Principles of Compiled Code

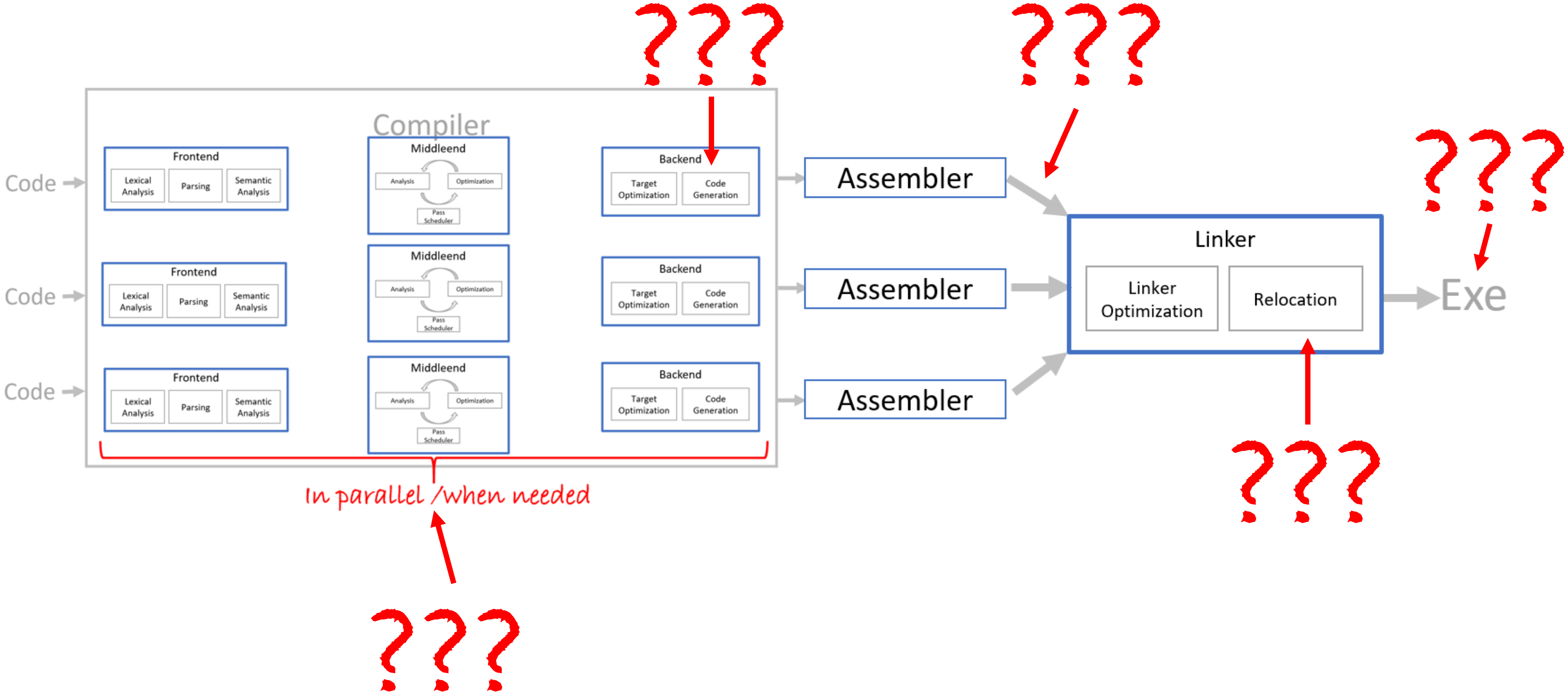
NI(E)-GEN, Spring 2021

https://courses.fit.cvut.cz/NI-GEN

ČESKÉ
VYSOKÉ
UČENÍ
TECHNICKÉ
V PRAZE

FIT

So we have a compiler…


Compiler
Code
Code
Code
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Middleend
Analysis
Optimization
Pass Scheduler
Middleend
Analysis
Optimization
Pass Scheduler
Middleend
Analysis
Optimization
Pass Scheduler
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
Assembler
Assembler
Assembler
Linker
Linker Optimization
Relocation
Exe
???
???
???
???
In parallel /when needed
???

Executables

Executable

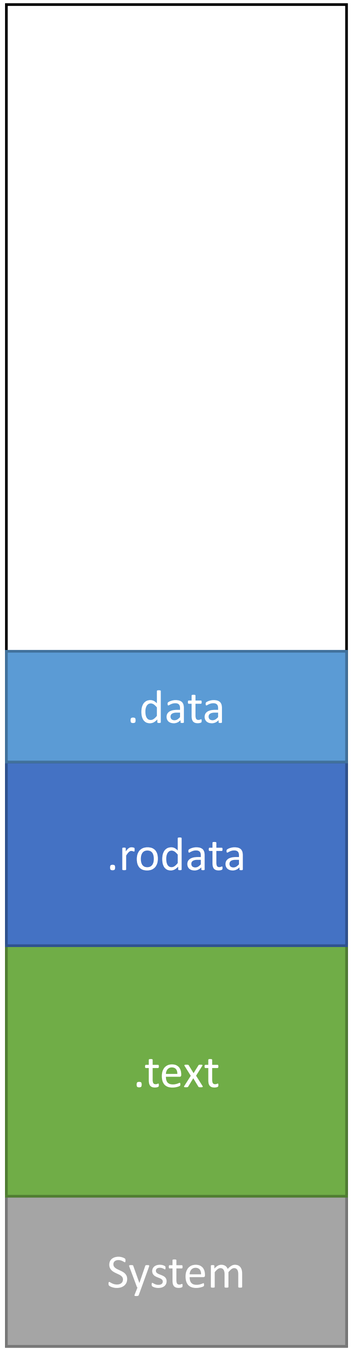
- contains binary data of various kinds partitioned in sections (.text, .data, .rodata, etc.)

```
.text:
    mov eax, hello_world
    …

.rodata
  hello_world:
    “hello all, this is a text”, 0

.data
  global-var:
    0x67
```

Executable

- contains binary data of various kinds partitioned in sections (.text, .data, .rodata, etc.)

- when executed, the section contents are copied to memory

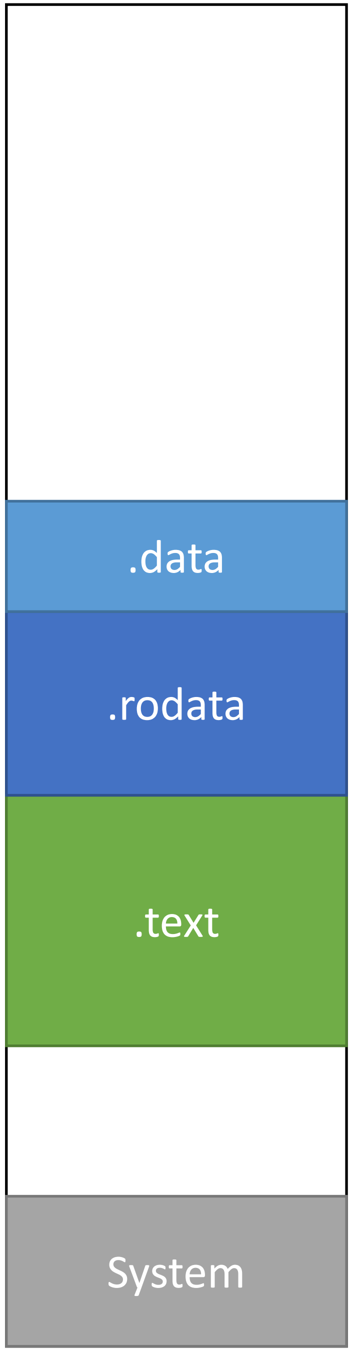


.data

.rodata

.text

System

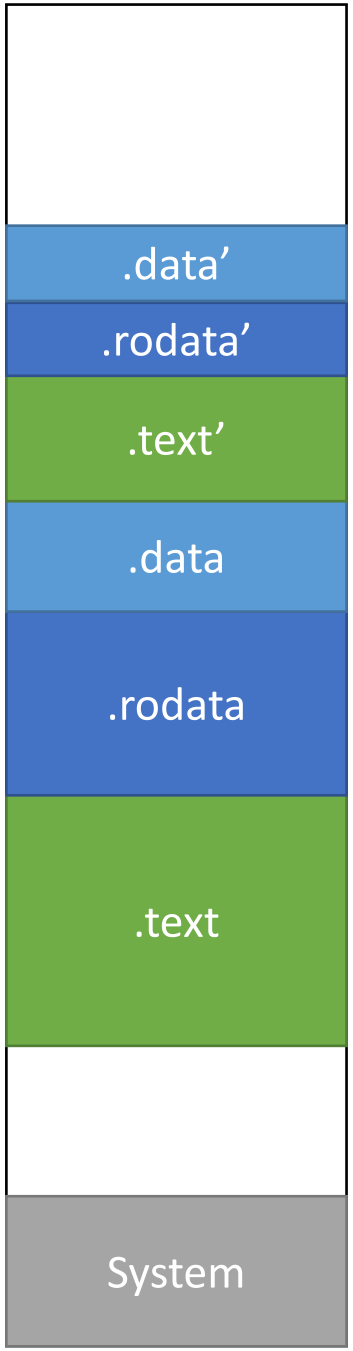


.data

.rodata

.text

System


.data’
.rodata’
.text’
.data
.rodata
.text
System

Relocation

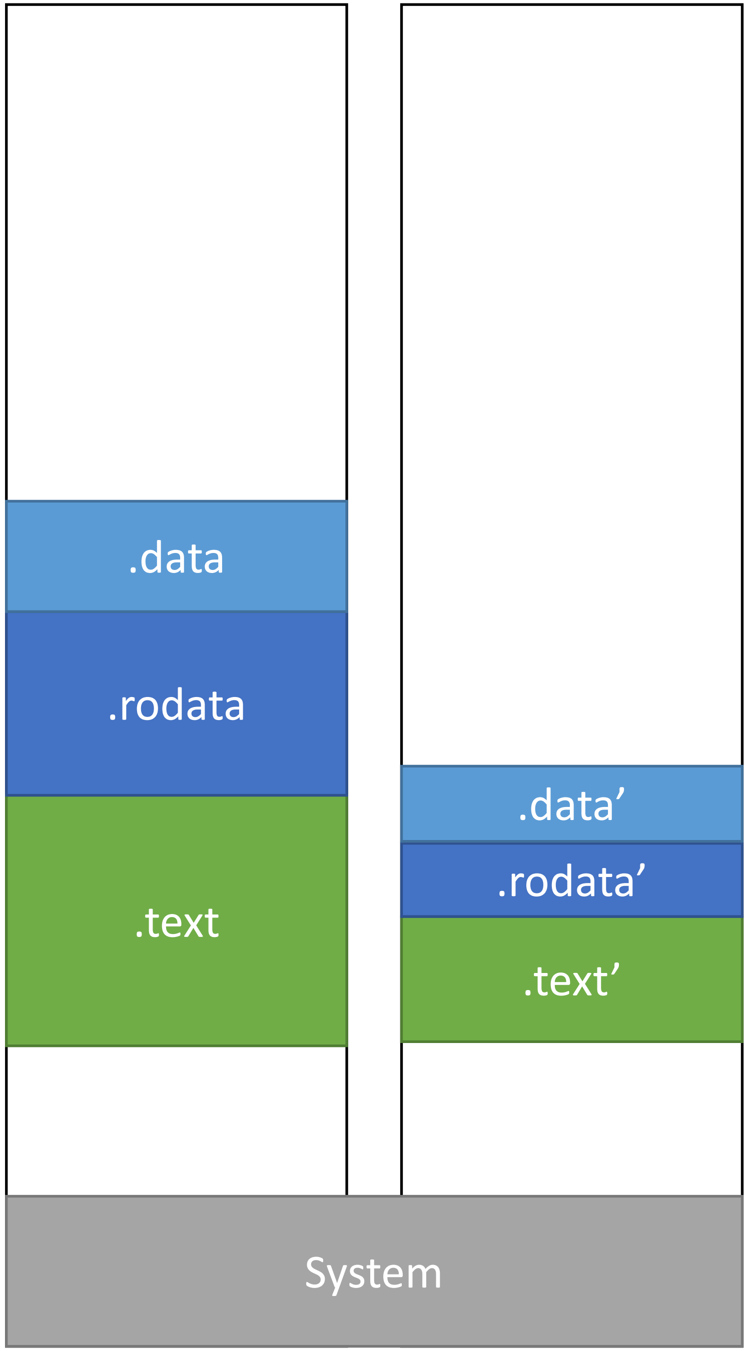
- absolute addresses of symbols cannot be known statically

- executable contains a relocation table
  - address to be patched
  - target symbol
  - patch type

- the loader then updates the section contents based on the relocation table once the section starts are known

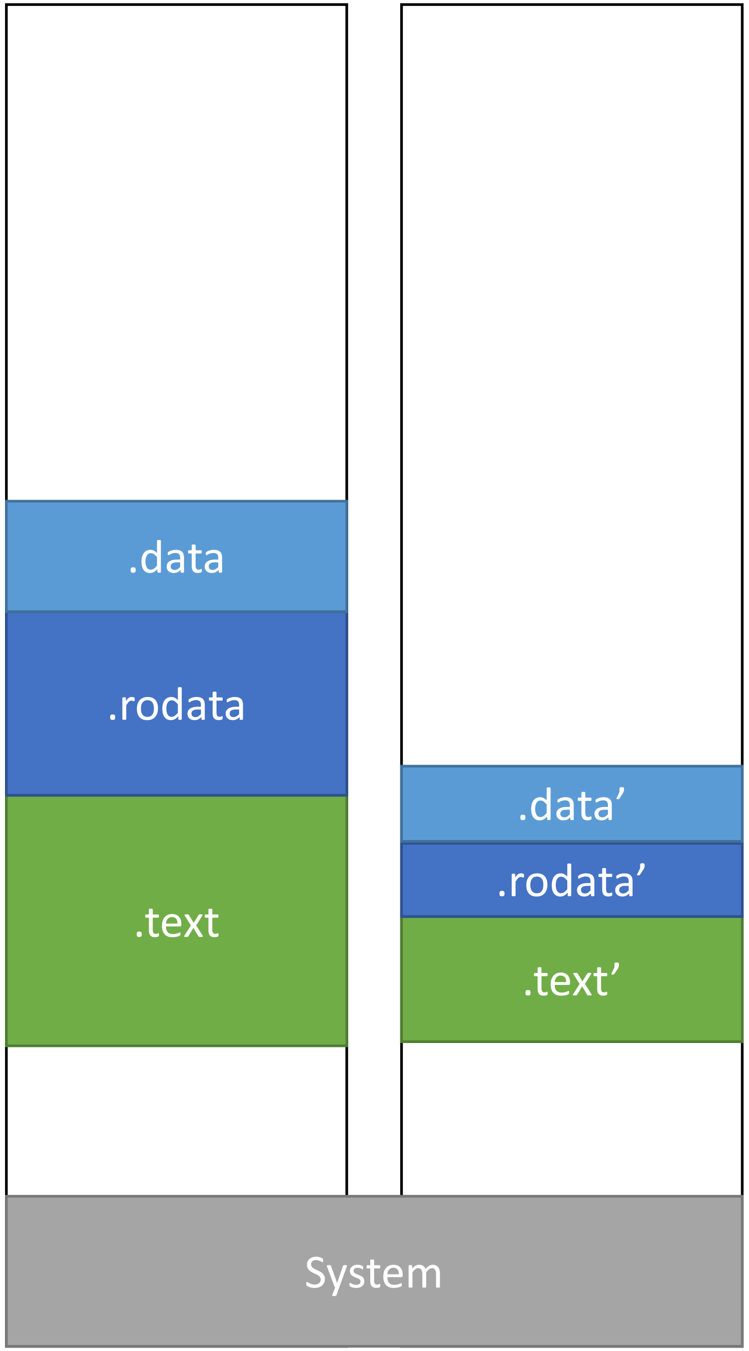
Executable

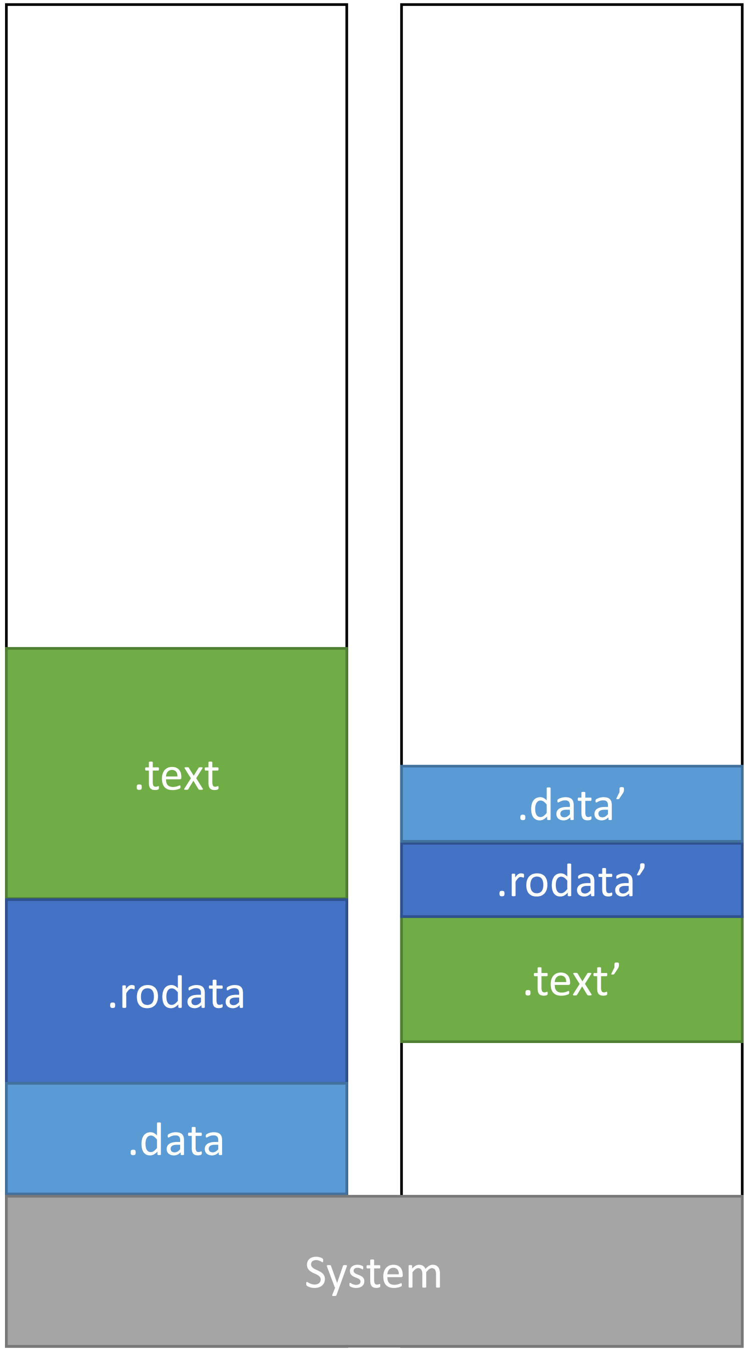
- contains binary data of various kinds partitioned in sections (.text, .data, .rodata, etc.)
- when executed, the section contents are copied to memory
- and relocated


