Universidade de Fortaleza
Centro de Ciências Tecnológicas
Curso de Ciência da Computação
Disciplina de Arquitetura de Computadores

Programação Assembly para Arquitetura MIPS: Ordenação de Lista

Equipe:

Gabriel de Sousa Nobre - 2410399

Victor Lins Gurgel do Amaral - 2410448

Fortaleza, 2024

Sumário

1 Introdução

Os processadores tiveram um longo caminho de evolução. No início, nos anos 1940, o **ENIAC** e outros primeiros computadores usavam válvulas eletrônicas e eram enormes, consumindo muita energia para realizar cálculos simples com a programação sendo realizada fisicamente rearranjando as válvulas sempre que fosse preciso mudar o programa. Na década de 1950, os transistores substituíram as válvulas, tornando os processadores menores e mais rápidos, o que possibilitou o surgimento de computadores mais compactos.

A partir dos anos 1970, o lançamento do Intel 4004, o primeiro microprocessador comercial, revolucionou a tecnologia. Com ele, processadores passaram a caber em um único chip, transformando o mercado de computação. Nas décadas seguintes, houve uma busca por velocidade e eficiência, com o desenvolvimento de processadores multicore e especializados, como as GPUs, que deram origem a avanços em áreas como processamento de imagens e inteligência artificial. Hoje, o estado da arte são processadores com múltiplos núcleos, capazes de suportar processamento paralelo e tarefas de alto desempenho, incluindo aprendizado de máquina em tempo real.

No início, a programação era feita em código binário, que era complexa e suscetível a erros. A linguagem assembly facilitou esse processo, permitindo que os programadores usassem mnemônicos, como MOV e ADD, ao invés de binários, o que simplificava o desenvolvimento. Assembly continua sendo importante até hoje, especialmente em sistemas embarcados, porque permite controle total sobre o hardware, sendo essencial em aplicações que demandam alto desempenho e precisão.

Este trabalho abordará o desenvolvimento de um programa utilizando a linguagem assembly do processador MIPS. Este programa fará basicamente as seguintes etapas:

1. Abrirá um arquivo lista.txt que contém uma lista de 100 números inteiros que variam de -989 à 996 todos separados por vírgula, ele irá ler toda a lista e gravará na memória como uma string de caracteres ASCII, em seguida, com um algoritmo de conversão, converterá a string de caracteres em uma lista de números inteiros que será salva na memória também, após isso irá fechar o arquivo .txt.

2. Ordenará toda a lista de inteiros utilizando um algoritmo de ordenação, que no caso deste trabalho é o chamado Buble sort, após a ordenação a lista de inteiros será convertida em uma lista de caracteres ASCII novamente utilizando um algoritmo de conversão parecido com o utilizado para a conversão de ASCII para inteiros.

3. Abrirá um novo arquivo chamado lista_ordenada.txt onde será escrita toda a lista de números inteiros em ordem do menor para o maior, e em seguida fechará o arquivo e encerrará o programa.

Para desenvolvermos o programa usaremos um emulador do processador MIPS chamado MARS na versão 4.5 para escrever o código e o executarmos. A meta é entender como otimizar o uso do hardware, essencial para sistemas embarcados e computação eficiente.

2 O Processador

Um **processador** (ou **CPU**, do inglês *Central Processing Unit*) é o componente principal de um sistema computacional. Ele é responsável por interpretar e executar instruções de programas. Ele realiza operações aritméticas, lógicas e de controle, permitindo que o computador processe dados e execute tarefas. O processador é composto por várias partes internas, como a unidade de controle (UC), que coordena as operações, a unidade lógica e aritmética (ULA), que realiza vários tipos de cálculos, e registradores, que são pequenas memórias que armazenam dados temporários, além disso processadores podem possuir memória cache interna também.

A **família de processadores** é um conjunto de processadores que compartilham uma mesma arquitetura básica, ou seja, um conjunto de características, instruções e padrões de design que facilitam o desenvolvimento e compatibilidade entre dispositivos e sistemas operacionais. Esses processadores geralmente possuem variações em termos de desempenho e recursos, mas seguem a mesma base tecnológica, o que permite que instruções e softwares desenvolvidos para uma versão sejam compatíveis com outras da mesma família.

Exemplos de arquiteturas notáveis:

- **Intel x86**: Pioneira em processadores para PCs, conhecida pelo uso em desktops e laptops.
- **ARM**: Famosa pela eficiência energética, amplamente utilizada em dispositivos móveis.
- **PowerPC**: Utilizada em consoles de videogame e computadores Apple na década de 1990 e início dos anos 2000.
- **SPARC**: Usada em estações de trabalho e servidores, desenvolvida pela Sun Microsystems.

2.1 Arquitetura Von Neumann e Arquitetura Harvard

As arquiteturas de computador **Von Neumann** e **Harvard** são dois modelos de arquitetura de computadores que definem como os dados e as instruções de um programa são armazenados e acessados pelo computador. A principal diferença entre elas é na forma como tratam a memória para dados e instruções. Abaixo, estão as principais diferenças entre as duas:

Arquitetura Von Neumann

Memória Unificada: Na arquitetura Von Neumann, tanto os dados quanto as instruções de programa são armazenados na mesma memória. Assim, eles compartilham o mesmo barramento de dados e de endereço para acessar a memória.

Acesso Sequencial: Como dados e instruções compartilham o mesmo barramento, o processador precisa alternar entre acessar dados e buscar instruções, o que pode causar um gargalo conhecido como o **bottleneck de Von Neumann** ou "gargalo de Von Neumann". Esse gargalo ocorre porque o processador deve esperar enquanto busca instruções e dados no mesmo barramento, o que limita a velocidade de execução.

