DLDCA Lab 2: Assembly
Week 3

August 26, 2025

Introduction

In this week's lab, you shall be learning about how structs are placed in memory. You will also use xmm registers and few special instructions to deal with doubles (64 bit floats). The cheatsheet for assembly instructions can be found in section 1. Refer to section 2 for an overview on instruction to deal with doubles using xmm registers.

Necessary tools

We shall be using `nasm` in order to compile x86 assembly code.

Before starting the lab, ensure that the tool is installed in your system. This can be verified by running the following:

```
$ nasm
nasm: fatal: no input file specified
Type nasm -h for help.
```

Structure of submission/templates

```
your-roll-no/
|- task1/
    |- cmplx_arith.asm (TODO)
    |- compile.sh
|- task2/
```

```
        |- polars_to_rect.asm (TODO)
        |- compile.sh
    |- task3/
        |- sieve.asm (TODO)
        |- sieve.c
        |- compile.sh
```

The folder structure of the expected submission for this lab is given above. We expect this folder to be compressed into a tarball with the naming convention of your-roll-no.tar.gz. The command given below can be used to do the same

```
$ tar -cvzf your-roll-no.tar.gz your-roll-no/
```

Tasks

An overview of tasks can be seen in `TODO.md`. **Please go through this file before starting the lab**. This task consists of two subtasks. This task is expected to be submitted in-lab. The code for this task can be found inside the `task1` folder.

Understanding structs in x86

Structs can be thought of as custom data structures placed continuously in memory. A struct has ‘fields’, where each field is a part of the struct. For example in this assignment complex number are represented as a struct with three fields. The first field is the name of the complex number which is a single byte character, the second field is a 64 bit double to represent the real part of the complex number and the third field is the imaginary part of the complex number.

Thus a complex number $3 + 4$i with the name ’a‘ is represented as

```
complex1:
    complex1_name db 'a'
    complex1_real dq 1.0
    complex1_img  dq 2.5
```

where dq represents a quad word (64 bits). But if you observe in the template code given the complex struct is represented as follows.

```
complex1:
    complex1_name db 'a'
    complex1_pad  db 7 dup(0)
    complex1_real dq 1.0
    complex1_img  dq 2.5
```

To understand why this is the case we need to understand structure padding. Click Here to watch the video for the same. This video is also captioned and provided to you.

The usecase for structs is visible from how we use it to represent complex numbers.

Now, a big issue is how to access the fields of the struct in ASM. Well, as we discussed, the structure is just a continuous data structure. So, the struct's placement in memory can be visualized like:

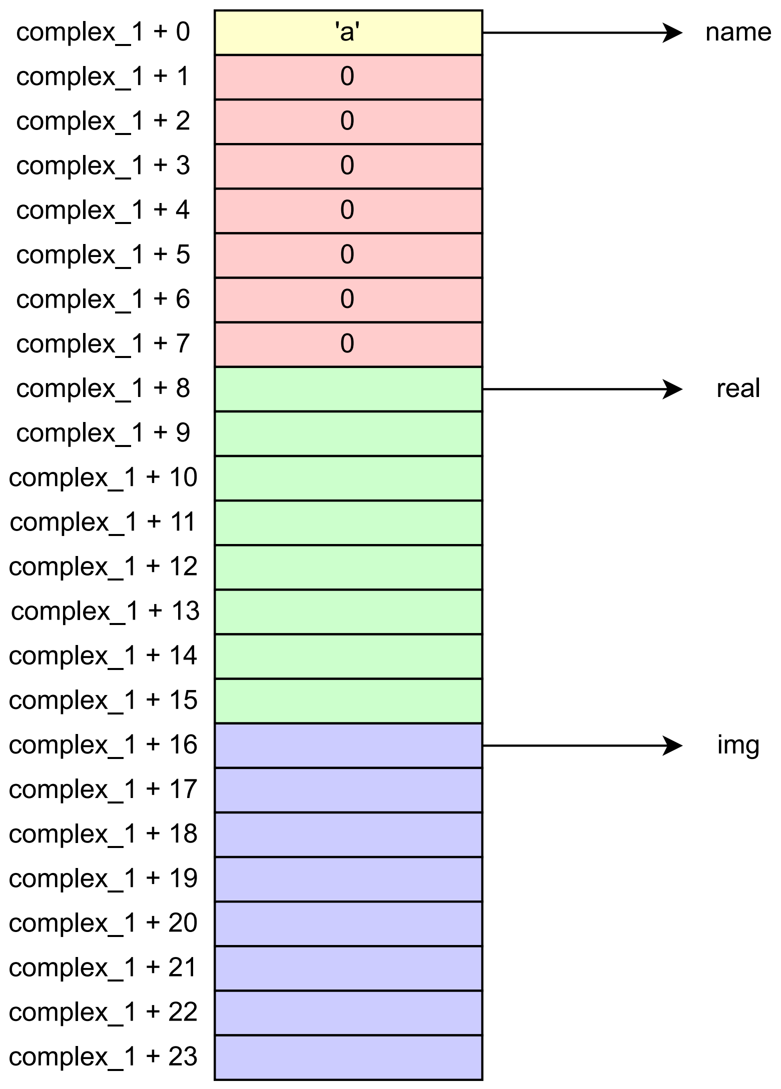

Figure 1: Layout of complex struct in memory: complex_1 is the address of the label complex_1

