

hi, please, isnt here core declared twice?

```
// Define core struct
typedef struct
{
    REGS_TYPE regs[16];
    TU8 rsvd[42*4]; //reserved (msg in/out buffers, breakpoints)
    PC_TYPE pc;
    TU8 data[DATA_SIZE];
    TU8 prog[PROG_SIZE];
} __attribute__((aligned(4))) Core;
```

```
Core core = {
    {0}, // initialize regs to all zeroes
    {0}, // initialize rsvd to all zeroes
    0, // initialize pc to zero
    {0}, // initialize data to all zeroes
    {0} // initialize prog to all zeroes
};
```

```
//main
int vmex(const TU8 testprog[], int progsz)
{
    TU8 opcode = 0;
    TU8 rs1 = 0;
    TU8 rs2 = 0;
    TU8 rd = 0;
    IMMS_TYPE imm = 0;
    PC_TYPE pc = 0;

    //temp debug
    TS32 r10i = 0;
    // TU32 r10u;
    // TS32 r11i;
    // TU32 r11u;
```

```
Core* _core_ = &core;

REGS_TYPE* _reg_ = _core_->regs;
TU8* _prog_ = _core_->prog;
TU8* _data_ = _core_->data;

//fill program
memcpy(_prog_, testprog, progsz);
```



Hi! In the code you provided, the **Core** struct is defined only once, and then an instance of that struct is created and initialized.

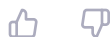
Here is the code with comments highlighting the struct definition and the instance creation:

c

Copy code

```
// Define core struct typedef struct { REGS_TYPE regs[16]; TU8 rsvd[42*4];
//reserved (msg in/out buffers, breakpoints) PC_TYPE pc; TU8 data[DATA_SIZE];
TU8 prog[PROG_SIZE]; } __attribute__((aligned(4))) Core; // Create and
initialize an instance of the Core struct Core core = { {0}, // initialize regs
to all zeroes {0}, // initialize rsvd to all zeroes 0, // initialize pc to zero
{0}, // initialize data to all zeroes {0} // initialize prog to all zeroes };
```

So, there is no issue with the **Core** struct being declared twice. The struct is defined once, and then an instance named **core** is created and initialized.



I mean at this line: `Core* _core_ = &core;`



Ah, I see what you mean now. In this line, a pointer to the **Core** struct is being created and initialized, rather than declaring the **Core** struct itself again:

```
Core* _core_ = &core;
```

`_core_` is a pointer to a **Core** struct, and it is initialized to point to the address of the previously declared and initialized **core** instance. This doesn't create a second instance of the **Core** struct; it just creates a pointer that refers to the existing **core** instance.



yaa, I see it now too, thanks - its really better to place asterisk immediatelly after type to declare pointer - please, this is C or C++ approach? I see it also aligned with required C# unsafe behavior, I will probably share the C source with C# having some defines to abstract this between C/C# ...



The code you've provided is written in C, not C++. The syntax, types, and constructs used in the code are consistent with the C programming language. However, C++ is mostly backward-compatible with C, so this code would likely compile and run in a C++ environment as well.

As for C#, it is a different programming language and has a different syntax. While C# is also part of the C family of languages, it has its own set of features and is designed to work with the .NET runtime. To use C-like code in C#, you can use the **unsafe** keyword to write code that uses pointers and other low-level constructs. However, using **unsafe** code in C# is generally discouraged, as it can potentially lead to memory-related issues and other bugs if not used carefully.

If you want to share the same code between C/C++ and C#, you can create a C/C++ library and use P/Invoke in C# to call functions from the native C/C++ library. This way, you can keep the low-level code in C/C++ and use C# for the high-level logic without using **unsafe** code in C#.

Here's a brief example of how to use P/Invoke in C#:

1. Create a C library with a function you want to use in C#:

c

 Copy code

```
// mylib.c #include <stdint.h> int32_t add(int32_t a, int32_t b) { return a + b;
}
```

2. Compile the C library into a shared library (.dll on Windows, .so on Linux, or .dylib on macOS).

3. Use P/Invoke in C# to call the function



well, if code is proven and tested to work in C, then unsafe isn't unsafe in fact, in C#, you know .... I need to share the same syntax, which is how C# was in fact designed in relation to unsafe pointers behavior ... that's okay, ... I want to know about pointers declaration standard as Core\* core or Core \*core ... which has which origin? C or C++?



Both C and C++ share the same syntax for pointer declarations, and the placement of the asterisk (\*) is a matter of personal preference or coding style. There is no strict rule that dictates whether to place the asterisk right after the type or before the variable name.

