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 1 – 2º SEMESTRE DE 2022

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.

1 Introdução

A arquitetura de Von Neumann (apresentada na Aula 2 da disciplina) divide o computador em três componentes principais: memória, CPU, e entrada e saída (E/S). Na memória são armazenados dados e instruções. A CPU, mais especificamente a unidade de controle, obtêm da memória a próxima instrução a ser lida, e então faz com que ela seja executada, o que pode envolver obter dados da memória. Ao executar algumas instruções pode-se comunicar com um dispositivo de entrada e saída (por exemplo, obter um caractere digitado ou imprimir algo na tela). Em um computador real a relação entre a memória, CPU e E/S envolve diversos outros detalhes, especialmente ao considerar questões de desempenho.

Nas disciplinas de Sistemas Digitais (PCS3115 e PCS3225), do segundo ano de Engenharia Elétrica, serão apresentados os conceitos, componentes básicos e princípios de projeto de um computador e de sistemas digitais em geral. No EP desta disciplina veremos alguns conceitos de forma bastante simplificada, usando o livro de Harris e Harris como referência[1]. Como base, usaremos a arquitetura de um microprocessador MIPS – um microprocessador é "um sistema digital que lê e executa instruções em linguagem de máquina"[2]. Existem diversas arquiteturas, como a ARM, RISC-V, x86, SPARC, PowerPC, além da MIPS.

Neste EP seguiremos uma arquitetura *monociclo*, que é bem simples – mas inadequada para microprocessadores reais (no próximo EP faremos algumas melhorias). Nela, o microprocessador MIPS é composto por 4 componentes:

- **Memória de Instruções**: memória onde são armazenadas as instruções a serem executadas.
- **Memória de dados**: memória onde são armazenados os dados do programa.

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

[2] Ibid, p.295.

- **Banco de registradores**: registradores são elementos extremamente rápidos que armazenam o último dado informado. Por uma questão de desempenho, os microprocessadores usam um conjunto desses elementos para armazenar dados temporários.
- **Unidade de controle**: obtêm a próxima instrução e faz com que ela seja executada.

Cada arquitetura possui um conjunto diferente de instruções, que são as operações executadas pelo microprocessador. Por exemplo, existem instruções para somar, subtrair, mover um dado para um registrador e mover um dado em uma posição da memória para um registrador. Cada instrução possui um código para identificá-la, chamado de *opcode* (de *operation code*, ou código de operação), e um conjunto de campos para indicar os operandos. Por exemplo, uma instrução para adicionar na arquitetura MIPS (chamada de ADD) tem *opcode* 0, e possui campos para três registradores (um registrador de destino e dois de origem) e o código de função 32. Uma instrução para ler um dado da memória (chamada de LW, de *load word*, ou carregar palavra) tem *opcode* 35 e dois campos: o registrador de destino (onde será colocado o dado) e o endereço da memória a ser carregado. Um compilador traduz o código escrito em uma linguagem de programação de alto nível, como C++, para um conjunto de instruções. Em geral, um único comando em C++ é traduzido para diversas instruções que executam a funcionalidade equivalente na arquitetura alvo da compilação.

O endereço da instrução em execução é guardado em um registrador especial, o contador do programa (*Program Counter* – PC). Ao terminar a execução de uma instrução, esse registrador é tipicamente incrementado. A exceção é quando se faz um desvio, saltando várias instruções, o que é necessário quando se tem laços, condicionais ou chamadas de função, por exemplo.

1.1 Simplificações

Para facilitar o desenvolvimento – e focar nos conceitos de Orientação a Objetos –, o EP faz algumas simplificações. A principal é trabalhar com inteiros completos (por exemplo, de 32 bits). Com isso, cada campo de uma instrução é um inteiro. Além disso, os endereços são inteiros.

As instruções também foram simplificadas, removendo alguns campos e usando alguns campos diferentes. Além disso, a quantidade de instruções foi limitada. Apesar de o MIPS ser uma arquitetura com um conjunto reduzido de instruções (RISC), trabalharemos com apenas 9 instruções – ao invés das mais de 50 especificadas no livro de Harris e Harris[3].

1.2 Objetivo

O objetivo deste projeto é fazer um emulador simplificado de um microprocessador MIPS. Este projeto será desenvolvido incrementalmente e **em dupla** nos dois Exercícios Programas de PCS3111.

