EPUSP - Escola Politécnica da Universidade de São Paulo
PCS3216 - Sistemas de Programação
2019

Projeto de implementação de um Motor de Eventos

Prof. Dr. João José Neto

Rafael Seiji Uezu Higa
NUSP: 9836878

1. Objetivo

O objetivo do projeto é a elaboração de um motor de eventos, capaz de executar simulações determinísticas de sistemas reativos.

Acessoriamente ao motor de eventos, espera-se a implementação de um conjunto de instruções tal que o motor de eventos implementado consiga executá-las, até mesmo compreendendo pequenos programas a serem determinados como casos de teste do motor.

2. Especificações Gerais e fundamentação teórica

Acerca do projeto, algumas especificações gerais são necessárias de se pontuar antes da implementação de quaisquer partes do projeto. Tais questões básicas são:

- O processador hospedeiro do motor de eventos;
- O hardware a se simular;
- Componentes do motor de eventos;
- Instruções a serem simuladas - a determinação de um conjunto de instruções o qual o motor de eventos compreenda e seus aspectos técnicos - funcionalidade de cada instrução e formato de tais instruções.

Juntamente às especificações, simplificações foram adotadas, entendendo-se se tratarem de itens que não agregariam tanto para o projeto no molde dos requisitos exigidos, permitindo a administração de esforços assim em outras partes do projeto. Ao longo do relatório, explicitam-se as especificações e eventuais simplificações detalhadamente.

2.1- Processador hospedeiro

Para a execução do projeto, foi escolhida a implementação de uma máquina virtual para a simulação do motor de eventos. Tal programa foi implementado através de programação em linguagem de alto nível. A linguagem utilizada para a implementação do projeto foi C, devido à familiaridade do aluno em tal linguagem, visto que fora ensinada utilizada em diversas disciplinas do curso de graduação em engenharia elétrica da POLI.

2.2- Hardware a se simular

Evidentemente, simula-se um hardware correspondente a um motor de eventos. Do motor de eventos, espera-se que:

- Seja capaz de executar pequenos programas, em um conjunto de instruções elaborado pelo aluno;
- Seja capaz de extrair os programas executados a partir de uma entrada de dados (input), dada através de um arquivo de texto (extensão .txt);
- Consiga executar as computações presentes em uma máquina de Turing, motivo pelo qual o conjunto de instruções implementado é um conjunto Turing-completo, como especificado na seção 4.

2.4- Componentes do motor de eventos

Alguns componentes importantes para o funcionamento do motor de eventos são:

- Memória;
- Registradores (de uso geral ou de uso específico);
- Equipamento de entrada e saída de dados;
- Blocos funcionais.

Sobre a memória:

- Da memória, espera-se a extração das instruções e operandos que alimentam o motor de eventos; além do armazenamento dos outputs decorrentes da execução das instruções pelo motor de eventos;
- A memória foi projetada para ter um espaço de endereçamento de **0000 a FFFF** (hexadecimal). Ou seja, foram adotados para endereçamento 16 bits. Considerando a baixa complexidade do programa, entendeu-se não ser necessário um grande espaço de armazenamento. Destaca-se o fato de que a memória foi implementada como um vetor com o número de casas equivalente ao tamanho de memória considerado. Em decimal, trata-se de um vetor de 65535 casas;
- Uma célula - ou seja, uma mínima unidade endereçável - foi tida como equivalente a um byte. Cada endereço portanto aponta para um byte de espaço de memória;
- Uma palavra é tida como equivalente a 4 bytes. isso se deve pelo tamanho das instruções implementadas, como se descreve na seção 4;
- Nota-se que, devido a simplicidade do projeto, não se preocupou com a separação da memória em regiões (como região para memória estática e memória dinâmica, por exemplo).

Sobre os registradores:

- Dos registradores auxiliares e registradores de uso geral, além do fato de auxiliarem na execução de instruções (afinal, são componentes que, excluindo-se o fato de se tratar de uma simulação de motor de eventos, numa implementação física do mesmo, se trataria de componentes de hardware), a análise dos registradores permite a determinação de estados do motor de eventos;
- Foram implementados oito registradores de uso geral, numerados de /01 a /08, a serem utilizados em instruções com endereçamento a registradores. Tais registradores armazenam operandos de instruções e os resultados delas decorrentes, ou seja, auxiliam na manipulação dos dados envolvidos nas operações executadas pelas instruções;
- Deve-se também considerar os registradores auxiliares. Estes têm funções específicas que auxiliam na operação do motor e execução das instruções. Trata-se do:
    - Program Counter (PC);
    - Stack Pointer (SP);
    - Frame Pointer (FP);
    - Return Address (RA).

Sobre os equipamentos de entrada e saída de dados:

Sobre a entrada de dados, implementou-se funções que executam papéis de loader - se tratam dos eventos artificiais com essa funcionalidade que foram sugeridos.

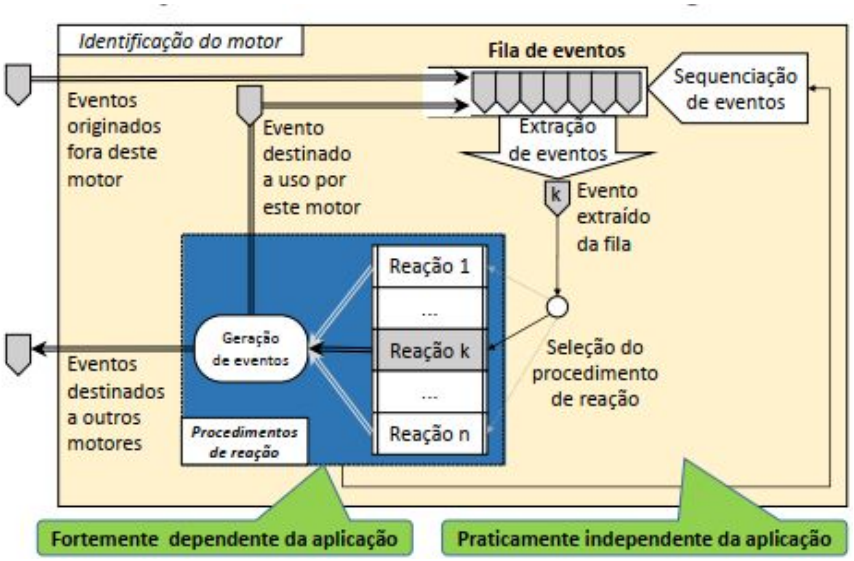
A respeito da saída de dados, através da exibição no console da aplicação o estado do motor de eventos, exibindo os valores nos registradores e os valores na memória. Como explicitado na seção 4, para isso utilizou-se um evento artificial, chamado “MD” - Memory Display, que exibe a partir de um endereço (operando desse evento artificial) o conteúdo de algumas posições de memória consecutivas e o valor armazenado nos registradores. MD foi implementado em substituição a um evento de Dumper, que fora sugerido no enunciado do projeto. Cumpre basicamente a mesma função de exibição na tela do console o conteúdo da memória, mas exibe o conteúdo de uma região da memória, de modo a deixar a exibição do conteúdo da memória mais confortável de se ler. Mais detalhes sobre MD constam na seção 4 do relatório.

Sobre blocos funcionais:

A despeito da utilização de uma linguagem de alto nível, que em termos técnicos, ou seja, em termos de implementação do projeto, não se fariam necessárias, por uma questão semântica, conferindo-se uma implementação coerente com o que seria na realidade, buscou-se implementar um bloco funcional capaz de executar operações aritméticas e um bloco capaz de executar operações lógicas. Se tratam de funções em C capazes de executar tais operações. Mais detalhes constam na seção 3 do relatório, em que são documentadas as funções implementadas para o projeto.

2.5- Extração de eventos e lista de eventos

Conceitualmente, a execução de eventos em um motor de eventos pode se dar como na figura abaixo, extraída dos materiais da disciplina, que em particular representa um motor de eventos de uso geral. Na figura, nota-se uma fita - na qual se dispõem em ordem as instruções do programa a ser executado; um extrator de eventos, que lê a fita; e um conjunto de procedimentos de reação aos eventos. A execução de cada um desses componentes, grosso modo, representa o funcionamento do motor de eventos.



No presente motor de evento, cada evento é equivalente a uma das instruções do programa a ser executado. Além desses eventos do programa - que são eventos independentes, tem-se eventos dependentes, que podem vir a ser inseridos na lista de eventos decorrentes da execução de outras instruções/eventos. Ou seja, tratam-se de eventos que podem ser inseridos na lista de eventos durante a execução de um programa.

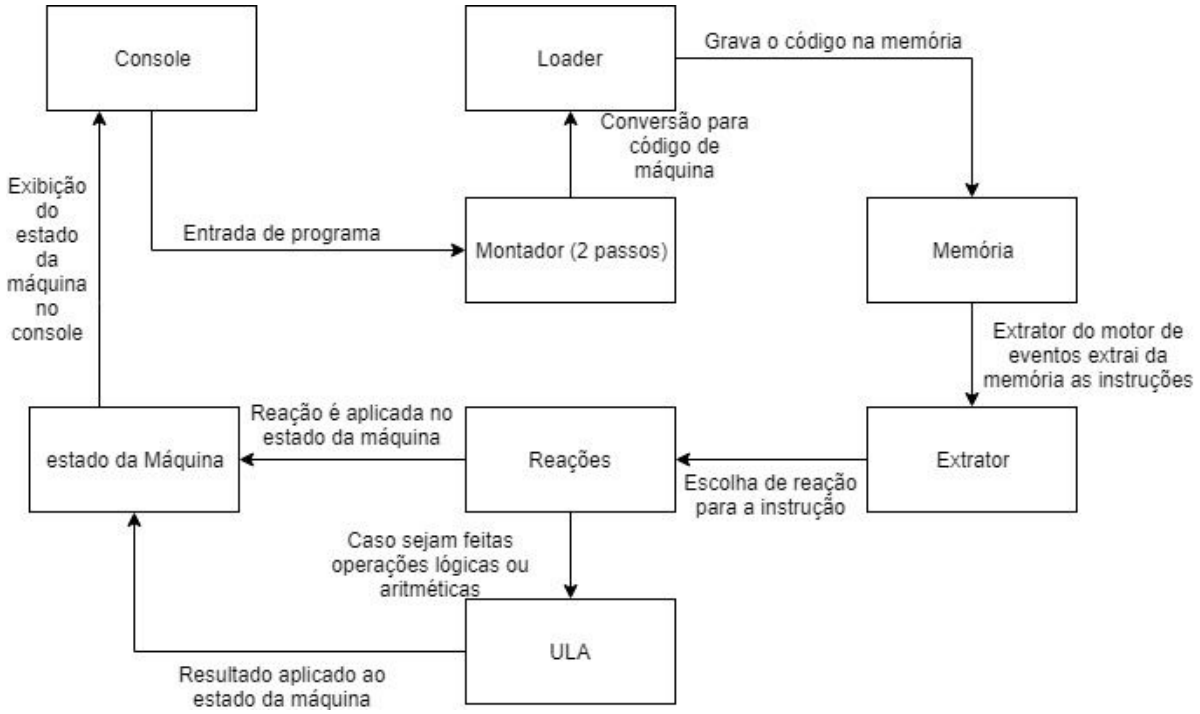
Os eventos independentes, ao invés de serem todos carregados na lista de eventos antes do início da execução do programa, como diz a teoria trabalhada em aula sobre motores de evento, serão lidos da memória um a um, com o auxílio do program counter. Tal fato é mais adequado para a aplicação em questão, vista a possibilidade de um trecho do código, graças à presença de loops e saltos, ser executado mais de uma vez ou nenhuma vez.

