

Universidade de Brasília
Departamento de Ciência da Computação



Laboratório 1 - Assembly - RISC-V
Organização e Arquitetura de Computadores

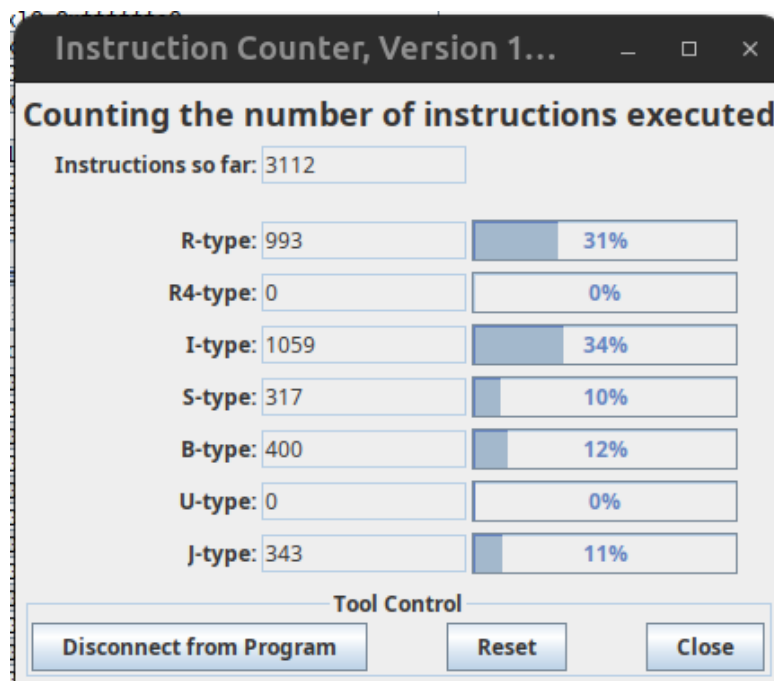
Evelyn Soares Pereira 170102785
André Rodrigues Modesto 211068234
Théo Henrique Gallo 170080781
Eduardo de Souza Costa Assunção 211068270
Alexandre Junqueira Correia Lima 211068225

Brasília, 18 de Maio de 2025

1) Simulador / Montador RARS

1.1 - a) Dado o vetor $V[30]$, o arquivo `sort.s` já faz a sua ordenação em ordem crescente. O algoritmo entra em SORT e salva os registradores, salva em `s2` o endereço base do vetor e depois salva o tamanho do vetor em `s3`. Entra no primeiro loop *for1* e checa se $s0 \geq N$, se não for, entra no segundo loop *for2* e aí começa a ordenação checando se um segundo índice salvo em `s1` é menor que zero, caso não seja, `t3` armazena `vetor[j]` e `t4` armazena `vetor[j+1]`, daí é feita a comparação em `bge t4,t3,exit2`: se $\text{vetor}[j+1] \geq \text{vetor}[j]$, então está em ordem e sai do laço interno, ou seja não faz a troca.

Para saber o número de instruções por tipo e o número total de instruções exigidas pelo procedimento `sort`, podemos ver no RARS que há uma interface que já faz essa conta, colocando breakpoints antes e depois do procedimento SORT, esse foi o resultado:



3112 instruções menos a instrução `jal SORT`, 3111 instruções para o procedimento SORT.

O tamanho em bytes do código executável pode ser medido da seguinte forma. No RARS, a parte `.text` (text segment) do código em assembly indica que estamos começando as instruções, enquanto a parte `.data` indica que estamos usando a memória. Para calcular o tamanho do código executável em bytes, precisamos saber onde fica a primeira instrução. No RARS, a primeira instrução do programa ficou em `0x00400000` e a última ficou em `0x00400034`, logo podemos ver que:

$0x00400034 - 0x00400000 = 0x34 = 52$ bytes é o tamanho em bytes do código executável.

Já para a memória de dados, o vetor declarado tem 30 elementos do tipo word e como estamos trabalhando com um processador de 32 bits, cada word tem 4 bytes. Então:

$30 * 4 = 120$ bytes é o tamanho da memória de dados usada.

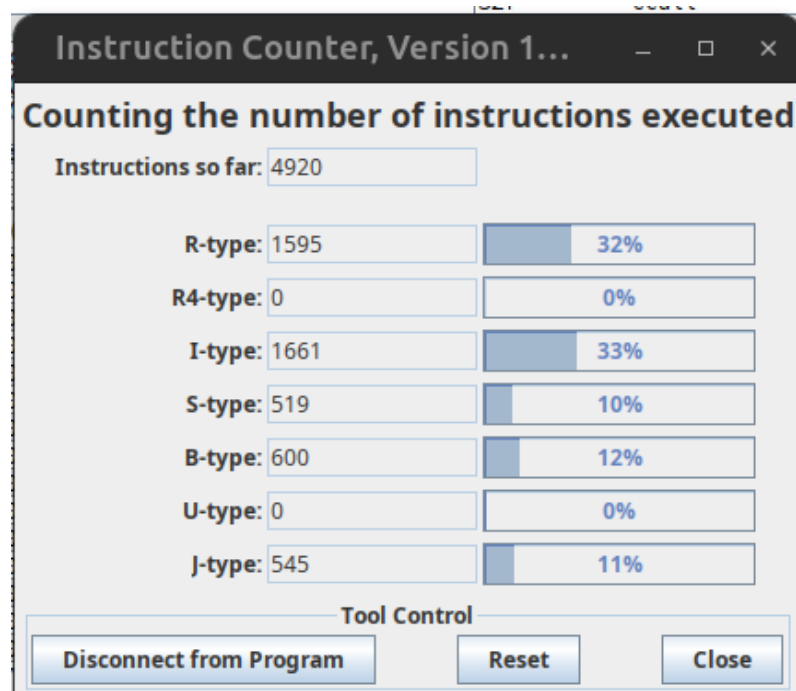
b) Sabemos que o código realiza a ordenação em ordem crescente por conta da seguinte instrução dentro da função SORT: bge t4, t3, exit2

