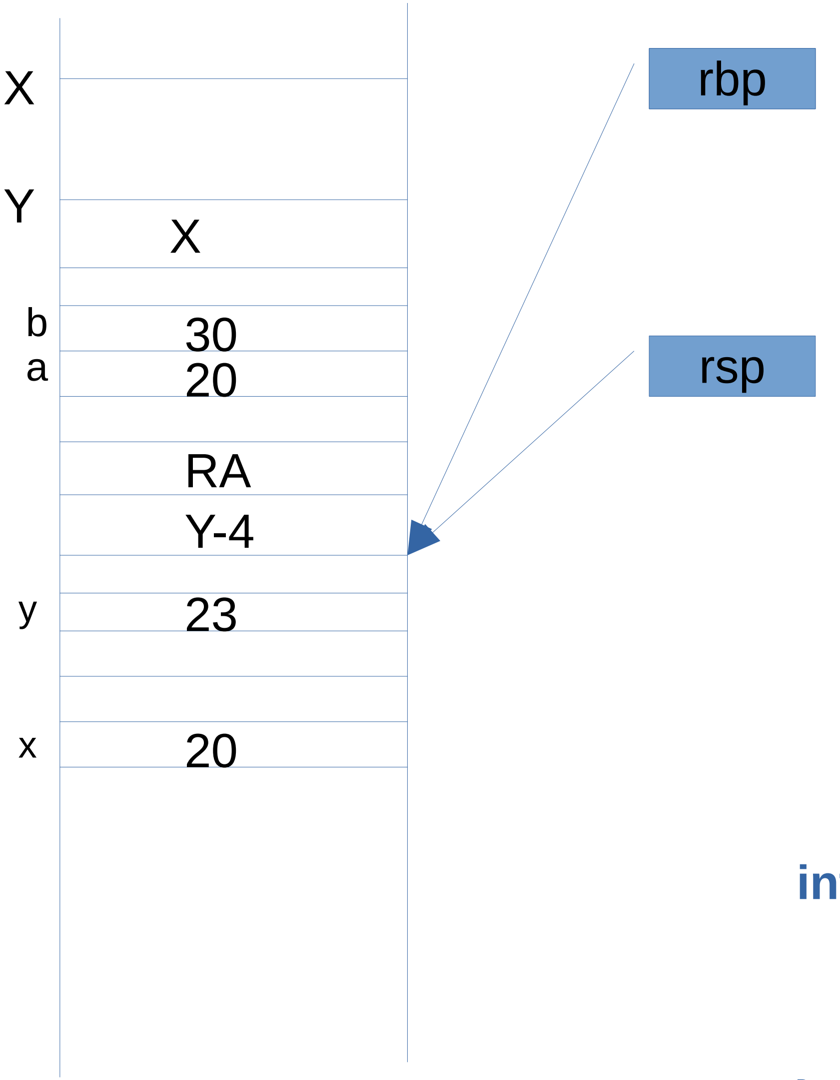
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

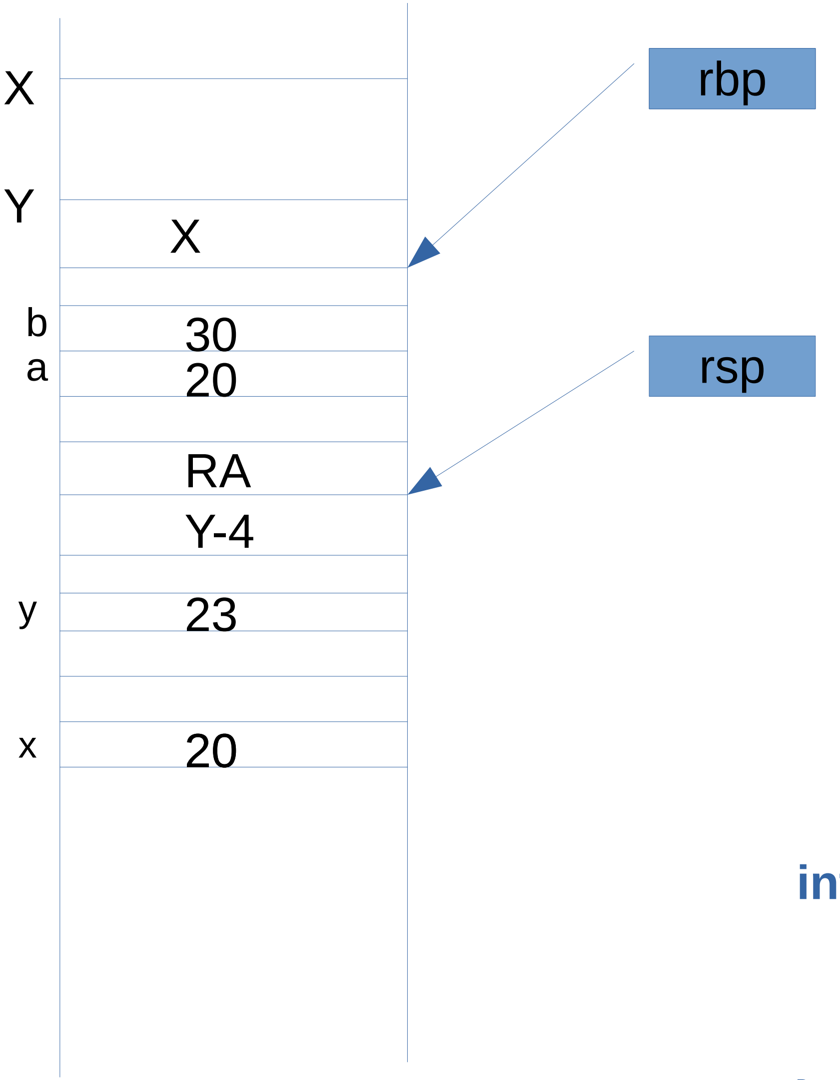
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 23

```c
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

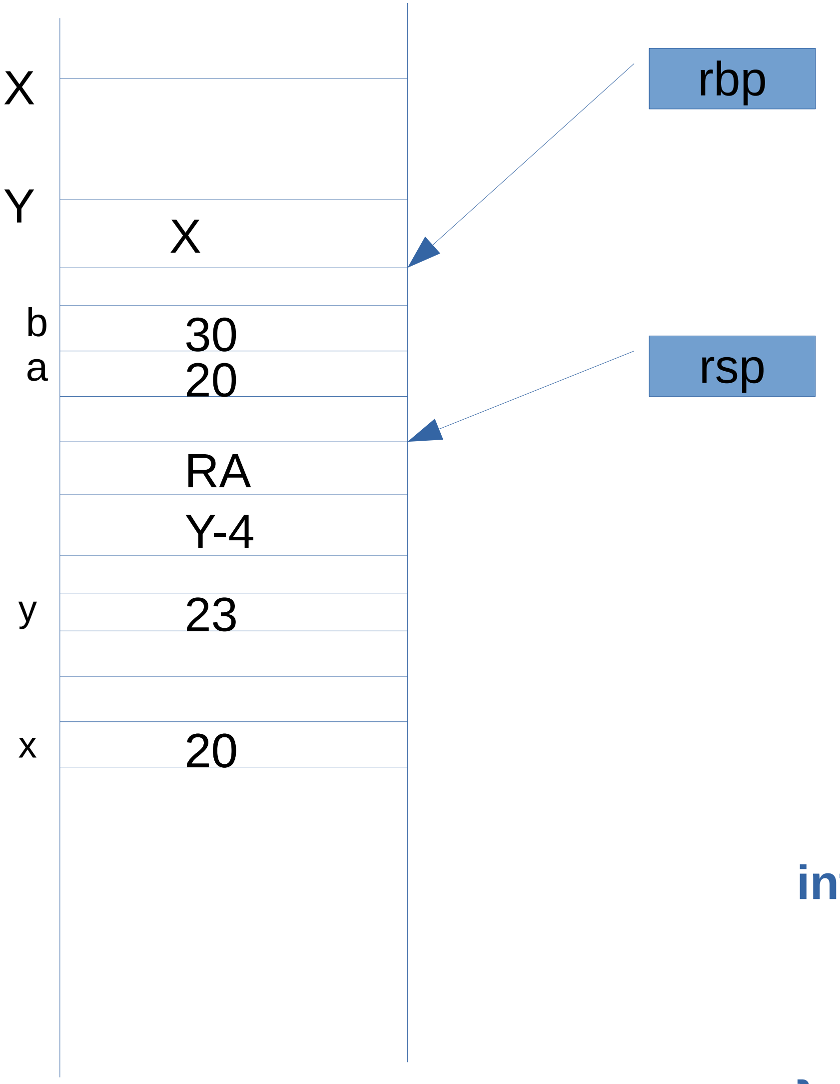
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

Eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

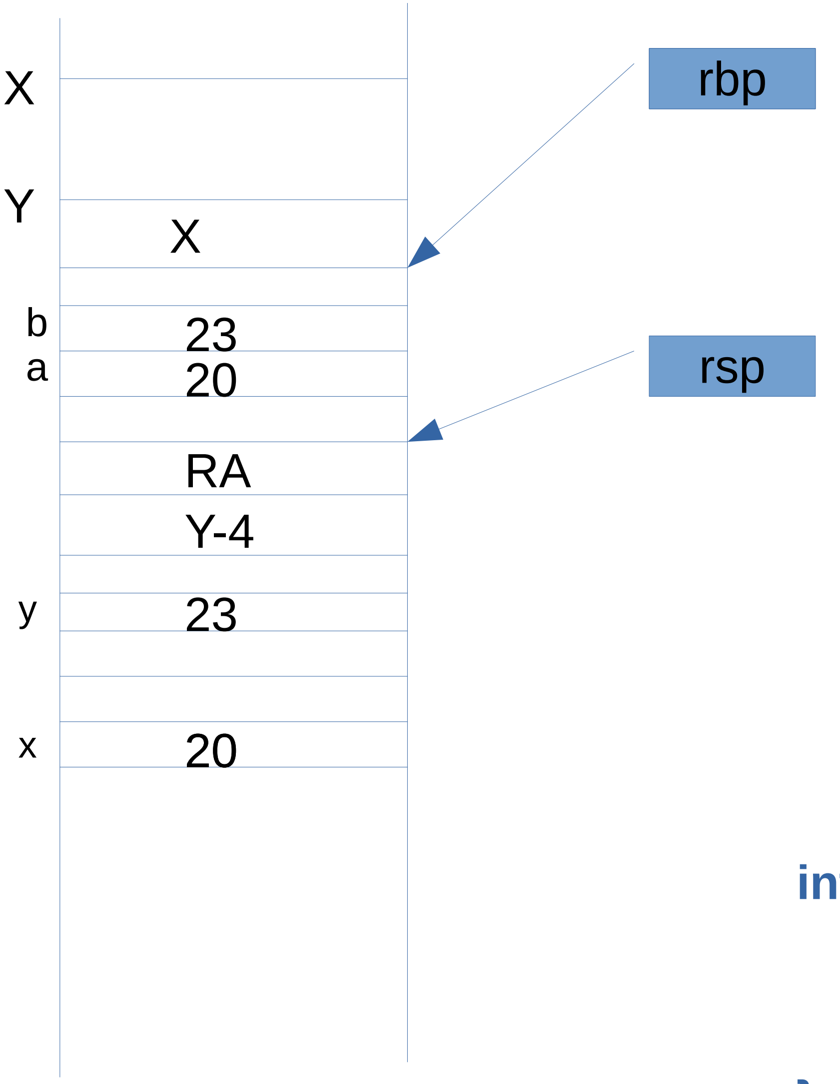
