# Machine-Level Programming: Addressing modes and Arrays

# ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

# Disclaimer

## Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

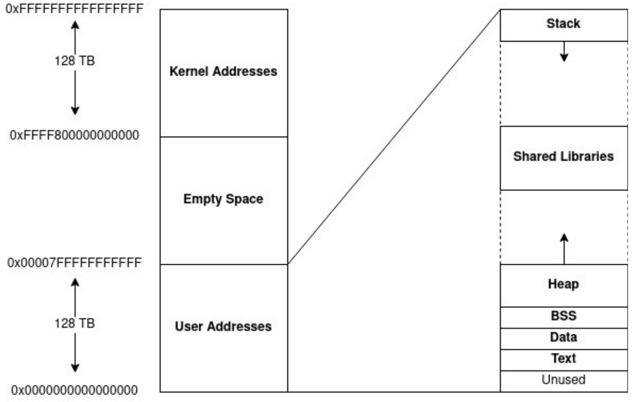
# Outline

# Memory

# Memory

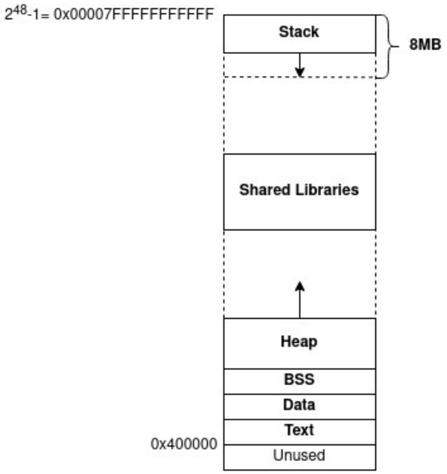
- Physical memory is a **list of bytes**, **each byte of which has an address**.
- Virtual memory acts as an **abstraction between the address space and the physical memory** available in the system.
  - This means that when a **program uses an address** that **address does not refer to the bytes in an actual physical location in memory**.
- So to this end, we say that **all addresses a program uses are virtual**.
- The operating system keeps track of **virtual addresses and how they are allocated to physical addresses**.
  - A **Page Table** is a data structure used by the operating system to keep track of the mapping between virtual addresses used by a process and the corresponding physical addresses in the system's memory.
- When a program does a load or store from an address, the operating system converts this virtual address to the actual address in the physical memory.

- The x86-64 architecture is 64-bit: registers (and addresses) are 64 bits wide.
- However, **virtual addresses on current x86-64 processors only have 48 meaningful bits**.

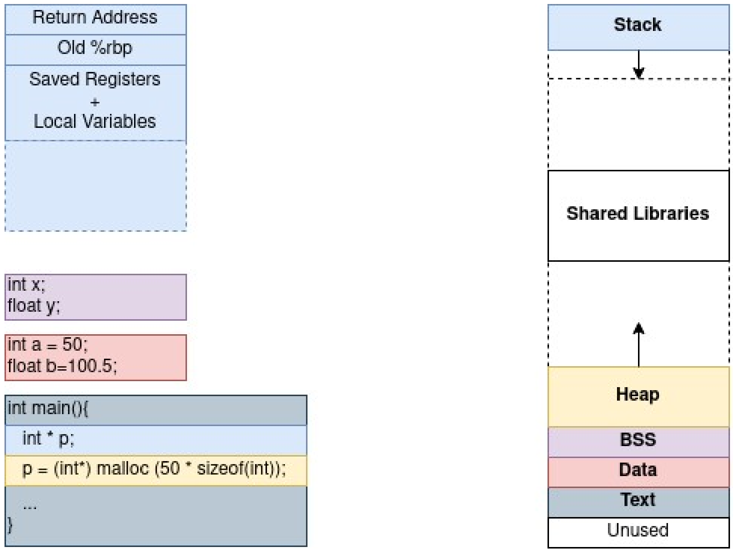


# Virtual Memory Layout

- Each process has the same uniform view of memory, which is known as its **virtual address space**.
- Variables are stored in memory
  - Global and static local variables in `data` or `bss` sections
  - Dynamically allocated variables in the `heap`
  - Some function parameters and local variables on the `stack`.



