

# **Algoritmo de Tomasulo Especulativo (Versão 2)**

27 de junho de 2018

**Disciplina: CES-25**

Estudantes: Felipe Guimarães, Felipe Uchida, Dennys e Lucas Jorge

Turma 19

**Instituto Tecnológico de Aeronáutica**



## I. Objetivo

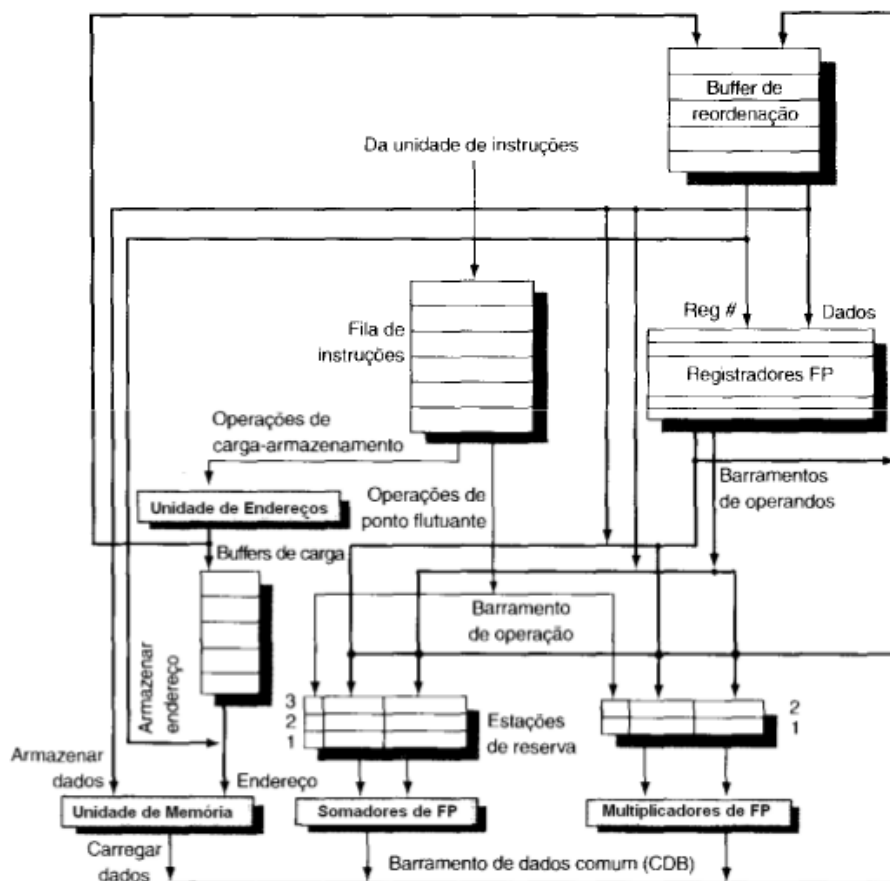
O objetivo deste projeto é aplicar os conhecimentos adquiridos na disciplina de CES-25 (Arquiteturas para Alto Desempenho). Com isso, foi desenvolvido na primeira parte desse projeto algoritmo de Tomasulo, que simula a realização de instruções paralelamente.

O algoritmo foi implementado usando a Python 3, e a interface gráfica foi desenvolvida com a library PyQt 5. O código desenvolvido pela equipe foi mantido no github ([https://github.com/Lucas1Jorge/Speculative\\_Tomasulo\\_Algorithm](https://github.com/Lucas1Jorge/Speculative_Tomasulo_Algorithm)).

## II. Descrição

A primeira parte do projeto (Algoritmo de Tomasulo não especulativo) já foi descrita anteriormente e encontra-se documentada (Report\_V1.pdf, no link do github já mencionado).