Flexibilidade: Esse modelo é simples e flexível, o que facilita a implementação e o uso em diversas plataformas. A arquitetura Von Neumann é amplamente utilizada em computadores de propósito geral, como desktops e notebooks, pela simplicidade no design da memória.

Exemplo de Uso: A maioria dos computadores pessoais (PCs) e de propósito geral usa a arquitetura Von Neumann. Processadores como Intel e AMD, que usam a arquitetura x86, foram projetados inicialmente com base nesse modelo.

Arquitetura Harvard

Memória Separada: Na arquitetura Harvard, dados e instruções são armazenados em memórias separadas. Assim, cada um possui seu próprio barramento dedicado para acesso, evitando que dados e instruções “compitam” entre si pelo uso do barramento.

Acesso Paralelo: Com memórias e barramentos separados, o processador pode acessar simultaneamente os dados e as instruções, eliminando o gargalo observado na arquitetura Von Neumann. Isso permite maior velocidade e eficiência de execução.

Design Complexo: Embora ofereça um desempenho superior, a arquitetura Harvard é mais complexa de implementar e possui menos flexibilidade para compartilhar memória entre dados e instruções. No entanto, isso pode ser uma vantagem para sistemas que exigem alta performance e baixa latência.

Exemplo de Uso: A arquitetura Harvard é comumente usada em micro-controladores e processadores de sinais digitais (DSPs), como os encontrados em dispositivos embarcados, sistemas de controle automotivo e eletrônicos de consumo (como micro-ondas e TVs). Um exemplo clássico são os processadores da família PIC e alguns chips ARM usados em dispositivos móveis, que implementam uma arquitetura Harvard modificada.

Resumo das Diferenças

Característica	**Arquitetura Von Neumann**	**Arquitetura Harvard**
Memória	Memória unificada para dados e instruções	Memórias separadas para dados e instruções
Barramento	Barramento único para dados e instruções	Barramentos separados
Velocidade de Acesso	Mais lenta devido ao gargalo de Von Neumann	Mais rápida por permitir acesso paralelo
Complexidade de Design	Mais simples	Mais complexa
Aplicação	Computadores de propósito geral	Sistemas embarcados e de alto desempenho

Concluindo, a escolha entre essas arquiteturas depende das necessidades específicas do sistema. A arquitetura Von Neumann é preferida em dispositivos de uso geral devido à sua simplicidade, enquanto a Harvard é escolhida em sistemas onde o desempenho e a eficiência energética são mais críticos.

2.2 RISC e CISC

Instruction Set Architecture (ISA) é o conjunto de instruções e a interface que define como o software controla o hardware de um processador. O ISA especifica os comandos básicos que o processador pode executar, incluindo operações aritméticas, lógicas, controle de fluxo e acesso à memória. Em outras palavras, o ISA serve como uma camada intermediária entre o hardware (processador) e o software, definindo como o sistema operacional e os programas interagem com o processador.

RISC vs. CISC

No desenvolvimento de processadores, surgiram duas abordagens principais para o ISA: **RISC** (*Reduced Instruction Set Computing*) e **CISC** (*Complex Instruction Set Computing*). Essas duas filosofias de design possuem diferentes estratégias para executar instruções, cada uma com suas vantagens e desvantagens.

RISC (Computadores com Conjunto de Instruções Reduzido)

Características:

- Processadores RISC utilizam um conjunto de instruções menor e mais simplificado, onde cada instrução executa uma operação muito específica e simples.
- Cada instrução é projetada para ser executada em um único ciclo de clock, tornando o processamento mais rápido e eficiente.
- Grande parte das instruções são executadas diretamente pelos registradores do processador, minimizando o acesso à memória.
- O design RISC simplifica a lógica do processador, permitindo mais eficiência e maior desempenho em muitas operações.

Histórico e Evolução:

- A arquitetura RISC surgiu na década de 1980, como uma resposta ao aumento da complexidade dos processadores CISC. Na época, os pesquisadores perceberam que muitas instruções complexas dos processadores CISC eram pouco utilizadas e que simplificar o conjunto de instruções poderia resultar em maior eficiência.
- Os projetos pioneiros de RISC, como o MIPS e o SPARC, foram desenvolvidos por universidades e tiveram impacto significativo na indústria, influenciando a criação de processadores como o ARM, que hoje é amplamente utilizado em dispositivos móveis.

Exemplos:

- ARM, MIPS, PowerPC e SPARC são alguns exemplos de arquiteturas que seguem a filosofia RISC.

CISC (Computadores com Conjunto de Instruções Complexo)

Características:

- Processadores CISC têm um conjunto de instruções maior e mais complexo, onde cada instrução pode realizar múltiplas operações, como acessar a memória, realizar cálculos e armazenar resultados.
- Instruções mais complexas podem levar mais de um ciclo de clock para serem executadas, mas podem ser mais "poderosas", o que reduz a quantidade total de instruções em um programa.
- As arquiteturas do tipo CISC favorecem a redução do número de instruções no código, permitindo que instruções complexas substituam várias instruções mais simples.

Histórico e Evolução:

- A arquitetura CISC começou a se desenvolver na década de 1970, quando a memória era muito cara e a execução de instruções complexas com menos linhas de código era vantajosa para reduzir o uso de memória.
- Um exemplo clássico de arquitetura CISC é a arquitetura x86, usada em processadores da Intel e AMD. Esses processadores implementam instruções complexas para facilitar a programação e reduzir a quantidade de código.

Exemplos:

- x86 (Intel, AMD) e VAX (DEC) são exemplos de arquiteturas CISC.