# Address Spaces



## Addressing mode

- An **addressing mode is an expression that calculates an address in memory to be read from/written to**.
- These expressions are used as the **source or destination** for a `mov` instruction and other instructions that access memory.
- It can be **direct or indirect**.

# Direct Addressing Mode

- The address field in the instruction **contains the effective address** of the operand and no intermediate memory access is required.
- Nowadays it is **rarely used**.
- Example: `movl 2000, %ecx`
  - Read four bytes starting at address `2000`
  - Load the value into the `%ecx` register
  - Notice the missing $ in front of `2000`
- Useful when **the address is known in advance**.

# Indirect Addressing Mode

- The address field in the instruction **contains the memory location or register where the effective address of the operand is present**.
- Load or store from a **previously-computed address**
  - Register with the address is embedded in the instruction
  - Example: `leaq num1(%rip), %rax`
    - `%rax` stores the 64-bit base address (e.g. 2000) of `num1`
- A subsequent instruction **reads from or writes to the address stored in register**
  - Example: `movl (%rax), %ecx`
    - Read four bytes starting at address pointed by `%rax` (e.g. 2000)
    - Load the value into the `%ecx` register

# Addressing Memory (I)

- **Simple** memory addressing modes
  - Normal: (R) $\rightarrow$ Mem[Reg[R]]
    - Register R specifies memory address
    - `movl (%rcx),%eax`
  - Displacement: D(R) $\rightarrow$ Mem[Reg[R]+D]
    - Register R specifies start of memory region
    - Constant displacement D specifies offset
    - `movl 8(%rbp),%edx`
    - `movl num1(%rip),%ecx`
- **Complete** memory addressing modes
  - D(Rb,Ri,S) $\rightarrow$ Mem[D + Reg[Rb] + Reg[Ri] * S]
    - D: Constant “displacement” in bytes
    - Rb: Base register: Any of the 16 integer registers
    - Ri: Index register: Any, except for `%rsp` and, unlikely, `%rbp`, either
    - S: Scale: 1, 2, 4, or 8 (why these numbers?)

# Addressing Memory (II)

- `D(Rb,Ri,S)`
  - The effective address corresponding to this specification is `(D + R[base] + R[index] * S)`.

| Register | Value |
|---|---|
| `%rdx` | 0xF000 |
| `%rcx` | 0x100 |

| Expression | Computation | Result |
|---|---|---|
| 0x8(`%rdx`) | 0x8 + 0xF000 | 0xF008 |
| (`%rdx`,`%rcx`) | 0xF000 + 0x100 | 0xF100 |
| (`%rdx`,`%rcx`,4) | 0xF000 + 0x100 * 4 | 0xF400 |
| 0x80(,`%rdx`,2) | 0x80 + 0xF000 * 2 | 0x1E080 |

# `leaq` (I)

- **Load effective address** of M into R
  - The `leaq` instruction **copies** an **effective address** from one place to another.

```
int vec[] = {1,2,3};
void f()
{
 int ptr = &vec[2];
 *ptr = 10;
}
```

```
f:
 leaq vec(%rip),%rsi       #%rsi = &vec[0]
 movl $2,%rcx
 leaq (%rsi,%rcx,4),%rsi   # %rsi = &vec[2]
 movl $10, (%rsi)

 ret
```

- Unlike `mov`, which copies data at the address S to the D, `leaq` **copies the value of S itself to the D**.

