



UNIVERSIDADE FEDERAL DO CEARÁ – UFC

CAMPUS DE SOBRAL

CURSO DE ENGENHARIA DE COMPUTAÇÃO

TURMA 1

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Professor: Danilo Alves

PIPELINING

JOÃO GABRIEL FERNANDES GOMES – 418270
ANTONIO ERALDO CAETANO MARTINS – 415221
FRANCISCO RENATO GRACIANO FREIRE – 428131

**Sobral – CE
2019.2**

INTRODUÇÃO

Durante nossa história sempre buscamos maneiras de calcular, no começo as máquinas de calcular eram usadas para obter somas e subtrações simples, como o total de impostos de uma cidade ou loja, mas essas máquinas foram evoluindo, tal qual, a quantidade de coisas necessárias de se calcular diariamente. Já não se era mais eficiente fazer longos e repetidos cálculos a mão ou buscar milhares de papéis para buscar informações sobre uma determinada pessoa ou objeto, era necessário uma maneira mais rápida de se computar tudo isso.

A ciência foi evoluindo ao passar dos anos, nos permitindo buscar tecnologias novas, portanto, existia uma janela para que se pesquisasse sobre maneiras de melhorar nossa calculadora. Sob esse viés, temos que destacar nossa longa jornada até as máquinas de computação atual, partimos de computadores mecânicos para os de válvulas, computadores grandes e pesados, que tinham o tamanho de uma sala, o que não era nada prático, e uma peça chamada de transistor, inventada anos depois por John Bardeen, Walter Brattain e William Shockley, ganhadores do nobel de física do ano, que permitiu que essas milhares de válvulas pudessem ser condensadas numa pequena peça semicondutora que desempenhava o papel dessas milhares de válvulas, desse momento em diante fomos reduzindo mais e mais o tamanho de peças e aumentando mais e mais a quantidade de circuitos que podíamos alocar num computador.

A priori, essa evolução gigante que tivemos após a invenção dos transistores e as integrações em larga escala dos circuitos foram nos proporcionando máquinas melhores, e quando chegamos num certo ponto fez-se necessário discutir a arquitetura das máquinas, portanto, foi importante filosofar e debater sobre a seguinte pergunta: “como tornar os computadores mais rápidos?”. Sob essa ótica temos que durante o final da década de 1970, após anos de experimentações com as capacidades de interpretações de máquina, o microprograma e o nível ISA passamos a pensar em como fechar a “lacuna semântica” entre a linguagem de alto nível e baixo nível.

A posteriori, agora os projetistas passaram a pensar em como tornar as máquinas mais simples em prol de se maximizar a taxa de instruções, uma forma de simplificar as instruções para que os computadores pudessem entender e processá-la mais rapidamente, e finalmente, em 1980, um grupo em Berkeley, liderado por David Patterson e Carlo Sequin construíram uma arquitetura de computador que ficou conhecida como **RISC** (Reduced Instruction Set Computer), que permitiu inúmeras vantagens, dentre elas:

1. Era necessária apenas um clock para que se executasse as instruções, além disso elas não precisavam de microprograma para interpretá-la.
2. Grandes números de registradores que agilizava o acesso e manipulação da memória.
3. E principalmente o paralelismo de instruções.

O paralelismo de instruções foi talvez a melhor forma que engenheiros e estudiosos encontraram para amenizar o gargalo na velocidade do acesso à memória, ela permitiu que várias instruções pudessem ser executadas mais rápido e permitia a busca antecipada das mesmas, além disso, essa maneira de se organizar as instruções ficou muito popular devido a sua eficiência, e é usada atualmente. O paralelismo dividia uma atividade em estágios, permitindo a execução de várias tarefas ao mesmo tempo, tendo uma parte de hardware e registradores dedicados para cada uma de suas fases, são elas:

1. O estágio de busca (Fetch), no qual ele busca a instrução na memória e a coloca num buffer até que ela seja utilizada;