Resumo da principais diferenças entre RISC e CISC

Aspecto	**RISC**	**CISC**
Conjunto de Instruções	Reduzido e simplificado	Complexo e amplo
Ciclos de Clock	Instruções rápidas (1 ciclo de clock por instrução)	Instruções podem levar múltiplos ciclos
Acesso à Memória	Minimizado, opera mais em registradores	Instruções podem acessar a memória diretamente
Tamanho do Código	Código maior, mas mais rápido de executar	Código menor, com instruções mais complexas
Complexidade do Hardware	Menor complexidade	Maior complexidade
Eficiência Energética	Geralmente mais eficiente em dispositivos móveis	Pode consumir mais energia

Como essa diferença conceitual afeta os processadores atuais

Atualmente, a distinção entre RISC e CISC é menos rígida do que no passado, pois as arquiteturas evoluíram para incorporar características de ambas. Essa convergência é uma resposta à necessidade de balancear eficiência de energia e desempenho, especialmente em dispositivos móveis e sistemas de alto desempenho.

Arquiteturas Híbridas:

- Os processadores modernos x86 (CISC) da Intel e AMD, por exemplo, têm incorporado muitas otimizações RISC internamente. Eles usam uma técnica chamada *micro-op translation*, onde instruções complexas CISC são convertidas em micro-operações mais simples (similares às instruções RISC) para processamento interno, otimizando o desempenho.
- Da mesma forma, arquiteturas RISC como ARM têm adotado instruções mais complexas para atender a certas demandas de software, criando variações como o ARMv8, que suporta instruções de 64 bits para aumentar a capacidade de processamento.

Uso em Diferentes Dispositivos:

- A arquitetura ARM, um exemplo de RISC, é preferida em dispositivos móveis e IoT devido à sua eficiência energética. Por outro lado, x86 (CISC) ainda é a escolha dominante para computadores pessoais e servidores devido à sua compatibilidade com um extenso ecossistema de software.

Tendências Atuais:

- A crescente popularidade do RISC-V, uma arquitetura RISC de código aberto, reflete a preferência da indústria por arquiteturas flexíveis, modulares e eficientes em termos energéticos. Ela permite personalizações que facilitam o uso em dispositivos embarcados, onde a eficiência energética é crucial.

Em resumo, a distinção entre RISC e CISC atualmente é mais sobre filosofia de design e otimizações do que uma divisão rígida, pois os processadores modernos combinam características de ambos para atender à demanda por desempenho, eficiência energética e versatilidade em diferentes aplicações.

Portanto, a linha entre RISC e CISC continua a aos poucos se desfazer com a evolução tecnológica, e a escolha de uma arquitetura depende das necessidades específicas de cada dispositivo e da carga de trabalho.

2.3 Ciclo de Busca Decodificação e Execução

O ciclo de busca, decodificação e execução é o processo pelo qual a CPU (Unidade Central de Processamento) lê e processa instruções de um programa da memória principal. Nesse ciclo, os principais componentes da CPU são, a Unidade de Controle (UC), a Unidade Lógica e Aritmética (ALU), e os registradores essenciais, incluindo o ACC (Acumulador), PC (Program Counter), MAR (Memory Address Register), IR (Instruction Register) e MBR (Memory Buffer Register).

As etapas da busca em um ciclo de instrução do processador começam quando a CPU precisa obter a próxima instrução do programa na memória. Para isso, a CPU utiliza o Contador de Programa (PC), que armazena o endereço da próxima instrução a ser buscada. Esse endereço é enviado para o Registrador de Endereço de Memória (MAR), que direciona a busca para a posição correta na memória.

Com o endereço configurado no MAR, a Unidade de Controle envia uma instrução de leitura a memória, que por sua vez, envia a instrução contida naquele endereço passado pelo MAR para o Registrador de Buffer de Memória (MBR). O MBR funciona como uma área intermediária onde a instrução é armazenada temporariamente antes de ser processada.

Após a instrução ser transferida para o MBR, ela é enviada para o Registrador de Instrução (IR), onde ficará disponível para ser decodificada. Nesse momento, o PC é incrementado para o próximo endereço, preparando-se para a próxima instrução na sequência. A instrução armazenada no IR está então pronta para a próxima fase do ciclo, que é a decodificação. Na etapa de decodificação, a CPU interpreta a instrução armazenada no IR para entender a operação a ser realizada, a Unidade de Controle examina a instrução no IR para identificar seu opcode (código da operação). Com base no opcode, a UC determina quais dados ou registradores serão usados como operandos. Se a instrução requer dados de um endereço específico da memória ou de um registrador, o UC os identifica e configura a CPU para a execução da operação.

Na etapa de execução, a operação especificada pela instrução é realizada, a Unidade de Controle ativa a ALU se a operação for aritmética ou lógica, a ALU recebe os operandos e realiza a operação, enviando o resultado para o ACC (Acumulador), se a instrução envolve leitura ou escrita na memória, a UC direciona o MAR para o endereço correto e o MBR para armazenar ou buscar o valor apropriado. Esse processo se repete continuamente, garantindo que a CPU obtenha e execute as instruções de forma sequencial (a menos que uma instrução de controle altere essa ordem).