- Computing **arithmetic expressions** with `leaq`

```
long add_3(long x) {
   t = x + 3;
   return t;
}
```

```
add_3:
  leaq 3(%rdi),%rax
  ret
```

```
t <- x + 3 // t = x + 3
```

```
long mul_5(long x) {
  t = x * 5;
  return t;
}
```

```
mul_5:
  leaq (%rdi,%rdi,4),%rax
  ret
```

```
t <- x + x * 4 // t = x *
         5
```

```
long mul_12(long x) {
  t = x * 12;
  return t;
}
```

```
mul_12:
  leaq (%rdi,%rdi,2),%rax
  shlq $2,%rax
  ret
```

```
t <- x + x*2 // t = x * 3
t = t << 2; // t = t * 4
```

```
    movq x(%rip), %rax
    movq y(%rip), %rcx
    leaq 6(%rax), %rdx              # %rdx = x + 6
    leaq (%rax, %rcx), %rdx         # %rdx = x + y
    leaq (%rax, %rcx, 4), %rdx      # %rdx = x + y * 4
    leaq 7(%rax, %rcx, 8), %rdx     # %rdx = x + y * 8 + 7
    leaq 7(, %rcx, 8), %rdx         # %rdx = + y * 8 + 7
```

# Arrays

## Arrays

- An array **is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations**.
- The concept is to **collect many objects of the same data type**.
- Arrays work on an index system starting from 0 to $(n - 1)$, where $n$ is the size of the array.
- There are majorly two types of arrays, they are:
  - One-Dimensional Arrays.
  - Multi-Dimensional Arrays.

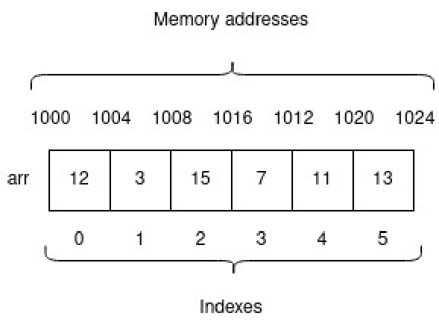
# One-Dimensional Arrays: Allocating

- `T A[L]`
  - Array of data type `T` and length `L`
  - Contiguously allocated region of `L * sizeof(T)` bytes
- Example

```
//Initialized
int arr1[6] = {12, 3, 15, 7, 11, 13};
int arr2[] = {12, 3, 15, 7, 11, 13};
int *arr3 = {12, 3, 15, 7, 11, 13};

//Uninitialized
int arr4[6];
```

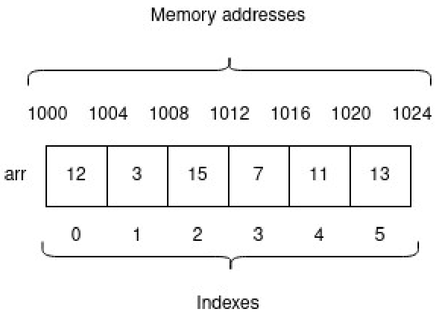
- Arrays in C are just **pointers to the start of the array**.

# One-Dimensional Arrays: Accessing (I)




- arr[3] = ??
- &arr[3] = ??
  - `&A[i] = A + i * sizeof(T)`
    - If you want to access a particular element at position (index) `i`, you start at the base memory address (array name) and step forward `i` times multiply by size of (data type) each element.
      `address of element = base address + (index * size of element)`

# One-Dimensional Arrays: Accessing (II)




```c
int get_element(int *vec, int i){
  return vec[i];
}
```

```asm
get_element:  #%rdi=vec rsi=i
    movl (%rdi,%rsi,4),%eax
    ret
```

```c
int get_element(int *vec, int i){
  return *(vec + i);
}
```

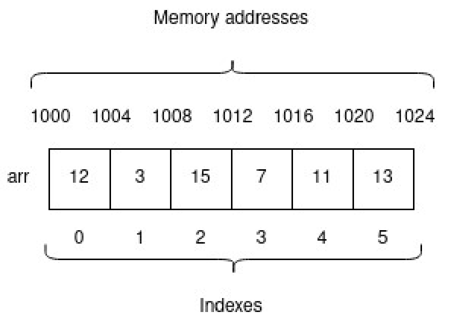
# One-Dimensional Arrays: Traversing (I)

