CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES II

Trabalho 3

Tomasulo

Caio Palhares Porto

Gustavo de Assis

Professora: Daniela Cassini

BELO HORIZONTE

2023

SUMÁRIO:

1. INTRODUÇÃO

Tomasulo é um método de escalonamento dinâmico, ou seja, o próprio hardware escalona o código de forma a diminuir os stalls. O princípio básico do funcionamento desse método é permitir a execução de instruções fora de ordem, mesmo que elas sejam emitidas em ordem.

Sendo assim, o processador deve verificar a presença de hazards estruturais e de dados na fase de emissão da instrução. Para isso, ele contará com buffers de memória auxiliares e estruturas chamadas de Estações de Reserva (RS).

As Estações de Reserva farão o controle de operandos, permitindo que a execução da instrução ocorra somente quando seus operandos de dados estiverem disponíveis. Ela também fará a renomeação de registradores, que por sua vez irá resolver os problemas de antidependência (WAR) e dependência de saída (WAW).

Esse método é dividido em três etapas: Despacho, execução e escrita de resultados.

O despacho é a etapa onde é obtida a instrução da fila e ela é destinada para uma Estação de Reserva. Sendo assim, nessa etapa que irá ocorrer o rastreamento de operandos (caso tenhamos o valor de um registrador, ele será gravado na RS, caso ainda não esteja disponível, será colocado a unidade funcional que irá gerá-lo) e a renomeação de registradores.

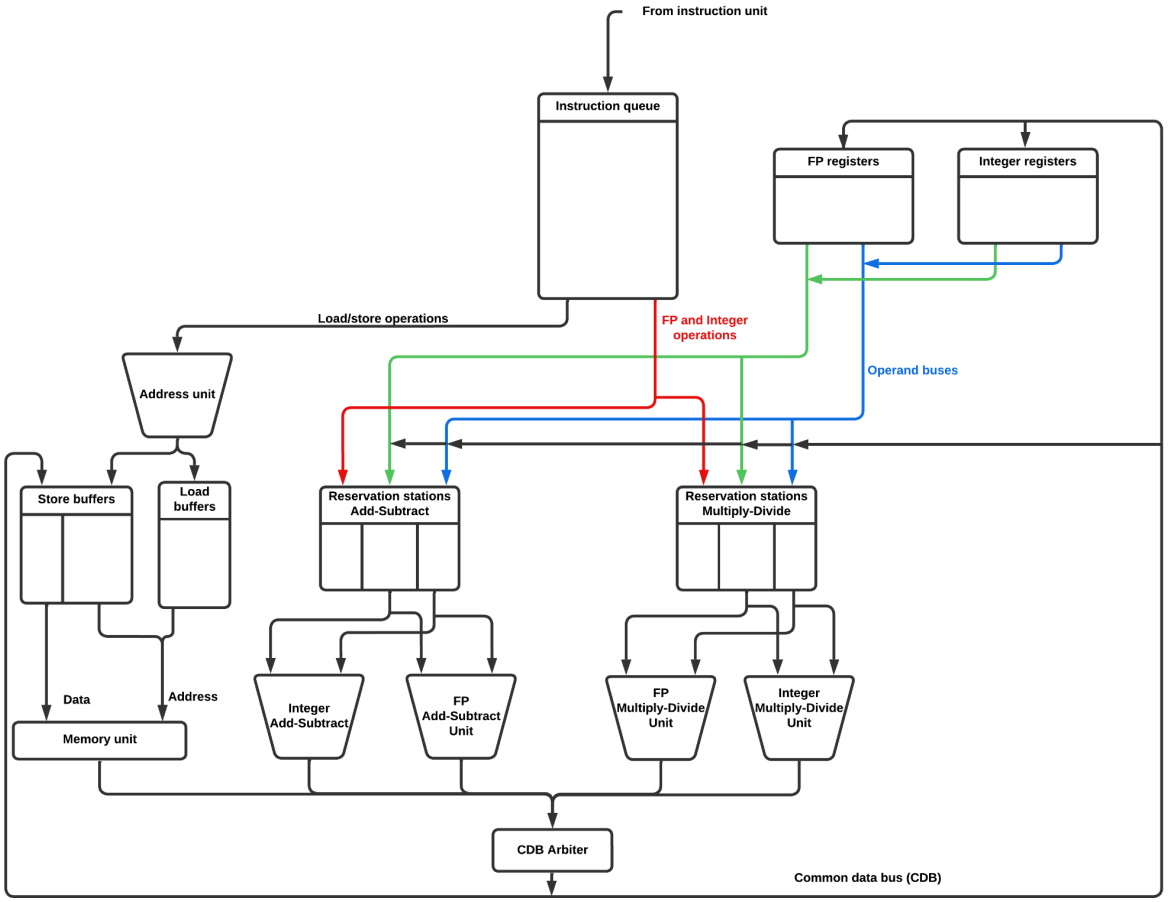
Execução: Quando os operandos estiverem disponíveis na RS, eles serão destinados para a unidade funcional executar a operação. Como as “instruções” ficam nas RS’s, a que estiver pronta no momento será executada, o que permite a execução fora de ordem (princípio do Tomasulo. No caso de mais de uma instrução ficar pronta para execução na mesma unidade funcional no mesmo ciclo, somente uma delas será executada naquele ciclo, e a outra será executada assim que possível.