Portanto, a troca só ocorre quando $\text{vetor}[j+1] < \text{vetor}[j]$, o que define uma ordenação crescente. Para inverter essa lógica e ordenar em ordem decrescente, basta substituir bge por ble, ficando: ble t4, t3, exit2

Agora, os elementos só serão trocados quando $\text{vetor}[j+1] > \text{vetor}[j]$, ou seja, fora da ordem decrescente, o resultado fica conforme a imagem abaixo:

Run I/O																															
9	1	2	5	1	8	2	4	3	6	7	10	2	32	54	2	12	6	3	1	78	54	23	1	54	2	65	3	6	55	95	
1	1	1	2	2	2	2	2	3	3	3	4	5	6	6	6	7	8	9	10	12	23	31	32	54	54	54	55	55	65		
-- program is finished running (0) --																															
9	2	5	1	8	2	4	3	6	7	10	2	32	54	2	12	6	3	1	78	54	23	1	54	2	65	3	6	55			
78	65	55	54	54	54	32	31	23	12	10	9	8	7	6	6	6	5	4	3	3	3	2	2	2	2	2	1	1			
-- program is finished running (0) --																															

Após essa troca, a quantidade de instruções totais executadas pode ser medida novamente e é:



4920 instruções menos a instrução jal SORT, 4919 instruções para o procedimento SORT.

c) Para o procedimento de ordenação crescente, usando os contadores de instruções e tempo do Banco de Registradores CSR, descomentando as linhas:

```
csrr s1,3074 # le o num instr atual
csrr s0,3073 # le o time atual

csrr t0,3073 # le o time atual
csrr t1,3074 # le o num instr atual
sub s0,t0,s0 # calcula o tempo de execução Texec em ms
sub s1,t1,s1 # calcula o número de instruções I executadas
```

a quantidade de instruções executadas e o tempo de execução, para a), é esse:

s0	8	0x0000003e
s1	9	0x00000c2a
s2	10	0x00000000

0x00000c2a = 3114 instruções e tempo de 0x0000003e = 62 milissegundos.

Já para o item b) obtemos:

s0	8	0x00000055
s1	9	0x0000133a
s2	10	0x00000000

0x0000133a = 4922 instruções e o tempo de execução de 0x00000055 = 85 milissegundos.

Comparando com o resultado anterior 3114 - 3111 = 3, e 4922 - 4919 = 3.

Essas instruções de leitura e subtração, assim como o salto (jal), também são instruções RISC-V e, portanto, são contadas no total. Isso gera uma diferença de 3 instruções em relação ao número real de instruções do corpo da função SORT.

1.2 - a) A equação básica para estimar o tempo de execução de um programa em um processador é dada por:

$$t_{exec} = I * CPI * T$$

Onde I é a contagem de instruções executadas, CPI é ciclos por instrução, que nesse caso é 1 o que nos diz que o processador executa cada instrução em único

ciclo de clock, e T é o tempo de ciclo de clock que pode ser medido a partir da frequência f , de $50 \text{ MHz} = 50 * 10^6 \text{ Hz}$ então $T = 1/f = 20 \text{ ns}$.

O algoritmo Insertion Sort funciona da seguinte forma, ele pega cada elemento (a partir do segundo) e o insere na posição correta à esquerda, movendo os maiores valores para a direita até encontrar a posição certa. Então podemos notar que estamos trabalhando com um algoritmo de Insertion Sort no arquivo sort.s pois a ordenação começa a partir do segundo elemento do vetor, a comparação é feita com o elemento atual e os anteriores fazendo trocas até encontrar a posição correta e inserir.

Para a contagem de instruções I , vai depender do programa, o procedimento sort, e da entrada que é o tamanho do vetor, então para determinar I , precisamos saber quantas instruções o procedimento sort executa para o tamanho do vetor especificado. Temos:

$I_o(n)$: O número total de instruções executadas quando a entrada é o vetor já ordenado $Vo[n]$.

$I_i(n)$: O número total de instruções executadas quando a entrada é o vetor ordenado inversamente $Vi[n]$.

Baseado na teoria de complexidade de algoritmos, o melhor caso (ordenado) resulta em uma complexidade linear ($O(n)$) para a contagem de instruções, e o pior caso (ordenado inversamente) resulta em uma complexidade quadrática ($O(n^2)$).

Daí já sabemos que para o melhor caso, no algoritmo sort.s, a instrução bge t4, t3, exit2 vai sempre pular a troca, vai só percorrer o laço externo e não vai entrar no laço interno. Já para o pior caso, com o vetor inversamente ordenado, sempre haverá a troca em bge t4, t3, sempre vai entrar sempre no segundo laço, fazendo todas as comparações e trocas possíveis.

Para o Insertion Sort o melhor caso geralmente envolve um número de operações (e instruções) que cresce linearmente com n . Assim, esperamos que $I_o(n)$ seja uma função linear de n , da forma aproximadamente $k1 * n + c1$, onde $k1$ e $c1$ são constantes determinadas pelo código Assembly. E para o caso $Vi[n]$ (vetor ordenado inversamente), este sendo o "pior caso" pois exige o número máximo de operações (e instruções), envolve um número de operações que cresce quadraticamente com n . Assim, esperamos que $I_i(n)$ seja uma função quadrática de n , da forma aproximadamente $k2 * n^2 + k3 * n + c2$, onde $k2$, $k3$ e $c2$ são constantes determinadas pelo código Assembly.


