

SSC0902 - Organização e Arquitetura de Computadores

Relatório - Trabalho 1

Alunos	Nº USP
Dante Brito Lourenço	15447326
João Gabriel Pieroli da Silva	15678578
Pedro Henrique de Sousa Prestes	15507819
Pedro Lunkes Villela	15484287

Trabalho 1 - Calculadora Sequencial

⇒ Introdução

Este trabalho tem como objetivo a implementação de uma calculadora sequencial utilizando a linguagem Assembly para a arquitetura RISC-V. A calculadora permite ao usuário realizar operações aritméticas básicas como adição, subtração, multiplicação e divisão, além de funcionalidades adicionais, como *undo*, em que se desfaz a última operação (comando u), e *finish*, no qual encerra o programa (comando f). A estrutura de dados utilizada para armazenar o histórico das operações é uma lista encadeada, implementada manualmente em Assembly.

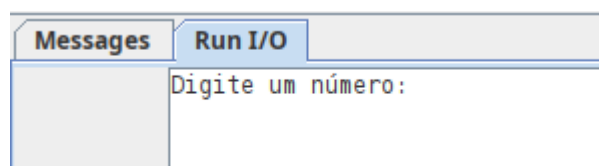
⇒ Objetivos

O principal objetivo desta atividade é proporcionar aos alunos familiaridade com:

- A sintaxe e semântica da linguagem Assembly RISC-V;
- A manipulação de registradores;
- Controle de fluxo e instruções de desvio;
- Alocação dinâmica de memória;
- Implementação de listas encadeadas para armazenar dados de forma dinâmica;
- Conceitos de arquitetura de computadores por meio de uma aplicação prática.

⇒ Implementação

O programa começa pedindo um número inteiro ao usuário, armazenando-o no registrador s1. Tanto a leitura das entradas, quanto as impressões dos textos de interação com o usuário são tratados por meio de chamadas de sistema (ecall), simuladas pelo ambiente *RARS*.

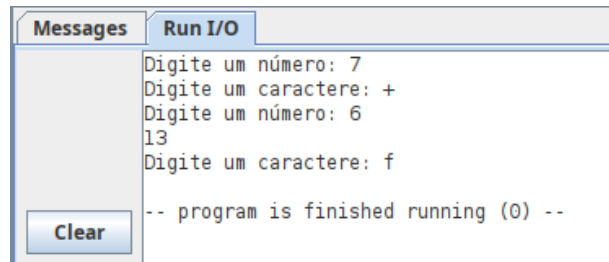


Em seguida, o programa entra em um laço onde é lida uma operação aritmética – a partir da chamada de sistema 12, em que se lê um caractere do usuário diretamente, isto é, sem necessidade de Enter – e um segundo operando para a realização do cálculo. Eles ficam armazenados, respectivamente, em s0 e s2.

Haverá um branch para as seções nas quais o caractere escrito seja referente a desfazer uma operação ou a finalizar o programa, impedindo que o segundo número seja lido.

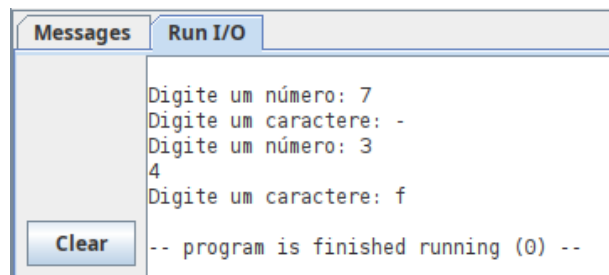
Em relação à aritmética, as operações de soma, subtração, multiplicação e divisão são implementadas dentro da função *operacao*, segundo os caracteres:

+ para soma:



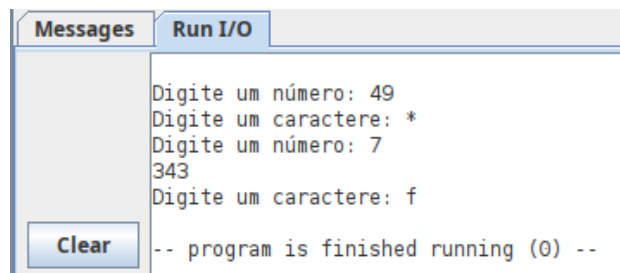
```
Messages Run I/O
Digite um número: 7
Digite um caractere: +
Digite um número: 6
13
Digite um caractere: f
-- program is finished running (0) --
Clear
```