edi = 20

eax = 23

```c
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

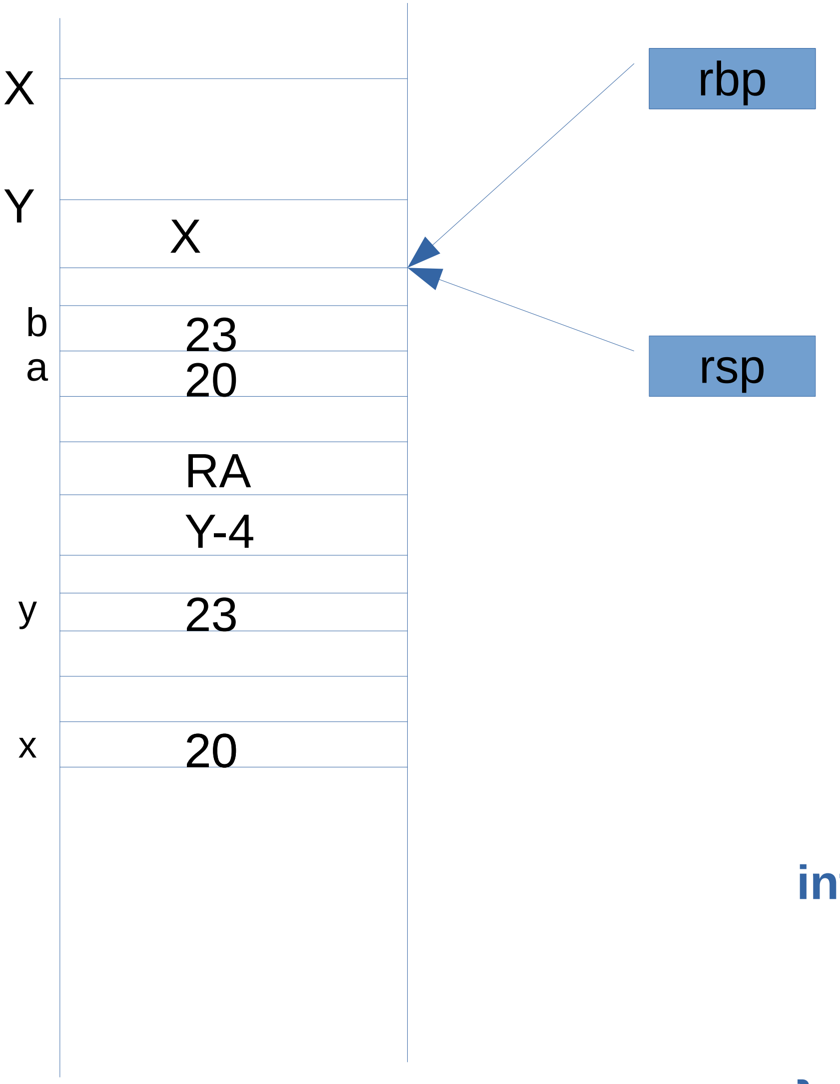
edi = 20

eax = 23

```c
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```c
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
# mov rbp rsp; pop rbp
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

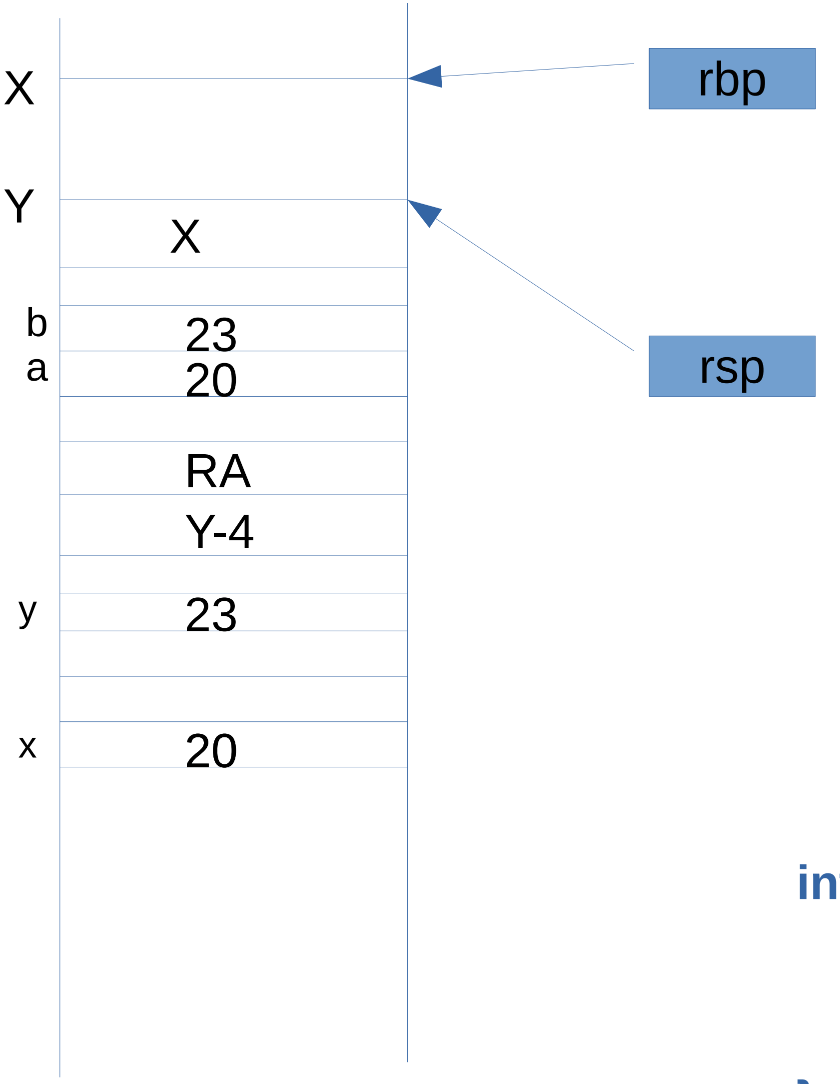
edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