Eventos dependentes, por sua vez, podem vir a ser inseridos na lista de eventos a qualquer instante, dado um pedido vindo durante a execução do programa.

Destacam-se também os eventos artificiais sugeridos no enunciado do projeto. Estes não fazem parte dos códigos em linguagem simbólica implementados pelo aluno e lidos como entrada do motor. Ao invés disso, são inseridos a mão pelo usuário, no console, de forma totalmente separada do programa inserido como entrada do dado motor de eventos. O uso desses eventos artificiais ficará mais claro e elucidado na seção 4.

3. Estrutura do Projeto

Um diagrama de blocos segue abaixo, explicitando de uma maneira abrangente o funcionamento do motor de eventos. Os tópicos destacados nesse diagrama de bloco são implementados por variáveis, funções e structs em C. Abaixo do diagrama de blocos, destacam-se as structs e funções implementados e uma relação dos mesmos com o diagrama de blocos abaixo.



Structs implementados:

```
21  struct instructionEmMnemonico {
29
30  struct instructionEmHexadec {
37
38  struct allInstructions {
45
46  struct estadoMaquina {
53
54  struct evento {
58
59  struct listaLigada {
62
63  struct labelTable {
67
```

Struct	instructionEmMnemonico
Parâmetros	`struct instructionEmMnemonico {` `  char *label;` `  char *mnemonico;` `  char *operando1;` `  char *operando2;` `  char *operando3;` `  char *comments;` `};`
Descrição	Esse struct serve para a leitura de um programa. Cada linha de um programa é armazenada em um struct instructionEmMnemonico, onde a label é armazenada no

	char* label, o mnemônico, em char *mnemonico e assim por diante.

Struct	instructionEmHexadec
Parâmetros	`struct instructionEmHexadec { int *mnemonico; int *operando1; int *operando2; int *operando3; int *comments; };`
Descrição	Trata-se do programa convertido para código hexadecimal. Assim, ao invé s de um char com o nome do mnemônico, aqui se tem um campo do tipo int com o código do mnemônico. Idem para os operandos. Nota-se que aqui labels se fazem desnecessárias, já tendo sido resolvidas em outra parte do programa. Por isso que não há nesse struct uma variável para armazenar a label atribuída a essa linha de código.

Struct	allInstructions
Parâmetros	`struct allInstructions { struct instructionEmHexadec instrucaoHexadecimal[500]; struct instructionEmMnemonico instrucaoMnemonico[500]; int inicio; //inicio do codigo, como um endereco de int fim; //fim do codigo, comouma posição de arra };`
Descrição	tem como parâmetros vetores de structs em mnemônicos e também convertidos em hexadecimal. O intuito de se tratar de vetores de struct é justamente armazenar todo o programa que foi tido como entrada. Além disso, esse struct também tem como entradas os endereços de início e de fim do código. Assim, a função geral do struct allInstruction é a determinação de aspectos gerais que tangem à execução da instrução - sua conversão para o formato adequado (hexadecimal) e a determinação dos pontos de início e fim da execução do código.

Struct	estadoMaquina

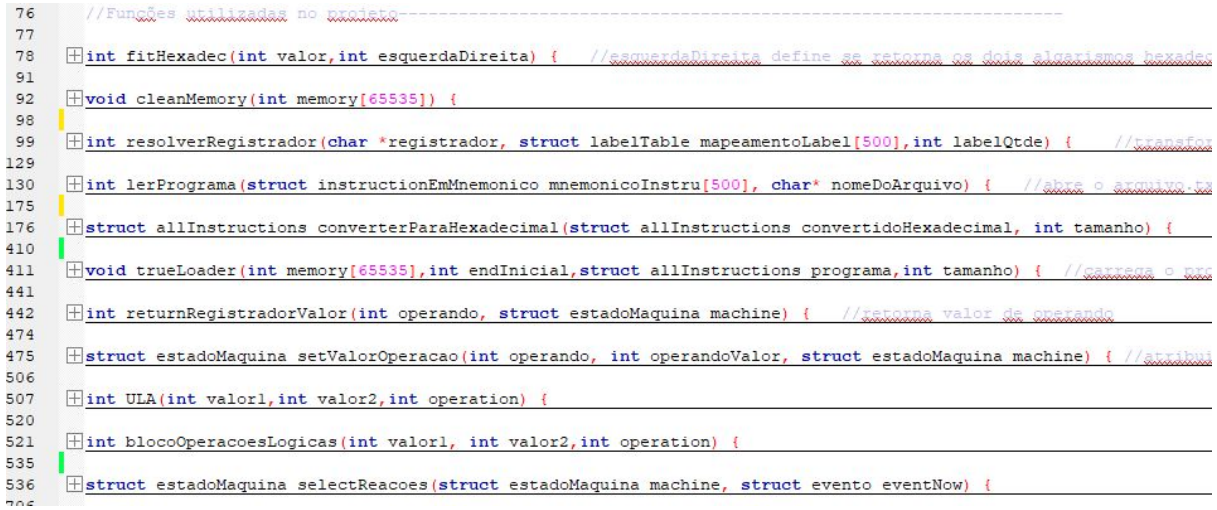
Parâmetros	`struct estadoMaquina { int memory[65535]; int R1,R2,R3,R4,R5,R6,R7,R8; int PC,SP,FP,RA; int enable; int isTrace; };`
Descrição	Contém a memória e os registradores. Envolve componentes que caracterizam o estado da máquina. Possui também uma variável isTrace, que determina se o modo trace está ativado ou não. Mais detalhes sobre o modo trace constam na seção 4.

Struct	Evento
Parâmetros	`struct evento { int instrucao[4]; };`
Descrição	Se trata de um evento a ser inserido na lista de eventos. o vetor int instrucao[4] é tal que na posição 0 seja armazenado o código do mnemônico, e nas posições 1, 2 e 3 sejam armazenados, se houverem, os operandos da instrução.

Struct	listaLigada
Parâmetros	`struct listaLigada { struct evento instruEvento[500]; };`
Descrição	Esse struct se trata de uma lista de eventos, daí se possuir como parâmetro um vetor de struct evento.

Struct	labelTable
Parâmetros	`struct labelTable { char* label; int labelAddress; };`
Descrição	Esse struct tem como parâmetros uma label e o endereço ao qual ela referencia. Um vetor com structs desse tipo acaba por funcionar como uma tabela de mnemônicos.

Funções implementadas:



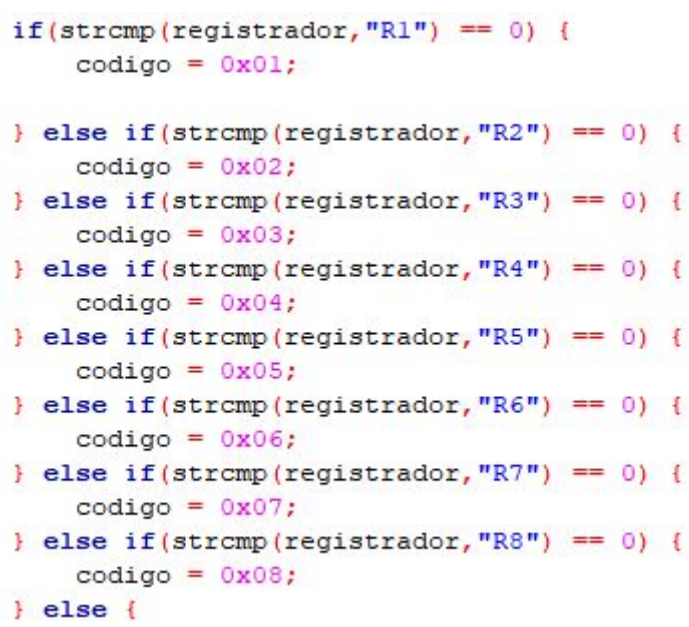
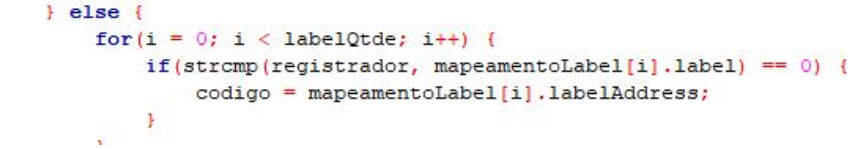
Especificação de cada função:

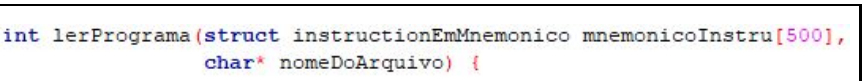
Função	`int fitHexadec(int valor,int esquerdaDireita)`
Especificação	função auxiliar para posicionar adequadamente nos 32 bits de instrução os dados nas instruções BEQ e BNE. Nessas instruções, os campos “operando 1”, “operando 2” e “operando 3” deixam de fazer sentido, em virtude do aproveitamento dos 32 bits de instrução. o formato dessa instrução é melhor detalhado na seção 4. Com essa instrução, dado um valor de 4 algarismos hexadecimais, é possível extraír os 2 mais significativos ou os 2 menos significativos.
Implementação	`int fitHexadec(int valor,int esquerdaDireita) {` `  int resultado;` `  resultado = valor >> 8;` `  if(esquerdaDireita == 1) {` `  resultado = resultado << 8;` `  resultado = valor - resultado;` `  }` `return resultado;` `}` Aqui, a variável “valor” indica o valor sobre o qual se aplicará um “shift”. • Se a variável “esquerdaDireita” for diferente de 1 conforme a implementação acima, primeiramente é feito shift para a direita, de 8 bits (equivalentes a dois algarismos hexadecimais, ou um byte, ou um endereço de memória), obtendo-se um valor deslocado de um byte

	para a direita. Assim, se obtém os dois algarismos hexadecimais mais significativos da variável “valor”; ● Se a variável “esquerdaDireita” for igual a 1, primeiramente se faz o shift para a direita, tal como no caso anterior, mas em seguida, se faz novamente um shift para a esquerda e se subtrai de “valor”. Isso permite que seja possível se obter os dois algarismos hexadecimais menos significativos da variável “valor”

Função	`void cleanMemory(int memory[65535])`
Especificação	Função que faz todos os endereços da memória conterem o valor zero. Foi uma das especificações do projeto a definição de um valor inicial para as posições de memória, que então foi escolhido como zero.
Implementação	`void cleanMemory(int memory[65535]) {` `  int i;` `  for(i = 0; i < 65535; i++)` `  memory[i] = 0;` `}` Somente faz com que todos os endereços de memória tenham seu conteúdo igualado a 1. Essa função é executada no começo do funcionamento do motor de eventos, para garantir que o valor inicial de todas as posições de memória seja nulo. Lembra-se também que 65535 equivale emhexadecimal a FFFF, daí o motivo de tal número como tamanho do vetor que representa a memória do motor de eventos.

