# Relatório do Projeto do Tomasulo Versão 1

12 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, é feita nesse trabalho uma implementação do algoritmo de Tomasulo, que simula a realização de instruções paralelas.

O algoritmo foi implementado usando a linguagem Python, assim como sua interface gráfica que foi implementada usando a biblioteca PyQt. O código foi mantido no github e desenvolvido pelos membros dessa equipe de trabalho.

# II. Descrição

A seguinte descrição do algoritmo foi usada como base para o código desenvolvido em Python 3:

**Algoritmo de Tomasulo**

Extraído de Hennesy & Patterson. Arquitetura de Computadores: Uma Abordagem Quantitativa. Tradução da 3ª. edição americana. Editora Campus-Elsevier.

| Estado de Instrução | | Esperar até | Ação ou contabilidade |
|---|---|---|---|
| **Emitir** | Operação de FP | Estação r vazia | `if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Qi=r;` |
| | Load ou Store | Buffer r vazio | `if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;` |
| | Load somente | | `RegisterStat[rt].Qi=r;` |
| | Store somente | | `if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};` |
| **Executar** | Operação de FP | (RS[r].Qj = 0) e (RS[r].Qk = 0) | Calcular resultado: operandos estão em Vj e Vk |
| | Load-Store etapa 1 | RS[r].Qj = 0 & r é o início da fila de carga-armazenamento | `RS[r].A ← RS[r].Vj + RS[r].A;` |
| | Load etapa 2 | Load etapa 1 concluída | Ler de Mem[RS[r].A] |
| **Gravar Resultado** | Operação de FP ou Load | Execução concluída em r e CDB disponível | `∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;` |
| | Store | Execução concluída em r e RS[r].Qk = 0 | `Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;` |

*Illustration 1: Descrição geral do algoritmo de Tomasulo*

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.

Após calcular a instrução, o **executer** da **reservation station** joga o resultado no **common data bus** para ser levado ao banco de registradores**.** Esse data bus também está conectado às próprias estações de reserva, pois elas podem precisar dos resultados de alguma operação para determinar seus operandos.

Porém, como instruções diferentes podem custar tempos de execução (add e mult, por exemplo) e o algorítmo permite execução paralela, as instruções lidas podem ser concluídas fora de ordem. Para evitar que elas sejam consolidadas fora de ordem, deve-se, depois que uma instrução deixa a fila de instruções, marcar no registrador (**issue**) em que ela vai escrever que ele deve esperar o resultado dessa instrução (o campo **Qi** de cada registrador guarda o label da estação de reserva que vai enviar o resultado ao registrador). Se outra instrução posteriormente for escrever no mesmo registrador enquanto o resultado da anterior ainda não foi calculado, a segunda deve apenas sobrescrever o **Qi** desse registrador. Depois que o resultado chegar ao banco de registradores pelo **common_data_bus**, ele é escrito no campo **Vi** do registrador, e o campo **Qi** é limpo.

Na implementação adotada, cada **reservation station** possui apenas um **executer**, mas nada impede que mais hardware fosse adicionado a cada uma, para que mais instruções fossem concluídas por clock. Da mesma forma, cada executor poderia ter seu próprio **common data bus** para que mais instruções fossem escritas nos registradores por clock.

De forma semelhante ao que ocorre com os registradores, as **reservation station**s também possuem valores **Q** e **V**. Quando uma instrução deixa a fila de instruções, busca-se os valores dos quais ela depende no banco de registradores. Se os valores são encontrados, eles são guardados nos campos **Vj** e **Vk** na estação de reserva da instrução. Senão, guarda-se em **Qj** ou **Qk** o label que o registrador está esperando (**Qi**). Quando a instrução no label guardado ficar pronta e for jogada no **common data bus**, ela já pode atender direto às estações de reserva que dependem dela, sem ter que passar antes pelo registrador.