Nessa nova versão, adiciona-se ao hardware o Buffer de Reordenação:



*Illustration 1: Hardware para o Tomasulo com Buffer de Reordenação no canto superior direito*

O algoritmo para implementar o ROB (Reorder Buffer) é descrito na imagem abaixo:

Status	Esperar até	Ação ou contabilidade
<b>Emitir todas as instruções</b>	Estação de reserva (r) e ROB (b) disponíveis	<pre> if (RegisterStat[rs].Busy) /* instr. grava rs durante execução */ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr. já concluída */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* espera por instrução */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;} RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no; </pre>
<b>Operações de FP e armazenamentos</b>		<pre> if (RegisterStat[rt].Busy) /* instr. grava rt durante execução */ {h ← RegisterStat[rt].Reorder; if (ROB[b].Ready) /* Instr. já concluída */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* Espera por instrução */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;} </pre>
<b>Operações de FP</b>		RegisterStat[rd].Qi=b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;
<b>Caregamentos</b>		RS[r].A ← imm; RegisterStat[rt].Qi=b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;
<b>Armazenamentos</b>		RS[r].A ← imm;
<b>Executar Op. de FP</b>	(Rs[r].Qj = 0) e (Rs[r].Qj = 0)	Calcular resultados – operandos estão em Vj e Vk
<b>Load etapa 1</b>	(Rs[r].Qj = 0) e não existe nenhum armazenamento anterior na fila	RS[r].A ← RS[r].Vj + RS[r].A;
<b>Load etapa 2</b>	Etapa 1 de Load concluída e todos os armazenamentos anteriores em ROB têm endereço diferente	Ler de Mem[Rs[r].A]
<b>Store</b>	(Rs[r].Qj = 0) e armazenamento no início da fila	ROB[h].Address ← RS[r].Vj + RS[r].A;
<b>Gravar, exceto store</b>	Execução feita em r e CDB disponível.	<pre> b ← RS[r].Reorder; RS[r].Busy ← no; vx(if (RS[x].Qj=b) {RS[x].Vj ← result; RS[x].Qj ← 0}); vx(if (RS[x].Qk=b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes; </pre>

Status	Esperar até	Ação ou contabilidade
<b>Store</b>	Execução feita em r e (Rs[r].Qk = 0)	ROB[h].Value ← RS[r].Vk;
<b>Consolidação</b>	A instrução está no início do ROB (entrada h) e ROB[h].ready = yes	<pre> d = ROB[h].Dest; /* dest. de registrador, se existir */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;}} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Address] ← ROB[h].Value;} else /* insere o resultado no destino de registrador */ {Regs[d] ← ROB[h].Value;} ROB[h].Busy ← no; /* libera entrada de ROB */ /* libera registrador de dest. se nenhuma outra instr. estiver gravando nele */ if (RegisterStat[d].Qi==h) {RegisterStat[d].Busy ← no;}; </pre>

*Illustration 2: Algoritmo para implementar o Buffer de Reordenação*

O algoritmo de Tomasulo permite que operações de adição/subtração, multiplicação/divisão e load/store sejam calculadas paralelamente, melhorando o desempenho do processamento. Para operar dessa forma, o hardware contém uma fila de instruções que lê as instruções da memória e redireciona para estações de reserva, que podem ser dos tipos “**add\_sub**”, “**mult\_div**” ou “**load\_store**”. Como não foi pedido para implementar a divisão, a estação “**mult\_div**” acabou sendo apenas “**mult**”.

Cada **reservation station** atua como uma queue (no código foi implementada como lista circular FIFO) associada a um hardware que executa (classe **executer**) as instruções, uma por vez. Assim, quando o hardware **executer** está livre (`busy == False`), a estação de reserva pode mandar a instrução no topo da queue para ser executada. Cada estação de reserva tem seu próprio hardware e pode operar independentemente das outras.

No algoritmo não especulativo, os resultados calculados nos **executers** eram jogados no **common data bus** para serem salvos no banco de registradores. A diferença na versão Especulativa é que, antes de serem salvos nos registradores, os resultados são enviados para o **Reorder Buffer**. Após isso, os resultados só serão salvos de fato nos registradores quando houver certeza de que devem ser salvos. Essa etapa é denominada **Consolidating phase**, e ocorre quando não há nenhuma instrução **branch** esperando para ser calculada.