Escrita de resultados: Ao terminar a execução e o resultado tiver sido gerado, o resultado é escrito no CDB (Common data bus) que é um barramento de uso comum. O CDB disponibilizará o resultado para o banco de registradores, as estações de reserva e para os buffers de memória.

Esse é o funcionamento básico do Tomasulo, sendo o projeto mais detalhado a seguir.

2. PROJETO

Seguindo o modelo apresentado na introdução. O projeto pode ser representado por:



NOTA: Irei fazer uma abstração de funcionamento a nível de hardware para se enquadrar ao processo. Exemplo: A unidade funcional de soma de inteiros no MIPS apresenta como comportamento: Recebe dois operandos inteiros das estações de reserva e realiza a soma ou a subtração deles. Se é uma soma ou uma subtração é definido pelos próprios números, uma vez que os negativos são representados por complemento de 2 para o processador e sendo a soma dada bit por bit. Mas para nosso caso, onde podemos representar com linguagem de programação ou texto, irei falar apenas que a unidade de soma recebe dois operandos e realiza a soma ou subtração conforme valores.

A descrição dos componentes considerando cada estágio é dado por (retiradas do livro texto com modificações para serem implementadas), para o entendimento geral do projeto será necessário ver o tópico de descrição, uma vez que nele está o código. Isso por que para definir estados, utilizaremos variáveis “globais”:

2.1. FILA DE INSTRUÇÕES

A fila de instruções é um buffer que contém as instruções. A organização da fila é FIFO (First In First Out). A fila de instruções verifica se há alguma entrada estação de reserva disponível para determinada instrução (caso seja add/sub, verifica se a RS de add/sub está livre), caso haja, ele envia a próxima instrução, caso não esteja, espera até estar.

2.2. ESTAÇÕES DE RESERVA

Estações de reserva são unidades que armazenam instruções. Com relação ao funcionamento: Em caso de desvios, todas as entradas da estação de reserva são apagadas. Como é um algoritmo sem especulação, nenhuma instrução pós desvio pode ser executada até que o resultado do desvio esteja concluído (**evitando Hazards de controle**).

A instrução enviada contém os valores dos operandos, se eles estiverem atualmente nos registradores, caso não estejam, RS acompanha as unidades funcionais que produzirão os operandos, assim que estiverem prontos, registra os valores e assim que possível executa a operação. **Isso evita RAW's**.

De forma complementar, as estações de reserva devem renomear os registradores de forma a **evitar dependência de dados verdadeira, anti-dependência e dependência de saída**

2.2.1. ESTAÇÕES DE RESERVA DE SOMA E SUBTRAÇÃO

A estação de reserva é um buffer que armazena instruções e contém os campos:

- Op: Especifica a operação que será realizada nos operandos fonte S1 e S2.
- Qj, Qk: Identificam as estações de reserva que fornecerão os operandos fonte correspondentes. Um valor de zero indica que o operando fonte já está disponível em Vj ou Vk, ou não é necessário.
- Vj, Vk: Contêm os valores dos operandos fonte. Nota-se que apenas um dos campos V ou Q é válido para cada operando. Para instruções de carregamento (loads), o campo Vk é usado para armazenar o campo de deslocamento (offset).
- Busy: Indica que essa estação de reserva e sua unidade funcional correspondente estão ocupadas.

2.2.2. ESTAÇÕES DE RESERVA DE MULTIPLICAÇÃO E DIVISÃO

Tem a mesma especificação das estações de reserva de soma e subtração.

