THIS DRAFT HAS BEEN SUPERSEDED BY GITHUB:
https://github.com/FFmpeg/asm-lessons/blob/main/lesson_one/index.md

FFmpeg Assembly Language Lesson One

Introduction

Welcome to the FFmpeg School of Assembly Language. You have taken the first step on the most interesting, challenging, and rewarding journey in programming. These lessons will give you a grounding in the way assembly language is written in FFmpeg and open your eyes to what's actually going on in your computer.

Required Knowledge

- Knowledge of C, in particular pointers. If you don't know C, work through The C Programming Language book
- High School Mathematics (scalar vs vector, addition, multiplication etc)

What is assembly language?

Assembly language is a programming language where you write code that directly corresponds to the instructions a CPU processes. Human readable assembly language is, as the name suggests, *assembled* into binary data, known as *machine code*, that the CPU can understand. You might see assembly language code referred to as “assembly” or “asm” for short.

The vast majority of assembly code in FFmpeg is what's known as *SIMD, Single Instruction Multiple Data*. SIMD is sometimes referred to as vector programming. This means that a particular instruction operates on multiple elements of data at the same time. Most programming languages operate on one data element at a time, known as scalar programming.

As you might have guessed, SIMD lends itself well to processing images, video, and audio which have lots of data ordered sequentially in memory. There are specialist instructions available in the CPU to help us process sequential data.

In FFmpeg, you'll see the terms “assembly function”, “SIMD”, and “vector(ise)” used interchangeably. They all refer to the same thing: Writing a function in assembly language by hand to process multiple elements of data in one go. Some projects may also refer to these as “assembly kernels”.

All of this might sound complicated, but it's important to remember that in FFmpeg, high schoolers have written assembly code. As with everything, learning is 50% jargon and 50% actual learning.

Why do we write in assembly language?

To make multimedia processing fast. It’s very common to get a 10x or more speed improvement from writing assembly code, which is especially important when wanting to play videos in real time without stuttering. It also saves energy and extends battery life. It's worth pointing out that video encode and decode functions are some of the most heavily used functions on earth, both by end-users and by big companies in their datacentres. So even a small improvement adds up quickly.

You'll often see, online, people use *intrinsics,* which are C-like functions that map to assembly instructions to allow for faster development. In FFmpeg we don't use intrinsics but instead write assembly code by hand. This is an area of controversy, but intrinsics are typically around 10-15% slower than hand-written assembly (intrinsics supporters would disagree), depending on the compiler. For FFmpeg, every bit of extra performance helps, which is why we write in assembly code directly. There's also an argument that intrinsics are difficult to read owing to their use of "Hungarian Notation".

You may also see *inline assembly* (i.e. not using intrinsics) remaining in a few places in FFmpeg for historical reasons, or in projects like the Linux Kernel because of very specific use cases there. This is where assembly code is not in a separate file, but written inline with C code. The prevailing opinion in projects like FFmpeg is that this code is hard to read, not widely supported by compilers and unmaintainable.

Lastly, you'll see a lot of self-proclaimed experts online saying none of this is necessary and the compiler can do all of this "vectorisation" for you. At least for the purpose of learning, ignore them: recent tests in e.g. the dav1d project showed around a 2x speedup from this automatic vectorisation, while the hand-written versions could reach 8x.

Flavours of assembly language
These lessons will focus on x86 64-bit assembly language. This is also known as amd64, although it still works on Intel CPUs. There are other types of assembly for other CPUs like ARM and RISC-V and potentially in the future these lessons will be extended to cover those.

There are two flavours of x86 assembly syntax that you'll see online: AT&T and Intel. AT&T Syntax is older and harder to read compared to Intel syntax. So we will use Intel syntax.

Supporting materials
You might be surprised to hear that books or online resources like Stack Overflow are not particularly helpful as references. This is in part because of our choice to use handwritten assembly with Intel syntax. But also because a lot of online resources are focused on operating system programming or hardware programming, usually using non-SIMD code. FFmpeg assembly is particularly focused on high performance image processing, and as you'll see it's a particularly unique approach to assembly programming. That said, it's easy to understand other assembly use-cases once you've completed these lessons

Many books go into a lot of computer architecture details before teaching assembly. This is fine if that's what you want to learn, but from our standpoint, it's like studying engines before learning to drive a car.

That said, the diagrams in the later parts of "The Art of 64-bit assembly" book showing SIMD instructions and their behaviour in a visual form are helpful:
https://artofasm.randallhyde.com/

A discord server is available to answer questions:
https://discord.com/invite/Ks5MhUhqfB

Registers
Registers are areas in the CPU where data can be processed. CPUs don't operate on memory directly, but instead data is loaded into registers, processed, and written back to memory. In assembly language, generally, you cannot directly copy data from one memory location to another without first passing that data through a register.

General Purpose Registers
The first type of register is what is known as a General Purpose Register (GPR). GPRs are referred to as general purpose because they can contain either data, in this case up to a 64-bit value, or a memory address (a pointer). A value in a GPR can be processed through operations like addition, multiplication, shifting, etc.

In most assembly books, there are whole chapters dedicated to the subtleties of GPRs, the historical background etc. This is because GPRs are important when it comes to operating system programming, reverse engineering, etc. In the assembly code written in FFmpeg, GPRs are more like scaffolding and most of the time their complexities are not needed and abstracted away.