Utilizando o RARS, foi executado o sort.s para os seguintes valores de $n=\{10, 20, 30\}$ com entradas $Vo[n]$ e $Vi[n]$ e obtivemos as seguintes contagens de instruções:

- $Io(10)$ resultou em 113 instruções
- $Io(20)$ resultou em 213 instruções
- $Io(30)$ resultou em 313 instruções

Já para o vetor ordenado inversamente:

- $Ii(10)$ resultou em 878 instruções
- $Ii(20)$ resultou em 3538 instruções
- $Ii(30)$ resultou em 7998 instruções

Esse procedimento pode ser visto no link:

 UnB – OAC Unificado – 2025-1 – Grupo 4 - Laboratório 1 - Questão 1

Para $Io(n)$, como esperamos uma relação linear, dois pontos são suficientes para encontrar k_1 e c_1 . Usando os pontos (10, 113) e (20, 213), temos que a inclinação k_1 da linha é a variação em Io dividida pela variação em n :

$$(213-113) / (20-10) = 100/10 = 10$$

E para encontrar c_1 , fazemos $113 = 10 \cdot 10 + c_1$, isolando c_1 obtemos $c_1 = 13$.

Para $Ii(n)$ como esperamos uma relação quadrática, precisamos de três pontos para encontrar os três coeficientes k_2 , k_3 , e c_2 .

$$\text{Para } n=10: k_2 * 10^2 + k_3 * 10 + c_2 = 878 \text{ (Equação 1)}$$

$$\text{Para } n=20: k_2 * 20^2 + k_3 * 20 + c_2 = 3538 \text{ (Equação 2)}$$

$$\text{Para } n=30: k_2 * 30^2 + k_3 * 30 + c_2 = 7998 \text{ (Equação 3)}$$

Agora resolvemos este sistema. Subtraindo a Equação 1 da Equação 2: $(400 * k_2 + 20 * k_3 + c_2) - (100 * k_2 + 10 * k_3 + c_2) = 3538 - 878$

$$300 * k_2 + 10 * k_3 = 2660 \text{ (Equação 4)}$$

Subtraindo a Equação 2 da Equação 3: $(900 * k_2 + 30 * k_3 + c_2) - (400 * k_2 + 20 * k_3 + c_2) = 7998 - 3538$

$$500 * k_2 + 10 * k_3 = 4460 \text{ (Equação 5)}$$

Agora temos um sistema de duas equações com duas variáveis (k_2 e k_3).
Subtraindo a Equação 4 da Equação 5: $(500 * k_2 + 10 * k_3) - (300 * k_2 + 10 * k_3) = 4460 - 2660$

$$200 * k_2 = 1800 \Rightarrow k_2 = 1800 / 200 = 9$$

Agora que temos k_2 , e podemos substituir seu valor em uma das equações para encontrar k_3 : $300 * 9 + 10 * k_3 = 2660$

$$2700 + 10 * k_3 = 2660 \Rightarrow 10 * k_3 = 2660 - 2700 = -40$$

$$k_3 = -40 / 10 = -4$$

Finalmente, substituímos os valores de k_2 e k_3 em uma das equações para encontrar c_2 : $100 * 10 + 10 * (-4) + c_2 = 878$

$$1000 - 40 + c_2 = 878 \Rightarrow 960 + c_2 = 878$$

$$c_2 = 878 - 960 = -82$$

Logo:

$$I_o(n) = 10n + 13$$

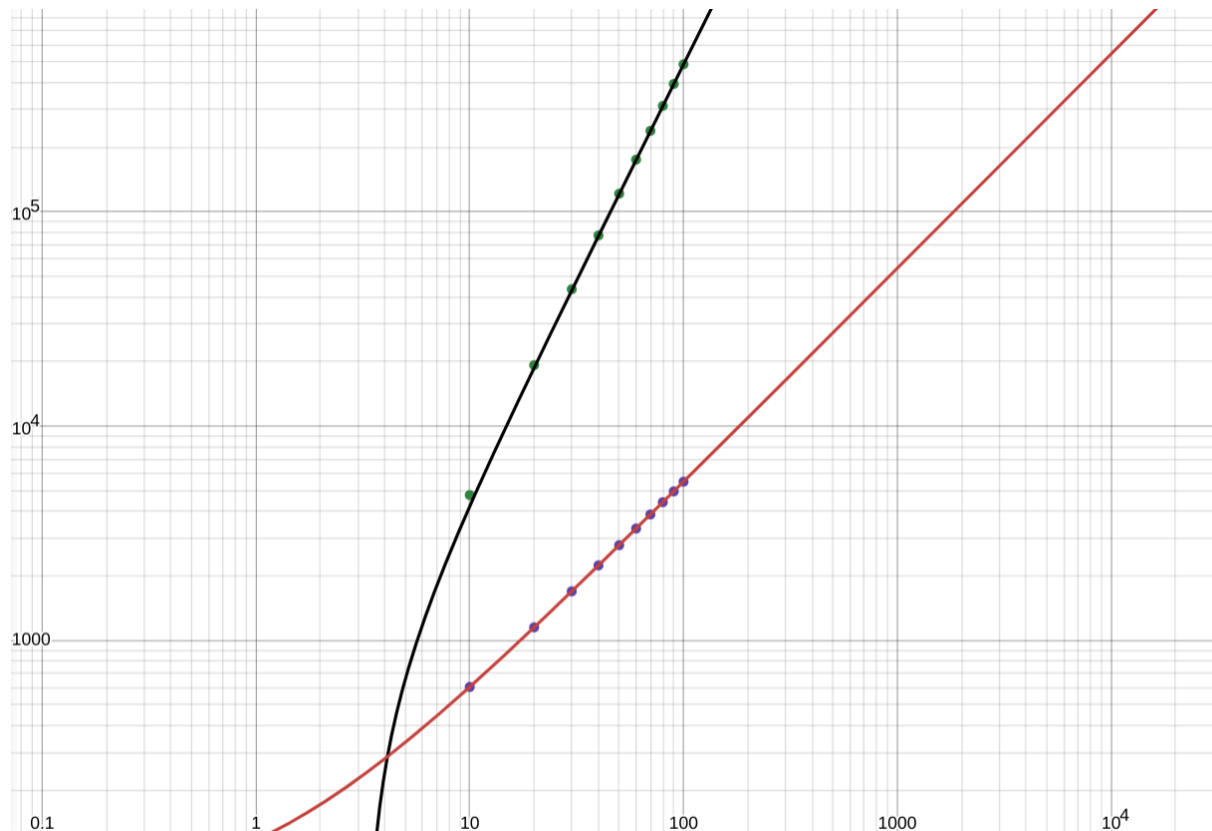
$$I_i(n) = 9n^2 - 4n - 82$$

Então as equações do tempo de execução em função de n são:

$$t_o(n) = (10n + 13) * 2 * 10^{-8}$$

$$t_i(n) = (9n^2 - 4n - 82) * 2 * 10^{-8}$$

b) Podemos plotar um gráfico de $t_o(n)$ e $t_i(n)$ e ele terá essa forma:

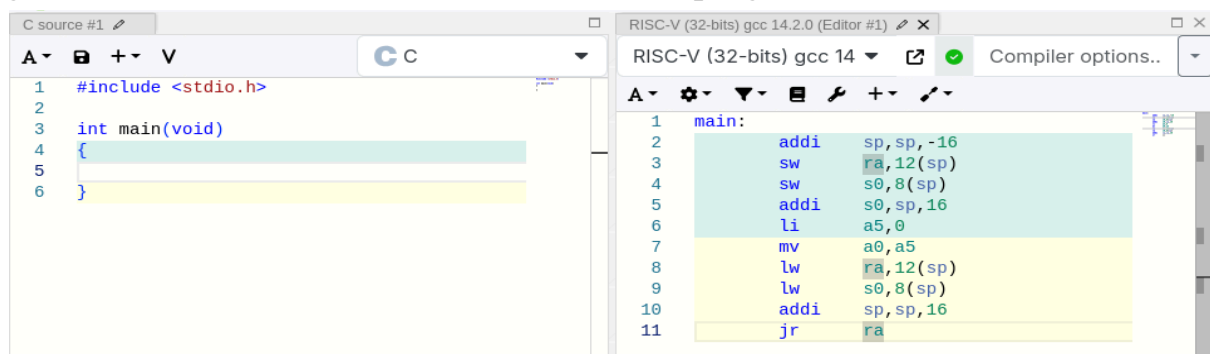


Acima, em escala logarítmica, podemos ver a linha vermelha representando $t_o(n)$, e os pontos onde $n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. E a linha preta representa $t_i(n)$ para os mesmos valores de n . Vemos que o aumento do tempo em função da quantidade de instruções exigidas para ordenar o vetor é bem maior quando ele está inversamente ordenado.

2) Compilador cruzado GCC

2.1 - Programas de teste

Ao compilar os programas de teste fornecidos na pasta arquivos, notamos algumas decisões tomadas pelo compilador para garantir a integridade do programa, assim como possível interoperabilidade entre programas. Mesmo a função vazia no teste 0, quando compilada com o **nível de otimização -O0**, produz um código assembly que guarda uma cópia do stack pointer (sp) e frame pointer (s0) na memória (pilha), para logo em seguida recuperar esses endereços antes de encerrar a função retornando 0 no registrador a0, que por convenção guarda o retorno da função ou, nesse caso, do programa.



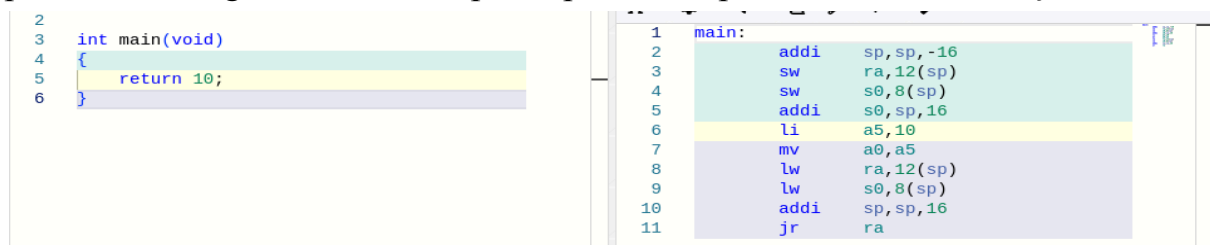
The screenshot shows the GCC IDE interface. On the left, the C source code for 'C source #1' is displayed:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5
6 }
```

On the right, the assembly output for 'RISC-V (32-bits) gcc 14.2.0 (Editor #1)' is shown:

```
1 main:
2     addi    sp, sp, -16
3     sw      ra, 12(sp)
4     sw      s0, 8(sp)
5     addi    s0, sp, 16
6     li      a5, 0
7     mv      a0, a5
8     lw      ra, 12(sp)
9     lw      s0, 8(sp)
10    addi    sp, sp, 16
11    jr      ra
```

Quando modificado, o teste 1 retorna o valor 10, manipulado na função por meio do registrador a5 e copiado para o a0, para o retorno da função main



The screenshot shows the GCC IDE interface. On the left, the C source code for 'C source #1' is modified to return 10:

```
2
3 int main(void)
4 {
5     return 10;
6 }
```

On the right, the assembly output is updated to reflect the return value:

```
1 main:
2     addi    sp, sp, -16
3     sw      ra, 12(sp)
4     sw      s0, 8(sp)
5     addi    s0, sp, 16
6     li      a5, 10
7     mv      a0, a5
8     lw      ra, 12(sp)
9     lw      s0, 8(sp)
10    addi    sp, sp, 16
11    jr      ra
```

No caso do teste 2 é carregado o valor 2047 em uma variável inteira, aqui podemos ver que o compilador utiliza o registrador a5 de forma temporária a fim de imediatamente armazenar o valor da variável i na memória usando o espaço reservado na pilha no início da função main(), dessa forma, o compilador libera o registrador a5 para a execução da função ao custo de ter que recuperar o valor da memória se necessário, nesse exemplo isso é feito logo em seguida para calcular o retorno da função (2*i).