# mov ebp esp; pop ebp
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA

Further on calling convention

This was a simple program

The parameter was passed in a register!

- What if there were many parameters?
- CPUs have different numbers of registers.

More parameters, more functions demand a more sophisticated convention

- May be slightly different on different processors, or 32-bit, 64-bit variants also.

Caller save and Callee save registers

Local variables

- Are visible only within the function
- Recursion: different copies of variables
- Stored on “stack”

Registers

- Are only one copy
- Are within the CPU

Local Variables & Registers conflict

- Compiler’s dillemma: While generating code for a function, which registers to use?
- The register might have been in use in earlier function call

Caller save and Callee save registers

Caller Save registers

Which registers need to be saved by caller function . They can be used by the callee function!

The caller function will push them (if already in use, otherwise no need) on the stack

Callee save registers

Will be pushed on to the stack by called (callee) function

How to return values?

On the stack itself – then caller will have to pop

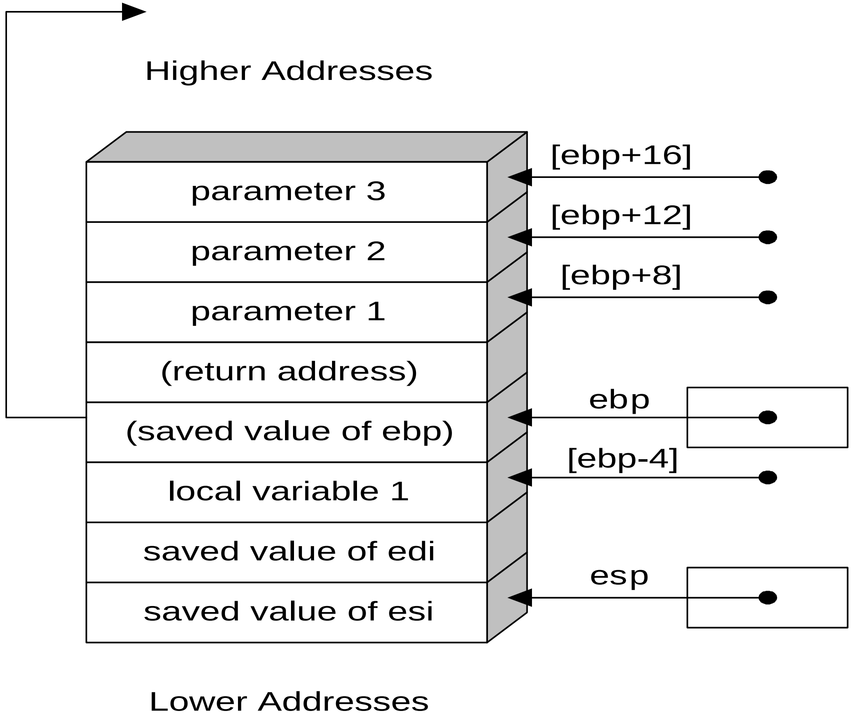
In a register, e.g. eax

X86 convention – caller, callee saved
32 bit

The caller-saved registers are EAX, ECX, EDX.