Vector registers
Vector (SIMD) registers, as the name suggests, contain multiple data elements. There are various types of vector registers:

- `mm` registers - MMX registers, 64-bit sized, historic and not used much any more
- `xmm` registers - XMM registers, 128-bit sized, widely available
- `ymm` registers - YMM registers, 256-bit sized, some complications when using these
- `zmm` registers - ZMM registers, 512-bit sized, limited availability

Most calculations in video compression and decompression are integer-based so we'll stick to that. Here's an example of 16 bytes in an `xmm` register:

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**

But it could be eight words (16-bit integers)

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Or four double words (32-bit integers)

a	**b**	**c**	**d**

Or two quadwords (64-bit integers):

a	**b**

To recap:

- **b**ytes - 8-bit data
- **w**ords - 16-bit data
- **d**oublewords - 32-bit data
- **q**uadwords - 64-bit data
- **d**ouble **q**uadwords - 128-bit data

The bold characters will be important later.

x86inc.asm include
You'll see in many examples we include the file x86inc.asm. X86inc.asm is a lightweight abstraction layer used in FFmpeg, x264, and dav1d to make an assembly programmer's life easier. It helps in many ways, but to begin with, one of the useful things it does is it labels GPRs, `r0, r1, r2.` This means you don't have to remember any register names. As mentioned before, GPRs are generally just scaffolding so this makes life a lot easier.

A simple scalar asm snippet

Let's look at a simple (and very much artificial) snippet of scalar asm (assembly code that operates on individual data items, one at a time, within each instruction) to see what's going on:

```
mov  r0q, 3
inc  r0q
dec  r0q
imul r0q, 5
```

In the first line, the *immediate value* 3 (a value stored directly in the assembly code itself as opposed to a value fetched from memory) is being stored into register r0 as a quadword. Note that in Intel syntax, the source operand (the value or location providing the data, located on the right) is transferred to the destination operand (the location receiving the data, located on the left), much like the behavior of memcpy. You can also read it as "r0q = 3", since the order is the same. The "q" suffix of r0 designates the register as being used as a quadword. `inc` increments the value so that r0q contains 4, `dec` decrements the value back to 3. `imul` multiplies the value by 5. So at the end, r0q contains 15.

Note that the human readable instructions such as `mov` and `inc`, which are assembled into machine code by the assembler, are known as *mnemonics*. You may see online and in books mnemonics represented with capital letters like `MOV` and `INC` but these are the same as the lower case versions. In FFmpeg, we use lower case mnemonics and keep upper case reserved for macros.

Understanding a basic vector function

Here's our first SIMD function:

```
%include "x86inc.asm"

SECTION .text

;static void add_values(const uint8_t *src, const uint8_t *src2)
INIT_XMM sse2
cglobal add_values, 2, 2, 2, src, src2
    movu  m0, [srcq]
    movu  m1, [src2q]

    paddb m0, m1

    movu  [srcq], m0

    RET
```

Let's go through it line by line:

```
%include "x86inc.asm"
```

This is a "header" developed in the x264, FFmpeg, and dav1d communities to provide helpers, predefined names and macros (such as `cglobal` below) to simplify writing assembly.

```
SECTION .text
```

This denotes the section where the code you want to execute is placed. This is in contrast to the .data section, where you can put constant data.

```
;static void add_values(const uint8_t *src, const uint8_t *src2);
INIT_XMM sse2
```

The first line is a comment (the semi-colon ";" in asm is like "//" in C) showing what the function argument looks like in C. The second line shows how we are initialising the function to use XMM registers, using the sse2 instruction set. This is because `paddb` is an sse2 instruction. We'll cover sse2 in more detail in the next lesson.

```
cglobal add_values, 2, 2, 2, src, src2
```

This is an important line as it defines a C function called "`add_values`".

Let's go through each item one at a time:

- The next parameter shows it has two function arguments.

- The parameter after that shows that we'll use two GPRs for the arguments. In some cases we might want to use more GPRs so we have to tell x86util we need more.
- The parameter after that tells x86util how many XMM registers we are going to use.
- The following two parameters are labels for the function arguments.

It's worth noting that older code may not have labels for the function arguments but instead address GPRs directly using r0, r1 etc.

```
movu  m0, [srcq]
movu  m1, [src2q]
```

`movu` is shorthand for `movdqu` (move double quad unaligned). Alignment will be covered in another lesson but for now `movu` can be treated as a 128-bit move from `[srcq]`. In the case of `mov`, the brackets mean that the address in `[srcq]` is being dereferenced, the equivalent of `*src` *in C.* This is what's known as a load. Note that the "q" suffix refers to the size of the pointer (i.e in C it represents `sizeof(src) == 8` on 64-bit systems, and x86asm is smart enough to use 32-bit on 32-bit systems) but the underlying load is 128-bit.

Note that we don't refer to vector registers by their full name, in this case `xmm0`, but as `m0`, an abstracted form. In future lessons you'll see how this means you can write code once and have it work on multiple SIMD register sizes.

```
paddb m0, m1
```

`paddb` (read this in your head as *p-add-b*) is adding each byte in each register as shown below. The "`p`" prefix stands for "packed" and is used to identify vector instructions vs scalar instructions. The "`b`" suffix shows that this is bytewise addition (addition of bytes).

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**

+

q	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**	**aa**	**ab**	**ac**	**ad**	**ae**	**af**

=

a+q	**b+r**	**c+s**	**d+t**	**e+u**	**f+v**	**g+w**	**h+x**	**i+y**	**j+z**	**k+aa**	**l+ab**	**m+ac**	**n+ad**	**o+ae**	**p+af**

```
movu  [srcq], m0
```

This is what's known as a store. The data is written back to the address in the `srcq` pointer.

```
RET
```

This is a macro to denote the function returns. Virtually all assembly functions in FFmpeg modify the data in the arguments as opposed to returning a value.

As you’ll see in the assignment, we create function pointers to assembly functions and use them where available.