ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia de Computação e Sistemas Digitais

PCS 3111 - LABORATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A ENGENHARIA ELÉTRICA

EXERCÍCIO PROGRAMA 2 – 2º SEMESTRE DE 2022

1 Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar um emulador de uma arquitetura de computador simples, mas moderna o suficiente para ser usada na prática.

2 Introdução

Neste EP serão implementadas algumas melhorias no emulador simplificado de um microprocessador MIPS criado anteriormente.

Na arquitetura monociclo apresentada no EP1 há memórias separadas para dados e instruções. Neste EP usaremos uma memória única para isso, assim como nos microprocessadores reais. Isso é possível porque as instruções são também dados armazenados na memória.

Uma outra limitação do microprocessador do EP1 é que ele não possui uma forma exclusiva de usar dispositivos – como um monitor, um teclado, uma placa de rede, etc. Uma solução comum em arquitetura de computadores é dedicar um espaço de endereçamento da memória para os dispositivos, o que é chamado de *entrada e saída mapeada em memória* (para mais detalhes consulte o livro do Harris e Harris[1]). A ideia é que cada dispositivo tenha ao menos um endereço definido na memória. Quando se pede para escrever algo nesse endereço, estamos na verdade escrevendo um valor em um registrador do dispositivo; quando se pede para ler algo nesse endereço, se obtém um valor do dispositivo. Por exemplo, considere que o microprocessador possui uma memória de tamanho 1024. Poderíamos reservar o espaço de 1001 a 1024 para os dispositivos. Um teclado poderia ficar no endereço 1001 e um monitor no endereço 1016. Ao executar uma instrução que lê o endereço 1001, seria lido o dado do teclado; ao executar uma instrução que escreve no endereço 1016, seria escrito o dado na tela.

Também será possível carregar o conteúdo da memória de um arquivo, facilitando a execução de programas (um conjunto de instruções). De forma similar, também será possível armazenar o conteúdo da memória em um arquivo.

[1] HARRIS, D. M.; HARRIS, S. L. **Digital Design and Computer Architecture**. 2a ed., Morgan Kaufmann, 2013,

Por fim, o projeto da solução evitará a criação de instruções inválidas e haverá verificação de erros para evitar problemas de execução.

2.1 Simplificações

Foram feitas algumas simplificações para diminuir o tamanho do projeto e exercitar alguns conceitos específicos da disciplina. Não foram criados subtipos para as instruções, fazendo com que, dependendo da instrução, alguns atributos não sejam úteis. No caso da entrada e saída mapeada em memória, o espaço dos dispositivos será variável e contado a partir do fim da memória normal.

Uma outra simplificação é considerar dois tipos de teclados e monitores: um que lê/escreve `int` e outro que lê/escreve `char`. Isso foi feito para simplificar os programas (que não precisarão de rotinas de conversão de dados).

2.2 Objetivo

O objetivo deste projeto é fazer um emulador simplificado de um microprocessador MIPS, evoluindo o programa desenvolvido no EP1. A solução deve empregar adequadamente conceitos de orientação a objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor, herança, classe abstrata, membros com escopo de classe, programação defensiva, persistência em arquivo e os containers da STL. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

Para desenvolver o EP deve-se manter a mesma dupla do EP1. Será possível apenas **desfazer a dupla**, mas não formar uma nova. Veja na Seção 5 como fazê-lo.

3 Projeto

Deve-se implementar em C++ as classes **BancoDeRegistradores**, **Dado**, **Dispositivo**, **ESMapeadaNaMemoria**, **GerenciadorDeMemoria**, **Instrucao**, **Memoria**, **MemoriaRAM**, **Monitor**, **MonitorDeChar**, **Teclado**, **TecladoDeChar** e **UnidadeDeControle**, além de criar uma `main` que permita o funcionamento do programa como desejado.

Atenção:

1. O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. As classes não devem possuir outros membros (atributos ou métodos) **públicos** além dos especificados. **Note que você poderá definir atributos e métodos privados** e **protegidos, caso necessário**.
2. Não é permitida a criação de outras classes além dessas.
3. As classes filhas podem (*e às vezes devem*) redefinir métodos da classe mãe.
4. Não faça `#define` para constantes. Você pode (e deve) fazer `#ifndef/#define` para permitir a inclusão adequada de arquivos.

O não atendimento a esses pontos pode resultar em **erro de compilação** na correção automática e, portanto, nota `0` na correção automática.

Cada uma das classes deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Os arquivos devem ter exatamente o nome da classe. Por exemplo, deve-se ter os arquivos "Dado.cpp" e "Dado.h". Note que você deve criar os arquivos necessários.

Em relação às exceções (assunto da Aula 9), todas as especificadas são da biblioteca padrão (não se esqueça de fazer `#include <stdexcept>`). O texto usado como motivo da exceção não é especificado no enunciado e não será avaliado. Jogue as exceções criando um objeto usando new. Por exemplo, para jogar um `logic_error` faça algo como:

```
throw new logic_error("Mensagem de erro");
```

Caso a exceção seja jogada de outra forma, pode haver erros na correção e, consequentemente, desconto na nota.

3.1 Classe Dado

Um **Dado** é um valor inteiro armazenado na memória de dados. Essa classe deve possuir apenas os seguintes **métodos públicos**:

```
Dado(int valor);
virtual ~Dado();
virtual int getValor();
virtual void imprimir();
```

A mudança do EP1 é só o uso do `virtual` em alguns métodos.

