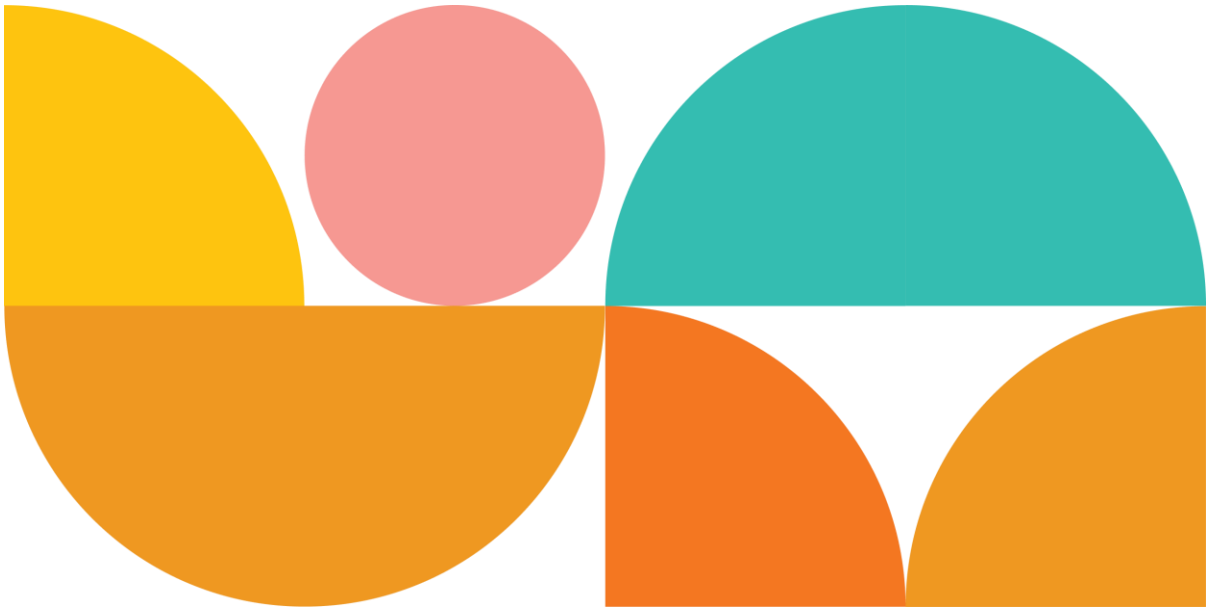
31st March 2024

CPL assignment 4

Analysing the assembled output of a C source Code



Aagam Sancheti BT22CSE116

The high-Level Language C Source Code

The C code mentioned here is a high-level C-language code that calculates factorial of a Number using Recursion and Loops.

Subsequently the code is converted in all the intermediate steps a compiler would take while generating an executable. The underlying code is explicitly attached by me, clearly stating what command lines were used in my System to obtain that intermediate representation.

Overview:

Steps taken by compiler to generate executable binary:

Preprocessing:

This stage involves handling preprocessor directives, such as **#include**, **#define**, and **#ifdef**.

Also, during this stage, MACROS are expanded, header files are included, and comments are removed from the code.

The output of this stage is a modified version of the original source code, known as a translation unit.

Compilation:

At this stage, the pre-processed source code is translated into assembly code or an intermediate representation (IR) such as GIMPLE.
The compiler performs syntax analysis, semantic analysis, and type checking to ensure the correctness of the code. (compile time error detection, that's what we call it as generally)

Optimization flags specified by the user or set by default are applied to optimize the generated code. (for example, optimization flags include **-O0**, **-O1**, **-O2**, **-O3**, **-Os**, **-Og**, and **-Ofast**. These flags control the level of optimization applied by the compiler. These aims to improve performance, memory usage or other characteristics by default or as specified by user.

Optimization can include techniques such as constant folding, loop unrolling, function inlining, dead code elimination, and many others. (more or less the same way the garbage collector functions).

The output of this stage is typically an assembly file or an intermediate representation file (.s, .gimple, or .o). (we'll use and explore the .s format, as gimple is complex and .s is more human understandable).

Assembly of code by Assembler:

The assembler translates the assembly code into machine code specific to the target architecture.

It generates an object file (.o) containing machine code and metadata.

The object file may also contain relocation information and symbol tables.

