ARQCP Flashcard
“Introduction to C”

Operations, conditionals, loops …

Operators:

- Arithmetic: +, -, *, /, %, ++, --
- Assignment: =, +=, -+, *=, …
- Relational: <, >, <=, >=, ==, !=
- Logic: &&, ||, !
- Bitwise: &, |, ^, <<, >>

Language constructs:

- if () {} else {}
- while () {}
- do {} while ()
- for(i = 0; i < 100; i ++){}
- switch () {case 0: …}
- break, continue, return

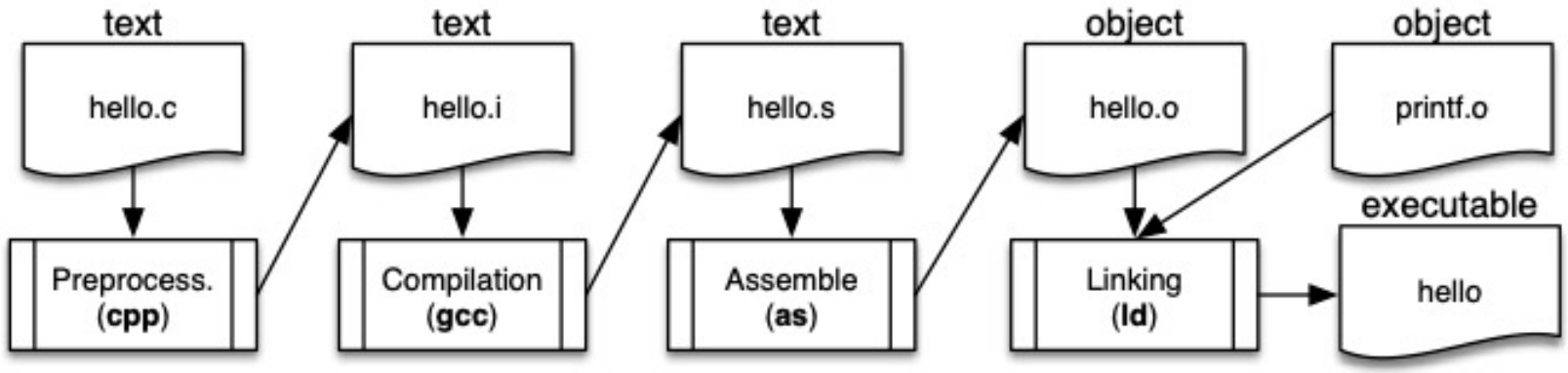
Note:

No exception handling statements !

Compiling C programs

C programs must be transformed into machine-code so they can be executed in a process called **compilation**

The compilation process involves several other steps:



GCC

The process of compilation can be made with the help of the GCC compiler

This are some of the important commands of GCC:

option	Description
-Wall	all warnings – **use always!**
-o*filename*	output filename for object or executable
-c	compile only, do not link; used to create an object file (.o) for a single (non-main) .c file (module)
-g	insert debugging information
-E	stop after the preprocessing stage; output goes to standard output
-v	show information about gcc and/or compilation process
-S	performs preprocessing and compilation only; that is, convert C source into assembly
-save-temps	keep temporary files created (.i, .s, .o, ...)
-l*library-name*	link with library called *library-name*
-I*dir*	add *dir* to the list of dirs to be searched for header files
-L*dir*	add *dir* to the list of dirs to be searched for the libraries specified with -l;

GCC

Notes about GCC:

Read all the error or warning message since they can be pretty helpful

Always fix the errors that come first, don’t ignore any errors or warnings

Some errors may be caused by a simple missing ;

C preprocessor

The C preprocessor (cpp) allows defining macros, which are brief abbreviations for longer constructs

The preprocessor directives start with a ‘#’ at the beginning of the line and are used for:

- Inserting content of another file into file to be compiled : #include
- Conditional compilation: #if; #ifndef
- Definition of macros and constants: #define

Before compilation, all this information is read by the preprocessor

Example 1:

#include <stdio.h> //searches for stdio.h in the system defined directories
#include ”example.h” //searches for example.h in the current directory

Example 2:
#define MAX_VALUE 100
#define check(x) ((x) < MAX_VALUE)

if check(x){…} -> this code is turned into -> if (x < 100) {…}

Note:
Use cpp wisely since is easy to make mistakes that are not “debuggable”

C data types in x86-64

```
char x = 0xAF              //hexadecimal declaration

char f = 0b00110011   //bit declaration

char c = 'A'

char b = 100

int i = -2343234

unsigned int ui = 100000000

float pi = 3.14

double longer_pi = 3.14159265359
```

Integer types

Type	Storage size	Value range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short	2 bytes	-32 768 to 32 767
unsigned short	2 bytes	0 to 65 535
int	4 bytes	-2 147 483 648 to 2 147 483 647
unsigned int	4 bytes	0 to 4 294 967 295
long	8 bytes	-9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
unsigned long	8 bytes	0 to 18 446 744 073 709 551 615

Floating-point types

Type	Storage size	Value range	Precision
float	4 bytes	1.2E-38 to 3.4E+38	6 significant digits
double	8 bytes	2.3E-308 to 1.7E+308	15 significant digits

C data types notes

The storage size of some types **varies** among architectures
E.g. A long is 4 bytes in IA32 and 8 bytes in x86-64 machines

char is a misleading type since it give the perspective that the variable is the character but it is in reality the ASCII code of that character

The **void** type comprises an empty set of values; it is an incomplete type

You **cannot** define variables of type **void**, however we can:
Indicate that a function as no return value with void. E.g: void set_name()
Indicate that a function as no parameters. E.g: int get_age(void)
Define a pointer that does not specify the type that it points to. E.g: void *ptr

There are 2 kinds of type conversions

Implicit:
int a = 1000; char b = a // b = -24 (lower 8 bits of a = … 11101000)

Explicit:
float = 1.2; int d = (float) f; d = 1

Sizeof function

C has a unary compile-time operator **sizeof**, that can be used to get the storage size of variables and data types

Example:
sizeof(int): returns the size of int
sizeof(a): returns the size of a
sizeof(char): returns the size of char that will be always 1

Note:

While, for most modern systems, the char type has 8 bits, there is no guarantee that this is always true

The number of bits of type char is defined in the CHAR_BIT constant in <limits.h>

The <limits.h> file is very useful since it contains a lot of information of the data types like:
CHAR_MAX, CHAR_MIN, INT_MAX …

Sizeof function - Example

Listing 3: sizeof.c

```c
#include <stdio.h> // needed for printf
#include <limits.h> // needed for CHAR_BIT, INT_MAX, INT_MIN
#include <float.h> // needed for FLT_MAX, FLT_MIN, FLT_DIG

int main() {
   char n='A';

   printf("\nStorage size for variable n: %lu\n", sizeof(n));

   printf("\nStorage size for char: %lu\n", sizeof(char));
   printf("Number of bits in a char: %d\n", CHAR_BIT);

   printf("\nStorage size for int: %lu\n", sizeof(int));
   printf("Minimum int value: %d\n", INT_MIN );
   printf("Maximum int value: %d\n", INT_MAX );

   printf("\nStorage size for float : %lu \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value for float: %d\n", FLT_DIG );

   printf("\nStorage size for double=%lu\n", sizeof(double));

   return 0;
}
```

Output of the example (Listing 2; sizeof.c)

```
Storage size for variable n: 1

Storage size for char: 1
Number of bits in a char: 8

Storage size for int: 4
Minimum int value: -2147483648
Maximum int value: 2147483647

Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value for float: 6

Storage size for double=8
```

Printf ()

In order to print information on the screen the **printf** function is used, but has ”codes” for each data type:

printf() format specifiers quick reference

- %d or %i: Signed decimal integer
- %u: Unsigned decimal integer
- %lu: Unsigned long integer
- %f: Decimal floating point, lowercase
- %E: Scientific notation (mantissa/exponent), uppercase
- %c: Character
- %s: String of characters
- see: `http://www.cplusplus.com/reference/cstdio/printf/`

Arrays in C

C allows to define arrays of elements of the same type

Initially it will only be possible to allocate the memory wanted before running, later on this file it is mentioned the malloc function that will allow to allocate memory dynamically

Examples of arrays defined statically (fixed size):
int a [10];
int a [] = {1,2,3,4,5}; //array of 5 integers, initialized to 1 thru 5
int a2[1000] = 0; // array of 1000 integers all with the value 0
short s[100]; //array of 100 shorts
float m[10][10] // 10x10 matrix (array of arrays)

The accessing of the element N is done the same away as in Java, by accessing the position N-1

Note:
GCC doesn't check when there is an overflow of an array
E.g: int x[10]; x[10] = 5

The arrays **cannot** be defined after being declared
E.g: int v [10]; v = {1,2,3,4,5};

In order to copy arrays use **memcpy(dest, src, size)**

Arrays in C

The function **sizeof()** only works for statically defined arrays, within the scope they are declared !

E.g:

```
{
   int a[10];
   printf("%lu",sizeof(a)); //prints 40 in x86-64

}
```

The {} define the scope of these statements

Size of the array can be computed by sizeof(a)/sizeof(a[0])

The same logic is applied to when an array is passed as an argument to a function !

Ways to know the array size:
Save it in a variable and then passing it in a parameter
Save it in a global variable
Define a data structure to save the array and its size
Defining a value inside of the array that indicates its end (e.g: -1)

Strings in C

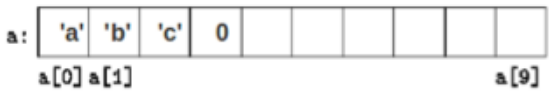
In C there is no concept such as an string

What exists in c is an array of chars that ends with the value zero (char ‘\0’) that represents NULL

E.g:

char a[10] = “abc”;

This defines an array of 10 chars that contains 4 chars, 3 chars (“abc”) and the final one (‘\0’)



Note:
There is a difference in using ‘’ and “”
The first one is used to chars and the other to strings

Arrays in C

Since strings are arrays of chars, we cannot copy strings by simply making something like:

char s [] = “Hello world”

s = “Not possible”

In order to copy strings we will apply the same logic of the arrays and use a function

To copy strings use **strncpy (dest_str, src_str, n_chars)** (always count with the ending char in the number of chars)

Note:
It is the programmers responsibility to make sure that the destination string has enough space to allocate the source string

Good practices

In C use good practices to have a good code

Use variables and function names that sugest what both define or make

Ident all the code

Comment the code

Avoid global variables

Initialize the variables before using them

Always compile with –Wall in order to don’t ignore the gcc warnings

Use C style naming conventions:

E.g: get_radius() instead of GetRadius()
Use i and j inside loops

ARQCP Flashcard
“Introduction to Assembly”

Section Identifiers

.section .data //identifies data (variables)

.section .bss //section used to define uninitialized memory areas

.section .text //identifies code (functions)

Note: ".section" can be omitted

Special Chars

. //starts an assembler directive

//starts a comment

% //starts a register name

$ //starts an immediate value

Global variables

Global variables should be declared in the .data section with .global

The data type and initial value MUST be defined

BIGGER types (more bytes) should be declared first , then the smaller types.

Data Types

.quad //8 bytes (64 bits)

.long and .int //4 bytes (32 bits)

.short //2 bytes (16 bits)

.byte //1 byte (8 bits)

.ascii //string with no automatic trailing zero byte ’/0’

.asciz //string terminated with zero

.float //4 bytes (32 bits), floating point

.double //8 bytes (64 bits), floating point

Note: C | Assembly

Long: 8 bytes | 4 bytes

Declarations

Integer

```
number:      #variable name
        .int 5  #initialization value
```

String

```
message:                          #variable name
        .asciz “Hello, World!”    #initialization value
```

Define Constants

Defined in the .data section

Unlike variables, if we define a constant we are not reserving memory space in the final program

Constants are replaced by their value during the generation of the code.

Declaration

.equ LINUX_SYS_CALL, 0x80

Usage

Movq $LINUX_SYS_CALL, %rax

Reserve generic memory areas

Defined in the .bss section

.comm //Declares a global memory area

.lcomm //Declares a local memory area

Declaration

```
.section .bss
        .lcomm buffer, 10000
```

This code above declares a memory area of 10000 bytes with the identifier buffer and can only be referenced by code belonging to the same module as it was declared with .lcomm

Registers (32 bits)

%ax //accumulator register, used for arithmetic operations (return)

%cx //counter register, used in shift/rotate instructions and loops

%dx //data register, used in arithmetic operations and I/O operations

%bx //base register, used as a pointer to data located in segment register DS, when in segmented mode (callee saved)

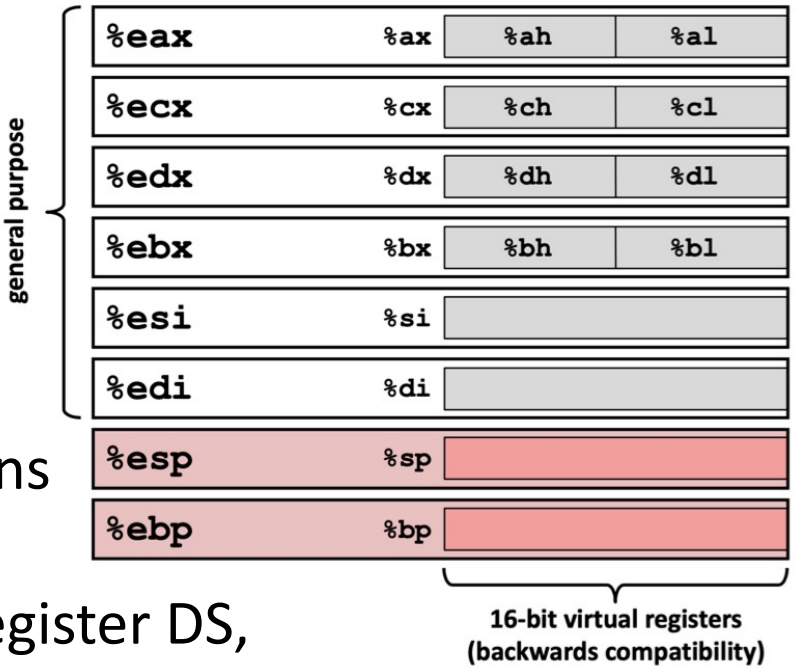
%si //source index, used as a pointer to a source in stream operations

%di //destination index, used as a pointer to a destination in stream operations

%sp //stack pointer, used to point to the top of the stack

%bp //base pointer, used to point to the bottom of the stack

Reserved to stack operations



Registers x64

Operand specifiers

Data type	**Suffix**	Size (bytes)
byte	b	1
word	w	2
long (double word)	l	4
quad word	q	8

Instructions

MOV Instruction

Used to copy data

Usage: **mov *origin*, *destination***

Origin can be a **memory address**, a **constant** or a **register**

Destination can be a **memory address** or a **register**

The operation size must be expressed by adding a char at the end of the instruction (8 (b), 16 (w), 32 (l), 64(q) bits)

2 memory addresses CANNOT be used at the same MOV operation

Origin and Destination must be of the same size

MOV Instruction – Effects

In the most cases the mov instruction will update only the specific register bytes or memory location indicated by the destination operand