From the diagram it is clear to see that the individual fields can be accessed by offsetting the address of the complex struct by appropriate amounts (complex + 8) in the case of the field 'real'.

Few notes while using doubles with XMM registers

- To load a constant there is no immediate values supported, a simple workaround is to initialize constants in the data segment and use them from memory
- When initializing doubles make sure to include a decimal point (Eg. 6.0 instead of 6). This makes sure that the value is interpreted as a double and not as an int.
- To pass double arguments to functions use xmm0, xmm1, xmm2... in that order. Further xmm registers are caller saved and need to be saved if they are needed in the future by the caller. In this case push xmm does not work.
- Instead of push, manually offset the stack pointer and store in the stack to push. Similarly instead of pop load from stack and increment the stack pointer manually.[1]

```
    sub rsp, 16    ;push
    movaps [rsp], xmm*
    ;function call here
    movaps xmm*, [rsp] ;pop
    add rsp, 16
```

  Replace xmm* with the relevant xmm register

Task 1: Complex numbers arithmetic

In this week's inlab you have implemented a few operations on complex numbers, the goal of this task is to implement a few more operations to complete this set. Note that incase you have done any of the parts mentioned in the inlab, you can feel free to reuse your implementations.

Note that print_cmplx is implemented already in the template given for the outlab. This need not be changed

[1]Note that the xmm register is 128 bits in size, the instructions we use only take 64 bits from it as a float value

Addition with complex numbers:

Our complex number struct can be represented as follows:

```
struct complex{
    char name;
    double real;
    double img;
};
```

The addition function you implement must take memory addresses as its parameters:

1. Source address of complex 1
2. Source address of complex 2
3. Destination address of result

In C, this could become something along the lines of:

```
void add_cmplx(struct complex* a,struct complex* b,struct
    complex* c){
    c->real = a->real + b->real
    c->img = a-> img + b->img
}
```

Subtraction with complex numbers:

The subtract function you implement must take memory addresses as its parameters:

1. Source address of complex 1
2. Source address of complex 2
3. Destination address of result

In C, this could become something along the lines of:

```
void sub_cmplx(struct complex* a,struct complex* b,struct
    complex* c){
    c->real = a->real - b->real
    c->img = a-> img - b->img
}
```

Multiplication with complex numbers:

The multiply function you implement must take memory addresses as its parameters:

1. Source address of complex 1
2. Source address of complex 2
3. Destination address of result

In C, this could become something along the lines of:

```c
void mul_cmplx(struct complex* a,struct complex* b,struct
   complex* c){
   c->real = a->real*b->real - a->img*b->img;
   c->img = a->real*b->img + a->img*b->real
}
```

Reciprocal with complex numbers:

The reciprocal function you implement must take memory addresses as its parameters:

1. Source address of complex
2. Destination address of result

In C, this could become something along the lines of:

```c
void recip_cmplx(struct complex* a,struct complex* b){
   b->real = a->real/(a->real^2 + a->img^2);
   b->img = - a->img/(a->real^2 + a->img^2);
}
```

Task 2: Polar representation of complexes

So far we have dealt with complex numbers in the rectangluar form (ie) as a + ib. All complexes have an equivalent 'polar' representation expressed as $re^{i\theta}$ or as (r,θ) for simplicity.

In this task your goal is to implement a function that returns the rectangular representation of a number given its polar representation.

The relation between polar (r,θ) and rectangular (a + ib) can be represented as

$$a = r\sin\theta$$
$$b = r\cos\theta$$

Clearly implementing sin,cos in assembly is not going to be trivial. We expect sin, cos functions to be implemented using the taylor series approximation.

You can assume the formulas given to be equivalent to sin and cos.

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

Figure 2: Forumlas for sin, cos

Subtask 1: Implement exponentiation

To implement sin,cos using the taylor series we expect you to start by implementing exp. exp can be implemented in c in the following manner

```
double exp(double base, double power){
    double temp = 1;
    for (int i = 0; i < power; i++){
        temp = temp*base;
    }
    return temp;
}
```

Implement the same in assembly (make sure to use xmm registers since the arguments are floats, they are filled in order of xmm0,xmm1,xmm2...).