2.2.3. ESTAÇÕES DE RESERVA DE MEMÓRIA (BUFFERS)

Similar as estações de reserva, mas possuem um campo adicional:

A**:** Usado para armazenar informações sobre o cálculo do endereço de memória para instruções de carregamento ou armazenamento (load/store). Inicialmente, o campo imediato da instrução é armazenado aqui; após o cálculo do endereço, o endereço efetivo é armazenado nesse campo.

2.2.3.1. LOAD BUFFER

Os Load buffers tem como função armazenar os componentes do endereço efetivo até que ele seja calculado, rastrear load's pendentes que estão aguardando a memória, e **armazenar os resultados de loads concluídos que estão aguardando o CDB.**

2.2.3.2. STORE BUFFER

Os Store buffers tem como função armazenar os componentes do endereço efetivo até que ele seja calculado, armazenar os endereços de memória de store's pendentes que estão aguardando o valor de dados a serem armazenados (uma vez que serão escritos quanto o valor estiver pronto), e **armazenar o endereço e o valor a serem armazenados até que a unidade de memória esteja disponível.** Da mesma forma que os RS's, se o dado não estiver pronto ele monitora

2.3. UNIDADES FUNCIONAIS

São as unidades que farão as operações de PF e inteiros. Para fins de implementação, elas carregarão como informação um campo Busy, que indica se está em uso ou não, permitindo identificar se a **unidade funcional está cheia ou não.**

2.3.1. SOMA/SUBTRAÇÃO DE INTEIROS

Recebe dois operandos inteiros das estações de reserva e realiza a soma ou a subtração conforme os valores dos mesmos.

2.3.2. MULTIPLICAÇÃO/DIVISÃO DE INTEIROS

Recebe dois operandos inteiros das estações de reserva e realiza a multiplicação ou a divisão conforme a instrução.

2.3.3. SOMA/SUBTRAÇÃO DE PONTO FLUTUANTE

Recebe dois operandos de tipo PF das estações de reserva e realiza a soma ou a subtração conforme os valores dos mesmos.

2.3.4. MULTIPLICAÇÃO/DIVISÃO DE PONTO FLUTUANTE

Recebe dois operandos de tipo PF das estações de reserva e realiza a multiplicação ou a divisão conforme a instrução.

2.3.5. CÁLCULO DE ENDEREÇO

Recebe dois valores, sendo eles o valor da posição desejada de memória e o deslocamento em bits que a mesma deve sofrer, dependendo do deslocamento, será realizada a soma ou subtração do descolamento em relação ao endereço.

2.4. BANCO DE REGISTRADORES

São um conjunto de registradores que armazenam dados, mas para fins de implementação, vamos considerá-lo um buffer. Nele teremos 3 campos:

- Cod: Nome do registrador. Pode ser F# ou R#. Considere # qualquer número.
- V: Valor armazenado em cada registrador.
- Qi: Indica o número da estação de reserva que contém a operação cujo resultado deve ser armazenado nesse registrador. Se o valor de Qi estiver em branco (ou for 0), significa que nenhuma instrução ativa atualmente está computando um resultado destinado a esse registrador, ou seja, o valor contido no registrador é o valor final.

2.5. CDB

O Common Data Bus (CDB) é um barramento que distribui os resultados das operações concluídas para todas as unidades funcionais e estações de reserva que aguardavam por esses valores. Quando um resultado é escrito no CDB, ele é disponibilizado para as instruções em espera, permitindo o despacho e execução das próximas instruções fora de ordem, o que melhora a eficiência e a latência associada à espera de valores. Há apenas um CDB nesse projeto e apenas uma unidade pode escrever nele por vez.

2.5.1. CDB ARBITER

É responsável por gerenciar o acesso ao Common Data Bus (CDB) quando múltiplas unidades tentam escrever seus resultados simultaneamente. Ele também realiza o broadcast do resultado para todas as unidades funcionais, garantindo que todos os componentes relevantes recebam o dado necessário para continuar a execução das instruções. A escolha de qual componente pode escrever no CDB pode ser arbitrária, porém iremos definir que: O que já estiver esperando a mais tempo tem prioridade, caso venham de forma simultânea, primeiro escreve Load's e Store's, depois resultados de Multiplicação e Divisão, depois a Soma e Subtração.

2.6. Memória

Apesar da memória não estar dentre os requisitos exigidos neste trabalho, para implementar o processador é necessário uma ideia básica de seu funcionamento. A memória será tratada como um mapa (HashMap) onde temos um endereço e uma informação relacionada a ele.