But there is an exception:

When doing **MOVL** and the **destination** is a register, it will define the **most significant 4 bytes to 0**

Why:
This exception arises from the convention, adopted in x86-64 that any instruction that generates a 32-bit value for a register also sets the high-order portion (most significant 4 bytes) to 0

MOV Instruction – Examples

```
movq $5236, %rax    # moves the integer value 5236 to RAX

movl $-345, %ecx    # moves the integer value -345 to ECX
                    # the 32 most significant bytes of RCX are set to zero

movw $0xFFB1, %dx   # moves the value -79 (0xFFB1 in hexadecimal)
                    # to the least significant 16 bits of RDX

movb $0x0A, %al     # moves the value 10 (0x0A in hexadecimal)
                    # to the least significant byte of RAX
```

%RIP – Relative addressing

A global variable named *a* is referenced as *a(%rip)* rather than *a*

This style of reference supports *position-independent code,* it is a security feature that supports *position independent executables* (PIE), this are programs that work independently of where their code is loaded into memory

So every time that the program is runed the program's functions and global variables have different addresses

In order to work, the variables are referenced relatively to the current value of the program counter (the %rip register in x86-64)

MOV Instruction – Examples

```
.section .data

#declare a variable called 'myinteger'
myinteger:
    .int 5

.section .text

function:
  movl myinteger(%rip), %eax   # copy value of variable (in memory) to register
  ...                          # do something with the value...
  movl %eax, myinteger(%rip)   # copy register value to variable (in memory)
  ...
  ret
```

MOVABSQ Instruction

MOVQ instruction can only have source operands that can be represented as 32-bit two's complement numbers (absolute value)
Then this value is sign extended to produce the 64-bit value to the destination

MOVABSQ (move absolute quad word) instruction can have a 64-bit immediate value as its **source and can only have a register as a destination**

Usage: movabsq imm, register

```
movabsq $0x0011223344556677, %rax # %rax = 0x0011223344556677
movb $-1, %al                     # %rax = 0x0011223344556FF
movw $-1, %ax                     # %rax = 0x0011223344455FFF
movl $-1, %eax                    # %rax = 0x00000000FFFFFFFF
movq $-1, %rax                    # %rax = 0xFFFFFFFFFFFFFFFF
```

MOVZ Instruction

The MOVZ instruction fills out the remaining bytes of the destination with zeros

Each instruction must have the last 2 chars as size designators, the first one referencing the source and the second the destination

Usage: movz [bw|bl|bq|wl|wq] source, destination

There is no movz instruction from word to quad, since the movl already fills the most significant bytes to 0 (when the destination is a register)

Instruction	Description
movzbw	Move zero-extended byte to word
movzbl	Move zero-extended byte to double word
movzbq	Move zero-extended byte to quad word
movzwl	Move zero-extended word to double word
movzwq	Move zero-extended word to quad word

MOVS Instruction

The MOVS instruction follows the same logic as the MOVZ but instead of filling with zeros, **the remaining bytes from the destination are filled by sign extension**, it is **replicated the most significant bit of the source** operand

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movsbq	Move sign-extended byte to quad word
movswl	Move sign-extended word to double word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word

MOVABSQ | MOVZ | MOVS - Example

```
movabsq $0x00112233445566 77 , %rax # %rax = 0x00112233445566 77
movb    $0xAA , %dl                 # %dl  = 0xAA
movb    %dl , %al                   # %rax = 0x00112233445566AA
movsbq  %dl , %rax                  # %rax = 0xFFFFFFFFFFFFFFAA
movzbq  %dl , %rax                  # %rax = 0x00000000000000AA
```

ARQCP Flashcard
“Assembly: Arithmetic Operations”

ADD Instruction

The ADD instruction adds two integers

Usage: add *origin*, *destination*

Performs the operation *destination = destination + origin*
The result is placed in the destination

Origin can be a memory address, constant or a register

Destination can be a memory address or a register

Like before origin and destination cant be a memory address at the same operation

The ADD operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

ADD Instruction – Examples

```
addb $10, %al              # adds 10 to the 8-bit AL register; AL=AL+10
addw %bx, %cx              # adds the value of BX to CX (16 bits); CX=CX+BX
addl var1(%rip), %eax      # adds the 32-bit value in var1 to EAX; EAX=EAX+var1
addl %eax, %eax            # adds EAX to itself; EAX=EAX+EAX
addq %rcx, %rax            # adds RCX to RAX; RAX=RAX+RCX
```

SUB Instruction

The SUB instruction subtracts two integers

Usage: sub *origin*, *destination*

Performs the operation *destination = destination - origin*
The result is placed in the destination

Origin can be a memory address, constant or a register

Destination can be a memory address or a register

Like before origin and destination cant be a memory address at the same operation

The SUB operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

SUB Instruction - Examples

```
subl $10, %eax          # subtract 10 to the current value of EAX; EAX=EAX-10

subw %bx, %cx           # subtract the value of BX to CX (16 bits); CX=CX-BX

subb var1(%rip), %al    # subtract the 8-bit value in var1 to AL; AL=AL-var1

subq %rcx, %rax         # subtract RCX to RAX; RAX=RAX-RCX
```

INC & DEC Instructions

The INC and Dec instructions increment (INC) and decrement (DEC) an integer by one, respectively

Usage: inc *destination*
dec *destination*

Performs the operation *destination = destination +/- 1*

Destination can be a memory address or a register

The INC & DEC operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

INC & DEC Instructions – Examples

```
incq %rax    # RAX=RAX+1 (64 bits)

incl %eax    # EAX=EAX+1 (32 bits)

incw %bx     # BX=BX+1 (16 bits)

decb %cl     # CL=CL-1 (8 bits)
```

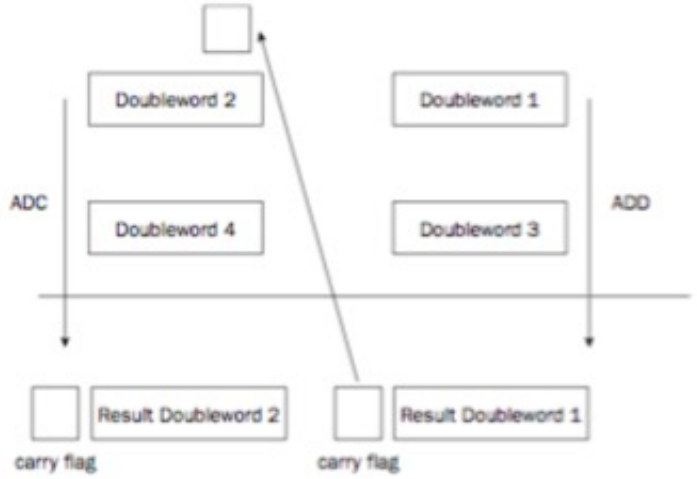
ADC (Add with carry) Instruction

The ADC instruction can be used to add two integer values, along with the value contained in the carry flag set by a previous addition

Performs the operation: *destination = destination + origin + CF*

The ADC operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

Note:
The presence of a carry flag on unsigned addition
Indicates that the result is wrong



ADC Instruction - Example

```
.global adctest
adctest:
    ...
    movb $0xFF, %al
    movb $0x1, %ah
    movb $0x1, %bl
    movb $0x0, %bh

    # bl = bl + al (8 bits)
    addb %al, %bl

    # bh = bh + ah + CF (8 bits)
    adcb %ah, %bh

    ...
    ret
```

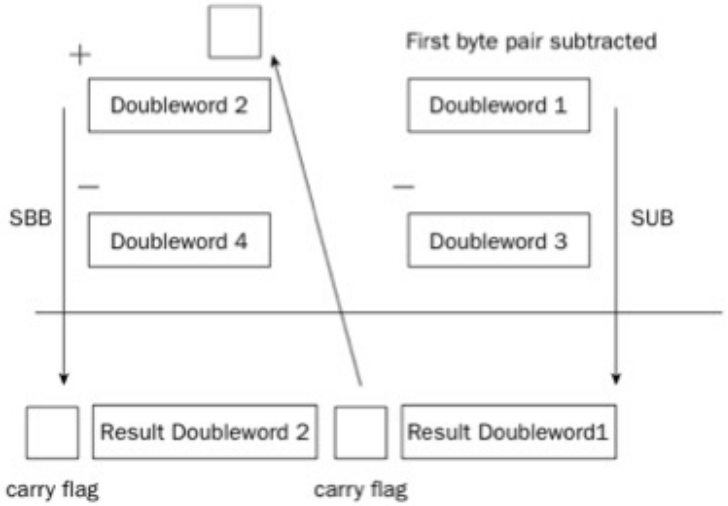
SBB (Subtract with Borrow) Instruction

The SBB instruction can be used to subtract two integer values, along with the value contained in the carry flag set by a previous subtraction

Performs the operation: *destination = destination – (origin + CF)*

The SBB operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

Note:
The presence of a carry flag on unsigned subtraction Indicates that the result is wrong



Practical problem

```
int x,y;  // 32 bits (signed)

long fun(){
  long t1 = (long)x + y;
  long t2 = (long)(x+y);
  return t1 - t2;
}
```

The two additions in this function proceed as follows:

Type conversion has higher precedence than addition, so the first addition calls for x to be converted to 64 bits and by operand promotion y is also converted.
Value *t1* is then computed using 64-bit addition

On the other hand, *t2* is computed by performing 32-bit addition and then extend the value to 64 bits

Practical problem

```
fun:
    movslq x(%rip),%rax      # Convert x to long
    movslq y(%rip),%rdx      # Convert y to long
    addq   %rdx,%rax         # t1 (64-bit addition)
    movl   y(%rip), %edx     # Move y to %edx (lower 32 bits of RDX)
    addl   x(%rip), %edx     # t2 (32-bit addition)
    movslq %edx, %rdx        # Convert t2 to long
    subq   %rdx, %rax        # Return t1 - t2
    ret
```

The two additions in this function proceed as follows:

Like said before, the value of *t1* is obtained by first sign extending the arguments. The *movslq* instructions will take the 32 bits of x and y and sign extend them to 64 bits in registers %rax and %rdx.
Then the *addq* instruction performs a 64-bit addition

For *t2* we get the same logic seen before in C but with the *addl* instruction doing the 32-bit addition and then *movslq* converting by sign extending *t2* in a 64-bit value using register %rdx (%edx is part of %rdx)

MUL (Unsigned Multiplication) Instructions

The MUL instruction is used to multiply **two unsigned integers**

Usage: *mul origin*

Performs *destination = origin x operand2*

Size of *origin*	*operand2*	*destination*
8 bits	AL	AX
16 bits	AX	DX:AX
32 bits	EAX	EDX:EAX
64 bits	RAX	RDX:RAX

Origin can be a memory address or a register

Operand*2* is the RAX, EAX, AX or AL register, depending on the size of *origin*

Destination is the RDX:RAX, EDX:EAX, DX:AX or AX registers, depending on the size of *origin*

The MUL operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

MUL Instruction - Example

```
.global multest
multest:

    ...

    movw $200, %ax
    movw $2, %cx

    # multiply %cx by %ax
    # result in %dx:%ax
    mulw %cx

    ...

    ret
```

IMUL (Signed Multiplication) Instructions

The IMUL instruction is used to multiply **two signed integers**

Usage: *imul origin*

imul origin, destination

(1)

For the single operand instruction the usage of registers is the same as presented in MUL

(2)

For the two-operand instruction we need to make sure that we can fit the multiplication result inside the *destination*

Origin can be a memory address, a constant or a register (16, 32,64) bits

Destination can be a (16, 32,64) bits register

IMUL Instruction - Example

```
.global imultest
imultest:
    # ...
    movl $200, %eax
    movl $-2, %ecx
    imull %ecx              # %edx:%eax = %ecx * %eax

    movq $-123, %rcx
    imulq $4, %rcx          # %rcx = 4 * %rcx
    # ...
    ret
```

DIV & IDIV (Unsigned and signed division) Instructions

The DIV and IDIV instructions are used in unsigned/signed division, respectively

Usage: *div divisor*
idiv divisor

Perform the operations:
quotient = dividend / divisor
remainder = dividend mod divisor

Size of *divisor*	*dividend*	*quotient*	*remainder*
8 bits	AX	AL	AH
16 bits	DX:AX	AX	DX
32 bits	EDX:EAX	EAX	EDX
64 bits	RDX:RAX	RAX	RDX

Divisor can be a memory address or a register

Dividend is the RDX:RAX, EDX:EAX, DX:AX or AX register, depending on the size of divisor

The quotient and remainder of the division are put in different sections of the RAX and RDX registers, depending on the size of divisor

DIV Instruction - Example

```
.global divtest
divtest:
...

  # dividend: ax
  movw $100, %ax
  # divisor: cl
  movb $3, %cl

  # divides %ax by %cl
  # remainder in %ah
  # quotient in %al
  divb %cl

  ...
  ret
```

DIV Instruction – Example 2

Note: Initialize all the bits of the dividend

```
.global bad_div

bad_div:
  ...
  # dividend: eax
  movl $100, %eax
  # divisor: ecx
  movl $3, %ecx

  # divides %edx:%eax by %ecx
  # Problem: the unkown content in %edx becomes part of the dividend

  # remainder in %edx
  # quotient in %eax
  divl %ecx

  ...
  ret
```

DIV & IDIV - Dividend

Before we do the DIV or IDIV instruction we must prepare the dividend

For the DIV instruction we must set the higher order part of the dividend to 0

For the IDIV instruction we must set the higher order part of the dividend by sign extension, for that we have a set of instructions:

cbw — sign-extend byte in %*al* to word in %*ax*
cwde — sign-extend word in %*ax* to double word (long) in %*eax*
cwd — sign-extend word in %*ax* to double word (long) in %*dx* : %*ax*
cdq — sign-extend double word (long) in %*eax* to quad word in %*edx* : %*eax*
cdqe — sign-extend double word (long) in %*eax* to quad word in %*rax*
cqo — sign-extend quad in %*rax* to oct word in %*rdx* : %*rax*

IDIV – Dividend Example

```
.global div_ok

div_ok:
  ...
  # dividend: %eax
  movl $-100, %eax
  # converts the signed long in %eax to the signed double long in %edx:%eax
  cltd

  # divisor: ecx
  movl $3, %ecx

  # divides %edx:%eax by %ecx (remainder in %edx, quotient in %eax)
  idivl %ecx
  ...
  ret
```

ARQCP Flashcard
“Assembly: Controlling execution flow”

RFLAG Register

The %rflags is a register of 64-bit length used as a collection of bits with Boolean value (0 – False/inactive | 1 – True/active), that represent the result of logical and arithmetic operations or the state of the processor, in order to control the execution flow of a certain program

This single bits are called *flags* and it's value can change when the code execute instructions

