# Software Security 02

Sven Schäge

# Learning Goals

- Recap: Compilation Process
- Recap: How do computers work?
  - Reason about neccessities of programming languages and differences to machine code.
  - Reason about the necessity and power of pointers.
- Modular Programming – Nesting of Functions
  - What does this entail on modern architectures?
  - Understand the concept of stack frames and use it to reason about values stored on the stack.
  - Understand the heap memory and the difference to the stack.

# The Compilation Process

Sven Schäge

# Compiler & Machine Code

- A compiler is a program that turns code written in a programming language into code that can be executed by computers.
- Such code is called machine/binary code (hexadecimal numbers in light grey).
- Assembler code translates almost directly into binary code (in color).

Source Code

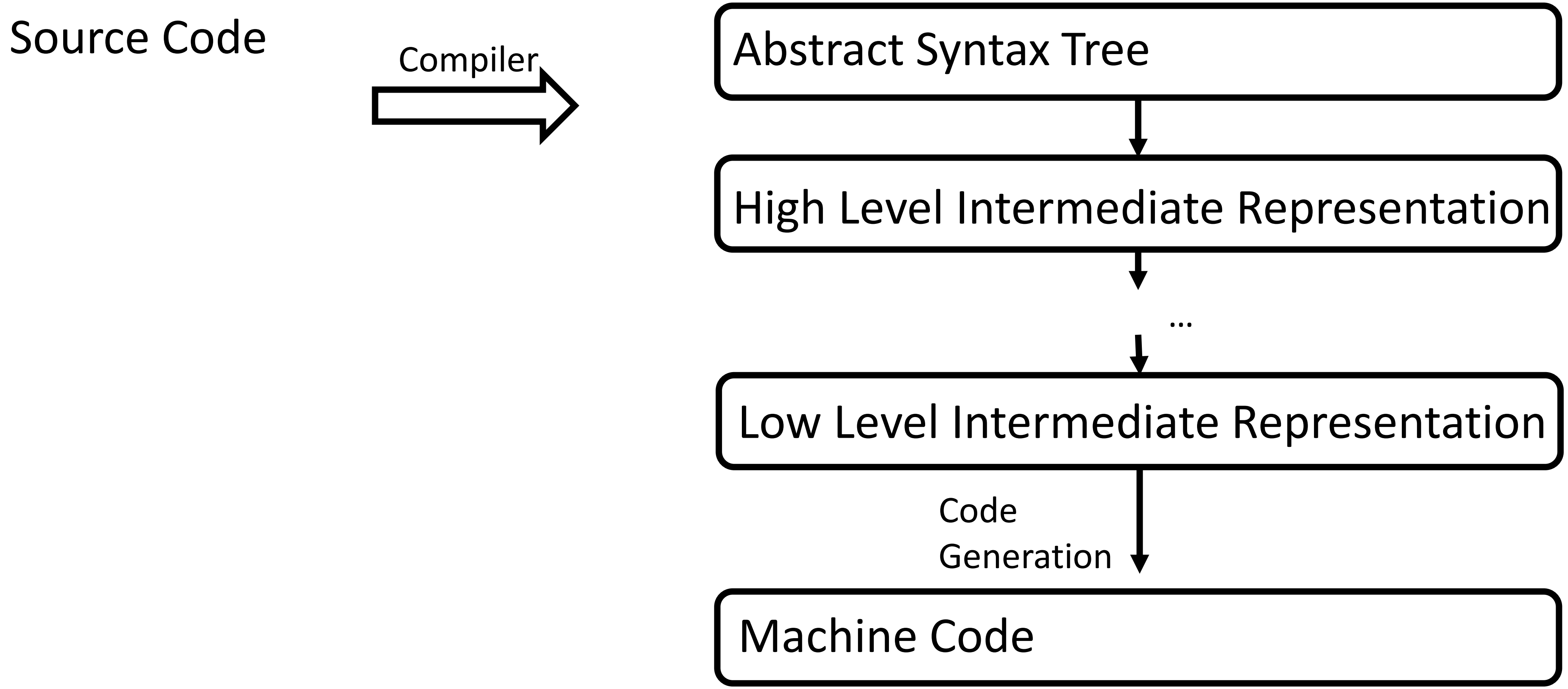
```c
void main()
{
    int j=8;
    int i=square(j);

}

int square(int num) {
    return num * num;
}
```

Compiler

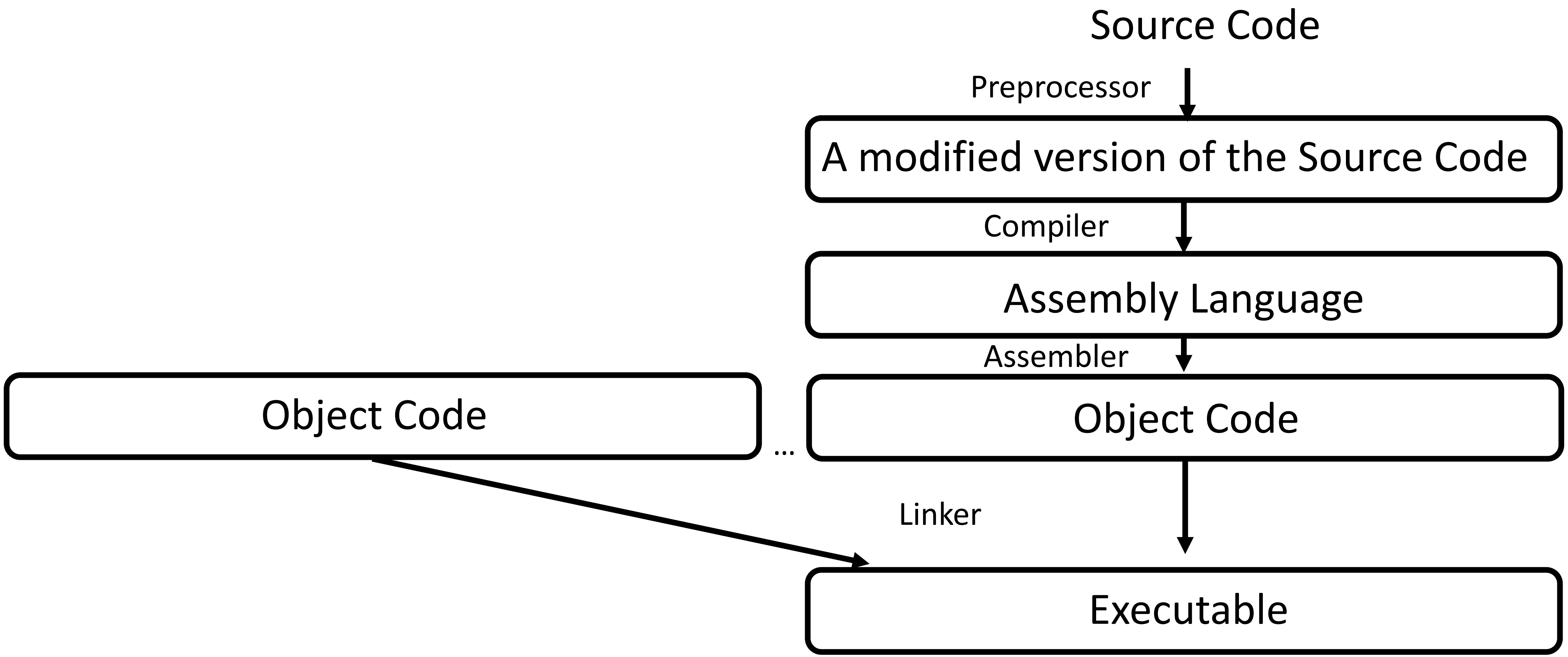
Binary Code/Disassembled Binary

```
        main:
         55
401106   push   rbp
         48 89 e5
401107   mov    rbp,rsp
         48 83 ec 10
40110a   sub    rsp,0x10
         c7 45 fc 08 00 00 00
40110e   mov    DWORD PTR [rbp-0x4],0x8
         8b 45 fc
401115   mov    eax,DWORD PTR [rbp-0x4]
         89 c7
401118   mov    edi,eax
         b8 00 00 00 00
40111a   mov    eax,0x0
         e8 06 00 00 00
40111f   call   40112a <square>
         89 45 f8
401124   mov    DWORD PTR [rbp-0x8],eax
         90
401127   nop
         c9
401128   leave
         c3
401129   ret
        square:
         55
40112a   push   rbp
         48 89 e5
40112b   mov    rbp,rsp
         89 7d fc
40112e   mov    DWORD PTR [rbp-0x4],edi
         8b 45 fc
401131   mov    eax,DWORD PTR [rbp-0x4]
         0f af c0
401134   imul   eax,eax
         5d
401137   pop    rbp
         c3
401138   ret
         0f 1f 80 00 00 00 00
401139   nop    DWORD PTR [rax+0x0]
```

# Compilation Process (in General)



# Compilation Process (Toolchain)



# Code Generation

- Binary code looks different!
- Compilers often **add** pieces of code
  - Code to save processor registers before jumping to subroutine
  - Start-up code to include libraries / initialize runtime
- Compilers can also **remove** specific information
  - Loop unrolling or inlining for optimization
  - Remove variable names

# Machine Language

Sven Schäge

# Machine Language

- Machine/**binary** code consists of a sequence of elementary machine instructions
- A CPU can only directly execute machine instructions
- Set of all possible machine instructions is called instructions set architecture (ISA)
- Machine language is a **textual** representation of machine code
- Machine language is architecture-specific: (x86 vs. Intel/AMD)

# Machine language is architecture-specific: (x86/Intel vs. ARM)

## Source Code

```c
void main()
{
    int j=8;
    int i=square(j);

}

int square(int num) {
    return num * num;
}
```

machine code in grey

## x84-64 gcc 11.2

```
main:
        55
401106  push   rbp
        48 89 e5
401107  mov    rbp,rsp
        48 83 ec 10
40110a  sub    rsp,0x10
        c7 45 fc 08 00 00 00
40110e  mov    DWORD PTR [rbp-0x4],0x8
        8b 45 fc
401115  mov    eax,DWORD PTR [rbp-0x4]
        89 c7
401118  mov    edi,eax
        b8 00 00 00 00
40111a  mov    eax,0x0
        e8 06 00 00 00
40111f  call   40112a <square>
        89 45 f8
401124  mov    DWORD PTR [rbp-0x8],eax
        90
401127  nop
        c9
401128  leave
        c3
401129  ret
square:
        55
40112a  push   rbp
        48 89 e5
40112b  mov    rbp,rsp
        89 7d fc
40112e  mov    DWORD PTR [rbp-0x4],edi
        8b 45 fc
401131  mov    eax,DWORD PTR [rbp-0x4]
        0f af c0
401134  imul   eax,eax
        5d
401137  pop    rbp
        c3
401138  ret
        0f 1f 80 00 00 00 00
401139  nop    DWORD PTR [rax+0x0]
```

## ARM gcc 11.2

```
abort@plt:
        e28fc600
102d8   add    ip, pc, #0, 12
        e28cca10
102dc   add    ip, ip, #16, 20 ; 0x10000
        e5bcfd34
102e0   ldr    pc, [ip, #3380]!    ; 0xd34
call_weak_fn:
        e59f3014
10314   ldr    r3, [pc, #20]    ; 10330 <call_weak_fn+0x1c>
        e59f2014
10318   ldr    r2, [pc, #20]    ; 10334 <call_weak_fn+0x20>
        e08f3003
1031c   add    r3, pc, r3
        e7932002
10320   ldr    r2, [r3, r2]
        e3520000
10324   cmp    r2, #0
        012fff1e
10328   bxeq   lr
        eaffffe6
1032c   b  102cc <__gmon_start__@plt>
        00010cdc
10330   .word  0x00010cdc
        00000018
10334   .word  0x00000018
main:
        b580
10394   push   {r7, lr}
        b082
10396   sub    sp, #8
        af00
10398   add    r7, sp, #0
        2308
1039a   movs   r3, #8
        607b
1039c   str    r3, [r7, #4]
        6878
1039e   ldr    r0, [r7, #4]
        f000 f805
103a0   bl 103ae <square>
        6038
103a4   str    r0, [r7, #0]
        bf00
103a6   nop
        3708
```