```

C source #1
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i, j;
6     i=2047;
7
8     return i+i;
9 }

RISC-V (32-bits) gcc 14.2.0 (Editor #1)
RISC-V (32-bits) gcc 14
Compiler options..
1 main:
2     addi    sp,sp,-32
3     sw      ra,28(sp)
4     sw      s0,24(sp)
5     addi    s0,sp,32
6     li      a5,2047
7     sw      a5,-20(s0)
8     lw      a5,-20(s0)
9     slli    a5,a5,1
10    mv      a0,a5
11    lw      ra,28(sp)
12    lw      s0,24(sp)
13    addi    sp,sp,32
14    jr      ra

```

No caso do exemplo 2 também é possível ver uma das limitações do formato de instrução utilizado pelo compilador nesse nível de otimização (O0). Ao inicializar a variável `i` como 2048, são necessárias 2 instruções para armazenar o valor correto no registrador, visto que a instrução **addi** (montada a partir da pseudoinstrução **li**) só pode receber 12 bits de imediato usando complemento de 2 para valores com sinal (signed), portanto seu maior valor positivo é $(2^{11})-1 = 2047$.

Ao montar o código abaixo, percebemos que a pseudoinstrução **li a5,4096** é montada como uma instrução **lui x15,1** para carregar o valor 2^{12} na parte superior do registrador, e logo em seguida a instrução **addi a5, a5, -2048** corrige o valor desse registrador por meio da soma com o sinal.

```

C source #1
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i, j;
6     i=2048;
7
8     return i+i;
9 }
10
11
12

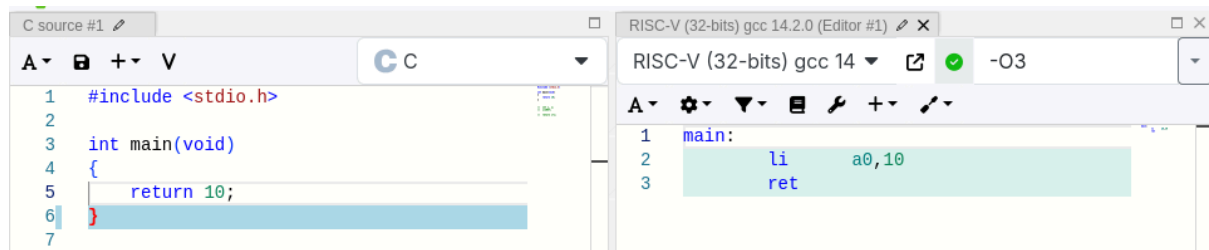
RISC-V (32-bits) gcc 14.2.0 (Editor #1)
RISC-V (32-bits) gcc 14
Compiler options..
1 main:
2     addi    sp,sp,-32
3     sw      ra,28(sp)
4     sw      s0,24(sp)
5     addi    s0,sp,32
6     li      a5,4096
7     addi    a5,a5,-2048
8     sw      a5,-20(s0)
9     lw      a5,-20(s0)
10    slli    a5,a5,1
11    mv      a0,a5
12    lw      ra,28(sp)
13    lw      s0,24(sp)
14    addi    sp,sp,32
15    jr      ra

```

Code	Basic	Source
0x000017b7	lui x15,1	9: li a5,4096

Ao compilar esses exemplos iniciais usando a **diretiva de otimização -O3**, percebemos que o compilador já não reserva nenhum espaço na pilha e não

guarda os ponteiros stack pointer (sp) e frame pointer (s0/fp), partindo direto para o retorno da função por meio do registrador a0



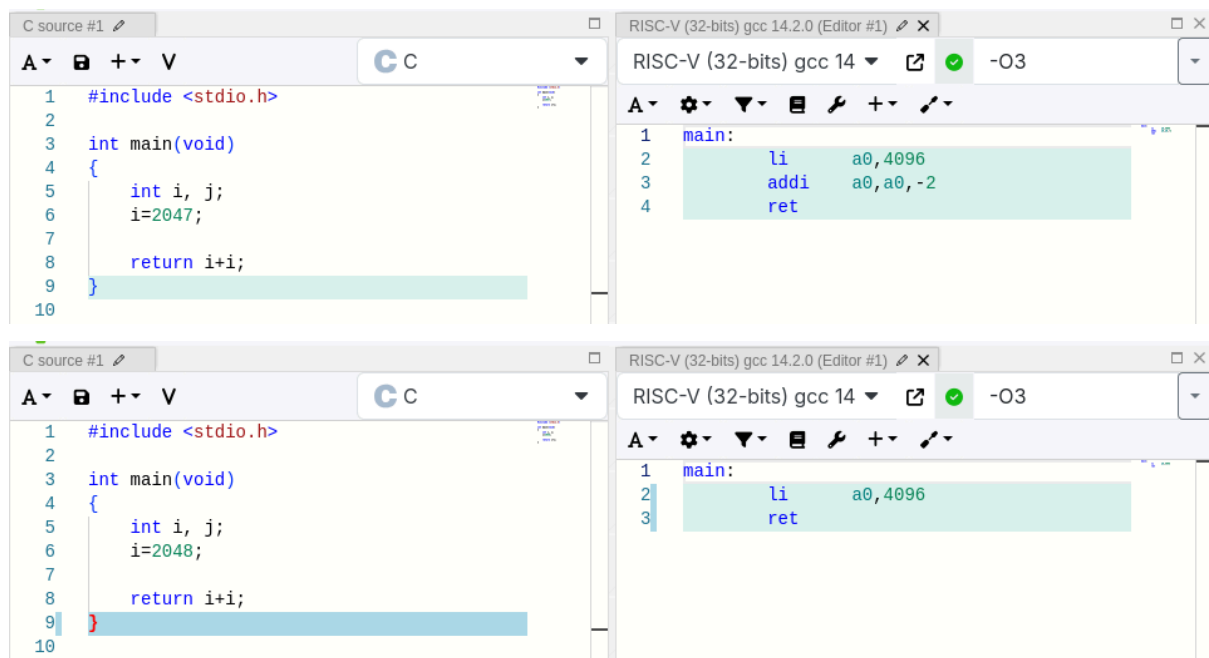
The screenshot shows a code editor with two panels. The left panel, titled 'C source #1', contains the following C code:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     return 10;
6 }
7
```

The right panel, titled 'RISC-V (32-bits) gcc 14.2.0 (Editor #1)', shows the assembly output for the same code, compiled with the -O3 optimization level. The assembly is as follows:

```
1 main:
2     li    a0,10
3     ret
```

Também é possível ver o reuso de valores e registradores de forma mais prática por parte do compilador. Em ambos os exemplos abaixo (ainda no teste 2), o compilador já calcula o valor alvo e armazena diretamente no registrador de saída, encurtando a operação de soma no retorno da função.