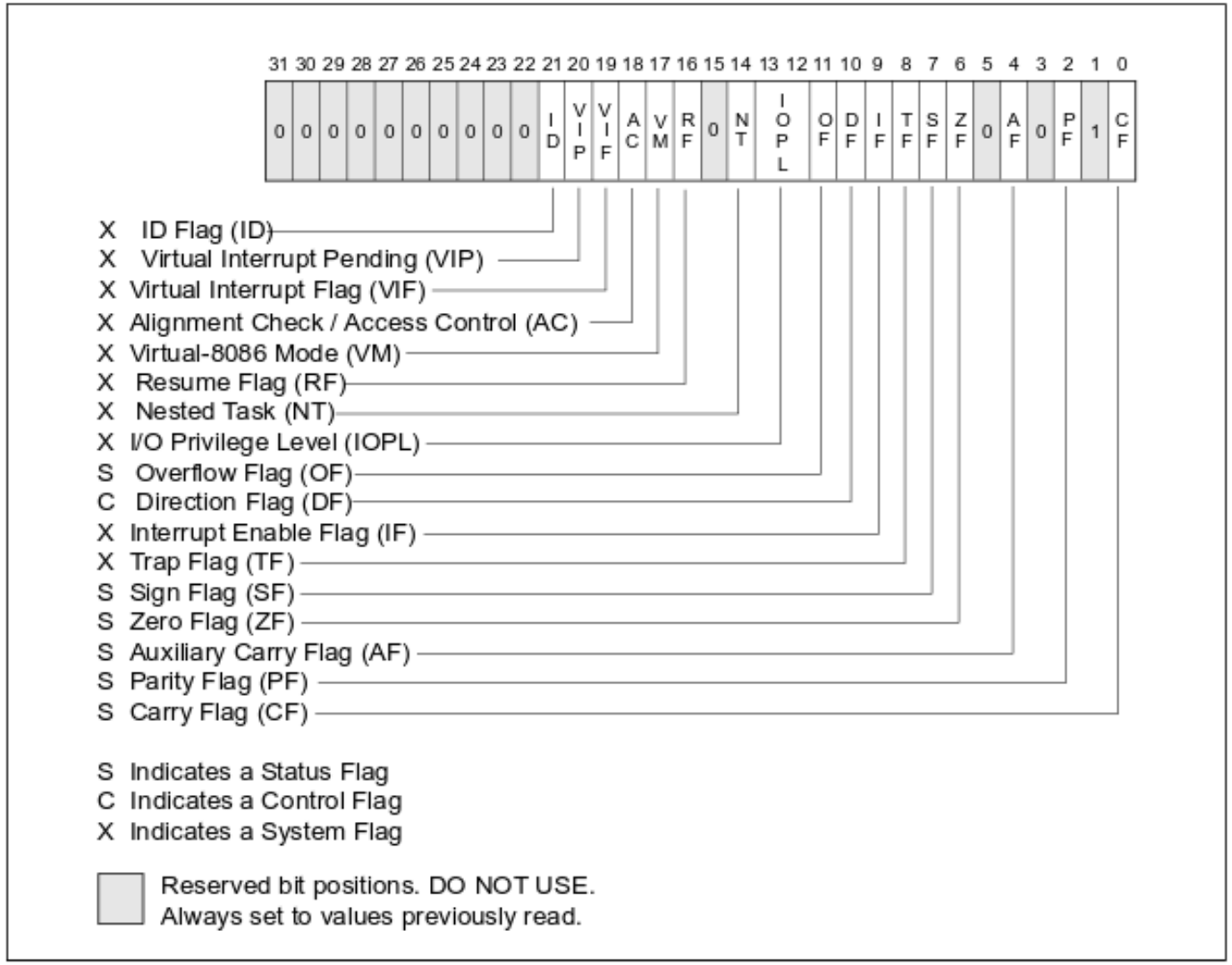
RFLAG Register – Important Flags

CF – Carry Flag (bit 0)
Set with most significant bit carry or borrow;
Cleared otherwise

ZF – Zero Flag (bit 6)
Set if the result is zero;
Cleared otherwise

SF – Sign Flag (bit 7)
Set equal to the most significant bit of result
(0 if positive, 1 if negative)

OF - Overflow Flag (bit 11)
Set if the result is too large (positive number)
or too small (negative number) to fit in destination;
Cleared otherwise



RIP Register

The %rip register is a program counter that indicates the address in memory of the next instruction to be executed

After the instruction's execution, %rip is automatically increased to the address of the next instruction

A way of changing the value of %rip is using jump

A jump instruction can cause the execution to switch to a completely different position in the program by setting %rip to other instruction address, this can be done in 2 ways:

Unconditionally: The instruction pointer is set to a new value

Conditionally: The instruction pointer is set to a new value if a condition is true

Unconditional Jump

Usage: *jmp address*

This instruction changes the RIP register to *address*

Unconditional Jump Example

```
.global jmptest
jmptest:
    ...
    movq %rax, %rcx
    addq %rdx, %rcx
    jmp end

    # this line is never executed!
    movq $1, %rax

end:
    movq $10, %rax

    ...
    ret
```

Conditional Jump

The conditional jumps are **taken or not** depending on the state of the **RFLAGS** register at the time the branch is executed

Each conditional jump instruction examines specific flag bits to determine if the condition is proper for the jump to occur
Some examples of flags:

JE – Jump if equal (ZF = 1)
JL – Jump if less (SF< >OF)
JG – Jump if greater (ZF = 0 e SF = OF)

Like the unconditional jump, they only take one argument indicating the address within the program to jump to

Before a conditional jump, RFLAGS bits must be set by an appropriately operation

Compare Instruction

Usage: *cmp operand1, operand2*

The compare instruction is the most common way to evaluate two values for a conditional jump

It compares the second operand with the first by executing a subtraction (operand2 – operand1)

This instruction **DO NOT** affect the operands, only affect the **RFLAGS** register

Examples:

operand2 == operand1 -> ZF = 1
operand2 > operand1 -> SF = 0
operand2 < operand1 -> SF = 1

The ADD operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

Execution Flow Table

Performing the signed subtraction *R* = *Destination* - *Source* yields a signed result.

Suppose there is no overflow - the usual arithmetic laws holds: if *R* = *Destination* - *Source* > 0 then *Destination* > *Source*.
Having no overflow means *OF* = 0 and *R* > 0 means *SF* = 0.

Now suppose there is an overflow - let's call *O* the most significant, non-sign, bit and *S* the sign bit.
An overflow condition means that either a) Computing the result's *O* needed a borrow and result's *S* didn't or b) result's *O* didn't need a borrow and *S* did.

In case a) since result's *S* didn't need a borrow, the two *S* bits of the operands were either (1, 0) (1, 1) or (0, 0).
Since result's *O* needed a borrow, and thus flipping the first source *S* bit, we must exclude the second and third option.
So the operands sign bits were 1 and 0 (thus *Destination* < *Source*), the result's sign bit *SF* = 0 and *OF* = 1 by hypothesis.

In case b) since result's *S* did need a borrow, the two *S* bits of the operands were (0, 1).
Since *O* didn't need a borrow, the first operand *S* bit has been not changed and we don't need to consider any further case.
So the operands sign bits were 0 and 1 (thus *Destination* > *Source*), the result's sign bit *SF* = 1 and *OF* = 1 by hypothesis.

To recap:

- If *OF = 0* then *Destination* > *Source* => *SF = 0*.
- If *OF = 1* then *Destination* > *Source* => *SF = 1*.

In short *OF* = *SF*.

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF) &~ZF`	Greater (signed)
`jge`	`~(SF^OF)`	Greater or Equal (signed)
`jl`	`(SF^OF)`	Less (signed)
`jle`	`(SF^OF) | ZF`	Less or Equal (signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Execution Flow - Example

Controlling Execution Flow

```
    ...
    # compares %rcx with %rsi through %rsi - %rcx
    cmpq %rcx, %rsi
    jg jmp_rsi_is_greater
    je jmp_rsi_is_equal
    jl jmp_rsi_is_less
jmp_rsi_is_greater:
    movq $1, %rax
    jmp end
jmp_rsi_is_equal:
    movq $0, %rax
    jmp end
jmp_rsi_is_less:
    movq $-1, %rax
end:
    ret
```

Execution Flow – Example 2

Consider the following C code

```c
long x;
long y;

long test_xy(){
  if(x > y)
    return 1;
  else
    return 0;
}
```

Can be written in Assembly as

```
test_xy:
    movq x(%rip), %rdi
    cmpq y(%rip), %rdi
    jle false
    movq $1, %rax
    jmp end
false:
    movq $0, %rax
end:
    ret
```

Execution Flow – Filling Example

Consider the following Assembly code

```
test_xyz:
  movq  x(%rip), %rdi
  movq  y(%rip), %rsi
  movq  z(%rip), %rdx
  movq  %rdi, %rax
  addq  %rsi, %rax
  subq  %rdx, %rax
  cmpq  $-3, %rdi
  jge   .L2
  cmpq  %rdx, %rsi
  jge   .L3
  movq  %rdi, %rax
  imulq %rsi, %rax
  jmp   .L4
.L3:
  movq  %rsi, %rax
  imulq %rdx, %rax
  jmp   .L4
.L2:
  cmpq  $2, %rdi
  jle   .L4
  movq  %rdi, %rax
  imulq %rdx, %rax
.L4:
  ret
```

Fill in the missing expressions in C code

```
long x;
long y;
long z;

long test_xyz()
{
  long val = _______;
  if (_______){
     if(_______)
       val = ______;
     else
       val = ______;
  }else if(_______)
    val = _______;

  return val;
}
```

Fill in the missing expressions in C code

```
long x;
long y;
long z;

long test_xyz()
{
  long val = x + y - z;
  if (x < -3){
     if(y < z)
       val = x * y;
     else
       val = y * z;
  }else if(x > 2)
    val = x * z;

  return val;
}
```

Jump if carry

JC – Jump if carry (CF = 1)

This flag is used in **unsigned** operations and is activated when those operations generate a carry or borrow for the most significant bit

The JC instruction is activated if the ***carry flag*** bit inside RFLAGS is 1

```
.global addtest_carry

addtest_carry:
    ...
    addq %rax, %rcx

    # jump if carry
    jc carry_detected
    movq $0, %rax
    jmp end

carry_detected:
    movq $1, %rax

end:
    ret
```

Jump if overflow

JO – Jump if overflow (OF = 1)

This flag is used in **signed** operations and is activated when those operations generate values that are too large to fit in the destination register

The JO instruction is activated if the ***overflow flag*** bit inside RFLAGS is 1

```
.global addtest_overflow

addtest_overflow:
    ...
    movb $-127, %cl
    addb $-10, %cl

    # jump if overflow
    jo overflow_detected
    movq $0, %rax
    jmp end

overflow_detected:
    movq $1, %rax

end:
    ret
```

Detecting carry and overflow – C

In C we don’t have any specific approach to verify the flags mentioned before, so the only thing we can do is after an operation that can activate these flags we verify certain conditions

For example in an **unsigned** addition the result will only overflow if *sum* < x || *sum* < y
For the **signed** addition this will occur when x>=0 && y>=0 but *sum* <0 (positive overflow) or x <0 && y<0 but *sum* >=0 (negative underflow)

```
/* return 1 if arguments x and y can be added
   without causing overflow/underflow          */
int add_ok(int x, int y) {
   int sum = x+y;
   int neg_over = x < 0 && y < 0 && sum >= 0;
   int pos_over = x >= 0 && y >= 0 && sum < 0;

   return !neg_over && !pos_over;
 }
```

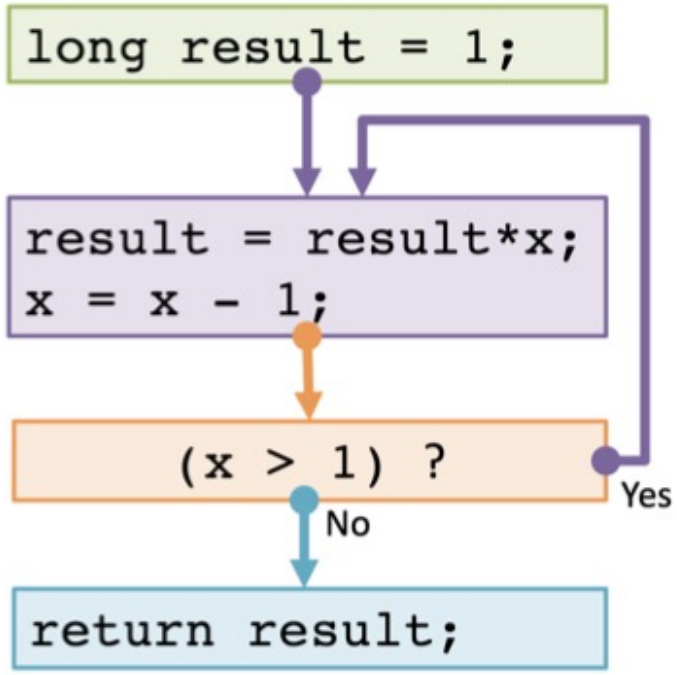
Loops in Assembly

C provides several looping constructs – namely, *do-while, while and for*

There is no corresponding instructions in machine code

So to emulate them we recur to the use of conditional tests and jumps

Do-while Loop



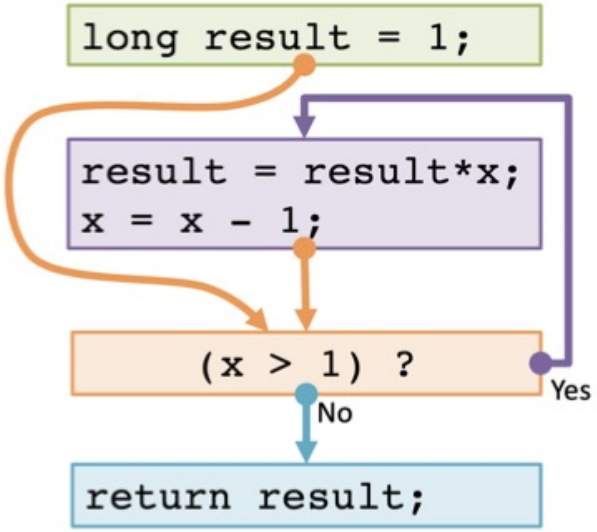
Consider the following C code

```
long x;

long fact_do_while()
{
  long result = 1;

  do{
    result = result * x;
    x = x - 1;
  } while(x > 1);

  return result;
}
```

Corresponding Assembly code

```
fact_do_while:
    movq   x(%rip), %rdi
    movq   $1, %rax

my_loop:
    imulq %rdi, %rax
    decq   %rdi

    cmpq   $1, %rdi
    jg     my_loop

ret
```

While Loop



Consider the following C code

```
long x;

long fact_while()
{
  long result = 1;

  while( x > 1 ){
    result = result * x;
    x = x - 1;
  }

  return result;
}
```

Corresponding Assembly code

```
fact_while:
    movq   x(%rip), %rdi
    movq   $1, %rax
    jmp    test_expression

my_loop:
    imulq %rdi, %rax
    decq   %rdi

test_expression:
    cmpq   $1, %rdi
    jg     my_loop

ret
```

Another possible pattern in Assembly code

```
fact_while:
    movq   x(%rip), %rdi
    movq   $1, %rax

my_loop:
    cmpq   $1, %rdi
    jle    end_my_loop

    imulq %rdi, %rax
    decq   %rdi

    jmp     my_loop

end_my_loop:
    ret
```

