

>



how to correct this to compile all files using cc65?

```
set PATH=%PATH%;c:\~sdk\cc65\bin
cc65.exe ..\..\src\*.c ..\..\src\*.h -o VMEX.s
cl65 -C plus4.cfg ..\VMEX.c -o VMEX.bin
```

---

To compile all C files in the **src** directory using the cc65 toolchain, you need to adjust the commands you're using. Here's a step-by-step guide:

1. Set the PATH environment variable to include the cc65 binary folder.

```
batch

set PATH=%PATH%;c:\~sdk\cc65\bin
```

2. Compile each C file in the **src** directory to an object file using cc65.

```
batch

FOR %%f IN (..\..\src\*.c) DO ( cc65 -O -t none -I ..\..\src %%f -o %%~nf.s ca65 -t none %%~nf.s -o %%~nf.o )
```

This loop compiles each C file in the **src** directory to an assembly file with the same name and then assembles it to an object file using ca65.

3. Link all the object files together to create the final binary using ld65.

```
batch

ld65 -C plus4.cfg -o VMEX.bin ..\..\src\*.o
```

This command links all the object files in the **src** directory using the provided **plus4.cfg** configuration file and outputs the final binary as **VMEX.bin**.

You can put these commands into a batch file or run them directly from the command prompt. Make sure the **plus4.cfg** file is available in the worki



no, I want src stay intact, place all generated file in current folder

---

I see. To keep the **src** folder intact and place all generated files in the current folder, you can use the following commands:

1. Set the PATH environment variable to include the cc65 binary folder.

```
batch

set PATH=%PATH%;c:\~sdk\cc65\bin
```

2. Compile each C file in the **src** directory to an object file using cc65 and place the output files in the current folder.

```
batch

FOR %%f IN (..\..\src\*.c) DO ( cc65 -O -t none -I ..\..\src ..\..\src\%%f -o %%~nf.s ca65 -t none %%~nf.s -o %%~nf.o )
```

This loop compiles each C file in the **src** directory to an assembly file with the same name in the current folder and then assembles it to an object file

3. Link all the object files together to create the final binary using ld65.

batch

```
ld65 -C plus4.cfg -o VMEX.bin *.o
```

This command links all the object files in the current folder using the provided **plus4.cfg** configuration file and outputs the final binary as **VMEX.bin**.

You can put these commands into a batch file or run them directly from the command prompt. Make sure the **plus4.cfg** file is available in the worki



place please path to src folder into variable too

---

Sure, you can define a variable for the **src** folder path and use it in the commands. Here's how to do it:

1. Set the PATH environment variable to include the cc65 binary folder and define a variable for the **src** folder path.

batch

```
set PATH=%PATH%;c:\~sdk\cc65\bin set SRC_DIR=..\src
```

2. Compile each C file in the **src** directory to an object file using cc65 and place the output files in the current folder.

batch

```
FOR %%f IN (%SRC_DIR%\*.c) DO ( cc65 -O -t none -I %SRC_DIR% %SRC_DIR%\%%f -o %%~nf.s ca65 -t none %%~nf.s -o %%~nf.o )
```

This loop compiles each C file in the **src** directory to an assembly file with the same name in the current folder and then assembles it to an object file

3. Link all the object files together to create the final binary using ld65.

batch

```
ld65 -C plus4.cfg -o VMEX.bin *.o
```

This command links all the object files in the current folder using the provided **plus4.cfg** configuration file and outputs the final binary as **VMEX.bin**.

You can put these commands into a batch file or run them directly from the command prompt. Make sure the **plus4.cfg** file is available in the worki



isn't better to use cl65?

---

Yes, using **cl65** can simplify the process since it combines the compilation, assembly, and linking steps into a single command. Here's how you can n

1. Set the PATH environment variable to include the cc65 binary folder and define a variable for the **src** folder path.

batch

```
set PATH=%PATH%;c:\~sdk\cc65\bin set SRC_DIR=..\..\src
```

2. Compile and link each C file in the **src** directory using **c165**, placing the output files in the current folder.

batch

```
FOR %%f IN (%SRC_DIR%\*.c) DO ( c165 -O -t none -C plus4.cfg -I %SRC_DIR% %SRC_DIR%\%%f -o %%~nf.o )
```

This loop compiles and links each C file in the **src** directory, creating object files in the current folder.