3. DESCRIÇÃO

Resolvemos implantar o código em java. Isso pois uma linguagem de programação por sí só já oferece um grande grau de detalhamento do funcionamento do projeto. Ademais, caso ocorra alguma situação não representável por código, iremos descrevê-la em texto.

Java é um linguagem com suporte ao POO, então colocaremos cada classe aqui, em texto, e também mandaremos um arquivo em anexo com o projeto.

Esse projeto não necessariamente irá ser executado, tendo em vista que nem toda lógica será implementada. Isso porque o funcionamento do processador é diferente do que podemos simular, pelo menos de forma simples. Inclusive, para complementar o entendimento geral, será criado um componente sistema (é só para efeitos de simulação) que controlará melhor o que acontece entre os componentes.

Por questão de simplicidade, começarei com os Unidades Funcionais:

- **Somador de inteiros:** Funcionamento simples, durante a execução mantém busy = 1, para não ocorrer dois chamados desse método ao mesmo tempo. (para a velocidade do computador isso não será possível de ser observado, porém a lógica está implementada), os operandos podem ser modificados através dos setters.

```
public class IntegerAdder {
    private int v1, v2;
    private boolean busy;

    //Caso queremos subtrair, enviamos um número negativo. Ex: 4 + (-2)
    public int somar(){
        busy = true;
        int soma = v1 + v2;
        busy = false;
        return soma;
    }

    public void setV1(int v1) {
        this.v1 = v1;
    }

    public void setV2(int v2) {
        this.v2 = v2;
    }

    public boolean isBusy() {
        return busy;
    }
}
```

- **Multiplicador de inteiros:** Lógica semelhante.

```
public class IntegerMultiplier {
    private int v1, v2;
    private boolean busy;

     //Caso queremos dividir, enviamos um número fracionário. Ex: 4 *
(0,5) = 2
    public int multiplicar(){
        busy = true;
        int produto = v1 + v2;
        busy = false;
        return produto;
    }

    public void setV1(int v1) {
        this.v1 = v1;
```

```
    }

    public void setV2(int v2) {
        this.v2 = v2;
    }

    public boolean isBusy() {
        return busy;
    }
}
```

- **Somador de PF:**

```
public class FPAdder {

    private float v1, v2;
    private boolean busy;

    //Caso queremos subtrair, enviamos um número negativo. Ex: 4 + (-2)
    public float somar() {
        busy = true;
        float soma = v1 + v2;
        busy = false;
        return soma;
    }

    public void setV1(float v1) {
        this.v1 = v1;
    }

    public void setV2(float v2) {
        this.v2 = v2;
    }

    public boolean isBusy() {
        return busy;
    }
}
```

- **Multiplicador de PF:**

```
public class FPMultiplier {
    private float v1, v2;
    private boolean busy;

      //Caso queremos dividir, enviamos um número fracionário. Ex: 4 *
(0,5) = 2
    public float multiplicar(){
        busy = true;
        float produto = v1 + v2;
        busy = false;
        return produto;
    }

    public void setV1(float v1) {
        this.v1 = v1;
    }

    public void setV2(float v2) {
        this.v2 = v2;
    }

    public boolean isBusy() {

        return busy;
    }
}
```

- **Somador de endereço:** No caso, onde o endereço ainda não estiver disponível para fazer o cálculo, ele ficará guardado no buffer de load/store até estar disponível.

```
public class AddressAdder {
    private int v1, v2;
       private boolean busy;

    //Caso queremos subtrair, enviamos um número negativo. Ex: 4 + (-2)
       public int somar(){
           busy = true;
           int soma = v1 + v2;
           busy = false;
```

```
            return soma;
        }

        public void setV1(int v1) {
            this.v1 = v1;
        }

        public void setV2(int v2) {
            this.v2 = v2;
        }

        public boolean isBusy() {
            return busy;
        }
```

- **Memória:** Map simples.

```
import java.util.HashMap;

public class MemoryUnit {

    //utilizo string pq hash n aceita tipos básicos
    private final HashMap<String, String> memoria;
    boolean busy;

    public MemoryUnit() {
        memoria = new HashMap<>();
        busy = false;
    }

    public int getFromMemory(int adress) {
        busy = true;
        String prov = memoria.get(Integer.toString(adress));
        busy = false;
        return Integer.valueOf(prov);

    }

    public void setMemory(int adress, int value) {
        busy = true;
        memoria.put(Integer.toString(adress), Integer.toString(value));
```

```
        busy = false;
    }
}
```