The condition changed from x > 1 to x <= 1

For Loop

Consider the following C code

```c
long x;

long fact_for()
{
  long result = 1;
  long i;

  for(i = 2; i <= x; i++) {
    result = result * i;
  }

  return result;
}
```

Can be translated to a while loop

```c
long x;

long fact_for_while()
{
  long result = 1;
  long i;

  i = 2;
  while(i <= x){
    result = result * i;
    i++;
  }

  return result;
}
```

Corresponding Assembly code

```
fact_for_while:
    movq  x(%rip), %rdi
    movq  $1, %rax
    movq  $2, %rdx

my_loop:
    cmpq  %rdi, %rdx
    jg    end_my_loop

    imulq %rdx, %rax
    incq  %rdx
    jmp my_loop

end_my_loop:
    ret
```

Loop Instructions

Loop instructions can be used in place of certain conditional jump instructions

Loop is a single instruction that functions the same as a DECQ RCX instruction followed by a JNZ instruction (Jump if not zero)

Notes:

The loop instruction test the flags but **do not change them**
The target label is encoded as a signed 8-bit offset, so only jumps offsets of -128 to +127 are allowed with these instructions

Loop Instructions

How to use:

In the first place enter the set of instruction to iterate and the load the number of iterations in the %rcx register

Then use the loop instruction at the end of that set

The loop will automatically decrement %rcx by one and jump to the label if %rcx is different from 0

`loop` **instruction example**

```
    ...
    movq $100, %rcx      # number of iterations
my_loop:
    ...                  # loop body
    ...
    loop my_loop
    ...
```

Loop Instructions variants

loop automatically decrements %rcx by one and jumps to the label if %rcx is different from 0

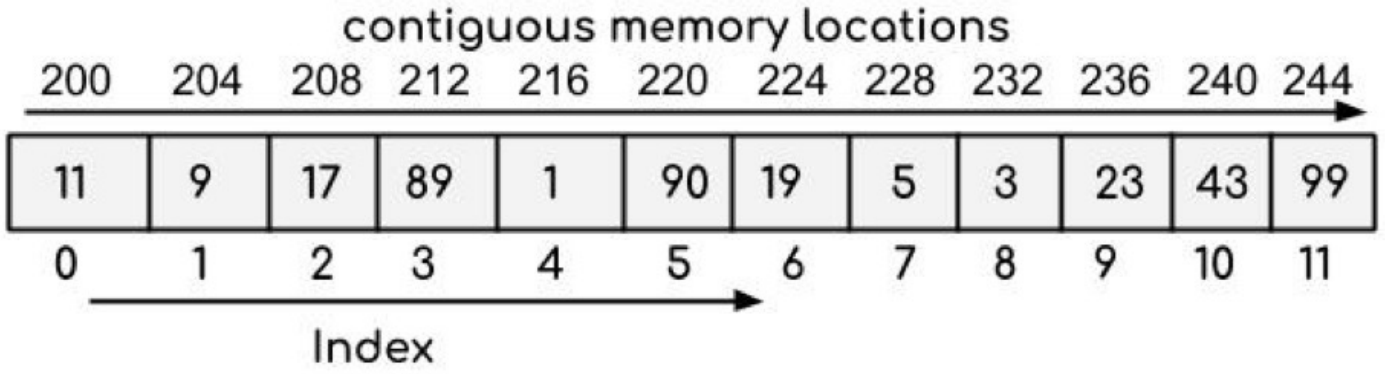
loope/loopz decrements %rcx by one and jumps to the label if %rcx is different from 0, and the flag ZF is active

loopne/loopnz decrements %rcx by one and jumps to the label if %rcx is different from 0, and the flag ZF is **not** active

ARQCP Flashcard
“Assembly: Array allocation and access”

Arrays

- Contiguously allocated memory region
- All *n* elements are of the same type
- Each element occupies the number of bytes determined by its data type
- Therefore, the size of the array is given by *n * sizeof(type)*



Arrays – Declaration Examples

Array declaration: Examples in C

```
char  array_a[12];
char  *array_b[8];
long  array_c[6];
long  *array_d[5];
int   array_e[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60};
```

Array	Element size (bytes)	Total size (bytes)
array_a	1	12
array_b	8	64
array_c	8	48
array_d	8	40
array_e	4	44

Arrays – Declaration Examples

Array declaration in C

```c
// uninitialized array
long array_c[6];

// initialized array
int array_e[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60};
```

Array declaration in Assembly

```
.section .bss          # section BSS (uninitialized variables)

.comm array_c, 48      # space for 6 long (6x8 bytes), name: array_c

.section .data         # section data (initialized variables)

array_e:               # name: array_e
    .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60 # array initialization
```

Arrays – Declaration Examples

String (array of char) declaration in C

```c
// uninitialized string (array of char)
char str_a[10];

// initialized string (array of char)
char str_b[] = "computer architecture";
```

String (array of char) declaration in Assembly

```
.section .bss          # section BSS (uninitialized variables)

.comm str_a, 10        # space for 10 bytes; variable name: str_a

.section .data         # section data (initialized variables)

str_b:                 # name: str_b
    .asciz "computer architecture"  # string initialization
```

Leaq - Address computation

leaq Src, Reg

- Computes the **effective address** of a memory location (first operand) and stores it in a general-purpose register (second operand)

- Instead of reading from the designated location, the *leaq* instruction copies the effective address to the destination (register)

- This instruction is used to generate pointers for later memory references

- It can be used to do arithmetic operations faster

- Since all addresses in x86-64 are 8 bytes this instruction only supports the suffix (q)

Leaq Example

Consider the following C code

```c
int vec[]={1,2,3,4,5};

int* get_vec_address()
{
  return vec;
}
```

Corresponding Assembly code

```
.section .data
  .global vec

.section .text
  .global get_vec_address

get_vec_address:
  leaq vec(%rip),%rax
  ret
```

Memory Addressing Modes

Indirect Addressing

- Occurs when a register is used as a pointer and it's placed between parenthesis

movX (register), destination

Notes:

- Byte addressing (addresses are manipulated byte by byte)
- To move the pointer to the next element, we must know the size of the element
- *Must* use the correct *mov* variant (movb, movw, movl, movq) according to the number of bytes that you want to copy
- We cannot do movX (%rbx), (%rax) since this are two memory addresses

Memory Addressing Example

Return the value of the 10th element

```
# copy the value of the 10th element of array_a to EAX
.section .data

array_a:
        .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

.section .text

.global tenth_element
tenth_element:
    leaq array_a(%rip), %rdi    # copy the address of array_a to %rdi
    addq $36, %rdi              # offset of 10th element ((10-1)*4=36)
    movl (%rdi),%eax            # copy the value pointed by %rdi to %eax
                                # this will be our return value
    ret
```

Return the value the 5th char

```
# copy the value of the 5th char of str to AL
.section .data

str:
        .asciz "computer architecture"

.section .text

.global fifth_char
fifth_char:
    leaq str(%rip), %rdi    # copy address of str to rdi
    addq $4, %rdi           # offset of fifth element
    movb (%rdi), %al        # copy the char value pointed by %rdi to %al
                            # this will be our return value
    ret
```

Memory Addressing Modes

Indirect Addressing with Offset

- When we add a constant value **(positive or negative)** to the address expression that indicates the offset relative to the base address in a register
  movX offset (register), destination

Example:

```
Movl 8 (%rdi), %eax          #%rdi is a pointer to an array of integers
```

Places in the eax register, the 4-byte value contained in memory, 8 bytes after the location pointed by the rdi register

Same as:

```
addq $8, %rdi                #add 8 bytes to rdi
movl (%rdi), eax
subq $8, %rdi                #value of rdi restored
```

Indexed memory mode

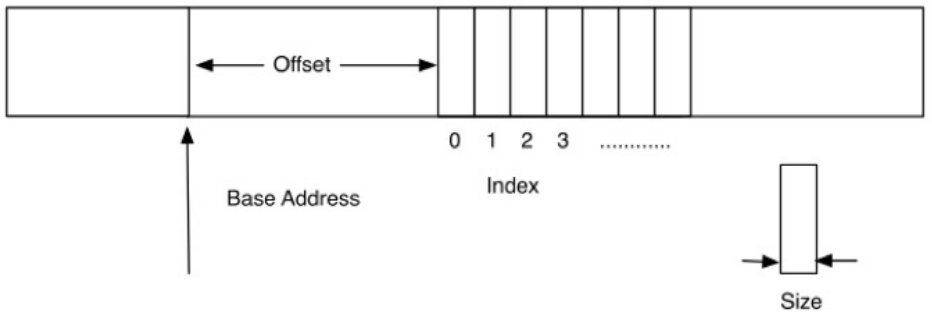
offset (base_address, index, size)

When accessing data in an array, once can use an index system to determine which value will be accessed. For this, we define:

- **Base address**: a pointer to the start of the memory we want to address **(register)**
- **Offset**: a displacement value between the beginning of the memory and when we start counting elements **(literal number)**
- **Size**: the size of each element **(literal number with value 1, 2, 4 or 8)**
- **Index**: the index of the element we want to access **(register)**

- The final address is calculated by the following expression:

base_address + offset + index *size



Indexed memory mode - Example

Indexed Memory Mode Example

```
.section .data

# declare integer array named 'array_a'
array_a:
    .int 10, 15, 20, 25, 30, 35

.section .text

example:
    # address of array_a in %rdi
    leaq array_a(%rip), %rdi

    # index to access: 3rd element
    movq $2, %rcx

    # copy array_a[2] to %eax
    movl (%rdi, %rcx, 4), %eax
    ret
```

Note:

If any of the values are zero, they can be omitted (but the commas are still required as placeholders)

If we omit the base_address, the offset can be an absolute address

Implementation examples

Expression	Type	Assembly
%*rax* ← array_e	`int *`	movq %rdi, %rax
%*eax* ← array_e[0]	`int`	movl (%rdi), %eax
%*eax* ← array_e[i]	`int`	movl (%rdi, %rcx, 4), %eax
%*rax* ← &array_e[i]	`int *`	leaq (%rdi, %rcx, 4), %rax
%*eax* ← *(array_e+i-3)	`int`	movl -12(%rdi, %rcx, 4), %eax

Increment by one all the elements of an array of ints

```
.section .data

    .global vec
vec:
    .int 1,2,3,4,5,6,7,8,9,10


.equ SIZE,10

.section .text

    .global inc_by_one
inc_by_one:
    leaq vec(%rip), %rdi     # store the vec address in %rdi
    movq $SIZE, %rcx         # store the number of elements of vec in %rcx
    cmpq $0, %rcx            # validate size of vec
    jle end

inc_loop:
    incl (%rdi)              # increment value pointed by %rdi
    addq $4, %rdi            # point to next int
    loop inc_loop
end:
    ret
```

Count the number of chars in a string

```
.section .data

str:
        .asciz "computer architecture 101"

.section .text

        .global str_count
str_count:
        leaq str(%rip), %rdi     # get address of string
        movl $0, %eax            # counter = 0
str_loop:
        movb (%rdi), %cl         # get char pointed by %rdi
        cmpb $0, %cl             # end of string?
        je end_str_loop
        incl %eax                # increase counter
        incq %rdi                # point to next char
        jmp str_loop
end_str_loop:
        ret
```

ARQCP Flashcard
“Assembly: Stack and Procedure calls”

Linux Memory Layout

Memory is viewed like an array of bytes

The range of virtual addresses that are available to a program is called virtual address space

Heap and stack **expand** and **contract** dynamically at run time

Unlike the code written and data areas, that have fixed size

Stack

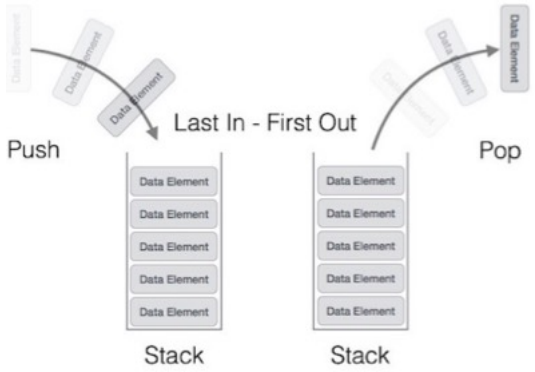
The stack is a FIFO structure (First in/First out)

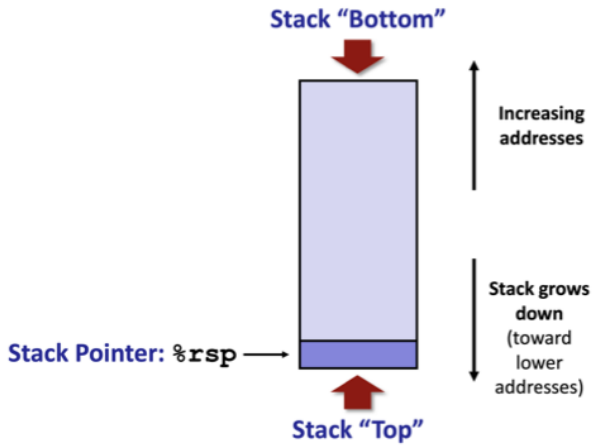
The new elements are added into the top of the stack

The existing elements can only be removed from the top of the stack

It grows toward lower addresses

%rsp register indicates the top of the stack (the lowest address/position)





Uses of Stack

Support procedure calls and return

Temporary data storage

Store local variables

Store parameters of a procedure about to call

Note:
Stack is allocated in frames

Stack – PUSH & POP Instructions

In order to access values in the stack we have 2 commands (PUSH & POP)