3. Link all the object files together to create the final binary using **c165**.

batch

```
c165 -C plus4.cfg -o VMEX.bin *.o
```

This command links all the object files in the current folder using the provided **plus4.cfg** configuration file and outputs the final binary as **VMEX.bin**.



You can put these commands into a batch file or run them directly from the command prompt. Make sure the **plus4.cfg** file is available in the worki



why I have this error reported on this:

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

```
..\..\src\VMEX.c:147: Error: Variable identifier expected
```

```
..\..\src\VMEX.c:147: Error: Undefined symbol: '_op_'
```

```
..\..\src\VMEX.c:147: Error: Assignment to rvalue
```

```
..\..\src\VMEX.c:147: Warning: Converting pointer to integer without a cast
```

It seems there is an issue with the syntax or declaration of the variables in the given line of code. Based on the code snippet you provided:

c

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

It looks like you are trying to declare a pointer to **TU8** (presumably a typedef for **uint8\_t** or a similar type) and assign it the address of an element in

The errors you're encountering might be caused by a missing or incorrect declaration of the **\_prog\_** array, the **pc** variable, or the **TU8** type. To help yo declarations of the variables and types involved.

However, if you just need a pointer to a specific element in the **\_prog\_** array at index **pc**, the provided code snippet should work fine, assuming that .



how to support uint64\_t in cc65?

**cc65** does not support 64-bit integers natively. However, you can work with 64-bit integers using a custom struct and implementing the required ari **uint64\_t**-like structure and implement addition for it:

1. Define a custom **uint64\_t**-like structure:

c

```
typedef struct { uint32_t low; uint32_t high; } uint64_t_custom;
```

2. Implement a function to add two custom `uint64_t`-like integers:

c

```
uint64_t_custom add_uint64_t_custom(uint64_t_custom a, uint64_t_custom b) { uint64_t_custom result; uint32_t low = a.low + b.low; uint32_t high = a.high + b.high + carry; return result; }
```

This function adds the **low** parts of **a** and **b**, calculates the carry (if any), and then adds the **high** parts of **a** and **b** along with the carry.

Please note that this is just a basic example to illustrate the concept. You will need to implement other arithmetic operations and handle signed integers. Implementing arithmetic operations in this way will likely be slower than using native support for 64-bit integers in compilers that support them.



say I have named such type TU64

If you have defined the custom 64-bit unsigned integer type as **TU64**, you can use it in a similar way to the example I provided earlier. Here's the updated example:

1. Define the custom **TU64** structure:

c

```
typedef struct { uint32_t low; uint32_t high; } TU64;
```

2. Implement a function to add two custom **TU64** integers:

c

```
TU64 add_TU64(TU64 a, TU64 b) { TU64 result; uint32_t low = a.low + b.low; uint32_t carry = (low < a.low) ? 1 : 0; result.low = low; result.high = a.high + b.high + carry; return result; }
```

This function adds the **low** parts of **a** and **b**, calculates the carry (if any), and then adds the **high** parts of **a** and **b** along with the carry.

As before, keep in mind that this is a basic example to illustrate the concept. You will need to implement other arithmetic operations and handle signed integers. Implementing arithmetic operations in this way will likely be slower than using native support for 64-bit integers in compilers that support them.



say uint32\_t is TU32 and int32\_t is TS32 and int64\_t is TS64

If you have defined the custom types as **TU32**, **TS32**, and **TS64**, you can follow the same approach as before with the updated type names:

1. Define the custom **TU64** (unsigned) and **TS64** (signed) structures:

c

```
typedef struct { TU32 low; TS32 high; } TU64; typedef struct { TU32 low; TS32 high; } TS64;
```

2. Implement a function to add two custom **TU64** integers:

c

```
TU64 add_TU64(TU64 a, TU64 b) { TU64 result; TU32 low = a.low + b.low; TU32 carry = (low < a.low) ? 1 : 0; result.low = low; result.h
```

This function adds the **low** parts of **a** and **b**, calculates the carry (if any), and then adds the **high** parts of **a** and **b** along with the carry.

As mentioned previously, this is a basic example to illustrate the concept. You will need to implement other arithmetic operations and handle signed implementing arithmetic operations in this way will likely be slower than using native support for 64-bit integers in compilers that support them.



and now I want to implement multiplication of 2 TS32 into one TS64 above, howto?