- **Banco de registradores:** Para fins de implementação, vamos considerar o banco de registradores só um conjunto de registradores e cada registrador carrega suas informações. (mais parecido com a realidade), com isso temos duas classes. Lembrando que Qi indica o número da estação de reserva que guarda o resultado (se ele ainda não estiver disponível). **Apesar de representar no diagrama como dois bancos separados, eles são um só.**

```
//Classe que representa um registrador de inteiro.
public class RegisterInteiro {
    private String nome; //nome é melhor que code para lembrar
    private int valor;
    private int qi;

    public RegisterInteiro(String nome){
    this.nome = nome;
    }

    public String getNome(){
        return nome;
    }

    public int getValor() {
        return valor;
    }

    public void setValor(int valor) {
        this.valor = valor;
    }

    public int getQi() {
        return qi;
    }

    public void setQi(int qi) {
        this.qi = qi;
    }
}
```

```
//Classe do reg de ponto flutuante
//mesma formato que  a de inteiros


//Classe do Banco de Registradores:
public class BancoRegistrador {

    private RegisterInteiro[] registradoresInteiros;
    private RegisterPF[] registradoresPF;

    public BancoRegistrador() {
        //10 REGS DE CADA
        registradoresInteiros = new RegisterInteiro[10];
        registradoresPF = new RegisterPF[10];
        //dá o nome para cada um
        for (int i = 0; i < 20; i++) {
            registradoresInteiros[i] = new RegisterInteiro("R" + i);
            registradoresPF[i] = new RegisterPF("F" + i);
        }
    }

    //NOTA: NÃO CHAMO NENHUM REG DE 0
    public RegisterInteiro getRegInteiro(String nomeReg) {
        for (int i = 1; i <= registradoresInteiros.length; i++) {
            if (registradoresInteiros[i].getNome() == nomeReg) {
                return registradoresInteiros[i];
            }
        }
        return null;
    }

    //REPITO PARA PF
    public RegisterPF getRegPF(String nomeReg) {
        for (int i = 1; i <= registradoresPF.length; i++) {
            if (registradoresPF[i].getNome() == nomeReg) {
                return registradoresPF[i];
            }
        }
        return null;
    }

    public void setarQidoRegInteiro(String nomeReg, int qi){
```

```
        for (int i = 1; i <= registradoresInteiros.length; i++) {
            if (registradoresInteiros[i].getNome() == nomeReg) {
               registradoresInteiros[i].setQi(qi);
            }
        }
    }
}
```

- **Instrução:** Para fins de implementação, para compreender melhor os próximos passos, é necessário definir uma instrução como um objeto, onde sua classe é dada por:

```
package com.mycompany.tomasulo;

public class Instrucao {
    private String tipo;    //LD, SD, ADD...
    //As instruçoes "normais" tem até 3 regs
    private String reg1;
    private String reg2;
    private String reg3;
    //elas podem ter valores tbm, como o caso de addi ou do
    //deslocamento em loads e storys
    private int valor;

      public Instrucao(String tipo, String reg1, String reg2, String
reg3, int valor) {
        this.tipo = tipo;
        this.reg1 = reg1;
        this.reg2 = reg2;
        this.reg3 = reg3;
        this.valor = valor;
    }
}
```

- **Estação de reserva do somador:** É um buffer na qual os campos estão listados na parte de Projeto. Vamos dividir sua implementação em duas classes, uma vai representar cada entrada do buffer, e uma uma buffer.