Função	`int resolverRegistrador(char *registrador,` `  struct labelTable mapeamentoLabel[500],int labelQtde)`
Especificação	Resolve a codificação dos operandos de uma instrução. Assim, todas as referências a registradores, que são escritas por símbolos, na forma “R1”, “R2”, etc, são substituídas pelos seus respectivos códigos (ao final dessa seção, há uma tabela com os códigos dos registradores de uso geral). Além disso, essa função também resolve a questão das labels: troca-as pelo endereço de memória da instrução referenciada por tal label.

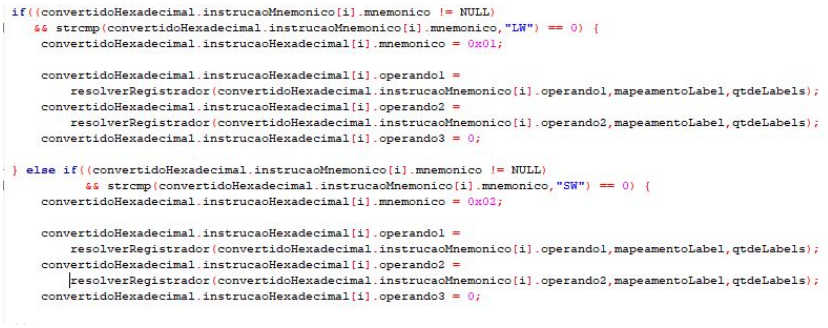
Implementação	 As condicionais para determinação de um código correspondente a um operando que seja registrador é simplesmente uma comparação de strings, utilizando-se da função “strcmp”, que retorna 0 caso as strings sejam iguais. Nessa função, encontra-se também um “else”, omitido na imagem acima e aqui reproduzido em seguida:  Caso, no campo de operandos da instrução digitada em assembly, fora digitada uma label, essa ficou armazenada em um vetor, associada também a um endereçamento. Esse vetor também é um dos argumentos dessa função, e é usado no “else” acima representado a fim de substituir as labels pelo seu “código”, ou seja, pelo seu endereço no programa.

Função	
Especificação	faz a leitura do programa o qual foi aplicado como entrada do motor de eventos, e o armazena no struct mnemonicoInstru[500] (nota-se que essa função retorna o valor desse vetor de structs atualizado). Ou seja, essa função lê o arquivo .txt com o programa e armazena seu conteúdo, ainda na forma de mnemônicos, num vetor de structs.
Implementação	Essa função faz a leitura de um programa escrito na linguagem simbólica implementada. A função faz a abertura de um arquivo em modo de leitura:

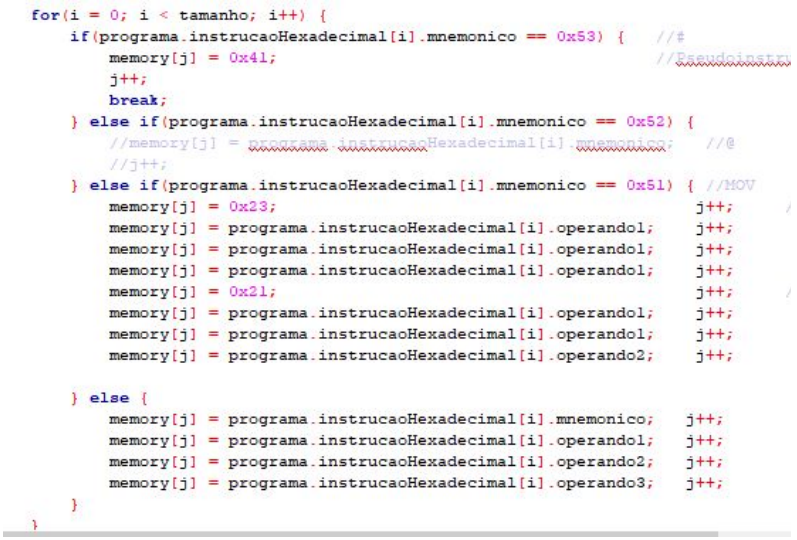
```
arquivo = fopen(nomeDoArquivo, "r");

while(!feof(arquivo)) {
    fgets(line, 90, arquivo);
    if(line[0] == '\n') {
        i -= 1;
    } else if(line[0] == ' ' || line[0] == '\t') {   //no label
        mnemonicoInstru[i].label = NULL;

        mnemonicoInstru[i].mnemonico = strdup(strtok(line,", \n\t"));

        mnemonicoInstru[i].operando1 = strdup(strtok(NULL,", \n\t"));

        mnemonicoInstru[i].operando2 = strdup(strtok(NULL,", \n\t"));

        mnemonicoInstru[i].operando3 = strdup(strtok(NULL,", \n\t"));

.

     } else {
         mnemonicoInstru[i].label = strdup(strtok(line,", \n\t"));

         mnemonicoInstru[i].mnemonico = strdup(strtok(NULL,", \n\t"));

         mnemonicoInstru[i].operando1 = strdup(strtok(NULL,", \n\t"));

         mnemonicoInstru[i].operando2 = strdup(strtok(NULL,", \n\t"));

         mnemonicoInstru[i].operando3 = strdup(strtok(NULL,", \n\t"));

     }
     i++;
 }
```

Em seguida, faz a leitura do arquivo, armazenando no vetor de structs “mnemonicoInstru” (onde cada casa desse vetor é uma linha do arquivo em linguagem simbólica lido) o conteúdo do arquivo, em um formato: os campos desse struct tem como parametros variáveis tipo string “label”, “operando1”, “operando2” e operando3”. Varrendo-se o código em ordem, supondo que não haja erros de escrita em tais códigos,os respectivos campos são armazenados nas respectivas variáveis do struct conforme o laço if-else acima representado.
Nota-se que o caso “if” representa a situação em que uma linha não tem label escrita, e o caso “else” representa uma situação em que a linha de código tem label escrita.
para fazer essa leitura, nota-se também a função “strdup”, que converte todos os caracteres lidos até que se encontre um dado caractere especificado, como foi feito no trecho de código acima.

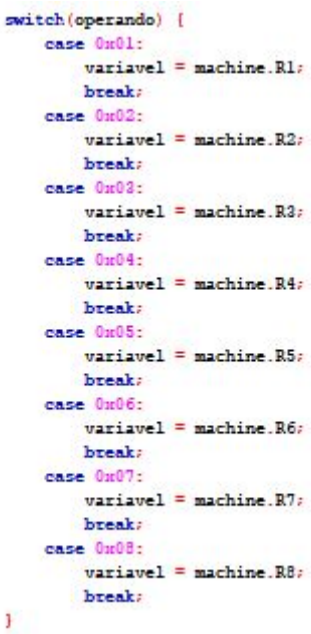
Função	`struct allInstructions converterParaHexadecimal (struct allInstructions convertidoHexadecimal, int tamanho)`
Especificação	converte o programa de entrada do motor de eventos da forma em mnemônicos para codificado em hexadecimal. Essa conversão se dá no struct allInstructions, que possui um vetor com as linhas do programa em mnemônico e um vetor com as linhas do programa a serem definidas em hexadecimal, como já

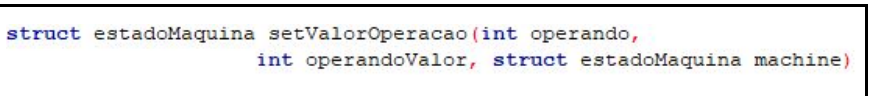
	documentado a respeito dos structs utilizados no projeto.
Implementação	Não será reproduzido esse código integralmente aqui, devido a seu tamanho, mas seguem pequenos trechos do mesmo a fim de indicar seu funcionamento:  Laços if-else identificam strings coletados na função “lerPrograma” anteriormente citada, e convertem todos os elementos de uma linha do código (tem-se acesso de forma organizada a especificamente uma linha do código graças ao vetor instructionEmMnemonico montado na função “lerPrograma”), ou seja, mnemonico e operandos (seja registradores, seja labels, seja endereços, seja valores imediatos), para uma codificação em hexadecimal, a qual será gravada na memória (obs: evidentemente, a memória grava números binários, mas o fato de que um endereço de memória corresponde a um byte, ou dois algarismos hexadecimais, facilitou o trabalho com variáveis em bytes, e não em números binários. Conceitualmente, compreendendo-se que 1 byte equivale a 8 bits, não há grandes prejuízos em se trabalhar com codificação hexadecimal em detrimento da binária, porém). Nota-se que fazer tais laços “if-else” adaptados para cada instrução é importante, pois cada uma delas exige uma compreensão diferente dos operandos, visto que há diferentes tipos de operandos (registrador, label, endereçamento direto, valores imediatos), e em diferentes quantidades (há instruções que possuem 3 operandos, como um “BEQ”- branch if equal; e há instruções que possuem 2 operandos, como “JAL” (Jump And Link); há instruções que possuem somente um operando, como “J”(Jump); e há instruções que não possuem nenhum operando (como “RTS”).

Função	`void trueLoader(int memory[65535],int endInicial,` `struct allInstructions programa,int tamanho)`
Especificação	Executa o papel de um loader, e grava na memória o programa já convertido em hexadecimal. Destaca-se que aqui foi feita a simplificação de se ter implementado o loader em linguagem de alto nível. Se o procedimento formal tivesse sido rigorosamente seguido, o loader na realidade deveria ter sido implementado em

	linguagem de máquina. O loader também não foi tido como um programa armazenado na memória, e sim somente uma etapa do evento artificial sugerido no enunciado do projeto, que executa a função de loader. Diante da simplicidade do projeto, entendeu-se que a implementação da funcionalidade de loader, ou seja, gravar na memória o programa já convertido em linguagem de máquina, já estava suficiente.
Implementação	 Essa função, através de um laço for (uma iteração para cada linha do programa), executa um condicional “if-else” para gravar na memória o programa já codificado em formato gravável na memória pela função “converterParaHexadecimal” anteriormente citada. As pseudoinstruções #, MOV e @, por serem pseudoinstruções, não são gravadas na memória, e algo equivalente é gravado na memória em seus lugares (no caso de #, é a instrução HM; no caso de @, nada é gravado; e no caso de MOV, as instruções SUB (para limpar o conteúdo de um registradro) e ADD são inseridas na memória.