- para subtração:



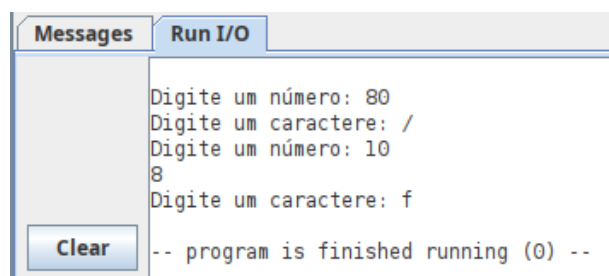
```
Messages Run I/O
Digite um número: 7
Digite um caractere: -
Digite um número: 3
4
Digite um caractere: f
-- program is finished running (0) --
Clear
```

* para multiplicação:



```
Messages Run I/O
Digite um número: 49
Digite um caractere: *
Digite um número: 7
343
Digite um caractere: f
-- program is finished running (0) --
Clear
```

/ para divisão:



```
Messages Run I/O
Digite um número: 80
Digite um caractere: /
Digite um número: 10
8
Digite um caractere: f
-- program is finished running (0) --
Clear
```

Uma observação válida é que, para o caso de o usuário requerer uma divisão por 0, foi feito um tratamento, tal que se exibe uma mensagem de aviso e pede-se novamente o carácter da operação e o segundo operando, sem que isso prejudique o funcionamento adequado do programa.

Messages	Run I/O
	Digite um número: 80
	Digite um caractere: /
	Digite um número: 0
	Divisão por 0 não é permitida.
	Digite um caractere: /
	Digite um número: 2
	40
Clear	Digite um caractere: f
	-- program is finished running (0) --

Ademais, houve uma preocupação quanto ao tratamento de entradas de operações inválidas feitas pelo usuário, de modo que isso também não influenciasse a execução adequada do programa.

Messages	Run I/O
	Digite um número: 40
	Digite um caractere: r
	Operação inválida.
	Digite um caractere: l
	Operação inválida.
	Digite um caractere: c
	Operação inválida.
Clear	Digite um caractere: +
	Digite um número: 1
	41
	Digite um caractere: f
	-- program is finished running (0) --

Conforme o algoritmo construído, é realizado um comparativo com instruções *beq*, identificando o caractere da operação digitada e redirecionando para a instrução correspondente. Após a execução, o resultado é retornado em *a0* e impresso na tela.

Além de imprimi-lo na tela, cada resultado de operação é armazenado em uma estrutura de lista encadeada para possibilitar a funcionalidade de *'undo'*, a qual remove o último valor calculado.

Nesse prisma, como as operações são sequenciais, o último valor adicionado nessa lista encadeada é aquele que fica no registrador do primeiro operando, permitindo fazer cálculos com ele, ou a partir do comando *'undo'*, com os resultados anteriormente obtidos.

Messages	Run I/O
	Digite um número: 80
	Digite um caractere: /
	Digite um número: 10
	8
	Digite um caractere: *
	Digite um número: 4
	32
	Digite um caractere: +
	Digite um número: 4
	36
Clear	Digite um caractere: /
	Digite um número: 6
	6
	Digite um caractere: -
	Digite um número: 10
	-4
	Digite um caractere: u
	6
	Digite um caractere: +
	Digite um número: 4
	10

A alocação de nós da lista é feita via `ecall` com o serviço 9, e os ponteiros são manipulados diretamente para formar a estrutura dinâmica.