O construtor recebe um valor, o qual deve ser retornado pelo método `getValor`. O método `imprimir` deve apenas imprimir o valor, **sem pular uma linha** ao final.

3.2 Classe Instrucao

A **Instrucao** agora é filha de **Dado**. Os campos definidos são os mesmos do EP1:

- **opcode**: representa o código que identifica a instrução (obrigatório);
- **origem1**: número do registrador de origem cujo valor será usado como primeiro operando;
- **origem2**: número do registrador de origem cujo valor será usado como segundo operando;
- **destino**: número do registrador de destino, onde o resultado da operação será armazenado;
- **imediato**: valor absoluto que será usado na operação (seu significado depende da operação); e
- **função**: o código da função, para o caso de operações envolvendo registradores.

O opcode deve ser considerado como o valor (int) do **Dado**. Os demais campos devem ser outros atributos inteiros. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
static const int LW = 35;
static const int SW = 43;
static const int J = 2;
static const int BNE = 5;
static const int BEQ = 4;
static const int TIPO_R = 0;
static const int FUNCAO_ADD = 32;
static const int FUNCAO_SUB = 34;
static const int FUNCAO_MULT = 24;
static const int FUNCAO_DIV = 26;

static Instrucao* criarLW(int destino, int imediato);
static Instrucao* criarSW(int destino, int imediato);
static Instrucao* criarJ(int imediato);
static Instrucao* criarBNE(int origem1, int origem2, int imediato);
static Instrucao* criarBEQ(int origem1, int origem2, int imediato);
static Instrucao* criarADD(int destino, int origem1, int origem2);
static Instrucao* criarSUB(int destino, int origem1, int origem2);
static Instrucao* criarMULT(int origem1, int origem2);
static Instrucao* criarDIV(int origem1, int origem2);

virtual ~Instrucao();
virtual int getOpcode();
virtual int getOrigem1();
virtual int getOrigem2();
virtual int getDestino();
virtual int getImediato();
virtual int getFuncao();
```

Diferentemente do EP1, os valores padrão do *opcode* e das funções são definidos como constantes (em vez de `defines`).

Para evitar que sejam criadas instruções inválidas, você deve fazer o construtor de **Instrucao** ser privado[2] (por isso ele não é especificado). Para criar uma **Instrucao** devem ser usados os métodos estáticos `criarLW`, `criarSW`, `criarJ`, `criarBNE`, `criarBEQ`, `criarADD`, `criarSUB`, `criarMULT`, `criarDIV`. Eles devem criar e retornar a **Instrucao** do tipo indicado pelo nome, considerando os parâmetros informados (você pode colocar o valor 0 ou outro valor nos demais campos).

Os métodos `getOpcode`, `getOrigem1`, `getOrigem2`, `getDestino`, `getImediato` e `getFuncao` devem retornar o valor dos campos definidos para o objeto. Considere que o *opcode* é o valor da **Instrucao**. Ou seja, os métodos `getOpcode` e `getValor` (herdado) devem retornar o mesmo valor.

Redefina o método `imprimir` para que ele imprima na saída padrão (`cout`) a seguinte informação, **sem pular uma linha no final**:

```
Instrucao <opcode>
```

Onde `<opcode>` é o valor do *opcode* dessa instrução.

As instruções definidas são as mesmas do EP1. Consulte o enunciado do EP1 para entender os campos e a semântica das instruções.

[2] A ideia é fazer uma *fábrica* simplificada de instruções.

3.3 Classe BancoDeRegistradores

O **BancoDeRegistradores** é o conjunto de registradores disponíveis ao microprocessador. Assim como no EP1, 3 registradores têm função específica:

- O registrador 0 possui sempre o valor 0.
- O registrador 24 armazena o resultado da multiplicação ou divisão.
- O registrador 25 armazena o resto da divisão.

Com isso, a classe **BancoDeRegistradores** deve possuir apenas os seguintes métodos **públicos** e o `define`:

```
#define QUANTIDADE_REGISTRADORES 32

BancoDeRegistradores();
virtual ~BancoDeRegistradores();
virtual int getValor(int registrador);
virtual void setValor(int registrador, int valor);
virtual void imprimir();
```

A mudança no EP1, além do virtual, é que os métodos `getValor` e `setValor` agora jogam exceções.

Assim como no EP1, o construtor não recebe parâmetros, mas deve reservar 32 inteiros, que serão usados como registradores. No início todos os registradores devem possuir o valor 0 (ou seja, devem ser zerados no construtor). O destrutor deve destruir todos os valores alocados dinamicamente.

O método `getValor` deve retornar o valor do registrador passado como parâmetro. No caso do registrador 0, o valor deve ser sempre 0. Caso o número do registrador seja inválido (menor que 0 ou maior ou igual a 32), o método deve jogar um `logic_error`.

O método `setValor` deve armazenar o valor passado como parâmetro no registrador indicado. No caso do registrador 0, esse método não deve fazer nada. No caso de um número de registrador inválido (menor que zero ou maior ou igual a 32), esse método deve jogar um `logic_error`.

O método `imprimir` tem o mesmo funcionamento do EP1. Ou seja, ele deve imprimir na saída padrão (`cout`) o valor dos registradores, no seguinte formato:

```
<número do registrador>: <valor>
```

Onde `<número do registrador>` é o número do registrador (entre 0 e 31) e `<valor>` é o valor do registrador. Consulte o enunciado do EP1 para um exemplo.

3.4 Classe Memoria

A classe **Memoria** é uma classe **abstrata** representando uma memória do microprocessador. Escolha o(s) método(s) mais adequado(s) para ser(em) **abstrato(s)** - para isso veja as classes **MemoriaRAM** e **ESMapeadaNaMemoria**. Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
Memoria();
virtual ~Memoria();
virtual int getTamanho();
virtual Dado* ler(int posicao);
virtual void escrever(int posicao, Dado* d);
virtual void imprimir();
```

O construtor não recebe nenhum parâmetro.