Função	`int returnRegistradorValor(int operando, struct estadoMaquina machine)`
Especificação	Chaveia o registrador de uso geral a ser acesso de acordo com o código de registrador de operando citado na instrução. O valor de retorno dessa função é o valor do registrador chaveado.

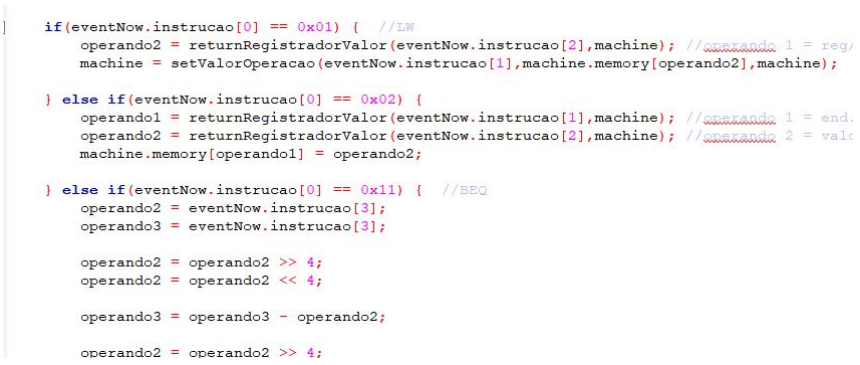
Implementação	 Simplesmente, um “Switch case” faz a leitura de um campo de operando e determina a qual registrador o código se referencia, e assim permite, no motor de eventos, a execução da operação em pauta devidamente.

Função	
Especificação	Assim como a função returnRegistradorValor, chaveia o registrador mencionado na instrução como operando, porém agora altera o valor do registrador, e não o retorna.
Implementação	 Assim como na função “returnRegistradorValor”, utiliza-se de um switch-case para o acesso dos registradores do motor de eventos.

Função	`int ULA(int valor1,int valor2,int operation)`
Especificação	Executa operações aritméticas: soma, subtração, multiplicação e divisão, e retorna o resultado da operação.
Implementação	```int ULA(int valor1,int valor2,int operation) { int resultado; if(operation == 1) resultado = valor1 + valor2; else if(operation == 2) resultado = valor1 - valor2; else if(operation == 3) resultado = valor1 * valor2; else if(operation == 4) resultado = valor1 / valor2; return resultado; }``` Essa função, apesar do nome, meramente executa operações aritméticas. Determinam-se os operandos como argumentos da função e a operação a ser executada, também como argumento da função. De acordo com o valor da variável “operation”, determina-se qual operação aritmética será executada, através do uso de condicionais if - else if.

Função	`int blocoOperacoesLogicas(int valor1, int valor2,int operation)`
Especificação	Executa operações lógicas: AND, OR, XOR e NOT
Implementação	Essa função é a responsável pela execução das operações lógicas que foram implementadas nesse projeto. ```int blocoOperacoesLogicas(int valor1, int valor2,int operation) { int result; if(operation == 1) { //AND result = valor1&valor2; } else if(operation == 2) { //OR result = valor1|valor2; } else if(operation == 3) { //XOR result = valor1^valor2; } else if(operation == 4) { //NOT result = ~valor1; } return result; }``` Tal como para as operações aritméticas, tem-se como argumentos da função dois valores sobre os quais será feita a

	operação lóigca, e uma variável “operation”, que juntamente a condicionais if e else if, determinam qual será a operação a ser executada. O resultado da operação lógica é parâmetro de retorno da função.

Função	`struct estadoMaquina selectReacoes (struct estadoMaquina machine, struct evento eventNow)`
Especificação	Se trata do conjunto de reações descrito no motor de eventos. De acordo com a instrução extraída da lista de eventos, escolhe uma dessas reações e a executa.
Implementação	Não será reproduzido aqui todo o código, pois este é deveras extenso. Cabe destacar, porém, que o código basicamente se trata de condicionais if-else, em que para cada instrução é executado um desses condicionais somente. Dentro de cada condicional, está o procedimento de reação necessário a ser executado, personalizado para cada instrução. Abaixo, um pequeno trecho do código:  Nota-se que o valor do código da instrução é analisado como condição dos “if-else”, e que alguns comandos, personalizados para cada instrução, são executados. Essa função “selectReacoes” é melhor explorada na seção 4, que documenta detalhadamente cada um desses procedimentos de reação conforme cada instrução. Ou seja, cada um desses condicionais “if-else” será detalhado, destacando-se seu funcionamento para cada instrução..

Quanto a função main, contém o loop do motor de eventos e também condicionais que preveem a inserção na lista de eventos dos eventos artificiais implementados, cujo funcionamento nos programas de entrada no motor de eventos não está previsto.

É importante ressaltar que a referência a registradores se deu através de códigos hexadecimais. É necessária a definição de uma codificação para os registradores, uma vez que a estes há referência nas instruções. Assim, aos registradores de uso geral 1 a 8 foram, simplesmente, atribuídos os valores hexadecimais /01 a /08, como documentado na tabela abaixo.

Registrador	**Código**

R1	/01
R2	/02
R3	/03
R4	/04
R5	/05
R6	/06
R7	/07
R8	/08

4. Instruções

Foi implementado um conjunto de instruções que se compusesse em uma linguagem turing-completa. Ou seja, trata-se de um conjunto de instruções capaz de codificar quaisquer operações possíveis de ser executadas em uma máquina de Turing.

Assim, os seguintes requisitos buscaram ser capazes de serem realizados pelo conjunto de instruções implementado:

- Instruções de referência a memória;
- Instruções de referência a registradores;
- Instruções de controle de fluxo;
- Instruções aritméticas e lógicas;
- Instruções de entrada e saída;
- Instruções de controle do estado da máquina.

O conjunto de instruções implementado foi inspirado tanto nas instruções utilizadas na presente disciplina como no conjunto de instruções da família MIPS de processadores, trabalhada na disciplina de Organização e Arquitetura de computadores.

Além disso, destaca-se inclusive o plano de codificação para cada uma das instruções - codificação essa a ser implementada na tabela de mnemônicos adotada para o motor de eventos em questão. A codificação das instruções está na forma /YZxxxx, onde os YZ's são algarismos hexadecimais que codificam as instruções, e xxxx correspondem a operandos, com endereçamento de acordo com a instrução em questão.

O tamanho de todas as instruções foi implementado como 4 bytes, ou 32 bits, decisão tomada em virtude da simplificação ganha na execução das instruções, sobretudo no que se refere ao program counter.

Os 4 bytes remetem a maior necessidade de tamanho de instrução demandada pelo dado conjunto de instruções. As instruções BEQ (Branch if Equal) e BNE (Branch if not Equal) exigem necessariamente o uso de 4 bytes.

Quanto às demais instruções, para os fins aqui levados em conta - o conteúdo de uma instrução somente ter o código da instrução e os operandos - precisam de menos de 4 bytes, embora em aplicações reais, que levariam em conta aspectos mais profundos, como o opcode de identificação do tipo da instrução, por exemplo, 32 bits de tamanho fariam sentido.

Entretanto, o “desperdício” de bits no processo de implementação de instruções foi compensado uma vez que a implementação da execução de instruções com saltos foi deveras facilitada.

Nesse contexto, um operando é armazenado em cada byte da instrução, e assim, o acesso aos operandos de instrução pode ser feito com o endereço de memória do byte em questão.

Já na situação já mencionada das instruções BEQ e BNE, devem ser armazenados nos 32 bits da instrução: o código do mnemônico (de 8 bits); o endereço para o qual se fará o salto (de 16 bits); e os dois operandos os quais se fará a comparação de igualdade ou desigualdade (4 bits cada um). Nesse panorama, o endereço para o qual se fará o salto é armazenado nos campos de operando 1 e operando 2 da instrução (campos como mencionado nos structs que guardam uma instrução em hexadecimal: há um valor int para o operando, e um valor int para cada um dos operandos 1, 2 e 3); e os dois operandos dessa instrução são comprimidos no campo de operando 3.

Faz-se a ressalva também de que as instruções BEQ, BRZ e BNE permitem saltos a todos os endereços da memória. Em uma aplicação real, com memórias com espaço de endereçamento muito maior, seria inviável armazenar dentro de uma instrução um endereço completo, fato pelo qual essas instruções, inclusive na própria família MIPS, não armazenam um endereço ao qual saltar, mas sim um offset em relação ao endereço atual do Program Counter. Como simplificação, uma vez que a memória tem espaço de endereçamento somente de 0000 a FFFF, optou-se por desprezar o uso de offsets nessas instruções, e se armazenar o próprio valor da instrução.

Quanto a implementação dessas instruções, ou seja, a programação de suas devidas funcionalidades, essas se deram na função “selectReacoes”, que tem como entrada um “struct estadoMaquina”, que possui como parametros os registradores e a memória; e um “struct evento”, que diz o evento que está sendo processado. De acordo com o evento sendo processado, através de um laço if...else if...else if…….else, a reação correta é executada, e o estado da máquina é alterado.

/01xxxx - Load World	
Categoria	Instruções de referencia a memória
Mnemônico	LW
Endereçamento	Registrador Direto
Operandos	R1, R2 (registradores)
Exemplo	LW R1,R2
Descrição	acessa o conteúdo de R1 no endereço especificado no registrador R2
Codificação	/01xxxx
Implementação	
Primeiramente, obtém-se o endereço que será acesso; e em seguida esse valor é salvo no devido registrador através da função “setValorOperacao”.	

```
if(eventNow.instrucao[0] == 0x01) {  //LW
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine); //operando 1 = reg/ ;
    machine = setValorOperacao(eventNow.instrucao[1],machine.memory[operando2],machine);
```

/02xxxx - Store World	
Categoria	Instruções de referencia a memória
Mnemônico	SW
Endereçamento	Registrador Direto
Operandos	R1,R2 (registradores)
Exemplo	SW R1,R2
Descrição	armazena em R1 o conteúdo do endereço especificado no registrador R2
Codificação	/02xxxx
Implementação	

Simplesmente se extraem os valores dos dois registradores mencionados na digitação da instrução. Um deles é o valor que será salvo e o outro é o endereço onde o valor será salvo.

```
} else if(eventNow.instrucao[0] == 0x02) {
    operando1 = returnRegistradorValor(eventNow.instrucao[1],machine); //operando
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine); //operando
    machine.memory[operando1] = operando2;
```

/11xxxx - Branch if equal	
Categoria	Loops, controle de fluxo, desvios e chamadas de procedimento
Mnemônico	BEQ
Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	BEQ R1,R2,R3
Descrição	faz um salto para o endereço armazenado em R1 caso os valores armazenados em R2 e R3 sejam iguais
Codificação	/11xxxx
Implementação	