- Traversal **is the process in which we visit every element of the array**.
  - Using an **index variable**.

```c
int i;
for (i = 0; i < 6; i++) {
  printf("%d\n",arr[i]);
}
```

  - Using **pointers**.

```c
int* p;
for (p = arr; p < arr + 6; p++) {
  printf("%d\n",*p);
}
```




```c
void array_incr(int *vec, int n){
  int * p;
  for (p = vec; p < vec + n; p++)
    (*p)++;
}
```

```
array_incr:         #%rdi = vec, %rsi = n
  movq %rsi, %rcx
  movq $0, %rax
.L1:
  addl $1, (%rdi,%rax,4)
  incq %rax
  loop .L1
  ret
```

# Multi-Dimensional Arrays (2D): Allocating (I)

- `T M[R][C]`
  - Array 2D of data type `T` with `R` rows, `C` columns.

Column indexes

| m | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 12 | 3 | 15 | 7 | 11 | 13 |
| 1 | 2 | 3 | 8 | 25 | 10 | 14 |
| 2 | 1 | 24 | 28 | 31 | 4 | 8 |

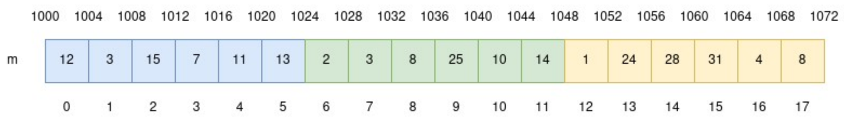
Row indexes

  - m[1][3] = ???

- `T M[R][C]`

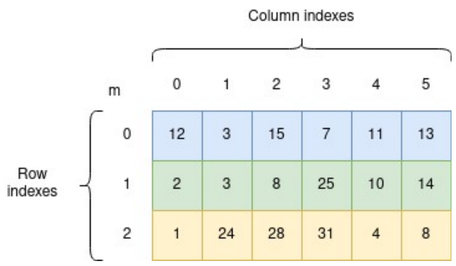
```c
//Initialized
int m1[3][6] = {{12,3,15,7,11,13}, {2,3,6,25,10,14},{1,24,28,31,4,8}};
int m2[][6] = {{12,3,15,7,11,13}, {2,3,6,25,10,14},{1,24,28,31,4,8}};
int m3[][6] = {{12,3,15,7,11,13,2,3,6,25,10,14,1,24,28,31,4,8}};
//Uninitialized
int m4[3][6];
```

- **Contiguously allocated region of `R * C * sizeof(T)` bytes**




- m[1][3] = m[1 * 6 + 3] = m[9] = 25
- &m[1][3] = ??
  - `&M[i][j] = M + (i * C + j) * sizeof(T)`

# Multi-Dimensional Arrays(2D): Accessing




```c
int get_m_element(int m[][6], int i, int j){
  return m[i][j];
}
int m[3][6] = {...}
int r = get_m_element(m, 1,3);
```

```
get_m_element:   #%rdi = m %rsi=i %rdx=j
  leaq (%rsi,%rsi,4), %rax # %rax = i * 5
  leaq (%rax,%rsi), %rax # %rax = i * 6
  leaq (%rax,%rdx), %rax # %rax = i * 6 + j
  movq (%rdi,%rax, 4), %rax

  ret
```