# III. Resultados Obtidos

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

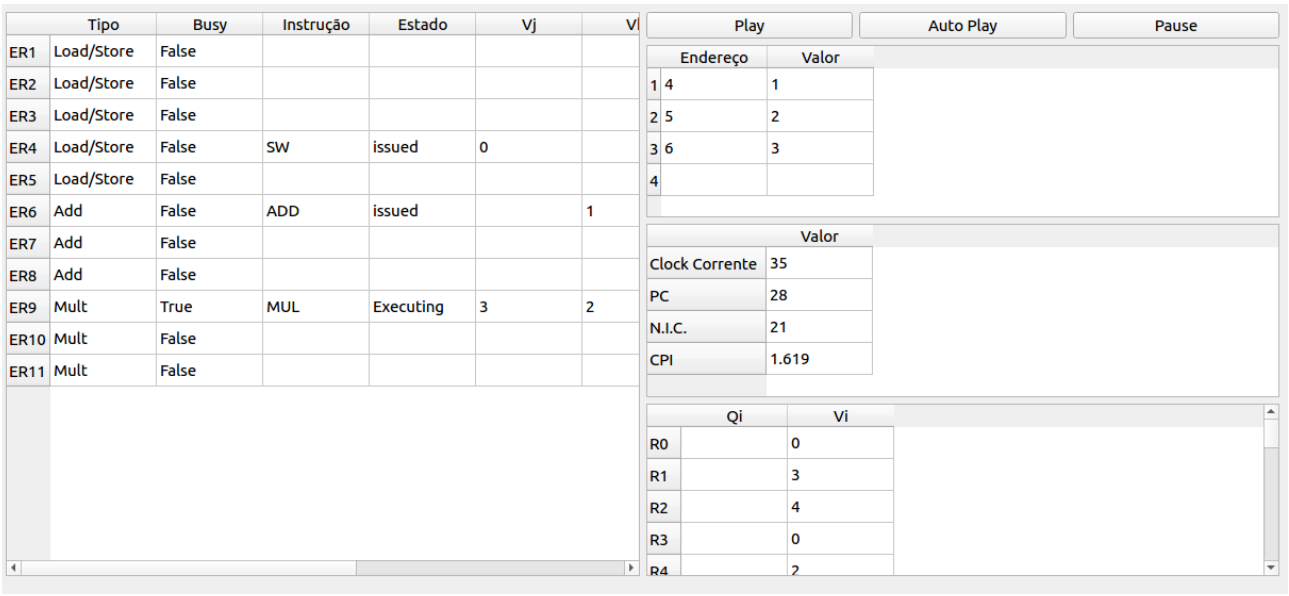


*Illustration 2: Graphic interface for a set of instructions processing*

**Testes 1:**

Para testar a exatidão do algoritmo, usou-se o seguinte benchmark, cuja saida 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 3: 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 a seguir:

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

*Illustration 4: Configurações a serem testadas: outputs esperados para cada input em R1*

**Configuração 1:**

| | Endereço | Valor |
|---|---|---|
| 1 | 11 | 8 |
| 2 | 9 | L0 |
| 3 | 11 | L4 |
| 4 | 12 | 9 |

| | Valor |
|---|---|
| Clock Corrente | 188 |
| PC | 144 |
| N.I.C. | 110 |
| CPI | 1.7 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 3 |
| R2 | | 45 |
| R3 | | 6 |
| R4 | | 3 |
| R5 | | 4 |
| R6 | | 0 |
| R7 | | 11 |
| R8 | | 8 |

Illustration 5 : Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o benchmark para a **Configuração 1** (R1 = 20 no início)

**Configuração 2:**

| | Endereço | Valor |
|---|---|---|
| 1 | 29 | 24 |
| 2 | 25 | L1 |
| 3 | 29 | L0 |
| 4 | 30 | 25 |

| | Valor |
|---|---|
| Clock Corrente | 475 |
| PC | 144 |
| N.I.C. | 285 |
| CPI | 1.663 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 5 |
| R2 | | 325 |
| R3 | | 20 |
| R4 | | 5 |
| R5 | | 6 |
| R6 | | 0 |
| R7 | | 29 |
| R8 | | 24 |

