



Laboratório 1

- Assembly RISC-V -

Objetivos:

- Familiarizar o aluno com o Simulador/Montador Rars;
- Desenvolver a capacidade de codificação de algoritmos em linguagem Assembly;
- Desenvolver a capacidade de análise de desempenho de algoritmos em Assembly;
- Familiarizar o aluno com a compilação C para Assembly RISC-V RV32IMF

(2.5) 1) Simulador/Montador Rars

Faça o download e deszip o arquivo Lab1.zip disponível no Moodle.

(0.0) 1.1) No diretório Arquivos, abra o Rars16_Custom1 e carregue o programa de ordenamento `sort.s`.

Dado o vetor: $V[30]=\{9,2,5,1,8,2,4,3,6,7,10,2,32,54,2,12,6,3,1,78,54,23,1,54,2,65,3,6,55,31\}$,

- a) ordená-lo em ordem crescente e contar o número de instruções por tipo e o número total exigido pelo procedimento `sort`. Qual o tamanho em bytes do código executável? E da memória de dados usada?
- b) Modifique o programa para ordenar o vetor em ordem decrescente e contar o número de instruções por tipo e o número total exigido pelo procedimento `sort`.
- c) Usando os contadores de instruções e tempo do Banco de Registradores CSR (veja no final), meça novamente a quantidade de instruções executadas e o tempo de execução dos itens a) e b).

(2.5) 1.2) Considere a execução deste algoritmo em um processador RISC-V com frequência de *clock* de 50MHz que necessita 1 ciclo de *clock* para a execução de cada instrução (CPI=1). Para os vetores de entrada de n elementos já ordenados $V_o[n] = \{1, 2, 3, 4, \dots, n\}$ e ordenados inversamente $V_i[n] = \{n, n-1, n-2, \dots, 2, 1\}$:

(1.5) a) Para o procedimento `sort`, escreva as equações dos tempos de execução, $t_o(n)$ e $t_i(n)$, em função de n .

(1.0) b) Para $n=\{10,20,30,40,50,60,70,80,90,100\}$, plote (em escala!) as duas curvas, $t_o(n)$ e $t_i(n)$, em um mesmo gráfico `next`. Comente os resultados obtidos.

(2.5) 2) Compilador cruzado GCC

Um compilador cruzado (*cross compiler*) compila um código fonte para uma arquitetura diferente daquela da máquina em que está sendo utilizado. Você pode baixar gratuitamente os compiladores gcc para todas as arquiteturas (RISC-V, ARM, MIPS, x86 etc.) e instalar na sua máquina, sendo que o código executável gerado apenas poderá ser executado em uma máquina que possuir o processador para qual foi compilado. No gcc, a diretiva de compilação `-S` faz com que o processo pare com a geração do arquivo em Assembly e a diretiva `-march` permite definir a arquitetura a ser utilizada.

```
Ex.: riscv64-unknown-elf-gcc -S -march=rv32imf -mabi=ilp32f # RV32IMF
      arm-eabi-gcc -S -march=armv7 # ARMv7
      gcc -S -m32 # x86
```

Para fins didáticos, o site [Compiler Explorer](https://compiler-explorer.com/) disponibiliza estes (e vários outros) compiladores C (com diretiva `-S`) *on-line* para as arquiteturas RISC-V, ARM, x86 e x86-64 e outras.

(0.0) 2.1) Teste a compilação para Assembly RISC-V com programas triviais em C disponíveis no diretório 'Arquivos', para entender a convenção do uso dos registradores e memória utilizada pelo gcc para a geração do código Assembly, usando as diretivas de otimização `-O0` e `-O3`.

Dica: Use o compilador "RISC-V (32-bits) gcc 14.2.0"

(0.5) 2.2) Dado o programa `sortc.c`, compile-o com a diretiva `-O0` e obtenha o arquivo `sortc.s`. Indique as modificações necessárias no código Assembly gerado para que possa ser executado corretamente no Rars.

Dica: Uso de Assembly em um programa em C. Use a função `show` definida no `sort.s` para não precisar implementar a função `printf`, conforme mostrado no `sortc_mod.c`

(1.0) 2.3) Compile o programa `sortc_mod.c` e, com a ajuda do Rars, monte uma tabela comparativa com o número total de instruções executadas pelo **programa todo**, e o tamanho em bytes dos códigos em linguagem de máquina gerados para cada diretiva de otimização da compilação `{-O0, -O3, -Os}`. Compare ainda com os resultados obtidos no item 1.1) com o programa `sort.s` que foi implementado diretamente em Assembly. Analise os resultados obtidos usando o mesmo vetor de entrada.

(0.5) 2.4) Usando o programa `teste12.c` meça com o Rars e compare os tempos de execução de cada função (`f1`, `f2`, `f3`, `f4`, `f5`, `f6`) usando `-O0` e `-O1`. Quais suas conclusões?

(0,5) 2.5) Pesquise na internet e explique as diferenças entre as otimizações `-O0`, `-O1`, `-O2`, `-O3` e `-Os`.