The top screenshot shows a code editor with two panels. The left panel, titled 'C source #1', contains the following C code:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i, j;
6     i=2047;
7
8     return i+i;
9 }
10
```

The right panel, titled 'RISC-V (32-bits) gcc 14.2.0 (Editor #1)', shows the assembly output for the same code, compiled with the -O3 optimization level. The assembly is as follows:

```
1 main:
2     li    a0,4096
3     addi  a0,a0,-2
4     ret
```

The bottom screenshot shows a code editor with two panels. The left panel, titled 'C source #1', contains the following C code:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i, j;
6     i=2048;
7
8     return i+i;
9 }
10
```

The right panel, titled 'RISC-V (32-bits) gcc 14.2.0 (Editor #1)', shows the assembly output for the same code, compiled with the -O3 optimization level. The assembly is as follows:

```
1 main:
2     li    a0,4096
3     ret
```

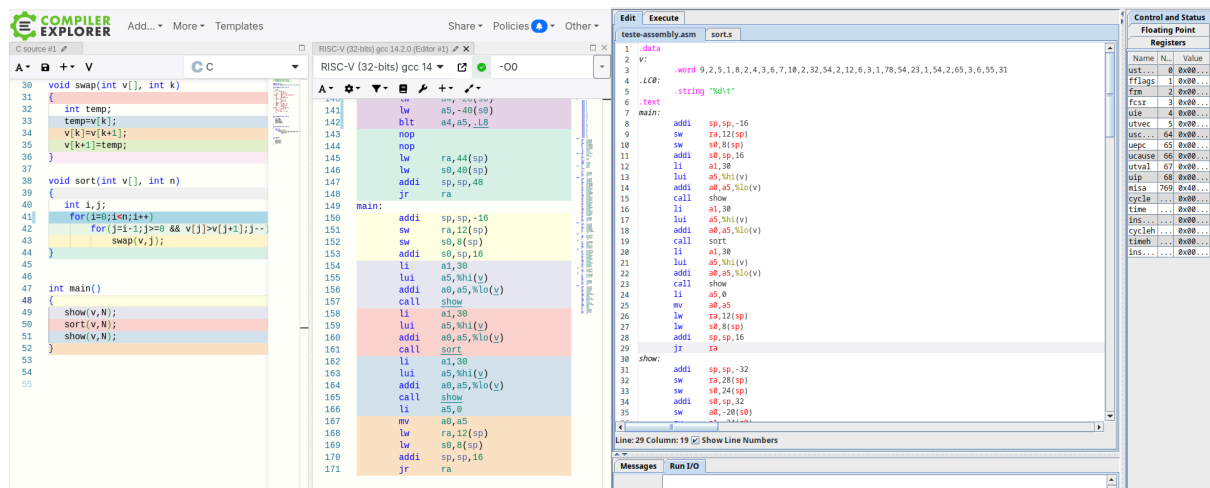
Por meio da compilação desses e dos outros testes presentes no diretório, foi possível perceber a diferença entre as diretivas de compilação -O0 e -O3 utilizadas pelo GCC. Por padrão, utilizando a diretiva -O0, o compilador utiliza muito mais a memória do sistema para armazenar não só o contexto da função (registradores **sp** e **s0/fp**) mas também valores de variáveis locais, essa compilação produz um código mais detalhado e em certos casos até mais legível, porém bem mais extenso. Por outro lado, a compilação utilizando a diretiva -O3, simplifica muito mais o código assembly resultante, concatenando etapas de cálculo, assim como evitando a reatribuição de valores em registradores distintos, além de não utilizar a memória para armazenar o contexto da função ou variáveis locais desnecessariamente.

É possível ver também o uso das convenções estudadas na disciplina para o código assembly gerado pelo compilador, como no caso dos registradores de argumentos (a0-a7) na chamada de funções e procedimentos (como no caso dos testes 4 a 6) assim como o reuso do registrador a0 para o retorno dessas funções.

2.2 - Modificações na compilação de sort.c

Para que seja executado, após compilado, o programa sort.c precisa de algumas adaptações em seu resultado de compilação, a primeira modificação é a separação dos segmentos de texto e dados no código fonte assembly, bem como a organização do vetor v[N] em uma única linha, para facilitar a leitura. Em seguida substituímos parte do código fonte em C por uma função show() implementada de forma direta (utilizando assembly dentro do código fonte em c), como sugerido na especificação do laboratório para que não seja necessário chamar a função printf() (presente na biblioteca stdio.h, que não é incluída pelo linker do GCC, visto que a compilação é interrompida na geração do código assembly).

O montador presente no RARS agora é capaz de montar o código, mas sua execução é interrompida pois a ordem das funções está incorreta e o código começa executando a função show, presente no começo do segmento de texto. Foi necessário modificar o código assembly para que a função main() fosse a primeira a ser executada no segmento de texto.