To implement the multiplication of two **TS32** integers into a single **TS64** result, you can follow these steps:

1. Convert the signed 32-bit integers into unsigned 32-bit integers.
2. Perform the unsigned multiplication using a custom function that takes two **TU32** integers and returns a **TU64** result.
3. Convert the unsigned 64-bit result back into a signed 64-bit result.

Here's an example implementation:

c

```
// Function to multiply two unsigned 32-bit integers and return a 64-bit result TU64 mul_TU32(TU32 a, TU32 b) { TU64 result; TU32 a_low = a << 16; TU32 a_high = a >> 16; TU32 b_low = b << 16; TU32 b_high = b >> 16; TU32 low_low = a_low * b_low; TU32 low_high = a_low * b_high; TU32 high_low = a_high * b_low; TU32 high_high = a_high * b_high; TU32 mid1 = low_high << 16; TU32 mid2 = high_low << 16; result.low += mid1; if (result.low < mid1) { result.high++; } result.low += mid2; result.low += low_high >> 16; result.high += high_high >> 16; return result; } // Function to multiply two signed 32-bit integers and return a 64-bit result TS64 mul_TS32(TS32 a, TS32 b) { TU32 a_unsigned = (TU32)a; TU32 b_unsigned = (TU32)b; TU64 mul_unsigned = mul_TU32(a_unsigned, b_unsigned); result.low = mul_unsigned.low; result.high -= b_unsigned; if (b < 0) { result.high += a_unsigned; } return result; }
```

The **mul\_TS32** function first converts the input **TS32** integers into **TU32** by casting them. Then it calls the **mul\_TU32** function to perform the unsigned multiplication of the input integers.



is this really correct?

```
typedef struct {
    TU32 low;
    TS32 high;
} TU64;
```

```
typedef struct {
    TU32 low;
    TS32 high;
} TS64;
```

Apologies for the oversight in my previous response. You are correct that the structure definition for **TS64** should have a signed **high** part. Here's the

c

```
typedef struct { TU32 low; TU32 high; } TU64; typedef struct { TU32 low; TS32 high; } TS64;
```

With this corrected definition, the **TU64** structure represents a 64-bit unsigned integer, and the **TS64** structure represents a 64-bit signed integer.



isnt really standard way to support int64 in cc65?

In cc65, the standard way to work with 64-bit integers is to use the **long long** type for signed integers and **unsigned long long** type for unsigned integers. To provide native support for 64-bit arithmetic operations, and using these types may lead to cumbersome and slow code when implementing arithmetic operations.

Here's an example of using the **long long** and **unsigned long long** types in cc65:

c

```
#include <stdio.h> int main() { long long a = -123456789012345LL; long long b = 98765432109876LL; long long c = a + b; unsigned long long ua = a; unsigned long long ub = b; unsigned long long uc = ua + ub; printf("a + b = %lld\n", c); printf("ua + ub = %llu\n", uc); return 0; }
```

If you decide to use the **long long** and **unsigned long long** types, you'll need to implement arithmetic operations (addition, subtraction, multiplication, division). This manual implementation will likely be slower than using native support for 64-bit integers in compilers that support them.



well, so it means that I can implement 32bit multiply for cc65 this way for all variants of riscv instructions mul, mulh, mulhu, mulhsu easily?

```
typedef struct {
    TU32 low;
    TU32 high;
} TU64;
```

```
typedef struct {
    TU32 low;
    TS32 high;
} TS64;
```

// Function to multiply two unsigned 32-bit integers and return a 64-bit result

```
TU64 mul_TU32(TU32 a, TU32 b) {
    TU64 result;
    TU32 a_low = a & 0xFFFF;
    TU32 a_high = a >> 16;
    TU32 b_low = b & 0xFFFF;
    TU32 b_high = b >> 16;

    TU32 low_low = a_low * b_low;
    TU32 low_high = a_low * b_high;
    TU32 high_low = a_high * b_low;
    TU32 high_high = a_high * b_high;

    result.low = low_low;
    result.high = high_high;

    TU32 mid1 = low_high << 16;
    TU32 mid2 = high_low << 16;

    result.low += mid1;
    if (result.low < mid1) {
        result.high++;
    }
}
```

```

    result.low += mid2;
    if (result.low < mid2) {
        result.high++;
    }

    result.high += low_high >> 16;
    result.high += high_low >> 16;

    return result;
}

// Function to multiply two signed 32-bit integers and return a 64-bit result
TS64 mul_TS32(TS32 a, TS32 b) {
    TS64 result;
    TU32 a_unsigned = (TU32)a;
    TU32 b_unsigned = (TU32)b;
    TU64 mul_unsigned = mul_TU32(a_unsigned, b_unsigned);
    result.low = mul_unsigned.low;
    result.high = mul_unsigned.high;

    // Adjust the sign
    if (a < 0) {
        result.high -= b_unsigned;
    }
    if (b < 0) {
        result.high -= a_unsigned;
    }

    return result;
}