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

**Configuração 3:**

| | Endereço | Valor |
|---|---|---|
| 1 | 41 | 35 |
| 2 | 36 | L3 |
| 3 | 41 | L2 |
| 4 | 42 | 36 |

| | Valor |
|---|---|
| Clock Corrente | 668 |
| PC | 144 |
| N.I.C. | 404 |
| CPI | 1.651 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 6 |
| R2 | | 666 |
| R3 | | 30 |
| R4 | | 6 |
| R5 | | 7 |
| R6 | | 0 |
| R7 | | 41 |
| R8 | | 35 |

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

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:

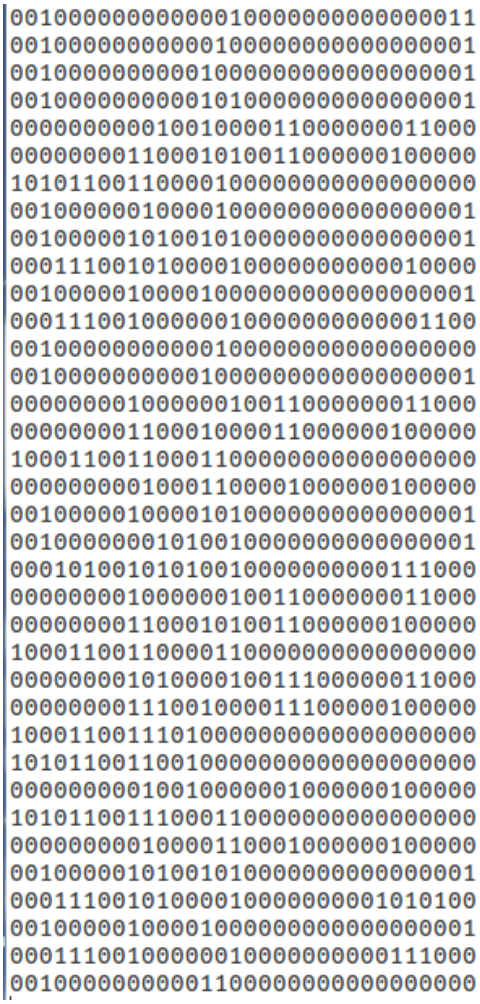


Illustration 8: Instruções em binário correspondentes ao programa benchmark mencionado acima (conforme formato requisitado na especificação do projeto).

Observação: para colocar os valores iniciais de R1=5 ou R1=6 (testes 2 e 3), deve-se alterar a constante binária (16 últimos bits) na primeira linha para 101 e 110, respectivamente (completando à esquerda com zeros).

**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 as figuras a seguir:

**R1 = 10:**

| | Endereço | Valor |
|---|---|---|
| 24 | 109 | 99 |
| 28 | 100 | L1 |
| 32 | 109 | L0 |
| 36 | 110 | 100 |

| | Valor |
|---|---|
| Clock Corrente | 1770 |
| PC | 144 |
| N.I.C. | 1090 |
| CPI | 1.623 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 10 |
| R2 | | 5050 |
| R3 | | 90 |
| R4 | | 10 |
| R5 | | 11 |
| R6 | | 0 |
| R7 | | 109 |
| R8 | | 99 |

Illustration 9: Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o benchmark para **R1 = 10** no início.