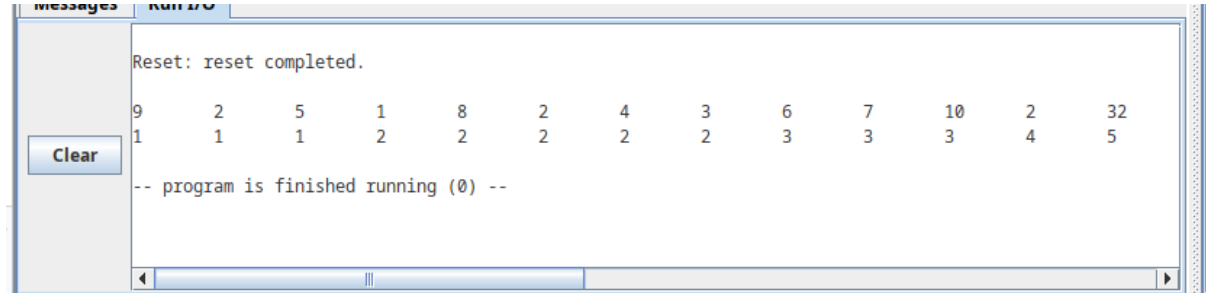
A execução do código funciona a partir da função main, mas ao finalizar, o programa ainda retorna um erro ao RARS tentando sair da função main por meio da instrução **jr ra**, sendo que o valor presente no registrador de retorno (ra), não aponta para nenhum endereço válido para o rars. Para corrigir isso, é

necessário realizar uma chamada de sistema (ecall) para sair da função main e, consequentemente, finalizar o programa corretamente.

```

28      lui      sp, sp + 16
29      #      jr      ra
30      li      a7, 10
31      ecall

```



2.3 - Comparativo de compilações do sort_mod.c

Para realizar o comparativo entre as diretivas de compilação no arquivo sort_mod.c, algumas adaptações precisaram ser feitas tanto no arquivo fonte (C) quando no assembly (como descrito no item 2.2), isso visa prevenir que o compilador necessite da declaração de parâmetros usando a diretiva de montagem .set (indisponível no RARS) e também para prevenir a duplicação do código assembly da função show() quando compilado em -O3, causando problemas durante a montagem.

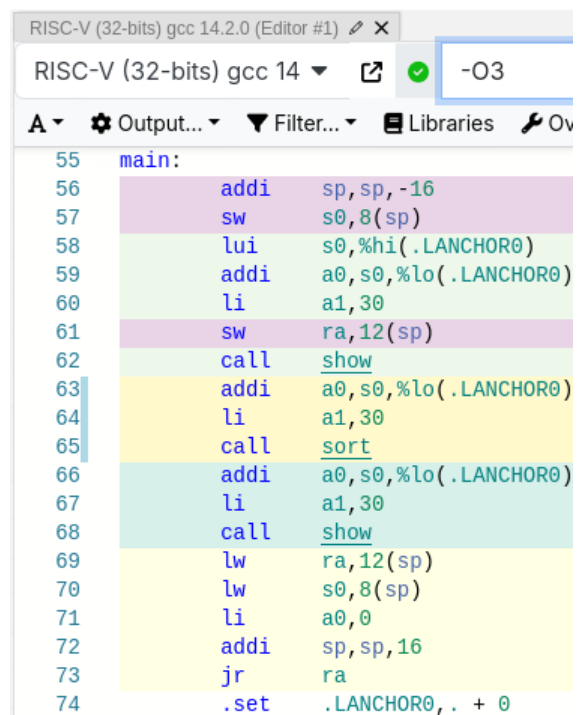
```

6      static int v[10] = {0, 2, 0, 1, 0, 2, 1, 0, 2, 1};
7
8      __attribute__((noinline))
9      void show(int v[], int n)
10     {
11         asm (
12             " mv      t0 %A \n"

```

Acima: atributo para o compilador não duplicar o código assembly inline durante a compilação da função show().

Direita: exemplo de compilação usando a diretiva -O3 e a declaração de uma label .LANCHOR0 problemática utilizando a diretiva assembly .set.



```

RISC-V (32-bits) gcc 14.2.0 (Editor #1)
RISC-V (32-bits) gcc 14 -O3
A Output... Filter... Libraries Ov
55 main:
56     addi    sp, sp, -16
57     sw      s0, 8(sp)
58     lui     s0, %hi(.LANCHOR0)
59     addi    a0, s0, %lo(.LANCHOR0)
60     li      a1, 30
61     sw      ra, 12(sp)
62     call    show
63     addi    a0, s0, %lo(.LANCHOR0)
64     li      a1, 30
65     call    sort
66     addi    a0, s0, %lo(.LANCHOR0)
67     li      a1, 30
68     call    show
69     lw      ra, 12(sp)
70     lw      s0, 8(sp)
71     li      a0, 0
72     addi    sp, sp, 16
73     jr      ra
74     .set    .LANCHOR0, . + 0

```

A contagem de tamanho de cada programa é feita com base no endereço de memória da primeira e última instrução, como no primeiro exemplo (-O0), o código começa em 0x00400000 e termina em 0x00400218, isso significa que esse programa ocupa 0x218 bytes, em seu segmento de texto, ou 536 bytes quando convertido de hexadecimal para decimal.

A tabela abaixo mostra o comparativo entre as diferentes compilações após a execução, ambos partindo do vetor

$v[N]=\{9,2,5,1,8,2,4,3,6,7,10,2,32,54,2,12,6,3,1,78,54,23,1,54,2,65,3,6,55,31\};$.

	Total de instruções executadas pelo programa	Tamanho do código em linguagem de máquina
sort_mod.c -O0	10103	536 bytes
sort_mod.c -O3	2180	264 bytes
sort_mod.c -Os	4098	340 bytes
sort.s (assembly)	3740	272 bytes

2.4 - Tempo de execução das funções f1-f6 usando as otimizações -O0 e -O1.