Neste primeiro EP será implementada uma arquitetura *monociclo* baseada no MIPS. No próximo EP serão feitas melhorias nessa arquitetura, por exemplo, para permitir o uso de dispositivos.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor – o que representa o conteúdo até, *inclusive*, a Aula 5. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

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

2 Projeto

Deve-se implementar em C++ as classes **Dado**, **Instrucao**, **BancoDeRegistradores**, **MemoriaDeDados**, **MemoriaDeInstrucoes**, **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, caso necessário**.
> 2. Não faça outros `#defines` além dos definidos neste documento. Use os valores de `#define` deste documento.
>
> 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.

A sugestão é que se comece a implementar o EP a partir da *Aula 4*. Todas as classes possuem construtores e destrutores (conceito da *Aula 5* – são os métodos com o mesmo nome da classe), mas é possível começar a implementar os métodos ainda sem esse conceito. Coloque a palavra *virtual* nos destrutores, como indicado. Não se preocupe com isso – será explicado o significado na Aula 7.

2.1 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();
int getValor();
void imprimir();
```

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.

2.2 Classe Instrucao

A **Instrucao** é uma operação executada pelo microprocessador. Dependendo do tipo da instrução, ela pode possuir os seguintes campos:

- **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.

Com isso, a classe **Instrucao** deve possuir apenas os seguintes **métodos públicos** e **defines**:

```
#define TIPO_R 0
#define FUNCAO_ADD 32
#define FUNCAO_SUB 34
#define FUNCAO_MULT 24
#define FUNCAO_DIV 26
#define J 2
#define BNE 5
#define BEQ 4
#define LW 35
#define SW 43