- A função `'novo_no'` insere um novo nó no fim da lista, implementando como se fosse uma pilha. Vale postular que cada nó contém uma word (4 bytes), com o valor inteiro do resultado acumulado, e uma outra word (4 bytes), em que há o endereço do nó anterior.
- A função `'remove_no'` remove o nó mais recente adicionado, isto é, o último da lista, como se fosse o topo da pilha, e atualiza a cabeça da lista para o nó anterior.
- Há um tratamento para a situação, a partir da função `'remove_no_except'`, em que a lista de resultados acumulados está vazia, de modo que se o usuário tentar realizar um *undo* e não existirem operações anteriores, uma mensagem é exibida informando a impossibilidade da ação. Por padrão, assume-se que o valor do primeiro operando, para uma lista vazia, passa a ser 0.

Messages	Run I/O
	Digite um número: 80
	Digite um caractere: /
	Digite um número: 10
	8
	Digite um caractere: *
	Digite um número: 9
	72
	Digite um caractere: -
	Digite um número: 12
	60
	Digite um caractere: +
	Digite um número: 3
	63
	Digite um caractere: /
	Digite um número: 7
	9
	Digite um caractere: u
	63
	Digite um caractere: /
	Digite um número: 9
	7
	Digite um caractere: -
	Digite um número: 2
	5
	Digite um caractere: u
	7
	Digite um caractere: u
	63
	Digite um caractere: u
	60
	Digite um caractere: /
	Digite um número: 10
	6
Clear	Digite um caractere: u
	60
	Digite um caractere: u
	72
	Digite um caractere: u
	8
	Digite um caractere: +
	Digite um número: 3
	11
	Digite um caractere: u
	8
	Digite um caractere: u
	80
	Digite um caractere: u
	Não há operações anteriores.
	0
	Digite um caractere: u
	Não há operações anteriores.
	0
	Digite um caractere: +
	Digite um número: 8
	8
	Digite um caractere: *
	Digite um número: 6
	48
	Digite um caractere: f
	-- program is finished running (0) --

⇒ **Decisões de Implementação**

A estrutura escolhida foi uma pilha via lista encadeada para refletir a lógica de desfazer operações, seguindo o paradigma LIFO (Last In First Out). Além disso, cada parte lógica foi isolada em seções: entrada, operações, tratamentos, gerenciamento da lista, visando a obter uma maior modularização. Nesse sentido, desejou-se tratar erros de operações inválidas e divisões por 0, assim como foi criada uma verificação para o caso de *'undo'* sem histórico, o qual poderia causar grandes problemas de acesso indevido à memória.

⇒ **Dificuldades Encontradas**

O controle do fluxo de dados em uma linguagem de baixo nível foi uma das maiores dificuldades na gestão do projeto no geral, pois em um projeto relativamente grande, o gerenciamento do banco de registradores torna-se cada vez mais complexo para várias funções, operações e variáveis.

Além disso, devido à abstração do código, demandou que o grupo mantivesse uma comunicação constante para que o entendimento e as ideias fossem compartilhadas de maneira coerente, além de – é claro – manter o código bem comentado.

Um outro ponto válido de discussão é que, ao longo dos testes, percebeu-se a necessidade de tratar erros, como a digitação de caracteres de operação inválidos e cálculos que não poderiam acontecer, como a divisão por 0. Um desafio foi analisar uma forma de implementar essas lógicas de verificação em um local adequado do código, sem que isso prejudicasse o bom funcionamento da calculadora sequencial, sobretudo, na questão de alocação dinâmica de novos nós da lista.

⇒ **Conclusão**

Este trabalho proporcionou uma experiência prática muito válida e coerente com a linguagem Assembly RISC-V e os conceitos de arquitetura de computadores. Foi possível aplicar o conhecimento prévio do grupo sobre registradores, instruções, chamadas de sistema e alocação dinâmica de memória para resolver um problema concreto e relativamente simples, uma calculadora sequencial.

A utilização de lista encadeada para gerenciar o histórico de resultados mostrou-se extremamente eficiente e reforçou o entendimento sobre estruturas de dados dinâmicas em uma linguagem de baixo nível e, portanto, o endereçamento de memória por posição.

Em relação à documentação, ficou nítida a importância de fazer bons comentários ao longo do código, a fim de compreender bem as diversas partes dele e como elas se conectam para que o programa funcione como deveria.