Avaliando cada uma das funções nas otimizações -O0 e -O1, tivemos os seguintes resultados:

Função	-O0 (instruções)	-O1 (instruções)
F1	12	2
F2	12	2
F3	12	2
F4	12	2
F5	12	2
F6	16	6

Dado isso, podemos concluir que, como o RISC-V é um processador uniciclo, cada instrução leva o mesmo tempo; portanto, a redução drástica no número de instruções com -O1 implica execução muito mais rápida que em -O0.

2.5 - Diferenças entre as otimizações -O0, -O1, -O2, -O3 e -Os.

Os níveis de otimização controlam as transformações que o compilador aplica, tendo impacto significativo no tempo de compilação, tempo de execução e tamanho do binário resultante.

- **-O0**

Nenhuma otimização significativa. De rápida compilação e depuração mais fácil.

- **-O1**

Ativa otimizações básicas de baixo custo (eliminação simples de código morto, propagação de constantes, inlining limitado). Esses passos melhoram o código sem grandes impactos no tempo de compilação.

- **-O2**

Inclui todas as otimizações de -O1 mais otimizações avançadas (eliminação agressiva de código morto, common subexpression elimination, otimizações de loops). É o padrão para builds de produção, oferecendo ótimo desempenho sem crescimento excessivo de tamanho.

- **-O3**

Inclui todas as otimizações de -O2, com passes mais agressivos em loops (desenrolamento, vetorização, software pipelining). Visa máxima velocidade embora produza binários bem maiores e aumente tempo de compilação.

- **-Os**

Baseado em -O2, mas desativa otimizações que aumentam o tamanho do código (ex.: alinhamento e unrolling). Gera executáveis compactos, úteis em sistemas embarcados, com desempenho próximo ao de -O2.

3) Cálculo das raízes da equação de segundo grau

3.1 - O procedimento baskara tem como função calcular as raízes da equação de segundo grau, utilizando a fórmula de Bhaskara. Sua lógica funciona da seguinte maneira:

- Verifica se a é igual a zero. Se for, não se trata de uma equação do segundo grau e o procedimento retorna 0 como sinal de erro
- Calcula o delta usando a expressão $\Delta = b^2 - 4 \cdot a \cdot c$
- Com base no valor de delta:

Delta > 0: calcula duas raízes reais distintas.

Delta = 0: calcula uma raiz real dupla.

Delta < 0: calcula duas raízes complexas conjugadas, separando parte real e parte imaginária.

- As raízes (ou a parte real e imaginária, no caso de complexas) são empilhadas na pilha de execução, e a função retorna:

1 para raízes reais (simples ou iguais),

2 para complexas,

0 em caso de erro.

3.2 - O procedimento show é responsável por exibir na tela as raízes calculadas, com base no valor do tipo t , retornado por baskara.

- Se $t = 1$, indica raízes reais:

Exibe as duas raízes reais armazenadas na pilha, com os rótulos $R(1)$ e $R(2)$.

- Se $t = 2$, indica raízes complexas:

Exibe as duas raízes no formato $\text{parte_real} \pm \text{parte_imaginaria} i$.

- Se $t = 0$, indica erro:

Exibe a mensagem: **"Erro! Não é equação do segundo grau!"**

O procedimento acessa os valores diretamente da pilha e utiliza chamadas de sistema (ecall) para exibir os dados na tela.

3.3 - A função main organiza a execução do programa como um todo:

- Solicita ao usuário que insira os coeficientes a, b e c, do tipo float.
- Chama o procedimento baskara, passando os valores como argumento.
- Em seguida, chama o procedimento show, passando o tipo de raiz retornado.
- Por fim, reinicia o processo ao chamar main novamente, permitindo, dessa maneira o devido funcionamento do programa de inserir uma nova equação.

3.4 - As execuções no RARS foram gravadas e a demonstração se encontra no seguinte link:

 [UnB – OAC Unificado – 2025-1 – Grupo 4 - Laboratório 1 - Questão 3.4 ...](#)

Os tempos de execução da rotina *bhaskara* foram calculados desconsiderando I/O:

$$t_{\text{EXEC}} = I \times \text{CPI} \times T$$

3.4

$$f = 1 \text{ GHz} = 1 \cdot 10^9 \text{ Hz}$$

a) $I = 31$

20 \rightarrow floating point $\rightarrow 20 \times 5 = 100$ cycles

11 \rightarrow CPI = 1 $\rightarrow 11 \times 1 = 11$ cycles

TOTAL = 100 + 11 = 111 cycles

$$t = C \times T = 111 \cdot \frac{1}{10^9} = 111 \cdot 10^{-9} = 111 \text{ ns}$$

b) $I = 27$ $f = 1 \text{ GHz} = 10^9 \text{ Hz}$

16 \rightarrow floating point $\rightarrow 16 \times 5 = 80$ cycles

11 \rightarrow CPI = 1 $\rightarrow 11$ cycles

TOTAL = 80 + 11 = 91 cycles

$$t = C \times T = 91 \cdot \frac{1}{10^9} = 91 \cdot 10^{-9} = 91 \text{ ns}$$

c) $I = 28$ $f = 10^9 \text{ Hz}$

18 \rightarrow floating point $\rightarrow 18 \times 5 = 90$ cycles

10 \rightarrow CPI = 1 $\rightarrow 10$ cycles

TOTAL = 100 cycles

$$t = C \times T = 100 \cdot \frac{1}{10^9} = 100 \text{ ns}$$

d) $I = 28$ $f = 10^9 \text{ Hz}$

18 \rightarrow floating point $\rightarrow 18 \times 5 = 90$ cycles

10 \rightarrow CPI = 1 $\rightarrow 10$ cycles

TOTAL = 100 cycles

$$t = C \times T = 100 \cdot \frac{1}{10^9} = 100 \text{ ns}$$

e) $I = 14$ $f = 10^9 \text{ Hz}$

6 \rightarrow floating point $\rightarrow 6 \times 5 = 30$ cycles

8 \rightarrow CPI = 1 $\rightarrow 8$ cycles

TOTAL = 38 cycles

$$t = 38 \cdot \frac{1}{10^9} = 38 \text{ ns}$$