Veja o comportamento específico dos métodos `getTamanho`, `ler`, `escrever` e `imprimir` na descrição das classes filhas. Note que o método `escrever` é agora `void`.

3.5 Classe MemoriaRAM

A classe **MemoriaRAM** é uma classe filha concreta de **Memoria**. Ela representa uma memória que armazena **Dados**. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
MemoriaRAM(int tamanho);
virtual ~MemoriaRAM();

virtual void escrever(list<Dado*>* dados);
```

O construtor deve receber o tamanho da memória e alocar um vetor de objetos **Dado** com o tamanho informado. Ao inicializar, deve ser colocado `NULL` em cada posição, indicando que ela não contém um dado. O destrutor deve destruir todos os objetos **Dado** que ainda estão no vetor, além de destruir o vetor alocado dinamicamente.

O tamanho da memória (retornado pelo `getTamanho`) deve retornar o tamanho da memória informado no construtor.

Assim como nas memórias do EP1, o método `ler` (definido na classe mãe) deve retornar o **Dado** na posição de memória passada como parâmetro. Caso a posição seja inválida (negativa ou maior ou igual ao tamanho), o método deve jogar uma exceção do tipo `logic_error`.

O método `escrever` definido na classe mãe deve armazenar na posição de memória informada o **Dado** passado como parâmetro. Caso a posição de memória seja inválida, o método deve jogar uma exceção do tipo `logic_error`. Caso na posição de memória já exista um **Dado**, o objeto existente deve ser destruído.

A classe possui um método `escrever` específico, que recebe um `list` de **Dados**. Ele deve escrever os **Dados** contidos no `list` na memória a partir da posição 0 (inclusive). Ou seja, o **Dado** na posição 0 do `list` deve ser colocado na posição 0 da memória, o **Dado** na posição 1 do `list` deve ser colocado na posição 1 da memória e assim por diante. Caso o `list` passado tenha tamanho maior do que o tamanho da memória, o método não deve armazenar nada e deve jogar um `logic_error`.

O método `imprimir` deve ter o mesmo comportamento definido para a **MemoriaDeDados do EP1**. Ou seja, ele deve imprimir na saída padrão (usando o `cout`) os **Dados** armazenados seguindo o formato:

```
0: <dado na posição 0>
1: <dado na posição 1>
…
n: <dado na posição n>
```

onde `<dado na posição x>` é a impressão do dado (chamando o método `imprimir` do **Dado)**. Caso na posição não haja um **Dado** (seja nulo), deve-se imprimir o caractere '-'. Por exemplo, para uma

memória de tamanho 8 com uma **Instrução** J na posição 0, uma **Instrução** ADD na posição 2, um **Dado** com valor 5 na posição 3, e um **Dado** com valor 10 na posição 4, e o restante nulo, a impressão seria:

```
0: Instrucao 2
1: -
2: Instrucao 0
3: 5
4: 10
5: -
6: -
7: -
```

3.6 Classe ESMapeadaNaMemoria

A classe **ESMapeadaNaMemoria** representa um tipo de memória que tem uma área reservada para dispositivos. Com isso, ela é uma classe concreta filha de **Memoria**. Ela possui uma **MemoriaRAM** subjacente, a qual responde pelos endereços iniciais da **Memória**. Os dispositivos adicionados ficam nas posições seguintes à da **MemoriaRAM** (veja adiante um exemplo). Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
ESMapeadaNaMemoria(MemoriaRAM* m);
ESMapeadaNaMemoria(MemoriaRAM* m, vector<Dispositivo*>* dispositivos);
virtual ~ESMapeadaNaMemoria();

virtual MemoriaRAM* getMemoriaSubjacente();
virtual void adicionar(Dispositivo* d);
virtual vector<Dispositivo*>* getDispositivos();
```

A classe possui dois construtores: um que recebe apenas a **MemoriaRAM** subjacente e outro que recebe a **MemoriaRAM** e um `vector` com os **Dispositivos** a serem adicionados (em ordem). O destrutor deve destruir a **MemoriaRAM** e também todos os dispositivos adicionados.

O método `getMemoriaSubjacente` deve retornar a **MemoriaRAM** informada no construtor.

O método `adicionar` deve adicionar o **Dispositivo** à **ESMapeadaNaMemoria**, deixando-o na posição seguinte da **Memoria**. Por exemplo, caso a **MemoriaRAM** subjacente tenha tamanho 64, o primeiro dispositivo adicionado deve ficar na posição 64 (já que a **MemoriaRAM** vai de 0 a 63). O segundo dispositivo adicionado ficará na posição 65 – e assim por diante. O método `getDispositivos` deve retornar todos os dispositivos adicionados, na ordem em que foram adicionados.