.data
.rodata
.text
.data’
.rodata’
.text’
System

Executable

- contains binary data of various kinds partitioned in sections (.text, .data, .rodata, etc.)
- when executed, the section contents are copied to memory
- and relocated (if necessary, such as libraries)


.data
.rodata
.text
.data’
.rodata’
.text’
System

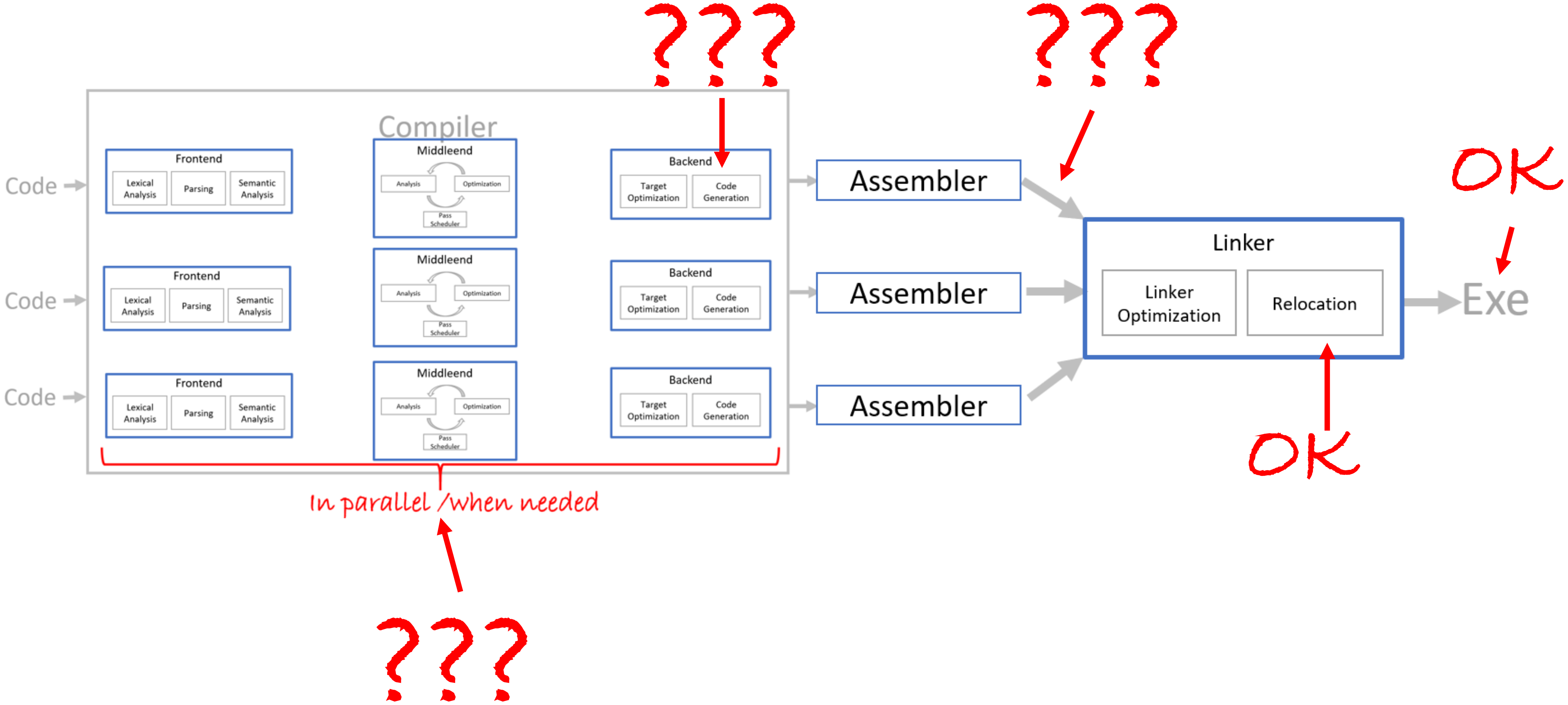

.text
.rodata
.data
.data’
.rodata’
.text’
System

Executable

- contains binary data of various kinds partitioned in sections (.text, .data, .rodata, etc.)

- when executed, the section contents are copied to memory

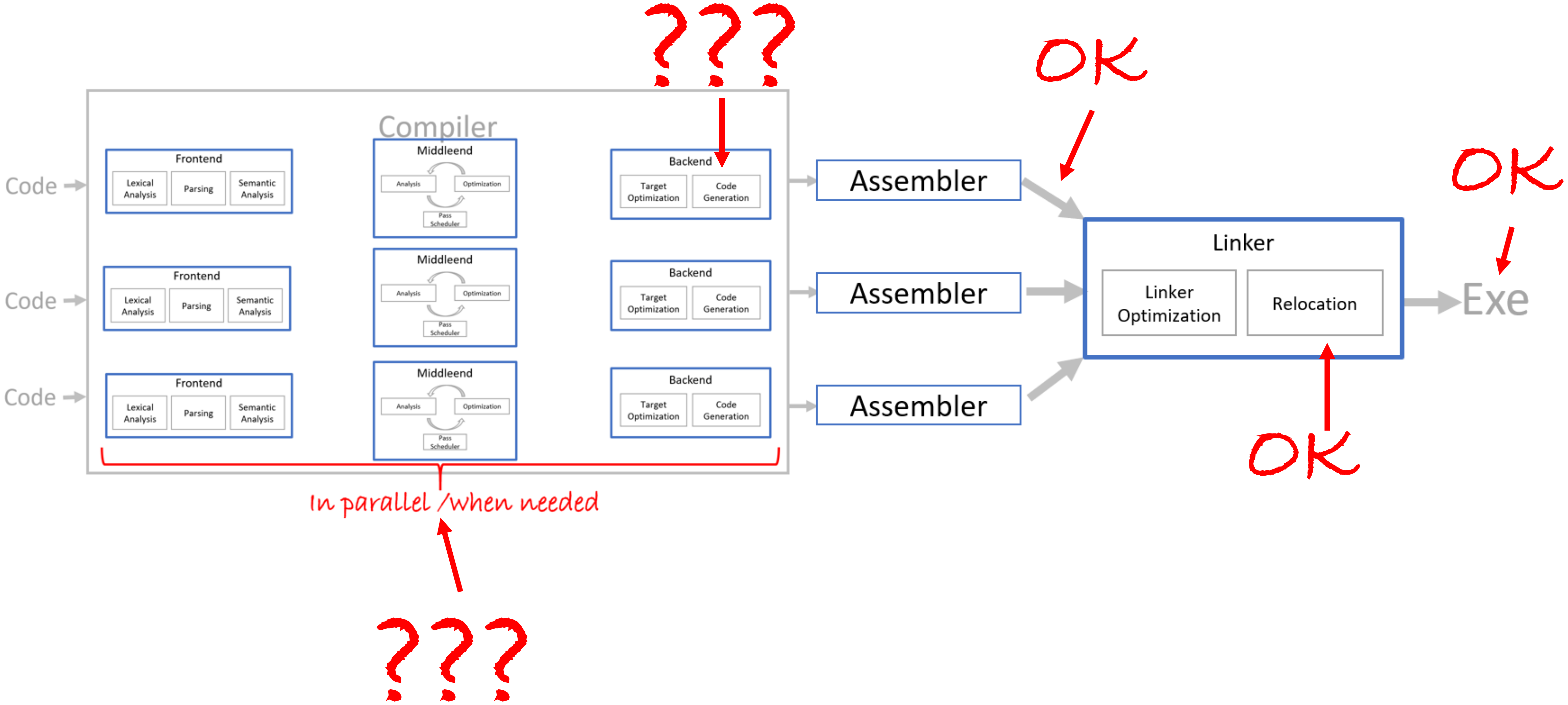
- and relocated (if necessary, such as libraries, or ASLR)


???
???
OK
Compiler
Code
Code
Code
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Middleend
Analysis
Optimization
Pass Scheduler
Backend
Target Optimization
Code Generation
Assembler
Assembler
Assembler
Linker
Linker Optimization
Relocation
Exe
OK
In parallel /when needed
???

Object Files

Object Files

- like executable
- + more sections, - loader
- a lot more relocations
- more information so that link-time optimization can be performed



???

OK

OK

Compiler

Code

Frontend

Lexical Analysis

Parsing

Semantic Analysis

Middleend

Analysis

Optimization

Pass Scheduler

Backend

Target Optimization

Code Generation

Code

Frontend

Lexical Analysis

Parsing

Semantic Analysis

Middleend

Analysis

Optimization

Pass Scheduler

Backend

Target Optimization

Code Generation

Code

Frontend

Lexical Analysis

Parsing

Semantic Analysis

Middleend

Analysis

Optimization

Pass Scheduler

Backend

Target Optimization

Code Generation

Assembler

Assembler

Assembler

Linker

Linker Optimization

Relocation

Exe

OK

In parallel /when needed

???

Source Granularity

Source Granularity

- project
- executable & libraries
- files
- functions
- basic blocks
- statements

Basic Block

- code sequence with single entry and single leave point (terminating instruction)

- once first instruction in a basic block gets executed, all instructions in basic block will execute sequentially

← no control flow

```
if (a < 0) {
  a = a + 1;
} else {
  b = b + 1;
}
c = a + b;
```

```
if (a < 0) {
    a = a + 1;
} else {
    b = b + 1;
}
c = a + b;
```

```
B0: cmp ax, 0 // a in ax
    jge B2
B1: add ax, 1
    jmp B3
B2: add bx, 1 // b in bx
B3: mov cx, ax // c in cx
    add cx, bx
```

```
if (a < 0) {
    a = a + 1;
} else {
    b = b + 1;
}
c = a + b;
```

```
B0: cmp ax, 0 // a in ax
    jge B2
B1: add ax, 1
    jmp B3
B2: add bx, 1 // b in bx
    jmp B3
B3: mov cx, ax // c in cx
    add cx, bx
```

BI-PPA, BI-PJP

Code Generation

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
f_min: # functions are labels
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
f_min:
    cmp ax, bx # arguments in
    jl x_less # ax and bx regs

x_less:
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
f_min:
    cmp ax, bx
    jl x_less
    mov ax, bx # result
    ret # passed in ax reg
x_less:
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
f_min:
    cmp ax, bx
    jl x_less
    mov ax, bx
    ret
x_less:
    ret # already in ax
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
jmp start # start with start
f_min:
    cmp ax, bx
    jl x_less
    mov ax, bx
    ret
x_less:
    ret
start: # start exec here
    mov ax, 67 # vars in regs
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
jmp start
f_min:
    cmp ax, bx
    jl x_less
    mov ax, bx
    ret
x_less:
    ret
start:
    mov ax, 67
    mov bx, 89
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
jmp start
f_min:
    cmp ax, bx
    jl x_less
    mov ax, bx
    ret
x_less:
    ret
start:
    mov ax, 67
    mov bx, 89
    call f_min # luck:) -> ax
```

Code Generation

```
int min(int x, int y) {
    return x < y ? x : y;
}

int a = 67;
int b = 89;
print(min(a, b));
```

```
jmp start
f_min:
    cmp ax, bx
    jl x_less
    mov ax, bx
    ret
x_less:
    ret
start:
    mov ax, 67
    mov bx, 89
    call f_min
    call print # ax -> ax
```

Code Generation

```
00:    jmp start
    f_min:
04:    cmp ax, bx
08:    jl x_less
0a:    mov ax, bx
0c:    ret
    x_less:
0d:    ret
    start:
0e:    mov ax, 67
12:    mov bx, 89
16:    call f_min
1a:    call print
```

Code Generation

```
00:     jmp 0e
04:     cmp ax, bx
08:     jl 0d
0a:     mov ax, bx
0c:     ret
0d:     ret
0e:     mov ax, 67
12:     mov bx, 89
16:     call 04
1a:     call print
```

Linker, help!

Memory Layout

AX

BX

CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

AX
BX
CX
DX
PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}
```

main()

```
fib(3)
```

AX

BX

CX

DX

PC

1 argument

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

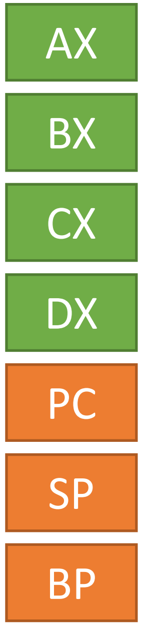
recursive

3 local variables

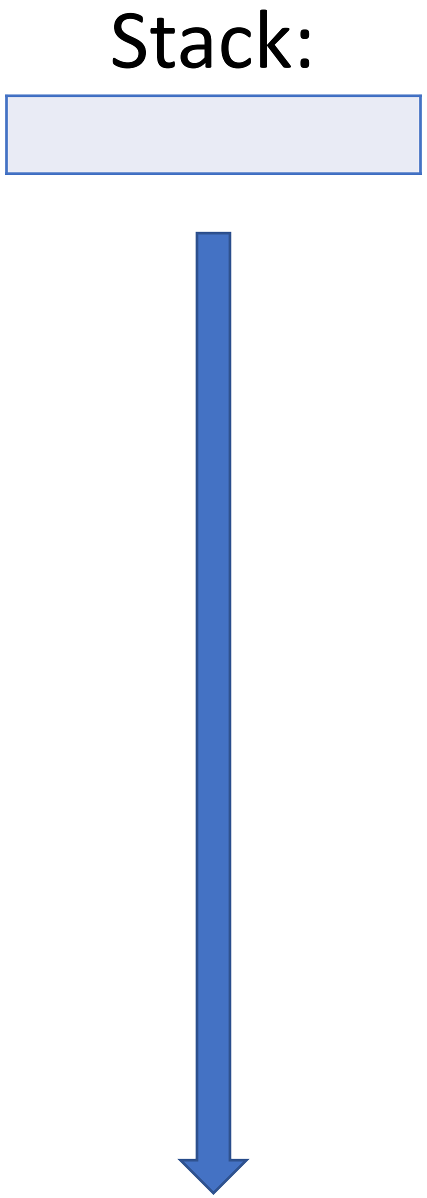
return value

Call Stack

- **L**ast **I**n – **F**irst **O**ut Structure
- Holds local variables, arguments and return addresses
- Each function’s data live in its own area called stack frame
- When function exits, the frame is popped off the stack



```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



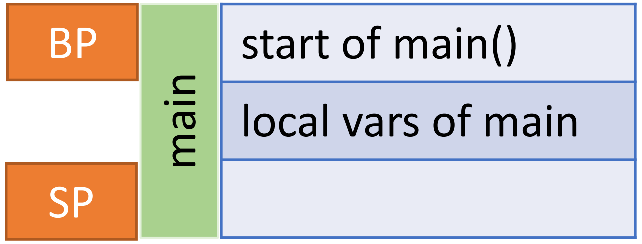
AX

BX

CX

DX

Stack:



BP

main

start of main()

local vars of main

SP

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}
```

PC fib(3)

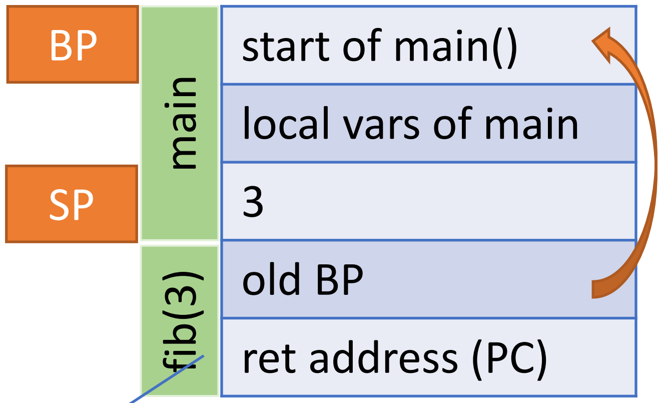
AX

BX

CX

DX

Stack:



```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}
```

PC fib(3)

AX

BX

CX

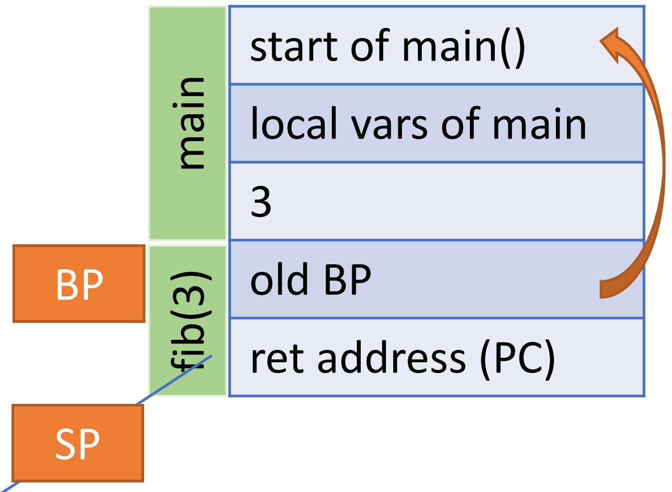
DX

[BP – 1]

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



AX

BX

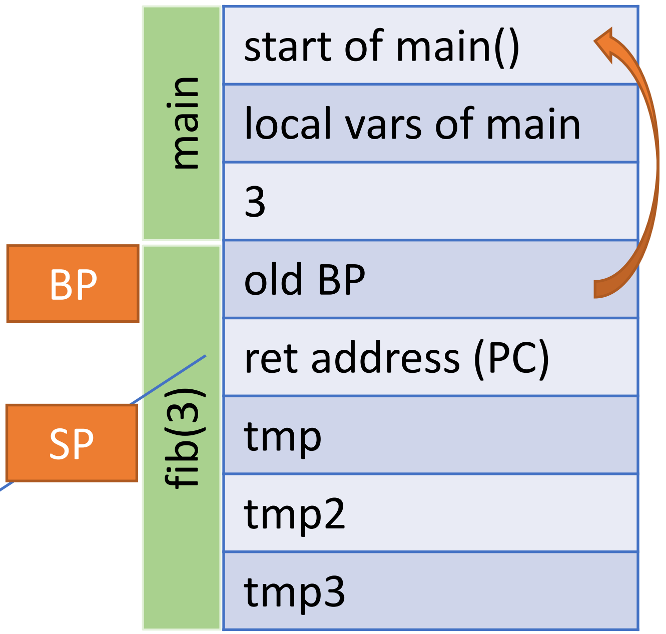
CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3

AX

BX

CX

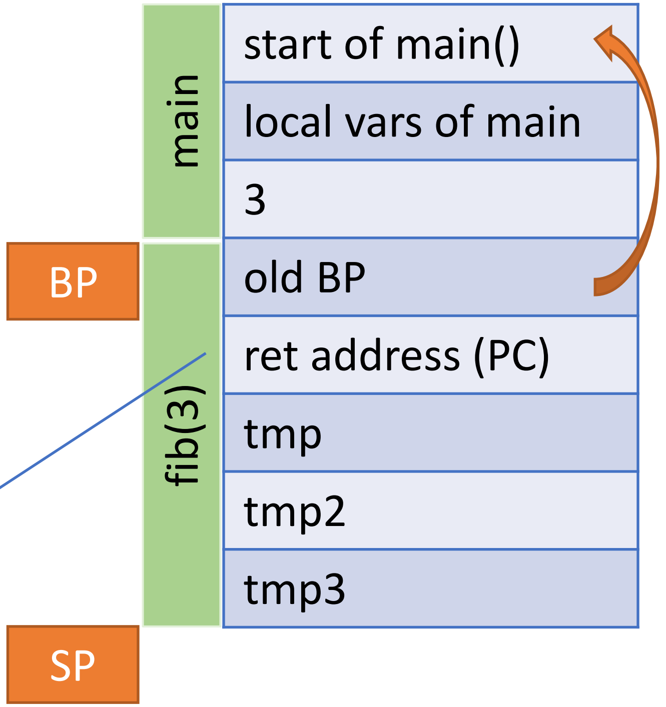
DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