# Assembler

- It is hard to understand machine code
- Assembler code is source code that is very close to binary code
  - Valid instructions are represented textually using **mnemonics** (sub eax, eax)
- Assembler code can be understood by humans
- Binary code can be "disassembled" into assembler code!

```
        main:
         55
401106   push   rbp
         48 89 e5
401107   mov    rbp,rsp
         48 83 ec 10
40110a   sub    rsp,0x10
         c7 45 fc 08 00 00 00
40110e   mov    DWORD PTR [rbp-0x4],0x8
         8b 45 fc
401115   mov    eax,DWORD PTR [rbp-0x4]
         89 c7
401118   mov    edi,eax
         b8 00 00 00 00
40111a   mov    eax,0x0
         e8 06 00 00 00
40111f   call   40112a <square>
         89 45 f8
401124   mov    DWORD PTR [rbp-0x8],eax
         90
401127   nop
         c9
401128   leave
         c3
401129   ret
        square:
         55
40112a   push   rbp
         48 89 e5
40112b   mov    rbp,rsp
         89 7d fc
40112e   mov    DWORD PTR [rbp-0x4],edi
         8b 45 fc
401131   mov    eax,DWORD PTR [rbp-0x4]
         0f af c0
401134   imul   eax,eax
         5d
401137   pop    rbp
         c3
401138   ret
         0f 1f 80 00 00 00 00
401139   nop    DWORD PTR [rax+0x0]
```

# Machine Language and Assembler are Useful

- Since binary code can be disassembled, a common attacker model is that a program is avaible as assembler code to the attacker
- Attackers can experiment with such a software and test exploits
- Machine language helps to understand computers

# Hardware Basics

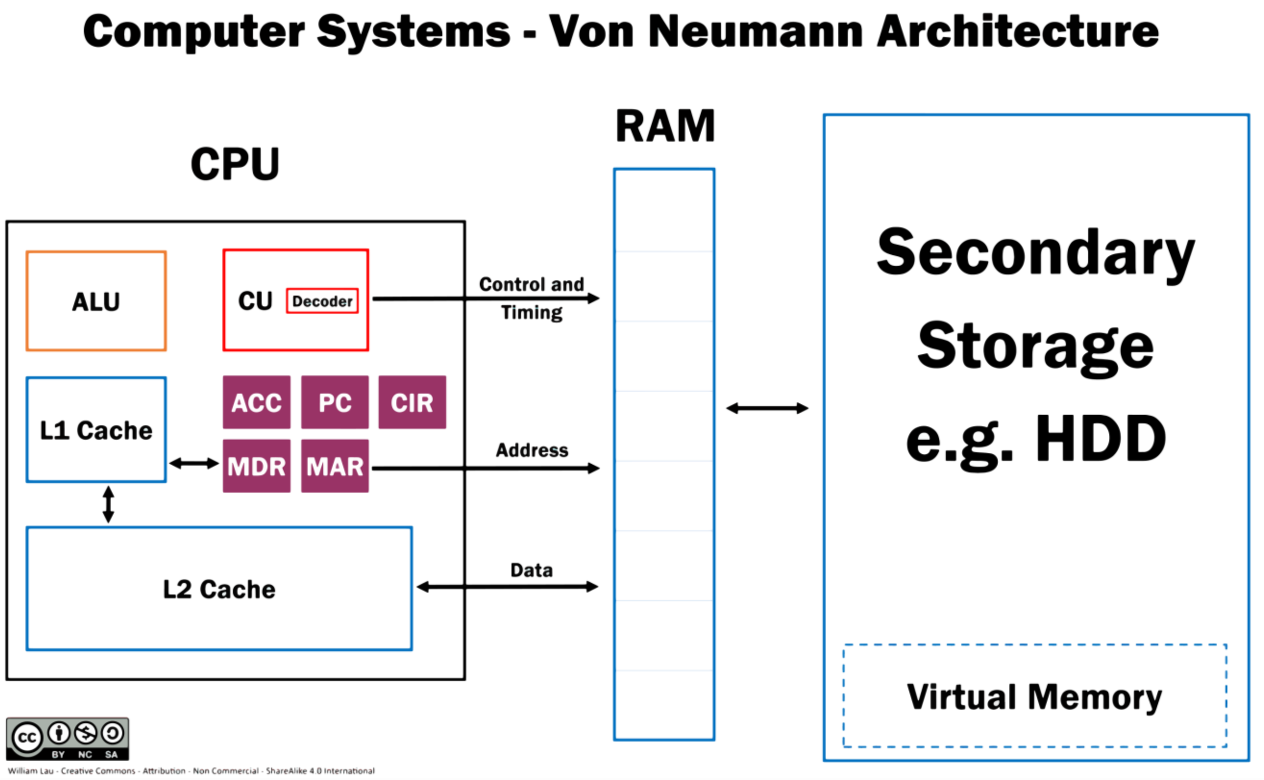
Sven Schäge

# CPU

- The central processing unit (CPU) or short, processor, is the engine of a computer
- Almost all other parts of the computer are controlled by the CPU
- The CPU consists of several parts:
  - Registers, which are basic storage cells to store data in that can be directly accessed by the CPU
  - Control Unit (CU) which manages proper program execution
  - Arithmetic Logical Unit (ALU) which performs computational tasks on operands stored in registers

# Von Neumann Architecture

- Data bus transfers data to read/write
- Address bus transfers (memory) addresses
- Control bus controls access to bus




Source: https://commons.wikimedia.org/wiki/File:Computer_Systems_-_Von_Neumann_Architecture_Large_poster_anchor_chart.svg

# Programs

- A program is a sequence of instructions
- Instructions and data is stored in memory (Von Neumann Architecture)
  - Can lead to many security problems
- Executing a program: Processor reads next instruction from memory and executes it with the corresponding data (Von Neumann Cycle)
  - 1. Fetch
  - 2. Decode
  - 3. Fetch Operands
  - 4. Execute
- Memory is Random Access Memory (RAM)
- Most instructions only operate on registers (CISC (x86) vs RISC (ARM) design)
  - Memory access only via dedicated load and store instructions

# Extending the Basic Computer Model: Pointers

- The main mechanism to extend the basic computer model is the notion of pointers
- The idea is to load **addresses** of memory units into registers. These registers now just point to/reference certain memory units and are thus called pointers.
- Via specific load and store addresses (“load or store effective address”) pointers can be “dereferenced”. This effectively loads and stores the contents of the referenced memory units.
- In this way, the CPU is able to access far more memory resources than the mere registers. However, this comes ta the cost of considerably decreased speed.
- To make the CPU access other hardware component of the computer as well, each hardware component is provided with a dedicated memory area. Via manipulation of the memory units in these areas, the CPU can control the hardware. In this sense the functionality of the hardware is mapped into memory (memory mapping).
  - Memory-mapped I/O uses the same address space to address both main memory and I/O devices
  - Port-mapped I/O uses a dedicated I/O address space and extra hardware to accomplish the communication between CPU and I/O.
- Often notation for dereferencing is via square brackets [ ]:
  [ebp] means the value stored in the memory unit(s) pointed to by the address in ebp

# Intel x86 Instruction Set

Sven Schäge

# Important Registers (and Conventions) X86-32

eax: Accumulator

ebx: Base address for addressing

ecx: Counter for loops, indices

edx: I/O Data

esi: Memory Address for String Source

edi: Memory Address for String Destination

esp: Stack Pointer, Frame Pointer

ebp: Base Pointer (to current stack frame)

eip: Instruction Pointer – address of next instruction to be executed

eflags: Status Flags - different flags that indicate certain events (e.g. result of computation was negative)

…

# Important Registers (and Conventions) X86-32

Depending on the size of bits of the register we are interested in, names could change slightly:

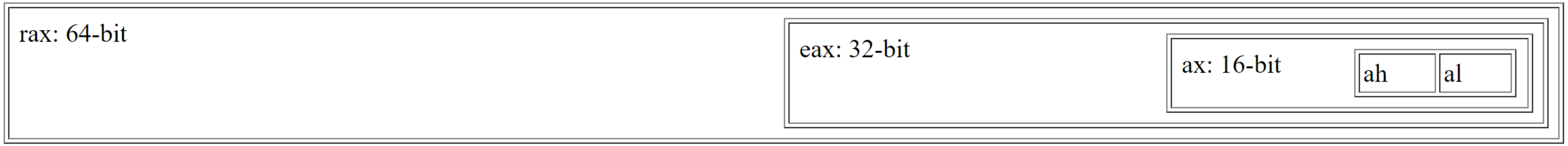
Consider 64 bit register rax:

rax reads and stores all 64 bits of rax

eax reads and stores all 32 bits of rax

ax reads and stores all 16 bits of rax

al reads and stores all 8 bits of rax



# X86 Instructions Set

Contains instructions for:

- Memory access (read/write operations)
- Arithmetic, logical and bitwise instructions
- Subroutine instructions (call, return, ...)
- Control transfer instructions (static jumps,conditional jumps)
- String instructions (string comparison, ...)
- …
- Instructions may have different number of operands (on x86 typically 2)
- Example: add eax,ebx
  - Means: compute eax = eax + ebx
- Important instructions:
  add, sub, and, or, xor, not, mov, je, jne, jz, load, store, call, ret, nop, push, pop

# Syntax

```c
void main()
{
    int j=8;
    int i=square(j);

}

int square(int num) {
    return num * num;
}
```

Compiler

### Intel Syntax

```
main:
         55
401106   push   rbp
         48 89 e5
401107   mov    rbp,rsp
         48 83 ec 10
40110a   sub    rsp,0x10
         c7 45 fc 08 00 00 00
40110e   mov    DWORD PTR [rbp-0x4],0x8
         8b 45 fc
401115   mov    eax,DWORD PTR [rbp-0x4]
         89 c7
401118   mov    edi,eax
         b8 00 00 00 00
40111a   mov    eax,0x0
         e8 06 00 00 00
40111f   call   40112a <square>
         89 45 f8
401124   mov    DWORD PTR [rbp-0x8],eax
         90
401127   nop
         c9
401128   leave
         c3
401129   ret
square:
         55
40112a   push   rbp
         48 89 e5
40112b   mov    rbp,rsp
         89 7d fc
40112e   mov    DWORD PTR [rbp-0x4],edi
         8b 45 fc
401131   mov    eax,DWORD PTR [rbp-0x4]
         0f af c0
401134   imul   eax,eax
         5d
401137   pop    rbp
         c3
401138   ret
         0f 1f 80 00 00 00 00
401139   nop    DWORD PTR [rax+0x0]
```

Target operand always on the left

### AT&T Syntax

```
main:
         55
401106   push   %rbp
         48 89 e5
401107   mov    %rsp,%rbp
         48 83 ec 10
40110a   sub    $0x10,%rsp
         c7 45 fc 08 00 00 00
40110e   movl   $0x8,-0x4(%rbp)
         8b 45 fc
401115   mov    -0x4(%rbp),%eax
         89 c7
401118   mov    %eax,%edi
         b8 00 00 00 00
40111a   mov    $0x0,%eax
         e8 06 00 00 00
40111f   call   40112a <square>
         89 45 f8
401124   mov    %eax,-0x8(%rbp)
         90
401127   nop
         c9
401128   leave
         c3
401129   ret
square:
         55
40112a   push   %rbp
         48 89 e5
40112b   mov    %rsp,%rbp
         89 7d fc
40112e   mov    %edi,-0x4(%rbp)
         8b 45 fc
401131   mov    -0x4(%rbp),%eax
         0f af c0
401134   imul   %eax,%eax
         5d
401137   pop    %rbp
         c3
401138   ret
         0f 1f 80 00 00 00 00
401139   nopl   0x0(%rax)
```