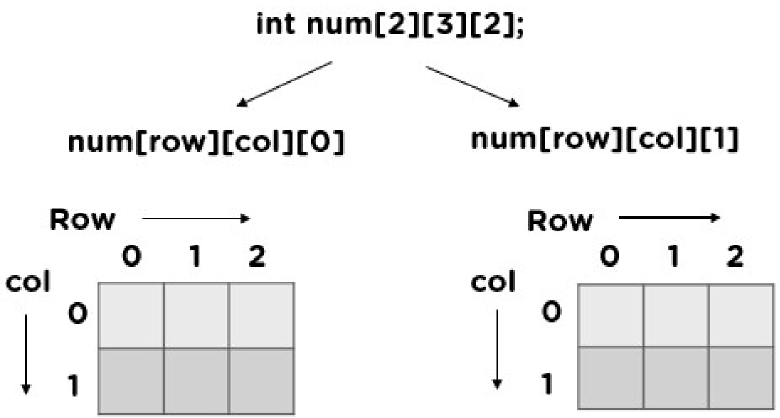
```c
int get_m_element(int *p, int i, int j){
  return *(p + i * 6 + j);
}
int m[3][6] = {...}
int r = get_m_element(&m[0][0], 1,3);
```

```c
void mat_incr(int m[][6], int r, int c){
  int i, j;
  for(i = 0; i < r; i++){
    for(j = 0; j < c; j++){
      m[i][j] +=1;
    }
  }
}
int m[3][6] = {...}
int r = mat_incr(m, 3,6);
```

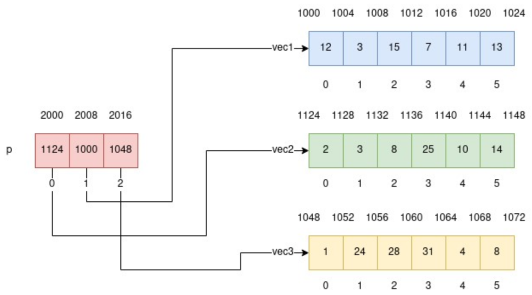
```c
void mat_incr2(int *vec, int r, int c){
  int *p = vec;
  for(p = vec; p < vec + (r * c); p++){
    (*p)++;

  }
}
int m[3][6] = {...}
int r = mat_incr2(&m[0][0], 3,6);
```

```
mat_incr:
  andq $0, %r8 #i
  movq %rsi, %rcx
.LI:
  movq %rcx, %rsi
  movq %rdx, %rcx
  andq $0, %r9 #j
.LJ:

  andq $0, %rax
  leaq (%r8,%r8,4), %rax
  leaq (%rax,%r8), %rax
  leaq (%rax,%r9), %rax
  addl $1, (%rdi, %rax,4)
  incq %r9
  loop .LJ

  incq %r8

  movq %rsi, %rcx

  loop .LI
  ret
```

## Multi-Dimensional Arrays (3D)



```
#define PCOUNT 3
#define LEN 6
int vec1[LEN] = {12,3,15,7,11,13};
int vec2[LEN] = {2,3,6,25,10,14};
int vec3[LEN] = {1,24,28,31,4,8};

int *p[PCOUNT] = {vec2,vec1,vec3};
```




- Variable `p` denotes array of 3 elements
- Each element **is a pointer**
- Each pointer points to array of `int`'s

# Multi-Level Arrays: Accessing

```c
int get_p_element(int **p, int i, int j){
  return p[i][j];
}
```

```
get_p_element:
  movq (%rdi, %rsi, 8), %rdi
  movl (%rdi, %rdx, 4), %eax

  ret
```

# Multi-Level Arrays vs Multi-Dimensional Arrays

- Accesses looks similar in C, but addresses are computed in a different way (more otpimized in Multi-Level Arrays)

```c
int get_p_element(int **p, int i, int j){
  return p[i][j];
}
```

```
get_p_element:
  movq (%rdi, %rsi, 8), %rdi
  movl (%rdi, %rdx, 4), %eax

  ret
```

- M[M[p + i * 8]+ j * 4]

```c
int get_m_element(int m[][6], int i, int j){
  return m[i][j];
}
```

```
get_m_element:   #%rdi = m %rsi=i %rdx=j
  leaq (%rsi,%rsi,4), %rax # %rax = i * 5
  leaq (%rax,%rsi), %rax # %rax = i * 6
  leaq (%rax,%rdx), %rax # %rax = i * 6 + j
  movq (%rdi,%rax, 4), %rax

  ret
```