Instrucao(int opcode, int origem1, int origem2, int destino, int imediato,
          int funcao);
virtual ~Instrucao();
int getOpcode();
int getOrigem1();
int getOrigem2();
int getDestino();
int getImediato();
int getFuncao();
```

O construtor recebe todos os possíveis campos. Os métodos `getOpcode`, `getOrigem1`, `getOrigem2`, `getDestino`, `getImediato` e `getFuncao` devem retornar o valor informado pelo construtor.

Para este EP são definidas as instruções ADD, SUB, MULT, DIV, J, BNE, BEQ, LW e SW, baseada na arquitetura MIPS. As instruções aritméticas (ADD, SUB, MULT e DIV) são instruções tipo R (de *tipo registrador*), possuindo o mesmo *opcode*, sendo diferenciadas pelo campo *função*. As demais possuem *opcodes* diferentes. Os valores dos *opcodes* e dos campos de função estão nos *defines* da classe.

Caso a instrução não use o campo em questão, o valor informado será ignorado pela **UnidadeDeControle** ao processá-la.

A seguir são apresentados os campos esperados e a semântica de cada instrução:

ADD destino, origem1, origem2
Instrução tipo R (opcode 0) e função FUNCAO_ADD (32). Soma o conteúdo do registrador de *origem 1* com o conteúdo do registrador de *origem 2*, colocando o resultado no registrador de *destino*.

Exemplo: `ADD 5, 2, 3`
Considerando que o registrador 2 possui o valor 8 e o registrador 3 possui o valor 10, após a execução da instrução será colocado no registrador 5 o valor 18.

SUB destino, origem1, origem2
Instrução tipo R (opcode 0) e função FUNCAO_SUB (34). É feito o conteúdo do registrador de *origem 1* menos o conteúdo do registrador de *origem 2*, colocando o resultado no registrador de *destino*.

Exemplo: `SUB 5, 2, 3`
Considerando que o registrador 2 possui o valor 8 e o registrador 3 possui o valor 10, após a execução da instrução será colocado no registrador 5 o valor -2.

MULT origem1, origem2
Instrução tipo R (opcode 0) e função FUNCAO_MULT (24). O conteúdo do registrador de *origem 1* é multiplicado pelo conteúdo do registrador de *origem 2*. O resultado da multiplicação é colocado no registrador[4] 24.

Exemplo: `MULT 2, 3`
Considerando que o registrador 2 possui o valor 8 e o registrador 3 possui o valor 10, após a execução da instrução será colocado no registrador 24 o valor 80.

DIV origem1, origem2
Instrução tipo R (opcode 0) e função FUNCAO_DIV (26). O conteúdo do registrador de *origem 1* é dividido pelo conteúdo do registrador de *origem 2*. O resultado da divisão é colocado no registrador 24, enquanto que o resto da divisão é colocado no registrador[5] 25.

Exemplo: `DIV 2, 3`
Considerando que o registrador 2 possui o valor 10 e o registrador 3 possui o valor 6, após a execução da instrução será colocado no registrador 24 o valor 1 e no registrador 25 o valor 4.

J imediato
O PC (*program counter*) é alterado para o valor do imediato.

Exemplo: `J 10`
Considerando que o PC está com valor 3, após a execução dessa instrução, o PC ficará com valor 10 e a próxima instrução a ser executada será a que estiver na posição 10 da memória de instrução.

BNE origem1, origem2, imediato
O conteúdo do registrador *origem* 1 é comparado com o conteúdo do registrador *origem2*. Caso o valor seja *diferente* (*not equal*), o PC é alterado para o valor do imediato.

Exemplo: `BNE 2, 3, 10`
Considerando que o registrador 2 possui o valor 10 e o registrador 3 possui o valor 6, após a execução da instrução o PC será alterado para o valor 10. A próxima instrução a ser executada será a que estiver na posição 10 da memória de instrução.

Caso o valor seja igual, não é feito nada (ou seja, o PC apenas é incrementado para a próxima instrução).

[4] Na arquitetura MIPS verdadeira, o valor é colocado nos registradores HI e LO e existem instruções específicas para lidar com eles.

[5] Assim como na multiplicação, na arquitetura MIPS verdadeira, o valor é colocado nos registradores HI e LO.

BEQ origem1, origem2, imediato
O conteúdo do registrador *origem* 1 é comparado com o conteúdo do registrador *origem2*. Caso o valor seja *igual*, o PC é alterado para o valor do imediato.

Exemplo: `BEQ 2, 3, 8`
Considerando que o registrador 2 possui o valor 10 e o registrador 3 possui o valor 10, após a execução da instrução o PC será alterado para o valor 8. A próxima instrução a ser executada será a que estiver na posição 8 da memória de instrução.

Caso o valor seja diferente, não é feito nada (ou seja, o PC apenas é incrementado para a próxima instrução).

LW destino, imediato
Carrega no registrador *destino* o valor apontado pelo endereço da memória de dados em *imediato*. Se a memória tiver nulo, deve-se carregar o valor 0 no registrador.

Exemplo: `LW 2, 10`
Considerando que a posição 10 na memória de dados contém o valor 4, após a execução da instrução será armazenado no registrador 2 o valor 4.

SW destino, imediato
Armazena no endereço da memória de dados em *imediato* o valor do registrador em *destino*.

Exemplo: `SW 2, 10`
Considerando que o registrador 2 possui o valor 30, após a execução da instrução será armazenado na posição de memória de dados 10 o valor 30.

2.3 Classe BancoDeRegistradores

O **BancoDeRegistradores** é o conjunto de registradores disponíveis ao microprocessador. A quantidade de registradores é fixa na arquitetura. No caso do MIPS, são definidos 32 registradores. Cada registrador possui uma função especial – por exemplo, existe um registrador que armazena o endereço de retorno da função e outro que guarda o retorno da função. Por simplicidade, apenas 3 registradores terão função específica neste EP:

- 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();
int getValor(int registrador);
void setValor(int registrador, int valor);
void imprimir();
```

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 retornar 0.

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

O método imprimir 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. Por exemplo, após criar o **BancoDeRegistradores** esse método deve imprimir:

```
0: 0
1: 0
2: 0
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
```

2.4 Classe MemoriaDeDados

A **MemoriaDeDados** é a memória que armazena os **Dados** do microprocessador. Essa classe deve possuir apenas os seguintes métodos **públicos**:

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

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 método `getTamanho` deve retornar o tamanho da memória, informado no construtor.

O método `ler` 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 retornar `NULL`.

O método `escrever` deve armazenar na posição de memória informada o **Dado** passado como parâmetro. Caso a posição de memória seja inválida, o método não deve fazer nada e retornar `false`, caso contrário ele deve retornar `true`. Caso na posição de memória já exista um **Dado**, o objeto existente deve ser destruído.