Source before destination

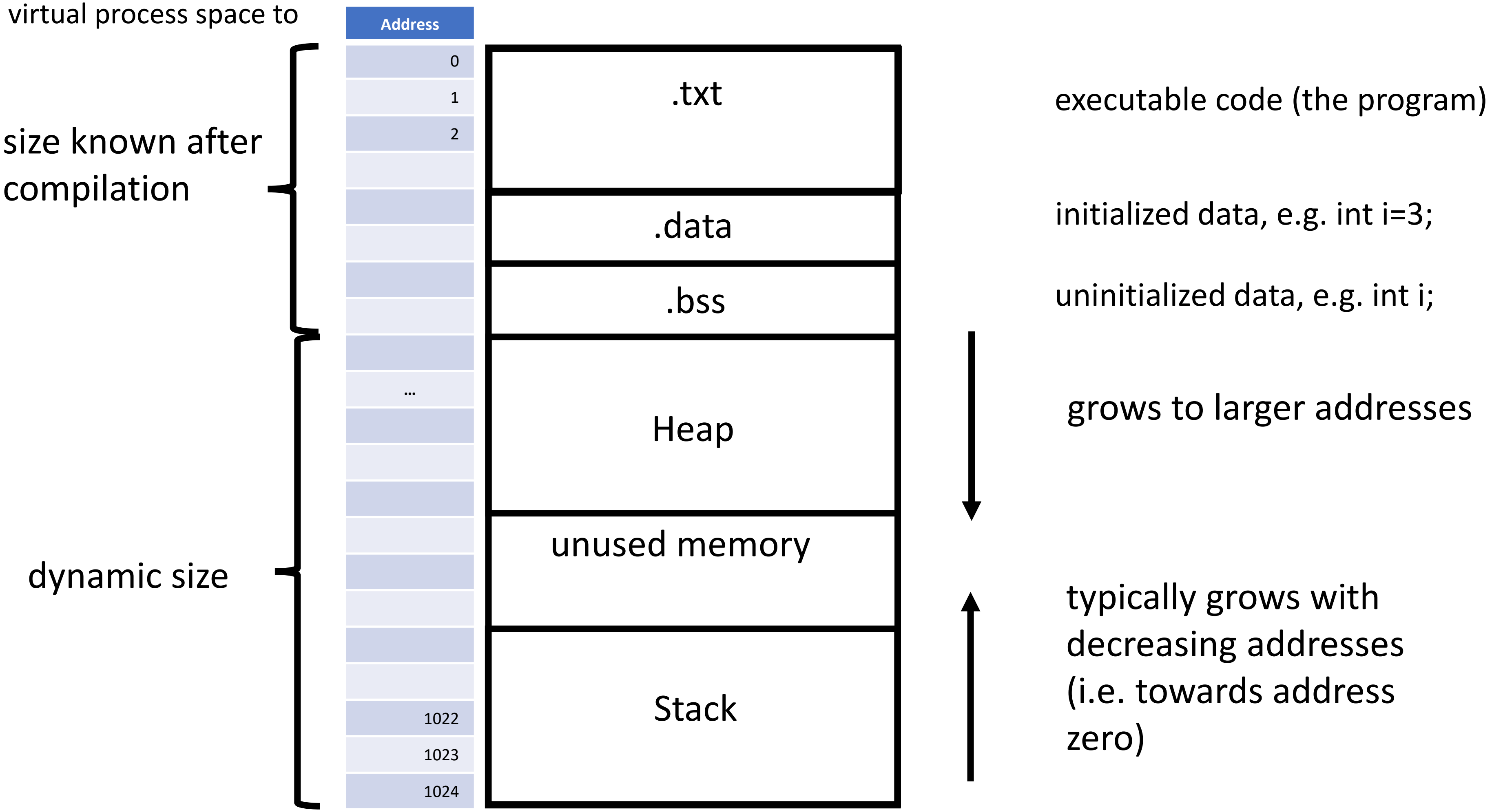
# Memory Layout

Sven Schäge

# Simplified Program Memory Layout

Typically, each program execution (process) is given a dedicated, virtual memory space by the operating system. The operating system then maps the manipulation of the virtual process space to real memory units.

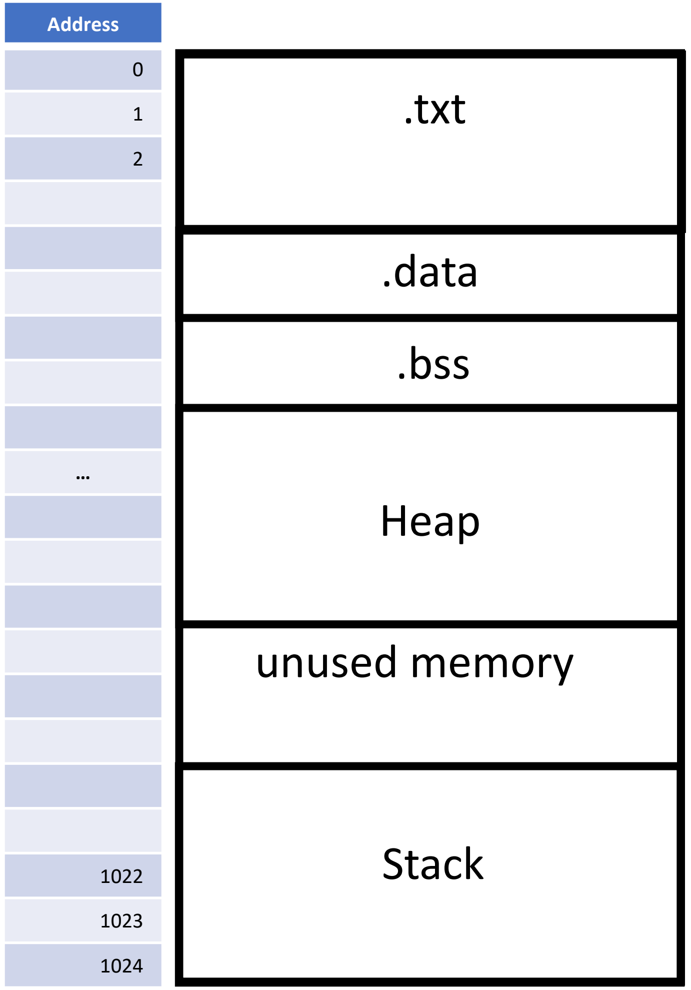
## Random Access Memory



# Programm Execution

## Random Access Memory

```c

int square(int z);

int main()
{
    int i;

    i=3;
    square(i);
    return 0;
}

int square(int z)
{
    int result;
    result=z*z;

    return result;
}
```




executable code (the program)

initialized data, e.g. int i=3;

uninitialized data, e.g. int i;

# Interpreting Binary Code (Simplified)

## Random Access Memory

```c

int square(int z);

int main()
{
    int i;

    i=3;
    square(i);
    return 0;
}

int square(int z)
{
    int result;
    result=z*z;

    return result;
}
```