[BP + 2]
[BP + 3]
[BP + 4]

Stack:



start of main()

local vars of main

3

old BP

ret address (PC)

tmp

tmp2

tmp3

AX

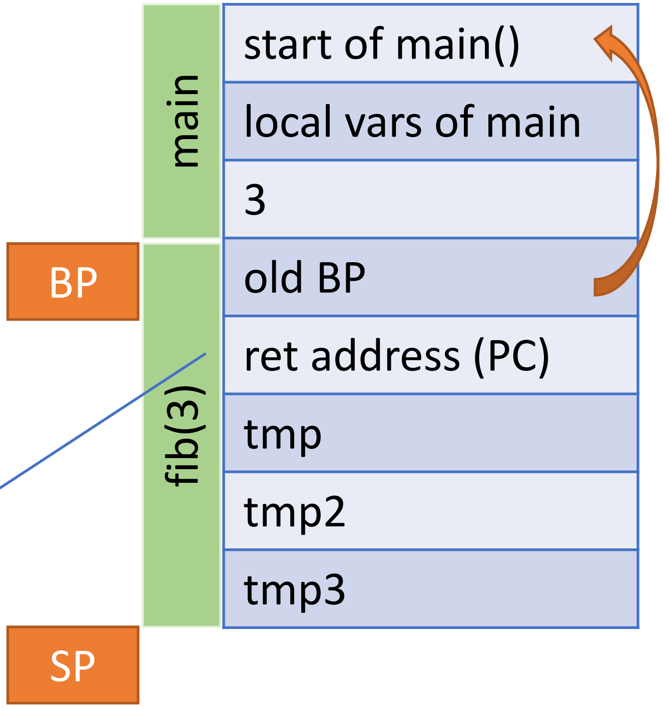
BX

CX

DX

```
int fib(int n) {
PC  if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



AX

BX

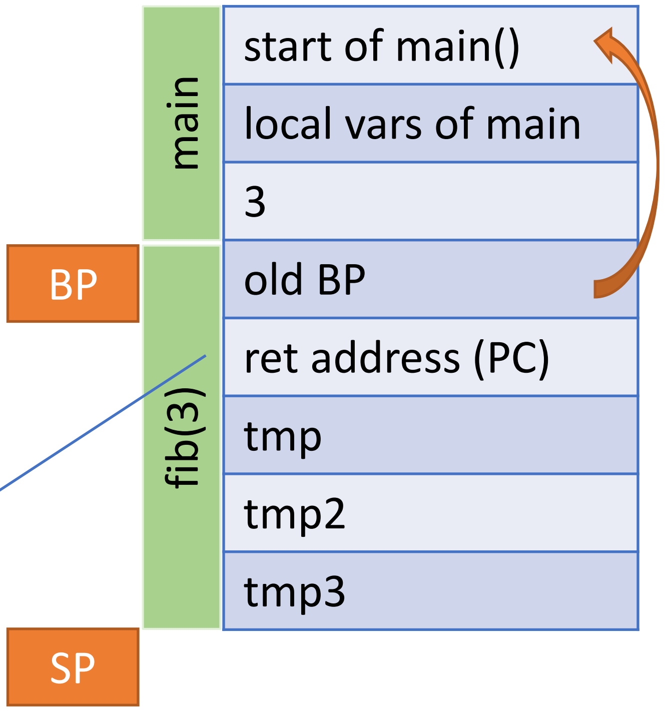
CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC

Stack:



main

fib(3)

BP

SP

start of main()

local vars of main

3

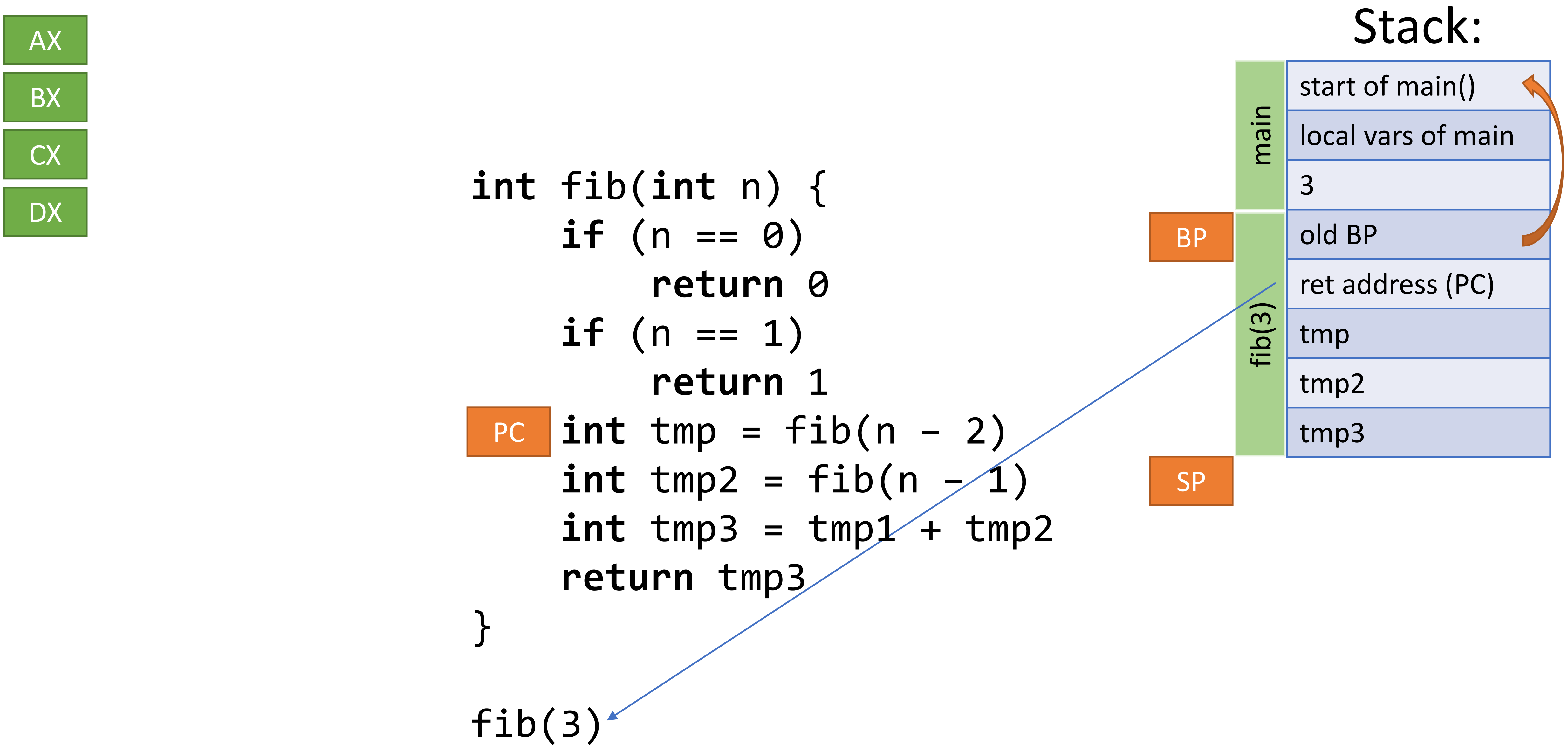
old BP

ret address (PC)

tmp

tmp2

tmp3



AX

BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC

Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3

BP

SP

AX

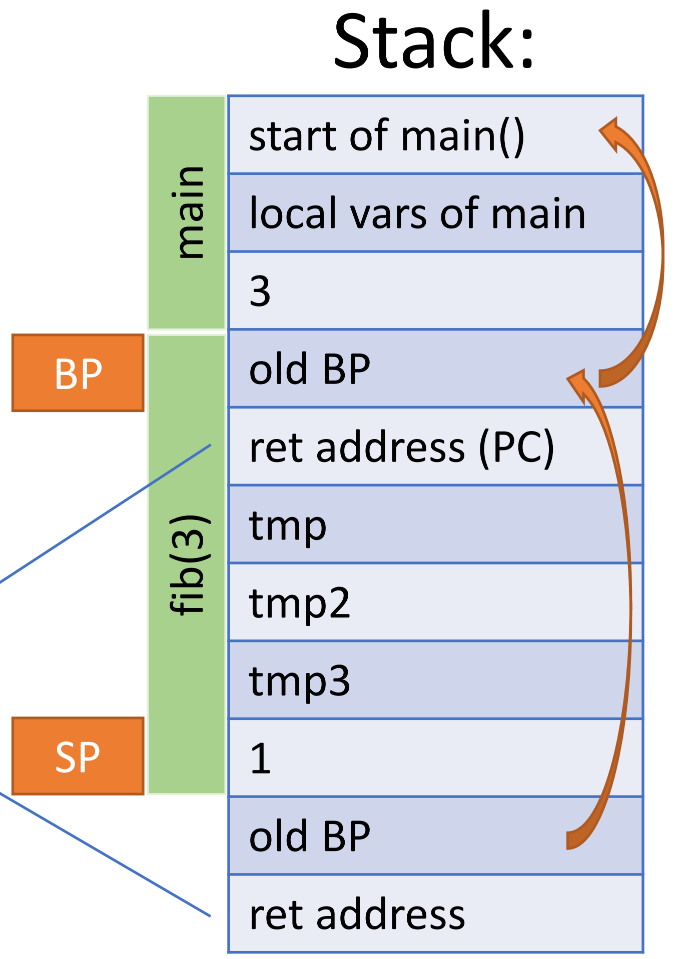
BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC



Stack:

AX

BX

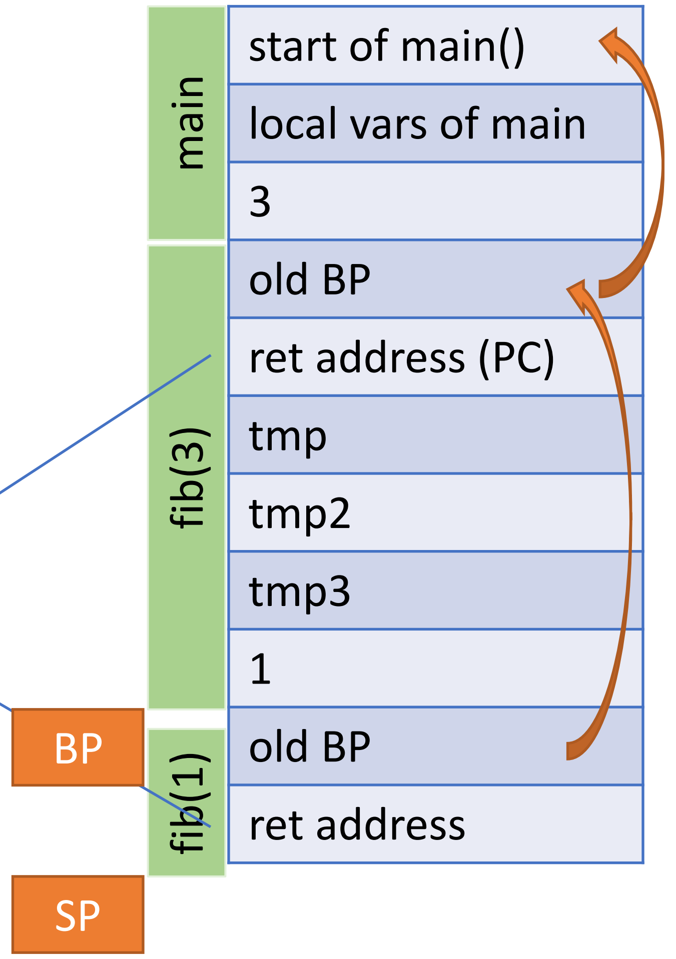
CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



start of main()

local vars of main

3

old BP

ret address (PC)

tmp

tmp2

tmp3

1

old BP

ret address

AX

BX

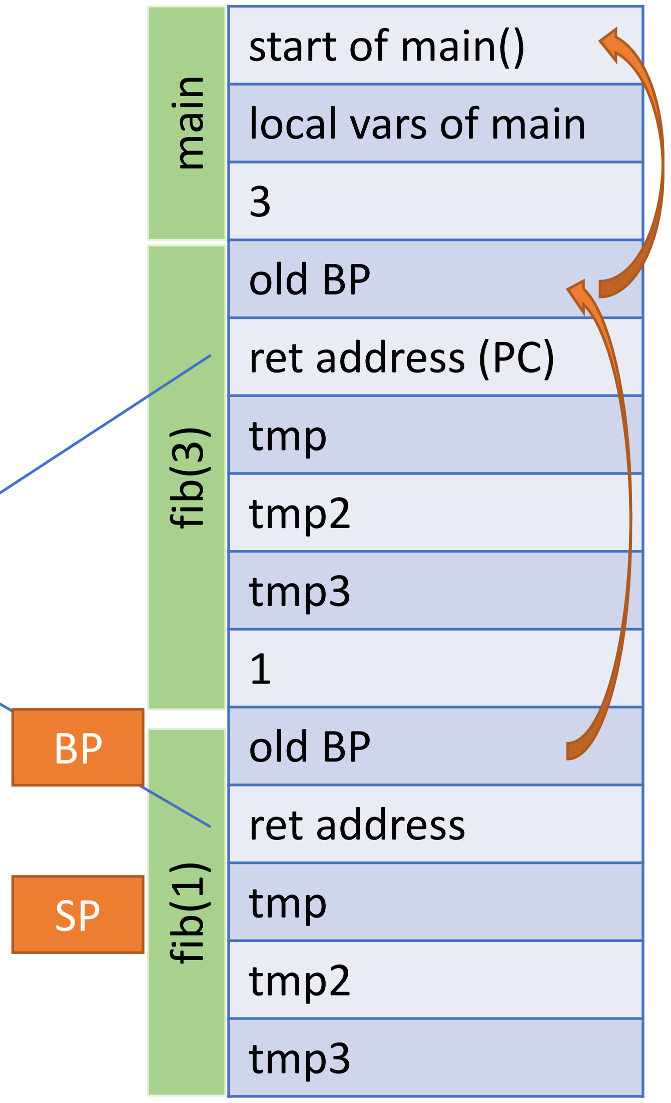
CX

DX

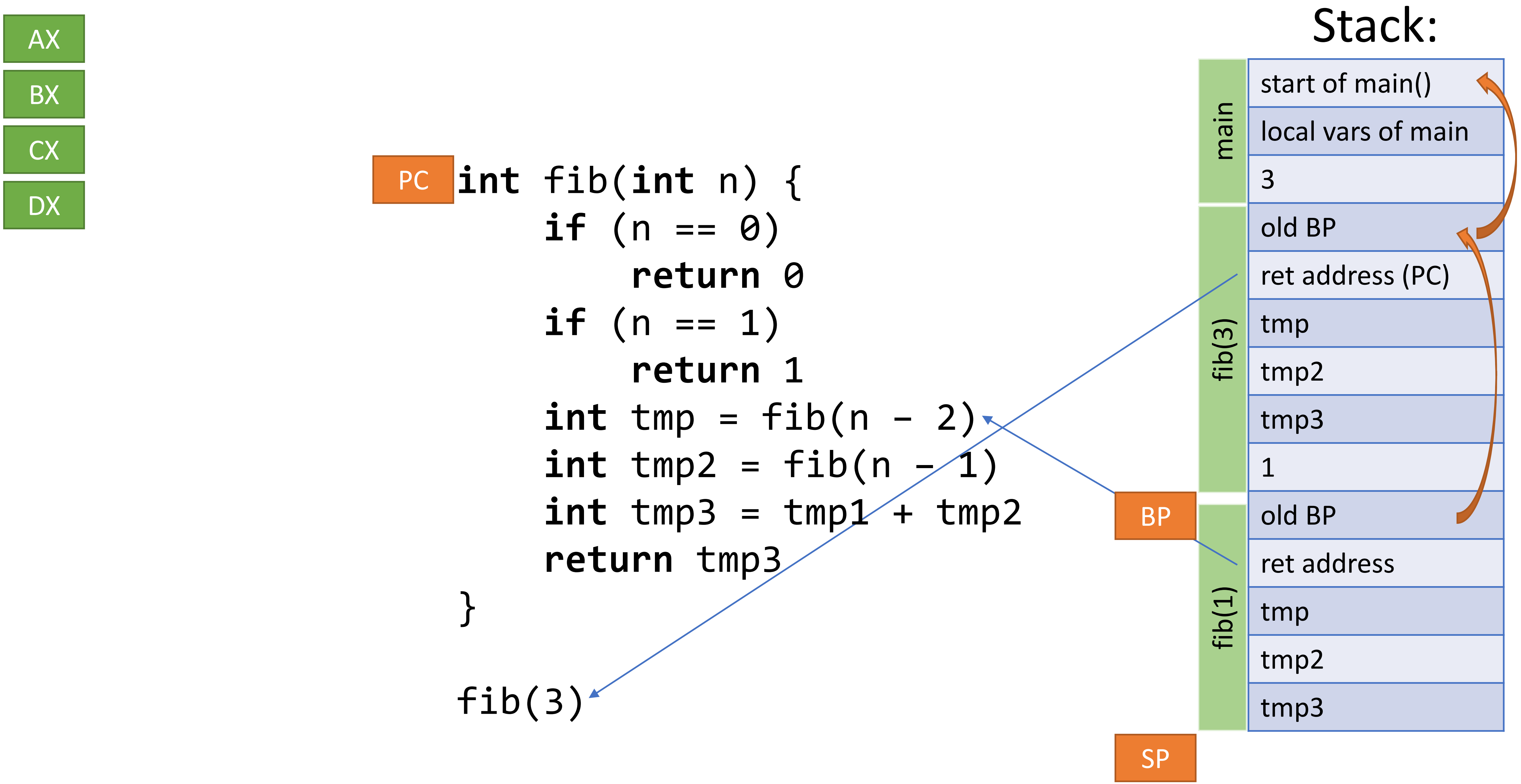
PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3
	1
fib(1)	old BP
	ret address
	tmp
	tmp2
	tmp3