2.4 Marcos Históricos

Linha do Tempo dos Processadores

- **Intel 4004 (1971)**: Primeiro microprocessador comercial, possuía apenas 4 bits, com 2.300 transistores.
- **Intel 8086 (1978)**: Primeiro processador de 16 bits da Intel, foi essencial para a popularização da arquitetura x86, ainda usada em processadores modernos. Seu design permitiu mais poder de processamento e foi a base para o IBM PC.
- **Intel 80486 (1989)**: Este processador foi o primeiro a integrar uma unidade de ponto flutuante (FPU) ao chip, e primeiro a introduzir no mercado a memória cache integrada no chip da CPU.
- **Intel Pentium (1993)**: Introduziu o processamento superescalar, permitindo a execução simultânea de múltiplas instruções.
- **AMD-K5 (1996)**: Primeiro processador projetado internamente pela AMD, foi feito para competir com o Intel Pentium.
- **Intel Pentium 4 (2000)**: Primeira CPU da intel com multi-thread, possuindo duas threads em seu núcleo que são processadas simultaneamente, melhorando o desempenho.
- **IBM Power4 (2001)**: Um marco na arquitetura multi-core, o Power4 foi o primeiro processador a incorporar dois núcleos de processamento no mesmo chip. Isso representou um avanço significativo na computação paralela.
- **AMD Athlon 64 (2003)**: Foi o primeiro processador de 64 bits para desktops baseado na arquitetura x86, revolucionando o mercado ao trazer suporte para maior capacidade de memória e melhorias de desempenho.
- **Intel Pentium Processor Extreme Edition 840 (2005)**: O primeiro processador da Intel para desktops com dois núcleos, suportando também multi-thread, o que permitia o processamento de até quatro threads ao mesmo tempo.
- **AMD Athlon 64 X2 (2005):** Primeiro processador de dois núcleos da AMD para desktops, oferecendo ótimo desempenho.
- **ARM Cortex-A8 (2007)**: Processador móvel que utilizava a arquitetura ARMv7 e oferecia suporte a múltiplas instruções por ciclo e decodificação avançada, tornando-o eficiente em energia e ideal para dispositivos móveis e tablets.
- **Intel Core i7 (2008)**: Com arquitetura Nehalem, o Core i7 trouxe avanços em desempenho multi-threaded, incluindo o Hyper-Threading e a memória cache compartilhada L3, otimizando a execução paralela em um único chip multi-core.

Esses processadores marcam pontos de virada tecnológicos importantes, desde a introdução da arquitetura x86 até o avanço de tecnologias multi-core e multi-thread que ainda impulsionam o desempenho dos processadores modernos. Essas inovações mudaram significativamente o desempenho e a capacidade dos processadores, permitindo avanços notáveis em computação multitarefa, jogos, inteligência artificial e aprendizado de máquina.

3 Arquitetura MIPS

A arquitetura MIPS foi desenvolvida inicialmente como projeto de um pesquisador de Stanford que desejava apenas explorar o conceito de **RISC** (Reduced Instruction Set Computer), introduzido ao contexto de desenvolvimento de processadores no início da década de 80. A ideia do pesquisador se tratava em usar um conjunto pequeno de instruções de forma mais eficiente a fim de simplificar o desenvolvimento de processadores e aumentar a velocidade de processamento. Na época, era mais comum o desenvolvimento de processadores baseados na arquitetura **CISC** (Complex Instruction Set Computer) que comportavam um número maior de instruções complexas, contudo essas instruções poderiam levar mais de um ciclo de clock para serem concluídas. O objetivo da arquitetura MIPS era focar no design de processadores capazes de executar instruções mais simples em um único ciclo de clock.

Entrando nos anos 90, a arquitetura MIPS foi usada amplamente no mercado, marcando presença em consoles de videogame como o **PlayStation** e **Nitendo 64**, que usavam os processadores R3000 de 32 bits e R4000 de 64 bits, respectivamente. Além disso, a arquitetura MIPS também se mostrava presente em **dispositivos de rede** e telecomunicações devido sua eficiência em processar tarefas simples e específicas rapidamente e com baixo consumo.

A partir de 2010, a arquitetura **ARM** e **x86** se popularizaram e tomaram o espaço que a arquitetura MIPS tinha no mercado. Atualmente, o modelo ARM se faz presente na maioria dos smartphones, dispositivos de automação e IoT, além de dispositivos desenvolvidos pela Apple, já x86 se faz mais presente em computadores pessoais, servidores, supercomputadores e consoles de videogames.

Após perder presença no mercado, parte da arquitetura MIPS foi disponibilizada de forma **Open-Source** a fim de estimular a inovação. Ainda hoje a arquitetura MIPS é utilizada para estudos no contexto de desenvolvimento de processadores e principalmente no campo de arquitetura de computadores.

3.1 Conjunto de Instruções MIPS32

A filosofia **ISA** (Instruction Set Architecture) do MIPS, por ser baseada no design RISC, apresenta um conjunto de instruções simples e de rápida execução. As instruções possuem 32 bits e são divididas em 3 tipos: tipo R, tipo I e tipo J.

O **tipo R** (Register Type) se trata de operações aritméticas e operações lógicas entre registradores. Os 32 bits da instrução indicam o código de operação (opcode) além de até 3 registradores que serão acessados para as operações como soma(ADD), AND, OR etc.

O **tipo I** (Immediate Type) se trata de operações com valores imediatos ou instruções relacionadas a algum tipo de acesso à memória, como load e store. Os valores imediatos são valores já inclusos na instrução ao invés de armazenados em um registrador. Esses valores imediatos são compostos por 16 dos 32 bits de uma instrução de tipo I, ou seja, comportam valores entre -32768 a 32767, já o restante comporta o opcode e até 2 registradores envolvidos à instrução.

O **tipo J** (Jump Type) são instruções de salto direto para endereços específicos na memória. Dedica 26 dos 32 bits para o valor imediato que representa um endereço na memória, o restante esta relacionado ao opcode.

3.1.1 Instruções e Pseudo-Instruções

As instruções básicas são diretamente executadas no hardware e fazem parte do conjunto de instruções definidas pela ISA do MIPS. Já as pseudo-instruções surgem para facilitar a programação em Assembly, são instruções com uma camada de **abstração**, elas representam mais de uma instrução básica e antes de serem executadas pelo processador elas são convertidas pelo montador em uma série de instruções básicas.