O tamanho da **ESMapeadaNaMemoria** depende da quantidade de dispositivos adicionados. O método `getTamanho` deve retornar o tamanho da **MemoriaRAM** subjacente somado à quantidade de dispositivos. No exemplo anterior, o tamanho é 66 (**MemoriaRAM** de tamanho 64 mais 2 dispositivos).

O método `ler`, definido na classe mãe, deve considerar os dispositivos. O conteúdo mapeado da **MemoriaRAM** sempre deve começar na posição 0. Portanto, caso a posição acessada seja uma posição da **MemoriaRAM**, ele deve retornar o valor obtido pelo método `ler` da **MemoriaRAM** subjacente. Caso a posição acessada seja de um dispositivo, deve-se chamar o método `ler` do dispositivo, retornando o

valor dele. Caso a posição passada seja inválida, o método deve jogar uma exceção do tipo `logic_error`. Por exemplo, suponha uma **MemoriaRAM** de tamanho 32 (de 0 a 31) e dois dispositivos: um teclado (posição 32) e um monitor (posição 33). A chamada de `ler` para a posição 10 deve retornar o valor da posição 10 da **MemoriaRAM**. Por outro lado, a chamada de `ler` da posição 32 deve retornar o valor do método `ler` do teclado.

O método `escrever`, definido na classe mãe, tem comportamento análogo ao do `ler`. Caso a posição acessada seja da **MemoriaRAM**, deve ser chamado o método `escrever` dela; caso a posição seja de um dispositivo, deve-se chamar o `escrever` do `Dispositivo`; caso a posição seja inválida deve ser jogado um `logic_error`.

Note que são as instruções LW e SW que indicam a leitura e a escrita nos dispositivos. Por exemplo, se um teclado estiver na posição 32 da **ESMapeadaNaMemoria**, a instrução `LW 2, 32` indica que deve ser "lido" do teclado um **Dado** e ele deve ser armazenado no registrador 2. De forma similar, se um monitor estiver na posição 33, ao fazer `SW 2, 33` deve ser "escrito" no monitor o **Dado** contido no registrador 2.

O método `imprimir` deve apenas chamar o método `imprimir` da **MemoriaRAM**.

3.7 Dispositivo

A classe **Dispositivo** representa um dispositivo que pode ser usado pelo microprocessador. Essa classe deve ser **abstrata**, com os métodos `ler` e `escrever` abstratos. Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
Dispositivo();
virtual ~Dispositivo();
virtual Dado* ler();
virtual void escrever(Dado* d);
```

Não é definido um comportamento para o construtor e o destrutor. Os métodos `ler` e `escrever` são especificados para as classes filhas.

3.8 Monitor

A classe **Monitor** é um dispositivo que representa um monitor. Com isso, ela deve ser uma classe concreta filha de **Dispositivo**. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Monitor();
virtual ~Monitor();
```

Não é definido um comportamento para o construtor e o destrutor.

O método `ler` deve sempre jogar uma exceção do tipo `logic_error`, dado que um monitor não é um dispositivo de entrada.

O método `escrever` deve escrever o valor do **Dado** na saída padrão (`cout`).

3.9 MonitorDeChar

A classe **MonitorDeChar** é um dispositivo que representa um monitor que escreve caracteres. Com isso, ela deve ser uma classe concreta filha de **Monitor**. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
MonitorDeChar();
virtual ~MonitorDeChar();
```

Não é definido um comportamento para o construtor e o destrutor.

O método `ler` tem o mesmo comportamento definido na superclasse. O método `escrever`, por outro lado, deve escrever o valor do **Dado** na saída padrão (`cout`) fazendo um *cast* do valor para `char`. Isso permite a escrita de caracteres seguindo a tabela ASCII[3]. Por exemplo, para um **Dado** com valor 97, ao fazer um `cout` desse valor convertido para `char` será impresso o caractere 'a' na tela.

3.10 Teclado

A classe **Teclado** é um dispositivo que representa um teclado. Com isso, ela deve ser uma classe concreta filha de **Dispositivo**. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Teclado();
virtual ~Teclado();
```

Não é definido um comportamento para o construtor e o destrutor.

O método `escrever` deve sempre jogar uma exceção do tipo `logic_error`, dado que um teclado não é um dispositivo de saída.

O método `ler` deve imprimir o texto "`Digite um numero: `" (sem as aspas; note o espaço após o ':' e não pule uma linha após o texto) e então ler um *número* da entrada padrão (`cin`). O número lido deve ser armazenado em um **Dado**, o qual é retornado pelo método.

3.11 TecladoDeChar

A classe **TecladoDeChar** é um teclado que lê caracteres. Com isso, ela deve ser uma classe concreta filha de **Teclado**. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
TecladoDeChar();
virtual ~TecladoDeChar();
```

Não é definido um comportamento para o construtor e o destrutor.

O método `escrever` tem o mesmo comportamento da classe mãe. O método `ler` deve imprimir o texto "`Digite um caractere: `" (sem as aspas; note o espaço após o ':' e não pule uma linha após o texto) e

[3] https://www.ime.usp.br/~pf/algoritmos/apend/ascii.html

então ler um caractere da entrada padrão (`cin`). O caractere lido deve ser convertido para inteiro (por um cast) e armazenado em um **Dado**, o qual é retornado pelo método.

3.12 Classe UnidadeDeControle

A **UnidadeDeControle** é a classe responsável por obter e executar as instruções do microprocessador. Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
UnidadeDeControle(BancoDeRegistradores* registradores, Memoria* memoria);
virtual ~UnidadeDeControle();

virtual BancoDeRegistradores* getBancoDeRegistradores();
virtual Memoria* getMemoria();
virtual int getPC();
virtual void setPC(int pc);
virtual void executarInstrucao();
```

A classe sofreu poucas alterações em relação ao EP1. A principal é que o construtor recebe apenas uma **Memoria** no construtor, além de um **BancoDeRegistradores**. Os métodos `getMemoria` e `getBancoDeRegistradores` retornam os objetos informados no construtor. Assim como no EP1, o construtor deve inicializar o contador de programa (PC) em 0. O destrutor deve destruir o **BancoDeRegistradores** e a **Memoria**.