```
int fib(int n) {
      if (n == 0)
            return 0
      if (n == 1)
            return 1
      int tmp = fib(n - 2)
      int tmp2 = fib(n - 1)
      int tmp3 = tmp1 + tmp2
      return tmp3
}

fib(3)
```

Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3
	1
fib(1)	old BP
	ret address
	tmp
	tmp2
	tmp3

AX

BX

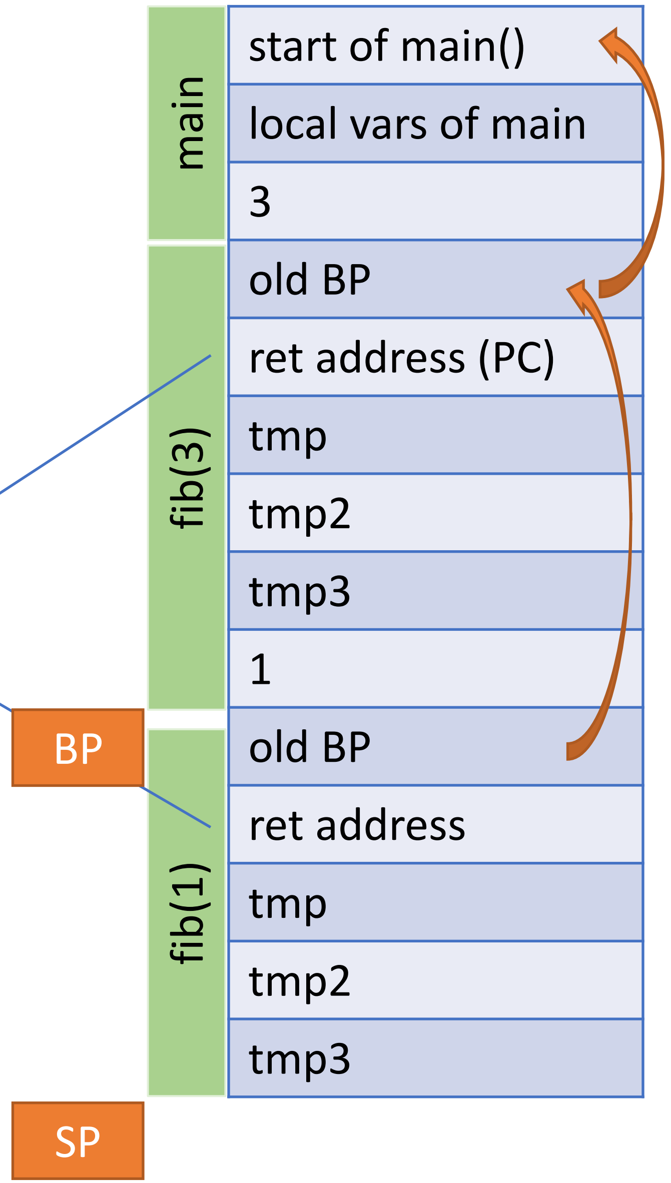
CX

DX

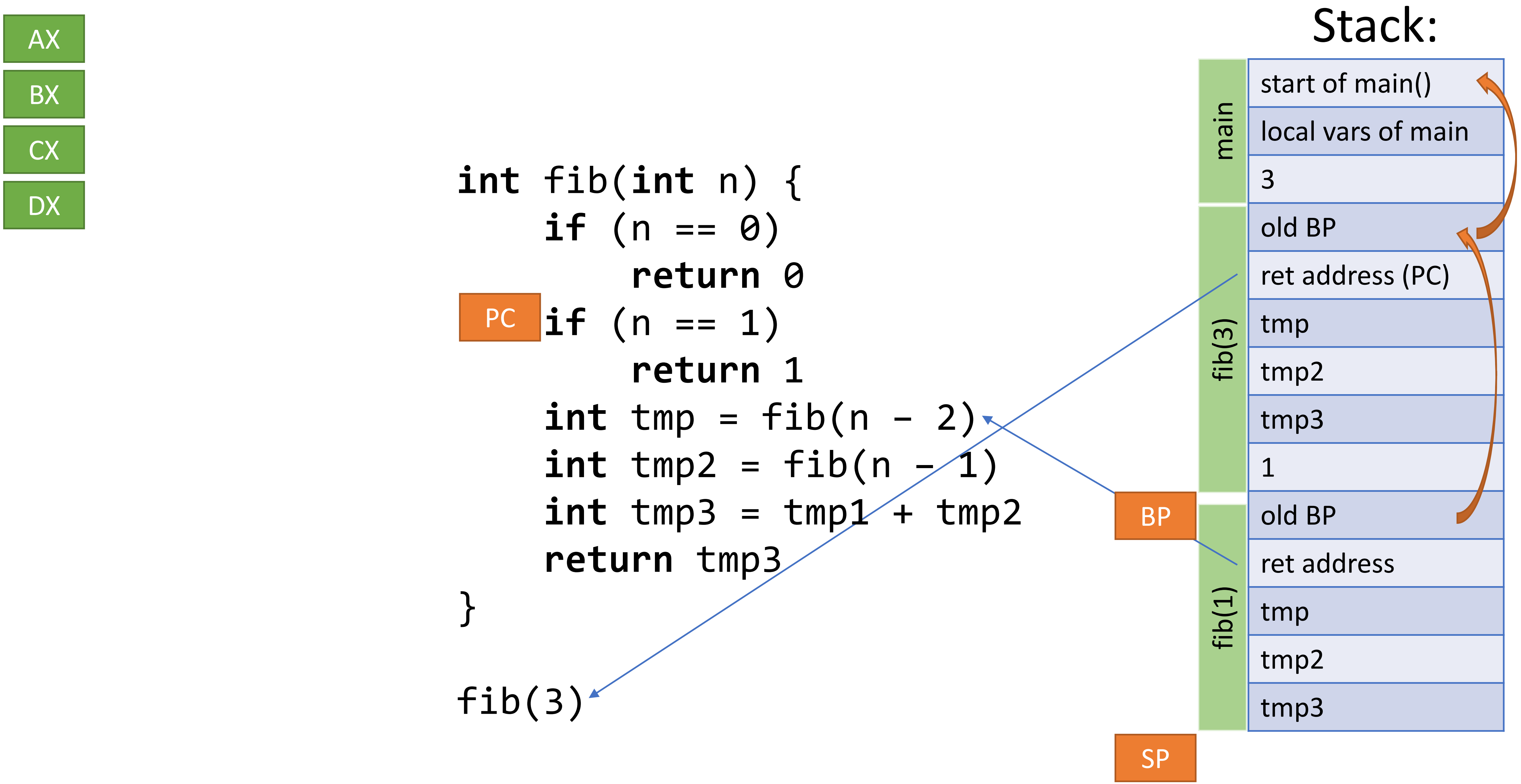
PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3
	1
fib(1)	old BP
	ret address
	tmp
	tmp2
	tmp3



AX

BX

CX

DX

PC

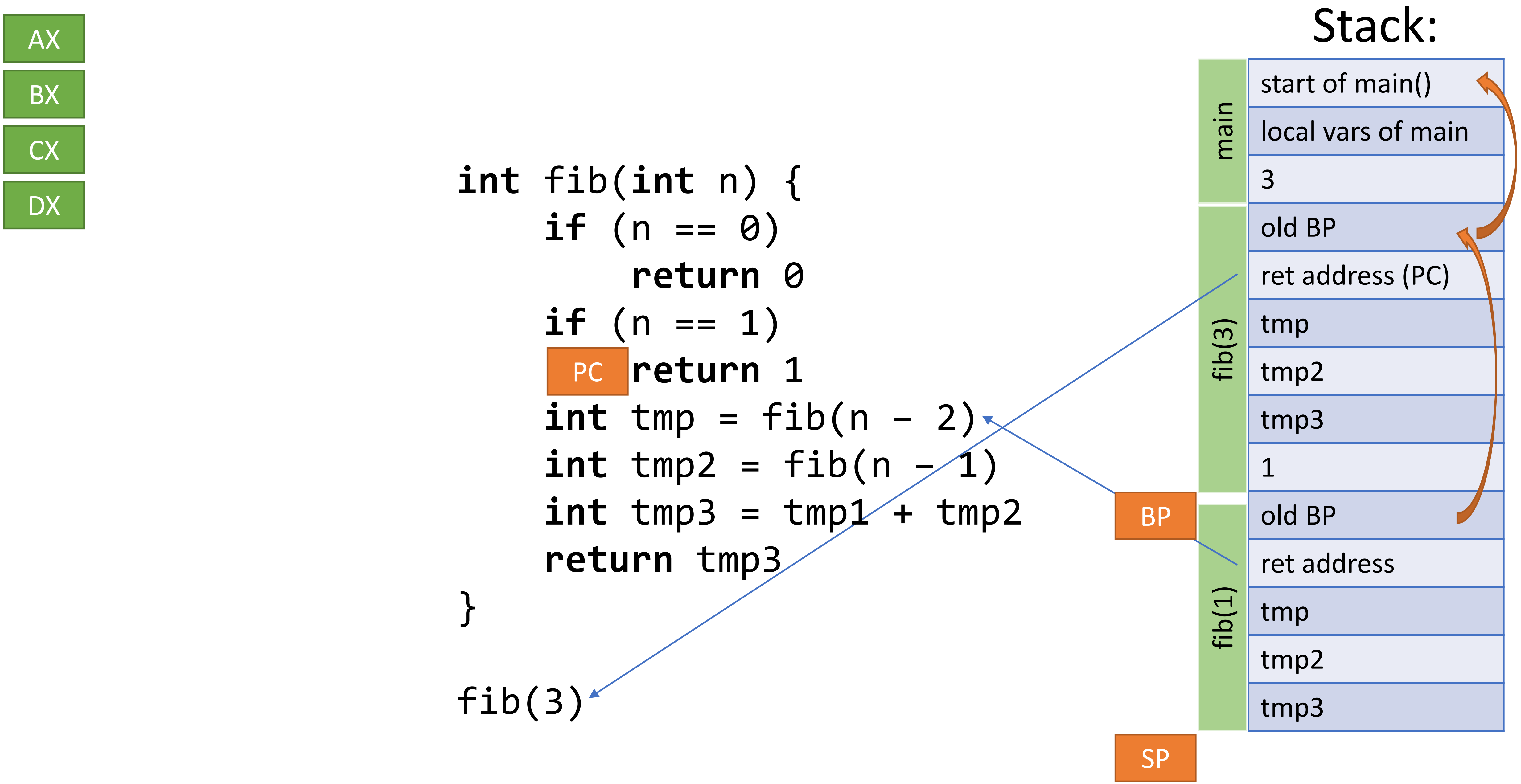
```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3
	1
fib(1)	old BP
	ret address
	tmp
	tmp2
	tmp3

BP

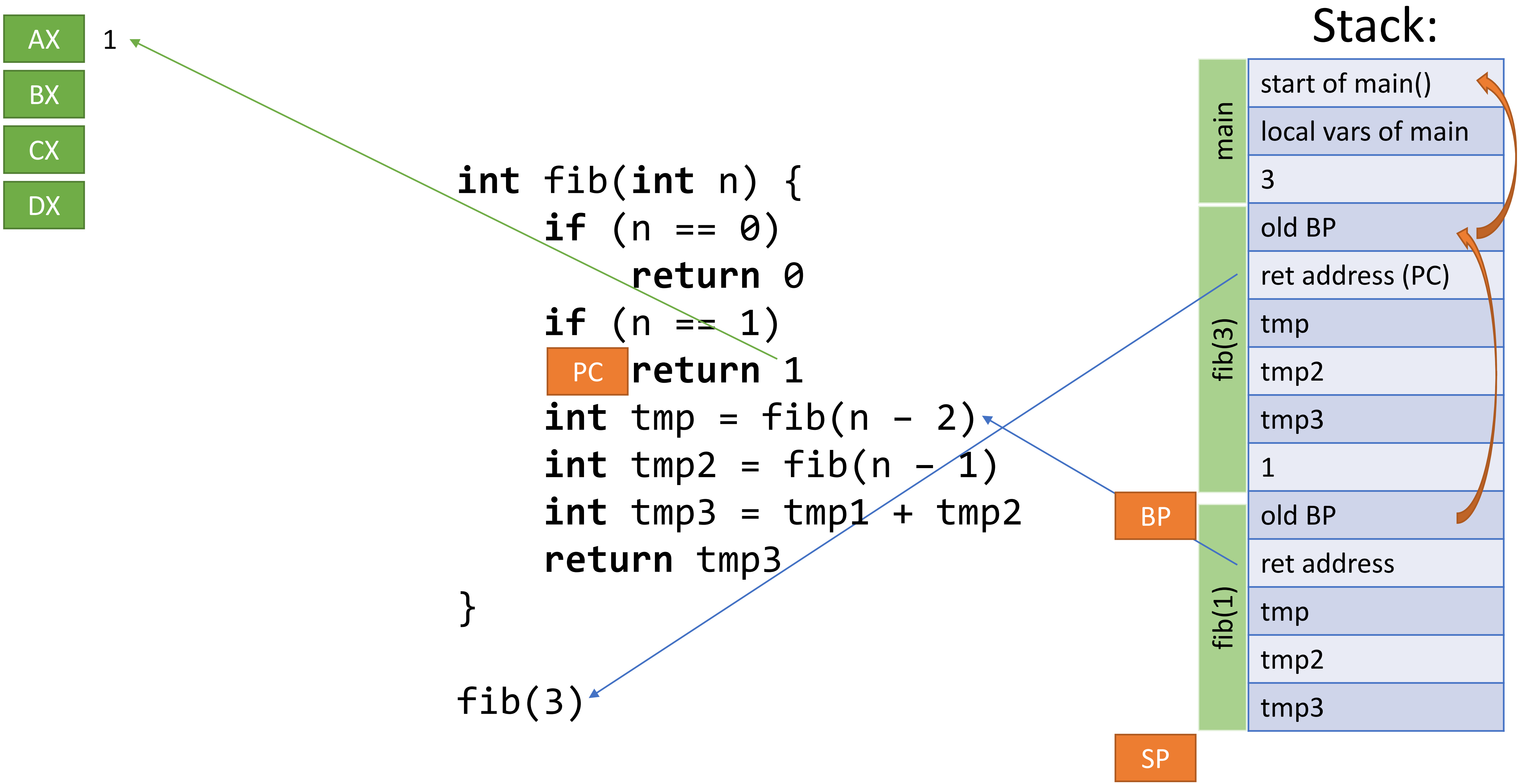
SP



```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:

start of main()
local vars of main
3
old BP
ret address (PC)
tmp
tmp2
tmp3
1
old BP
ret address
tmp
tmp2
tmp3



Stack:

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

AX 1
BX
CX
DX

PC

main
- start of main()
- local vars of main
- 3

fib(3)
- old BP
- ret address (PC)
- tmp
- tmp2
- tmp3
- 1

fib(1)
- old BP
- ret address
- tmp
- tmp2
- tmp3

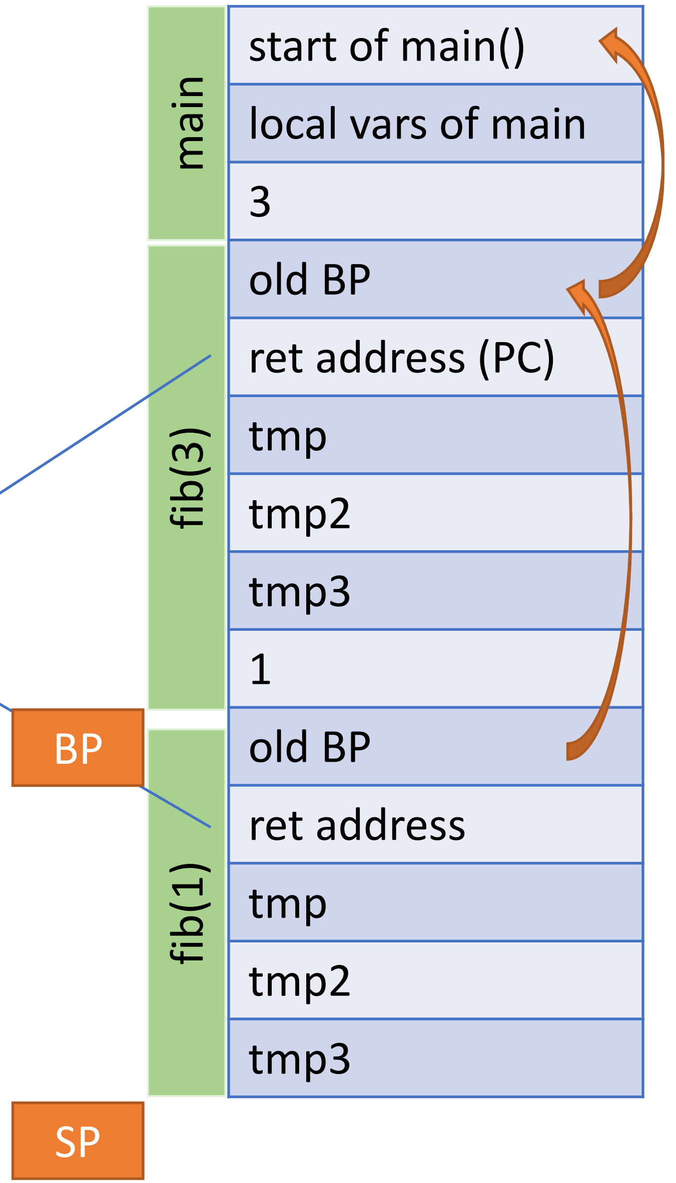
BP
SP

AX	1
BX	
CX	
DX	

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:

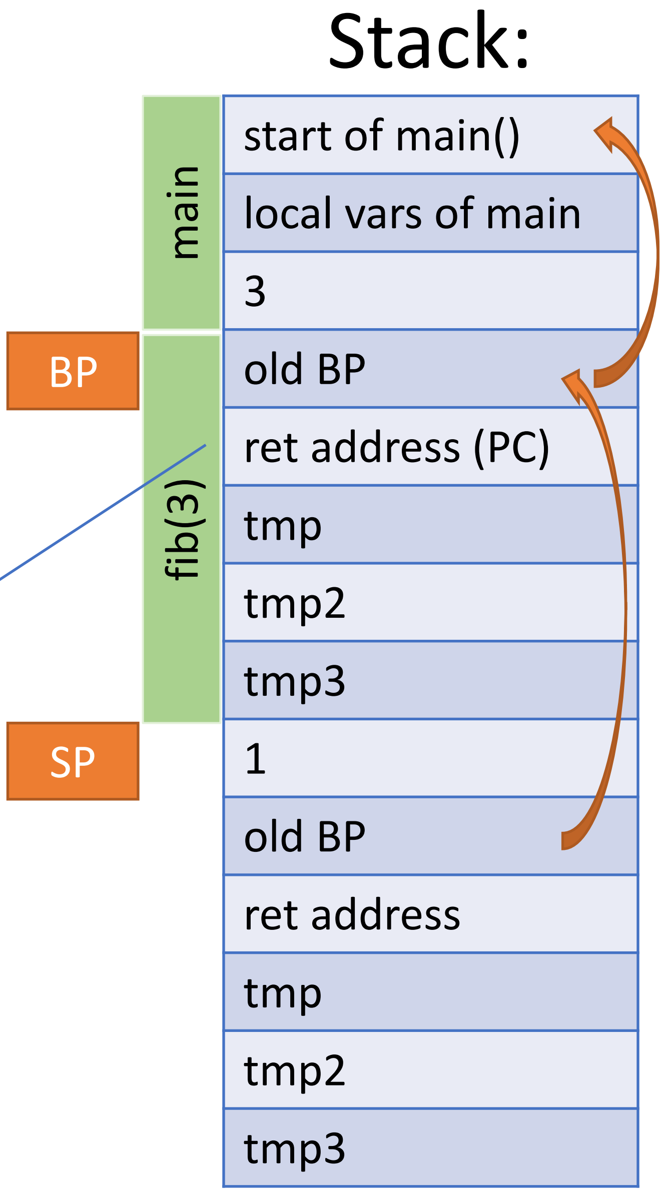


start of main()
local vars of main
3
old BP
ret address (PC)
tmp
tmp2
tmp3
1
old BP
ret address
tmp
tmp2
tmp3

AX	1
BX	
CX	
DX	

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



Stack:

start of main()

local vars of main

3

old BP

ret address (PC)

tmp

tmp2

tmp3

1

old BP

ret address

tmp

tmp2

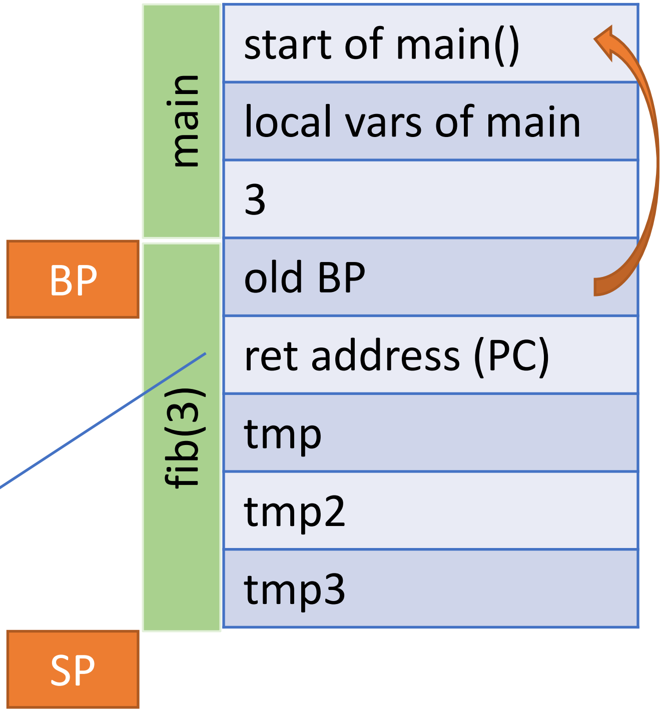
tmp3

AX	1
BX	
CX	
DX	

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main

start of main()

local vars of main

3

fib(3)

BP

old BP

ret address (PC)

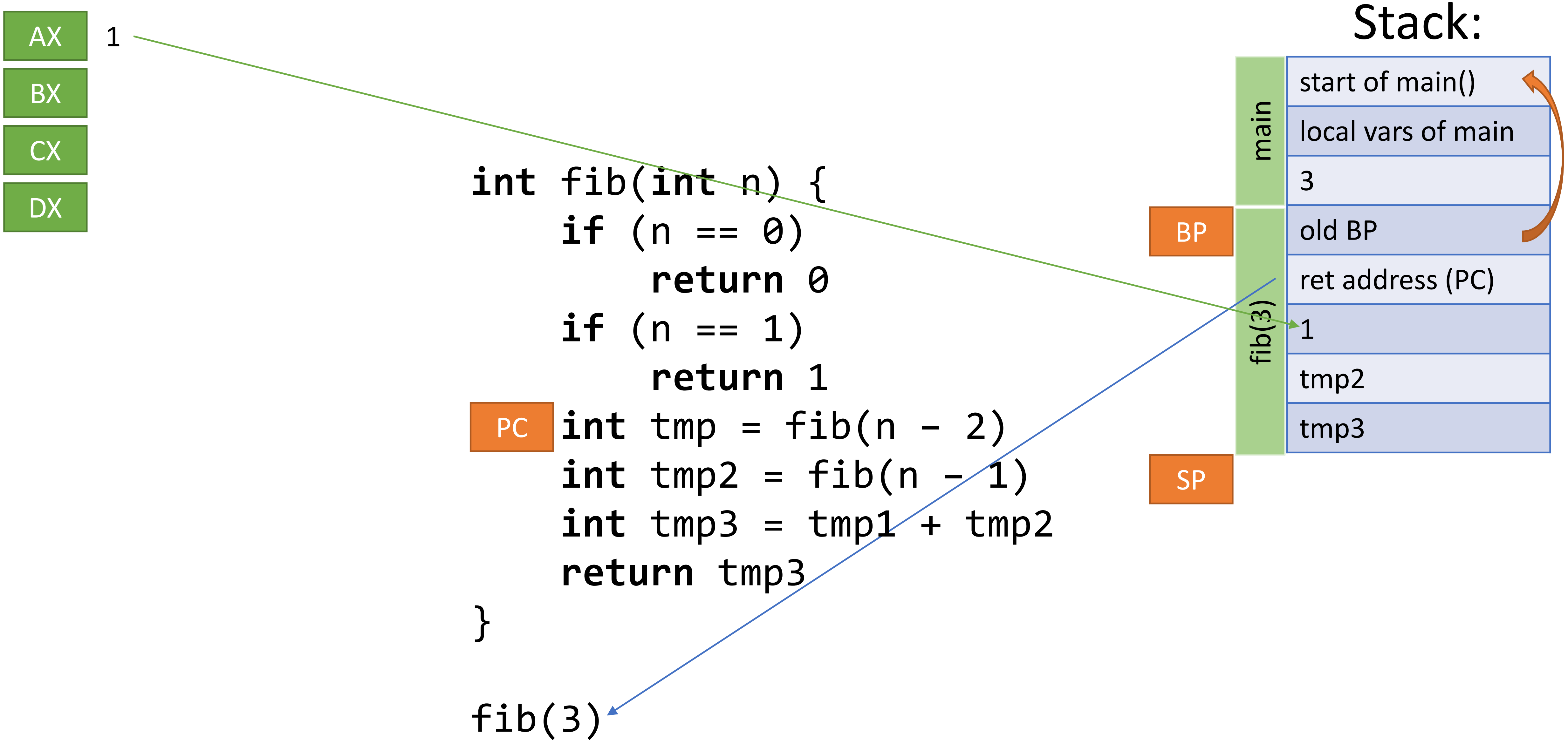
tmp

tmp2

tmp3

SP

Stack:



```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

AX

BX

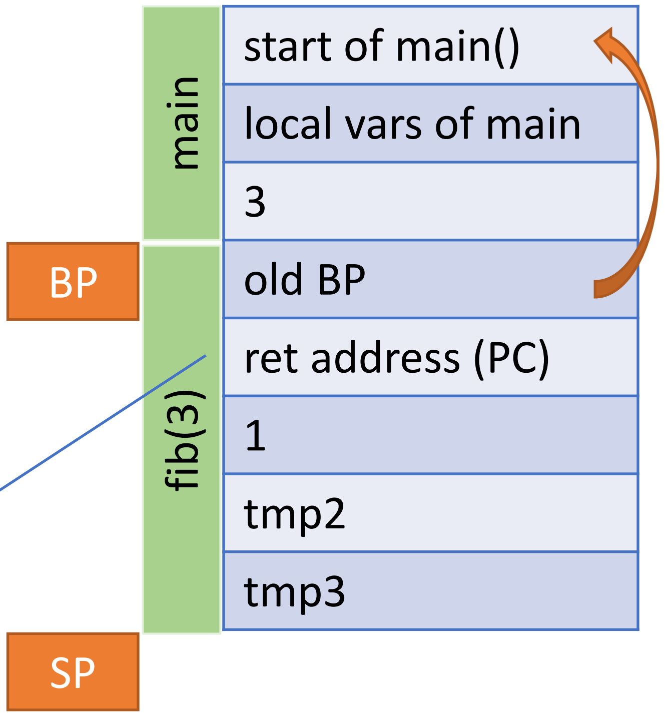
CX

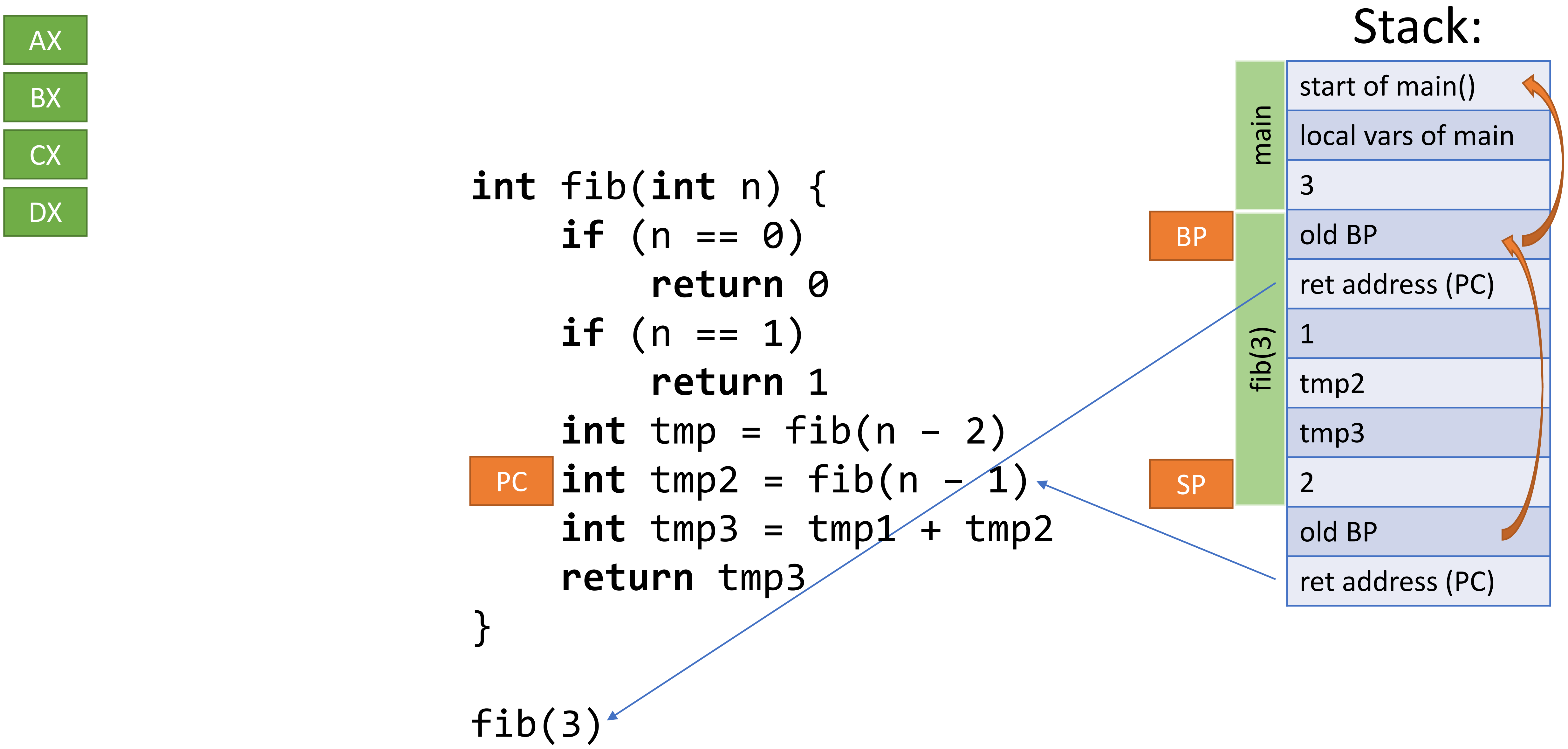
DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC

Stack:





```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:

AX

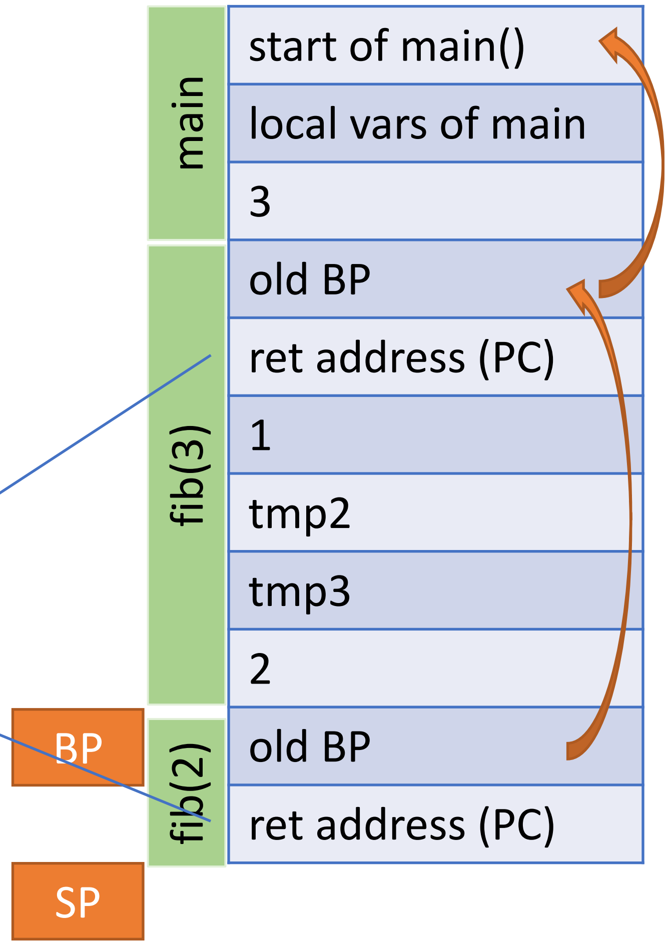
BX

CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	tmp2
	tmp3
	2
fib(2)	old BP
	ret address (PC)

AX

BX

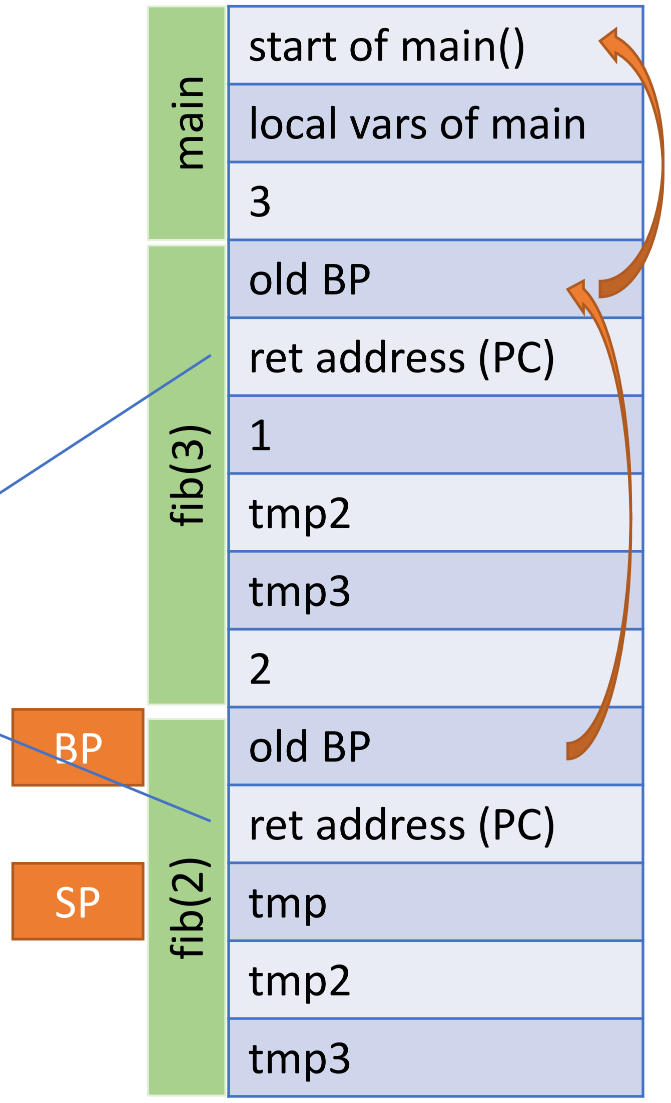
CX

DX

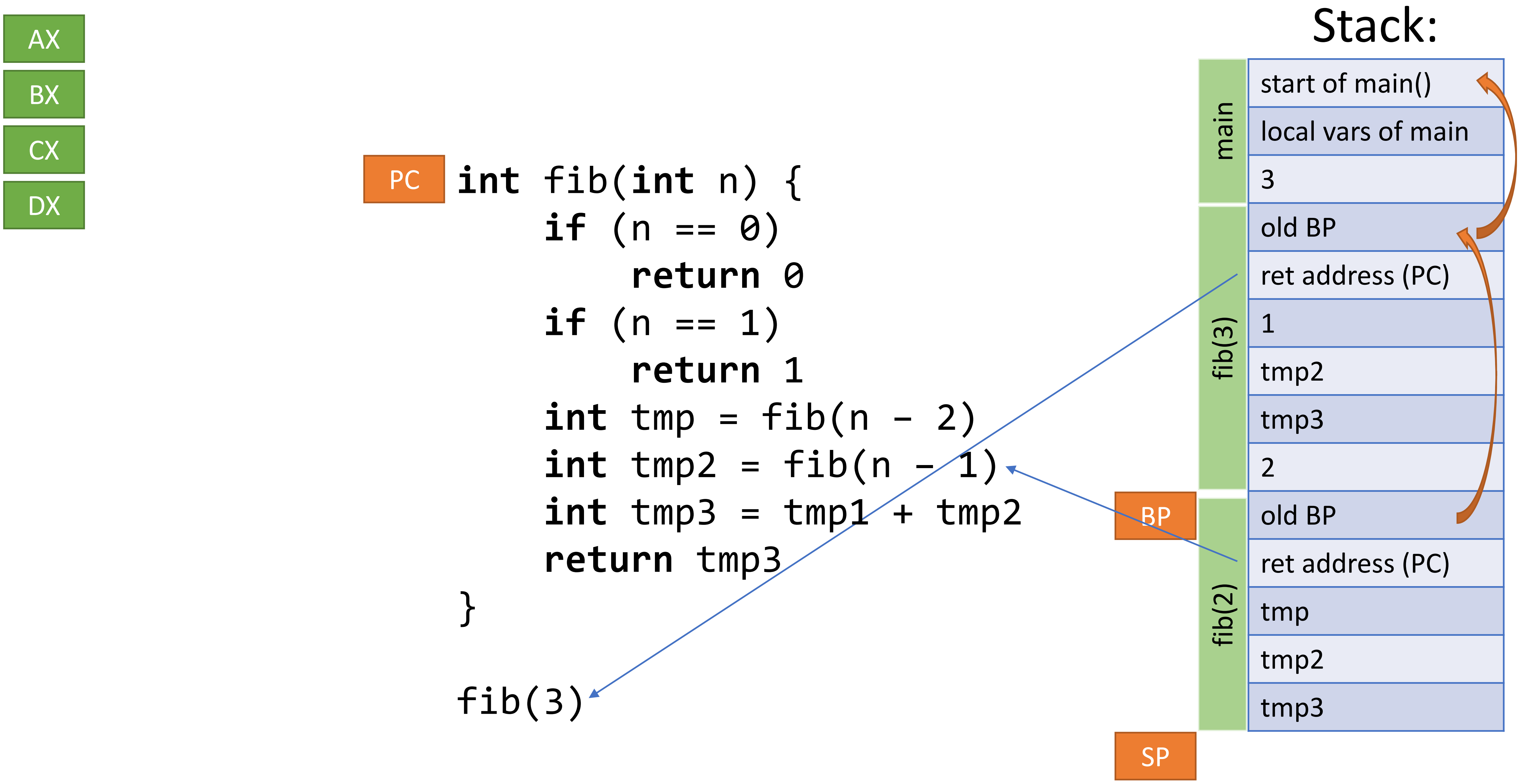
PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	tmp2
	tmp3
	2
fib(2)	old BP
	ret address (PC)
	tmp
	tmp2
	tmp3