```

Yes, using the provided implementation, you can implement the RISC-V instructions **mul**, **mulh**, **mulhu**, and **mulhsu** for 32-bit multiplication in cc65.

Here's how you can implement each instruction using the provided functions:

1. **mul**: Multiply two signed 32-bit integers and return the lower 32-bit result.

```

c
TS32 mul(TS32 a, TS32 b) { TS64 result = mul_TS32(a, b); return (TS32)result.low; }

```

2. **mulh**: Multiply two signed 32-bit integers and return the upper 32-bit result.

```

c
TS32 mulh(TS32 a, TS32 b) { TS64 result = mul_TS32(a, b); return result.high; }

```

3. **mulhu**: Multiply two unsigned 32-bit integers and return the upper 32-bit result.

```

c
TU32 mulhu(TU32 a, TU32 b) { TU64 result = mul_TU32(a, b); return result.high; }

```

4. **mulhsu**: Multiply a signed 32-bit integer by an unsigned 32-bit integer and return the upper 32-bit result.

c

```
TS32 mulhsu(TS32 a, TU32 b) { TS64 result = mul_TS32(a, (TS32)b); return result.high; }
```

With these implementations, you can handle the required RISC-V multiplication instructions in cc65 using the provided 32-bit multiplication function



can you please rename there TU64 for TUX64 and TS64 for TSX64 ?

```
typedef struct {
    TU32 low;
    TU32 high;
} TU64;

typedef struct {
    TU32 low;
    TS32 high;
} TS64;

// Function to multiply two unsigned 32-bit integers and return a 64-bit result
TU64 mul_TU32(TU32 a, TU32 b) {
    TU64 result;
    TU32 a_low = a & 0xFFFF;
    TU32 a_high = a >> 16;
    TU32 b_low = b & 0xFFFF;
    TU32 b_high = b >> 16;

    TU32 low_low = a_low * b_low;
    TU32 low_high = a_low * b_high;
    TU32 high_low = a_high * b_low;
    TU32 high_high = a_high * b_high;

    result.low = low_low;
    result.high = high_high;

    TU32 mid1 = low_high << 16;
    TU32 mid2 = high_low << 16;

    result.low += mid1;
    if (result.low < mid1) {
        result.high++;
    }

    result.low += mid2;
    if (result.low < mid2) {
        result.high++;
    }

    result.high += low_high >> 16;
    result.high += high_low >> 16;

    return result;
}

// Function to multiply two signed 32-bit integers and return a 64-bit result
TS64 mul_TS32(TS32 a, TS32 b) {
    TS64 result;
    TU32 a_unsigned = (TU32)a;
```



```

TU32 b_unsigned = (TU32)b;
TU64 mul_unsigned = mul_TU32(a_unsigned, b_unsigned);
result.low = mul_unsigned.low;
result.high = mul_unsigned.high;

// Adjust the sign
if (a < 0) {
    result.high -= b_unsigned;
}
if (b < 0) {
    result.high -= a_unsigned;
}

return result;
}

```

```

TS32 mul(TS32 a, TS32 b) {
    TS64 result = mul_TS32(a, b);
    return (TS32)result.low;
}

```

```

TS32 mulh(TS32 a, TS32 b) {
    TS64 result = mul_TS32(a, b);
    return result.high;
}

```

```

TU32 mulhu(TU32 a, TU32 b) {
    TU64 result = mul_TU32(a, b);
    return result.high;
}

```

```

TS32 mulhsu(TS32 a, TU32 b) {
    TS64 result = mul_TS32(a, (TS32)b);
    return result.high;
}