- **Instrução de desvio:** será realizado o cálculo com os registradores para o desvio, espera-se o resultado das estações de reserva que também dependem do resultado. Em caso do desvio ser tomado, as estações de reserva e a fila de instruções serão limpas e um sinal será enviado à Memória de Instruções para que altere o PC (Program Counter) para o novo endereço.

```
package com.mycompany.tomasulo;
```

```
public class EstacaoReserva {

        // Instrução de desvio: será realizado o cálculo com os
registradores para o desvio, espera-se o resultado das estações de
reserva que também dependem do resultado.
    // Em caso do desvio ser tomado, as estações de reserva e a fila de
instruções serão limpas e um sinal será enviado à Memória de Instruções
para que altere o PC (Program Counter)
    // para o novo endereço.

    private EntryRSAdder[] entradas;
    private boolean busy;
    private BancoRegistrador bancoRegistradores;
    private IntegerAdder somador;

          public  EstacaoReserva(BancoRegistrador  bancoRegistradores,
IntegerAdder somador) {
        //vamo supor que temos 10 entradas na RS
        entradas = new EntryRSAdder[10];
        this.inicializarEntradas();
        this.bancoRegistradores = bancoRegistradores;
        this.somador = somador;
    }

    //NOTA: N TEM ENTRADA COM NUMERO 0
    private void inicializarEntradas() {
        for (int i = 1; i <= entradas.length; i++) {
            entradas[i] = new EntryRSAdder();
        }
    }

    public boolean isBusy() {
        return busy;
    }

    public void inserirEntrada(Instrucao instrucao) {
        for (int i = 0; i < 20; i++) {
            if (!entradas[i].isBusy()) {
                this.separarEInsserir(instrucao, i);
                break;
            }
        }
```

```
    }

    private void separarEInsserir(Instrucao instrucao, int i) {
        //entry é formada pela operação, e os valores se tiver.

        //primeiro pego qual o tipo de opecacao
        String operacao = instrucao.getTipo();

        //aqui faço a lógica para pesquisar nos reg inteiros ou pf's
        //para simplificar, vou só separar operacao de inteiros e de pf
        if (operacao == "INTEIROS") {
            entradas[i].setOperacao(operacao);
            //pega os registradores
                                                RegisterInteiro    reg2    =
bancoRegistradores.getRegInteiro(instrucao.getReg2());
                                                RegisterInteiro    reg3    =
bancoRegistradores.getRegInteiro(instrucao.getReg3());

             //olho se esse reg está esperando um dado de outra estação
de reserva
             //caso esteja, não posso executar a operação por que daria
dependencia de dados
            //verdadeira, mando ele para Qi ou para Qj dependendo.
            if (reg2.getQi() != 0) {
                //significa que depende de alguma outra
                entradas[i].setQj(Integer.toString(reg2.getQi()));
            } else {
                //significa que o valor já está pronto
                entradas[i].setVj(Integer.toString(reg2.getValor()));
            }

            if (reg3.getQi() != 0) {
                //significa que depende de alguma outra
                entradas[i].setQk(Integer.toString(reg3.getQi()));
            } else {
                //significa que o valor já está pronto
                entradas[i].setVk(Integer.toString(reg3.getValor()));
            }

            entradas[i].setBusy(true);
```

```
            //AGORA DEFINO O REGISTRADOR QUE VAI RECEBER A RESPOSTA DA
OPERACAO
            //DE FORMA QUE EVITA DEPENDECIAS DE NOME E DE SAIDA:

            //PRIMEIRO: OLHO SE ELE JÁ VAI RECEBER OUTRA OPERAÇÃO
              //caso ele tenha o qi diferente de 0, eu tenho que mudar
esse reg
             if(bancoRegistradores.getRegInteiro(instrucao.getReg1()).qi
!= 0);
                //NÃO VOU IMPLEMENTAR EM CÓDIGO, MAS AQ EU PROCURO O
PRIMEIRO REG
               //QUE N ESTÁ ESPERANDO RESULTADO DE OUTRO, E QUE N ESTÁ
SENDO USADO EM OUTRA
            //RS E TROCO O NOME DO REG ATUAL PELO DELE

             //AGORA SETO QUE ELE DEVE RECEBER O RESULTADO DA OPERAÇÃO A
SER REALIZADO
            //POR ESSA ENTRADA DE RS
             bancoRegistradores.setarQidoRegInteiro(instrucao.getReg1(),
i);

        }

        //REPITO O PROCESSO PARA OS PF
        if (operacao == "PF") {
            entradas[i].setOperacao(operacao);
            //pega os registradores
                                                RegisterPF    reg2    =
bancoRegistradores.getRegPF(instrucao.getReg2());
                                                RegisterPF    reg3    =
bancoRegistradores.getRegPF(instrucao.getReg3());

             //olho se esse reg está esperando um dado de outra estação
de reserva
             //caso esteja, não posso executar a operação por que daria
dependencia de dados
            //verdadeira, mando ele para Qi ou para Qj dependendo.
            if (reg2.getQi() != 0) {
                //significa que depende de alguma outra
                entradas[i].setQj(Float.toString(reg2.getQi()));
            } else {
                //significa que o valor já está pronto
                entradas[i].setVj(Float.toString(reg2.getValor()));
```

```
            }

            if (reg3.getQi() != 0) {
                //significa que depende de alguma outra
                entradas[i].setQk(Float.toString(reg3.getQi()));
            } else {
                //significa que o valor já está pronto
                entradas[i].setVk(Float.toString(reg3.getValor()));
            }

            entradas[i].setBusy(true);
        }

    }

    //A logica de mandar executar é feita por um externa
    public int qualEntradaEstaProntaParaExecutar() {
        for (int i = 1; i <= entradas.length; i++) {
            //pego as entradas que estão ocupadas:
            if (entradas[i].isBusy()) {
                  //vejo se o valor Vk e VI estão prontos, no caso isso
evita dependencias reais de dados
                 if (entradas[i].getQj() == "0" && entradas[i].getQk()
== "0") {
                    return i;
                }
            }
        }
        return 0;
    }
}
```