Toda vez que acontecia um **branch** no Tomasulo não especulativo, era necessário esperar seu resultado para saber se o **Program Counter** deveria ler a próxima instrução em sequência ou a instrução do destino para o qual saltaria. Agora, no algoritmo especulativo, assume-se que um caminho mais provável será seguido e processa-se as instruções nesse caminho, porém, guardando os resultados no **ROB** enquanto não se tem certeza de que o caminho seguido foi o correto. Quando o **branch** pendente é calculado, as instruções no **ROB** podem ser consolidadas se a previsão estava correta, ou devem ser apagadas sem ser salvas se a previsão errou.

Para a previsão dos **branches**, utilizou-se um Buffer de Destino de Desvios (**destiny buffer**). Quando um branch é calculado, se o desvio for seguido, adiciona-se o PC de origem do branch no **destiny buffer**, junto com o correspondente destino do desvio. Assim, da próxima vez que aquele mesmo branch aparecer o algoritmo assumirá que deve saltar para o mesmo destino. Na prática, essa suposição é a mais provável devido ao princípio da localidade. Isto é, o código tende a passar várias vezes pelo mesmo lugar, processando algum laço de repetição. Se a previsão de saltar foi incorreta, o par PC origem/ PC destino é apagado do destiny buffer de modo que, se o algoritmo passar novamente pelo branch, ele não saltará a princípio. No código, o **destiny buffer** foi implementado como um dicionário no qual para cada **PC de origem** (key) de um branch correspondia um **PC de destino** (value).

Da forma como foi implementado, considera-se que a ROB pode consolidar instruções paralelamente. Isto é, se **k** instruções estiverem consecutivamente prontas para serem consolidadas no topo do ROB, as **k** instruções serão consolidadas.

Assim, era, a princípio, esperado que houvesse uma melhora no desempenho do algoritmo em relação à versão 1, uma vez que não é mais necessário esperar até que um desvio seja calculado para processar as próximas instruções. E ainda, devido ao princípio da localidade, a grande maioria das previsões estará provavelmente correta.

### III. Resultados Obtidos

Desenvolveu-se a seguinte interface gráfica para o algoritmo em python, usando a library PyQt5:

Play

Auto play

Pause

Main

ROB

	Tipo	Busy	Instrução	Estado	Vj		Endereço	Valor
ER1	Load/Store	False					24 4	1
ER2	Load/Store	False	SW	issued	2		28	
ER3	Load/Store	False					32	
ER4	Load/Store	False					36	
ER5	Load/Store	False						
ER6	Add	False	ADD	issued	2			
ER7	Add	False						
ER8	Add	False						
ER9	Mult	False						
ER10	Mult	True	MUL	Executing	3	1		
ER11	Mult	False						

Valor

Clock Corrente	17
PC	28
N.I.C.	11
CPI	1.455

	Qi	Vi
		0
		3
R2		2
R3		0
R4		1
R5		2
R6		4

*Illustration 3: Graphic interface for a set of instructions processing*

Play

Auto play

Pause

Main

ROB

	Ocupado	ID	Estado	Destino	Valor
1	True	A1	Not Ready	4	
2					
3					
4					
5					
6					
7	True	S0	Not Ready		
8	True	A2	Consolidati...	2	
9	True	A0	Consolidati...	5	
10		BLE			