O método `getPC` deve retornar o valor do contador de programa (PC). O método `setPC` deve definir o valor do PC.

O método `executarInstrucao` tem o mesmo comportamento do EP1. Consulte o enunciado do EP1 para entender o seu funcionamento. Note que não há um tratamento especial de exceções nessa classe (essa é tarefa do *main*).

4 Persistência

O software permitirá carregar e salvar o estado da **MemoriaRAM** em arquivo. Para isso deve ser implementada a classe **GerenciadorDeMemoria**. A seguir é apresentado o formato do arquivo e a especificação da classe. Junto com o enunciado são fornecidos dois exemplos de arquivos (`hello.txt` e `fatorial.txt`).

4.1 Formato do arquivo

A persistência da **MemoriaRAM** deve seguir o formato de arquivo apresentado a seguir. Entre "<" e ">" são especificados os valores esperados. Por simplicidade, será utilizado um caractere de nova linha ('\n') ou espaço (' ') como delimitador (pode ser qualquer um deles). O *n* representa o tamanho da memória.

```
<n>
<dado na posição 1>
<dado na posição 2>
...
<dado na posição n>
```

Note que **há uma linha em branco no final do arquivo**.

O formato do dado deve ser o seguinte:

- Se for um **Dado** (não **Instrucao**):

  `D <valor>`

  Onde `<valor>` é o valor do dado. Por exemplo:

  ```
  D 5
  D 10
  ```

- Se for uma **Instrucao**, o formato geral é o seguinte:

  `<tipo> <campo 1> <campo 2> ... <campo n>`

  Onde:

  - `<tipo>` é uma string indicando o tipo da instrução: "ADD", "SUB", "MULT", "DIV", "LW", "SW", "J", "BNE", "BEQ".
  - `<campo 1>`, `<campo 2>` ... `<campo n>` são os campos necessários para a instrução (valores inteiros).

  Dependendo do tipo de instrução, a quantidade de campos é diferente:

  - Instruções tipo LW e SW possuem 2 campos, *destino* e *imediato*:

    ```
    LW <destino> <imediato>
    SW <destino> <imediato>
    ```

  - Instruções tipo J possui apenas 1 campo *imediato*:

    ```
    J <imediato>
    ```

  - Instruções tipo BNE e BEQ possuem 3 campos, para *origem1*, *origem2* e *imediato*:

    ```
    BNE <origem 1> <origem 2> <imediato>
    BEQ <origem 1> <origem 2> <imediato>
    ```

  - Instruções ADD e SUB possuem 3 campos, para *destino*, *origem1* e *origem2*:

    ```
    ADD <destino> <origem 1> <origem 2>
    SUB <destino> <origem 1> <origem 2>
    ```

  - Instruções MULT e DIV possuem 2 campos, para *origem1* e *origem2*:

    ```
    MULT <origem 1> <origem 2>
    DIV <origem 1> <origem 2>
    ```

- Se a memória tiver nulo[4], deve ser colocada a string "-".

Como exemplo veja os arquivos *hello.txt* e *fatorial.txt* entregues junto com o enunciado.

4.2 Classe GerenciadorDeMemoria

A classe **GerenciadorDeMemoria** é a classe responsável por carregar e armazenar o conteúdo da memória de ou para um arquivo texto. Os únicos métodos públicos que a classe deve possuir são:

```
GerenciadorDeMemoria();
virtual ~GerenciadorDeMemoria();
virtual void load(string arquivo, MemoriaRAM* m);
virtual void dump(string arquivo, MemoriaRAM* m);
```

[4] Na memória real não existe nulo. Isso é apenas uma simplificação do EP.

O construtor e o destrutor não têm comportamento definido. O método `load` deve carregar o conteúdo do arquivo para a `MemoriaRAM` passada como parâmetro, considerando o formato descrito anteriormente. Caso o arquivo não exista, ou haja um problema de leitura (erro de formato ou outro problema), ou o conteúdo do arquivo não caiba na memória, jogue uma exceção do tipo `runtime_error`.

O método `dump` deve armazenar no arquivo passado como parâmetro o conteúdo da `MemoriaRAM` informada, seguindo o formato descrito anteriormente. Caso não seja possível escrever no arquivo, ou exista uma instrução desconhecida, jogue uma exceção do tipo `runtime_error`. Se o arquivo informado não existir, deve ser criado um novo arquivo (é o comportamento padrão da escrita).