O método `imprimir` deve imprimir na saída padrão (usando o `cout`) os **Dados** armazenados na **MemoriaDeDados**, 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**, deve-se imprimir o caractere '-'. Por exemplo, para uma memória de tamanho 8 com o **Dado** com valor 5 na posição 3, um **Dado** com valor 10 na posição 4 e o restante nulo, a impressão seria:

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

2.5 Classe MemoriaDeInstrucoes

Em um processador monociclo as instruções são armazenadas em uma memória específica, a **MemoriaDeInstrucoes.** Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
MemoriaDeInstrucoes(int tamanho);
virtual ~MemoriaDeInstrucoes();
int getTamanho();
Instrucao* ler(int posicao);
bool escrever(int posicao, Instrucao* d);
```

O funcionamento dessa classe é muito similar à **MemoriaDeDados**. O construtor deve receber o tamanho da memória e alocar um vetor de objetos **Instrucao** com o tamanho informado. Ao inicializar, deve ser colocado `NULL` em cada posição, indicando que ela não contém uma instrução. O destrutor deve destruir todos os objetos **Instrucao** que ainda estão no vetor, além de destruir o vetor alocado dinamicamente.

O método `getTamanho` deve retornar o tamanho da memória, informado no construtor.

O método `ler` deve retornar a **Instrucao** na posição de memória passada como parâmetro. Caso a posição seja inválida (negativa ou maior ou igual ao tamanho), o método deve retornar `NULL`.

O método `escrever` deve armazenar na posição de memória informada a **Instrucao** passada como parâmetro. Caso a posição de memória seja inválida, o método não deve fazer nada e retornar `false`, caso contrário ele deve retornar `true`. Caso na posição de memória já exista uma **Instrucao**, o objeto existente deve ser destruído.

Note que essa classe não possui um método `imprimir`.

2.6 Classe UnidadeDeControle

A **UnidadeDeControle** é a classe responsável por obter e executar as instruções do microprocessador. Para isso ele deve ter um contador de programa (PC – *program counter*), que armazena a posição da memória de instruções onde está a próxima **Instrucao** a ser lida[6]. Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
UnidadeDeControle(BancoDeRegistradores* registradores,
                  MemoriaDeInstrucoes* instrucoes, MemoriaDeDados* dados);
virtual ~UnidadeDeControle();
BancoDeRegistradores* getBancoDeRegistradores();
MemoriaDeDados* getMemoriaDeDados();
MemoriaDeInstrucoes* getMemoriaDeInstrucoes();
int getPC();
void setPC(int pc);
void executarInstrucao();
```

[6] Na verdade, o contador do programa faz parte do *fluxo de dados*. Para simplificar, preferiu-se evitar essa classe neste EP.

O construtor da **UnidadeDeControle** deve receber o **BancoDeRegistradores**, a **MemoriaDeInstrucoes** e a **MemoriaDeDados** que serão usadas. Além disso, ele deve inicializar o contador de programa (PC) em 0. O destrutor deve destruir o **BancoDeRegistradores**, a **MemoriaDeInstrucoes** e a **MemoriaDeDados**.

Os métodos `getBancoDeRegistradores`, `getMemoriaDeDados` e `getMemoriaDeInstrucoes` devem retornar os objetos passados no construtor. O método `getPC` deve retornar o valor do contador de programa (PC). O método `setPC` deve definir o valor do PC.
O método `executarInstrucao` é o método principal dessa classe. Ele deve ler a instrução apontada pelo contador de programa e executá-la. Caso a instrução apontada seja nula, deve-se incrementar o PC e retornar. Caso contrário, deve-se executar a instrução. Para isso, deve-se obter o *opcode* da instrução e executar o seu comportamento, especificado na Seção 2.2. Após executar as instruções TIPO_R, LW e SW deve-se incrementar em 1 o PC (caso a instrução tenha *opcode* inválido ou função inválida, apenas incremente em 1 o PC). No caso das instruções BNE e BEQ, só deve-se incrementar o PC caso o valor do PC não seja alterado pela instrução. Ou seja, no BNE o PC só deve ser incrementado se os registradores tiverem valores iguais e na instrução BEQ caso os registradores tiverem valores diferentes. Na instrução J o PC nunca deve ser incrementado após a execução (uma vez que o valor dele já será alterado pela instrução).

Considere que não serão escritas instruções com registradores inválidos ou tentativa de acesso à memória fora do limite. Isso será tratado no EP2. Considere também que não ocorre divisão por 0.