```

---

As requested, I've renamed the **TU64** and **TS64** structs to **TUX64** and **TSX64**, respectively:

c

```

typedef struct { TU32 low; TU32 high; } TUX64; typedef struct { TU32 low; TS32 high; } TSX64; // Function to multiply two unsigned 32
b) { TUX64 result; TU32 a_low = a & 0xFFFF; TU32 a_high = a >> 16; TU32 b_low = b & 0xFFFF; TU32 b_high = b >> 16; TU32 low_low = a_l
b_low; TU32 high_high = a_high * b_high; result.low = low_low; result.high = high_high; TU32 mid1 = low_high << 16; TU32 mid2 = high_
result.high++; } result.low += mid2; if (result.low < mid2) { result.high++; } result.high += low_high >> 16; result.high

```



can you please rename there TU64 for TU64X and TS64 for TS64X ?

```

typedef struct {
    TU32 low;
    TU32 high;
} TU64;

```

```

typedef struct {
    TU32 low;
    TS32 high;
} TS64;

// Function to multiply two unsigned 32-bit integers and return a 64-bit result
TU64 mul_TU32(TU32 a, TU32 b) {
    TU64 result;
    TU32 a_low = a & 0xFFFF;
    TU32 a_high = a >> 16;
    TU32 b_low = b & 0xFFFF;
    TU32 b_high = b >> 16;

    TU32 low_low = a_low * b_low;
    TU32 low_high = a_low * b_high;
    TU32 high_low = a_high * b_low;
    TU32 high_high = a_high * b_high;

    result.low = low_low;
    result.high = high_high;

    TU32 mid1 = low_high << 16;
    TU32 mid2 = high_low << 16;

    result.low += mid1;
    if (result.low < mid1) {
        result.high++;
    }

    result.low += mid2;
    if (result.low < mid2) {
        result.high++;
    }

    result.high += low_high >> 16;
    result.high += high_low >> 16;

    return result;
}

// Function to multiply two signed 32-bit integers and return a 64-bit result
TS64 mul_TS32(TS32 a, TS32 b) {
    TS64 result;
    TU32 a_unsigned = (TU32)a;
    TU32 b_unsigned = (TU32)b;
    TU64 mul_unsigned = mul_TU32(a_unsigned, b_unsigned);
    result.low = mul_unsigned.low;
    result.high = mul_unsigned.high;

    // Adjust the sign
    if (a < 0) {
        result.high -= b_unsigned;
    }
    if (b < 0) {
        result.high -= a_unsigned;
    }

    return result;
}

```

```

TS32 mul(TS32 a, TS32 b) {
    TS64 result = mul_TS32(a, b);
    return (TS32)result.low;
}

TS32 mulh(TS32 a, TS32 b) {
    TS64 result = mul_TS32(a, b);
    return result.high;
}

TU32 mulhu(TU32 a, TU32 b) {
    TU64 result = mul_TU32(a, b);
    return result.high;
}

TS32 mulhsu(TS32 a, TU32 b) {
    TS64 result = mul_TS32(a, (TS32)b);
    return result.high;
}

```

---

As requested, I've renamed the **TU64** and **TS64** structs to **TU64X** and **TS64X**, respectively:

c

```

typedef struct { TU32 low; TU32 high; } TU64X; typedef struct { TU32 low; TS32 high; } TS64X; // Function to multiply two unsigned 32
b) { TU64X result; TU32 a_low = a & 0xFFFF; TU32 a_high = a >> 16; TU32 b_low = b & 0xFFFF; TU32 b_high = b >> 16; TU32 low_low = a_l
b_low; TU32 high_high = a_high * b_high; result.low = low_low; result.high = high_high; TU32 mid1 = low_high << 16; TU32 mid2 = high_
result.high++; } result.low += mid2; if (result.low < mid2) { result.high++; } result.high += low_high >> 16; result.high += high_low
integers and return a 64-bit result TS64X mul_TS32(TS32 a, TS32 b) { TS64X result; TU32 a_unsigned = (TU32)a; TU32 b_unsigned = (TU32
result.low = mul_unsigned.low; result.high = mul_unsigned.high; // Adjust the sign if (a < 0) { result.high -= b_unsigned; } if (b <
TS32 b) { TS64X result = mul_TS32(a, b); return (TS32)result.low; } TS32 mulh(TS32 a, TS32 b) { TS64X result = mul_TS32(a, b); return
mul_TU32(a, b); return result.high; } TS32 mulhsu(TS32 a, TU32 b) { TS64X result = mul_TS32(a, (TS32)b); return result.high; }

```

---