5 Main

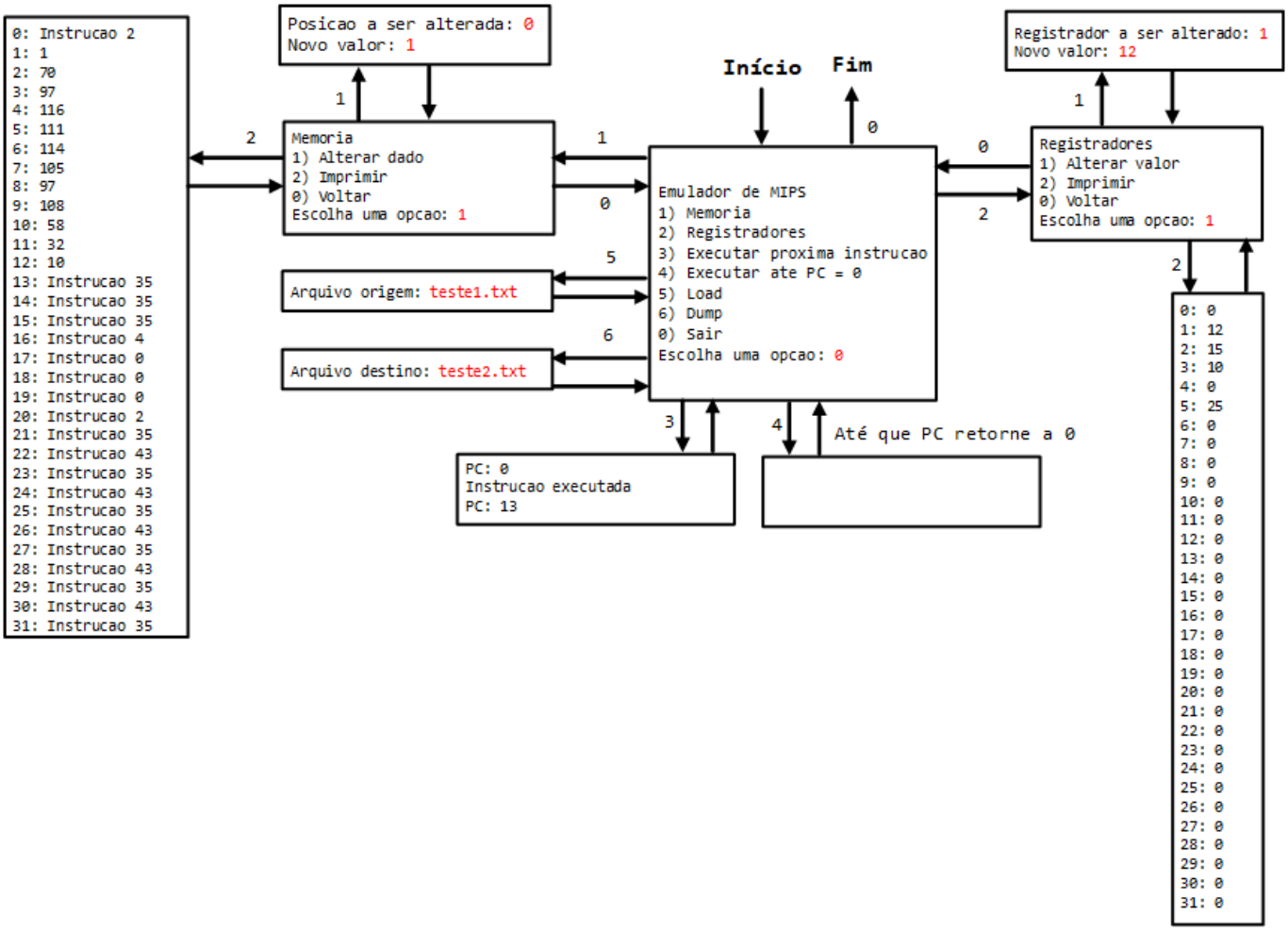
Coloque a função `main` em um arquivo separado, chamado `main.cpp`. Nela você deverá criar um **BancoDeRegistradores**, uma **MemoriaRAM** de tamanho 64 e uma **ESMapeadaNaMemoria** usando essa **MemoriaRAM**. Na **ESMapeadaNaMemoria** você deve adicionar um **Teclado**, um **TecladoDeChar**, um **Monitor** e um **MonitorDeChar**, nessa ordem (ou seja, o **Teclado** fica na posição 64, o **TecladoDeChar** na 65, o **Monitor** na 66 e o **MonitorDeChar** na 67). Crie uma **UnidadeDeControle** com o **BancoDeRegistradores** e a **ESMapeadaNaMemoria** criados**.**

5.1 Interface

Além de criar os objetos, a `main` deve possuir uma interface em console similar à do EP1 (com 3 opções a mais), que permite manipular a **UnidadeDeControle**. Essa interface é apresentada esquematicamente no diagrama a seguir. Cada retângulo representa uma "tela", ou seja, um conjunto de texto impresso e solicitado ao usuário. As setas representam as transições de uma tela para outra – os textos na seta representam o valor que deve ser digitado para ir para a tela destino ou a condição necessária (quando não há um texto é porque a transição acontece incondicionalmente). Em **vermelho** são apresentados exemplos de dados inseridos pelo usuário. Para reduzir o tamanho do diagrama considerou-se uma **MemoriaRAM** de tamanho 32, ao invés da de tamanho 64.