Usage: *pushq source*

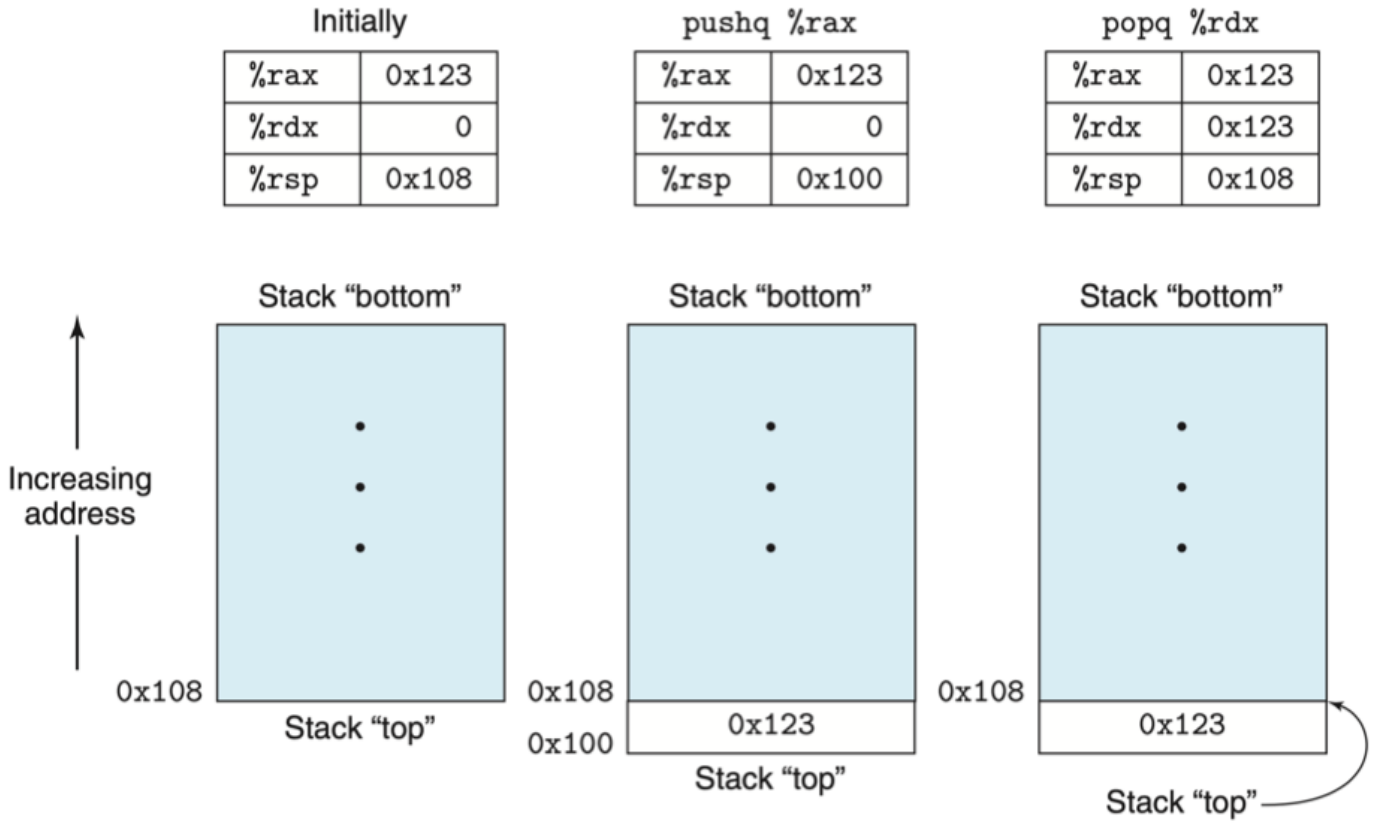
Decrements %rsp by 8 bytes
Writes *source* at address given by %rsp

Usage: *popq destination*

Increments %rsp by 8 bytes
Reads the value at address given by %rsp and writes it to destination

This instructions only accept 16|64 bit operations
Only w and q variants can be used
%rsp is only updated by 2 or 8 bytes

Stack – PUSH & POP Instructions | Example



Stack – PUSH & POP Instructions | Example

PUSH/POP Examples

```
.section .data
x:
        .int -125

.section .text
.global test_stack
test_stack:
    movl  $24420, %ecx
    movw  $350, %dx
    pushq %rcx
    pushw %dx
    pushq x(%rip)              # sets the 4 most significant bytes to 0
    leaq  x(%rip), %rdi
    pushq %rdi
    popq  %rsi
    popq  %rax                 # the value of x is in %eax
    popw  %dx
    popq  %rcx
    ret
```

Procedures in Assembly

Identified by a label

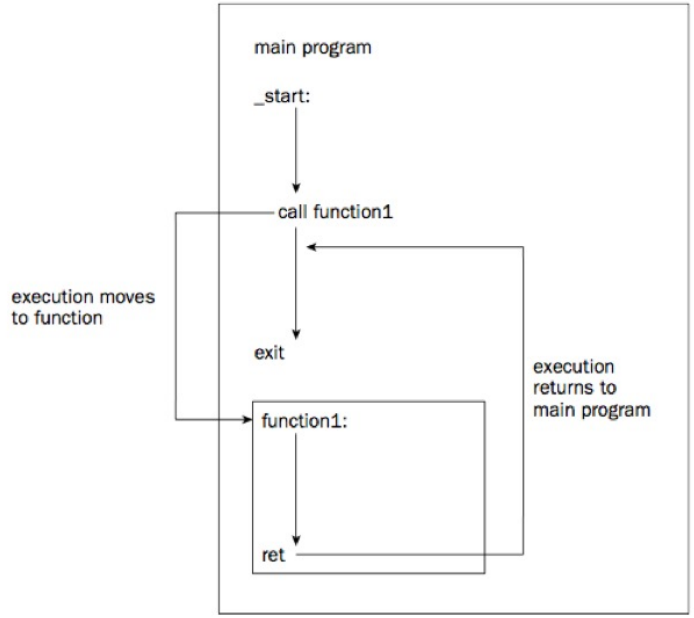
Invoked through the ***call*** instruction

End with the ***ret*** instruction

The return value must be stored in the %rax register before the execution of the ***ret*** instruction

Procedure Call Example

```
procedure2:

        ...

        ret


procedure1:

        ...

        call procedure2

        ...
```



Transfer of Control

The *call* instruction has only parameter, the label from the called procedure that is converted into the corresponding memory address

The *ret* instruction allows the execution flow to continue in the instruction immediately after the procedure call

ret doesn’t have any argument

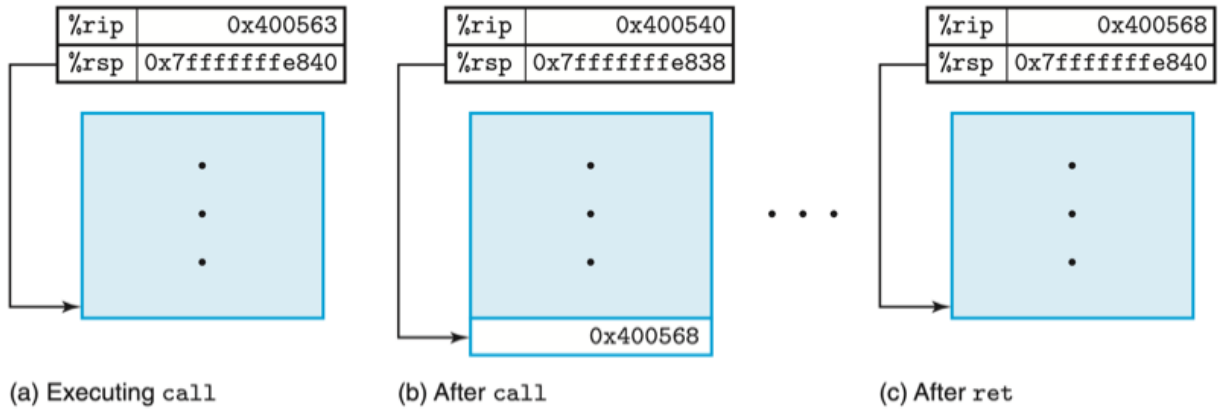
Stack & execution flow

Procedure call :
call *label*

Push return address on stack (address of the instruction following the call)
Jumps *(jmp)* to label (sets %rip to the corresponding address)

Procedure return:
ret

Pop address from stack
Jumps (*jmp)* to address



Note:
The programmer as to make sure that before the return of the
called function the value pointed by %rsp
is the the address previously pushed by *call*

Saving and restoring register state

When an x86-64 procedure requires storage beyond what it can hold in registers, it allocates space on the stack

Whenever a procedure calls another, it is possible that the called uses registers that are being used in the callee and rewrite information, so we need to make sure that information isn't lost between calls

Save and Restore a Register

```
...
pushq %rdx        # save current value of %rdx

# divide %edx:%eax by %ecx
movl $0, %edx     # dividend
movl $1000, %eax  # dividend
movl $500, %ecx   # divisor
divl %ecx         # unsigned division
...               # remainder on %edx

popq %rdx         # restore old value %rdx
...
```

```
p1:
    ...
    movq $10, %rcx
    ...
    call p2           # can overwrite %rcx
    cmpq %rax, %rcx   # there is no guarantee on the current content of %ecx
    ...
```

Register saving conventions

There is a convention that determines which registers should be saved prior and restored after a call

“Caller Save”

Caller saves the current contents of a set of registers prior to call
After the procedure return, it restores those registers to their old values

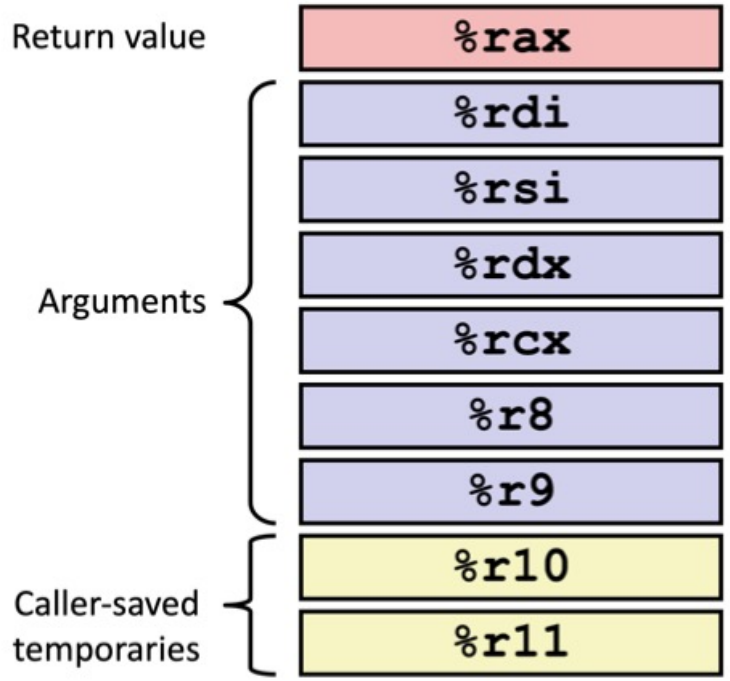
“Callee Save”

Calle saves the current contents of another set of registers before using them
Before returning, it restores those registers to their old value

Caller and Callee registers

Caller is responsible for their management

If it uses them after the call
Therefore, they can be modified by the **callee**



Callee must preserve their value

By either not changing it
By pushing the original value on the stack, altering it, and then popping the old value before returning

Caller and Callee registers - Example

Function f1

```
long f1(){
        return a + b + f2();
}
```

Function f2

```
long f2(){
        return c + 1;
}
```

Function f1

```
f1:
  # f1 is called by other functions
  # (e.g., main in C)
  # save %rbx on stack
  pushq %rbx

  movq a(%rip), %rdx
  movq b(%rip), %rbx

  # caller is responsible for %rdx
  # save only those that are used
  # after the call
  # saves %rdx on stack
  pushq %rdx
  call f2
  # restore %rdx
  popq %rdx

  addq %rdx, %rbx
  addq %rbx, %rax

  # restore %rbx before returning
  popq %rbx
  ret
```

Function f2

```
f2:
  # callee is responsible for %rbx
  # save only those that are used
  # save %rbx on stack
  pushq %rbx

  movq c(%rip), %rbx
  incq %rbx
  movq %rbx, %rax

  # restore %rbx before returning
  popq %rbx
  ret
```

ARQCP Flashcard
“Assembly: Stack and Functions – Parameters and Local Variables”

Passing parameters via register

Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Passing parameters via register - Example

Function f1

```
long f1(long a, long b, long c){
  return a + b + f2(c);
}
```

Function f2

```
long f2(long c){
  return c + 1;
}
```

Function f1

```
f1:
  # a in %rdi, b in %rsi, c in %rdx
  # a + b
  addq %rsi, %rdi
  # store %rdi on stack
  pushq %rdi
  # prepare function call f2(c)
  movq %rdx, %rdi
  call f2
  # restore %rdi
  popq %rdi
  # a + b + f2(c)
  addq %rdi, %rax
  ret
```

Function f2

```
f2:
  # c in %rdi
  # c + 1
  incq %rdi
  movq %rdi, %rax
  ret
```

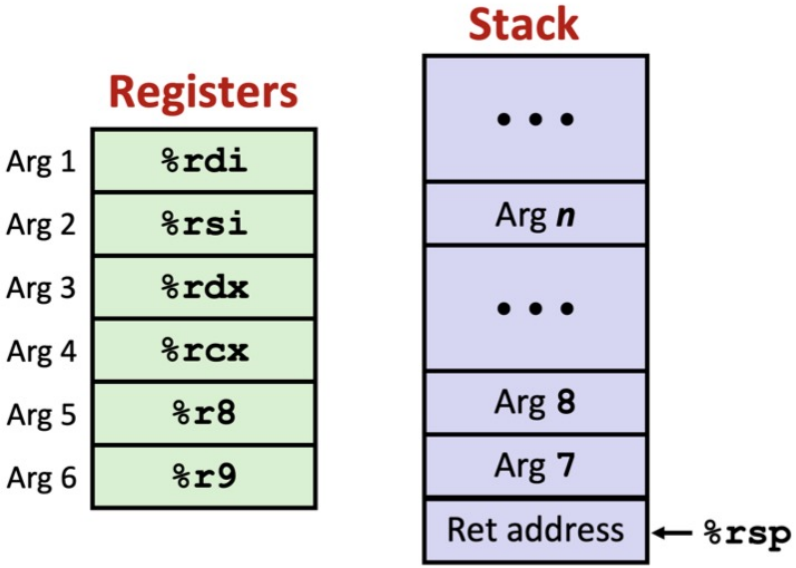
Passing parameters via stack

If we assume that a procedure *P* calls procedure *Q* with *n* arguments, such that *n* > 6

The arguments 1 to 6 will be in the default registers for this operation, the 7 through *n* arguments are putted onto the stack by inverse order (*n* to 7)

When passing parameters on the stack, all data sizes are rounded up to be multiples of eight

Note that after the call the first value saved in the stack is the return address to the caller



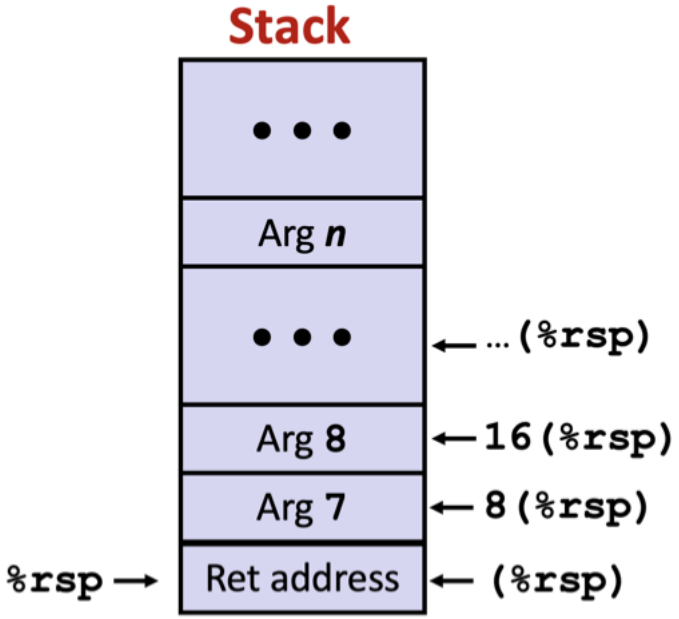
Accessing parameters on the stack

After a *call* %rsp pointer will point to the top of the stack, where the return address is located