Subtask 2: Implement sin, cos

Sin, Cos can be implemented in assembly in the following manner. Note that the arguments to sin are given in radian (not relevant still).

```
double sin(double theta){
    return (theta - (exp(theta,3))/6
    + (exp(theta,5)/120) - (exp(theta,7)/5040));
}

double cos(double theta){
    return (1 - (exp(theta,2))/2
    + (exp(theta,4)/24) - (exp(theta,6)/720));
}
```

Implement the same functions in assembly and use them for the following subtask.

Subtask 3: Polars to rectangular

Since polar representation are just two doubles, we can represent it in the same way as the previous representation.

1. Source address of complex
2. Destination address of result

In C, this could become something along the lines of:

```
struct complex_rect{
    char name;
    double real;
    double img;
};
struct complex_polar{
    char name;
    double r;
    double theta;
};
void polars_to_rect(struct complex_polar* a,struct
   complex_rect* b){
    b->real = a->r * cos(a->theta);
    b->img = a->r * sin(a->theta);
}
```

Task 3: Sieve of eratosthenes

In this task we will be implementing the sieve of eratosthenes used to efficiently compute all the primes which are less than a given threshold n. The version to be implemented takes in an integer input from the user and for all numbers $nk \geq 2$ outputs 0 if k is prime else output the lowest factor of k(not 1 or the number itself).

The C code for implementing sieve of eratosthenes has been given to you. Please go through it to understand the algorithm. In a simple sense all we do is that we iterate from 2 to n, if the current value is i then all number i which are divisible by i and not yet marked, are marked as composite (lowest factor is i). The numbers remaining are the prime numbers needed. To perform this calculation we maintain an array (say `arr`) of size n where the i^{th} value of the array being 0 implies the number is prime

Learning objective

The goal of this lab is to give an idea on how programs allocate memory dynamically using the heap. Many times the program requires memory that depends on inputs which are only known at runtime. Furthermore it is also true that the stack has a limited size in most cases which may not be able to service these requirements.

Understanding brk

One natural problem that arises while trying to do this task is the fact that the program has no hints on how much memory to allocate to the array `arr` before the program runs (it changes depending on n).

For programs that require varying amounts of memory allocated at run time, we use the heap. To allocate and deallocate memory on the heap, the `brk` syscall is used.

- **Syscall number:** 12
- **rdi:** Pass the address to set the new program break to

The `brk` syscall returns the address of the **end** of the program break in the `rax` register.

The brk system call is used to manage the program break, which marks the end of the process's data segment.

Instruction	Effect
`mov rax, rbx`	Copies the value inside `rbx` into `rax`
`mov [rax], rbx`	Copies the value inside `rbx` into the memory pointed to be `rax`
`mov rax, [rbx]`	Copies the value inside the memory pointed to by `rbx`, into `rax`

Table 1: Different ways to use the `mov` instruction to manipulate memory

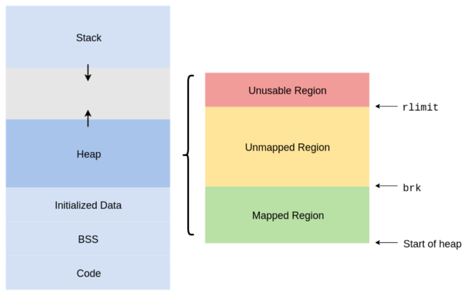

Figure 3: Program break placement in memory

Increasing the program break: To allocate more memory, you call brk with an address higher than the current program break. The system will then extend the process's data segment up to this new address, making the newly allocated memory available for use.

Decreasing the program break: To deallocate memory, you call brk with an address lower than the current program break. This effectively shrinks the process's data segment, releasing the memory back to the system.

You can get the current program break by passing 0 as an argument. You are expected to manipulate the program break to get memory to place the array you would need to compute (refer to the array 'arr' in the given c code)

Table 1 shows how to use the move instruction to manipulate memory.

Implementation

The template file has been very extensively commented on the steps to be followed in each section of the code. Please refer to the template.

To run your code, you may run (from the corresponding task directory):

```
$ bash compile.sh
$ ./sieve
```

Here are some expected outputs:

```
$ ./sieve
7
00202$
```

Explanation:

Numbers	2	3	4	5	6
Outputs	0	0	2	0	2
Explanation	Prime	Prime	Composite, lowest factor=2	Prime	Composite, lowest factor=2

Table 2: Explanation for the expected output for when input is 7

```
$ ./sieve
15
0020202320202$
```

You have been given the C code for the program as well, your output should match that of the C code for all inputs.