- **Estação de reserva do multiplicador:** Mesma ideia da estação do somador, só irá mudar a operação destino.

- O Load inicia quando uma instrução de carga (load) é enviada para o Load Buffer. Se um registrador envolvido na operação já possui um valor pronto, este valor é armazenado na variável A da entrada correspondente no buffer. Caso contrário, o Qi da entrada é preenchido com a estação de reserva da qual o valor depende, sinalizando a dependência de dados. Uma vez que a dependência é resolvida e o valor está disponível, o registrador de destino é atualizado para indicar que está esperando o resultado dessa operação de load. O valor é então encaminhado para a Unidade de Endereços que calcula o endereço da memória para carregar o dado, finalizando o ciclo.

- Store é processado pelo Store Buffer. Quando uma instrução de armazenamento (store) é enviada, o endereço de memória onde o dado será armazenado é calculado e armazenado na variável A da entrada correspondente no buffer. Se o valor a ser armazenado no endereço ainda não estiver disponível, a variável Qi é preenchida com a estação de reserva que fornecerá o dado, indicando a dependência. Assim que o valor estiver pronto, ele é transferido para o Store Buffer. Em seguida, o valor e o endereço são enviados para a Unidade de Memória, onde o dado é finalmente armazenado na memória no local especificado.

- **Buffer de load's:** Funcionamento bem parecido com as RS's. Inclusive vou usar o mesmo formato praticamente. Tratatei Qj e Qk da mesma forma (LD Qj, index(Qk)). Essa pode conter erros Cassini, fiz bem básica mesmo 🙂.

```
public class EntryBufferLoad {
    private boolean busy;
    private int qi;
    private int A; //endereço

    public EntryBufferLoad (){
        busy = false;
        qi = 0;
        A=0;
    }

    public boolean isBusy() {
        return busy;
    }

    public void setBusy(boolean busy) {
        this.busy = busy;
    }

    public int getQi() {
        return qi;
    }

    public void setQi(int qi) {
        this.qi = qi;
    }

    public int getA() {
        return A;
    }

    public void setA(int A) {
```

```
        this.A = A;
    }


}


public class BufferLoad {

    private EntryBufferLoad[] entradas;
    private boolean busy;
    private BancoRegistrador bancoRegistradores;
    private IntegerAdder somador;

     public BufferLoad(BancoRegistrador bancoRegistradores, IntegerAdder
somador) {
        //vamo supor que temos 10 entradas na RS
        entradas = new EntryBufferLoad[10];
        this.inicializarEntradas();
        this.bancoRegistradores = bancoRegistradores;
        this.somador = somador;
    }

    //NOTA: N TEM ENTRADA COM NUMERO 0
    private void inicializarEntradas() {
        for (int i = 1; i <= entradas.length; i++) {
            entradas[i] = new EntryBufferLoad();
        }
    }

    public boolean isBusy() {
        return busy;
    }

    public void inserirEntrada(Instrucao instrucao) {
        for (int i = 0; i < 20; i++) {
            if (!entradas[i].isBusy()) {
                this.separarEInsserir(instrucao, i);
                break;
            }
        }

    }
```

```
    private void separarEInsserir(Instrucao instrucao, int i) {

        //pega os registrador
                                            RegisterInteiro     reg2     =
bancoRegistradores.getRegInteiro(instrucao.getReg2());

          //olho se esse reg está esperando um dado de outra estação de
reserva
           //caso esteja, não posso executar a operação por que daria
dependencia de dados
        //verdadeira, mando ele para Qi ou para Qj dependendo.
        if (reg2.getQi() != 0) {
            //significa que depende de alguma outra
            entradas[i].setQi(reg2.getQi());
        } else {
            //significa que o valor já está pronto
            entradas[i].setA(reg2.getValor());
        }

        entradas[i].setBusy(true);

           //AGORA DEFINO O REGISTRADOR QUE VAI RECEBER A RESPOSTA DA
OPERACAO
        //DE FORMA QUE EVITA DEPENDECIAS DE NOME E DE SAIDA:
        //PRIMEIRO: OLHO SE ELE JÁ VAI RECEBER OUTRA OPERAÇÃO
          //caso ele tenha o qi diferente de 0, eu tenho que mudar esse
reg
         if (bancoRegistradores.getRegInteiro(instrucao.getReg1()).qi !=
0);
          //NÃO VOU IMPLEMENTAR EM CÓDIGO, MAS AQ EU PROCURO O PRIMEIRO
REG
          //QUE N ESTÁ ESPERANDO RESULTADO DE OUTRO, E QUE N ESTÁ SENDO
USADO EM OUTRA
        //RS E TROCO O NOME DO REG ATUAL PELO DELE

         //AGORA SETO QUE ELE DEVE RECEBER O RESULTADO DA OPERAÇÃO A SER
REALIZADO
        //POR ESSA ENTRADA DE RS
        bancoRegistradores.setarQidoRegInteiro(instrucao.getReg1(), i);

    }
```

```
    //A logica de mandar executar é feita por um externa
    public int qualEntradaEstaProntaParaExecutar() {
        //mesma lógica de procurar qual está pronta.
        return 0; //aqui retornaria o resultado certo no caso
    }
}
```