A lógica é a mesma da instrução BNE citada mais abaixo dessa instrução, a menos de uma comparação de igualdade de operandos, e não desigualdade.

```
} else if(eventNow.instrucao[0] == 0x11) {  //BEQ
    operando2 = eventNow.instrucao[3];
    operando3 = eventNow.instrucao[3];

    operando2 = operando2 >> 4;
    operando2 = operando2 << 4;

    operando3 = operando3 - operando2;

    operando2 = operando2 >> 4;

    if(operando2 == operando3) {
        eventNow.instrucao[1] = eventNow.instrucao[1] << 8;
        eventNow.instrucao[1] += eventNow.instrucao[2];
        machine.PC = eventNow.instrucao[1];
    }
```

/12xxxx - Branch if zero	
Categoria	Loops, controle de fluxo, desvios e chamadas de procedimento
Mnemônico	BZR
Endereçamento	Registrador Direto
Operandos	R1,R2
Exemplo	BZR R1,R2
Descrição	faz um salto para o endereço armazenado em R1 caso o valor armazenado em R2 seja zero.
Codificação	/12xxxx

Implementação

Extrai-se o valor do registrador para fazer a comparação com zero, com a função “returnRegistradorValor”. Faz-se a comparação do dado valor com zero, e caso ela se confirme, o endereço de salto é extraído e gravado no Program Counter. A lógica da obtenção desse endereço de salto é explicada melhor nas instruções a seguir (BNE e jumps)

```
} else if(eventNow.instrucao[0] == 0x12) {  //BZR
    operando2 = returnRegistradorValor(eventNow.instrucao[3],machine);
    if(operando2 == 0) {
        eventNow.instrucao[1] = eventNow.instrucao[1] << 8;
        eventNow.instrucao[1] += eventNow.instrucao[2];
        machine.PC = eventNow.instrucao[1];
    }
```

/13xxxx - Branch if not equal	
Categoria	Loops, controle de fluxo, desvios e chamadas de procedimento
Mnemônico	BNE

Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	BNE R1,R2,R3
Descrição	faz um salto para o endereço armazenado em R1 caso os valores armazenados em R2 e R3 não sejam iguais.
Codificação	/13xxxx

Implementação

Devido ao tamanho da instrução BNE, os dois operandos os quais será comparada a desigualdade foram confinados num único byte, É necessário separá-los para poder fazer a comparação.
Assim, os operandos 2 e 3 no código abaixo inicialmente são iguais, pois ambos tiveram seus valores extraídos da mesma variável.
Em seguida o operando 2 é deslocado em 1 algarismo hexadecimal para a direita, e redeslocado para a esquerda em um algarismo hexadecimal (eliminou-se o operando a direita dessa forma).
Subtrai-se do operando 3 o valor do operando 2, e obtém-se o operando mais a direita na variável operando 3.
Faz-se novamente o shift na variável operando2 e assim é obtido o operando mais a esquerda.
Está possível a realização de comparação entre os dois operandos dessa instrução. Nota-se que é feito um ajuste do endereço de salto, que estava ocupando dois bytes. Foi necessário um shift no s 2 algarismos mais significativos e uma soma com o menos significativo para que de fato o endereço de salto fosse reconhecido como um número. Antes, afinal, estava como duas partes salvas em campos distintos da instrução.

```
} else if(eventNow.instrucao[0] == 0x13) {   //BNE
    operando2 = eventNow.instrucao[3];
    operando3 = eventNow.instrucao[3];

    operando2 = operando2 >> 4;
    operando2 = operando2 << 4;

    operando3 = operando3 - operando2;

    operando2 = operando2 >> 4;

    if(operando2 != operando3) {
        eventNow.instrucao[1] = eventNow.instrucao[1] << 8;
        eventNow.instrucao[1] += eventNow.instrucao[2];
        machine.PC = eventNow.instrucao[1];
    }
```

/14xxxx - Jump	
Categoria	Instruções de controle de fluxo
Mnemônico	J
Endereçamento	Direto
Operandos	<label> ou <endereço>
Exemplo	J R1
Descrição	faz um salto ao endereço do operando. O valor do Program Counter é alterado para esse valor de operando.

Codificação	/14xxxx

Implementação

o endereço para o salto não cabe em somente 1 byte, e por isso é gravado em 2 bytes da instrução, necessitando-se o uso do campo de operando 1 e também o campo de operando 2. Assim, o campo de operando 1 é deslocado de 1 byte e somado ao campo de operando 2, para que seja formado o endereço de salto. Feito esse procedimento, o valor do endereço de salto é salvo no Program Counter.

```
} else if(eventNow.instrucao[0] == 0x14) {  //J
    eventNow.instrucao[1] = eventNow.instrucao[1] << 8;
    eventNow.instrucao[1] += eventNow.instrucao[2];
    machine.PC = eventNow.instrucao[1];
```

/15xxxx - Jump And Link

Categoria	Instruções de controle de fluxo
Mnemônico	JAL
Endereçamento	<label> ou <endereço>
Operandos	R1
Exemplo	JAL <label>
Descrição	faz um salto para o endereço no operando, ao se armazenar tal valor no Program Counter. O valor (Program Counter + 4) é armazenado no registrador RA(assim, quando o fluxo do programa voltar a rotina que chamou uma subrotina pelo comando JAL voltará na instrução seguinte ao JAL, e não na própria instrução JAL).
Codificação	/15xxxx

Implementação

Primeiramente, salva o valor PC+4 no RA, e depois prepara o endereço para o salto e o salva no PC, da mesma forma que na instrução jump, já explicada.

```
} else if(eventNow.instrucao[0] == 0x15) { //JAL
    machine.RA = machine.PC + 4;  //+4 para pegar next instruction
    eventNow.instrucao[1] = eventNow.instrucao[1] << 8;
    eventNow.instrucao[1] += eventNow.instrucao[2];
    machine.PC = eventNow.instrucao[1];
```

/16xxxx - Return to Subroutine

Categoria	Instruções de controle de fluxo
Mnemônico	RTS

Endereçamento	Nenhum
Operandos	Nenhum
Exemplo	RTS
Descrição	faz um salto para o endereço armazenado no RA. Esse endereço é escrito no Program Counter
Codificação	/16xxxx

Implementação

Simplesmente atribui o valor do RA (Return Address) ao Program Counter

```
} else if(eventNow.instrucao[0] == 0x16) { //RTS
    machine.PC = machine.RA;
```

/21xxxx - ADD

Categoria	Operações Aritméticas
Mnemônico	ADD
Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	ADD R1,R2,R3
Descrição	Soma os valores armazenados em R2 e R3, e armazena o resultado em R1.
Codificação	/21xxxx

Implementação

Os operandos 2 e 3 da instrução são extraídos, e com eles é calculada a soma através da função "blocoOperacoesAritmeticas", que executa operações aritméticas. O resultado é então gravado no primeiro registrador mencionado na instrução, através da função "setValorOperacao".

```
} else if(eventNow.instrucao[0] == 0x21) { //ADD
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesAritmeticas(operando2,operando3,1);

    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/22xxxx - ADDI - Add Immediate

Categoria	Operações Aritméticas
Mnemônico	ADDI
Endereçamento	Registrador direto. O valor do operando é imediato, mas o endereço em que os dados

	serão salvos ainda é um registrador.
Operandos	R1,R2,<valor>
Exemplo	ADDI R1,R2,<valor>
Descrição	Soma o valor em R2 com o valor imediato inserido como operando e armazena o resultado em R1.
Codificação	/22xxxx

Implementação

Primeiramente, extrai-se o valor de uma das parcelas da soma, que está armazenada em um registrador. A parcela que é valor imediato já está disponível na própria instrução, bastando extraí-la diretamente. Em seguida, é feita a soma, e enfim o resultado é gravado no devido registrador (o primeiro a ser mencionado na instrução), através do uso da função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x22) { //ADDI
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = eventNow.instrucao[3];
    operando2 += operando3;
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/23xxxx - SUB

Categoria	Operações Aritméticas
Mnemônico	SUB
Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	SUB R1,R2,R3
Descrição	Subtrai o valor em R3 do valor armazenado em R2, e armazena o resultado em R1.
Codificação	/23xxxx

Implementação

Os operandos 2 e 3 da instrução são extraídos, e com eles é calculada a subtração através da função “blocoOperacoesAritmeticas”, que executa operações aritméticas. O resultado é então gravado no primeiro registrador mencionado na instrução, através da função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x23) { //SUB
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesAritmeticas(operando2,operando3,2);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/24xxxx - MUL

Categoria	Operações Aritméticas
Mnemônico	MUL

Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	SUB R1,R2,R3
Descrição	Multiplica o valor em R2 pelo valor armazenado em R3, e armazena o resultado em R1.
Codificação	/24xxxx

Implementação

Os operandos 2 e 3 da instrução são extraídos, e com eles é calculada a multiplicação através da função “blocoOperacoesAritmeticas”, que executa operações aritméticas. O resultado é então gravado no primeiro registrador mencionado na instrução, através da função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x24) { //MUL
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesAritmeticas(operando2,operando3,3);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/25xxxx - DIV

Categoria	Operações Aritméticas
Mnemônico	DIV
Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	DIV R1,R2,R3
Descrição	Divide o valor em R2 pelo valor armazenado em R3, e armazena o resultado em R1.
Codificação	/25xxxx

Implementação

Os operandos 2 e 3 da instrução são extraídos, e com eles é calculada a divisão através da função “blocoOperacoesAritmeticas”, que executa operações aritméticas. O resultado é então gravado no primeiro registrador mencionado na instrução, através da função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x25) { //DIV
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesAritmeticas(operando2,operando3,4);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/31xxxx - AND

Categoria	Operações Lógicas
Mnemônico	AND

Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	AND R1,R2,R3
Descrição	Faz a operação AND bit a bit entre os valores armazenados nos registradores R2 e R3, e armazena o resultado em R1
Codificação	/31xxxx

Implementação

primeiramente, extraem-se os operandos 2 e 3 dessa instruções - sobre os quais será executada a operação AND. Em seguida, chama-se a função “blocoOperacoesLogicas”, que executa operações lógicas. Enfim, o resultado é gravado no registrador mencionado no primeiro operando da instrução, com a função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x31) { //AND
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesLogicas(operando2,operando3,1);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/32xxxx - OR

Categoria	Operações Lógicas
Mnemônico	OR
Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	OR R1,R2,R3
Descrição	Faz a operação OR bit a bit entre os valores armazenados nos registradores R2 e R3, e armazena o resultado em R1
Codificação	/32xxxx

Implementação

primeiramente, extraem-se os operandos 2 e 3 dessa instruções - sobre os quais será executada a operação OR. Em seguida, chama-se a função “blocoOperacoesLogicas”, que executa operações lógicas. Enfim, o resultado é gravado no registrador mencionado no primeiro operando da instrução, com a função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x32) { //OR
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesLogicas(operando2,operando3,2);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/33xxxx - XOR

Categoria	Operações Lógicas

Mnemônico	XOR
Endereçamento	Registrador Direto
Operandos	R1,R2,R3
Exemplo	XOR R1,R2,R3
Descrição	Faz a operação XOR bit a bit entre os valores armazenados nos registradores R2 e R3, e armazena o resultado em R1
Codificação	/33xxxx

Implementação

primeiramente, extraem-se os operandos 2 e 3 dessa instruções - sobre os quais será executada a operação XOR. Em seguida, chama-se a função “blocoOperacoesLogicas”, que executa operações lógicas. Enfim, o resultado é gravado no registrador mencionado no primeiro operando da instrução, com a função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x33) { //XOR
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando3 = returnRegistradorValor(eventNow.instrucao[3],machine);
    operando2 = blocoOperacoesLogicas(operando2,operando3,3);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/34xxxx - NOT

Categoria	Operações Lógicas
Mnemônico	NOT
Endereçamento	Registrador Direto
Operandos	R1,R2
Exemplo	NOT R1,R2
Descrição	Faz a operação NOT bit a bit entre os valores armazenados no registrador R2, e armazena o resultado em R1.
Codificação	/34xxxx

Implementação

Primeiramente, o operando 2, sobre o qual se executará a operação NOT, é extraído da instrução. Em seguida, utiliza-se a função “blocoOperacoesLogicas”, que executa operações lógicas. Assim, é feita a opetação “NOT”. Em seguida, o resultado é gravado no registrador definido como primeiro operando da instrução, com a função “setValorOperacao”.