| Address |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| |
| |
| |
| |
| |
| … |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| 1023 |
| 1024 |

```asm
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 3
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square(int)
        mov     eax, 0
        leave
        ret
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

.txt, .data, .bss

Heap

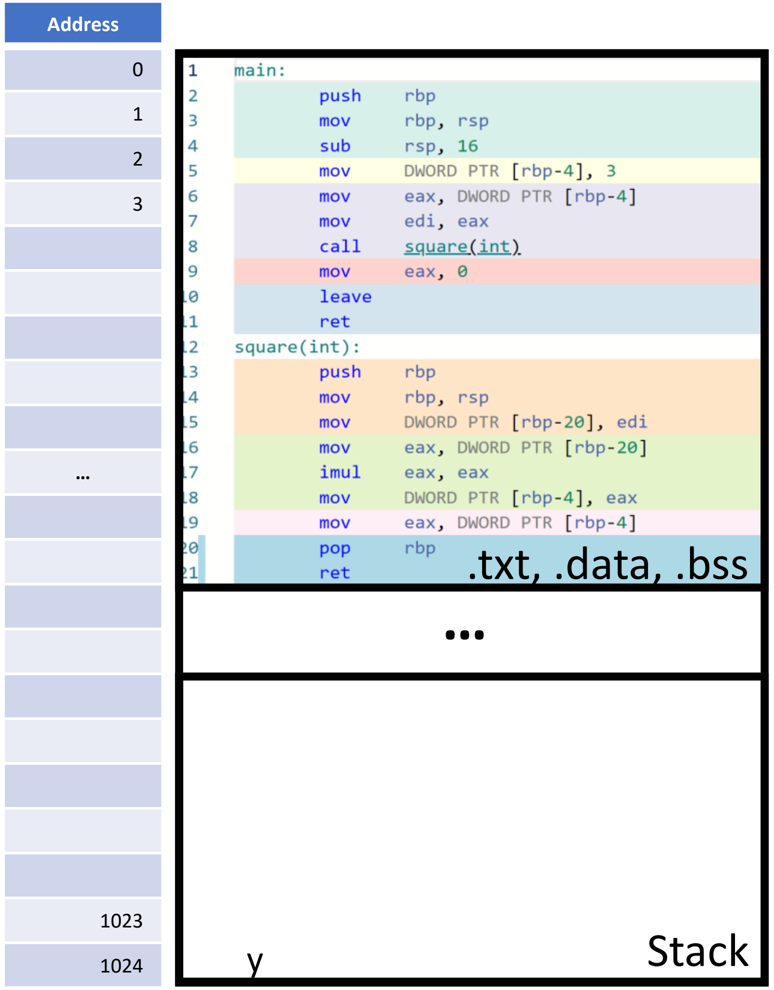
…

Stack

# Program Execution: Basic Mechanism

Random Access Memory

Program Counter PC

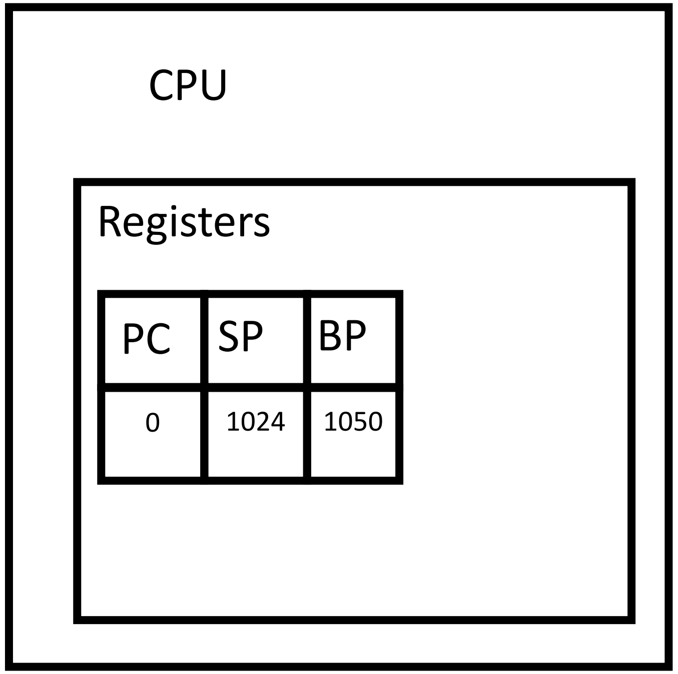



The PC points to the next command to be executed.
The CU orchestrates the execution of this command via the Von Neumann cycle and then increments PC.

The basic mechanism would only allow a linear control flow! Commands that should be executed consecutively need to be aligned as neighbors. No nesting (function calls) can be realized in this way.

Stack Pointer SP (rsp,esp)

# The Stack




- The stack implements a Last-In-First-Out (LIFO) Memory
  - Can be used to manage nesting and recursions
- Processor instructions that influence the stack
  - PUSH X
    - Decreases the stack pointer (register sp) by 1
    - Stores X at the position of the stack pointer
  - POP
    - Returns what is currently stored at the stack pointer
    - Increases the stack pointer by 1
  - Call (simplified)
    - PUSHes the current program counter (register pc) incremented by 1 on the stack and loads the address of the called function (callee) into the PC
    - This stores a **return address** to later continue the computations in the calling function!
  - RET (simplified)
    - POPs the last a value from the stack, interprets it as a return address, and loads it into the PC.
    - In this way, programs can return to the original function after they have worked on a sub-function.
    - The computation continues at in the calling function (caller).

# Using the Stack

## Random Access Memory

Program Counter PC ⟹

| Address |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| ... |
| 1023 |
| 1024 |

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 3
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square(int)