3 Main

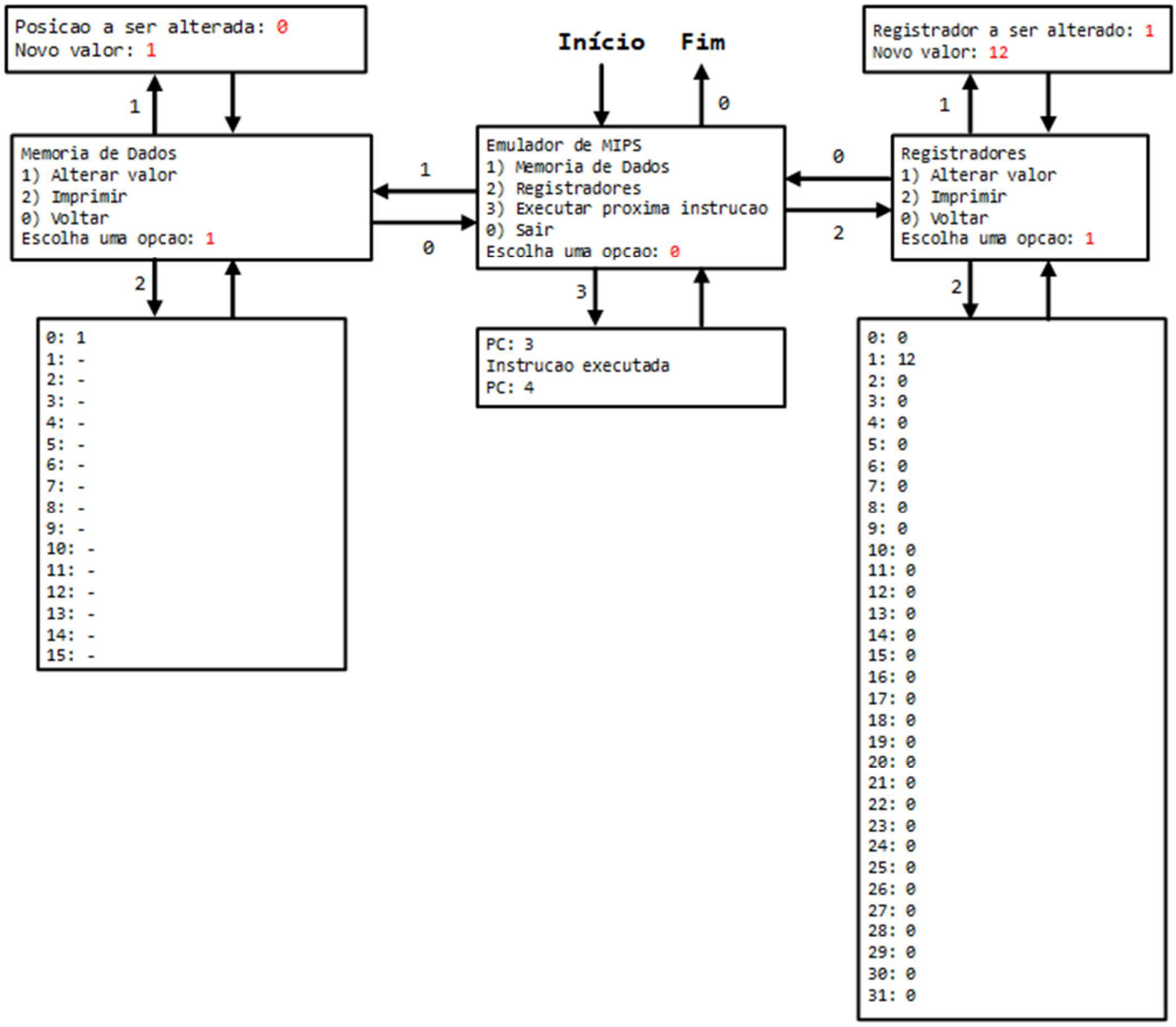
Coloque a `main` em um arquivo separado, chamado `main.cpp`. Nele você deverá criar um **BancoDeRegistradores**, uma **MemoriaDeDados** com tamanho 16, uma **MemoriaDeInstrucoes** com tamanho 16 e uma **UnidadeDeControle** (usando os outros objetos criados). Na **MemoriaDeInstrucose** escreva o seguinte programa a partir da posição 0:

```
LW 8, 0
LW 9, 1
LW 10, 2
BNE 9, 10, 7
MULT 8, 10
SW 24, 3
J 0
ADD 9, 10, 8
SW 9, 3
J 10
```

3.1 Interface

Além de criar os objetos e armazenar as instruções indicadas anteriormente, o `main` deve possuir uma interface em console que permite alterar os valores de registradores e dados na memória de dados criados e executar instruções seguindo o PC da **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.