```
} else if(eventNow.instrucao[0] == 0x34) { //NOT
    operando2 = returnRegistradorValor(eventNow.instrucao[2],machine);
    operando2 = blocoOperacoesLogicas(operando2,0,4);
    machine = setValorOperacao(eventNow.instrucao[1],operando2,machine);
```

/41xxxx - HM - Halt Machine

Categoria	Operações Lógicas

Mnemônico	HM
Endereçamento	nenhum
Operandos	nenhum
Exemplo	HM
Descrição	Interrompe o funcionamento do motor de eventos
Codificação	/41xxxx
Implementação	
O atributo “enable” do struct estadoMaquina é posto em zero, e assim o motor para de funcionar. `} else if(eventNow.instrucao[0] == 0x41) { //HM - Halt Machine` `machine.enable = 0;`	

/51xxxx - MOV - Move	
Categoria	Pseudoinstruções
Mnemônico	MOV
Endereçamento	Registrador Direto
Operandos	R1,R2
Exemplo	MOV R1,R2
Descrição	Move o conteúdo do registrador R2 para o registrador R1
Codificação	/51xxxx
Implementação	
MOV é uma pseudoinstrução, portanto não tem um procedimento de reação definido. Ao invés disso, MOV é substituído pela execução dos comandos SUB e ADD.	

/52xxxx - @	
Categoria	Pseudoinstruções
Mnemônico	@
Endereçamento	Endereçamento direto
Operandos	<endereço>
Exemplo	@ <endereço>

Descrição	faz o programa ser gravado na memória a partir do endereço de operando dessa pseudoinstrução.
Codificação	/52xxxx
Implementação	
Não há um tratamento para este comando, pois se trata de uma pseudoinstrução. @ nem ao menos é gravado na memória. A função do @ é informar a partir de qual endereço se deve fazer a carga do programa.	

/53xxxx - #	
Categoria	Pseudoinstruções
Mnemônico	#
Endereçamento	nenhum
Operandos	nenhum
Exemplo	#
Descrição	Na leitura dessa pseudoinstrução, no lugar, é lida a instrução HM, e a execução do programa é finalizada.
Codificação	/53xxxx
Implementação	
Não há, uma vez que se trata de uma pseudoinstrução. Essa pseudoinstrução é substituída pela execução da instrução “HM”	

/61xxxx - MD	
Categoria	Eventos Artificiais
Mnemônico	MD
Endereçamento	Endereçamento Direto
Operandos	<endereço>
Exemplo	MD <endereço>
Descrição	Exibe na tela o conteúdo de cada posição de memória a partir do endereço usado como operando dessa instrução. Exibe o conteúdo de 160 endereços
Codificação	/61xxxx
Implementação	
Primeiramente, é feito o printf dos valores dos registradores, formatados para os valores serem impressos em hexadecimal; a seguir, são impressas as posições de memória, a partir do endereço fornecido na digitação da instrução (daí, é feita a atribuição startPrintAddress = eventNow.instrucao[1];). Dois laços “for”, um dentro do outro, começam a imprimir o conteúdo da memória, na forma hexadecimal, a partir do valor “startPrintAddress”, ou seja, o valor inicial fornecido na digitação da instrução. Ressalta-se que o primeiro “for” determina o número de linhas a serem impressas (10), e o segundo “for”, dentro do primeiro “for”, serve para determinar quantos endereços de memória são impressos por linha (16 no caso).	

Assim, um comando “Memory Display” permite a impressão de 10 * 16 = 160 endereços de memória.

```
} else if(eventNow.instrucao[0] == 0x61) { //MD
    int startPrintAddress;
    int counter1,counter2;
    printf("\n-------------------------------------------------------------------------------------------\n");
    printf("PC= %04x     SP= %04x     FP= %04x     RA= %04x\n",machine.PC,machine.SP,machine.FP,machine.RA);
    printf("R1= %04x     R2= %04x     R3= %04x     R4= %04x\n",machine.R1,machine.R2,machine.R3,machine.R4);
    printf("R5= %04x     R6= %04x     R7= %04x     R8= %04x\n",machine.R5,machine.R6,machine.R7,machine.R8);
    startPrintAddress = eventNow.instrucao[1];

    for(counter1 = 0; counter1 < 10; counter1++) {
        for(counter2 = 0; counter2 < 16; counter2++) {
            printf("%02x  ",machine.memory[startPrintAddress]);
            startPrintAddress++;
        }

        printf("\n");
    }
```

/62xxxx - MM - Memory Modify	
Categoria	Eventos Artificiais
Mnemônico	MM
Endereçamento	Endereçamento Direto
Operandos	<endereço> <conteúdo>
Exemplo	MM <endereço> <conteúdo>
Descrição	Grava no endereço aplicado como primeiro operando da instrução o segundo operando dessa instrução.
Codificação	/62xxxx

Implementação

Sem grandes dificuldades, no Memory Modify simplesmente se deve modificar um atribuir um valor a um dado endereço. Ambos valor e endereço são fornecidos na digitação desse evento. Assim, com uma linha, é feita a atribuição de valor ao dado endereço da memória:

```
} else if(eventNow.instrucao[0] == 0x62) { //MM - Memory Modify
    machine.memory[eventNow.instrucao[1]] = eventNow.instrucao[2];
```

/63xxxx - HE - Help	
Categoria	Eventos Artificiais
Mnemônico	HE

Endereçamento	nenhum
Operandos	nenhum
Exemplo	HE
Descrição	Exibe na tela os eventos artificiais e pseudoinstruções disponíveis para uso.
Codificação	/63xxxx

Implementação

Trata-se somente da execução de comandos “printf”:

```
} else if(eventNow.instrucao[0] == 0x63) {  //HE
    printf("Catálogo de instruções disponíveis\n");
    printf("Eventos artificiais----------------------------------------\n");
    printf("HE: Help\n->Sintaxe: HE\n\n");
    printf("MD: Memory display. \n->Sintaxe: MD <endereco>\n\n");
    printf("MM: Memory Modify. \n->Sintaxe: MM <endereco> <conteúdo>\n\n");
    printf("TRC: Trace. Esse comando tanto ativa como desativa o modo trace\n->Sintaxe: TRC\n\n");
    printf("Pseudoinstruções-------------------------------------------\n");
    printf("@: Início do programa. \n->Sintaxe: @ <endereço de início do programa>\n\n");
    printf("#: Final Físico do Programa. \n->Sintaxe: #\n\n");
    printf("MOV: MOVE. \n->Sintaxe: MOV <Registrador 1> <Registrador 2>, em que registrador 1 é o de
```

/64xxxx - TRC - Trace	
Categoria	Eventos Artificiais
Mnemônico	TRC
Endereçamento	nenhum
Operandos	nenhum
Exemplo	TRC
Descrição	Ativa ou desativa o modo trace: a cada instrução do programa que for executada, o comando MD <endereço do program counter> é executado.
Codificação	/64xxxx

Implementação

O modo trace foi implementado como um estado de máquina. Assim, o “struct estadoMaquina” tem uma variável int isTrace, que vale 1 caso o modo trace esteja ativado e zero caso contrário. Assim, ao se executar o evento artificial “Trace”, simplesmente, é feita uma troca do valor da variável “isTrace”:

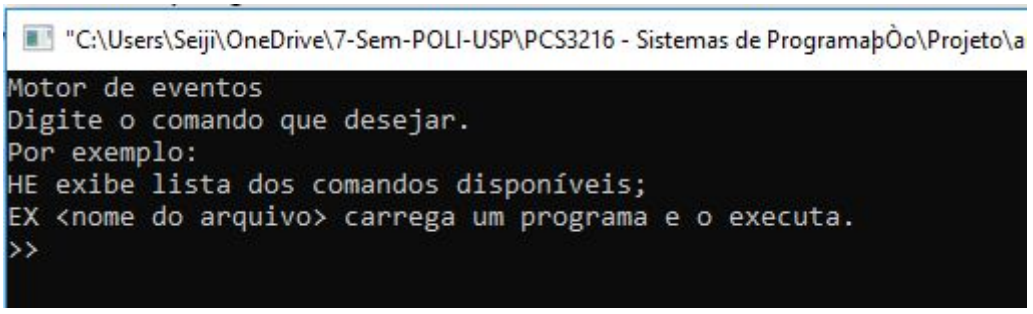
```
} else if(eventNow.instrucao[0] == 0x64) { //TRC
    if(machine.isTrace == 0) {              //TRC deve,
        machine.isTrace = 1;
        printf("Modo Trace Ativado!\n");
    } else if(machine.isTrace == 1) {
        machine.isTrace = 0;
        printf("Modo Trace Desativado!\n");
    }
```

Resumo de todas as instruções do conjunto de instruções da linguagem simbólica implementada:

Mnemônico	Endereçamento	Exemplo	Codificação
Acesso à memória			
LW	Registrador direto	LW R1,R2	/01xxxxxx
SW	Registrador direto	SW R1,R2	/02xxxxxx
Loops, controle de fluxo, desvios e chamadas de procedimento			
BEQ	Registrador direto	BEQ R1,R2,R3	/11xxxxxx
BZR	Registrador direto	BZR R1,R2	/12xxxxxx
BNE	Registrador direto	BNE R1,R2,R3	/13xxxxxx
J	Direto	J <label> ou J <endereço>	/14xxxxxx
JAL	Direto	JAL <label> ou JAL <endereço>	/15xxxxxx
RTS	nenhum	RTS	/16xxxxxx
Operações Aritméticas			
ADD	Registrador direto	ADD R1,R2,R3	/21xxxxxx
ADDI	Registrador direto	ADDI R1,R2,<número em hexadecimal>	/22xxxxxx
SUB	Registrador direto	SUB R1,R2,R3	/23xxxxxx
MUL	Registrador direto	MUL R1,R2,R3	/24xxxxxx
DIV	Registrador direto	DIV R1,R2,R3	/25xxxxxx
Operações Lógicas			
AND	Registrador direto	AND R1,R2,R3	/31xxxxxx
OR	Registrador direto	OR R1,R2,R3	/32xxxxxx
XOR	Registrador direto	XOR R1,R2,R3	/33xxxxxx
NOT	Registrador direto	NOT R1,R2	/34xxxxxx
Outros			
HM	Nenhum	HM	/41xxxxxx

Pseudoinstruções			
MOV	Registrador direto	MOV R1,R2	/51xxxxxx
@	Direto	@ <endereço>	/52xxxxxx
#	Nenhum	#	/53xxxxxx
Eventos artificiais			
MD	Direto	MD <endereço>	/61xxxxxx
MM	Direto	MM <endereço> <valor>	/62xxxxxx
HE	Nenhum	HE	/63xxxxxx
TRC	Nenhum	TRC	/64xxxxxx

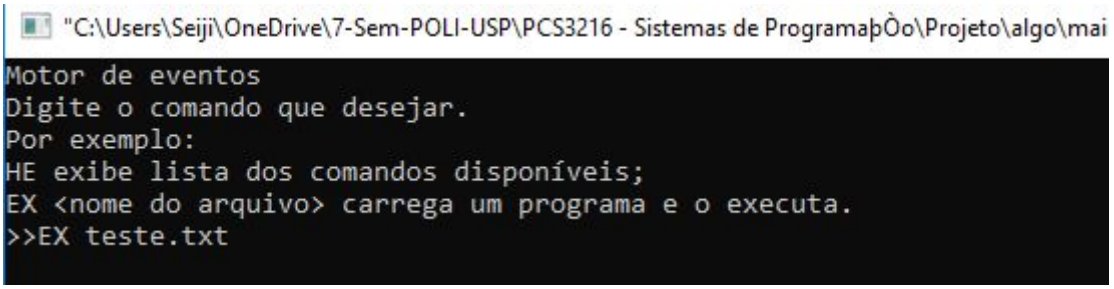
5. Modo de uso do programa



Essa é a tela inicial do programa. Há vários eventos artificiais os quais podem ser inseridos na lista de eventos, extraídos da mesma e executados. A especificação de eventos artificiais consta na seção de instruções.

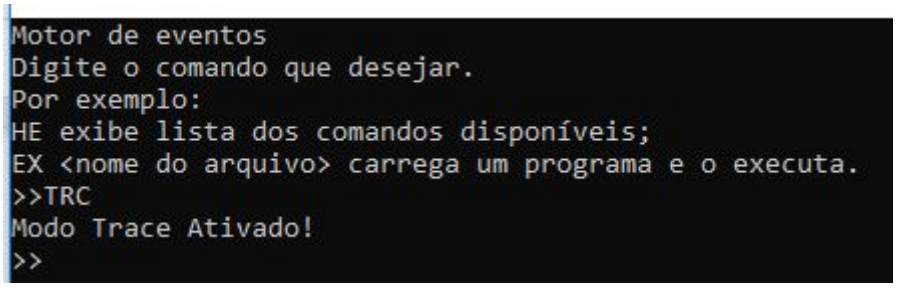
Destaca-se também o uso da biblioteca <locale.h>, que permitiu o uso de acentos, não previstos com o uso simplesmente o uso de ASCII que a linguagem C define.

Em particular, o evento artificial EX permite que seja carregado na memória um programa e em seguida que esse seja executado. Para tal, basta, por exemplo, digitar o nome do arquivo a frente do “EX”, como mostra a figura abaixo.



Basta pressionar a tecla “Enter” e o programa é executado.

Recomenda-se a ativação do modo “Trace”, a fim de que ao término da execução de cada instrução do programa, seja possível a verificação do conteúdo da memória e dos registradores. Com o modo “Trace” ativado, o evento artificial MD (memory Display) é executada, com o endereço do Program Counter, ao fim da execução de cada instrução pelo motor de eventos, inserindo-se assim sucessivamente o evento artificial MD na lista de eventos.



6. Testes

Os testes que foram executados com o intuito de verificar o funcionamento do programa seguem abaixo. Esses testes visam, em geral, verificar se o funcionamento do conjunto de instruções implementado está correto. Foram feitos três testes para verificar esse funcionamento do conjunto de instruções. Em geral, intentou-se elaborar testes que contemplassem a verificação dos seguintes tópicos:

- Instruções de load e store;
- instruções de operações aritméticas;
- instruções de operações lógicas;
- uso de labels, e das instruções que as manipulam;
- Pseudoinstruções (a de MOV deve ser de fato testada, enquanto sobre @ e #, não é necessária muita preocupação, pois pertencem a qualquer programa a ser executado, e sem essas pseudoinstruções, o código simplesmente não funciona. Funcionando os programas elaborados, entende-se que @ e # também funcionam corretamente).

1) Algumas operações aritméticas e uma gravação na memória

O arquivo desse teste foi nomeado como “teste1.txt”.

```
@    4000
ADDI R1,R2,5
MOV  R3,R1
MUL  R4,R1,R1
ADDI R5,R5,2000
SW   R5,R4
#
```

- Linha 1: grava o programa na memória a partir do endereço 4000 (4000 em base decimal);
- Linha 2: Soma o valor 5 no registrador R1 (o registrador 2 não importa, pois carrega o valor zero, supondo que o programa foi executado de um estado em que todos os registradores estavam zerados);
- Linha 3: Move o conteúdo do registrador R1 para o registrador R3;
- Linha 4: Armazena no registrador R4 o valor em R1 ao quadrado;
- Linha 5: salva em R5 o valor 2000;
- Linha 6: salva o conteúdo em R4 no endereço em R5, ou seja, 2000.

Ao se digitar no console, após a execução do programa em questão, o evento artificial “MD 2000”, observa-se como resultado a tela abaixo:

```
>>MD 2000

-------------------------------------------------------------------------------------------------
PC= 0fbc     SP= 0000     FP= 0000     RA= 0000
R1= 0005     R2= 0000     R3= 0005     R4= 0019
R5= 07d0     R6= 0000     R7= 0000     R8= 0000
19  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
```

O resultado obtido mostra que, no endereço 2000 da memória foi gravado o valor /19, como mostra a figura acima. /19 vale 25 na base decimal, confirmando o resultado desejado, visto que o esperado era justamente o valor em R1 ao quadrado, ou seja, o quadrado de 5.

Dessa forma, esse teste verificou o funcionamento:

- Da instrução de Store do conjunto de instruções implementado;
- Da instrução Add Immediate, que permite a soma de um valor imediato;
- Da pseudoinstrução Move, que move o conteúdo de um registrador a outro

2) Operações Lógicas, saltos condicionais, labels

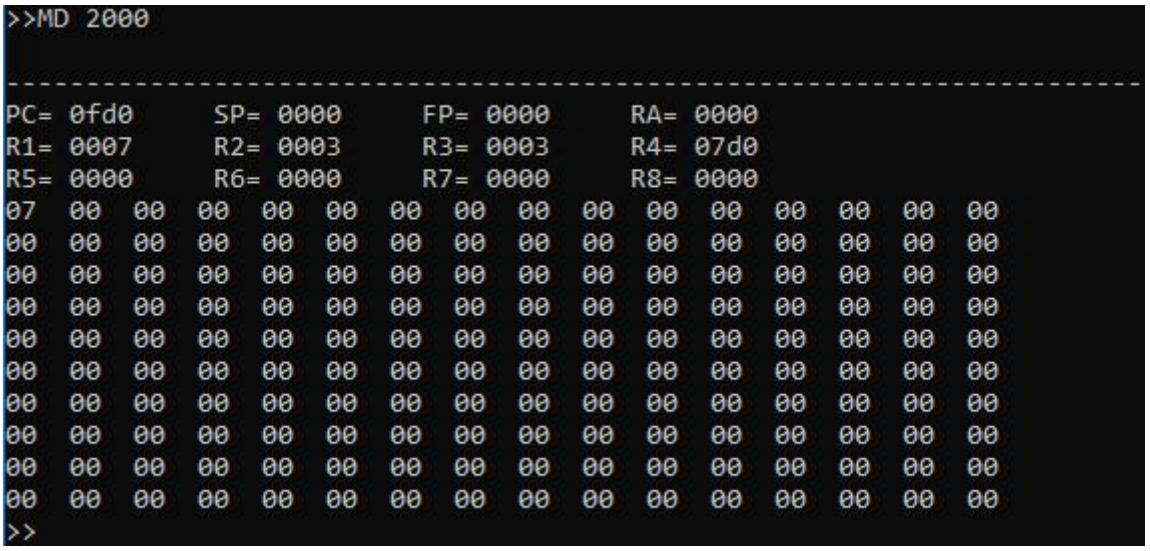
O arquivo desse teste foi nomeado como “teste2.txt”.