        mov     eax, 0
        leave
        ret
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

.txt, .data, .bss

...

y

Stack

Stack Pointer SP (rsp,esp) ⟹

CPU

Registers

| PC | SP | BP |
| --- | --- | --- |
| 0 | 1024 | 1050 |

Essentially, these three special pointers control the program flow and allow function calls!

- PC points to current command
- SP points to top of stack
- BP points to reference stack address in current level of nesting of function calls

# Implementing Nested Function Calls

- For each level of nesting, whenever function X (caller) calls function Y (callee) we have to store information that let us return to the next command in the code of the caller - right after the function call.
- To this end, we have to store (on the stack) the address of the next command to be executed (PC+1) by the caller. This is the return address (RA).
- After the callee finishes, we need to load the RA back to the PC.
- The RA should be stored on the stack since nested function calls require a LIFO structure:
  the function called on the deepest level of nesting will finish first and give control back to the next higher level.

| Value | Address | Comment |
|---|---|---|
| | [ebp – X] | current stack pointer |
| | … | |
| | [ebp – 8] | 2nd local variable |
| | [ebp – 4] | 1st local variable |
| oldbp | [ebp] | old base pointer |
| RA of caller | [ebp + 4] | |
| 10 | [ebp + 8] | 1st function argument |
| 5 | [ebp + 12] | 2nd function argument |
| 2 | [ebp + 16] | 3rd function argument |
| | … | |

Typical Stack Layout:
When executing a function, local variables and input variables on the stack are referenced relative to the base pointer associated to that function. The location of the memory unit containing the return address is also fixed relative to that base pointer.

# Implementing Nested Function Calls

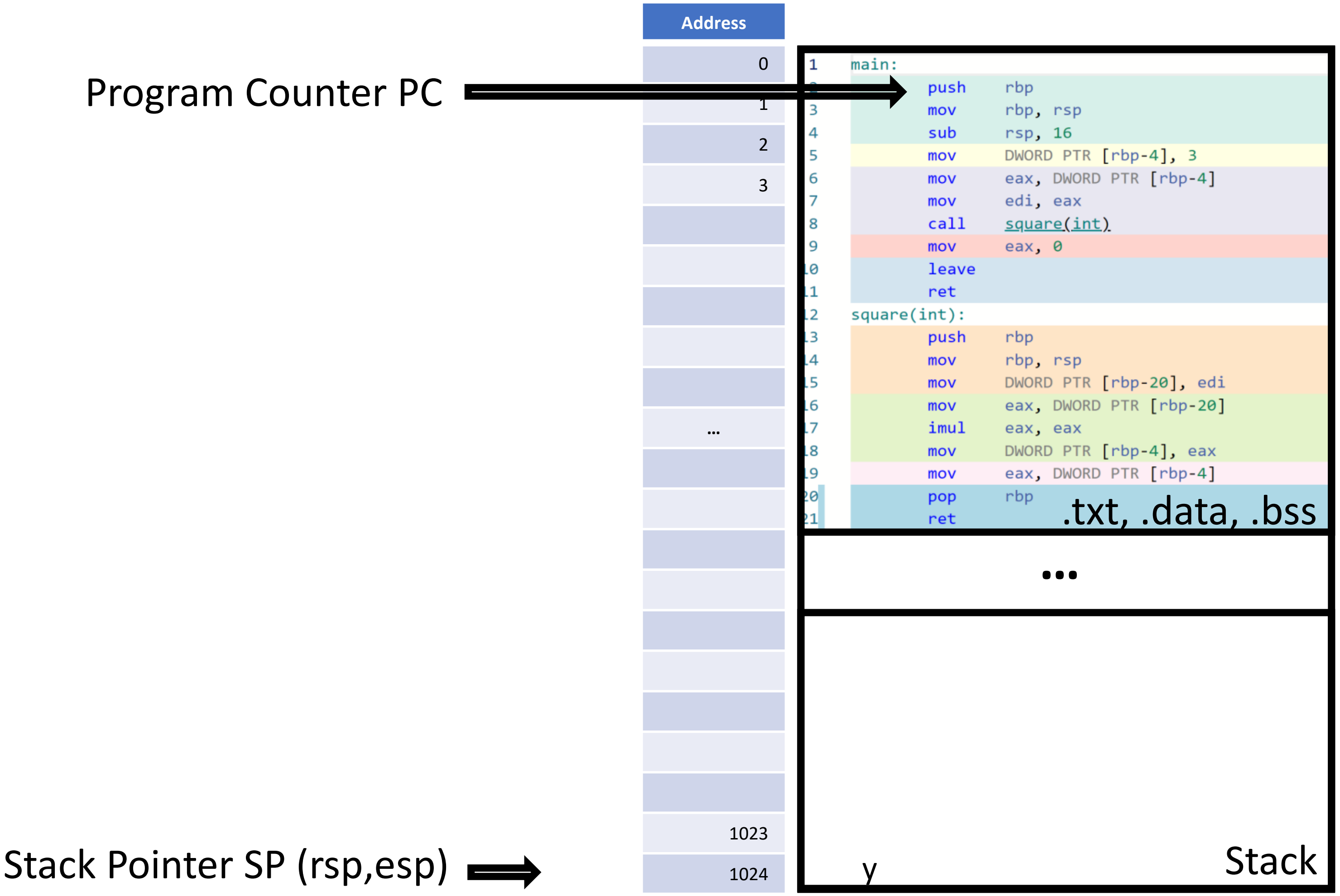
- For each function, there must be a dedicated area in the stack that stores local values belonging to that function like local variables. This is often referred to as the **stack frame** of that function or the activation record.
