


Exercise 5: Compiler Usage to Generate Assembly Code

In this exercise you will use a C compiler to generate RISC-V assembly code.

Before starting do the following:

1. `cesp_setup` 
2. `git pull` (if you have conflicts, backup the your files and do `git stash`)
3. `cd $HOME/cesp_course/exercises/05_compiler`

Task 1. Inline Assembly

Steps:

1. Open Compiler Explorer in your web browser <https://godbolt.org/>
2. Set compiler to **RISC-V rv32gc clang (trunk)** and compiler options to **-O0** as shown in Figure 1.
3. Copy the C code from Listing 1 in the C-Code field (see also `play_sound.c` in directory `05_compiler`).
4. Complete function `play_sound` to play a sound with parameters `pitch`, `duration` `instrument` and `volume` using RARS ecalls:
 - Use the `MidiOutSync` ecall
 - Set the required registers using inline assembly. See Listing 2 for a reference of inline assembly in C.
5. Copy the resulting RISC-V assembly code to RARS. You might have to do some minor modifications to get it running. Hint: labels and sections.
6. Check the functionality: You should hear sound on your laptop's speakers

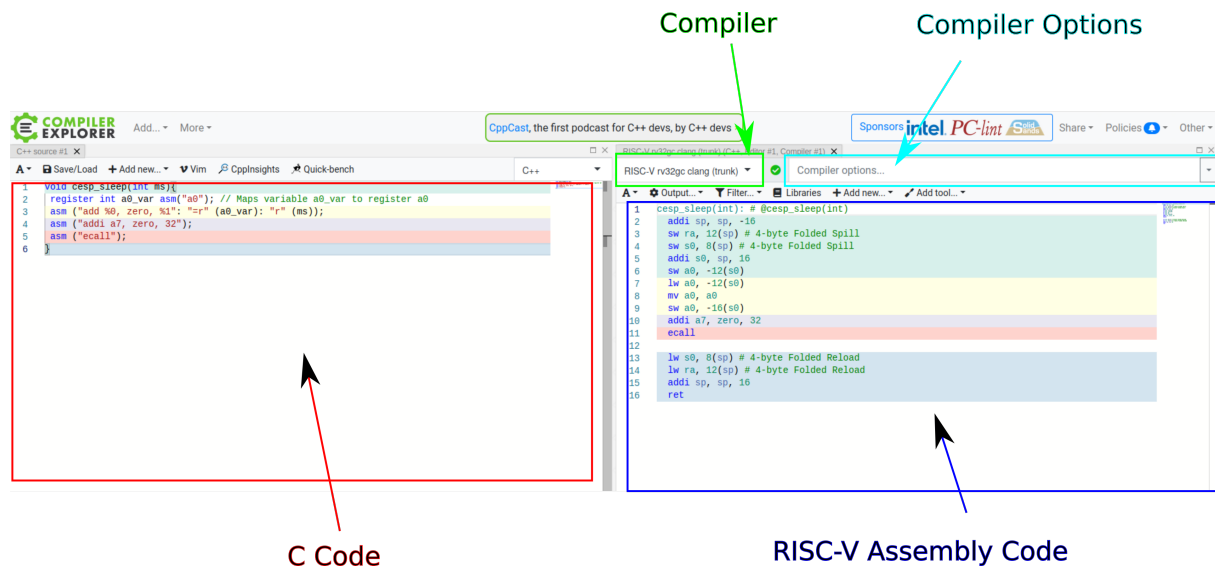


Figure 1: Compiler Explorer.

```

1 void play_sound(int pitch, int duration){
2     int instrument = 1;
3     int volume = 127;
4     ...
5     asm ("ecall");
6 }
7
8
9
10 int main(){
11     const int e = 64;
12     const int g = 67;
13     const int b = 71;
14     const int c = 72;
15     const int p = 0;
16
17     int pitches[8] = {e, g, b, c, b, g, e, p};
18     int duration[8] = {400, 400, 400, 400, 400, 400, 400, 30};
19     for(int i = 0; i < 64; i++)
20         play_sound(pitches[i%8], duration[i%8]);
21     return 0;
22 }

```

Listing 1: Main function to play sound theme.

Task 2. Optimization

Compiler optimization level can make your program run faster. Here we will explore what effects these optimizations might have and what you need to keep in mind when writing low-level C code.

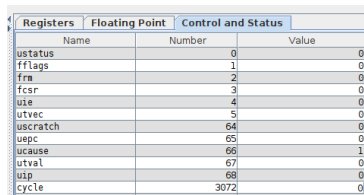
Steps:

1. Try compiling your program with compiler optimizations: To do this add the arguments `-O1`, `-O2` or `-O3` in the compiler options field of Compiler Explorer.
2. Run your optimized code and unoptimized code in RARS. Only continue if both work as expected.
Hint: If you have a problems executing the optimize program, consider using the `volatile` keyword for inline assembly as shown in Listing 2.
3. Analyze the differences on assembly code level between the optimized and unoptimized version of your code.

Task 3. Performance Comparison

Determine the number of execution cycles of the optimized and unoptimized version of your RISC-V assembly code and calculate the speedup factor that is achieved by compiler optimizations.

Hint: You can find the number of executed cycles in the `cycles` register in RARS.



Registers		
Name	Number	Value
ustatus	0	0
fflags	1	0
ffrm	2	0
fcsr	3	0
use	4	0
utvec	5	0
uscratch	64	0
uopc	65	0
ucause	66	1
utval	67	0
uip	68	0
cycles	3072	0

```
1 // General usage of inline assembly in C
2 asm [ volatile ] ( AssemblyTemplate : OutputOperands : InputOperands [ :
   ClobberList ] );
3 //
4 // The clobber list prevents the compiler from using registers.
5 // The volatile keyword prevent the compiler to apply optimizations
6
7 //Example
8 register int out_var asm("a2");
9 int in_var; //variable in_var is mapped by compiler to a register X
10 asm ( "mv %0, %1" : "=r" (out_var) : "r" (in_var) : "a0", "a1", "a3" );
11 // With the clobber list we tell the compiler explicitly NOT to use a0, a1
   or a3 for anything else, e.g. for loading the content of variable in_var
   or an offset to the address of out_var in the memory.
12
13 // This example is translated to:
14 mv a0, X
```

Listing 2: Inline Assembly in C.