Some developers prefer to write the pointer declaration as:

c

 Copy code

```
Core *core;
```

Others prefer:

c

 Copy code

```
Core* core;
```

Both styles are valid in both C and C++ and have the same meaning. The choice of which style to use typically depends on the coding guidelines or conventions followed by the project or organization.

It is worth noting that the C++ community tends to prefer **Core\* core** because it emphasizes the type being a "pointer to Core," whereas the C community might prefer **Core\*core** as it emphasizes the variable being a pointer. However, these are general trends and not strict rules.

Whichever style you choose, it is essential to be consistent throughout your codebase for better readability and maintainability.



I know that's both valid, but some approach was more common in C and some in C++ originally, now it doesn't matter, but what WAS original preference?



Historically, in the C programming language, the more common practice was to place the asterisk next to the variable name, like this:

c

 Copy code

```
Core *core;
```

This style emphasizes that the variable **core** is a pointer. In C++, the trend shifted towards placing the asterisk next to the type:

c++

 Copy code

```
Core* core;
```

This style emphasizes that the type is a "pointer to Core." The reason behind this shift is that C++ introduced features like classes, templates, and references, which made the type system more complex. As a result, placing the asterisk next to the type made it clearer what the type represents.

To summarize, the original preference in C was **Core \*core**, while in C++ it shifted towards **Core\* core**. However, both styles are valid and used in both languages, and the choice ultimately depends on the coding style and conventions followed by the project or organization.



ya, thanks, that was my question, appologize, tn timer again - the C++ is shared identically with C# syntax for unsafe pointers and thats great, I will use this ....



You're welcome! I'm glad the information was helpful.

Yes, C#'s unsafe pointer syntax is similar to C++ syntax, and this can be convenient when working with low-level code or interoperating between C++ and C#. However, please remember to use the **unsafe** keyword and be cautious when working with unsafe code in C#, as it can potentially lead to memory-related issues and other bugs if not used carefully.

If you have any further questions or need assistance, please feel free to ask. Good luck with your project!



In fact, my original ide was just this, to share C and C# VMEX core source this way, but later I was considering the new VMEX related ASLIX assembler to write VMEX core in, in itself but here is chicken-egg problem or catch2 ... you know



I understand your situation. It's common to face a "chicken-and-egg" problem when trying to develop a system that relies on its own components, like writing a VMEX core in an assembler language (ASLIX) designed for that same VMEX core. In such cases, you might need to use a multi-stage or bootstrapping approach to develop your system.

Here's a high-level strategy you can follow:

1. **Stage 1:** Write the initial VMEX core and ASLIX assembler in a language like C, C++, or C# (using unsafe code if necessary). This stage will allow you to develop, test, and validate

the functionality of both components.

2. **Stage 2:** Write a minimal version of the VMEX core in ASLIX assembler. This minimal version should be just enough to support the basic functionality required to execute ASLIX assembler code. You can use the ASLIX assembler developed in Stage 1 to assemble this minimal VMEX core.
3. **Stage 3:** Use the minimal VMEX core (now running in ASLIX assembler) to develop more advanced features and optimizations for the VMEX core. You can iteratively improve the VMEX core using the ASLIX assembler from Stage 1 as needed.
4. **Optional:** Once the VMEX core is mature and stable, you can consider writing the ASLIX assembler itself in ASLIX assembler, effectively making the entire system self-hosting.

This multi-stage approach allows you to develop the VMEX core and ASLIX assembler iteratively and bootstrap them using a more familiar language, like C, C++, or C#. Once both components are functional, you can gradually migrate the code to the target language (ASLIX assembler) and make the system self-hosting

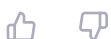


yes, C/C# will be reference implementation, later to be rewritten in any kind of assembler, so may be also the ASLIX ...



That sounds like a good plan. Starting with a reference implementation in C/C# will allow you to develop, test, and refine your VMEX core and ASLIX assembler with familiar tools and languages. Once your reference implementation is stable and working as expected, you can then rewrite it in various assembler languages, including ASLIX or any other suitable assembler, depending on your project requirements and goals.

This approach provides flexibility and helps you ensure that the core functionality is well-defined and working correctly before moving on to more specialized implementations. Additionally, having a reference implementation in a higher-level language like C/C# can be useful for documentation, testing, and maintenance purposes. Good luck with your project, and if you have any further questions or need assistance, please feel free to ask!





okay, the ASLIX thing is challenging, as I in fact want to use single VMEX/RV32E based syntax to write assembly both for the VMEX itself and also to be able to emit code for various target native platforms, so transpile ASLIX assembly into target assembly, having defined some common registers structure VMEX-like so 16 32bit registers, on 8bit systems as 16\*4 8bit or 32\*2 16bit register pairs to be handled from single ASLIX syntax and transpiled into target assembly. It will NOT use all the target native assembly features and will be a little bit slower than pure native target, but will allow to use SINGLE SHARED syntax of ASLIX everywhere to reimplement even the VMEX core this unified way ... It's like the original C purpose to be used to write UNIX ...