- The base pointer (BP) will hold a central reference address in the stack. This address references the current stack frame (the current callee) of the function.
  - The first address larger than the base pointer will store the return address of the calling function.
  - Addresses lower than the base pointer (growing direction of stack) will contain local variables of the current function. The calling function cannot access these variables.
  - Addresses higher than the RA will contain the local variables of the calling function. They can be referenced by the callee as input parameters. Alternatively, input parameters can be transferred to the callee via registers (e.g. edi).
  - The value stored at the base pointer is the address of the old base pointer. If the callee finishes, the old base pointer is restored. This updates the current stack frame back to the calling function.
  - Moving from the current stack frame to previous ones (up to the highest function in the hierarchy) via loading increasingly older base pointers is often called **stack walking**.

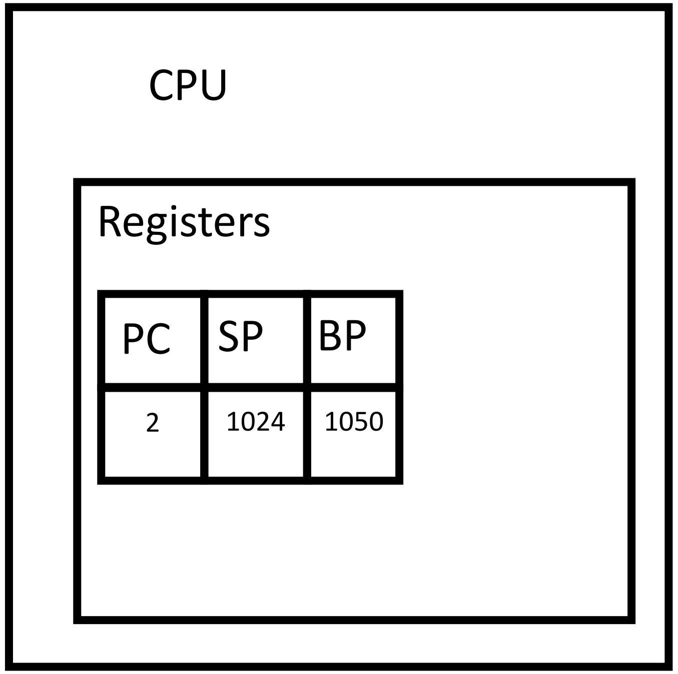
| Value | Address | Comment |
|---|---|---|
| | [ebp – X] | current stack pointer |
| | … | |
| | [ebp – 8] | 2nd local variable |
| | [ebp – 4] | 1st local variable |
| oldbp | [ebp] | old base pointer |
| RA of caller | [ebp + 4] | |
| 10 | [ebp + 8] | 1st function argument |
| 5 | [ebp + 12] | 2nd function argument |
| 2 | [ebp + 16] | 3rd function argument |
| | … | |

Typical Stack Layout:
When executing a function, local variables and input variables on the stack are referenced relative to the base pointer associated to that function. The location of the memory unit containing the return address is also fixed relative to that base pointer.

# Using the Stack

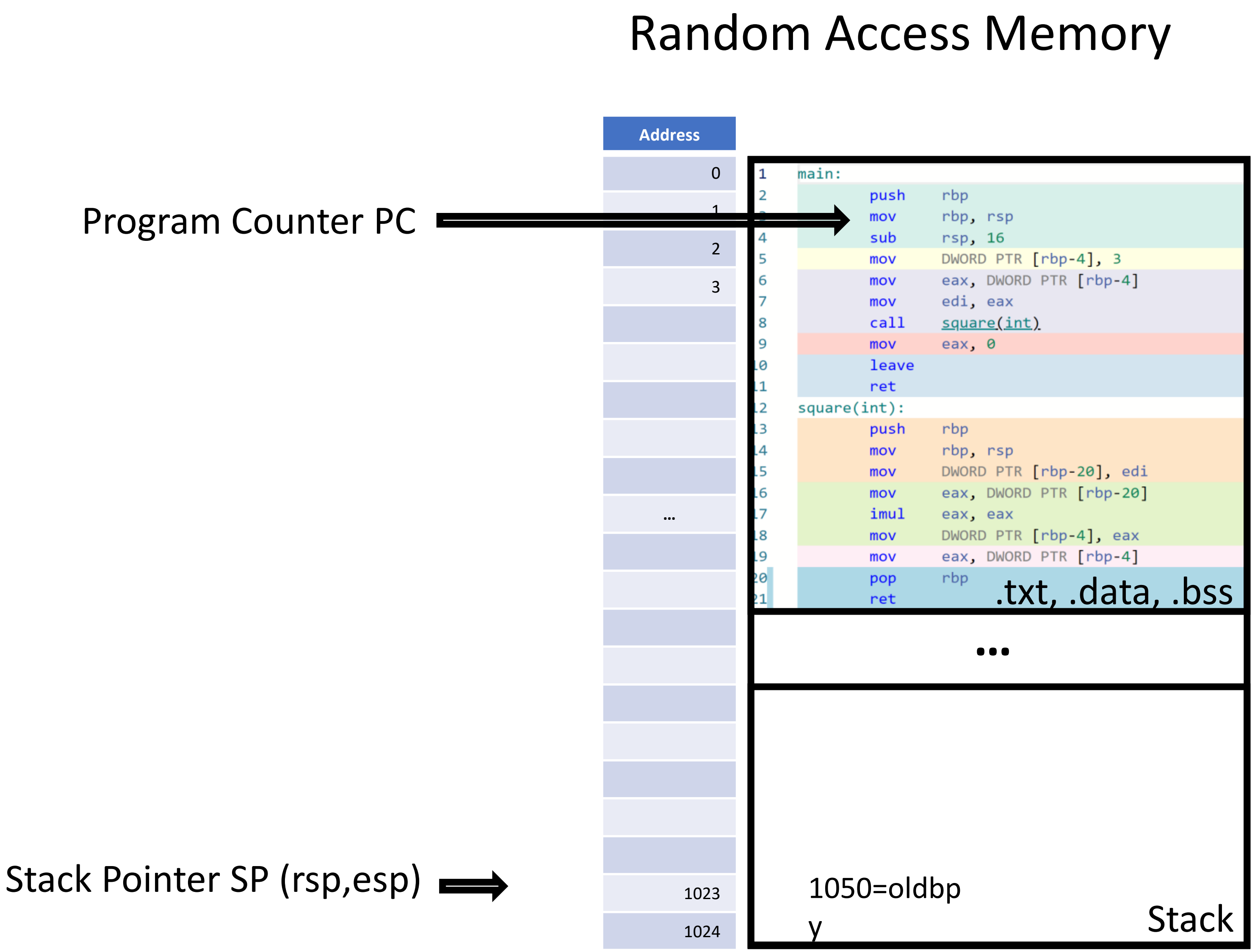
Random Access Memory **before** execution of command at PC

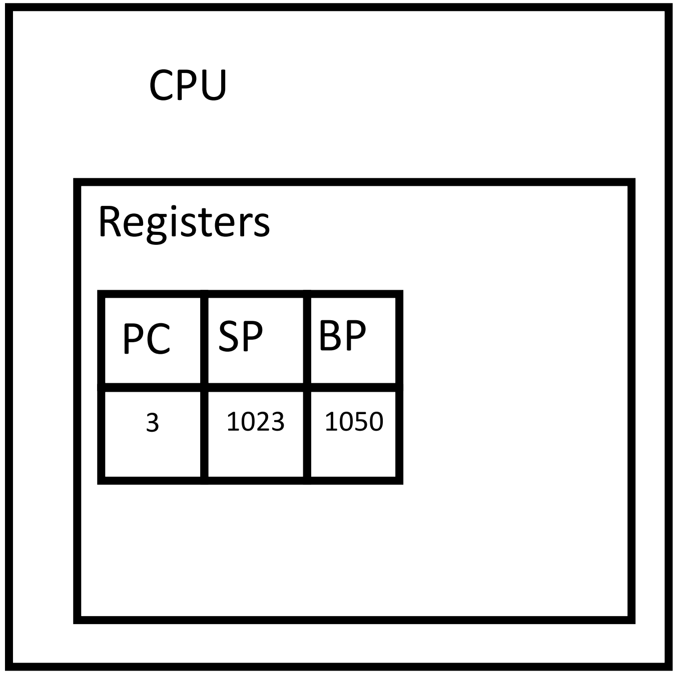



Old bp will be saved on stack. This saves an important reference address that is required to go back to the calling function or the OS.

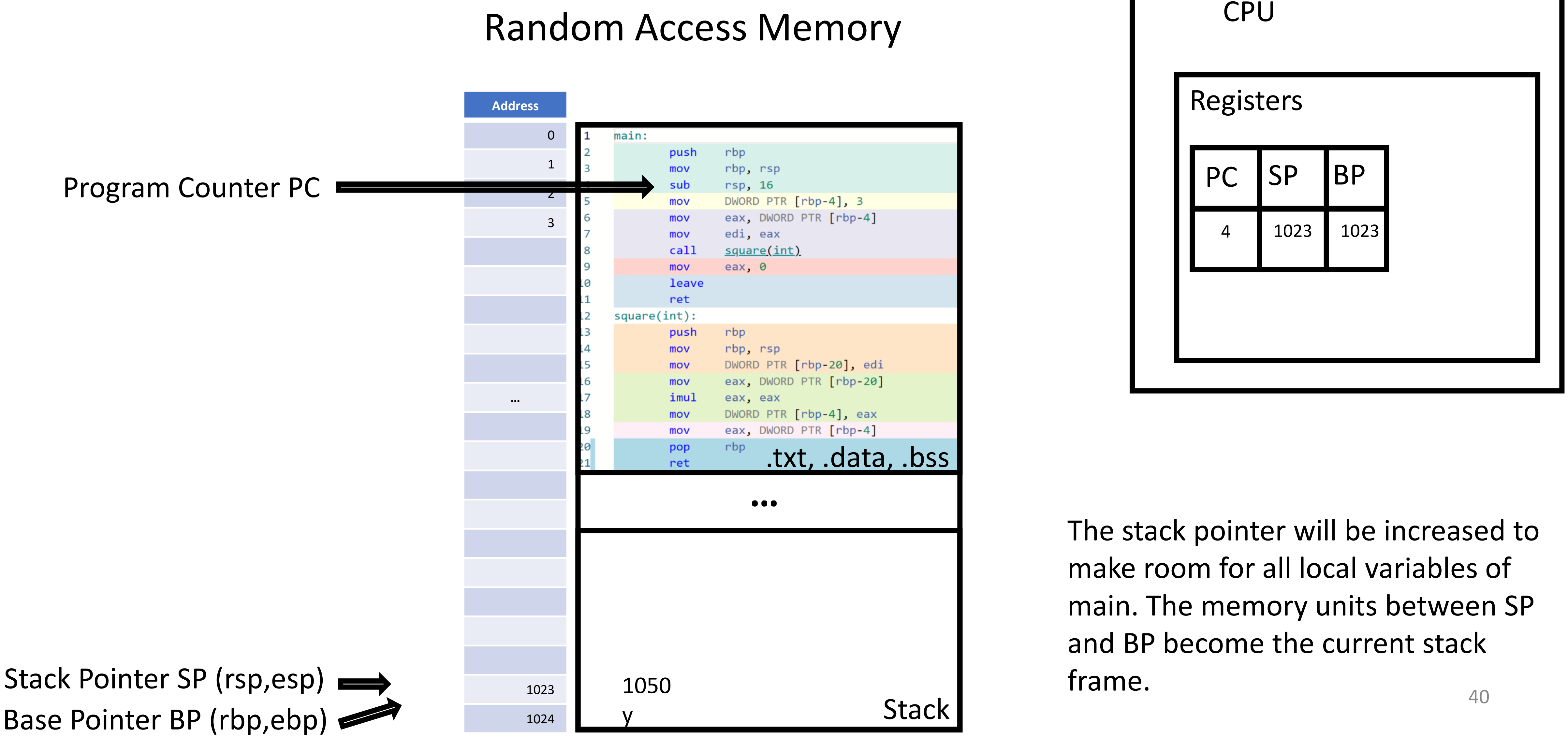
# Using the Stack




Will set the new bp to be the current sp. This prepares a new base reference for the function that has just been entered.