2. O segundo estágio está relacionado a decodificação do problema, ele é o processo de compreensão do computador para que ele entenda o que lhe foi enviado, logo é neste processo de “análise” que ocorre a determinação de que tipo de instrução aquilo é e de quais operandos ela necessita;
3. No estágio 3, ocorre a busca dos operandos nos registradores ou na memória, dependendo do que o estágio 2 necessite;
4. No estágio 4 ocorre a execução da instrução;
5. E o estágio 5 faz com que a execução da instrução seja gravada no registrador adequado;

Essa forma de se organizar o processo de buscar-decodificar-executar ficou tão popular que passou a existir novas maneiras de se organizar a pipeling, com métodos ainda mais rápidos do que seu modelo básico, mas mesmo com o modelo padrão de pipeling ainda é possível notar ganhos massivos no tempo de execução de várias instruções.

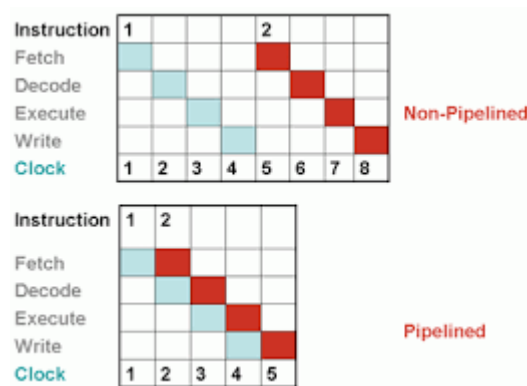


Figura 01:Tempo ganho pela Pipeling

Podemos notar que graças a pipeling, mostrada na figura 01, temos um ganho de 3 estágios de tempo para que se execute duas instruções, um ganho de 60% para ser mais preciso, e em computação, cada segundo que você puder otimizar é contabilizado no fim de um processo de milhões de instruções.

Como mencionado brevemente antes existe mais que um tipo de pipeling, e agora nós iremos abordar a pipelining que usa da ideia “quanto mais melhor”, a pipeline do tipo superescalar, que divide um estágio específico em vários miniprocessos, esse estágio costuma ser o quarto devido a ele necessitar de mais tempo que os outros e a dupla, que usa da ideia padrão da pipeline, e a duplica, contendo um pipeline principal, denominado de **u** e outro secundário, denominado de **v**, que executava apenas instruções com inteiros. Seguindo essa perspectiva vale ressaltar que essas pipelines necessitam que suas instruções não dependam uma da outra, caso contrário haverá muitos erros durante os estágios de execução das duas pipelines, logo sua principal limitação é a dependência de dados.

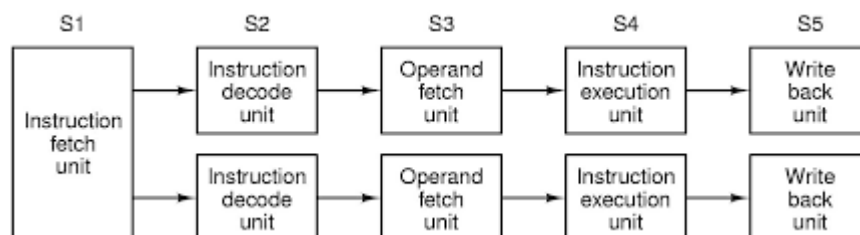


Figura 02 : Pipeline dupla

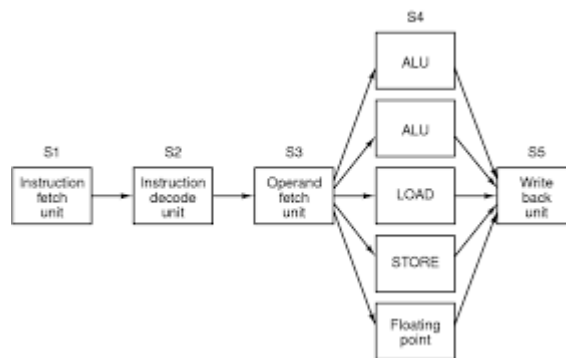


Figura 03: Pipeline superescalar

Ademais, temos ainda outro tipo de pipeline, que surgiu como forma de combater o calor dissipado pelos processadores durante a busca de processadores cada vez melhores. Receberam o nome de paralelismo a nível de processador e são divididos entre computadores matriciais, multiprocessadores e multicomputadores, esse tipo de pipelining foi concebida graças a capacidade que temos de colocar várias cpus dentro de um computador, o que resulta num tipo de pipelining cerca de 10 vezes mais rápidas que as previamente debatidas.