The callee-saved registers are EBX, EDI, and ESI

Activation record looks like this



F() called g()

Parameters-i refers to parameters passed by f() to g()

Local variable is a variable in g()

Return address is the location in f() where call should go back

X86 caller and callee rules(32 bit)

Caller rules on call

- Push caller saved registers on stack
- Push parameters on the stack – in reverse order. Why?
  - Substract esp, copy data
- call f() // push + jmp

Caller rules on return

- return value is in eax
- remove parameters from stack : Add to esp.
- Restore caller saved registers (if any)

X86 caller and callee rules

Callee rules on call

1) push ebp

   mov ebp, esp

ebp(+-offset) normally used to locate local vars and parameters on stack

ebp holds a copy of esp

Ebp is pushed so that it can be later popped while returnig

2) Allocate local variables

3) Save callee-saved registers

https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

X86 caller and callee rules

Callee rules on return

1) Leave return value in eax

2) Restore callee saved registers

3) Deallocate local variables

4) restore the ebp

5) return

https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

32 bit vs 64 bit calling convention

Registers are used for passing parameters in 64 bit , to a large extent

- Upto 6 parameters
- More parameters pushed on stack

See

- https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf

Beware

When you read assembly code generated using

- gcc -S

You will find

- More complex instructions
- But they will essentially follow the convention mentioned

Comparison

	MIPS	x86
Arguments:	First 4 in %a0–%a3, remainder on stack	Generally all on stack
Return values:	%v0–%v1	%eax
Caller-saved registers:	%t0–%t9	%eax, %ecx, & %edx
Callee-saved registers:	%s0–%s9	Usually none

Figure 6.2: A comparison of the calling conventions of MIPS and x86

From the textbook by Misruda

simple3.c and simple3.s

```
int f(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int x10) {
    int h;
    h = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + 3;
    return h;
}
int main() {
    int a1 = 10, a2 = 20, a3 = 30, a4 = 40, a5 = 50, a6 = 60, a7 = 70, a8 = 80, a9 = 90, a10 = 100;
    int b;
    b = f(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10);
    return b;
}
```

simple3.c and simple3.s

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $48, %rsp
    movl    $10, -44(%rbp)
    movl    $20, -40(%rbp)
    movl    $30, -36(%rbp)
    movl    $40, -32(%rbp)
    movl    $50, -28(%rbp)
    movl    $60, -24(%rbp)
    movl    $70, -20(%rbp)
    movl    $80, -16(%rbp)
    movl    $90, -12(%rbp)
    movl    $100, -8(%rbp)
```

```
movl    -24(%rbp), %r9d
 movl    -28(%rbp), %r8d
 movl    -32(%rbp), %ecx
 movl    -36(%rbp), %edx
 movl    -40(%rbp), %esi
 movl    -44(%rbp), %eax
 movl    -8(%rbp), %edi
 pushq   %rdi
 movl    -12(%rbp), %edi
 pushq   %rdi
 movl    -16(%rbp), %edi
 pushq   %rdi
 movl    -20(%rbp), %edi
 pushq   %rdi
```

```
movl    %eax, %edi
call    f
addq    $32, %rsp
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
ret
```

simple3.c and simple3.s

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    %edx, -28(%rbp)
    movl    %ecx, -32(%rbp)
    movl    %r8d, -36(%rbp)
    movl    %r9d, -40(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -28(%rbp), %eax
    addl    %eax, %edx
    movl    -32(%rbp), %eax
    addl    %eax, %edx
    movl    -36(%rbp), %eax
    addl    %eax, %edx
    movl    -40(%rbp), %eax
    addl    %eax, %edx
    movl    16(%rbp), %eax
    addl    %eax, %edx
    movl    24(%rbp), %eax
    addl    %eax, %edx
    movl    32(%rbp), %eax
    addl    %eax, %edx
    movl    40(%rbp), %eax
    addl    %edx, %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```