	R0	R1	R2	R3	R4	R5	R6
Reordenação			7		0	8	
Ocupado			True		True	True	

*Illustration 4: ROB graphic Interface*

## Testes:

Para testar a exatidão do algoritmo, usou-se o seguinte benchmark, cuja saída era conhecida para uma entrada conhecida:

```
ADDI R1,R0,3
ADDI R2,R0,1
ADDI R4,R0,1
P1:
ADDI R5,R0,1
P2:
MUL R6,R1,R4
ADD R6,R6,R5
SW R2,0(R6)
ADDI R2,R2,1
ADDI R5,R5,1
BLE R5,R1,P2
ADDI R4,R4,1
BLE R4,R1,P1
ADDI R2,R0,0
ADDI R4,R0,1
P3:
MUL R6,R4,R1
ADD R6,R6,R4
LW R6,0(R6)
ADD R2,R2,R6
ADDI R5,R4,1
ADDI R9,R1,1
BEQ R5,R9,P5
P4:
MUL R6,R4,R1
ADD R6,R6,R5
LW R3,0(R6)
MUL R7,R5,R1
ADD R7,R7,R4
LW R8,0(R7)
SW R8,0(R6)
ADD R2,R2,R8
SW R3,0(R7)
ADD R2,R2,R3
ADDI R5,R5,1
BLE R5,R1,P4
ADDI R4,R4,1
BLE R4,R1,P3
P5:
ADDI R6,R0,0
```

*Illustration 5: benchmark assembly program used to test the outputs of the algorithm*

Sabe-se que, iniciando o programa colocando em R1 os valores 3, 5 ou 6, deve-se, respectivamente, terminar o programa com os valores 45, 325 e 666 no registrador R2. Tais resultados foram confirmados, conforme as figuras abaixo:

	Valor inicial em R1	Valor Final esperado em R2
Configuração 1	3	45
Configuração 2	5	325
Configuração 3	6	666

*Illustration 6: Configurações a serem testadas: outputs esperados para cada input em R1*

Como o algoritmo lê entradas em binário, o benchmark mencionado acima foi convertido para o seguinte formato, que foi o input de fato utilizado nos testes:

```
00100000000000010000000000000011
00100000000000010000000000000001
0010000000000100000000000000001
00100000000001010000000000000001
000000000001001000011000000011000
000000000110001010011000000100000
10101100110000100000000000000000
00100000010000100000000000000001
00100000010100101000000000000001
00011100101000010000000000010000
00100000100001000000000000000001
00011100100000010000000000001100
00100000000000100000000000000000
00100000000001000000000000000001
00000000100000010011000000011000
00000000110001000011000000100000
10001100110001100000000000000000
00000000010001100001000000100000
00100000100001010000000000000001
00100000001010010000000000000001
00010100101010010000000000111000
00000000100000010011000000011000
00000000110001010011000000100000
10001100110000110000000000000000
00000000101000010011100000011000
00000000111001000011100000100000
10001100111010000000000000000000
10101100110010000000000000000000
00000000010010000001000000100000
10101100111000110000000000000000
00000000010000110001000000100000
00100000101001010000000000000001
00011100101000010000000001010100
00100000100001000000000000000001
00011100100000010000000000111000
00100000000001100000000000000000
```

*Illustration 9: Instruções em binário correspondentes ao programa benchmark mencionado acima (conforme formato requisitado na especificação do projeto).*

**Observação:** para alterar o valor inicial de R1, deve-se alterar a constante binária (16 últimos bits) na primeira linha para o valor desejado (completando à esquerda com zeros).

### Configurações 1, 2 e 3:

Endereço		Valor
24	11	8
28	9	8
32	11	6
36	12	9

Valor	
Clock Corrente	209
PC	144
N.I.C.	110
CPI	1.891

Qi	Vi
	0
	3
R2	45
R3	6
R4	3
R5	4
R6	0

 | Endereço |    | Valor | |----------|----|-------| | 24       | 29 | 24    | | 28       | 25 | 24    | | 32       | 29 | 20    | | 36       | 30 | 25    |     | Valor          |       | |----------------|-------| | Clock Corrente | 521   | | PC             | 144   | | N.I.C.         | 285   | | CPI            | 1.825 |     | Qi | Vi  | |----|-----| |    | 0   | |    | 5   | | R2 | 325 | | R3 | 20  | | R4 | 5   | | R5 | 6   | | R6 | 0   | | | Endereço |    | Valor | |----------|----|-------| | 24       | 41 | 35    | | 28       | 36 | 35    | | 32       | 41 | 30    | | 36       | 42 | 36    |     | Valor          |       | |----------------|-------| | Clock Corrente | 726   | | PC             | 144   | | N.I.C.         | 404   | | CPI            | 1.795 |     | Qi | Vi  | |----|-----| |    | 0   | |    | 6   | | R2 | 666 | | R3 | 30  | | R4 | 6   | | R5 | 7   | | R6 | 0   | |

Illustration 7 : Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o benchmark para a **Configuração 1** (R1 = 3 no início), **Configuração 2** (R1 = 5 no início) e **Configuração 3** (R1 = 6 no início), respectivamente.