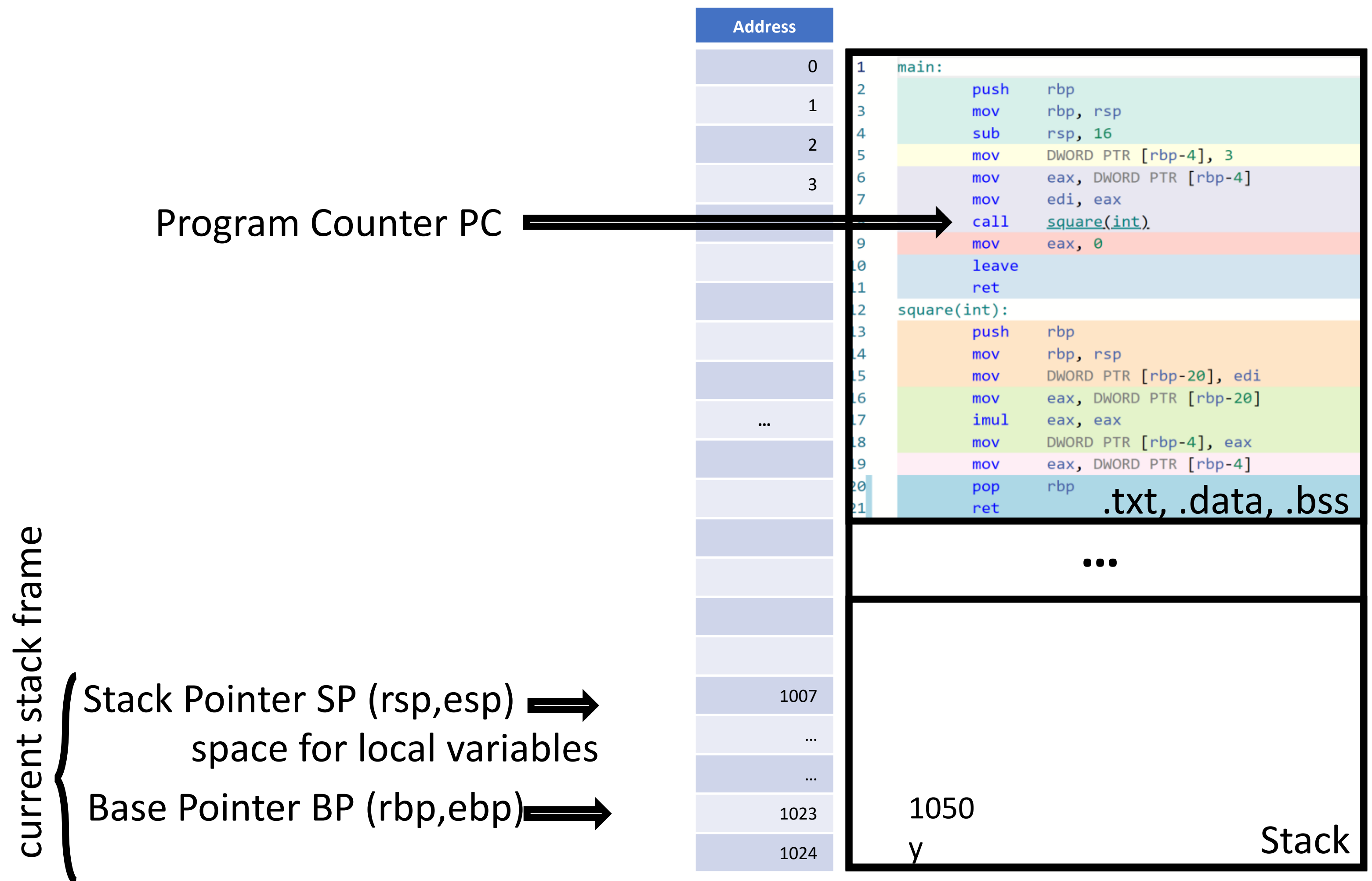
# Using the Stack

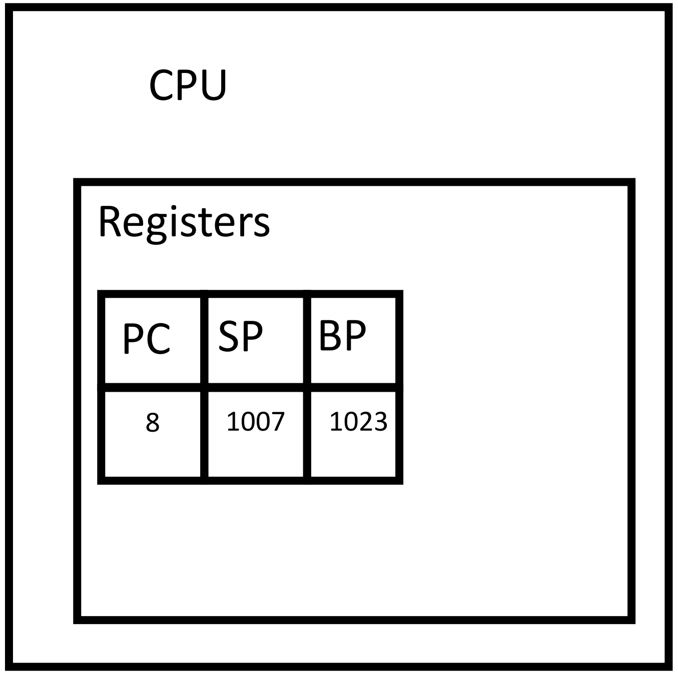



The stack pointer will be increased to make room for all local variables of main. The memory units between SP and BP become the current stack frame.

# Using the Stack – Nesting

## Random Access Memory

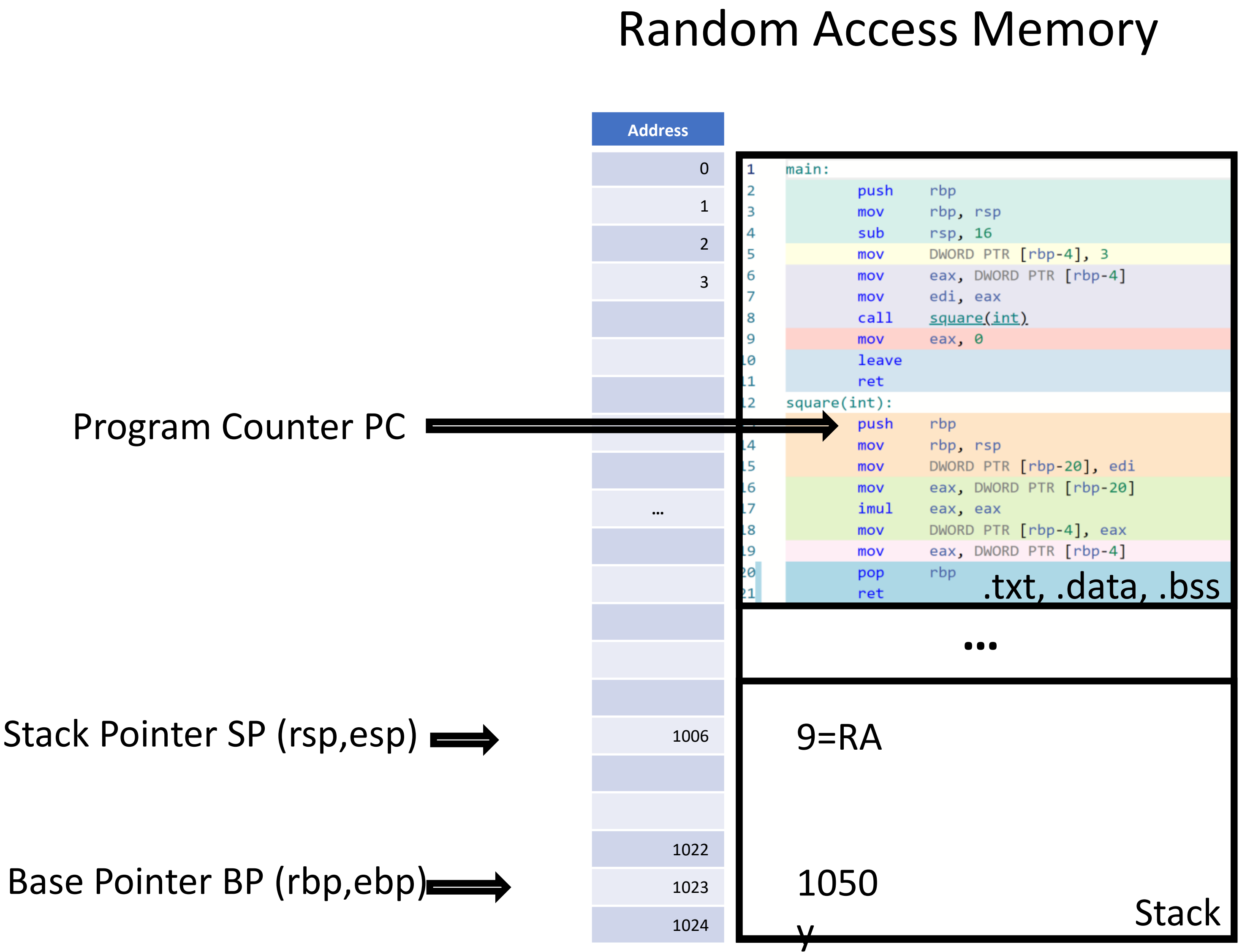


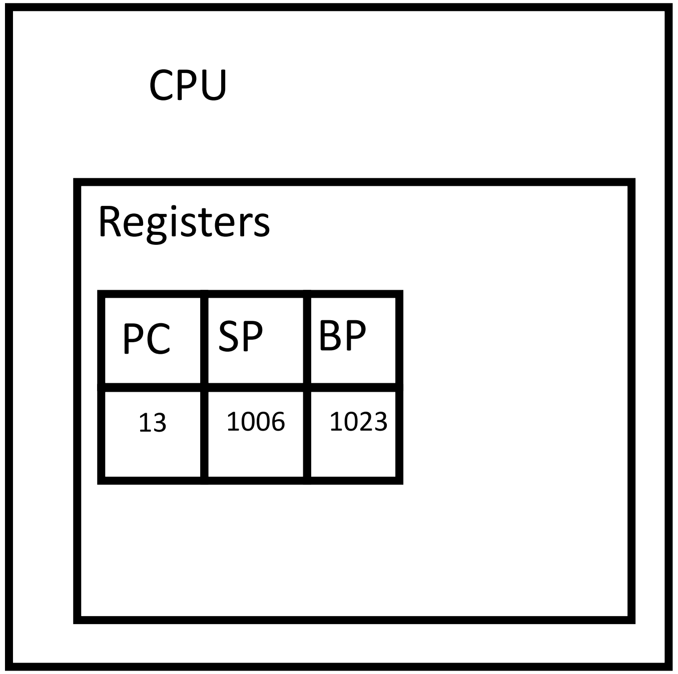




Call = PUSH PC to Stack:

- will decrease SP
- will save RA=PC+1 on stack at position SP
- will set PC = Address of function square

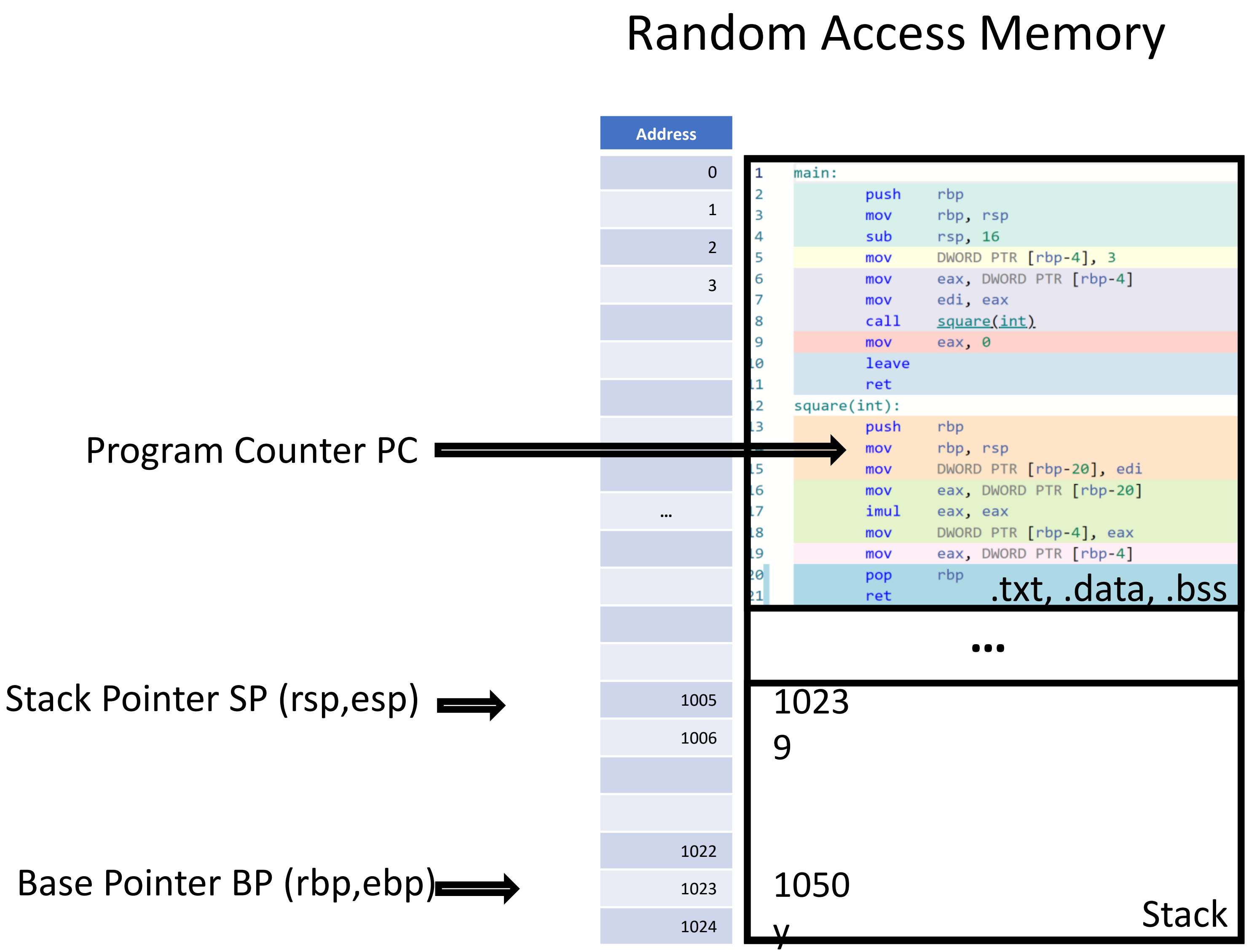
# Using the Stack

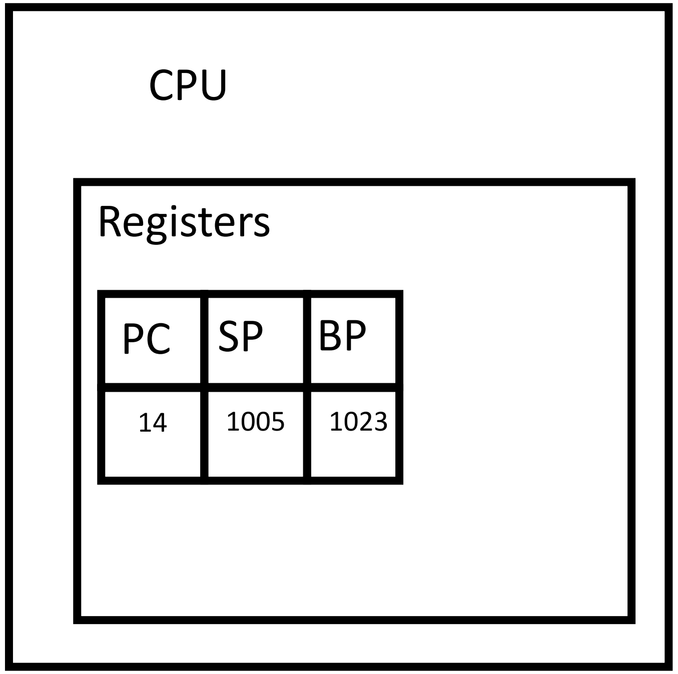



Old bp will be saved on stack. This saves an important reference address that is required to go back to the calling function main.

# Using the Stack




Will set the new bp to be the current sp. Will prepare new base reference for the function that has just been entered. To this end, use new space at the top of the stack.

# Using the Stack

## Random Access Memory

| Address |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| |
| |
| |
| |
| |
| … |
| |
| |
| |
| |
| 1005 |
| 1006 |
| |
| |
| 1022 |
| 1023 |
| 1024 |

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 3
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square(int)
        mov     eax, 0
        leave
        ret
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

.txt, .data, .bss

…

1023
9

1050
y

Stack

Program Counter PC

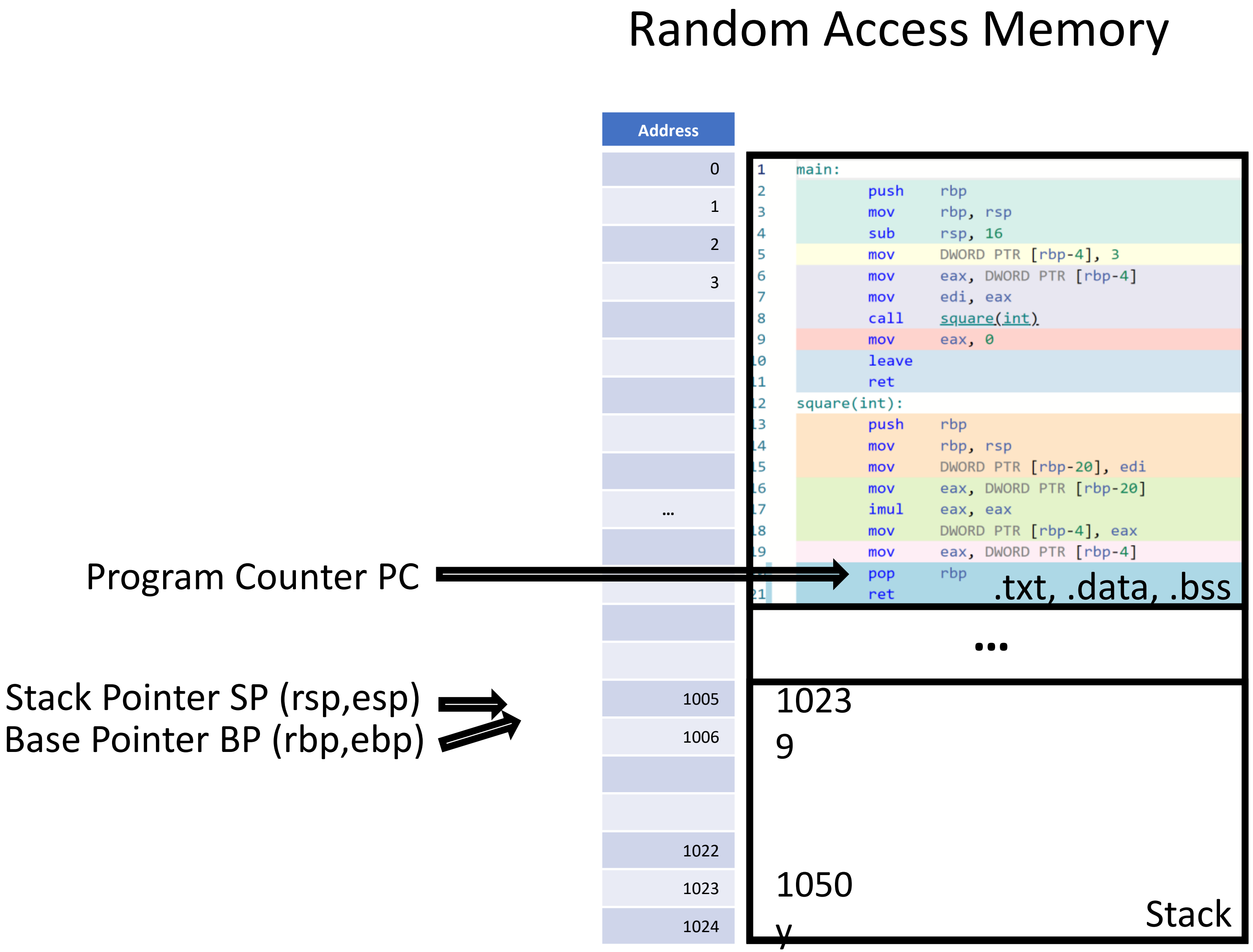
Stack Pointer SP (rsp,esp)
Base Pointer BP (rbp,ebp)

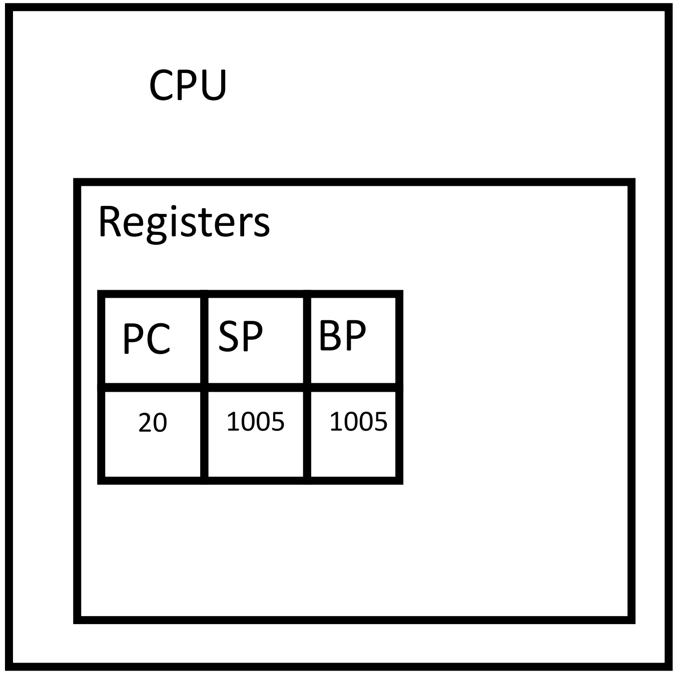
CPU

Registers

| PC | SP | BP |
| --- | --- | --- |
| 15 | 1005 | 1005 |

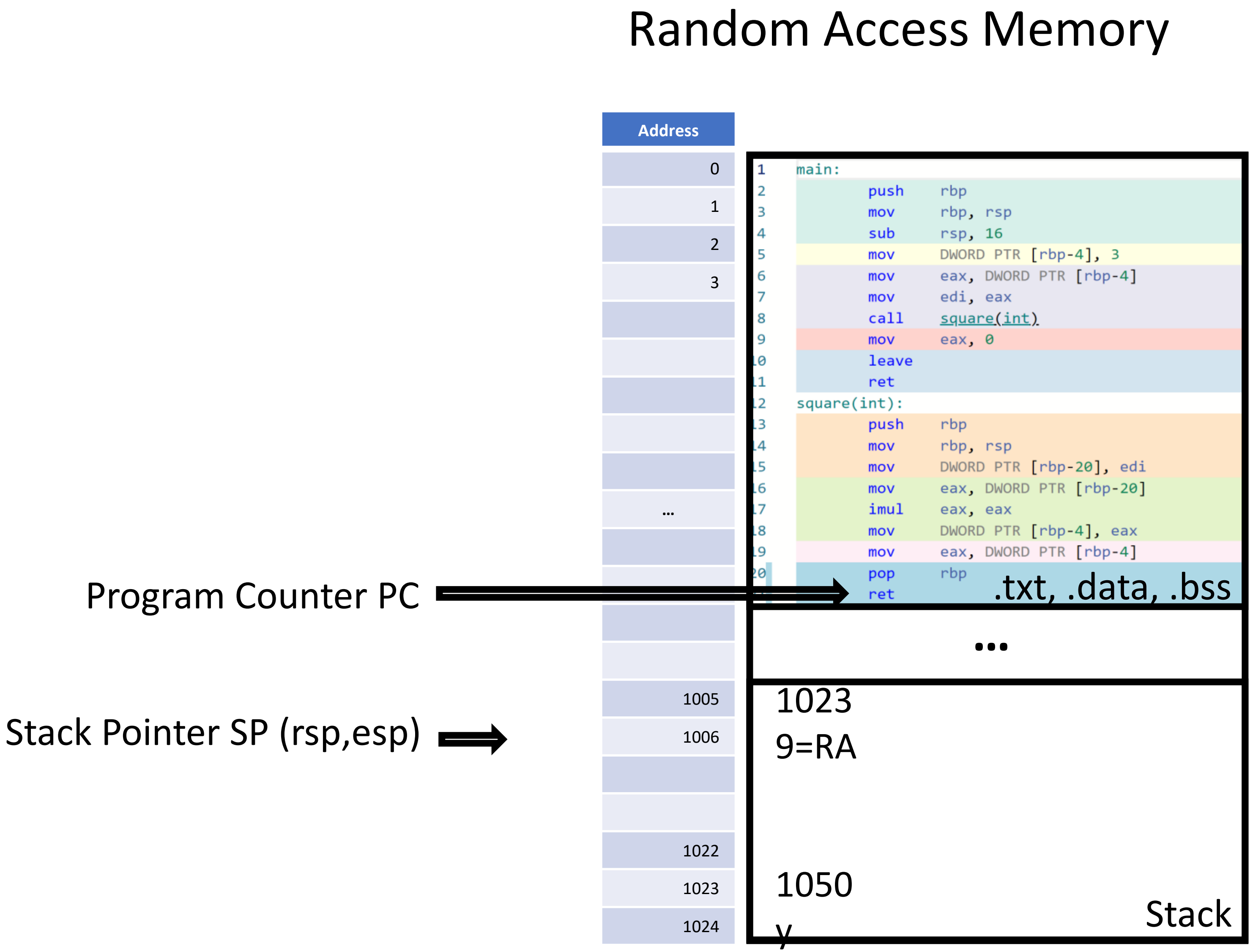
# Using the Stack

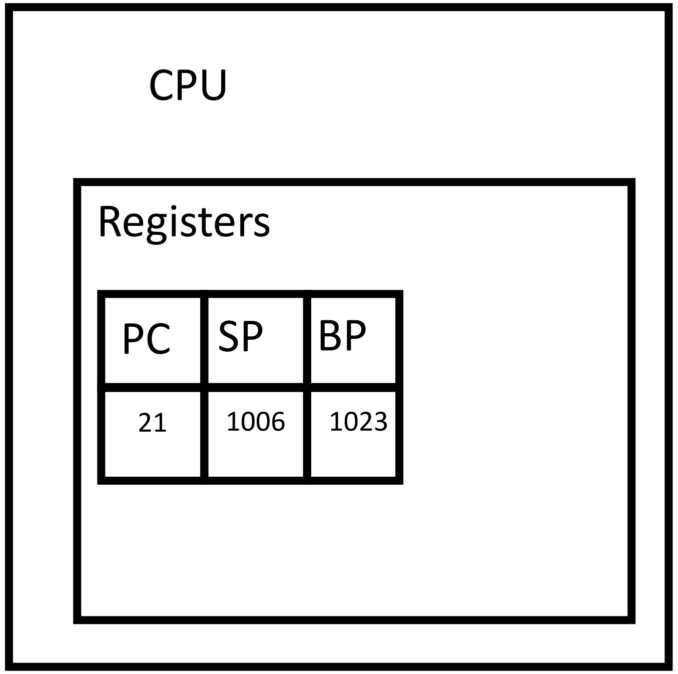



Will store the highest value from the stack to the base pointer. This restores the base pointer of main.

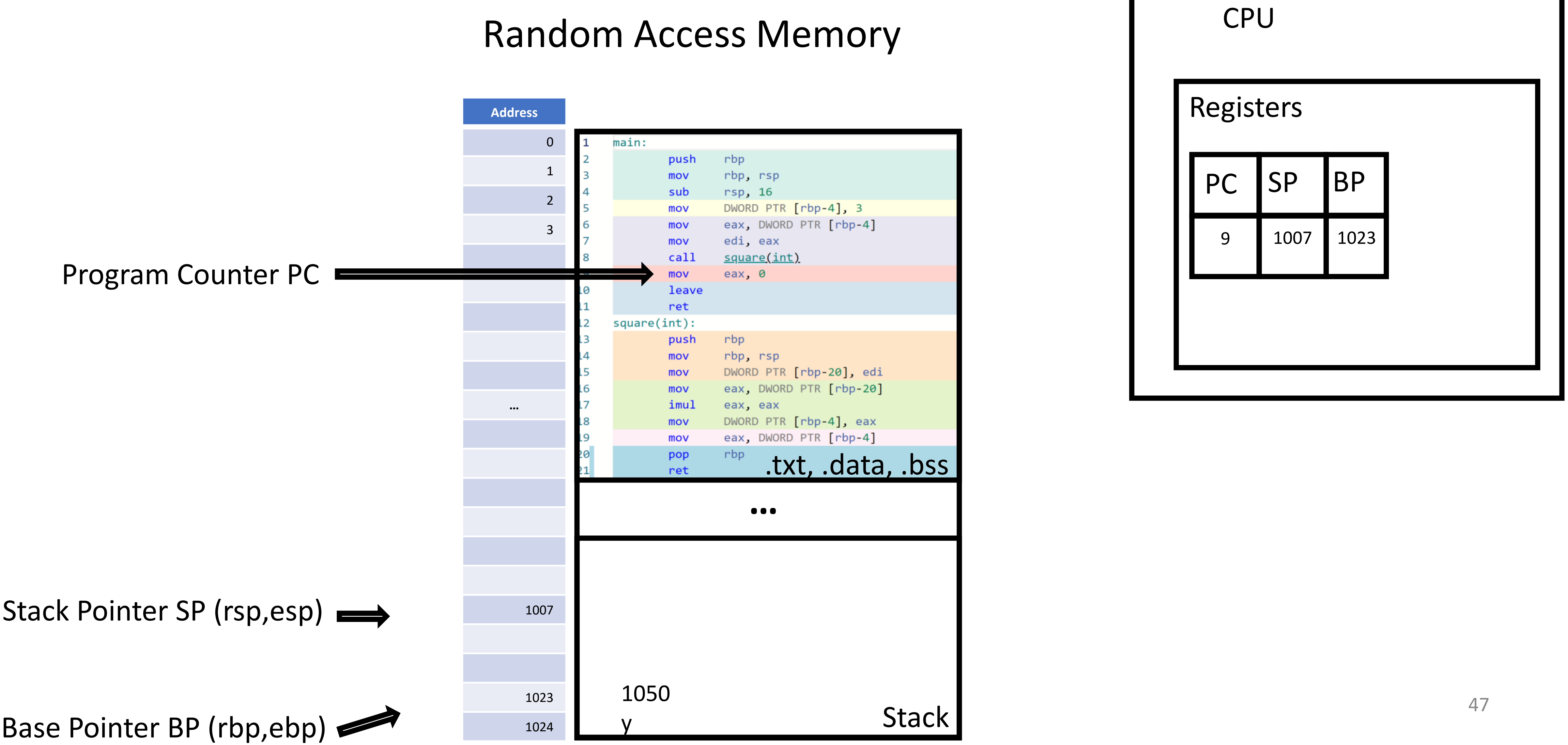
# Using the Stack



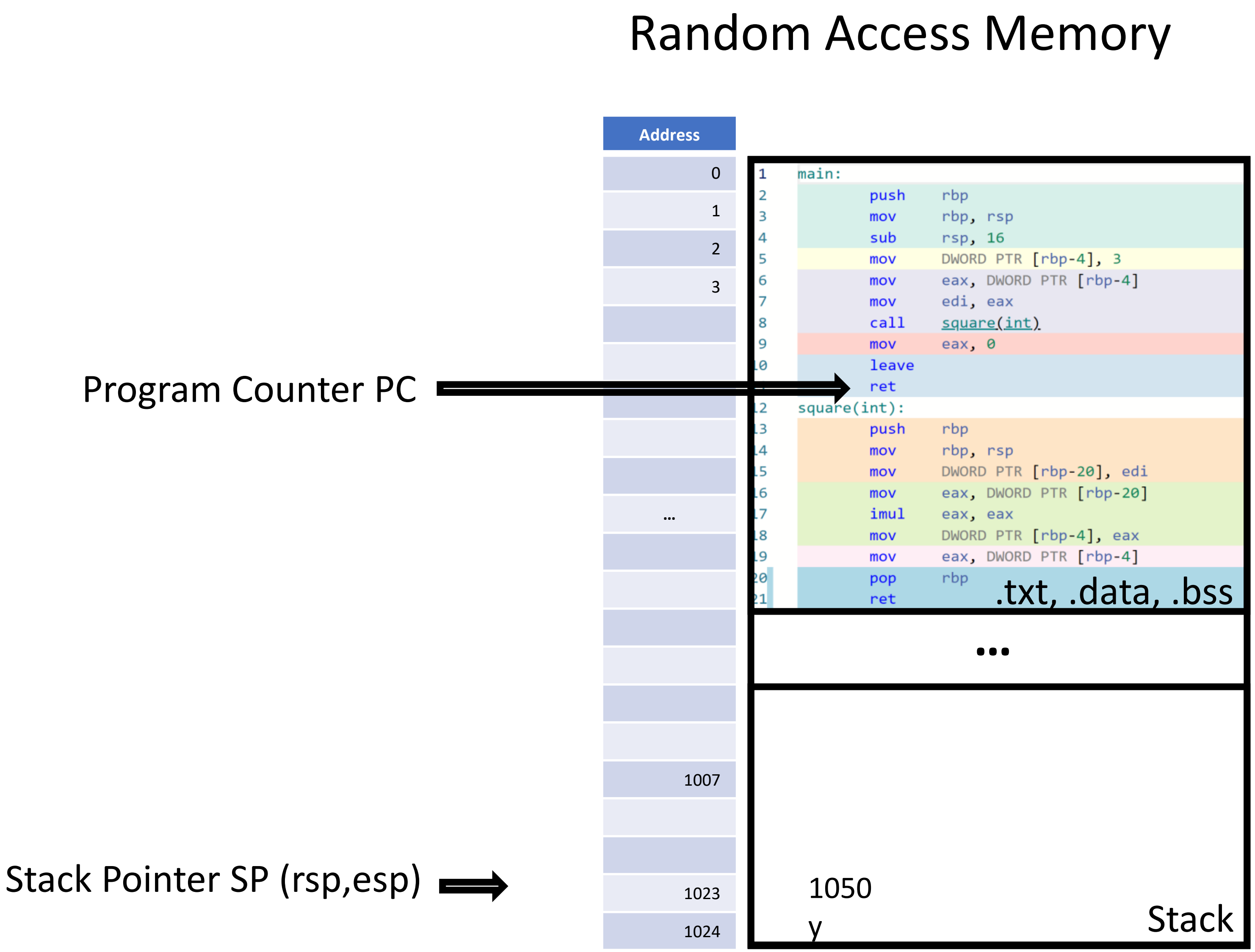



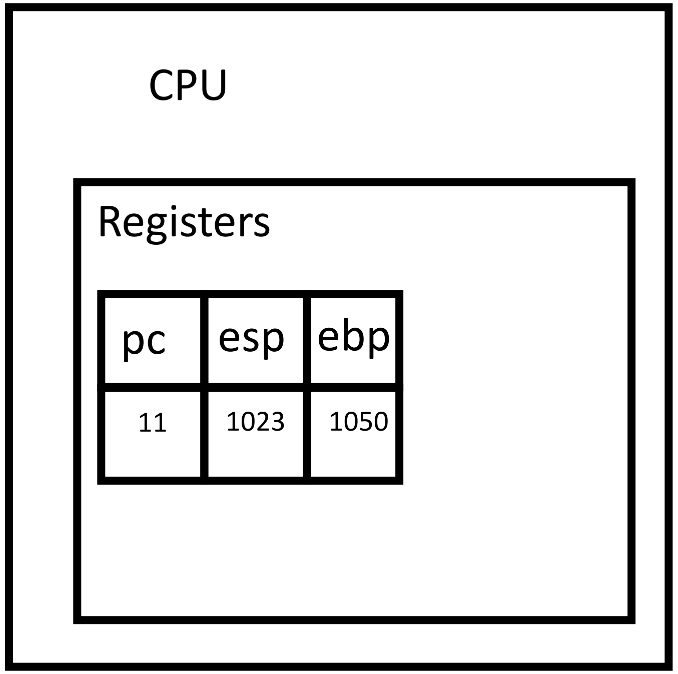

POPs [SP]=RA into PC
Returns by setting the current PC to the RA that was stored on the stack

# Using the Stack



# Using the Stack