O primeiro é o computador matricial que consiste um grande número de processadores idênticos, o que eleva o custo desse tipo de pipeline, eles efetuam a mesma sequência de instruções em diferentes conjuntos de dados, esses processadores são organizados em quatro quadrantes, totalizando uma matriz de 8x8(Figura 04) e possuem apenas 1 unidade de controle.

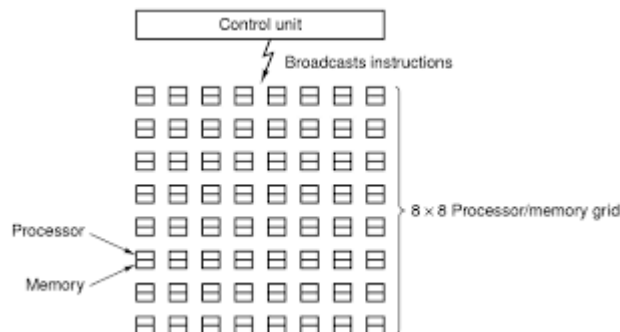


Figura 05 : Processador matricial

Essa tipo de pipeline ainda não é usado para os computadores comuns, seu custo elevado dificulta sua comercialização, seu custo é tão caro que o projeto do primeiro computador matricial foi reduzido para $\frac{1}{4}$ da matriz original, ficando apenas com um quadrante de dois por dois, mesmo assim ela tem o status de pipeline mais eficiente e caso sua meta original tivesse sido atingida a capacidade de computação teria duplicado naquela época.

Na pipelining chamada de multiprocessadores, uma ideia similar a de computadores matriciais é empregada, porém, temos que nessa pipelining as cpus são independentes, visto que cada uma tem sua unidade de controle independente uma da outra e eles compartilham de uma memória única, portanto, ao mesmo tempo que é fácil de se trabalhar e programar com uma memória só, fica difícil a questão do conflito de dados, já que vários processadores tentando acessar a uma memória ao mesmo tempo com certeza irá causar o conflito. Uma alternativa para superar a vicissitude foi dar uma memória local a cada processador, o que reduziria a necessidade de ir várias vezes a memória compartilhada, diminuindo o conflito de

dados.

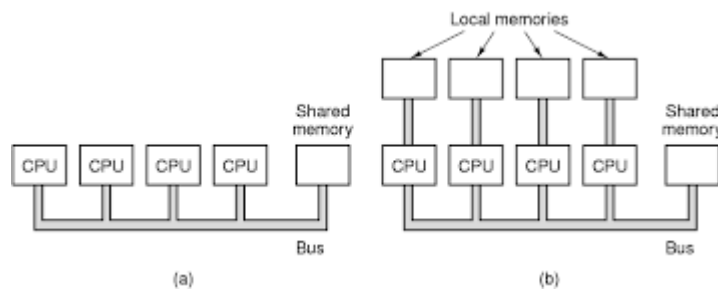


Figura 06: Computador com multiprocessador

Por fim, os Multicomputadores empregam as filosofias das pipelinings de processadores passadas e a junta num sistema de redes de computadores, ou seja, elas são um sistema de um grande número de computadores interconectados que enviam mensagens uns aos outros em prol de se executarem tarefas mais rapidamente, com cada computador possuindo sua memória, Unidade de controle e todos os outros componentes de um computador.

