# Atividade 2 - Org. Arg. 1

**Profa. Dra. Cíntia Borges Margi**
**(cintia@usp.br)**

**Guilherme S. Salustiano**
**(salustiano@usp.br)**

## Parte 1 - Instruções e chamadas de funções

Na primeira parte dessa atividade explorar as instruções geradas por algumas funções pelo compilador, observando chamadas de funções e como isso se relaciona com a pilha.

### Contexto

Os computadores entendem e executam um conjunto bem restrito de instruções, com a evolução da computação buscou-se criar maneiras mais simples e expressivas de se criar código, e assim surgem as primeiras linguagens de programação de alto nivel que são compiladas para as instruções que o processador entende.

### Analisando o disassembly de funções

Vamos usar o godbolt.org, um compilador online que suporta diferentes arquiteturas. Ao abrir o site você encontra dois principais painéis conforme a Figura 1. Na esquerda você encontra um editor de texto, no menu superior selecione a linguagem “C”. Na direita você pode observar as instruções em assembly que foram gerado pelo código da esquerda, no menu superior escolha o compilador “RISC-V (64-bits) gcc 13.2.0”, também temos a opção de passar flags para o compilador na caixa de texto do canto superior direito que iremos usar mais adiante.

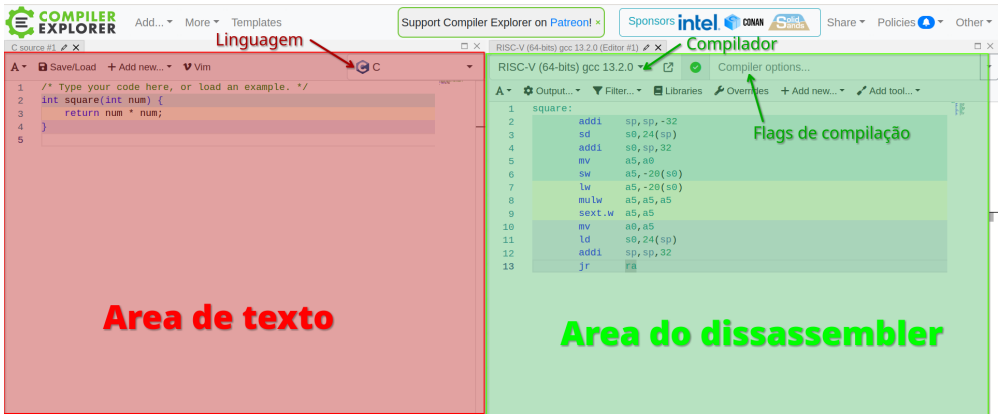


Figura 1: Visão geral do godbolt.org

#### Soma simples

Escreva o seguinte código no editor. O campo de flags no canto superior direito deve estar vazio.

```
int sum(int a, int b) {
  return a + b;
}
```

*Quais operações você consegue identificar?*

*Consegue visualizar alguma melhoria possível?*

#### Soma simples otimizado

Agora vamos adicionar a flag `-O1` no compilador com o mesmo código do exemplo anterior, por padrão o compilador não otimiza o código (`-O0`).

Ligar uma flag de otimização faz com que o compilador tente melhorar o desempenho e/ou o tamanho do código em detrimento do tempo de compilação e possivelmente da capacidade de depurar o programa.

Você pode ler mais sobre as flags e otimizações do gcc aqui.

*O que mudou?*

*Porque por padrão o compilador não otimiza o código?*

**Fatorial recursivo sem otimização**
Agora substitua o código do editor pelo código abaixo e também apague as flags de compilação.

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

*Quais as novas operações que você consegue identificar?*

*Consegue visualizar alguma melhoria possível?*

**Fatorial recursivo com `-O1`**
Agora vamos adicionar a flag `-O1` no compilador com o mesmo código do exemplo anterior.

*O que mudou?*

*Porque ele continua a usar a stack?*

*Consegue visualizar alguma melhoria possível?*

**Fatorial recursivo com `-O2`**
Agora vamos adicionar a flag `-O2` no compilador com o mesmo código do exemplo anterior.

*O que mudou?*

# Parte 2 - Benchmark de linguagens e flags de compilação

Nessa parte vamos executar o mesmo código da atividade 1, o conjunto de mandelbrot, em diferentes linguagens e comparar seu desempenho.

## Requisitos

Primeiramente baixe o os códigos.

Nessa parte iremos usar novamente o perf, gcc e make. Caso você ainda não tenha instalado siga as instruções disponivel na atividade 1

Além disso vamos precisar do pypy e do java, se você ainda não tiver eles instalados no seu sistema você pode instalar com o comando abaixo:

```
$ sudo apt install pypy openjdk-17-jdk
```

Um container docker também está disponível para uso, basta usar o script `run_by_docker.sh`.

## Executando o código

Abra um terminal na pasta `src/` do código baixado e execute os comandos para cada linguagem.

**C sem otimizações**

```
$ # Compila o código
$  gcc -o mandelbrot_00 -std=c11 -O0 mandelbrot.c -lm

$ # Roda o programa e salva o resultado em um arquivo
$ perf stat -x ';' -r 10 -e cycles,instructions,duration_time ./mandelbrot_00 -2.5 1.5
-2.0 2.0 4096 2>&1 | tee c_O0.csv
```

### C com otimização

```
$ # Compila o código
$ gcc -o mandelbrot_02 -std=c11 -O2 mandelbrot.c -lm

$ # Roda o programa e salva o resultado em um arquivo
$ perf stat -x ';' -r 10 -e cycles,instructions,duration_time ./mandelbrot_02 -2.5 1.5
-2.0 2.0 4096 2>&1 | tee c_O2.csv
```

### Java

```
$ # Compila o código
$ javac mandelbrot.java

$ # Roda o programa e salva o resultado em um arquivo
$ perf stat -x ';' -r 10 -e cycles,instructions,duration_time java Mandelbrot -2.5 1.5
-2.0 2.0 4096 2>&1 | tee java.csv
```

### PyPy

```
$ # Roda o programa e salva o resultado em um arquivo
$ perf stat -x ';' -r 10 -e cycles,instructions,duration_time pypy3 mandelbrot.py -2.5
1.5 -2.0 2.0 4096 2>&1 | tee pypy.csv
```

### CPython

```
$ # Roda o programa e salva o resultado em um arquivo
$ perf stat -x ';' -r 10 -e cycles,instructions,duration_time python3 mandelbrot.py
-2.5 1.5 -2.0 2.0 4096 2>&1 | tee python.csv
```

## Analisando os resultados

Para facilitar a comparação dos resultados vamos usar o script `src/plot.py` que gera um gráfico com os resultados disponíveis na pasta `plots/`.

```
$ python plot.py
```

*Qual a classificação de desempenho de cada linguagem?*

*Como o desempenho de cada linguagem se relaciona ao modo de execução (compilada, interpretada, JIT) dela?*

# Entrega final

Ao final, gere um zip `atv2.zip` com os arquivos.

```
atv2.zip
├── c_O0.csv
├── c_O2.csv
├── java.csv
├── pypy.csv
├── python.csv
```