3.1.2 Instruções de Salto

As instruções de salto servem para direcionar um fluxo para o código, são utilizadas para execução de instruções dentro de uma condicional, instruções de sub-rotinas ou ainda instruções dentro de laços de repetição e chamadas recursivas. As instruções de salto podem ser do tipo Branch ou Jump.

Branch são aquelas acompanhadas de uma condição, aplicando o salto para determinado endereço apenas se atendida a condição. Essas instruções incluem 16 bits de um valor imediato e calculam um endereço de salto associado. O cálculo consiste em aplicar um incremento ao PC da próxima instrução, lembre que o PC indica o sequenciamento de instruções. Expressão: (PC + 4) + (immediate * 4). Perceba que “4” é um ajuste feito para garantir o alinhamento de memória do MIPS que apresenta instruções sempre de 32 bits, ou seja, 4 Bytes. Note ainda que o valor imediato de 16 bits limita o alcance do salto para um endereço próximo de um intervalo de ± 32KB.

Jump são saltos incondicionais que incluem um valor imediato de 26 bits, dessa forma permitindo saltos para endereços mais distantes no código. Agora o cálculo consiste em concatenar os 4 bits mais significativos do PC com o valor imediato e ainda concatenar com “00”, por fim gerando um endereço de 32 bits e garantindo o alinhamento de memória do MIPS. Expressão: PC[31:28] || immediate || 00. Por fim, percebe-se que agora os 26 bits do valor imediato permitem saltos dentro de um intervalo próximo de ± 64MB.

3.1.3 Chamada de Sistema

A instrução **Syscall** permite o uso de serviços oferecidos pelo OS (Operating System), geralmente relacionados a **Input** e **Output** de dados. Por padrão o serviço que indica a operação realizada pelo OS é armazenado no registrador “$v0”, enquanto os parâmetros utilizados são registrados em “$a0, $a1… “

Em sala, vimos alguns serviços de syscall como print string que registra 4 em $v0, enquanto o endereço da string a ser exibida é registrado em $a0. Outro exemplo é o serviço de open file que registra 13 em $v0, o endereço do arquivo a ser aberto em $a0, o modo de abertura (leitura, escrita, ambos) em $a1 e por fim o modo de permissão (geralmente 0) em $a2. Para resolução do problema proposto usamos serviços de open file, read from file, write to file, close file e terminate execution.

3.2 Registradores MIPS

O MIPS apresenta um total de 32 registradores principais divididos em alguns conjuntos com funcionalidades específicas. Dos principais alguns que valem ser citados são, $zero, $v0, $a0-$a3, $t0-$t9, $s0-$s7.

O registrador **$zero** possui o valor constante de zero, sendo comumente utilizado para comparações e garantir valores nulos.

O registrador **$v0** é usado para especificar o serviço de chamadas ao sistema (syscall), o que indica a operação a ser realizada.

Os registradores **$a0** até o **$a3** são usados como argumentos de funções como os serviços do syscall.

Os registradores **$t0** até o **$t9** são de propósito geral, comumente utilizados para guardar valores temporários.

Os registradores de **$s0** até **$s7** também são de propósito geral e podem ser usados para guardar valores temporários, mas além disso, é garantido que eles vão manter valores salvos entre chamadas de funções.

Existem ainda o **Coprocessador 0** e **Coprocessador 1**. O Coprocessador 0 esta relacionado ao gerenciamento de **exceções** e interrupções dentro de um código, registrando a resposta do sistema ao erro correspondente, como operações ou acessos inválidos. Já o Coprocessador 1 é usado para operações de **ponto flutuante** (float), pois seus registradores podem ser combinados para operações de dupla precisão com 64 bits.

4 Desenvolvimento e Resultados

Problema Proposto: Ordenar uma lista de numérica com 100 valores contidos em um arquivo chamado lista com extensão .txt.

Informações relevantes: Os casos de testes usarão números inteiros, separados por vírgula, sem espaço, em sistema decimal, com no máximo 3 dígitos. Exemplo: 1,3,2,50,300,-100,153,-554,7,10,11,99,215, …

Etapas:

a) Abra um arquivo “lista.txt” contendo os números.

b) Leia o arquivo e guarde-os na memória.

c) Implemente um algoritmo de ordenação e guarde os valores ordenados na memória.

d) Escreva a lista ordenada em um arquivo “lista_ordenada.txt”.

A princípio, foi decidido que o problema seria dividido em 3 partes a serem desenvolvidas separadamente: Leitura, Ordenação, Escrita.

Leitura:

Inicialmente, é preciso abrir o arquivo “lista.txt” e armazenar seu conteúdo em uma lista de caracteres. Em seguida é feito o processo de conversão dessa lista de caracteres em uma lista numérica. A conversão segue o seguinte raciocínio: para converter um dígito é subtraído seu valor com 0x30 (‘0’ em ASCII) e para casos de números com múltiplos dígitos, o atual (acumulado) é multiplicado por 10 e somado ao próximo dígito. Entenda o código:

```
INÍCIO:
    LEITURA ( registrar “lista.txt” em um buffer )
    LOOP para cada caractere em buffer
        carregar charAtual
        SE charAtual = ‘ , ‘ :
            PRÓXIMA ITERAÇÃO
        FIM SE
        SE charAtual = ‘ – ‘ :
            atribuir flagSign = 1
            atualizar charAtual para o próximo
        FIM SE
        inicializar acc como 0
        ENQUANTO charAtual for um dígito:
            acc = acc * 10 + ( charAtual – 0x30 )
            atualizar charAtual para o próximo
        FIM ENQUANTO
        SE flagSign = 1:
            acc = –acc
        FIM SE
        ATRIBUIR list_int(readIndex) = acc
        incrementar readIndex
    FIM LOOP
FIM LEITURA
```