```
        @     4000
        ADDI  R1,R1,5
        ADDI  R2,R2,3
        ADDI  R3,R3,3
        BEQ   loop1,R2,R2
        BNE   loop2,R2,R3

loop1   OR    R1,R1,R2
        J     fim
loop2   AND   R1,R1,R2
        J     fim
fim     ADDI  R4,R4,2000
        SW    R4,R1
        #
```

- Linha 1: Endereço inicial de gravação do programa em questão na memória;
- Linha 2: conteúdo do registrador 1 como 5;
- Linha 3: conteúdo do registrador 2 como 3;
- Linha 4: conteúdo do registrador 3 como 3;
- Linha 5: caso os registradores R2 e R3 possuam mesmo valor, faz salto para a label “loop1”;
- Linha 6: caso os registradores R2 e R3 possuam valores diferentes, faz salto para a label “loop2”;
- Linha 7: vazia;
- Linha 8: label “loop1”: faz a operação “OR” e armazena resultado no registrador R1;
- Linha 9: faz salto para a linha da label “fim”;
- Linha 10: label “loop2”: faz uma operação “AND” no conteúdo do registrador R1 com o conteúdo do registrador R2;
- Linha 11: salto para a linha de label “fim”;
- Linha 12: label “fim”: armazena em R4 o valor 2000;
- Linha 13: Armazena conteúdo do registrador R1 no endereço em R4 (no caso, 2000).

Nesse programa, é interessante o teste para valores em R2 e R3 tanto iguais como diferentes, a fim de que sejam testados tanto os casos em que o programa varre as linhas de label “loop1” como também as linhas de label “loop2”.

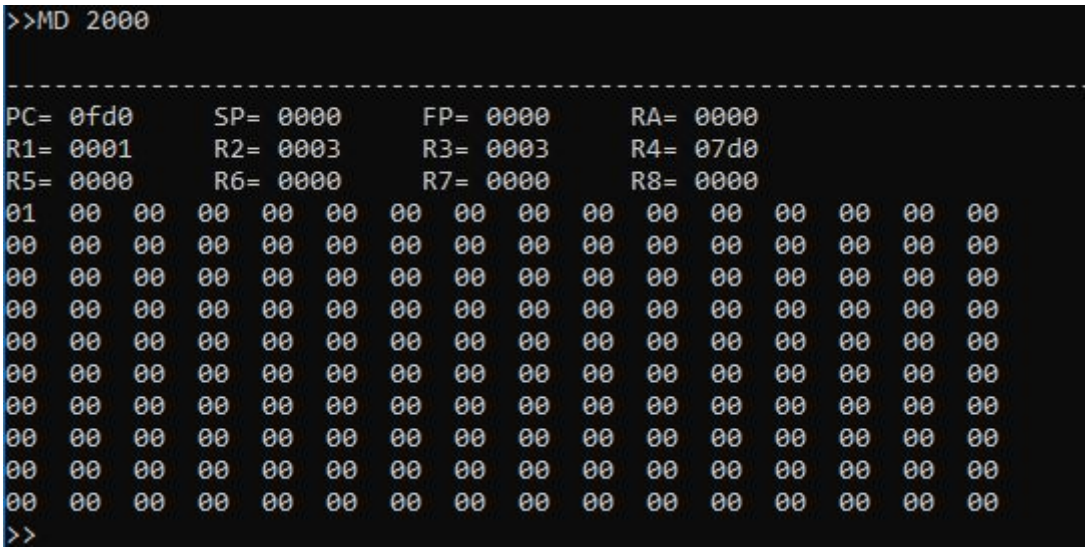
No caso em que ocorrem R2 e R3 iguais, como mostra o código acima reproduzido, o resultado da operação “OR” deveria ser 7. Assim, tem-se o seguinte resultado:



De fato, nota-se que o registrador R1, bem como a posição de memória 2000, possuem o valor 7 após a execução do programa, confirmando a obtenção do resultado esperado.

Já quando R2 e R3 são diferentes, o trecho de código de label “loop2” é executado, enquanto o trecho de código de label “loop1”, não.

Faz-se assim uma operação AND de 5 com 3 (5 and 3), cujo resultado esperado é 1 (0101 and 0011 = 0001, e os bits mais significativos que esses quatro são todos 0, portanto o resultado da operação and é 0 para tais bits).



Como se nota, o valor “1” de fato foi salvo na posição 2000 da memória, como planejado.

Dessa forma, o presente teste verificou adequadamente o funcionamento:

- Do uso de labels e instruções que as manipulam;
- Do uso de operações lógicas.

3) Instrução Load

O arquivo desse teste foi nomeado como “teste3.txt”.

```
        @     4000
        ADDI  R1,R1,1
        ADDI  R2,R2,2
        ADDI  R3,R3,3
        ADDI  R4,R4,3000
        BEQ   loopA,R2,R3
        BNE   loopB,R2,R3

loopA   SW    R4,R1
        LW    R8,R4
        J     fim
loopB   SW    R4,R2
        LW    R8,R4
        J     fim
fim     #
```

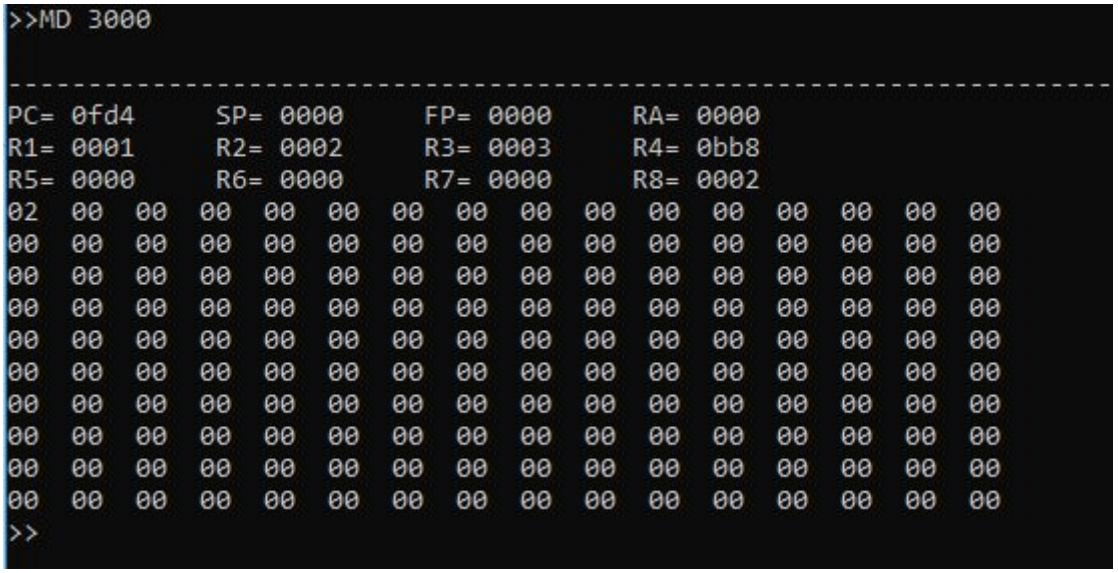
- Linha 1: Endereço inicial a partir do qual se dá a gravação do programa na memória;
- Linha 2: Faz-se o armazenamento do valor 1 no registrador R1;
- Linha 3: Faz-se o armazenamento do valor 2 no registrador R2;
- Linha 4: Faz-se o armazenamento do valor 3 no registrador R3;
- Linha 5: Faz-se o armazenamento do valor 3000 no registrador R4. Esse valor será utilizado como endereço adiante no código;
- Linha 6: Uso da instrução “Branch if Equal”. Se os registradores R2 e R3 armazenarem o mesmo valor, faz-se um salto para o endereço com label “LoopA”.
- Linha 7: Uso da instrução “Branch if not Equal”. Se os registradores R2 e R3 armazenarem valores diferentes (o que de fato ocorre simplesmente se essa linha for executada, visto que se forem iguais os conteúdos desses registradores, o salto para “LoopA” teria ocorrido e a instrução BNE não teria sido executada).
- Linha 8: Label “LoopA”. é armazenado no endereço 2000 (registrador R4) o valor no registrador R1;
- Linha 9: é armazenado no registrador R8, utilizando-se do “load”, o valor que havia no endereço 2000 (Registrador R4);
- Linha 10: Salto para o endereço da label “fim”;
- Linha 11: é armazenado no endereço 3000 (Registrador R4) o valor que havia no registrador R2;
- Linha 12: é armazenado no registrador R8 o valor que havia no endereço 3000 (registrador R4);
- Linha 13: Salto para o endereço da label “fim”;
- Linha 14: Label “fim”, que possui a pseudo instrução “#”, que encerra a execução do programa.

Tal como no teste 2, nesse programa se acaba explorando o uso das instruções de branch, tanto na versão “if equal” quanto na versão “if not equal”. Os registradores R2 e R3

podem ter valor alterado justamente para que seja possível testar as duas possibilidades de execução do código.

Como primeiro caso, supondo que R2 e R3 armazenem, respectivamente, 2 e 3, conforme o código reproduzido acima, a label “LoopB” é que será executada.

Espera-se assim que haja o valor “2” no endereço 3000 da memória, e que no registrador R8, seja também armazenado o valor 2, obtido justamente do endereço 3000 da memória.



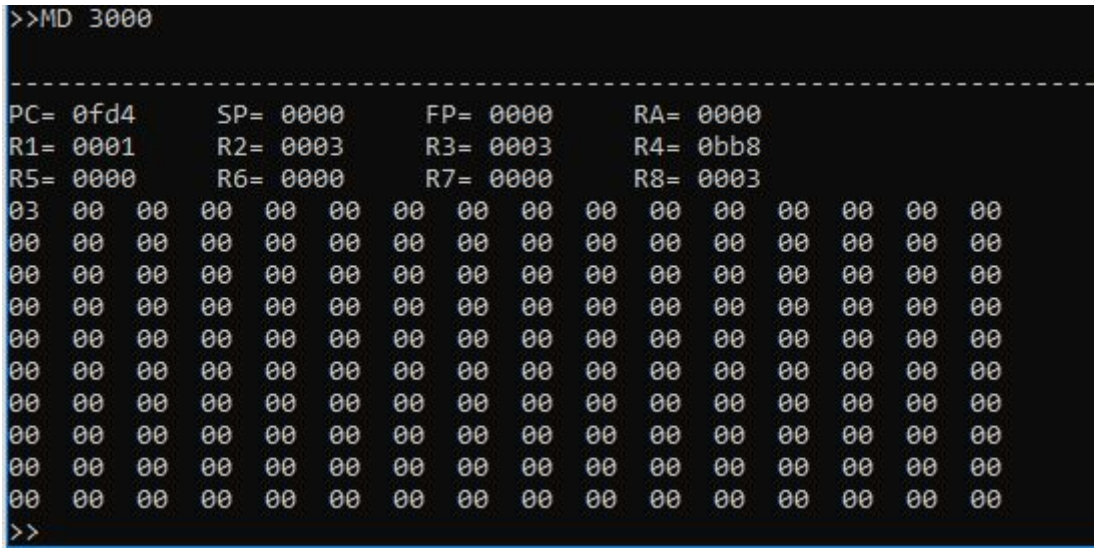
Como mostra a figura acima, ao se executar o comando “MD3000”, nota-se que o endereço 3000 da memória armazena o valor 2, e que o registrador R8 também armazena o valor 2.

Outra alternativa de execução do código desse teste 3 seria fazer com que ocorresse um salto para a label “LoopA”, caso decorrente da igualdade dos valores nos registradores R2 e R3. Para obter tal resultado, basta alterar a linha 3 do código para:

```
ADDI R2,R2,3
```

Assim, os valores armazenados em R2 e R3 no momento da execução da instrução “BEQ” passam a ser iguais.

‘ Com a execução desse código, espera-se então que o registrador R8 passe a armazenar o valor 3, o que de fato ocorre.



Na figura acima, observa-se que o registrador R8 passa a armazenar o valor 3, o que de fato ocorreu. Assim, o teste em questão permitiu a verificação do funcionamento da instrução que executa o “load” da memória para o registrador.

Conclui-se que o resultado esperado fora obtido em todos os testes realizados. Assim, esperou-se validar o fato de que as instruções implementadas, apesar de terem sido classificadas em diferentes categorias e possuírem diferentes implementações, quantidades de operandos e tipos de endereçamento de seus operandos, funcionam de acordo com seus comportamentos previstos.