```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:

main
start of main()
local vars of main

fib(3)
3
old BP
ret address (PC)
1
tmp2
tmp3
2

fib(2)
old BP
ret address (PC)
tmp
tmp2
tmp3

AX

BX

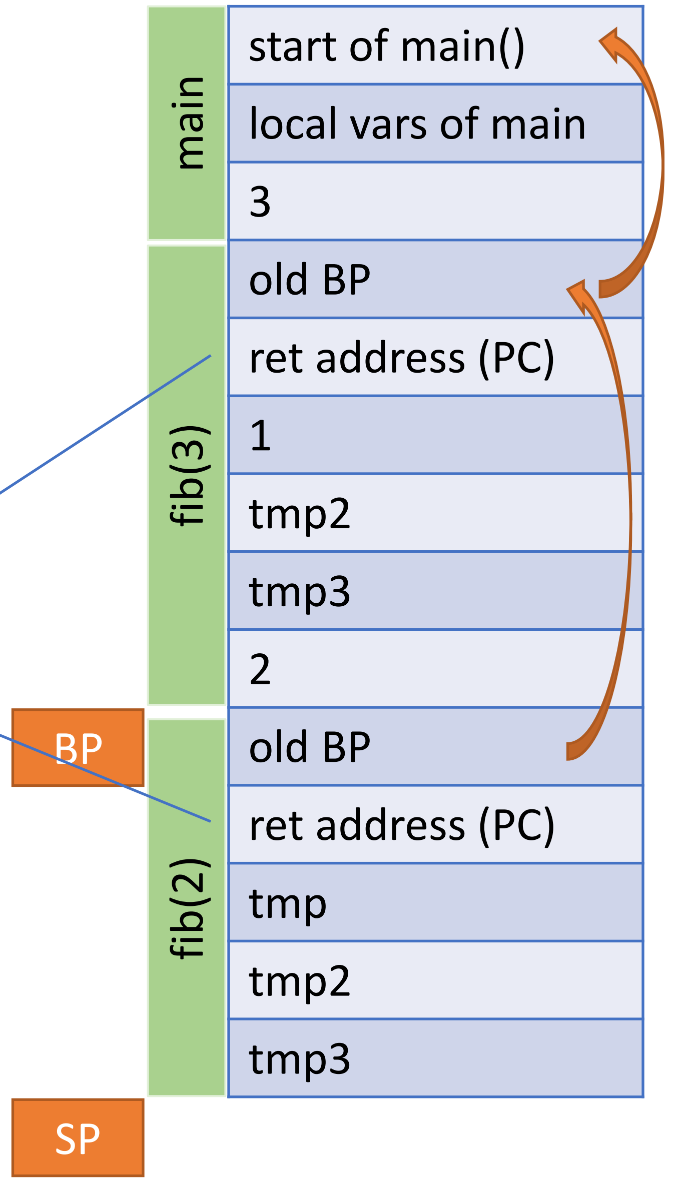
CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



start of main()

local vars of main

3

old BP

ret address (PC)

1

tmp2

tmp3

2

old BP

ret address (PC)

tmp

tmp2

tmp3

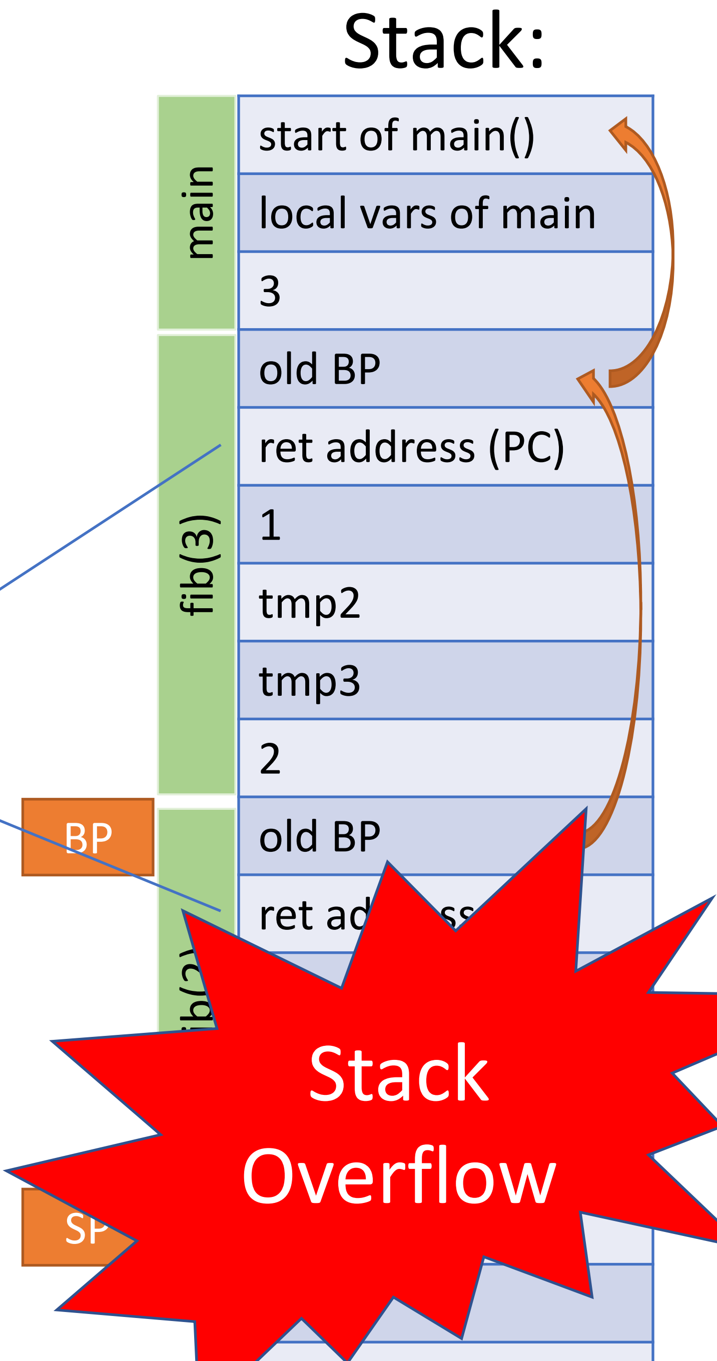
AX

BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
PC  int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



AX

BX

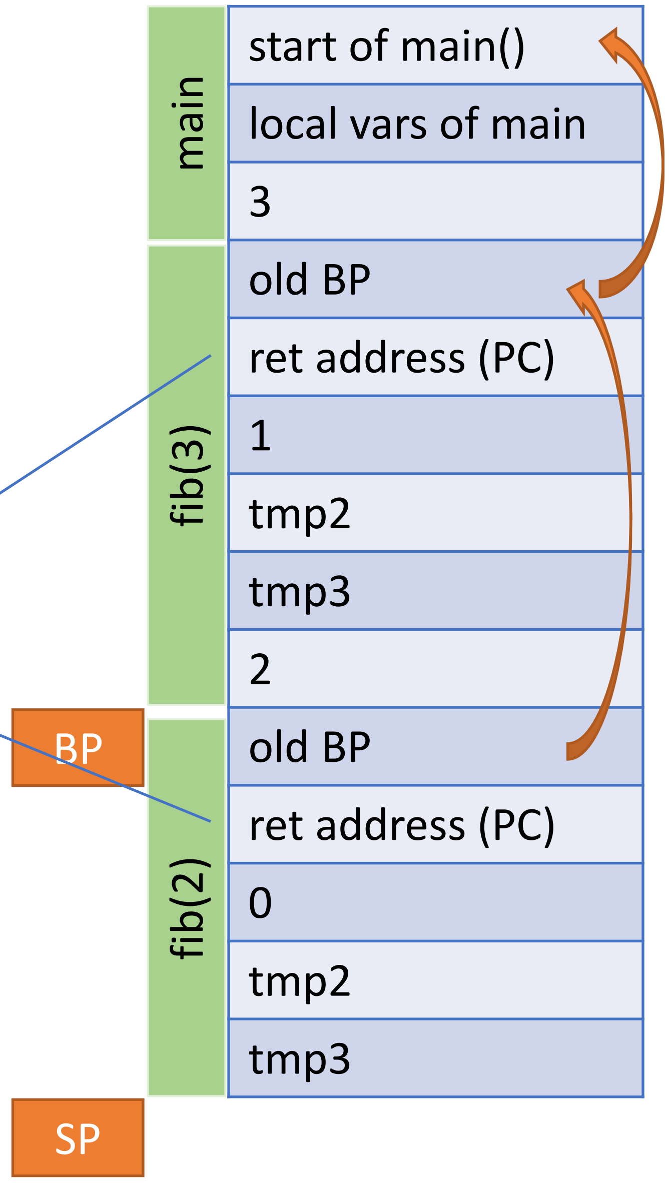
CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	tmp2
	tmp3
	2
fib(2)	old BP
	ret address (PC)
	0
	tmp2
	tmp3

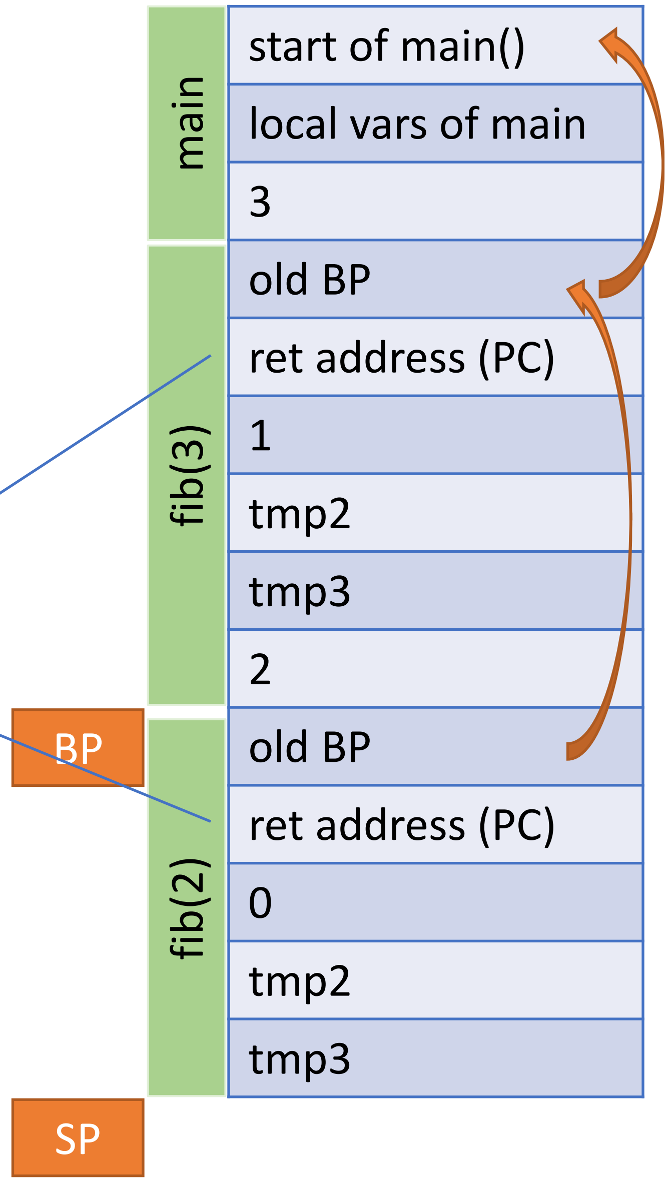
AX

BX

CX

DX

Stack:



main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	tmp2
	tmp3
	2
fib(2)	old BP
	ret address (PC)
	0
	tmp2
	tmp3

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC

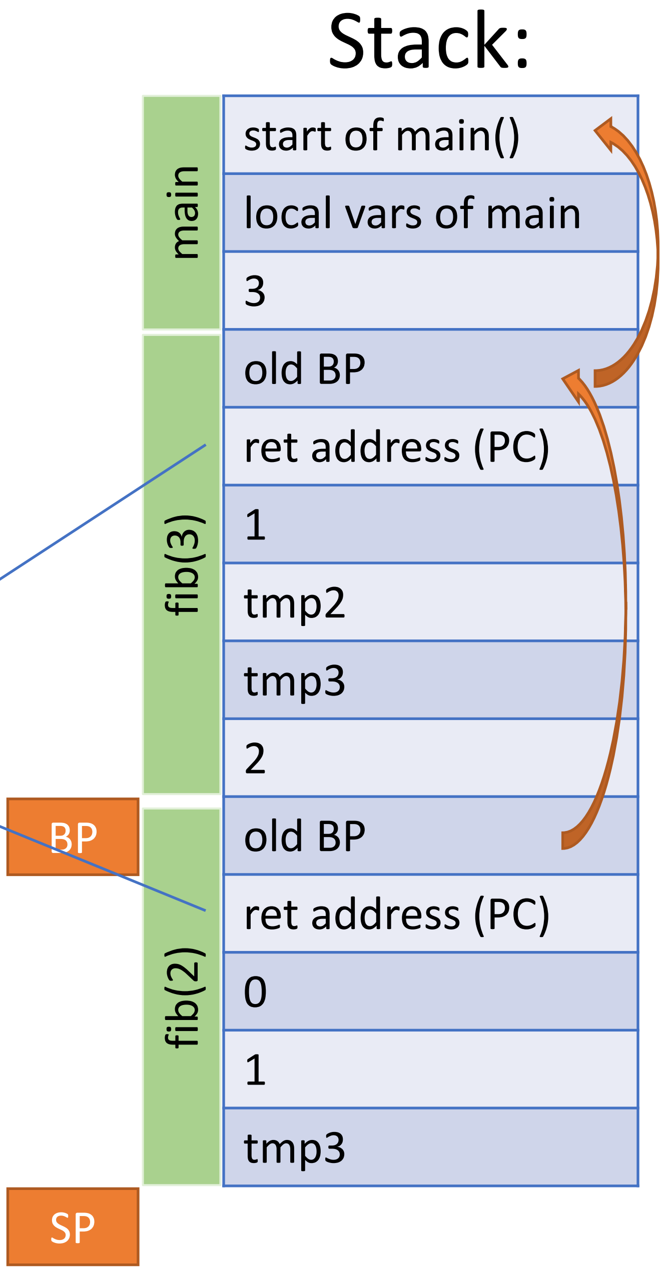
AX

BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
PC  int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



Stack:

main
- start of main()
- local vars of main
- 3

fib(3)
- old BP
- ret address (PC)
- 1
- tmp2
- tmp3
- 2

fib(2)
- old BP
- ret address (PC)
- 0
- 1
- tmp3

BP

SP

AX

BX

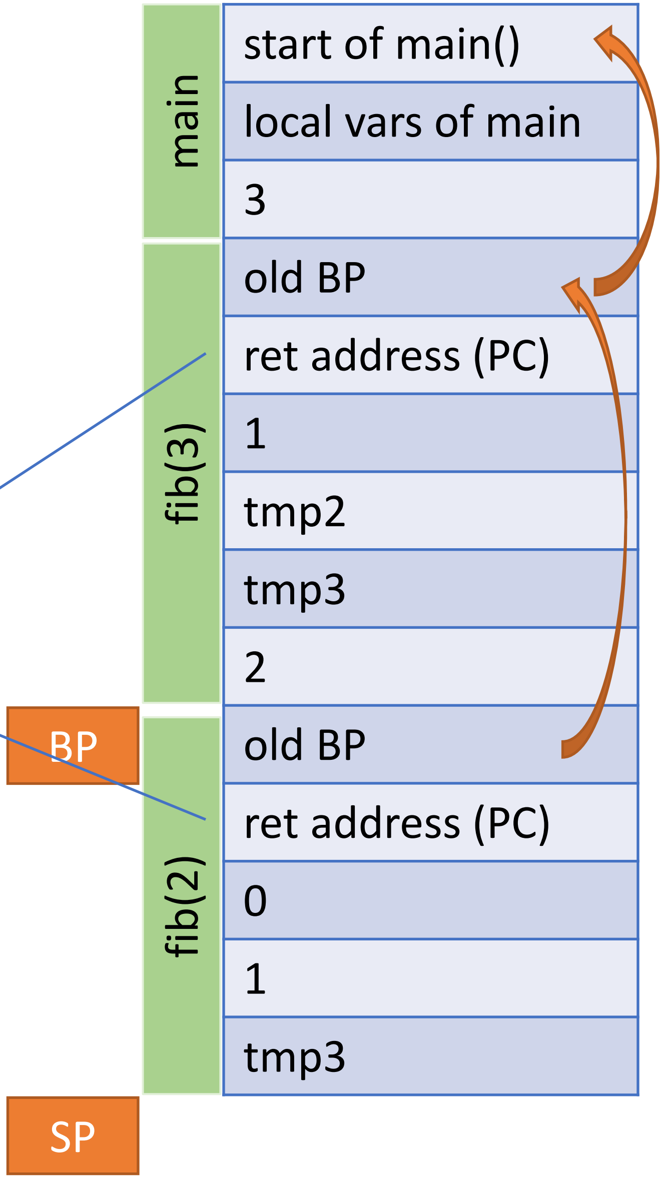
CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



main

start of main()

local vars of main

3

fib(3)

old BP

ret address (PC)

1

tmp2

tmp3

2

BP

fib(2)

old BP

ret address (PC)

0

1

tmp3

SP

AX

BX

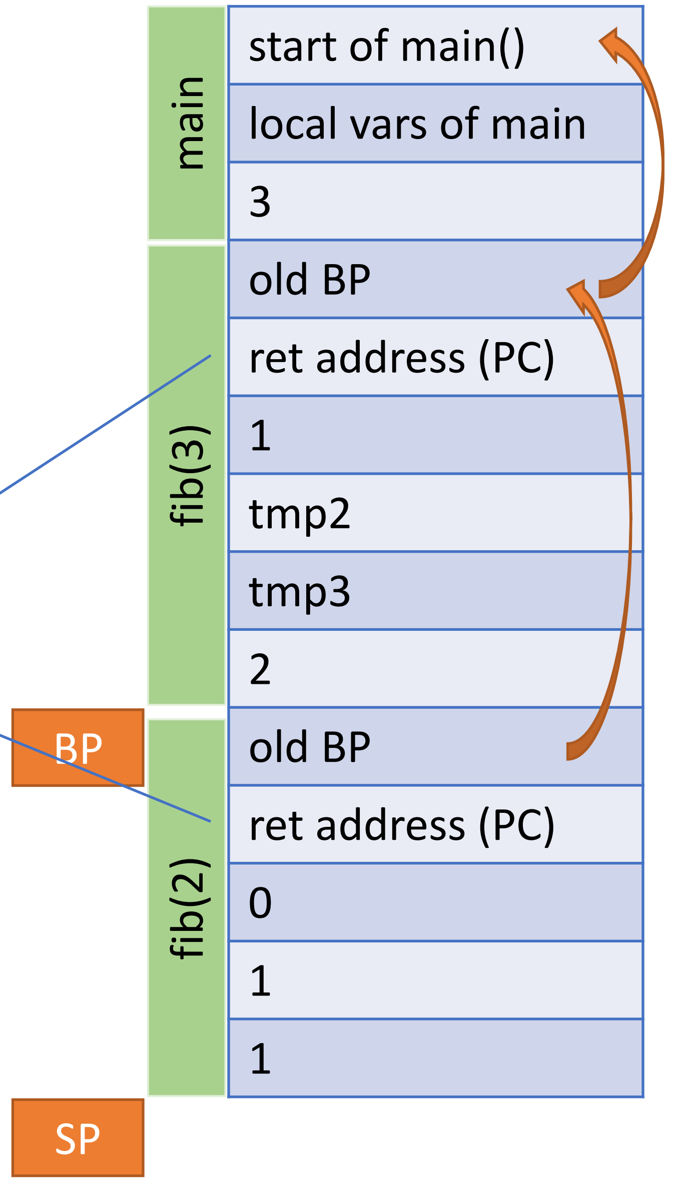
CX

DX

PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:



start of main()

local vars of main

3

old BP

ret address (PC)