### Testes 2:

Na especificação do projeto, foi pedido que se testasse mais 3 configurações: colocando o valor inicial de R1 em **10, 15 ou 20**. Para esses testes, obteve-se em R2 os valores finais **5050, 25425 e 80200**, respectivamente, conforme a figura a seguir:



### Configurações 4, 5 e 6:

Endereço		Valor
24	109	99
28	100	99
32	109	90
36	110	100

Valor	
Clock Corrente	1906
PC	144
N.I.C.	1090
CPI	1.748

Qi	Vi
	0
	10
R2	5050
R3	90
R4	10
R5	11
R6	0

Endereço		Valor
24	239	224
28	225	224
32	239	210
36	240	225

Valor	
Clock Corrente	4172
PC	144
N.I.C.	2420
CPI	1.724

Qi	Vi
	0
	15
R2	25425
R3	210
R4	15
R5	16
R6	0

Endereço		Valor
24	419	399
28	400	399
32	419	380
36	420	400

Valor	
Clock Corrente	7304
PC	144
N.I.C.	4275
CPI	1.708

Qi	Vi
	0
	20
R2	80200
R3	380
R4	20
R5	21
R6	0

Illustration 8: Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o benchmark para a **Configuração 4** (R1 = 10 no início), **Configuração 5** (R1 = 15 no início) e **Configuração 6** (R1 = 20 no início), respectivamente.