- M[m + (i * 6 + j) * 4]

# Access optimizations (I)

- Whenever a program uses some constant as an array dimension or buffer size, it is best to associate a name with it via a `#define` declaration, and then use this name consistently, rather than the numeric value.
- This is known at compile time, so the compiler can generate optimized code

```
#define N 16
/* Get element a[i][j] */
int fix_elem(int m[N][N], int i, int j){
 return m[i][j];
}
```

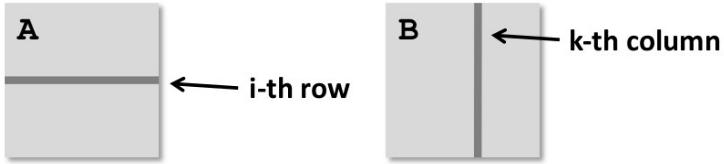
```
                            #m in %rdi, i in %rsi, j in %rdx
salq $6, %rsi               # (16*4)*i = 64*i
addq %rsi, %rdi             # m + 64*i
movl (%rdi,%rdx,4), %eax    # Mem[m + 64*i + 4*j]
```

  - Address: M + i * (C * K) + j * K
  - Where C = 16, K = 4
  - It can compute in advance: i * (C * K)

# Access optimizations (II)

- Within a loop, **index computations can often be optimized by exploiting the regularity of the access patterns**.

```c
#define N 16
/* Compute i,k of fixed matrix product */
int fix_prod_elem(int A[N][N], int B[N][N], int i, int k){
 int j;
 int result = 0;
 for(j=0; j < N; j++){
  result += A[i][j] * B[j][k];
 }
 return result;
}
```



# Access optimizations (III)

- The loop will access just the **elements of row `i` of array `A`**
  - Create a local pointer to provide direct access to row `i`
  - Initialize pointer to `&A[i][0]`, and so array element `A[i][j]` can be accessed as `Arow[j]`
- The loop will access the **elements of array `B` as `B[0][k]`, `B[1][k]`, ... , `B[15][k]` in sequence**
  - These elements occupy positions in memory starting with `&B[0][k]` and spaced (16 * 4) 64 bytes apart.
  - Use a pointer `Bcol` to access these successive locations

```c
#define N 16
int fix_prod_elem_opt(int A[N][N], int B[N][N], int i, int k) {
 int *Arow = &A[i][0]; /* Points to elements in row i of A */
 int *Bcol = &B[0][k]; /* Points to elements in column k of B*/
 int *Bend = &B[N][k]; /* Marks stopping point for Bcol */
 int j, result = 0;
 do{
  result += *Arow * *Bcol;
  Arow++;
  Bcol += N;
 }while(Bcol != Bend);
 return result;
}
```

# Access optimizations (IV)

```
fix_prod_ele:                  #%rdi=A, %rsi=B, %rdx=i, %rcx=k
  salq $6, %rdx               #64 * i
  addq %rdx, %rdi             #Arow = &A[i][0]
  leaq (%rsi,%rcx,4), %rcx    #Bcol = &B[0][k]
  leaq 1024(%rcx), %rsi       #Bend = &B[N][k]
  movl $0, %eax               #result = 0
.L7:                           #loop:
  movl (%rdi), %edx           #Get *Arow
  imull (%rcx), %edx          #Multiply by *Bcol
  addl %edx, %eax             #Add to result
  addq $4, %rdi               #Arow++
  addq $64, %rcx              #Bcol += N
  cmpq %rsi, %rcx             #Compare Bcol:Bend
  jne                         #If !=, goto loop
.L7
  ret
```

- Machine code considers every pointer to be a byte address
  - `%rsi` is incremented by 64 and `%rdi` by 4 within the loop