- A opção 1 ("Memoria") deve ir para a tela de Memória:
  - A opção 1 da tela de Memória ("Alterar dado") deve armazenar na posição de memória informada o dado digitado (por simplicidade não é possível armazenar uma instrução).
  - A opção 2 da tela de Memória ("Imprimir") deve chamar o método `imprimir` da **Memoria**.
  - A opção 0 da tela de Memória ("Voltar") deve retornar à tela principal.
- A opção 2 ("Registradores") deve ir para a tela de Registrador (é a mesma do EP1):
  - A opção 1 da tela de Registrador ("Alterar valor") deve armazenar no registrador informado o valor informado.
  - A opção 2 da tela de Registrador ("Imprimir") deve chamar o método `imprimir` da **BancoDeRegistradores**.
  - A opção 0 da tela de Registrador ("Voltar") deve retornar à tela principal.
- A opção 3 ("Executar proxima instrucao") deve imprimir o valor do PC antes de executar a instrução, chamar o método `executarInstrucao` da **UnidadeDeControle** e imprimir o novo

valor do PC (igual ao EP1). Caso aconteça uma exceção, apresente a mensagem da exceção (o `what`) e volte ao menu.



- A opção 4 (“Executar ate PC = 0”) deve fazer um laço que executa instruções (através do método `executarInstrucao` da **UnidadeDeControle**) até que o PC *volte* ao valor 0. Ou seja, caso o PC valha 0, ao escolher essa opção deve ser chamado o método `executarInstrucao` até que o PC retorne a 0. Caso aconteça uma exceção, pare a execução, apresente a mensagem da exceção (o `what`) e volte ao menu.
- A opção 5 (“Load”) deve carregar o conteúdo da **MemoriaRAM** de um arquivo, usando o método `load` do **GerenciadorDeMemoria**. Caso aconteça uma exceção, apresente a mensagem dela e volte ao menu.
- A opção 6 (“Dump”) deve salvar o conteúdo da **MemoriaRAM** em um arquivo, usando o método `dump` do **GerenciadorDeMemoria**. Caso aconteça uma exceção, apresente a mensagem dela e volte ao menu.
- A opção 0 (“Sair”) deve terminar o programa, destruindo a unidade de controle.
- No diagrama não são apresentados casos de erro (por exemplo, a digitação de uma opção de menu inválida ou de um texto em um valor que deveria ser um número). Não é necessário fazer tratamento disso. Assuma que o usuário *sempre* digitará um valor correto.

Atenção: A interface com o usuário deve seguir exatamente o especificado (incluindo ":" e espaços entre ")" e ":" e o restante do texto). Se ela não for seguida, haverá desconto de nota. Não adicione outros textos além dos apresentados no diagrama e especificados.

5.2 Exemplo

Segue um exemplo de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário.

```
Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 5

Arquivo origem: hello.txt

Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 3

PC: 0
Instrucao executada
PC: 15


Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 3

PC: 15
Instrucao executada
PC: 16


Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 3

PC: 16
HInstrucao executada
PC: 17
```

```
Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 2

Registradores
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 2

0: 0
1: 0
2: 72
3: 0
4: 0
5: 0
6: 0
7: 0
8: 0
9: 0
10: 0
11: 0
12: 0
13: 0
14: 0
15: 0
16: 0
17: 0
18: 0
19: 0
20: 0
21: 0
22: 0
23: 0
24: 0
25: 0
26: 0
27: 0
28: 0
29: 0
30: 0
31: 0

Registradores
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 0

Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 4

ello, World!
```

```
Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 5

Arquivo origem: fatorial.txt

Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 4

Digite um numero: 5
Fatorial: 120

Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 1

Memoria
1) Alterar dado
2) Imprimir
0) Voltar
Escolha uma opcao: 2

0: Instrucao 2
1: 1
2: 70
3: 97
4: 116
5: 111
6: 114
7: 105
8: 97
9: 108
10: 58
11: 32
12: 10
13: Instrucao 35
14: Instrucao 35
15: Instrucao 35
16: Instrucao 4
17: Instrucao 0
18: Instrucao 0
19: Instrucao 0
20: Instrucao 2
21: Instrucao 35
22: Instrucao 43
23: Instrucao 35
24: Instrucao 43
25: Instrucao 35
26: Instrucao 43
27: Instrucao 35
```

```
28: Instrucao 43
29: Instrucao 35
30: Instrucao 43
31: Instrucao 35
32: Instrucao 43
33: Instrucao 35
34: Instrucao 43
35: Instrucao 35
36: Instrucao 43
37: Instrucao 35
38: Instrucao 43
39: Instrucao 35
40: Instrucao 43
41: Instrucao 43
42: Instrucao 35
43: Instrucao 43
44: Instrucao 2
45: -
46: -
47: -
48: -
49: -
50: -
51: -
52: -
53: -
54: -
55: -
56: -
57: -
58: -
59: -
60: -
61: -
62: -
63: -

Memoria
1) Alterar dado
2) Imprimir
0) Voltar
Escolha uma opcao: 0

Emulador de MIPS
1) Memoria
2) Registradores
3) Executar proxima instrucao
4) Executar ate PC = 0
5) Load
6) Dump
0) Sair
Escolha uma opcao: 0
```

6 Entrega

O projeto deverá ser entregue até dia **04/12** em <https://laboo.pcs.usp.br/ep/>.

Atenção:

- Deve ser mantida a mesma dupla do EP1. É possível apenas *desfazer* a dupla. Com isso, cada aluno deve fazer uma entrega diferente (e em separado). Caso você deseje fazer isso, envie um e-mail para levy.siqueira@usp.br até dia **25/11** informando os números USP dos alunos e mandando-o com cópia para a sua dupla.
- Não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e os grupos envolvidos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um colega de outro grupo!