Linking:

In this stage, the linker combines one or more object files along with any necessary libraries to create an executable binary.
It resolves external references, such as function calls and variable definitions, by finding and linking them to their respective locations.

For example:

- If a source file contains a function call to **printf**, which is defined in the standard C library (**libc**), the linker will locate the definition of **printf** in the **libc** library and link it to the function call in the program.
- Similarly, if a program uses a global variable that is defined in another source file or library, the linker will locate the definition of that variable and link it to the references in the program.

The linker also performs additional tasks like generating startup code, setting up the program's memory layout, and creating the final executable file.

The output of this stage is an executable binary file. (depends on the OS system, .exe for windows & .out on UNIX-like operating systems, i.e. it is hardware specific)

Loading (Optional):

On some systems, an additional loading stage may be necessary if the executable binary is not directly executable. (due to difference in system hardware architecture).

The loader loads the executable binary into memory and prepares it for execution by the CPU.

This step is typically handled by the operating system and is transparent to the user.

Sample Code:

```c
// standard high level language C source Code

#include <stdio.h>

int factorial_recursion(int n);
int factorial_loops(int n);

int main()
{
int n;
printf("Enter n of which factorial has to be calculated:\n");
scanf("%d", &n);
printf("Factorial using recursion is : %d\n", factorial_recursion(n));
factorial_loops(n);
}

int factorial_recursion(int n)
{
if (n == 1 || n == 0)
{
return 1;
}
int factorialNminus1 = factorial_recursion(n - 1);
int factorial = factorialNminus1 * n;
return factorial;
}

int factorial_loops(int n)
{
int factorial = 1;
for (int i = n; i >= 1; i--)
```

```
{
factorial = factorial * i;
}
printf("Factorial using loops is: %d\n", factorial);

return 0;
}
```

Pre-processed Code:

This preprocessed code generated by the IDE (Visual Studio Code) could be obtained by the following command:

"gcc-13 -E CPL_assignment_4.c -o preprocessed_file.i"

Here,
-E flag: used to obtain a Preprocessed file.
The format of this file is "filename.i"

https://drive.google.com/file/d/1IbUNNXBnzKzv_y4-eL7Eb8cL0WsiHjZZ/view?usp=drive_link

Compiled code (assembly code, .s) with different degree of optimisations:

1) **No optimisation:** "gcc -00 -S CPL_assignment_4.c -o assembly_file_00.s"

-S flag: used to specify the compiler to output assembly file rather than executable file.
-O0: degree of optimisation
(Files are attached within the email)

2) **Degree 1 optimisation:** "gcc -O1 -S CPL_assignment_4.c -o assembly_file_O1.s"

-O1: Degree of optimisation for speed of program

3) Degree 2 optimisation: "gcc -O2 -S CPL_assignment_4.c -o assembly_file_O2.s"

4) Degree 3 optimisation: "gcc -O3 -S CPL_assignment_4.c -o assembly_file_O3.s"

-O4: Degree of optimisation for maxer speed possible by compilation stage.

5) Optimisation for Size & Space efficiency: "gcc -Os -S CPL_assignment_4.c -o assembly_file_Os.s"

-Os: Optimisation for smaller code size, aiming to produce smaller executables.

6) Fast and aggressive optimisation: "gcc -Ofast -S CPL_assignment_4.c -o assembly_file_Ofast.s"

-Ofast: Compile with aggressive optimization. Fastest possible optimisation for speed.

Analyzation of Gimple Output:

Non-Optimised Gimple representation of above code:

Command: "gcc-13 CPL_assignment_4.c -fdump-tree-gimple -O0 -o gimple_O0.gimple"

Only changes made by gimple from the original high level code, are commented and explained.