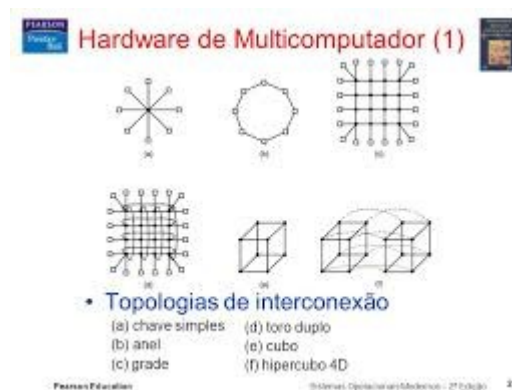


Figura 07: Formas de se organizar uma rede de computadores

Cada uma dessas formas contém suas vantagens e desvantagens e são formas válidas de organização de multicomputadores, porém, por estarmos focando na pipeling a nível de processador não é necessário que se debata cada uma individualmente.

DESENVOLVIMENTO

Após introduzirmos todos os conceitos de Pipeling, foi necessário verificar seu funcionamento na prática, utilizando de uma das funções oferecidas pela linguagem JAVA, as threads, que é a tarefa que um determinado programa realiza. Fio de execução, também conhecido como linha ou encadeamento de execução, é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente. Tendo isso em vista, essa é a forma ideal de se ver o funcionamento da Pipelining “ao vivo”.

Um breve resumo das variáveis inteiras estáticas que iremos usar:

- aux: Recebe e a quantidade de instruções a serem executadas é consultada para a execução de saídas e é decrementada ao acionar de funções.
- aux2: Responsável por clonar aux no início do programa e de proteger a

quantidade de instruções totais solicitadas pelo usuário, consultada em casos que são solicitadas até duas instruções, não seria viável com aux, pois esta sofre alterações ao longo do decorrer do programa.

- cont: Contador, um trunfo de consultas, essencial nestas para a execução de funções no momento certo, além de saídas de sistema.
- cont_a: Classificado como um contador auxiliar, é consultado para o programa chamar operandos.
- inst2: Clona inst e é usada na saída das threads por ser global e estática.

As variáveis locais:

- int inst: Variável que carregará o valor das instruções trabalhadas, podendo ser incrementada ou decrementada dependendo da situação.
- int n: Variável que pega a quantidade de instruções solicitadas.

Primeiro, precisamos importar as funções necessárias para nosso código, logo após criamos uma public class Arquitetura, para que pudéssemos englobar todo o código.

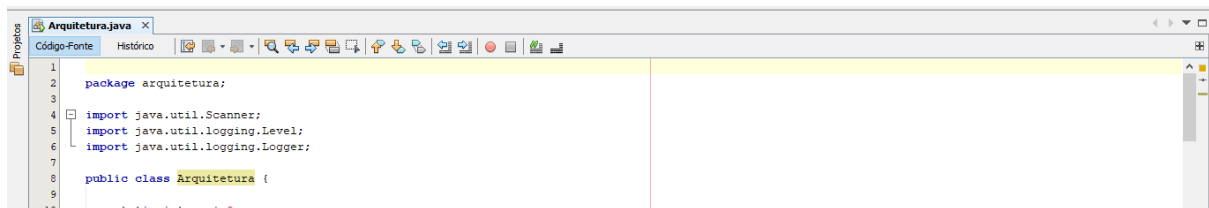


Figura 08: Importações usadas e classe Arquitetura

Depois declaramos as variáveis estáticas genéricas que auxiliarão condições de testes dentro de funções e métodos.