Making all the arguments form 7 to *n* “underneath” that return address

In order to access those values we can just pop the values but that could lead to errors since we can lose the return address

So in order to avoid that, we will access them with the help of offset from %rsp
Ex. movl 8(%rsp), %eax



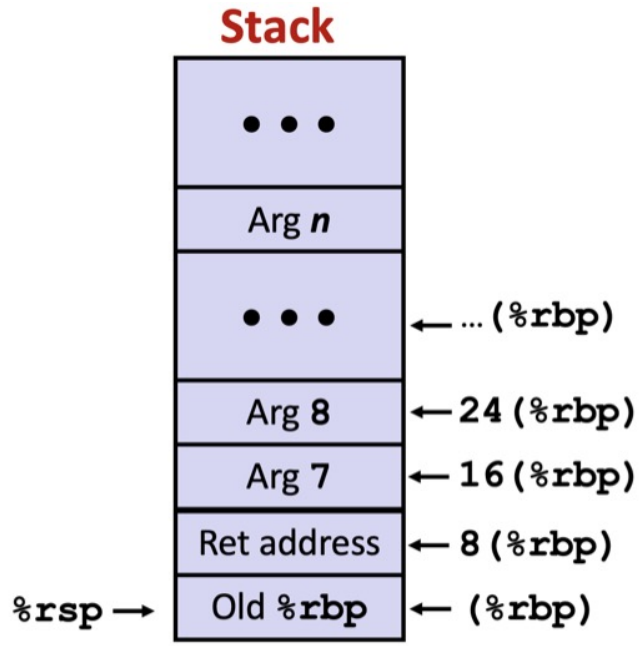
Accessing parameters on the stack – Use %rbp

The last slide explains an approach to access the arguments from 7 to *n*, but there is a problem associated with that

If we push any data to the stack while the callee code is being executed, the value of %rsp is altered and all the data being accessed using last's slide theory will be wrong

To avoid this, we will copy the %rsp register value to the %rbp register, with this we are making sure that all the information remains there and in the case of new data being added to the stack the use of offset to access the arguments is always correct

In order to not corrupt %rbp value we will do the same logic as a callee register, it will be pushed to the stack before assigning the %rsp register value and before the *ret* instruction it will be popped from the stack



Prologue and Epilogue

The Stack is allocated in **frames**

A **frame** is a section of stack used by one procedure call to store context while running, delimited by %rbp and %rsp

Procedure code manages stack frames

- Prologue/Setup : allocate space at start of procedure
- Epilogue/Clean-up: deallocate space before return

Prologue and Epilogue

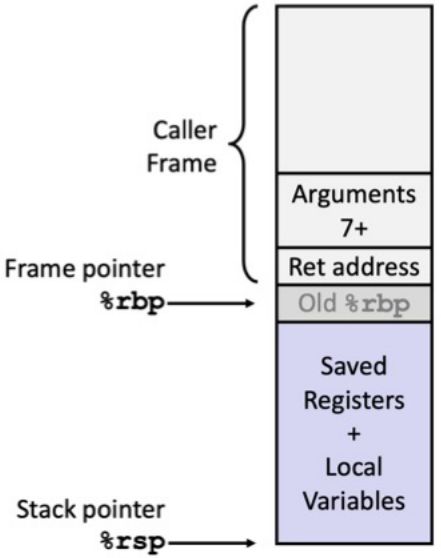
The prologue saves the **caller's** value for **%rbp** into the **callee's stack**

After saving %rbp value, the %rsp value is passed to %rbp, making them the same as the callee's program start running

With this we will have the space necessary to allocate memory to local variables from callee and it's saved registers

Before returning the value of %rbp is popped from the stack and restored, and making the top of the stack the returning address

```
function:
   # prologue
   pushq %rbp        # save the original value of RBP
   movq  %rsp, %rbp  # copy the current stack pointer to RBP

   # function body
   ...

   #epilogue
   movq  %rbp, %rsp  # retrieve the original RSP value
   popq  %rbp        # restore the original RBP value
   ret
```



Cleaning out the stack

As stated before, in order to call a function that as 7+ arguments, all of those must be inserted into the stack

So its logical that after the return we must clean the stack since the callee accesses the values but without popping them from the stack

This can be done by just popping all the arguments, or a faster and more readable way is to simply increment the %rsp by (8 or 2) x number arguments

Cleaning the Stack after Return

```
function:
  ...

  pushq %rax      # push 8th parameter onto stack
  pushq %rbx      # push 7th parameter onto stack
  call utilfunc
  addq $16, %rsp # get the stack back to where it was before the function call

  ...
  ret
```

Practical Example (Call with arguments)

Function f1

```
long f1(long a, long b, long c,
        long d, long e, long f){
  return a + f2(a*b,b,c,d,e,f,10,-20);
}
```

Function f2

```
long f2(long a, long b, long c,
        long d, long e, long f,
        long g, long h){
  /* do someting with parameters */
  ...
  return h/g;
}
```

Passing Parameters

```
#long f1(a,b,c,d,e,f)
f1:
  # a in %rdi, b in %rsi, c in %rdx
  # d in %rcx, e in %r8, f in %r9

  pushq %rdi                # save %rdi on stack

  imulq %rsi, %rdi          # a * b
                            # prepare function call
  pushq $-20                # 8th parameter
  pushq $10                 # 7th paramerer
  call f2                   # f2(a*b,b,c,d,e,f,10,-20);
  addq $16, %rsp            # clear 7th and 8th parameters

  popq %rdi                 # restore %rdi
  addq %rdi, %rax           # a + f2(...)
  ret
```

Accessing Parameters

```
#long f2(a,b,c,d,e,f,g,h)
f2:
  pushq %rbp                #prologue
  movq %rsp, %rbp

  # a in %rdi, b in %rsi, c in %rdx
  # d in %rcx, e in %r8, f in %r9
  # g in 16(%rbp), h in 24(%rbp)

  pushq %rbx                # save %rbx

  # do something with parameters
  ...
  movq 16(%rbp), %rbx       # get g from stack
  movq 24(%rbp), %rax       # get h from stack
  cqto                      # h/g
  idivq %rbx

  popq %rbx                 # restore %rbx

  movq %rbp, %rsp           # epilogue
  popq %rbp
  ret
```

Local storage on the stack

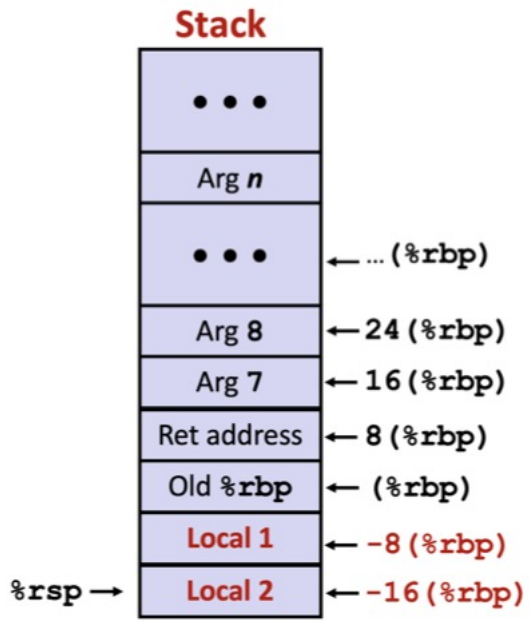
In order to allocate storage for local variables when entering a function and release it before returning, we will decrease the value of %rsp

This allocation must be done after setting %rbp value to the top of the stack (prologue)

Doing this we will can access the local variables using %rbp

Reserving Space for Local Variable

```
function:
    # prologue
    pushq %rbp
    movq  %rsp, %rbp
    subq  $16, %rsp    # reserve 16 bytes for local variables
                       # this space can be used, for e.g., as either
                       # four 32-bit values or two 64-bit values
    ...
```



Local storage on the stack

After doing all of the last's slide "instructions" we will be capable of access the local variables that will only be available in the callee (thus the term "local variables")

When the epilogue is reached and the %rsp register set to the original value, all the local variables will be lost from the stack

Function f1

```
long f1(){
  long arg1 = 100;
  long arg2 = 200;
  long sum = swap_add(&arg1,&arg2);
  long diff = arg1 - arg2;
  return sum * diff;
}
```

Function f1

```
#long f1()
f1:
  pushq %rbp                # prologue
  movq  %rsp, %rbp
  subq $16, %rsp            # allocate 16 bytes for local variables

  movq $100, -8(%rbp)       # arg1 = 100;
  movq $200, -16(%rbp)      # arg2 = 200;

                            # prepare function call
  leaq -8(%rbp), %rdi       # compute &arg1 as first argument
  leaq -16(%rbp), %rsi      # compute &arg2 as second argument
  call swap_add             # call swap_add(&arg1,&arg2)

  movq -8(%rbp), %rdx       # get arg1
  subq -16(%rbp), %rdx      # diff = arg1 - arg2
  imulq %rdx                # sum * diff

  movq  %rbp, %rsp          # epilogue (deallocates 16 bytes)
  popq  %rbp
  ret
```

ARQCP Flashcard
“Assembly: Bit-level Operations”

Bit-level Operations

Bit-level operations modify one or more bits at a time and are used to manipulate binary numbers

This operations are very efficient because they are directly supported by the processor

There are **two** major groups of bit-level operations:

Boolean Logic

Each bit is compared individually using the logic function specified
Logic functions: AND, OR, XOR, NOT

Bit Movements:

Shift bits left/right(multiply/divide by powers of two)
Rotate bits left or right

AND Logic Operator

Usage: AND *origin, destination*

Operation: *destination* = *destination* **AND** *origin* (the result is placed in *destination*)

Origin can be a memory address, a constant value or a register

Destination can be a memory address or a register

The AND operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

OR Logic Operator

Usage: OR *origin, destination*

Operation: *destination* = *destination* **OR** *origin* (the result is placed in *destination*)

Origin can be a memory address, a constant value or a register

Destination can be a memory address or a register

The OR operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

XOR Logic Operator

Usage: XOR *origin, destination*

Operation: *destination* = *destination* ***X*OR** *origin* (the result is placed in *destination*)

Origin can be a memory address, a constant value or a register

Destination can be a memory address or a register

The XOR operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

NOT Logic Operator

Usage: NOT *destination*

Operation: *destination* = **NOT** *destination* (the result is placed in *destination*)

Destination can be a memory address or a register

The NOT operation can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

NEG Logic Operator

Usage: NEG *destination*

Operation: *destination* = **NEG** *destination* (the result is placed in *destination*)

Destination can be a memory address or a register

The NEG operation can be used in numbers 32(l) bits

Note:
”!” considers the entire number as a logical value

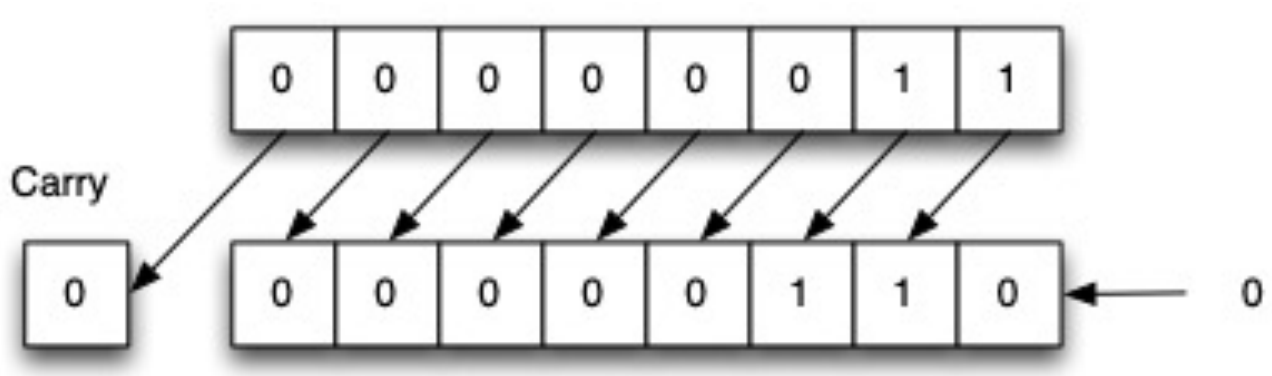
y	NOT y	NEG y	!y
0	-1	0	1
-1	0	1	0
1	-2	-1	0

Shift Left

There are two operations to shift left n bytes

SHL/SAL are equivalent operations, they do the same thing because shifting left don't mean losing signal in signed numbers

When the processor executes an shift left operation, zeros enter on the right and the last bit exists left to the carry flag



Shift Left

SHL/SAL

SHL *destination* (or SAL *destination*)

Shifts the *destination* value left one position (equivalent to *destination* *= 2)

SHL %cl, *destination* (or SAL %cl, *destination*)

Shifts the *destination* value left by the number of times specified in the CL register (equivalent to *destination* *= 2^CL)

SHL $n, *destination* (or SAL $n, *destination*)

Shifts the *destination* value left by the number of times specified in n (equivalent to *destination* *= 2^n)

For all the formats the *destination* ***can be a memory address or a register***

The SHL/SAL operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

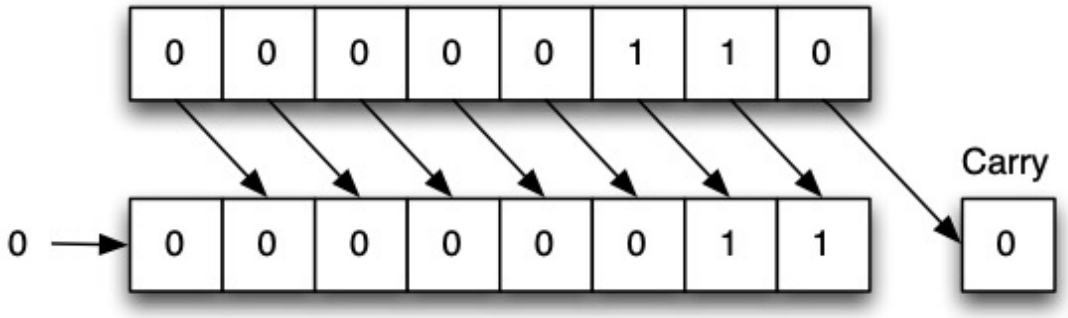
Shift Right

There are two operations to shift left n bytes

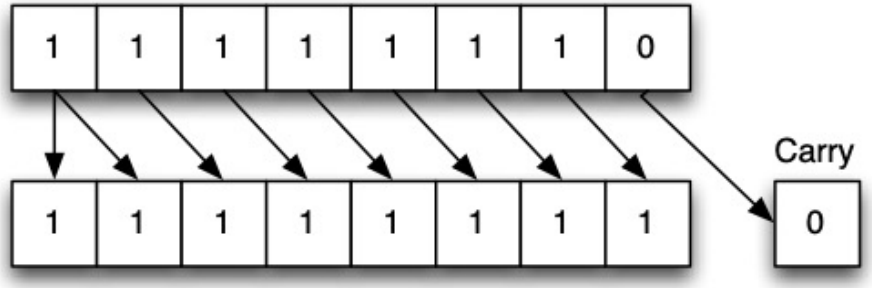
SHR (Logic shift, **Applied in c with unsigned**)
This type of right shift will fit the bits entering on the left side with zeros, and passing the last bit on right to the carry