Adicionalmente, criou-se um programa que calcula o **MDC** de dois números para testar o algoritmo desenvolvido:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  void swap(int* a, int* b) {
6      int temp = *a;
7      *a = *b;
8      *b = temp;
9  }
10
11
12  int mdc(int a, int b) {
13      if (a == 0 || b == 0) return 0;
14
15      int temp;
16
17      while (b != 0) {
18          if (b > a) swap(&a, &b);
19          temp = b;
20          b = a - b;
21          a = temp;
22      }
23
24      return a;
25  }
26

```

*Illustration 12: Programa em C para calcular o MDC de dois números*

0010000000000000100000000000111000	; ADDI R1, R0, 56
0010000000000000100000000000110001	; ADDI R2, R0, 49
000111000100000010000000000011000	; BLE R2, R1, 24
0000000000100000100010000000100000	; ADD R2, R1, R2
0000000000100000100001000000100010	; SUB R1, R2, R1
0000000000100000100010000000100010	; SUB R2, R2, R1
0000000000100000000001100000100000	; ADD R3, R2, R0
0000000000100010000010000000100010	; SUB R2, R1, R2
0000000000110000000000100000100000	; ADD R1, R3, R0
000101000100000000000000000011100	; BEQ R2, 28, R0
00001000000000000000000000001000	; JMP 8

*Illustration 15: Equivalente ao programa em C para calcular MDC entre 56 e 49 em binário e linguagem MIPS CES25.2018SE. Os registradores R1 e R2 devem ser inicializados com os dois números para os quais se quer obter o MDC.*

MDC (56, 59):

Endereço		Valor
24		
28		
32		
36		
		Valor
Clock Corrente		66
PC		68
N.I.C.		41
CPI		1.585
Qi		Vi
		0
		7
R2		0
R3		7
R4		0
R5		0
R6		0
R7		0
R8		0
R9		0
R10		0
R11		0

Illustration 14: Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o MDC (56, 49). O registrador R1 contém o resultado: 7.

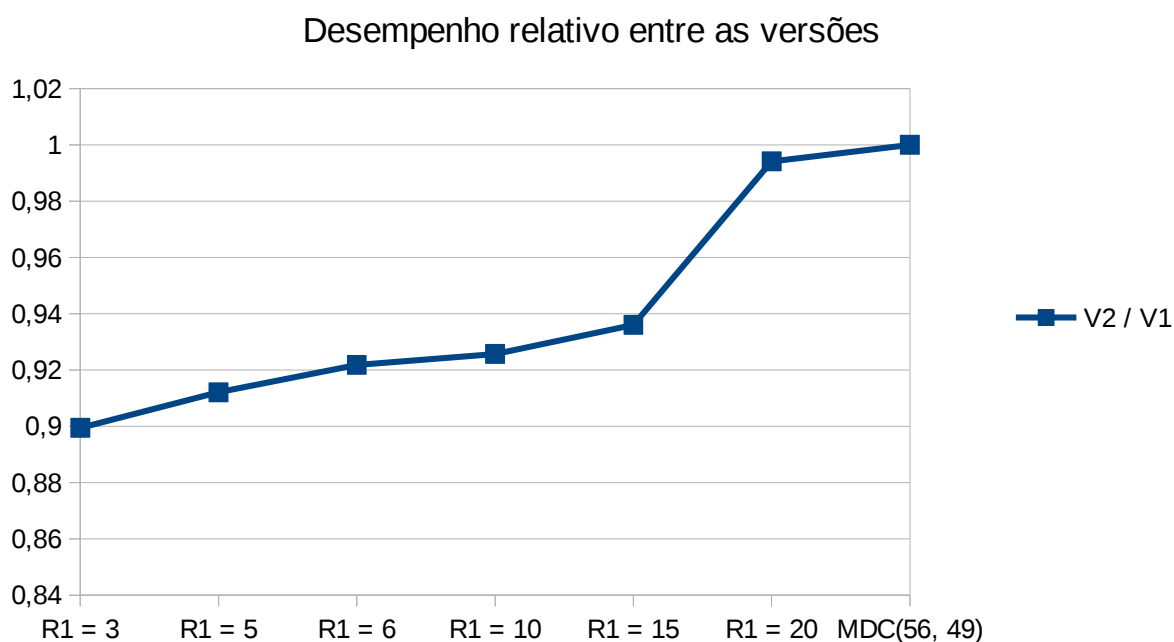
## IV. Comparação entre as versões 1 e 2

Conforme será descrito adiante, o algoritmo de benchmark calcula a soma dos valores de 1 até o valor em R1 ao quadrado. Para isso, ele percorre dois **loops** de ordem 2 em relação ao valor em R1. Como o predictor implementado foi o de 1 bit, a previsão sempre errará no início e no final de cada loop (no início, ela não salta, mas devia saltar; no final, ela salta, mas não devia saltar).

Dessa forma, para  $R1 = 3$ , por exemplo, a previsão será incorreta 2 a cada 3 vezes em cada loop interno, e cada loop interno acontece 3 vezes, em 2 partes diferentes dos código, totalizando 12 erros em 18 previsões. Para  $R1 = 5$  tem-se 2 erros a cada 5 previsões nos loops internos, e o restante do raciocínio é análogo. O mesmo vale para os outros valores de  $R1$ .

Então, espera-se um ganho de desempenho maior (ou perda menor) quanto maior o valor de  $R1$ .

Conforme os valores, já mostrados acima, obtidos quando o algoritmo processava o benchmark com diferentes valores em  $R1$ , obtém-se a seguinte comparação em relação aos valores obtidos para os mesmos testes com a versão 1 (Tomasulo não Especulativo):



Observa-se que, diferente do que era esperado, o desempenho piorou na versão 2. Um motivo para a perda de desempenho no algoritmo especulativo é que, cada vez que se processa um LW, deve-se esperar até que não haja nenhum SW na ROB escrevendo naquela posição de memória. Isso porque, como a escrita na memória é irreversível, deve-se esperar até que se tenha certeza de que o desvio foi correto para processar essa escrita. Assim, a memória escrita pelo SW só está disponível para o LW após a consolidação. Talvez se houvesse também um buffer de reordenação para a memória isso pudesse ser contornado. Mas isso foge ao escopo deste projeto.

Como será descrito abaixo, na **seção V: Análise do benchmark**, o programa de benchmark envolve um loop de ordem 2 escrevendo uma matriz  $R1$  por  $R1$ , e um segundo loop de ordem 2 somando os elementos dessa matriz e, adicionalmente, transpondo a matriz. Nessa operação de transposição, a dependência entre LW e SW mencionada acima ocorre em cada iteração. Por isso ela é impactante no desempenho.

Por outro lado, conforme esperado, quanto maior o valor de  $R1$ , menor a perda de desempenho, devido ao aumento da proporção de acertos nos saltos dos loops (erra no início e no

final e acaba no interior do loop).

## V. Análise do benchmark:

O programa de benchmark cria uma matriz quadrada de lado =  $R1$  na memória e preenche ordenadamente as linhas e colunas com os valores de 1 até  $R1^2$ . Em seguida o programa itera a matriz de nível em nível, somando cada elemento (o elemento na diagonal principal, mais os valores acima da diagonal, mais os valores à esquerda da diagonal). Isto é, o programa soma todos os valores de 1 até  $R1^2$ . Desta forma, obtém-se:

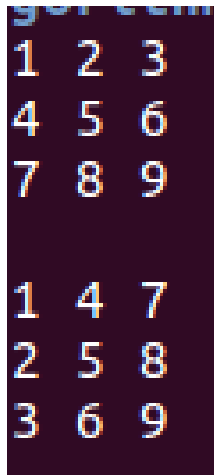
- $R1 = 3$ :  $1 + 2 + 3 + \dots + 9 = 45$
- $R1 = 5$ :  $1 + 2 + 3 + \dots + 25 = 325$
- ...
- $R1 = 20$ :  $1 + 2 + \dots + 400 = (1 + 400) * 400 / 2 = 80200$

Além de somar os elementos, enquanto itera a matriz o programa faz um swap entre os elementos  $[i][j]$  e  $[j][i]$ . Isto é, o programa também transpõe a matriz inicialmente escrita na memória.

Um equivalente em python foi escrito a partir do Assembly do benchmark dado:

```
1 memo = [0] * 4000
2 r1 = 3
3 r2 = 1
4
5 for i in range(1, r1 + 1):
6     for j in range(1, r1 + 1):
7         memo[i * r1 + j] = r2
8         r2 += 1
9
10 r2 = 0
11
12 for i in range(1, r1 + 1):
13     r2 += memo[i * r1 + i]
14
15     for j in range(i + 1, r1 + 1):
16         aux = memo[i * r1 + j]
17
18         memo[i * r1 + j] = memo[j * r1 + i]
19
20         r2 += memo[j * r1 + i]
21
22         memo[j * r1 + i] = aux
23
24         r2 += memo[j * r1 + i]
25
26
27 # r6 = 0
28
29 print(r2)
30
```

*Illustration 15: Equivalente em python para o programa de benchmark*



1	2	3
4	5	6
7	8	9

1	4	7
2	5	8
3	6	9

Illustration 16: Matriz criada e transposta na memória para o cálculo do benchmark com  $R1 = 3$  no início.

## VI. Conclusões

Dessa forma, embora não se tenha atingido o ganho de desempenho buscado na implementação da especulação, acredita-se que o problema esteja não no método, mas nas especificidades do benchmark. Para testes mais conclusivos, o programa deveria ser testado com outros benchmarks. Embora o algoritmo tenha sido testado também com o programa desenvolvido para MDC, esse processamento era curto e não tinha loops suficientes para manifestar diferenças entre as duas versões. Confirmou-se, porém, que maiores valores de  $R1$  ocasionam menor taxa de erro nas previsões e, conseqüentemente, menor perda de desempenho.

Outra possibilidade de melhoria seria, conforme já mencionado, implementar um análogo ao buffer de reordenação para as modificações na memória (porém, não temos conhecimento da viabilidade teórica dessa implementação).

## VII. Referências

Tutorial de PyQt: <https://www.youtube.com/playlist?list=PLQVvvaa0QuDdVpDFNq4FwY9APZPGSUyR4>