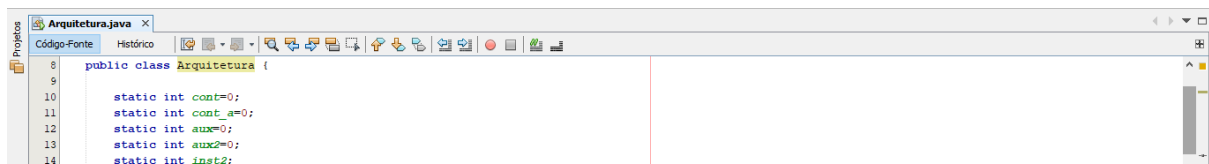


Figura 10: Variáveis estáticas

Para a próxima parte do código temos a primeira função dele, a Ativar1, a única acionada na main, recebe e quantidade total de instruções que serão trabalhadas, nela é clonado este valor para as variáveis aux e aux2, os dois contadores são iniciados, inst passa a ser 1 para trabalharmos de forma crescente e assim quando inst chegar ao valor de aux2, ainda na primeira é chamado a Thread para buscar pela primeira vez, o estágio se encerra e ela chamar a função Ativa2, quando o contador for diferente de 0, toda vez que a função for chamada, ela aumentará o número de instruções, podendo assim exibir que vai buscar a próxima instrução, terminando cada estágio e por meio do contador auxiliar verifica se está na hora de acionar operandos. A função ChamarT1 é a função responsável por ativar a primeira thread (buscando), os “IFs” e outras condições de teste, nos dizem a respeito do que é necessário chamar na função, no caso, a primeira execução executa um estágio da thread e logo após irá buscar pelo segundo estágio da thread (decodificando), tendo isso em vista, o código funciona como uma série de testes, enquanto houver a necessidade que uma função continue executando ela executará até que se pare, além disso, a chamada da função “ChamarT1” representa o momento em que o processo(run) da thread começa.

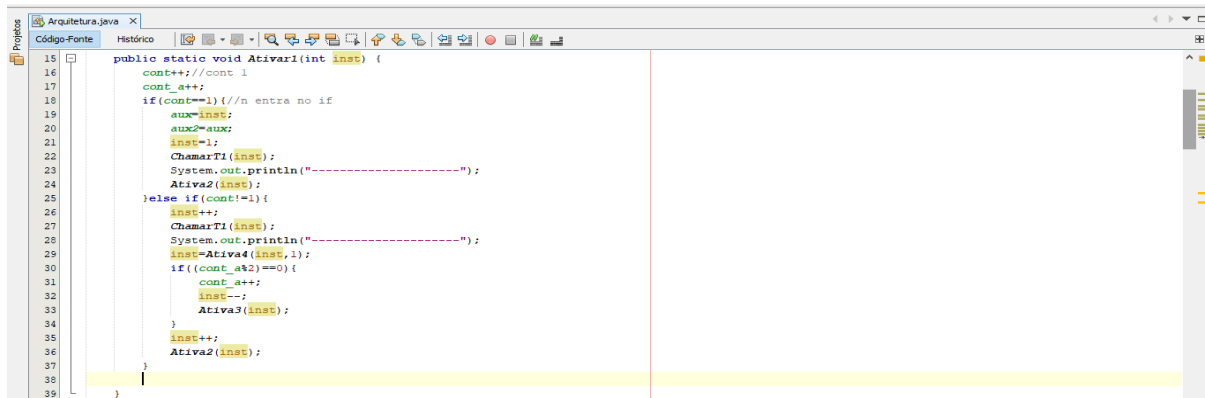


Figura 11: Função de ativação do primeiro estágio de thread

Para a função Ativar2, ela também recebe um inteiro como argumento, usa deste para passar o valor da instrução executada para inst2, a função é voltada para chamar a decodificação, verifica se tem mais de uma instrução a ser executada, se sim, decrementa aux, verifica se o contador é 1, altera ele se necessário, chama a Thread para decodificar e envia tudo novamente para o Ativar1(int inst) e termina o estágio 2, Quando é acionada pela 2ª vez ou posterior a esta, aumenta o contador e decodifica, se for somente uma instrução, ela decodifica e chama Ativa4(int inst, int tipo) e Ativa3(int inst).