SAR (Arithmetic shift, **Applied in c with signed**)
This type of right shift will fit the bits entering on the left side **with the preservation of the signal,** this will be used in **signed numbers**

SHR



SAR



Shift Right

SHR/SAR

SHR *destination* (or SAR *destination*)

Shifts the *destination* value right one position (equivalent to *destination /*= 2)

SHR %cl, *destination* (or SAR %cl, *destination*)

Shifts the *destination* value right by the number of times specified in the CL register (equivalent to *destination /*= 2^CL)

SHR $n, *destination* (or SAR $n, *destination*)

Shifts the *destination* value right by the number of times specified in n (equivalent to *destination /*= 2^n)

For all the formats the *destination* ***can be a memory address or a register***

The SHR/SAR operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

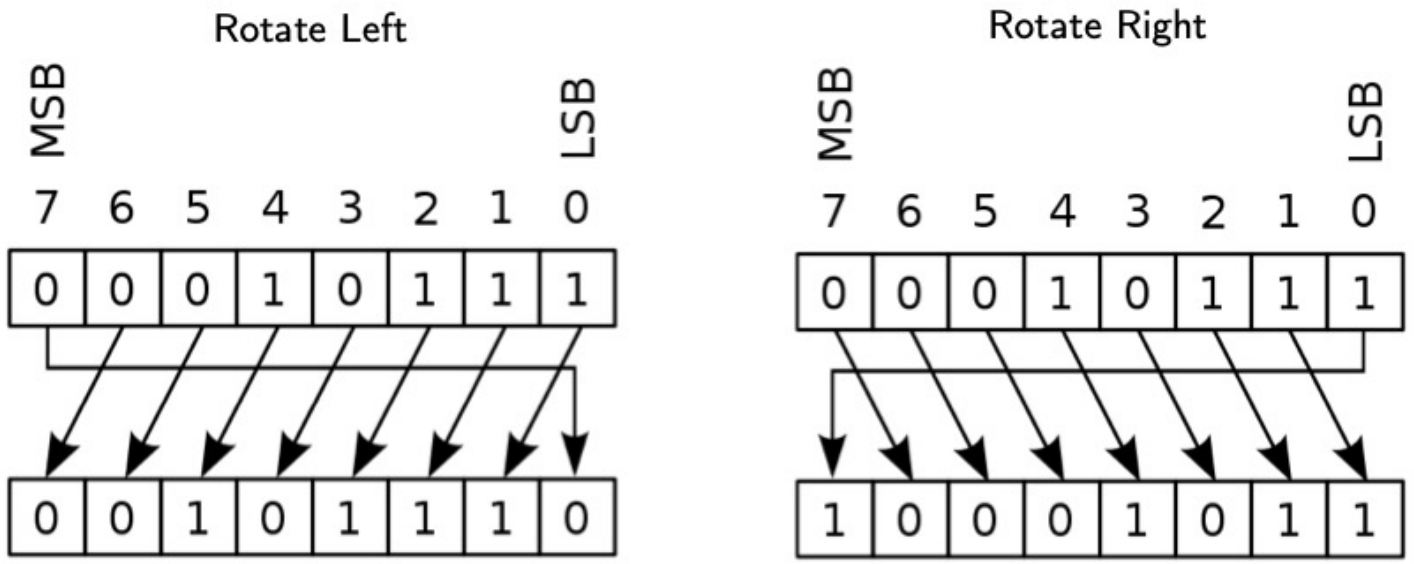
Rotating Bits

ROL/ROR

ROL – Bit rotation to the left

ROR – Bit rotation to the right

Perform just like the shift instructions, except the overflow bits are pushed back into the other end of the value instead of being dropped



Rotating Bits

ROL/ROR

ROL/ROR *destination*

ROL/ROR %cl, *destination*

ROL/ROR $n, *destination*

For all the formats the *destination* ***can be a memory address or a register***

The ROL/ROR operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

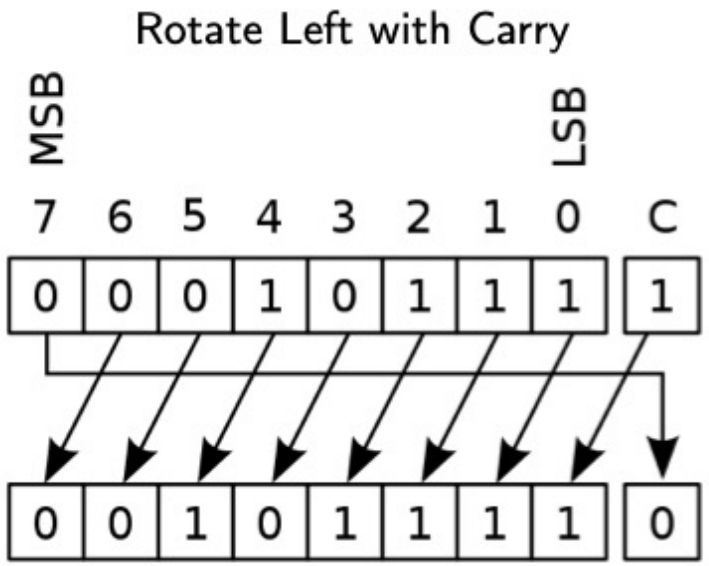
Rotating Bits

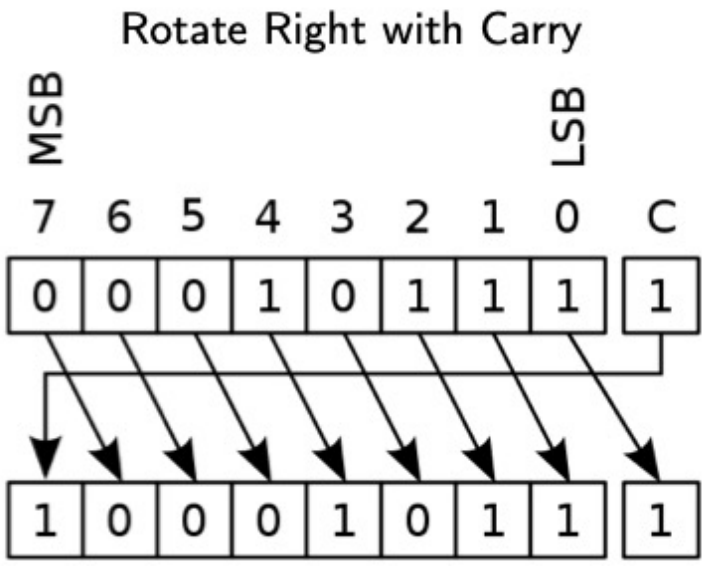
RCL/RCR

RCL – Bit rotation to the left with carry

RCR – Bit rotation to the right with carry

Perform bit rotation, but the first entering bit comes from carry and overflow bits go to carry





Rotating Bits

RCL/RCR

RCL/RCR *destination*

RCL/RCR %cl, *destination*

RCL/RCR $n, *destination*

For all the formats the *destination* ***can be a memory address or a register***

The RCL/RCR operations can be used in numbers of 8(b), 16(w), 32(l), 64(q) bits

Bit Masks

One common use of bit-level operations is to implement *masking* operations

We use this *masking* operations to obtain certain bits from a number

A *mask* is a bit pattern that indicates a selected set of bits within a number

Example:

If we have the value 00101100 in %al and we want the least 4 bits, we can apply a 00001111 bits mask

```
movb $0b00101100, %al
movb $0b00001111, %ah
andb %ah, %al          # %al = 0b00001100
```

Bit Masks

Assume now that we want to replace the 4 least significant bits of %al with the 4 least significant bits of %cl

First we need to set the 4 least significant bits of %al to 0
This is done by applying a 11110000 mask with AND operator

Then we need to set the 4 most significant bits of %cl to 0
This is done by reversing the mask turning it into 00001111 and apply it with the AND operator

Then apply the AND operator with %al as destination and %cl as origin

```
movb $0b00101100, %al
movb $0b01000011, %cl

movb $0b11110000, %ah
andb %ah, %al           # %al = 0b00100000
notb %ah                # %ah = 0b00001111 (inverts the mask)
andb %ah, %cl           # %cl = 0b00000011

orb  %cl, %al           # %al = 0b00100011
```

Bit Masks

We can also use a *mask* with and XOR operator in order to invert the bits of a number

Or even use XOR operator to compare 2 numbers

```
xorl %ebx, %ecx
jz is_equal
```

```
movb  $0b00101100, %al
movb  $0b00001111, %ah
xorb  %ah, %al            # %al = 0b00100011
```

Bit Masks

Like the XOR operator, others operators can be used with *masks* in order to manipulate bits

The **AND** operator allows to:

- **Set bits to zero** – use a *mask* with those bits set to zero
- **Get value of few bits** – use a *mask* with those bits set to one
- **"Round" a number to a power of 2** – use a *mask* with x least significant bits to zero
- **Get the remainder of a division** – use a *mask* with x least significant bits to one

The **OR** operator allows to:

- **Set bits to one** – use a *mask* with those bits set to one
- **Join the bits of two numbers** – perform an **OR** between the two numbers

ARQCP Flashcard
“Heterogeneous Data Structures”

Heterogenous data structures in C

There are 2 ways of combining variables of different types in C:

Structures, that are declared using the keyword *struct* and can aggregate multiple variables under the same name

Unions, that are declared using the keyword *union* and allow a single variable to use several data types

(The next slides will focus only in structures)

Structures

A structure is a continuous memory region (similar to arrays), composed of different members
This members can be accessed by their name and can have different types

There are 3 different ways of having structures in C:

Defining a single structure in a variable (slide 125)
Declaring a structure type (slide 126)
Declaring a new data type (slide 127)

Structures in a variable

The structure has only one instantiation (c1); no other instantiations of the structure exist

Structure in C - Defining a structure held by a variable

```c
#include <stdio.h>

/* define a structure in a global variable c1 */
struct  {
  int   n_wheels;
  int   n_passengers;
  float fuel_consumption;
} c1;

int main(){
  /* init and access the data in c1 */
  c1.n_wheels=4;
  c1.n_passengers=2;
  c1.fuel_consumption=12.5;
  printf("The car has %d wheels, %d passengers.Consumption is %f l/100 km.\n",
         c1.n_wheels,c1.n_passengers,c1.fuel_consumption);
  return 0;
}
```

Structures in a structure type

The structures are instantiated by creating new variables using the name of the structure after the keyword *struct*

Structure in C - Declaring a structure type

```c
#include <stdio.h>

/* declare a structure type called s_car */
struct s_car {
  int   n_wheels;
  int   n_passengers;
  float fuel_consumption;
} ;

int main(){
  /* define two structures of type s_car */
  struct s_car c1, c2;

  /* init and acess the data in c1 and c2 */
  c1.n_wheels=4; c1.n_passengers=2; c1.fuel_consumption=12.5;
  c2.n_wheels=5; c2.n_passengers=4; c2.fuel_consumption=20.0;
  printf("Car 2 has %d wheels, %d passengers. Consumption is %f l/100 km.\n",
         c2.n_wheels,c2.n_passengers,c2.fuel_consumption);
  return 0;
}
```

Structures in a new data type

The structures are instantiated by creating new variables using the name of the new data type

Structure in C - Declaring a data type for a structure

```c
#include <stdio.h>

/* declare a data type called t_car */
typedef struct { /* we could write 'typedef struct s_car' */
  int   n_wheels;
  int   n_passengers;
  float fuel_consumption;
} t_car; /* this is the name of the structure data type */

int main(){
  /* define two cars */
  t_car c1, c2;

  /* init and access the data in c1 and c2 */
  c1.n_wheels=4; c1.n_passengers=2; c1.fuel_consumption=12.5;
  c2.n_wheels=5; c2.n_passengers=4; c2.fuel_consumption=20.0;
  printf("Car 2 has %d wheels, %d passengers. Consumption is %f 1/100 km.\n",
         c2.n_wheels,c2.n_passengers,c2.fuel_consumption);
  return 0;
}
```

Pointers to structures in C

In C is possible to define pointers to structures

struct s_car *ptr //pointer to structure type
t_car *ptr // pointer to structure data type

To access the fields of a structure referenced by a pointer, use ”->”

ptr -> n_wheels
ptr -> n_passengers

Instead of:

(*ptr).n_wheels
(*ptr).n_passengers

NOTE:
If we want the address of a structure, use the “&” operator
Structures are very similar to arrays **except** in this aspect

Pointers to structures in C – Example

Structure in C - Obtaining a pointer to a structure

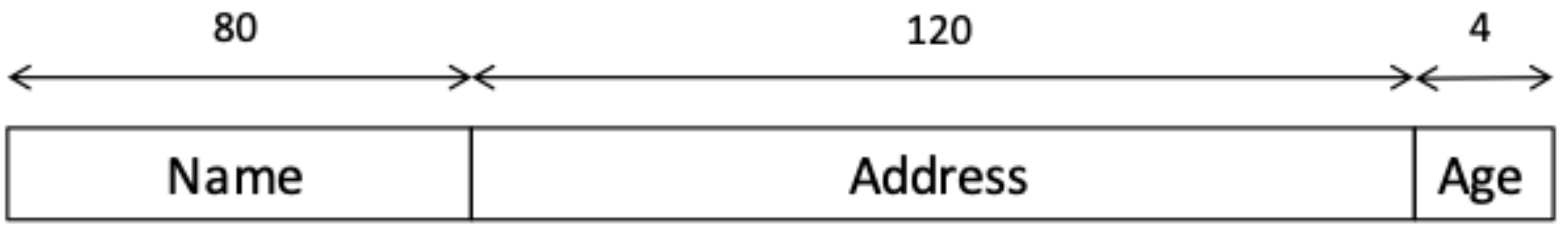
```c
#include <stdio.h>

typedef struct {
   int   n_wheels;
   int   n_passengers;
   float fuel_consumption;
} t_car;

int main(){
   /* define a car */
   t_car c1;
   /* define a pointer to a car */
   t_car *car_ptr = &c1;

   /* init and access the data in c1 using the pointer */
   car_ptr->n_wheels=4;
   car_ptr->n_passengers=2;
   car_ptr->fuel_consumption=12.5;
   printf("The car has %d wheels, %d passengers. Consum. is %f l/100 km.\n",
           car_ptr->n_wheels,car_ptr->n_passengers,car_ptr->fuel_consumption);

   return 0;
}
```

Storing structures in memory

Structure in C:

```
struct s_person_data{ /* we could also have used typedef ... */
   char name[80];     /* 80 bytes  */
   char address[120]; /* 120 bytes */
   int  age;          /* 4 bytes   */
} ;
```

This struct will occupy a total of 204 bytes (80 chars for variable name, 120 bytes for variable address and 4 bytes to the age variable)



Structures in assembly

In assembly we will access the variables of a structure using a proper *offset*

The offset will refer to the number of bytes that exist in a certain structure before the desired variable is present

To have an easier access to this offsets we will define different constants with the desired offsets

```
.equ DATA_SIZE, 204          #total size of the struct
.equ NAME_OFFSET, 0          #name variable starts at the byte 0
.equ ADDRESS_OFFSET, 80      #address variable starts at the byte 80
.equ AGE_OFFSET, 200         #age variable starts at the byte 200
```

Structures in assembly – Examples

Function to set age:

Function set_age in C

```
void set_age(struct
  s_person_data *pdata){
  /* age = 30 */
  pdata->age = 30;
}
```

Function set_age in Assembly

```
set_age:
  # *pdata on %rdi
  # age = 30
  movl $30, AGE_OFFSET(%rdi)
  ret
```

Access to a structure variable and set a new value for it

Structure Declaration

```
struct rec{
  int x;
  int a[3];
  int *p;
};
```

Access in C

```
void set_x(struct rec *r,
           int val){
  /* access by name */
  r->x=val;
}
```

Access in Assembly

```
# *r in %rdi
# val in %esi

# access by offset
movl %esi,(%rdi)
```

Note:

The address of x (&x) and the address of rec (&rec) are the same !