1 Assembly instructions cheatsheet (non-exhaustive)

Instruction	Operands	Effect
`mov`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} \,/\, *\texttt{mem} \,/\, \texttt{imm}$
`mov`	`*mem`, `reg`	$*\texttt{mem} \leftarrow \texttt{reg}$
`add`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} + \texttt{reg/*mem/imm}$
`add`	`*mem`, `reg/imm`	$*\texttt{mem} \leftarrow *\texttt{mem} + \texttt{reg/imm}$
`sub`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} - \texttt{reg/*mem/imm}$
`sub`	`*mem`, `reg/imm`	$*\texttt{mem} \leftarrow *\texttt{mem} - \texttt{reg/imm}$
`xor`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} \oplus \texttt{reg/*mem/imm}$
`xor`	`*mem`, `reg/imm`	$*\texttt{mem} \leftarrow *\texttt{mem} \oplus \texttt{reg/imm}$
`and`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} \wedge \texttt{reg/*mem/imm}$
`and`	`*mem`, `reg/imm`	$*\texttt{mem} \leftarrow *\texttt{mem} \wedge \texttt{reg/imm}$
`or`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} \vee \texttt{reg/*mem/imm}$
`or`	`*mem`, `reg/imm`	$*\texttt{mem} \leftarrow *\texttt{mem} \vee \texttt{reg/imm}$
`imul`	`reg`, `reg/*mem/imm`	$\texttt{reg} \leftarrow \texttt{reg} \times \texttt{reg/*mem/imm}$
`imul`	`*mem`, `reg/imm`	$*\texttt{mem} \leftarrow *\texttt{mem} \times \texttt{reg/imm}$
`div`	`reg`	$\texttt{rax} \leftarrow (\texttt{rdx:rax}) \,/\, \texttt{reg}$ $\texttt{rdx} \leftarrow (\texttt{rdx:rax}) \,\%\, \texttt{reg}$

`cmp`	`reg/*mem`, `reg/imm`	Sets flags based on subtraction: $\texttt{reg/*mem} - \texttt{reg/imm}$.
		ZF = 1 if equal, SF = 1 if result negative,
		CF = 1 if borrow (unsigned), OF = 1 if signed overflow.

`test`	`reg/*mem`, `reg/imm`	Sets flags based on bitwise AND: $\texttt{reg/*mem} \wedge \texttt{reg/imm}$. ZF = 1 if result is zero, SF = 1 if high bit set, CF and OF are cleared. No result is stored.
`inc`	`reg/*mem`	$\texttt{reg/*mem} \leftarrow \texttt{reg/*mem} + 1$
`dec`	`reg/*mem`	$\texttt{reg/*mem} \leftarrow \texttt{reg/*mem} - 1$
`jmp`	`label`	Unconditional jump to `label`
`je` / `jz`	`label`	Jump if Zero Flag (ZF) = 1
`jne` / `jnz`	`label`	Jump if Zero Flag (ZF) = 0
`jl`	`label`	Jump if Less (SF $\neq$ OF)
`jle`	`label`	Jump if Less or Equal (ZF = 1 or SF $\neq$ OF)
`jg`	`label`	Jump if Greater (ZF = 0 and SF = OF)
`jge`	`label`	Jump if Greater or Equal (SF = OF)
`syscall`		Performs a syscall with syscall number stored in `rax` Arguments passed in other registers starting from `rdi`
`push`	`reg`	Pushes the value in `reg` onto the stack
`pop`	`reg`	Pops the value on top of the stack onto `reg`
`call`	`label`	Calls the function called `label`

Table 3: Common x86-64 Instructions and Their Effects

2 xmm register instructions

Instruction	Operands	Effect
`movsd`	`xmm`, `xmm/*mem`	`xmm` $\leftarrow$ `xmm/*mem`
`movsd`	`*mem`, `xmm`	`*mem` $\leftarrow$ `xmm`
`addsd`	`xmm`, `xmm/*mem`	`xmm` $\leftarrow$ `xmm` $+$ `xmm/*mem`
`subsd`	`xmm`, `xmm/*mem`	`xmm` $\leftarrow$ `xmm` $-$ `xmm/*mem`
`mulsd`	`xmm`, `xmm/*mem`	`xmm` $\leftarrow$ `xmm` $\times$ `xmm/*mem`
`divsd`	`xmm`, `xmm/*mem`	`xmm` $\leftarrow$ `xmm` $\div$ `xmm/*mem`
`movq`	`xmm`, `reg/*mem`	`xmm` $\leftarrow$ 64-bit `reg/*mem`
`movq`	`reg/*mem`, `xmm`	64-bit `reg/*mem` $\leftarrow$ `xmm`
`movaps`	`xmm`, `xmm/m128`	`xmm` $\leftarrow$ 128-bit `xmm/mem` (aligned)
`movaps`	`xmm/m128`, `xmm`	128-bit `xmm/mem` (aligned) $\leftarrow$ `xmm`

Table 4: Common XMM Double-Precision Instructions