---

Your idea of creating a unified ASLIX assembly language that can be used for both VMEX/RV32E and various target native platforms is ambitious and interesting. It indeed resonates with the original purpose of C, which was to create a high-level language for writing system software, such as UNIX, that could be easily ported across different hardware platforms.

To achieve your goal, you might consider the following high-level steps:

1. **Define the ASLIX assembly language:** Create a specification for the ASLIX assembly language, including the common register structure, instructions, and addressing modes. Make sure to keep it general enough to be adaptable to various target platforms while maintaining compatibility with the VMEX/RV32E.
2. **Develop an ASLIX assembler:** Implement an assembler that can parse ASLIX assembly code and generate the corresponding binary code for the VMEX/RV32E virtual machine.
3. **Implement transpilers for target platforms:** Develop transpilers that can take ASLIX assembly code and generate the equivalent assembly code for various target platforms. These transpilers should be able to map the common register structure and instructions from ASLIX to the target platform's native assembly language. Keep in mind that some platforms might require specific optimizations or workarounds to achieve good performance.
4. **Testing and optimization:** Thoroughly test your ASLIX assembler and transpilers with various code samples to ensure correct functionality and performance. Optimize your implementations to minimize overhead and maximize the efficiency of the generated code on each target platform.



By following these steps, you can create a versatile assembly language that can be used for a wide range of platforms while maintaining a single, unified syntax. This approach can simplify development, maintenance, and porting efforts, albeit at the potential cost of not fully utilizing each platform's unique features and performance capabilities.

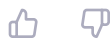


ya, tahts the goal I have, thanks



You're welcome! It's an exciting goal, and I wish you the best of luck in your project. Developing a unified assembly language like ASLIX has the potential to streamline development processes and make it easier to write and maintain code across different platforms.

If you have any further questions or need assistance during your project, please don't hesitate to ask. We're here to help!



well, now some experiments - can you please try to translate this snippet into assembly of 6502? I need well documented and structured assembly code following this C implementation, thanks in advance



Of course! However, I'll need the C code snippet that you want to be translated into 6502 assembly. Please provide the C code, and I'll be happy to help you with the translation.



sorry, here it si below:

```
while (pc < progsiz)
{
    pc = core.pc;

    //parameters parsing
```

```
TU8* _op_ = (TU8*)&_prog_[pc];  
opcode = *_op_;
```

```
switch (opcode & 0xF0)
```

```
{
```

```
    case R3_TYPE:
```

```
    case R2_TYPE:
```

```
    case RC_TYPE:
```

```
    case RD_TYPE:
```

```
    {
```

```
        rd = _op_[1] >> 4;
```

```
        rs1 = _op_[2] >> 4;
```

```
        rs2 = _op_[3] >> 4;
```

```
        break;
```

```
    }
```

```
    case I1_TYPE:
```

```
    case IA_TYPE:
```

```
    case I4_TYPE:
```

```
    case IE_TYPE:
```

```
    {
```

```
        rd = _op_[1] >> 4;
```

```
        rs1 = _op_[1] & 0xF;
```

```
        imm = *((TU16 *) (_op_ + 2)); // Read 16-bit immediate value directly
```

```
        break;
```

```
    }
```

```
    case SF_TYPE:
```

```
    case SB_TYPE:
```

```
    case S8_TYPE:
```

```
    {
```

```
        rs1 = _op_[1] >> 4;
```

```
        rs2 = _op_[1] & 0xF;
```

```
        imm = *((TU16 *) (_op_ + 2)); // Read 16-bit immediate value directly
```

```
        break;
```

```
    }
```

```
    case U7_TYPE:
```

```
    case U6_TYPE:
```

```
    {
```

```

        rd = _op_[1] >> 4;
        imm = *((TU16 *) (_op_ + 2)); // Read 16-bit immediate value directly
        break;
    }

    case UNKNOWN_TYPE:
    default:
        println("Unknown opcode type");
        break;
}

```



Here's the 6502 assembly code equivalent to the provided C code snippet. Please note that the 6502 assembly language is more limited compared to modern processors, so the code might be more verbose and require additional registers and memory locations to store temporary values. I have tried to keep the structure similar to the C code and added comments for clarity.

java

Copy code

```
; Variables and constants progsiz =
```

network error