Ordenação:

Para essa parte foi usado um dos algoritmos de ordenação mais simples de serem implementados, o **Bubble Sort**. O algoritmo consiste em percorrer a lista e verificar se uma posição pode ser ordenada com a posição seguinte, e após percorrer algumas vezes a lista pode-se então dizer que ela está ordenada. O código seguiu a seguinte estrutura:

```
INÍCIO:
    definir N como tamanho do array
    LOOP ( i ) de 0 a N – 4, a passos de 4:
        LOOP ( j )  de 0 a N – 4 – i, a passos de 4:
            SE array[ j ] > array [ j+4 ]:
                TROCA de array[ j ] com array [ j+4 ]
            FIM SE
        FIM LOOP j
    FIM LOOP i
FIM ORDENAÇÃO
```

Para reproduzir esse código em Assembly foram necessários alguns registradores reservados para armazenar valores como i, j, N, N – 4, N – 4 – i, array[j], array[j+4]. Note que foram usados incrementos de 4, visto que o array em questão vai ser formado pelas words do MIPS (4B).

Escrita:

Uma vez ordenado, só resta escrever o arquivo “lista_ordenada.txt” convertendo os números em string. A ideia por trás dessa etapa é a seguinte: percorrer todo array de números convertendo para string e em seguida ordenando a partir de um buffer para identificar o número na ordem correta e adicionar ao array da string que será escrita no arquivo de texto. A conversão de int para string é feita isolando os dígitos que compõe o número através de divisões sucessivas por 10 e somando o valor a 0x30 (‘0’ em ASCII), dessa forma cada dígito é convertido para o valor correspondente em ASCCI.

Todo o processo de escrita pode ser descrito da seguinte forma:

INÍCIO:

definir array_int e array_char como lista de inteiros e lista de caracteres

definir buffer como array para armazenar dígitos

LOOP para cada number em array_int:

carregar number atual

incrementar writeIndex

atribuir flagSign = 0

SE number < 0:

atribuir flagSign = 1

atribuir number = –number

FIM SE

ENQUANTO number != 0:

atribuir resto = number % 10

atribuir number = number / 10

atribuir character = resto + 0x30

acrescentar character ao buffer

FIM ENQUANTO

SE flagSign = 1:

acrescentar '–' ao buffer

FIM SE

ENQUANTO bufferIndex > 0:

decrementar bufferIndex

array_char[charIndex] = buffer[bufferIndex]

incrementar charIndex

FIM ENQUANTO

acrescentar de trás pra frente o buffer ao array_char

acrescentar ' , ' ao char_array para separar os números

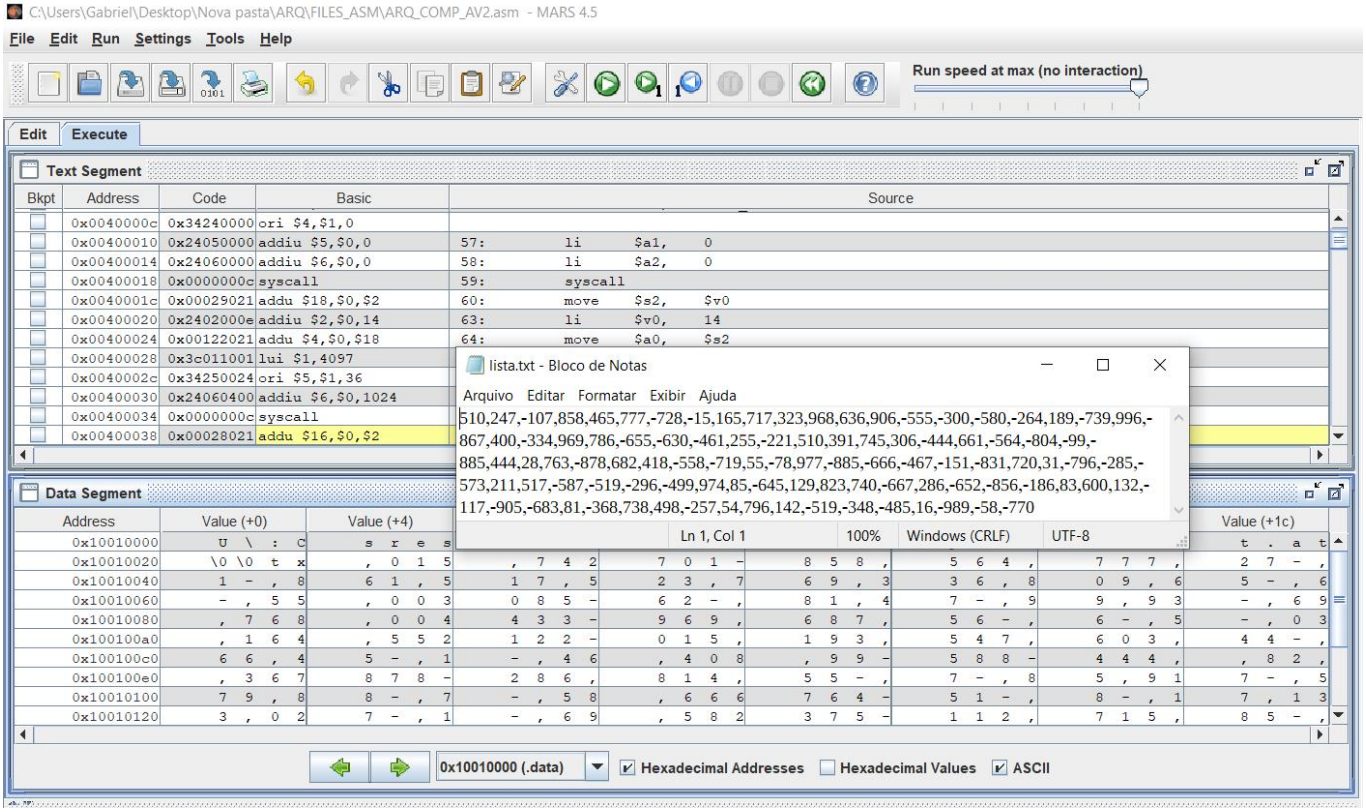
FIM LOOP writeIndex

ESCREVER (gravar array_char no arquivo "lista_ordenada.txt")