```
leave:
  mov rsp, rbp
  pop rbp
```

# Function Calls and Stack

C

```c
void MyFunction()
{
  int a, b, c;
  ...
```

```c
a = 10;
b = 5;
c = 2;
```

Assembler

```asm
_MyFunction:
  push ebp      ; save the value of ebp
  mov ebp, esp ; ebp now points to the top of the stack
  sub esp, 12   ; space allocated on the stack for the local variables
```

```asm
mov [ebp -  4], 10  ; location of variable a
mov [ebp -  8], 5   ; location of b
mov [ebp - 12], 2   ; location of c
```

# Function Calls and Stack

C

```c
void MyFunction2(int x, int y, int z)
{
  ...
}
```

Assembler

```
_MyFunction2:
  push ebp
  mov ebp, esp
  sub esp, 0      ; no local variables, most compilers will omit this line
```

```c
MyFunction2(10, 5, 2);
```

```
push 2
push 5
push 10
call _MyFunction2
```

Typical Stack Layout

| Value | Address | Comment |
|---|---|---|
| | [ebp – X] | current stack pointer |
| | … | |
| | [ebp – 8] | 2nd local variable |
| | [ebp – 4] | 1st local variable |
| BP | [ebp] | old ebp |
| RA | [ebp + 4] | |
| 10 | [ebp + 8] | 1st function argument |
| 5 | [ebp + 12] | 2nd function argument |
| 2 | [ebp + 16] | 3rd function argument |
| | … | |

# Function Calls and Stack

C

```c
void MyFunction3(int x, int y, int z)
{
  int a, b, c;
  ...
  return;
}
```

Assembler

```asm
_MyFunction3:
  push ebp
  mov ebp, esp
  sub esp, 12 ; sizeof(a) + sizeof(b) + sizeof(c)
  ;x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
  ;a = [ebp - 4] = [esp + 8], b = [ebp - 8] = [esp + 4], c = [ebp - 12] = [esp]
  mov esp, ebp
  pop ebp
  ret
```

function prologue

function epilogue

function perilogue= function prologue + function epilogue

Typical Stack Layout

| Value | Address | Comment |
|---|---|---|
| | [ebp – X] | current stack pointer |
| | … | |
| | [ebp – 8] | 2nd local variable |
| | [ebp – 4] | 1st local variable |
| BP | [ebp] | old ebp |
| RA | [ebp + 4] | |
| 10 | [ebp + 8] | 1st function argument |
| 5 | [ebp + 12] | 2nd function argument |
| 2 | [ebp + 16] | 3rd function argument |
| | … | |

# Calling Conventions

- Agreement on callees receive parameters from their caller and how they return a result.
  - To transfer parameters to the callee functions typically store parameters on the stack. This decreases the SP effectively making extraspace on the stack for these parameters. After the function call, the SP has to be increased again. This process is often called cleanup.
- __cdecl: caller does cleanup: ret
- __stdcall: callee does cleanup: ret 8
- __fastcall: the first two parameters are passed using registers and the remaining parameters (if any), would be pushed to stack

| X86 Calling Convention | How parameters are passed | Who does the stack clean-up? |
|---|---|---|
| __stdcall | Pushed to the stack in right to left order | Callee |
| __cdecl | Pushed to the stack in right to left order | Caller |
| __fastcall | First two parameters are passed in ECX, EDX. Remaining are pushed to the stack in right to left order | Callee |

# Heap vs. Stack

Sven Schäge

# Heap Memory in C/C++

- It is often not clear how much memory a program will require. This can be highly dependent on the input data.
    - Example: Think of a list structure that per input character, generates a new list object.
- Reserving (allocating) the worst-case amount of memory at start-up is usually too wasteful.
- To obtain the required memory dynamically for new objects, a program can explicitly ask for a new set of consecutive memory units to be reserved on the heap memory (in C/C++ via the **malloc** or **new** command).
- In C/C++, this memory area is **referenced** by a **pointer**, (i.e. a mere address value that points to the first byte of the reserved memory=holds the address of the first byte of the reserved memory.)
- Moreover, C/C++ has means to instruct the processor to **dereference** the pointer, i.e. to evaluate the address and gain access to the memory units it points to.
- After usage, we have to tell the compiler to **free** the reserved memory again, so that it can be used for other purposes.
- It is common practice that a function which is supposed to return a pointer, does return a default pointer called **NULL pointer**, in case it encountered an error.