1

tmp2

tmp3

2

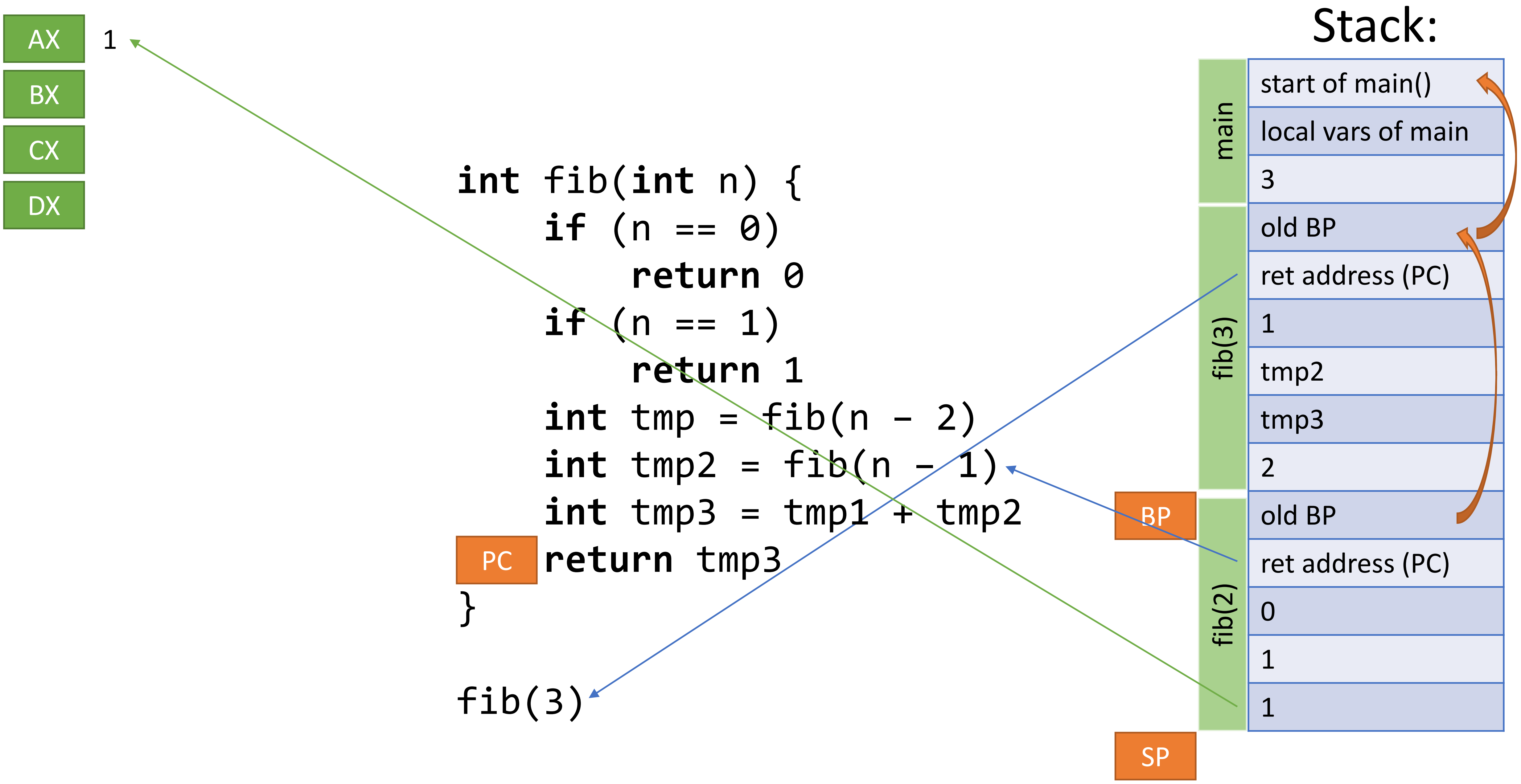
old BP

ret address (PC)

0

1

1



```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

Stack:

main
- start of main()
- local vars of main
- 3

fib(3)
- old BP
- ret address (PC)
- 1
- tmp2
- tmp3
- 2

fib(2)
- old BP
- ret address (PC)
- 0
- 1
- 1

AX 1

BX

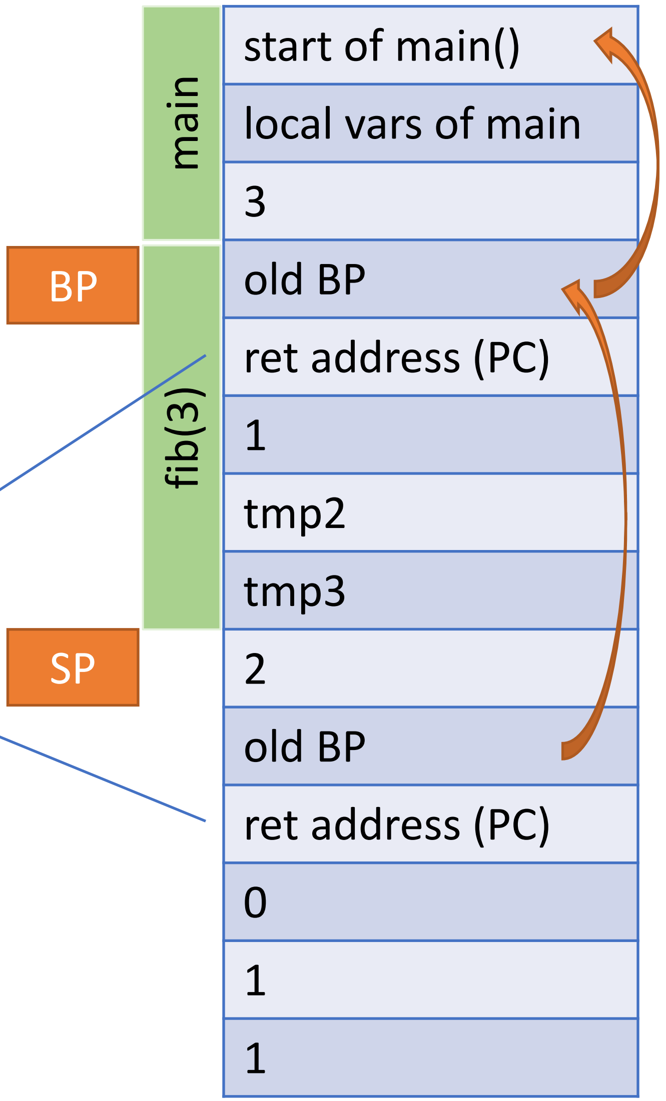
CX

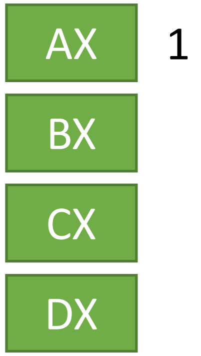
DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC

Stack:

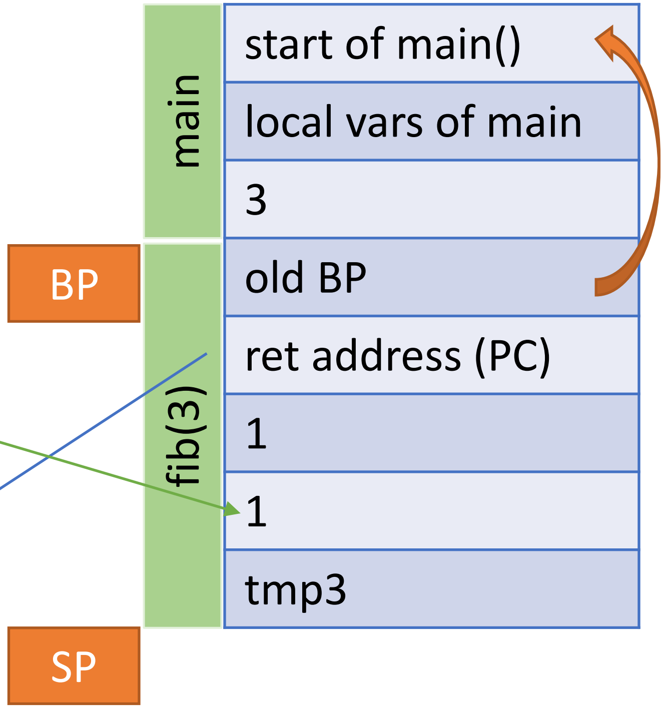


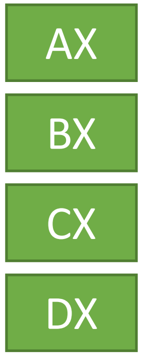


```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

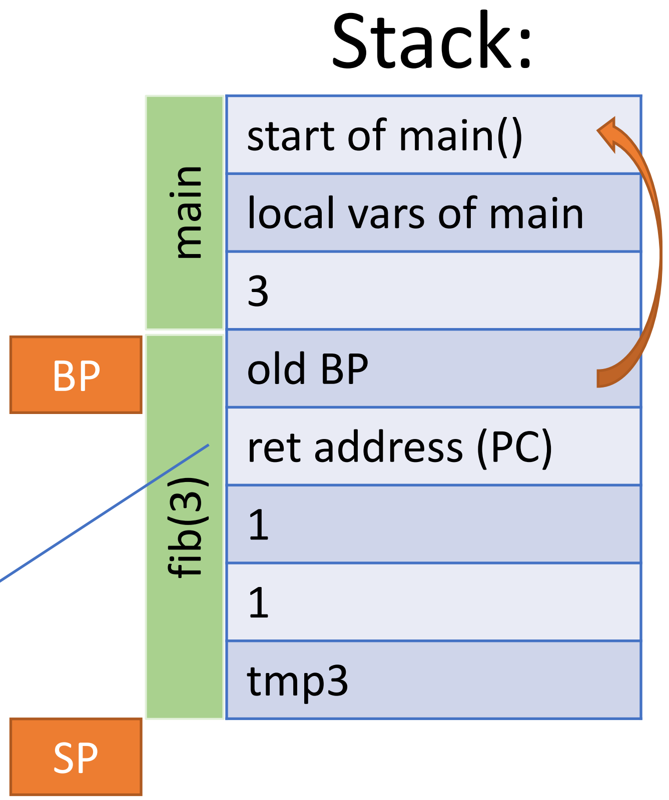
PC

Stack:





```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
PC  int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



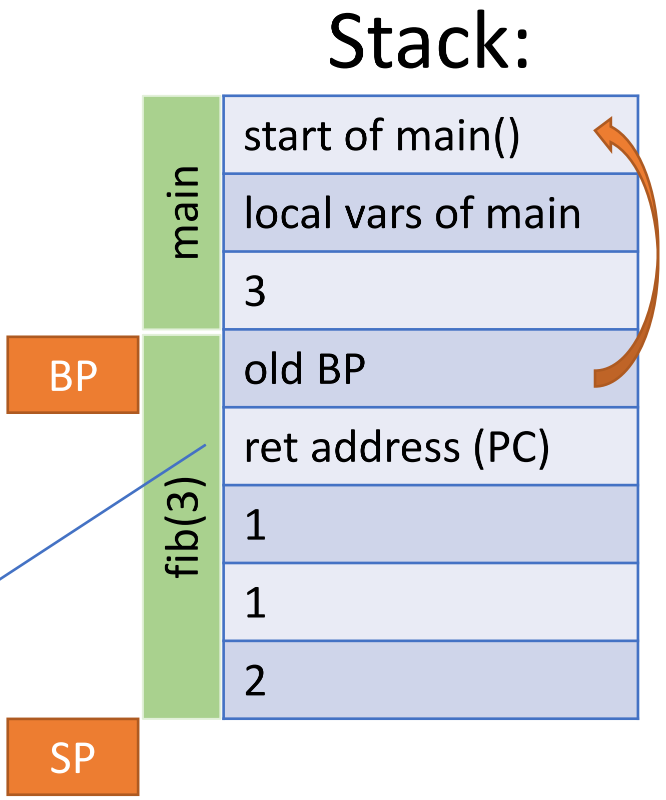
AX

BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
PC  int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	1
	2

BP

SP

AX

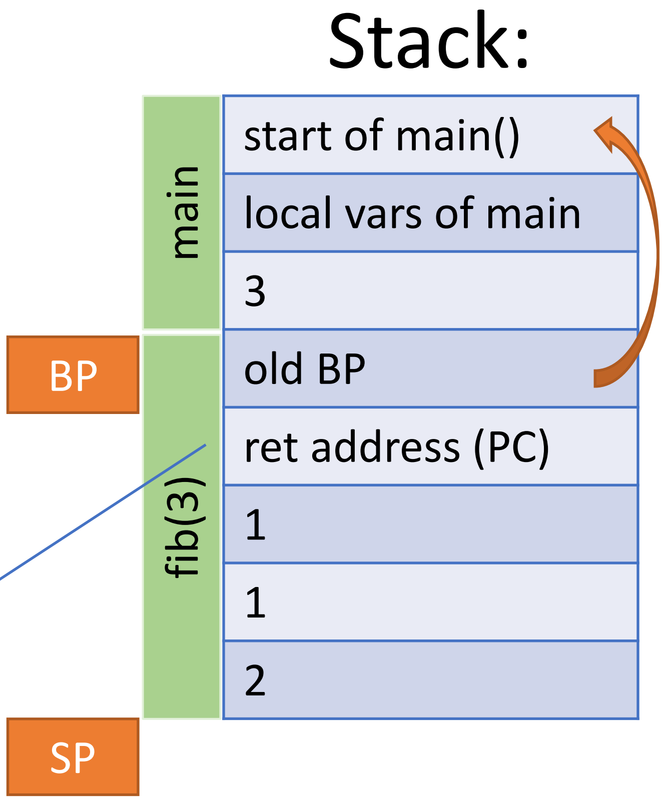
BX

CX

DX

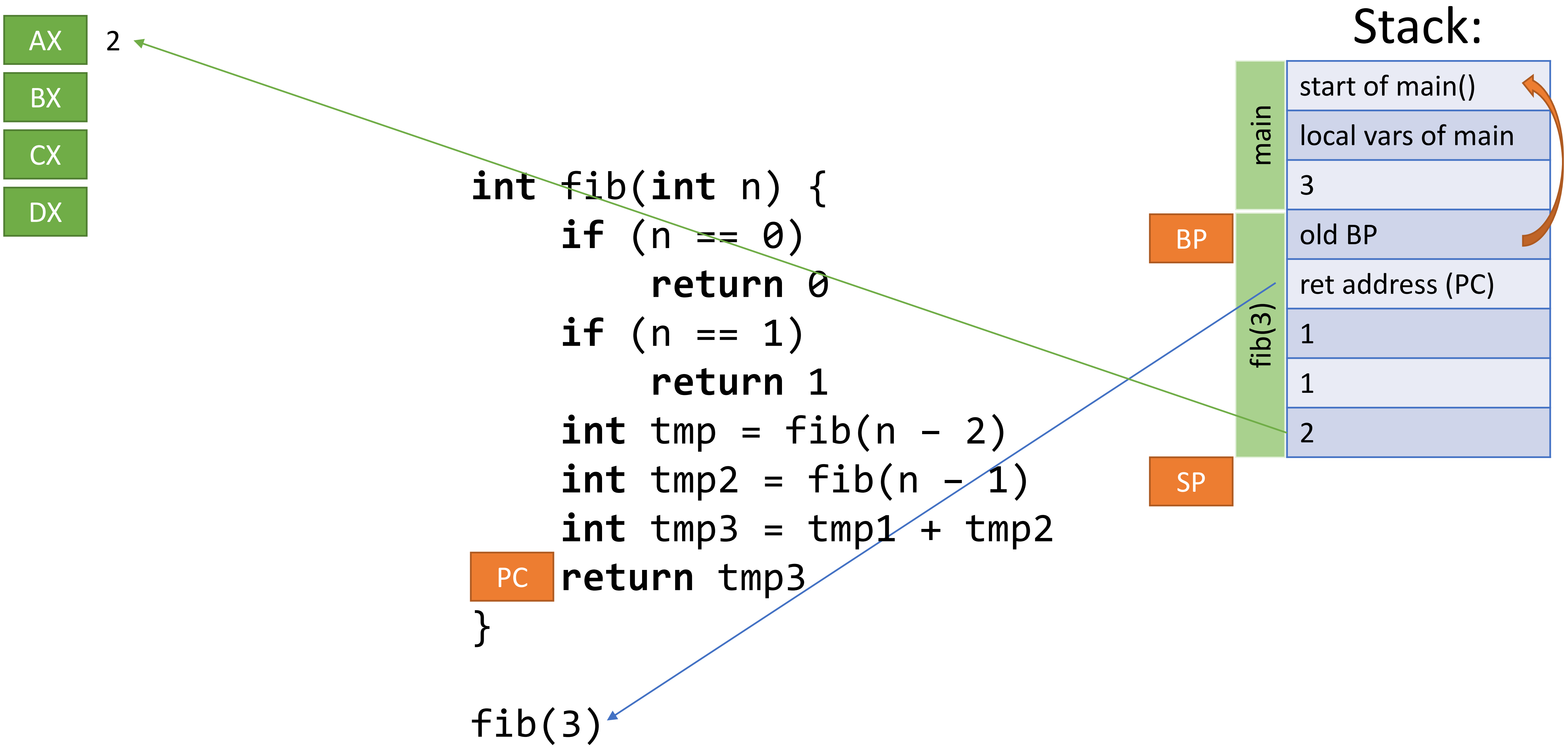
PC

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```



Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	1
	2



AX 2

BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}

fib(3)
```

PC

Stack:

main	start of main()
	local vars of main
	3
fib(3)	old BP
	ret address (PC)
	1
	1
	2