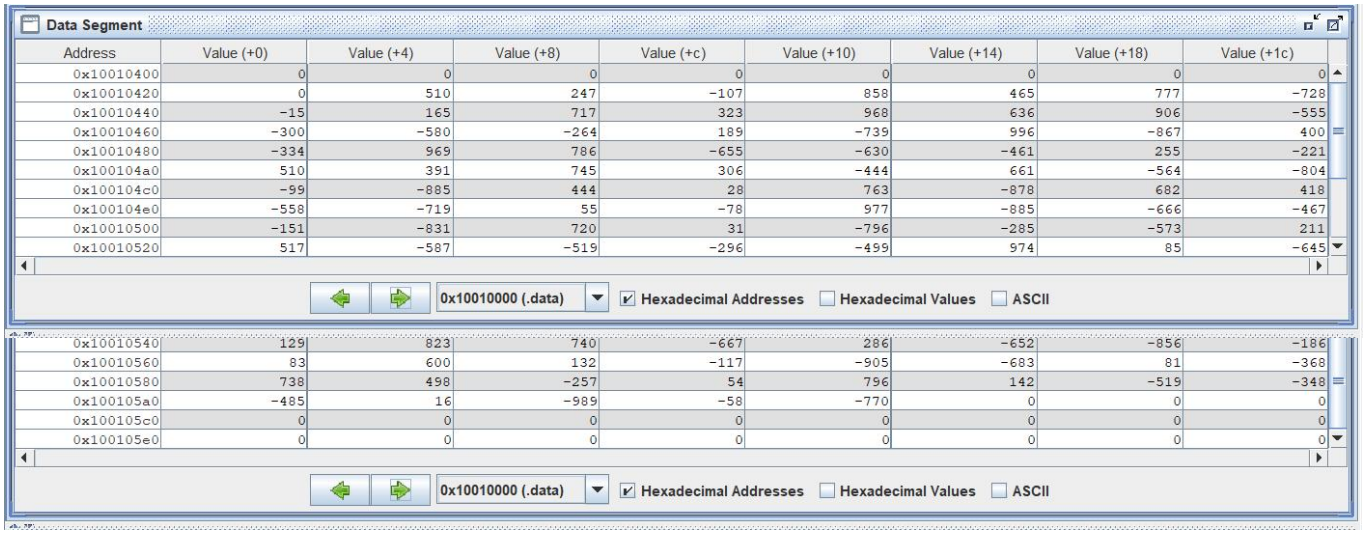
FIM ESCRITA

Código:
https://github.com/gabrieldsn2006/Unifor/blob/main/2oSemestre/ArquiteturaComp/sorting_text_file.asm

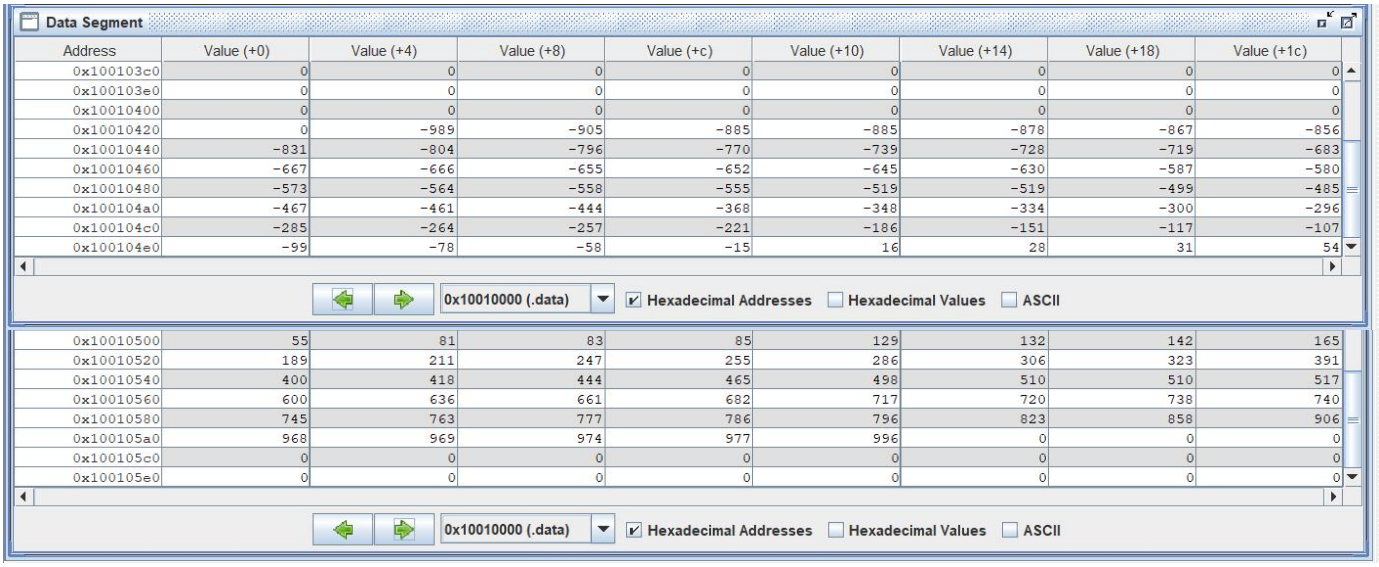
PRINTS:



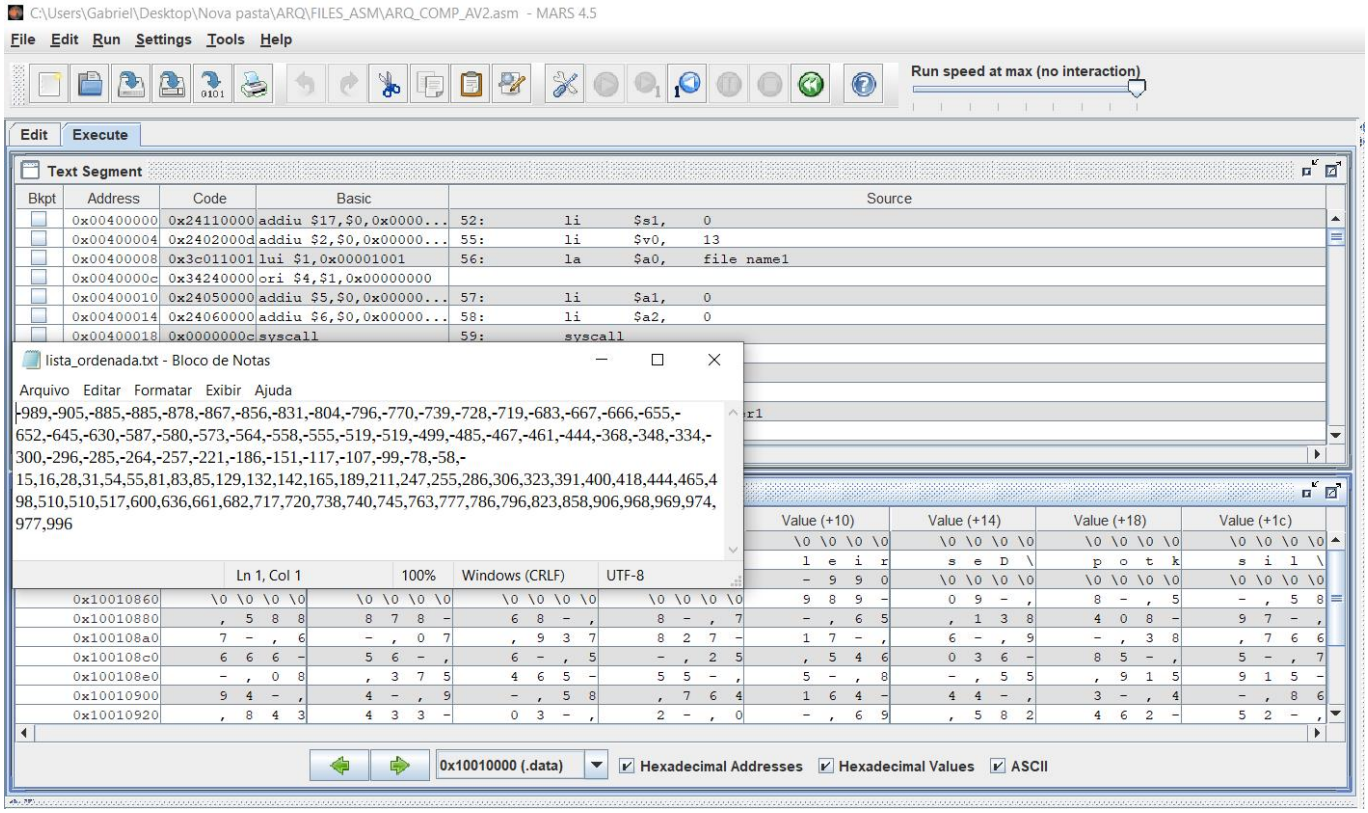
PRINT 1: Nessa imagem temos o registro do arquivo “lista.txt” em comparação com o espaço na memória alocado para leitura que foi feita do arquivo, ou seja, o espaço alocado para o buffer1. Marcando a visualização em ASCII podemos notar que os primeiros caracteres estão armazenados a partir do endereço 0x10010020 + 4. Nesse momento a memória ainda consta com a lista desordenada em caracteres.



PRINTS 2 e 3: Aqui é possível identificar o espaço na memória reservado para a lista já convertida de carateres para números inteiros. Nesse momento a lista numérica ainda está desordenada.



PRINTS 4 e 5: Após o algoritmo de bubble sort já possível observar a lista numérica ordenada corretamente no mesmo espaço de memória.



PRINT 6: já nessa imagem, finalizada a execução do programa, é possível observar e comparar o arquivo escrito “lista_ordenada.txt” e o espaço que foi separado na memória para lista de caracteres a serem escritos, ou seja o char_list. Note que os primeiros caracteres são armazenados a partir do endereço 0x10010860 + 10.

5 Conclusão

Finalizando, vale destacar a importância de desenvolver um trabalho como esse em Assembly. Quanto menor a abstração da linguagem, mais do programador é exigido sua capacidade de compreender e abstrair o problema proposto. Foi de extrema importância dividir o problema nas etapas de Leitura, Ordenação e Escrita, e ainda dividir cada uma delas em etapas mais específicas. Além disso, para resolução do problema foi essencial entender o funcionamento do processador e como cada instrução deve ser realizada por ele, como cada endereço de memória, registrador ou valor imediato deve ser manipulado corretamente. Por fim, pode-se destacar como o MIPS acabou se mostrando uma ótima ferramenta para o estudo de processadores e instruções de baixo nível na programação.