(5.0) 3) Cálculo das raízes da equação de segundo grau:

Dada a função de segundo grau: $f(x) = a.x^2 + b.x + c$

(1.0) 3.1) Escreva em Assembly RISC-V um procedimento `int baskara(float a, float b, float c)` que retorne 1 caso as raízes sejam reais, 2 caso as raízes sejam complexas conjugadas e 0 em caso de erro, e coloque na **pilha** os valores das raízes.

(1.0) 3.2) Escreva um procedimento `void show(int t)` que receba o tipo (t=1 raízes reais, t=2 raízes complexas, t=0 erro), retire as raízes da pilha e as apresente na tela, conforme os modelos abaixo:

Para raízes reais:

R(1) = 12345

R(2) = 56789

Para raízes complexas:

R(1) = 12345 + 56789 i

R(2) = 12345 - 56789 i

Para erro:

Erro! Não é equação do segundo grau!

(1.0) 3.3) Escreva um programa `main` que leia do teclado os valores float de a, b e c, execute as rotinas `baskara` e `show` e volte a ler outros valores.

(2.0) 3.4) Escreva as saídas obtidas para as seguintes funções com coeficientes [a, b, c] e, considerando um processador RISC-V de 1GHz, onde instruções da ISA de inteiros são executadas em 1 ciclo de clock e as instruções de ponto flutuante em 5 ciclos de *clock*, considere que `flw` e `fsw` tenham CPI 1. Calcule os tempos de execução da sua rotina `baskara` (portanto, sem considerar I/O). Filme as execuções no Rars e coloque os links no relatório.

a) [1, 0, -9.86960440]

b) [1, 0, 0]

c) [1, 99, 2459]

d) [1, -2468, 33762440]

e) [0, 10, 100]

Dicas para medir o desempenho

O RISC-V possui o banco de registradores de Status e Controle (visto mais tarde em aula) no qual armazena continuamente diversas informações úteis, e que podem ser lidos pela instrução:

```
csrr t1, fcsr # Control and Status Register Read
```

onde t1 é o registrador de destino da leitura e fcsr é um imediato de 12 bits correspondente ao registrador a ser lido.

Os registradores abaixo são registradores de 64 bits que contém as informações:

{timeh, time} = tempo do sistema em ms

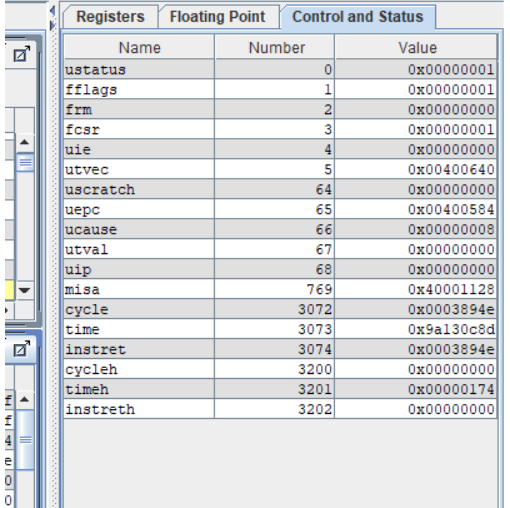
{instreth, instret} = número de instruções executadas

{cycleh, cycle} = número de ciclos executados (se CPI=1 é igual ao instret)

Geralmente nossos programas não precisarão dessa precisão de 64 bits. Podemos usar então apenas os 32 bits menos significativos: time, instret e cycle.

Ex.: Para medir o tempo e o número de instruções do procedimento PROC para os registradores s0 e s1 respectivamente.

```
main: ...
...
csrr s1,3074 # le o num instr atual
csrr s0,3073 # le o time atual
jal PROC
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
...
```



Name	Number	Value
ustatus	0	0x00000001
fflags	1	0x00000001
frm	2	0x00000000
fcsr	3	0x00000001
uie	4	0x00000000
utvec	5	0x00400640
usratch	64	0x00000000
uepc	65	0x00400584
ucause	66	0x00000008
utval	67	0x00000000
uip	68	0x00000000
misa	769	0x40001128
cycle	3072	0x0003894e
time	3073	0x9a130c8d
instret	3074	0x0003894e
cycleh	3200	0x00000000
timeh	3201	0x00000174
instreth	3202	0x00000000

Note que terá um erro de 3 instruções na medida do número de instruções. Por quê?

Como o RARS simula um processador RISC-V Unicycle, pode-se calcular a frequência do processador equivalente: $f = \frac{1}{T_{\text{exec}}}$

Para a apresentação da verificação dos laboratórios (e projeto) nesta disciplina, crie um canal para o seu grupo no YouTube e poste os vídeos dos testes (sempre com o nome 'UnB – OAC Unificado – 2025-1' – Grupo Y - Laboratório X - <palavras-chaves que identifiquem este vídeo em uma busca>'), coloque os links clicáveis no relatório.

Passos do vídeo:

- Apresente o grupo e seus membros;
- Explique o projeto a ser realizado;
- Apresente os testes solicitados;
- Apresente suas conclusões.