Structures in assembly – Examples

Obtaining the address of a structure member:

Structure Declaration

```
struct rec{
 int x;
 int a[3];
 int *p;
}
```

Access in C

```
int* find_a(struct rec *r,
            unsigned int i){
  /* address of a[i] */
  return &(r->a[i]);
}
```

Access in Assembly

```
# *r in %rdi
# i in %esi

# %rax = r + 4 + (4 * i)
leaq 4(%rdi,%rsi,4),%rax
```

In this case the value wanted is an element from the array *a[3]*

To access it it is used indexed memory mode combined with the *leaq* command and an offset of 4 bytes

The offset must be of 4 bytes since the variable a starts at the 4^{th} byte of this structure

Structures in assembly – Examples

Obtaining the address of a structure member and then save the address in a pointer:

Structure Declaration

```
struct rec{
 int x;
 int a[3];
 int *p;
}
```

Access in C

```
void set_p(struct rec *r){
  /* address of a[i],
  where i is the value
  of member x */
  r->p = &(r->a[r->x]);
}
```

Access in Assembly

```
# *r in %rdi
# %rcx = r->x
movslq (%rdi),%rcx
# %rax = r + 4 + 4*(r->x)
leaq 4(%rdi,%rcx,4),%rax
# r->p = %rax
movq %rax,16(%rdi)
```

In this case it is applied the same logic seen in the last slide but this time the position (*i*) inside of the array will be the *x* variable

After having the address it is simply addressed to a pointer with a *movq* instruction

Data alignment

The memory is accessed in blocks of fixed size, **usually blocks of 4 or 8 bytes** (depending of the system used)

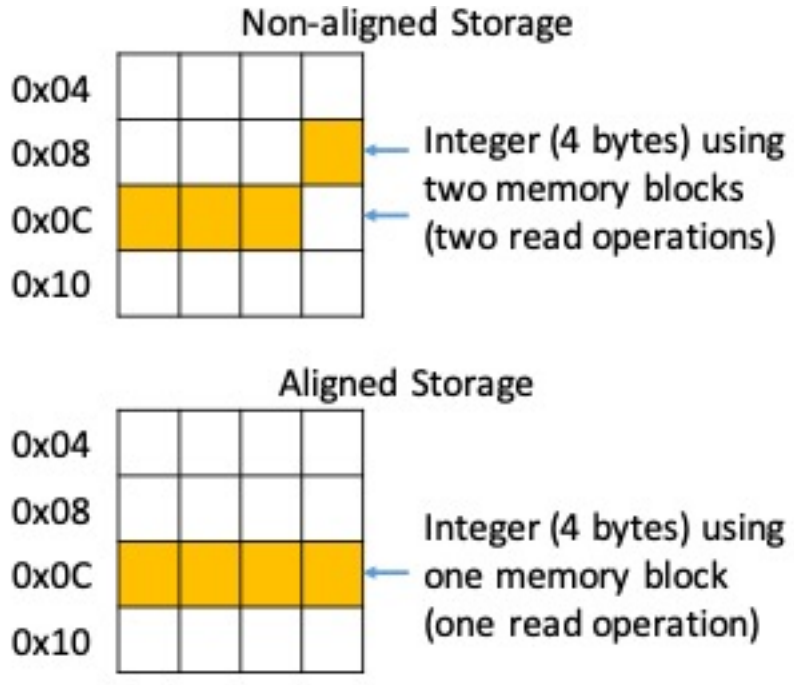
Storing data at addresses multiple of the block size used is a good way to optimize the process of accessing the memory blocks, since it will not be necessary to access different blocks of memory in the same variable

Example:

If the memory is not aligned an integer could be divided between 2 blocks of memory like seen in the top example

This will cost some processing time

So, is a good practice to allocate the memory in addresses that are multiple of the block size



Data alignment in x86-64

The process of having the data aligned is performed by default by the compiler

In case of memory not being correctly aligned to the block size, the complier will insert “blank spaces” between data stored in memory to ensure that all the memory is aligned

Doing this, the compiler will contribute to a faster performance of the code

Note:

This is just recommended in the x86-64 architecture but it can be required by others

In this case, having not aligned memory will only cost a performance penalty

Data alignment in x86-64 - Rules

The alignment rules are based on the principle that any primitive object of **K** bytes must have an address that is multiple of **K**

K = 1 byte (char)

No restrictions

K = 2 bytes (short)

The least significant bit must be 0 (with this we guarantee that the address is a multiple of 2)

K = 4 bytes (int, float, …)

The 2 least significant bit must be 0 (with this we guarantee that the address is a multiple of 4)

K = 8 bytes (long, double, …)

The 3 least significant bit must be 0 (with this we guarantee that the address is a multiple of 8)

Data alignment in structures

It is important to acknowledge how data is aligned in C structures since it is necessary to count with the “blank spaces” to access the elements of a structure in assembly

Inside a structure:

We must satisfy all the alignment rules for each member of the structure

Each structure has an alignment requirement of **K**, which may require implicit internal padding, depending on the previous member

The placement of a structure in memory:

Given **K**, the largest alignment requirement inside the structure

The starting address of the structure must be a multiple ok **K**

The total size of the structure must be a multiple of **K**, which may require padding after the last member (external padding)

Data alignment in structures - Examples

Total size: 24 bytes

- $\mathbb{K}$=8, due to the member *c* of type `long`

```
struct S1{
  char a;
  int b[2];
  long c;
};
```

a	3 bytes	b	4 bytes	c
0	[gap]	4 ... 11	[gap]	16 ... 23

Total size: 24 bytes

- $\mathbb{K}$=8, due to the member *b* of type `long`

```
struct S2{
  int a;
  long b;
  short c;
};
```

a	4 bytes	b	c	6 bytes
0 ... 3	[gap]	8 ... 15	16 .. 17	[gap]

Important note

The starting address of both structures must be a multiple of $\mathbb{K} = 8$

ARQCP Flashcard
“Dynamic Memory Allocation”

Virtual Memory

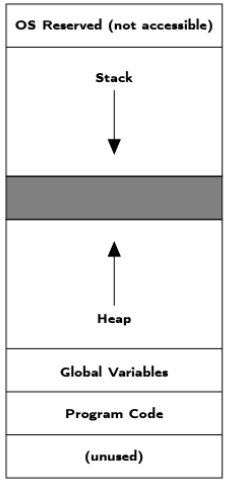
The OS gives the illusion that a certain process has exclusive access to a contiguous memory address space

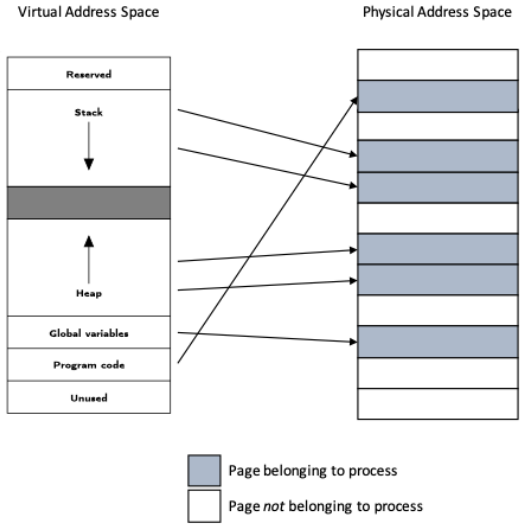
This continuous memory address space is known as **virtual memory address space**

Like the stack, the heap expands and contracts dynamically at run time as a result of calls to allocate/free memory

A difference between this two is that heap grows toward higher addresses

The heap is usually used by C programmers when they need to acquire additional virtual memory at run time





Dynamic memory allocation in C

The process of dynamically allocate memory gives a more flexible and efficient memory management by allocating and freeing space on the heap depending on the needs of the program

In C the standard library provides calls to dynamically allocate/free memory on the heap

- **void* malloc (size_t size):**
  - Allocates a continuous memory block of at least ***size*** bytes and returns a pointer to it
  - The allocated memory **IS NOT** initialized
  - If malloc() encounters a problem then it returns **NULL**

- **void free (void *ptr):**
  - Free allocated heap memory blocks previously allocated with a memory allocation call
  - The ptr argument must point to the beginning of an allocated block that was obtained from a memory allocation call

malloc() and free() Example

```c
#include <stdlib.h> /* malloc() and free() are part of stdlib */

/* declare pointer */
int *ptr_int=NULL;
/* allocate 10 integers in the heap */
ptr_int=(int *) malloc(10 * sizeof(int));

/* use allocated memory */
ptr_int[0] = 10;
*(ptr_int + 1) = 20;

/* free memory */
free(ptr_int);
```

Dynamic memory allocation in C

- **void* calloc (size_t n, size_t size):**
  - Reserves a continuous block of memory of at least **n** * **size** bytes
  - The memory block is initialized to 0
  - Returns a pointer to the allocated memory block, or **NULL** in error

`calloc()` **Example**

```c
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointer */
int *ptr_int=NULL;
/* allocate 100 integers in the heap */
ptr_int=(int *) calloc(100, sizeof(int));
...
/* free memory */
free(ptr_int);
```

- **Void* realloc (void *ptr, size_t size):**
  - Changes the size of a memory block previously allocated with malloc() or calloc()
  - Return a pointer to the allocated memory block, or **NULL** in error
  - Data is kept if size is increased, or truncated if size is decreased

`realloc()` **Example**

```c
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointer */
int *ptr_int=NULL;
/* allocate 100 integers in the heap */
ptr_int=(int *) calloc(100, sizeof(int));

/* allocate one more integer in the heap */
ptr_int=(int *) realloc(ptr_int, 101 * sizeof(int));

/* free memory */
free(ptr_int);
```

Dynamic memory allocation in C

It is important to verify if realloc() fails

If it fails the pointer to the previous memory block is lost

To avoid this, make sure to use a temporary pointer to check the result of realloc ()

This could happen when there are enough memory to allocate but not all is available in one contiguous block that can satisfy the allocation request (**memory fragmentation**)

Check Return of `realloc()` Example

```c
#include <stdlib.h> /* *alloc(), free() are part of stdlib */

/* declare pointers */
int *ptr_int=NULL, *ptr_tmp=NULL;
/* number of ints to allocate */
int n=100;

/* allocate n integers in the heap */
ptr_int=(int *) calloc(n, sizeof(int));

/* allocate one more integer in the heap
   NOTE: return is stored in temporary pointer */
ptr_tmp=(int *) realloc(ptr_int,(n+1) * sizeof(int));

/* check realloc() return */
if(ptr_tmp!=NULL){
        ptr_int=ptr_tmp;
        ptr_tmp=NULL;
}

/* free memory */
free(ptr_int);
```

ARQCP Flashcard
“Dynamic Memory Allocation II”

Static Multidimensional Arrays

In C, multidimensional arrays are implemented as unidimensional arrays with row-major ordering, basically they are arrays of arrays

Example:
int md_array [5][2];

Declares an array of 5 one-dimensional arrays of 2 integers each
The array occupies 5 x 2 x sizeof(int) bytes
It can be statically initialized with something like:

int md_array [5][2] = {{1,2},{3,4},{5,6},{7,8},{9,10}};

Accessing value in C

```
int get_value(int md_array[][2], int i, int j){
  /* return md_array[i][j]; */
  return *(*(md_array + i) + j);
}
```

Accessing value in Assembly

```
# only one memory access to get m[i][j]
get_value:
  # m in %rdi, i in %esi, j in %edx

  shll $3, %esi                # each line has 2 ints (8 bytes)
  addq %rsi, %rdi              # address of line i
  movl (%rdi,%rdx,4), %eax     # m[i][j]
  ret
```

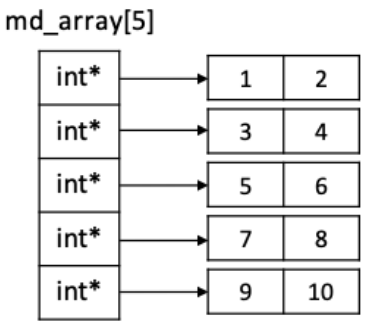
Variable-size Multidimensional Arrays

To allocate dynamically the matrix[y][k] seen before we will have a ”pointer of pointers”

Basically, the matrix should be a dynamic array of pointers to int, with y positions

Each of those pointers will be an array of k integers

With this, writing matrix[y][k] is the same as *(matrix[y] + k) and *(*(matrix +y) + k)



Expression	Type
$md\_array[2]$	Pointer to integer
md_array	Pointer to array of two integers
$md\_array + 1$	Pointer to array of two integers
$*(md\_array + 1)$	Pointer to integer
$*(md\_array + 2) + 1$	Pointer to integer
$*(*(md\_array + 2) + 1)$	Integer (md_array[2][1])
$*md\_array$	Pointer to integer
$**md\_array$	Integer (md_array[0][0])
$*(*md\_array + 1)$	Integer (md_array[0][1])

Allocate variable-size multidimensional array

```
int main(void)
{
  int i, y=5,k=10; /* number of lines (Y) and columns (K) */
  int **a;       /* address of the multi-dimensional array */

  /*  array of int* with size Y */
  a = (int**) calloc(y,sizeof(int*));
  if(a == NULL){
    printf("Error reserving memory.\n"); exit(1);
  }
  for(i = 0; i < y ; i++){
    /* in each position of the pointer array,
       reserve memory for K integers */
    *(a+i) = (int*) calloc(k,sizeof(int)); //Note: *(a+i) same as a[i]
    if(a[i] == NULL){
      printf("Error reserving memory.\n"); exit(1);
    }
  }
  ... /* use multi-dimensional array */
  /* free memory */
  for(i = 0; i < y ; i++)
    free(*(a+i));
  free(a);
  return 0;
}
```

Accessing the multidimensional array values

Accessing value in C

```c
int get_value(int **md_array, int i, int j){
  /* return md_array[i][j]; */
  return *(*(md_array+i) + j);
}
```

Accessing value in Assembly

```
# two memory accesses to get m[i][j]
get_value:
   # m in %rdi, i in %esi, j in %edx

   movq (%rdi,%rsi,8), %rdi      # address of line i
   movl (%rdi,%rdx,4), %eax      # m[i][j]
   ret
```