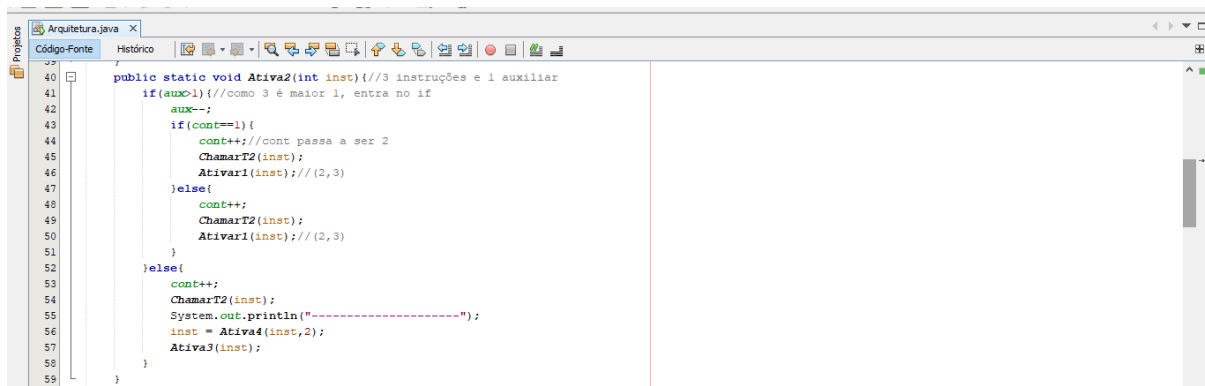


Figura 12: Função de ativação do segundo estágio de thread

Já a função Ativa3 é mais voltada para os operandos, possui consultas relacionadas a aux e aux2, cujo, se forem maiores que 1, busca os operandos, senão, verifica se aux2 é 1 ou 2 com inst igual a 2 para evitar decréscimos desnecessários e puder efetuar o término correto do estágio.

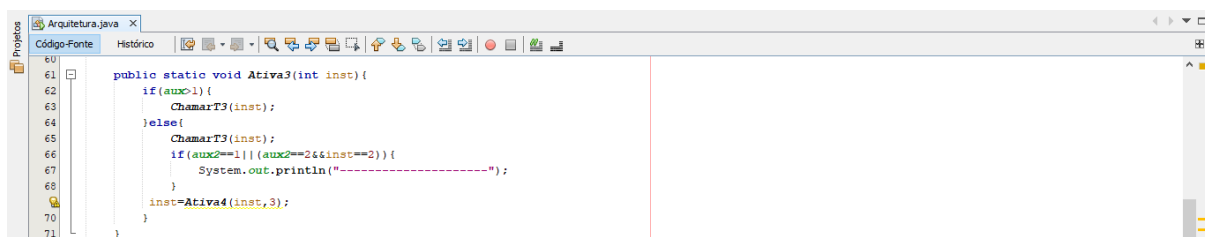
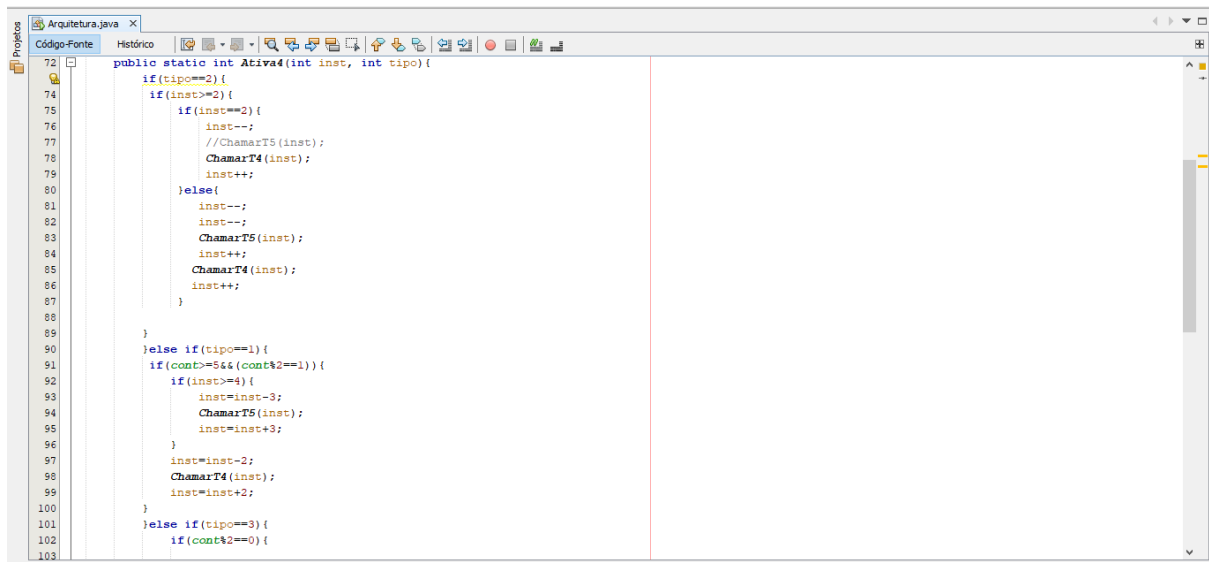