**R1 = 15:**

| | Endereço | Valor |
|---|---|---|
| 24 | 239 | 224 |
| 28 | 225 | L1 |
| 32 | 239 | L0 |
| 36 | 240 | 225 |

| | Valor |
|---|---|
| Clock Corrente | 3890 |
| PC | 144 |
| N.I.C. | 2420 |
| CPI | 1.607 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 15 |
| R2 | | 25425 |
| R3 | | 210 |
| R4 | | 15 |
| R5 | | 16 |
| R6 | | 0 |
| R7 | | 239 |
| R8 | | 224 |

Illustration 10: Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o benchmark para **R1 = 15** no início

**R1 = 20:**

| | Endereço | Valor |
|---|---|---|
| 24 | 419 | 399 |
| 28 | 400 | L1 |
| 32 | 419 | L0 |
| 36 | 420 | 400 |

| | |
|---|---|
| Clock Corrente | 6835 |
| PC | 144 |
| N.I.C. | 4275 |
| CPI | 1.599 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 20 |
| R2 | | 80200 |
| R3 | | 380 |
| R4 | | 20 |
| R5 | | 21 |
| R6 | | 0 |
| R7 | | 419 |
| R8 | | 399 |

Illustration 11: Valores resultantes de Memória Recentemente Usada, Clock e valores nos Registradores após processar o benchmark para **R1 = 20** no início

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

```c
#include <stdio.h>
#include <stdlib.h>


void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}


int mdc(int a, int b) {
    if (a == 0 || b == 0) return 0;

    int temp;

    while (b != 0) {
        if (b > a) swap(&a, &b);
        temp = b;
        b = a - b;
        a = temp;
    }

    return a;
}
```

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

```
00100000000000010000000000111000        ; ADDI R1, R0, 56
00100000000000010000000000110001        ; ADDI R2, R0, 49
00011100010000010000000000011000        ; BLE R2, R1, 24
00000000010000010001000000100000        ; ADD R2, R1, R2
00000000010000010000100000100010        ; SUB R1, R2, R1
00000000010000010001000000100010        ; SUB R2, R2, R1
00000000010000000001100000100000        ; ADD R3, R2, R0
00000000001000100001000000100010        ; SUB R2, R1, R2
00000000011000000000100000100000        ; ADD R1, R3, R0
00010100010000000000000000011100        ; BEQ R2, 28, R0
00001000000000000000000000001000        ; JMP 8
```

*Illustration 13: 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 |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |

| | |
|---|---|
| Clock Corrente | 66 |
| PC | 68 |
| N.I.C. | 41 |
| CPI | 1.585 |

| | Qi | Vi |
|---|---|---|
| R0 | | 0 |
| R1 | | 7 |
| R2 | | 0 |
| R3 | | 7 |
| R4 | | 0 |
| R5 | | 0 |
| R6 | | 0 |
| R7 | | 0 |
| R8 | | 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. Proposta de Alteração para Versão 2

A equipe implementará o Tomasulo Especulativo como alteração para a segunda versão desse projeto. Ele melhorará o desempenho do projeto pois na versão atual, quando se tem um jump condicional, a execução é travada, até saber para ao onde o programa salta, caso salte.

Já o especulativo, assume-se o caso mais provável para não pausar a execução, mas utiliza-se um buffer de reordenação pra salvar as instruções executadas caso o caminho assumido não seja o correto.

Logo, esse será o projeto que contará como o exame dos membros da equipe.

# V. Conclusões

O algoritmo de Tomasulo permite a eliminação de conflitos WAW e WAR, além de previnir conflitos RAW devido ao atraso das instruções que estão sem os operandos disponíveis. Além disso, tem a distribuição da lógica de detecção de conflitos e controle de execução pois ao estarem disponíveis os operandos, permitem a emissão de múltiplas instruções ao mesmo tempo, já que os resultados são disponíveis para as estações de reserva diretamente das unidades funcionais ao mesmo tempo.

Vimos também que as instruções com desvios condicionais são travadas devido ao não uso da especulação, algo que vai ser melhorado na versão segunda versão desse projeto. Além disso, ele aumenta consideravelmente a complexidade do hardware e tem sua performance limitada pois só há a presenta de um *common data bus*.

# VII. Referências

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