- **Buffer de Store's:** Bem parecido com o Buffer de Load's, tem suas diferenças mas não acho necessário representá-lo.

- **Fila de instruções:** Conjunto de instruções, caso, a estação de reserva esteja vazia, envia a instrução para lá;

```
import java.util.ArrayList;

public class FilaInstrucoes {

    //uma fila de instruções comum com organização FIFO
    ArrayList<Instrucao> instrucoes;
    EstacaoReserva rs;

    public FilaInstrucoes(EstacaoReserva rs) {
        instrucoes = new ArrayList<>();
        this.rs = rs;
    }

    public void adicionarInstrucao(Instrucao instrucao) {
        instrucoes.add(instrucao);
    }

    //CASO SEJA OPERAÇÕES:
    //IMAGINE UM IF AQ
    public void mandarParaRS() {
        if (!instrucoes.isEmpty()) {
            if (!rs.isBusy()) {
                rs.inserirEntrada(instrucoes.get(0));
                //removo ela do "topo"
                instrucoes.remove(0);
            }
        }
    }

    //CASO SEJA UM LOAD OU STORE MANDA PARA O BUFFER DELES
}
```

- **CDB Arbiter:** Explicação em texto: Esse irei explicar por que tem um funcionamento relativamente simples, porém chato de implementar. Ele vai ter um busy assim como os demais, que vai informar os outros componentes que não poderá ser escrito nele no momento. Então as informações dos demais que não podem ser escritas ficaram paradas, ou seja, a estação de reserva não poderia mandar executar a próxima instrução por que na unidade funcional busy estaria = 1 (na verdade não estaria, por que ela já retorna assim que possível, mas, isso será melhor regulado pela componente **sistema**). Ele faz uma escolha arbitrária, mas como não há "arbitrário" em um computador, vamos tratá-lo como um buffer. Ele registra um de cada vez, sentando o busy = 1 enquanto passa os dados para o CDB e busy = 0 quando termina. Ele guarda as informações de quem tentou escrever no CBD a mais tempo e dá preferência para ele. Caso seja simultâneas as chamadas de escrita no CDB, ele dá as preferẽncias descritas no tópico de projeto.

- **Sistema:**
    - **CDB:** CDB é um barramento de dados. Considerando nosso contexto, ele seria bem parecido com a classe Sistema criada. Isso porque a classe Sistema, é que vai receber o retorno de cada um dos componentes. A diferença é que o sistema também fará a chamada desses métodos. Em outras palavras, irá regular todas as informações exteriores aos componentes. No caso, ele vai chamar o método da fila de instruções de mandar instruções, depois vai chamar o da estação de reserva que fala qual entrada está livre, depois vai chamar a unidade funcional com a entrada da RS, vai pegar o resultado e jogar no CDB arbiter, e depois vai jogar no "CDB" que na verdade é só disponibilizar através dos setters o resultado da operação para todos aqueles que estão esperando. Da mesma forma, ele pega os dados do buffer de load ou store, faz o cálculo do endereço se a unidade estiver livre, pega o resultado da memória se ela estiver livre, joga no CDB Arbiter e por aí vai.