Figura 13: Função dos operandos e terceiro estágio da thread

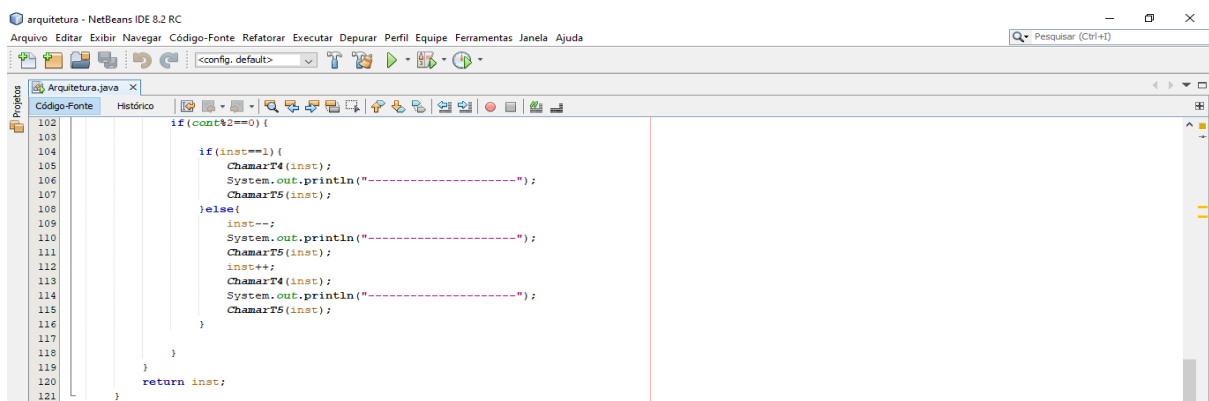
Por último a função Ativa4 é a única a possuir dois inteiros como argumento, com a maior quantidade de condições pois é chamada por 3 outras funções, quando atende o tipo 1 é porque foi chamada pela função Ativar1(int inst), quando ao tipo 2(int inst) é porque foi

chamada por 2 e quando ao tipo 3(int inst) é porque foi chamada por 3.



```
72 public static int Ativa4(int inst, int tipo){
73     if(tipo==2){
74         if(inst>=2){
75             if(inst==2){
76                 inst--;
77                 //ChamarT5(inst);
78                 ChamarT4(inst);
79                 inst++;
80             }else{
81                 inst--;
82                 inst--;
83                 ChamarT5(inst);
84                 inst++;
85                 ChamarT4(inst);
86                 inst++;
87             }
88         }
89     }
90     else if(tipo==1){
91         if(cont%5==1 && (cont%2==1)){
92             if(inst>=4){
93                 inst=inst-3;
94                 ChamarT5(inst);
95                 inst=inst+3;
96             }
97             inst=inst-2;
98             ChamarT4(inst);
99             inst=inst+2;
100         }
101     }
102     else if(tipo==3){
103         if(cont%2==0){
```

