



## **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;

### **(2.5) 1) Simulador/Montador Rars**

Faça o download e deszippe o arquivo Lab1.zip disponível no Moodle. Serão criados 2 diretórios.

(0.0) 1.1) No diretório System, abra o `Rars15_Custom1` e carregue o programa de ordenamento `sort.s`. Dado o vetor:  $V[30] = \{9, 2, 5, 1, 8, 2, 4, 3, 6, 7, 10, 2, 32, 54, 2, 12, 6, 3, 1, 78, 54, 23, 1, 54, 2, 65, 3, 6, 55, 31\}$ , 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?

(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  $V_0[n] = \{1, 2, 3, 4, \dots, n\}$  e ordenados inversamente  $V_1[n] = \{n, n-1, n-2, \dots, 2, 1\}$ :

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

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

(0.0) 1.3) Sabendo que as chamadas do sistema padrão do Rars usam um console (parte do SO) para entrada e saída de dados, execute o programa `testeECALLv21.s`. Note que essas chamadas usam diretamente as ferramentas KDMIO e BITMAP DISPLAY.

### **(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](#) disponibiliza estes (e vários outros) compiladores C (com diretiva `-S`) *on-line* para as arquiteturas RISC-V, ARM, x86 e x86-64. (usar C e compilador RISC-V `rv32gc 10.2`).

(0.0) 2.1) Teste a compilação para Assembly RISC-V com programas triviais em C disponíveis no diretório 'ArquivosC', 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`.

(1.0) 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`.

(1.5) 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, -O1, -O2, -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.

(0.0) 2.4) Exemplos de uso da linguagem C para acesso às ferramentas KDMIO e BITMAP DISPLAY (`teste10.c`).

### (5.0) 3) Senha (Mastermind)

O jogo Senha (<https://pt.wikipedia.org/wiki/Mastermind>) é um jogo de 1971 que consiste em descobrir um código de 4 cores de uma lista de N cores a partir tentativas e de dicas baseadas nas cores da tentativa.

Pino Branco: Cor certa no lugar errado

Pino Preto: Cor certa no lugar certo



(1.0) 3.1) Crie um programa no Rars que emule o jogo com máximo de 10 tentativas e N inicial igual a 5;

(1.0) 3.2) Cada cor possui um efeito sonoro único quando colocada no tabuleiro. O preto possui uma pequena música e o branco outra pequena música.

(1.0) 3.3) As cores são escolhidas através do teclado, escreva a codificação das teclas em cores na tela.

(1.0) 3.4) A cada vitória  $N=N+1$ , isto é, o número de cores é incrementado, aumentando a dificuldade do próximo nível.

(0.5) 3.5) Filme vc jogando até o máximo N que seu grupo conseguir. (Não vale usar cheat!!!)

(0.5) 3.6) Faça os gráficos  $N \times t_{\text{exec}}$  e  $N \times I$ , onde  $t_{\text{exec}}$  é o tempo de execução e I o número de instruções requeridas pelo seu algoritmo para a análise do pior caso da tentativa do jogador. Sabendo que o Rars simula uma CPU RISC-V com CPI=1, qual a a frequência de clock desta CPU?

Dicas:

<https://www.youtube.com/watch?v=dMHxyulGrEk>

<https://www.youtube.com/watch?v=-Z9ijoc7RRk>

Dicas: o RISC-V possui um banco de registradores de Status e Controle (visto mais tarde) 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

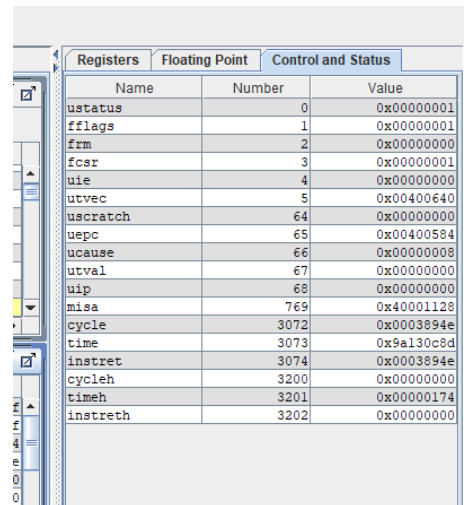
{`cycleh`, `cycle`} = número de ciclos executados (se `CPI=1` é igual ao `instret`)

Geralmente nossos programas não precisarão dessa precisão de 64 bits. Podemos usar então apenas os 32 bits menos significativos.

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
sub s1,t1,s1 # calcula o número de instruções executadas
...
```

Note que terá um erro de 2 instruções na medida do número de instruções. Por quê?



Name	Number	Value
ustatus	0	0x00000001
fflags	1	0x00000001
frm	2	0x00000000
fcsr	3	0x00000001
uie	4	0x00000000
utvec	5	0x00400640
uscratch	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

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 Turma A - 2021/2' – 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.