Entregue todos os arquivos, inclusive o `main` (que deve **obrigatoriamente** ficar em um arquivo "main.cpp"), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, *podem* não ser reconhecidos e acarretar **nota 0**). O nome do arquivo não pode conter espaço, ".", acentos ou ter mais de 11 caracteres. Os códigos fonte não devem ser colocados em pastas. A submissão pode ser feita por qualquer um dos membros da dupla – recomenda-se que os dois submetam.

Atenção: faça a submissão do mesmo arquivo nos 4 problemas (Parte 1, Parte 2, Parte 3 e Parte 4). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

Siga a convenção de nomes para os arquivos ".h" e ".cpp". O não atendimento disso pode levar a erros de compilação (e, consequentemente, **nota zero**).

Ao submeter os arquivos no Judge será feita **apenas** uma verificação básica buscando evitar erros de compilação devido à erros de digitação do nome das classes e dos métodos públicos. **Note que a nota dada não é a nota final**: nesse momento não são executados testes – o Judge apenas tenta chamar todos os métodos definidos neste documento para todas as classes.

Você pode submeter quantas vezes quiser, sem desconto na nota.

7 Dicas

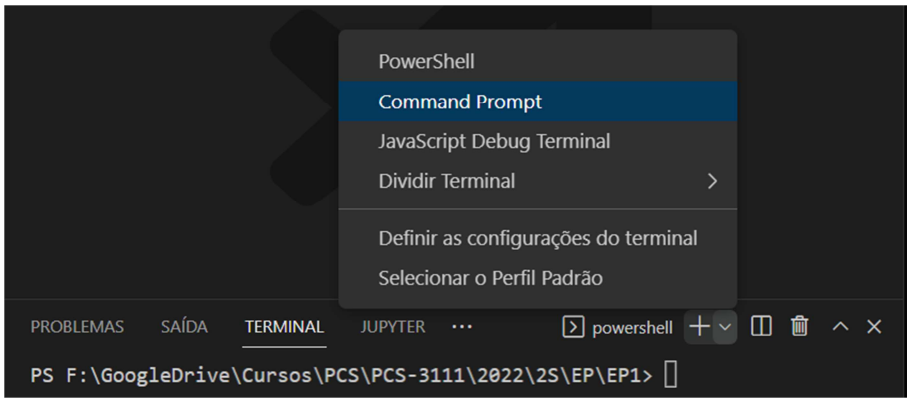
- Para testar o método `dump` do **GerenciadorDeMemoria** você pode fazer o `load` de um dos exemplos e ver o resultado do `dump` dele.
- **Entregue um EP que compila**! Caso você não consiga implementar um método (ou ele esteja com erro de compilação), faça uma implementação *dummy* dele. Uma implementação dummy é a implementação mais simples possível do método, que permite a compilação. Por exemplo, uma implementação *dummy* – e errada – do método `getDispositivos` da classe **ESMapeadaNaMemoria** é:

```
vector<Dispositivo*>* ESMapeadaNaMemoria::getDispositivos() {
      return nullptr;
}
```

- Caso o programa esteja travando (terminando abruptamente), execute o programa no modo de depuração. O depurador informará o erro que aconteceu – além de ser possível depurar para descobrir onde o erro acontece!
- Faça `#include` apenas das classes que são usadas naquele arquivo. Por exemplo, se o arquivo .h não usa a classe **X**, mas o .cpp usa essa classe, faça o include da classe **X** *apenas* no .cpp. Incluir classes desnecessariamente pode gerar erros de compilação estranhos (por causa de referências circulares).
    - Inclua todas as dependências necessárias. Não dependa de #includes feitos por outros arquivos incluídos.
- É muito trabalhoso testar o programa ao executar o main *com menus*, já que é necessário informar vários dados para inicializar os registradores e a memória de dados. Para testar o programa faça o `main` chamar uma função de teste que cria objetos com valores interessantes para testar, sem pedir entrada para o usuário. Não se esqueça de remover a função de teste ao entregar a versão final do EP.
    - Uma outra opção para testar é usar o comando:

      ```
      ep < entrada.txt > saida.txt
      ```

      Esse comando executa o programa ep usando como entrada do teclado o texto no arquivo entrada.txt e coloca em saída.txt os textos impressos pelo programa (sem os valores digitados). No caso do Windows, para rodar esse comando você precisa de um prompt de comando (por uma limitação do *PowerShell*). Para fazer isso, clique na seta para baixo do lado do + no terminal e escolha Command Prompt.



- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- Separe o `main` em funções para organizar melhor o código.
- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *RuntimeErrors* e outros tipos de erros no Judge que não aparecem ao executar o programa no Windows. Veja a mensagem de erro do Judge para descobrir em qual classe acontece o problema. Caso você queira testar o projeto em um compilador similar ao do Judge, use o site https://repl.it/ (note que ele não tem depurador *ainda*).

    - Em geral *RuntimeErrors* acontecem porque você não inicializou um atributo que é usado. Por exemplo, caso você não crie um vetor ou não inicialize o atributo quantidade, para controlar o tamanho do vetor, ocorrerá um *RuntimeError*.
- Use o Fórum de dúvidas do EP no e-Disciplinas para esclarecer dúvidas no enunciado ou problemas de submissão no Judge.
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**