<-->

```
int main ()
{
  int D.4868;      // integer variable D.4868

  {
    int n;    // nested int variable n

    try     // error handling implemented by compiler
      {
        printf ("Enter n of which factorial has to be
calculated :\n");
        scanf ("%d", &n);
        n.0_1 = n;
        _2 = factorial_recursion (n.0_1);
 // factorial recursion function is called
        printf ("Factorial using recursion
is : %d\n", _2);
```

```
        n.1_3 = n;
        factorial_loops (n.1_3); // Factorial loops
is called
      }
    finally
      {
        n = {CLOBBER(eol)};

// Clobbering is CPU register optimisation keyword.
clobber" refers to invalidating or destroying the
contents of a register or memory location
      }
  }
  D.4868 = 0;
  return D.4868;
}


int factorial_recursion (int n)
{
  int D.4873;
  int factorialNminus1;
  int factorial;

  if (n == 1) goto <D.4870>; else goto <D.4872>;
 //Gimple conversion of the if-else statement.
Observe carefully the difference. Amazing!

  <D.4872>:
  if (n == 0) goto <D.4870>; else goto <D.4871>;
  <D.4870>:
  D.4873 = 1;
  // predicted unlikely by early return (on trees)
predictor.
  return D.4873;
  <D.4871>:
  _1 = n + -1;
```

```
  factorialNminus1 = factorial_recursion (_1);
  factorial = factorialNminus1 * n;
  D.4873 = factorial;
  return D.4873;
}


int factorial_loops (int n)
{
  int D.4875;
  int factorial;

  factorial = 1;
  {
    int i;

    i = n;
    goto <D.4866>;
    <D.4865>:
    factorial = factorial * i;
    i = i + -1;
    <D.4866>:
    if (i > 0) goto <D.4865>; else goto <D.4863>;
//implementation of loops using goto statements!
    <D.4863>:
  }
  printf ("Factorial using loops is: %d\n",
factorial);
  D.4875 = 0;
  return D.4875;
}
```

<--->

Optimised Gimple representation of above code:

Command: “gcc-13 CPL_assignment_4.c -fdump-tree-gimple -Ofast -o gimple_Ofast.gimple”

The optimized code is very similar, the only change is:

- In **factorial_recursion()**, the condition **if (n <= 1)** has been replaced with **(unsigned int) n <= 1**. This is an optimization to leverage unsigned integer comparison, which might be more efficient on some architectures.
- The variable **_2** has been replaced with **D.4872** in **factorial_recursion()**, maintaining consistency with the rest of the code.

<--->

```
int factorial_recursion (int n)
{
  int D.4872; //renaming of variable from D.4873 to
D.4872 to adhere optimisation
  int factorialNminus1;
  int factorial;

  n.2_1 = (unsigned int) n; // condition has been
changed
  if (n.2_1 <= 1) goto <D.4870>; else goto <D.4871>;
  <D.4870>:
  D.4872 = 1;
  // predicted unlikely by early return (on trees)
predictor.
  return D.4872;
  <D.4871>:
  _2 = n + -1;
  factorialNminus1 = factorial_recursion (_2);
  factorial = factorialNminus1 * n;
  D.4872 = factorial;
  return D.4872;
}
```

<--->

Analyzation and Comparison of Assembly Output:

Non-Optimised Assembly code:

Please open the drive link to view unoptimized assembly code.

https://drive.google.com/file/d/1oaHlG29sf6zjZmeaXbhxQaG2gurlAG9X/view?usp=drive_link

Optimised Assembly code:

Please open the drive link to view fast, optimised assembly code.

https://drive.google.com/file/d/1Qcr0UeknvqYM-QysoA3G4l_6nTHthQsH/view?usp=drive_link

The differences are commented in the assembly code itself, but some major observations and differences are as follows:

- **Reduction in Number of Function Calls**: The biggest difference I noticed is significant reduction in number of function calls! The major optimization techniques involve avoiding function calls by inlining the code of small functions directly into the main function. This can greatly reduce the overhead associated with function calls.
- **Removal of Redundant Code**: The optimized code removes redundant or unnecessary code, which can improve performance and reduce code size.
- **Removal of Unwanted Exception Handling**: Exception handling code that was not necessary, is removed. It was added by compiler itself to make code very robust and handle all corner cases, but then to increase speed, it removed some of it.
- **Increased Space Requirement**: While the optimized code may reduce function calls and improve performance, it may require more space in terms of memory consumption and code size due to inlining and other optimizations. This is the

downfall that comes with increasing speed! The trade-off between time and space is eternal.

- **Increased Register Requirement**: Optimized code requires more registers to store intermediate values and optimize calculations. This makes system use more resources and registers are not available for other tasks.
- **Exception Handling**: The code still includes exception handling to manage and avoid unwanted outcomes. Compiler puts it by default.
- **Inlining Functions into Main**: Other major changes that I noticed was, All the functions seem to be added to the main function itself in the code, making them inline functions. This can lead to better performance due to reduced overhead of function calls, but space is required more.

<----------The End----------
----->