- A opção 1 (“Memoria de Dados”) deve ir para a tela de Memória de Dado:
  - A opção 1 da tela de Memória de Dado (“Alterar valor”) deve armazenar na posição de memória informada o valor informado.
  - A opção 2 da tela de Memória de Dado (“Imprimir”) deve chamar o método `imprimir` da **MemoriaDeDados**.
  - A opção 0 da tela de Memória de Dado (“Voltar”) deve retornar à tela principal.
- A opção 2 (“Registradores”) deve ir para a tela de Registrador
  - 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.
- 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.

3.2 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 de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 1

Memoria de Dados
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 1

Posicao a ser alterada: 0
Novo valor: 3

Memoria de Dados
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 1

Posicao a ser alterada: 1
Novo valor: 4

Memoria de Dados
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 1
Posicao a ser alterada: 2
Novo valor: 4

Memoria de Dados
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 0

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
```

```
Escolha uma opcao: 3

PC: 0
Instrucao executada
PC: 1

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 3

PC: 1
Instrucao executada
PC: 2

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 3

PC: 2
Instrucao executada
PC: 3

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 2

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

0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 3
9: 4
10: 4
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 de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 3

PC: 3
Instrucao executada
PC: 4

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 3

PC: 4
Instrucao executada
PC: 5

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 3

PC: 5
Instrucao executada
PC: 6

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 3

PC: 6
Instrucao executada
PC: 0

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 1

Memoria de Dados
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 2

0: 3
```

```
1: 4
2: 4
3: 12
4: -
5: -
6: -
7: -
8: -
9: -
10: -
11: -
12: -
13: -
14: -
15: -

Memoria de Dados
1) Alterar valor
2) Imprimir
0) Voltar
Escolha uma opcao: 0

Emulador de MIPS
1) Memoria de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao: 2

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

0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 3
9: 4
10: 4
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: 12
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 de Dados
2) Registradores
3) Executar proxima instrucao
0) Sair
Escolha uma opcao:0
```

4 Entrega

O projeto deverá ser entregue até dia **16/10** em um Judge específico, disponível em <https://laboo.pcs.usp.br/ep/> (nos próximos dias vocês receberão um login e uma senha).

As duplas podem ser formadas por alunos de qualquer turma e elas devem ser informadas no e-Disciplinas até dia 30/09. Caso não seja informada a dupla, será considerado que o aluno está fazendo o EP sozinho. **Note que no EP2 deve-se manter a mesma dupla do EP1 (será apenas possível *desfazer* a dupla, mas não formar uma nova).**

> **Atenção:** não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e **todos** os alunos dos 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 3 problemas (Parte 1, Parte 2 e Parte 3). 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.

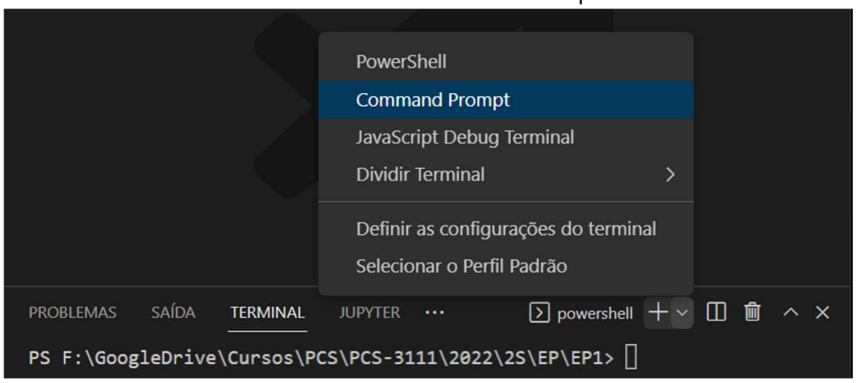
Você pode submeter quantas vezes quiser, sem desconto na nota.

5 Dicas

- Caso o programa esteja travando, 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.**