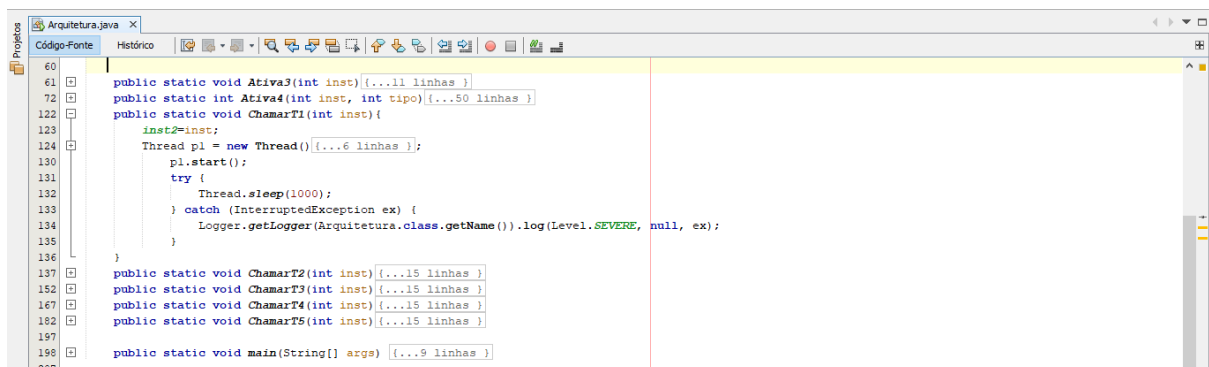
Figura 14: Ativar 4, a função que mais recebe argumentos



```
102     if(cont%2==0){
103         if(inst==1){
104             ChamarT4(inst);
105             System.out.println("-----");
106             ChamarT5(inst);
107         }else{
108             inst--;
109             System.out.println("-----");
110             ChamarT5(inst);
111             inst++;
112             ChamarT4(inst);
113             System.out.println("-----");
114             ChamarT5(inst);
115         }
116     }
117 }
118
119 return inst;
120 }
121 }
```

Figura 15: Continuação da Figura 14

As funções Chamar t1 à t5 referem-se a todos os estágios da threads em execução, são funções que chamam Threads e que clonam o valor de inst para inst2 para a saída correta.



```
60
61 public static void Ativa3(int inst){...11 linhas }
62
63 public static int Ativa4(int inst, int tipo){...50 linhas }
64
65 public static void ChamarT1(int inst){
66     inst2=inst;
67     Thread p1 = new Thread(){...6 linhas };
68     p1.start();
69     try {
70         Thread.sleep(1000);
71     } catch (InterruptedException ex) {
72         Logger.getLogger(Arquitetura.class.getName()).log(Level.SEVERE, null, ex);
73     }
74 }
75
76 public static void ChamarT2(int inst){...15 linhas }
77 public static void ChamarT3(int inst){...15 linhas }
78 public static void ChamarT4(int inst){...15 linhas }
79 public static void ChamarT5(int inst){...15 linhas }
80
81 public static void main(String[] args){...9 linhas }
```

Figura 18: Funções de chamada

Por fim a main, onde ocorre a única chamada de função, a ativar 1, que irá desencadear o resto do processo:

```
198 public static void main(String[] args) {  
199     // TODO code application logic here  
200     Scanner in = new Scanner(System.in);  
201     int n;  
202     System.out.println("Digite o numero de instrucoes: ");  
203     n = in.nextInt();  
204     Ativar1(n);  
205 }  
206
```

Figura 19 : Função main

CONCLUSÃO

Neste relatório, apresentamos os resultados do estudo do comportamento de pipelining, pudemos ver como um processador trabalha com várias tarefas em paralelo por meio da ferramenta de threads, ofertada pela linguagem JAVA. Sob esse viés, pudemos verificar em prática o funcionamento do paralelismo de instruções, permitindo-nos maior compreensão da teoria vista em sala. De acordo com a figura 01, comprovou-se que a pipeline é eficiente em reduzir o tempo necessário para se executar várias instruções, ou seja, várias instruções estarem sendo executadas ao mesmo tempo, ou várias instruções estarem em diferentes estágios da pipeline, já nas figuras 2 à 7 mostramos visualmente a abstração dos tipos de pipeline. Por fim, o código mostra o resultado esperado, com os atrasos de thread e especificações, por exemplo, o uso do método “run”, assim como foi pedido pelo docente.