BP

SP

AX 2

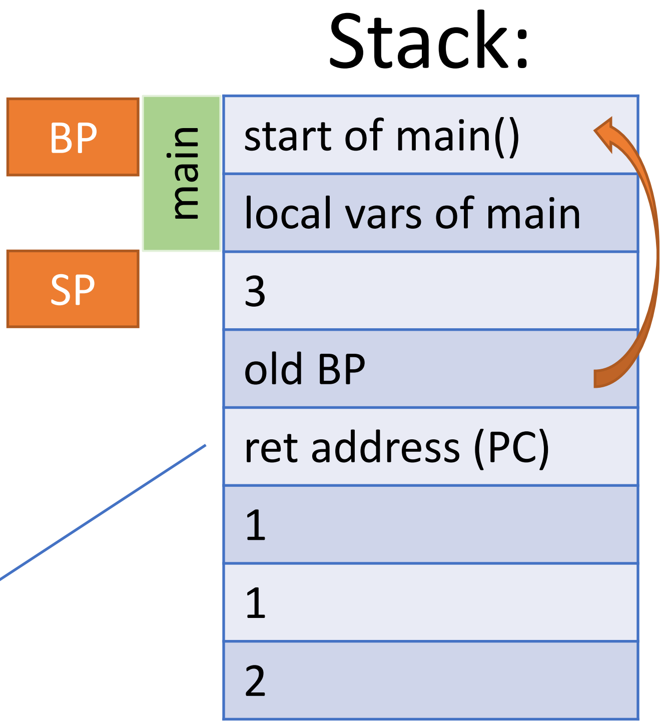
BX

CX

DX

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n - 2)
    int tmp2 = fib(n - 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}
```

PC fib(3)



Stack:

BP

SP

main

start of main()

local vars of main

3

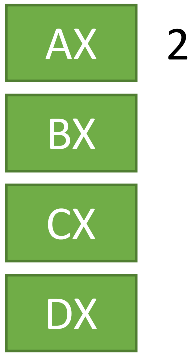
old BP

ret address (PC)

1

1

2



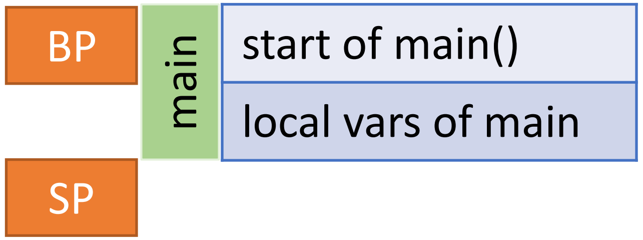
AX 2

BX

CX

DX

Stack:



BP

main

start of main()

local vars of main

SP

```
int fib(int n) {
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    int tmp = fib(n – 2)
    int tmp2 = fib(n – 1)
    int tmp3 = tmp1 + tmp2
    return tmp3
}
```

PC fib(3)

Call Stack

- stack frames used also for nested blocks
- elements on stack must have statically known sizes

why?

- slightly more complicated for languages that allow nested functions
  - stack frame saves the old BP as well as BP to the closest parent function
- when function returns data on stack cease to exist
  - this is **not** ok for other than local variables

Data Segment

- stores global variables
- has statically known size
- data in DS are valid throughout the duration of the program

Is this always true?

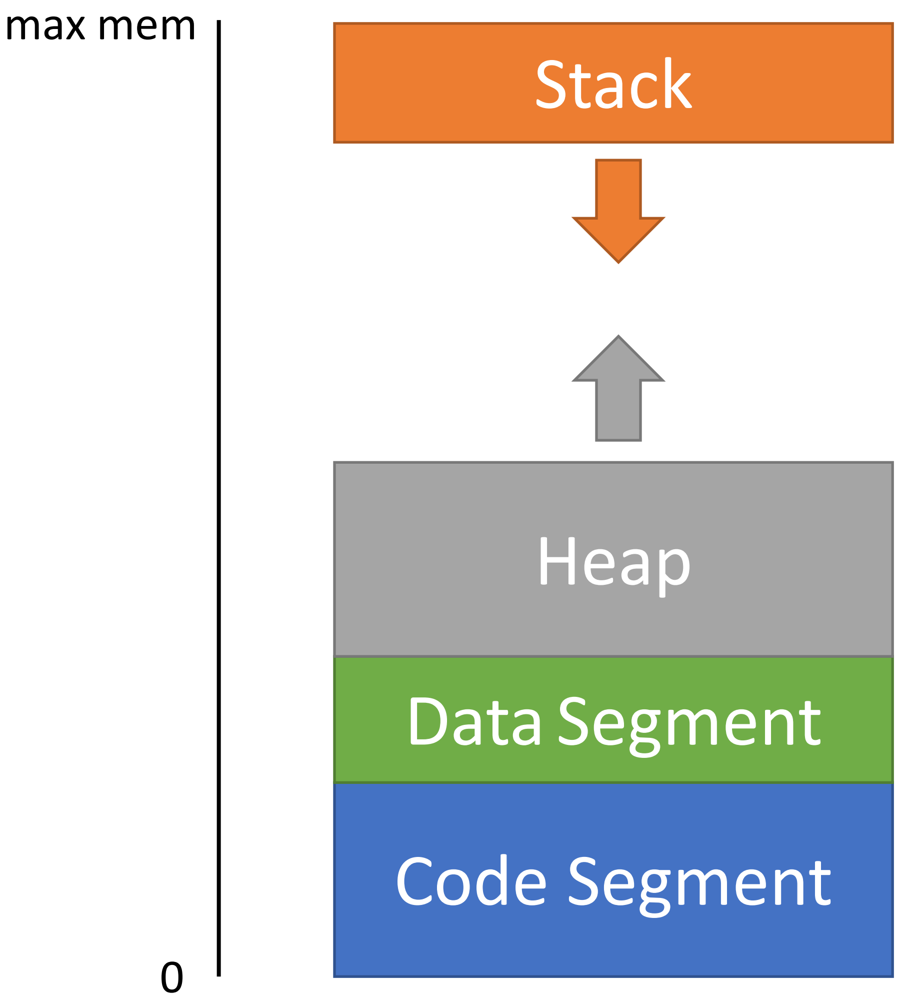
Data Segment

- stores global variables
- has statically known size ← But so does stack!!
  Where to put dynamically sized data?
- data in DS are valid throughout the duration of the program

Heap

- memory area for dynamically allocated data
- allows allocation & deallocation in arbitrary order
- manual or automatic (garbage collector) management
- dynamic size depending on the actual allocation at any given time (sort of)

Memory Layout



Calling Conventions

Calling Conventions

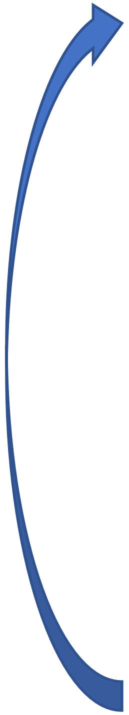
- how to pass arguments?
  - stack, registers, both, argument order
- how to return the result of the function?
  - stack, registers, type-dependent
- which registers are free for caller, and which for callee

- who is responsible for cleanup?
  - caller cleanup, callee cleanup
- architecture, OS, and language dependent

cdecl (x86)

- Microsoft’s C compiler, caller saved
- arguments on stack, right to left
- result in eax, st0, or stack
- eax, ecx, edx caller saved, rest callee saved

cdecl (x86)



```
push    ebp         ; save caller’s frame and start new frame
mov     ebp, esp
push    3           ; push the arguments left to right
push    2
push    1
call    f           ; call function f
add     esp, 12     ; remove arguments from frame (caller saved)
add     eax, 1      ; result returned in eax (if integer)
mov     esp, ebp    ; ebp is callee cleaned (call to itself!)
pop     ebp
ret
```

cdecl (x86)

- sometimes the bp is updated by callee (BP - addressing in callee, like our example)
- larger results returned on stack (caller allocated)
- results in more registers (eax:edx typically)
- stack alignment considerations

Other x86 Calling Conventions

- fastcall (arguments in registers, callee cleanup)
- thiscall (this pushed on stack as first argument, some variants in ecx, caller cleanup)
- stdcall (callee cleanup, right to left args, Win32API)
- x86_64 calling conventions (Microsoft vs the world)

ARM (A32)

- link register (LR) for return address
  - faster than stack for calls to leaf subroutines
  - not really that necessary with inlining

- more registers
  - r0, r1, r2, r3 = arguments (callee saved)
  - r4 – r11 = local variables (callee saved)

ARM64

- even more registers
- r19-r29 callee saved
- r9 – r15 caller saved
- r0-r7 arguments and results

Target Architectures

Target Architectures

- x86, x86-64, ARM, RISC-V, MIPS, Itanium, Sparc, AVR, …, …
- RISC, CISC, EPIC
- all have memory, registers, instructions

1960

CISC

- memory is fast(!), and pricey
- clock speed is function of available technology only, cpu size & power is not an issue
- since there are transistors to spare, complex CPU instructions are possible and they greatly speed up the program and lower the memory requirements

1960

CISC

- reg-mem arithmetics
- very complex addressing modes
- special instructions to support calls, control flow and other higher level language features
- variable instruction length

1980

CISC

- memory is fast(!), and pricey
- clock speed is function of available technology only, cpu size & power is not an issue
- since there are transistors to spare, complex CPU instructions are possible and they greatly speed up the program and lower the memory requirements

1980

CISC

- memory is slow and cheap
- clock speed is largely function of cpu size & power
- cpu features on die compete with caches, pipeline stages, super-scalar ALUs, etc., there are no transistors to spare, complex instructions make the whole CPU slower

1980

RISC

- reg-reg arithmetics
- simple addressing modes
- fixed instruction length
- highly regular decoding

SuperScalar Processors

- as instruction is processed by the CPU, different parts are utilized

- instead of idling the unused circuits, more instructions can be processed at the same time

- instruction pipeline

Pipeline Stalls

- pipeline only helps if it is full

- branches, more complex ALU operations, data dependencies, memory accesses may cause holes (stalls) in the pipeline

- the instructions must be scheduled to minimize these

1980

RISC

- small, fast to execute instructions soon reached the limit of 1 IPC
- clock rates limited by technology & power
- can we go below 1 IPC?

VLIW

- Very Large Instruction Words

- large instructions encoding multiple smaller operations

- that are executed in parallel

- (the compiler schedules the operations into the VLIWs ahead of time)

EPIC

- non-determinism of memory access is a problem for VLIW
- Mitigated by EPIC (Explicitly Parallel Instruction Computing) where instructions and continuation logic is put by compiler into bundles
- explicit prefetching, explicit speculative pre-loading, predicated execution all used to make the bundles into more deterministic sequence

Intel Itanium

The Awesome Compiler

The Awesome Compiler

Not so much…

Super-Super Processors

- as superscalar or EPIC processors got more and more complex and powerful, the scheduling demands placed on the compiler were too great

- multiple execution units per stage were added to lower the congestion

- the super-super architecture

SMT

- often computers do more than one thing anyways
- SMT (Simultaneous Multithreading) makes multiple “cores” share parts of the execution units
- better utilization of the resources
- no thread context switching required

Hyperthreading

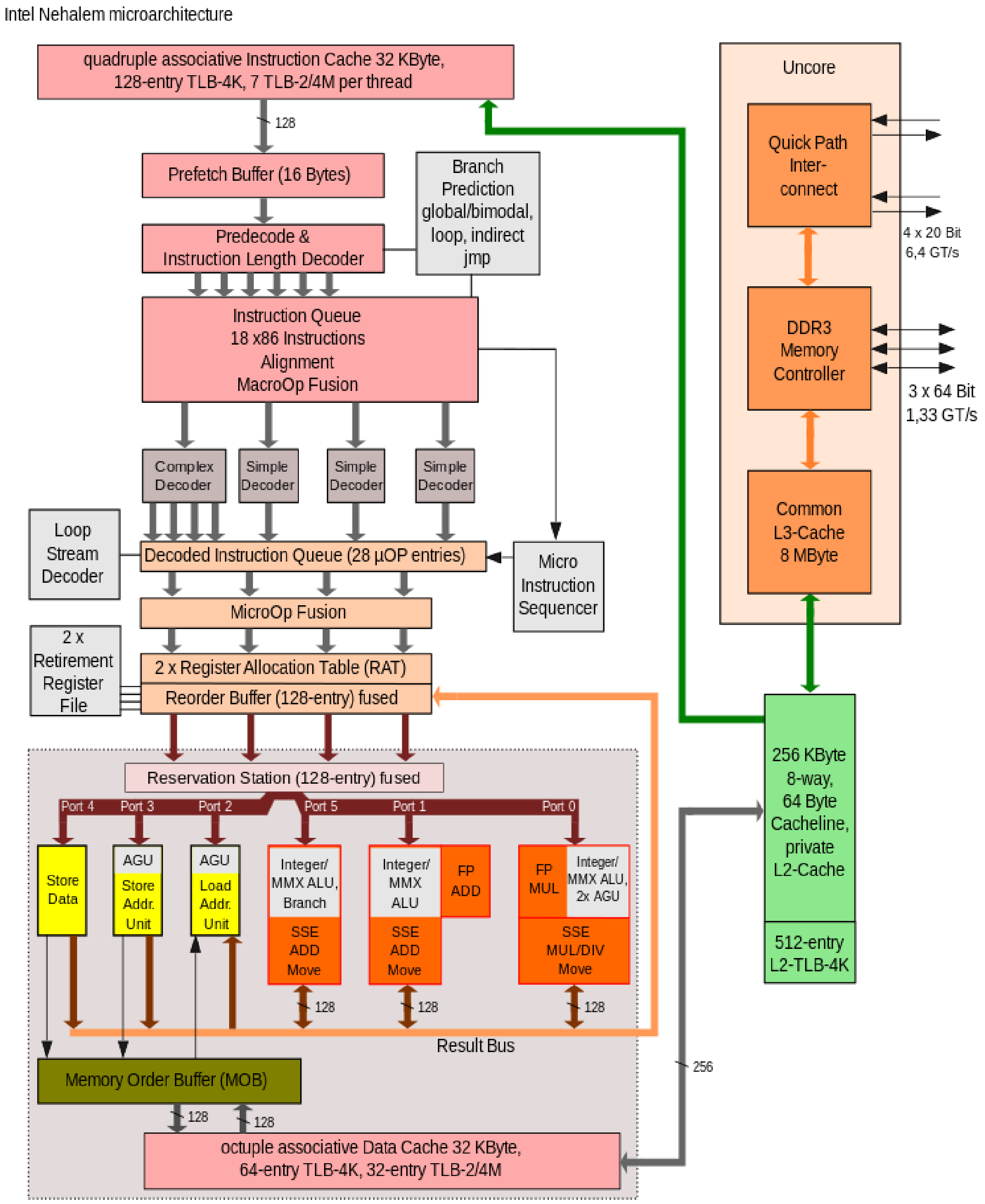
Out of Order Execution

- CPU keeps a pool of fetched instructions

- these are issued dynamically not according to their order in program, but availability of required execution units and inputs

- requires complex bookkeeping to preserve sequential semantics

- makes compiler scheduling much less important


Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU
Store Addr. Unit
AGU
Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
FP ADD
SSE ADD Move
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit
6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit
1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

GT/s: gigatransfers per second

https://en.wikipedia.org/wiki/Nehalem_(microarchitecture)#/media/File:Intel_Nehalem_arch.svg

Practical ISA

```
struct pos {
    int32_t x;
    int32_t y;
};
pos items[128];


int32_t j = items[i].y;
```

```
// r1 contains & items
// r2 contains i

ldi r3, r2
shr r3, 3
add r3, r1
ldi r4, 4
add r3, r4
load r4, r3
```

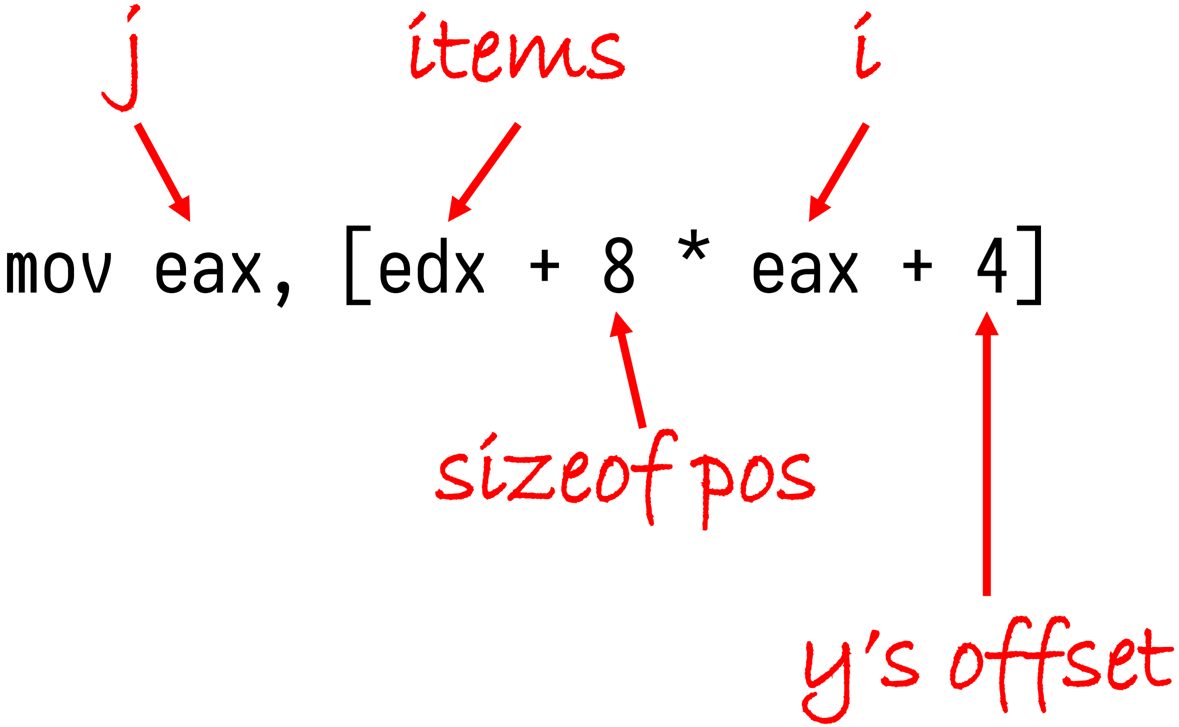
* by 8 (sizeof pos)

load from memory

```
struct pos {
    int32_t x;
    int32_t y;
};
pos items[128];
```

```
// edx contains & items
// eax contains i
```

```
int32_t j = items[i].y;
```



```
mov eax, [edx + 8 * eax + 4]
```

```
struct pos {
    int32_t x;                   // r1 contains & items
    int32_t y;                   // r2 contains i
};
pos items[128];                  ldi r3, r2
                                 shr r3, 3
                                 add r3, r1
int32_t * j = & items[i].y;      ldi r4, 4
                                 add r3, r4
```

r3 contains the address

```
struct pos {
    int32_t x;
    int32_t y;
};
pos items[128];
```

```
int32_t * j = & items[i].y;  lea eax, [edx + 8 * eax + 4]
```

Fun with CISC

- complex instructions not created equal
- some are deprecated (made slower), or made faster
- often complex instructions can be used for surprising purposes
- consider lea

```
lea ecx, [edx + 8 * eax + 4]
lea ecx, [edx + 8 * eax + 4]

mov ecx, eax
shl ecx, 3
add ecx, 4
add ecx, edx
```

Code Size Matters

- variable length encodings even on RISC ISAs
- immediate argument sizes
- relative branch distances
- not all registers created equal

Further Reading

https://medium.com/swlh/what-does-risc-and-cisc-mean-in-2020-7b4d42c9a9de