

# ARQUITETURA DE COMPUTADORES

2022-2023

## Laboratório 4 – Análise de um Processador Pipelined

Este laboratório destina-se a consolidar conhecimentos lecionados sobre as arquiteturas de processadores em pipeline, utilizando o simulador [Ripes](https://github.com/mortbopet/Ripes) (<https://github.com/mortbopet/Ripes>).

## Componentes do laboratório:

O laboratório tem duas componentes:

- Exercícios de preparação para o laboratório, que, não sendo avaliados, servem de guia para o trabalho a realizar durante a aula de laboratório.
- Trabalho de laboratório, que corresponde a um exercício surpresa e diferente em cada turno de laboratório, cuja complexidade é semelhante aos exercícios de preparação.

## Arquitetura do Processador Pipelined

O simulador [Ripes](https://github.com/mortbopet/Ripes) contempla a simulação de várias variantes da arquitetura de processador em pipeline, tendo como base a arquitetura ilustrada na Figura 1, correspondente a um *pipeline* de 5 estágios (IF, ID, EX, MEM e WB) e barramentos de 64 bits para endereçar as instruções e os dados. A unidade de armazenamento é composta por um banco de 32 registos de inteiros de 64 bits (x0-x31), tal como descrito nas aulas teóricas.

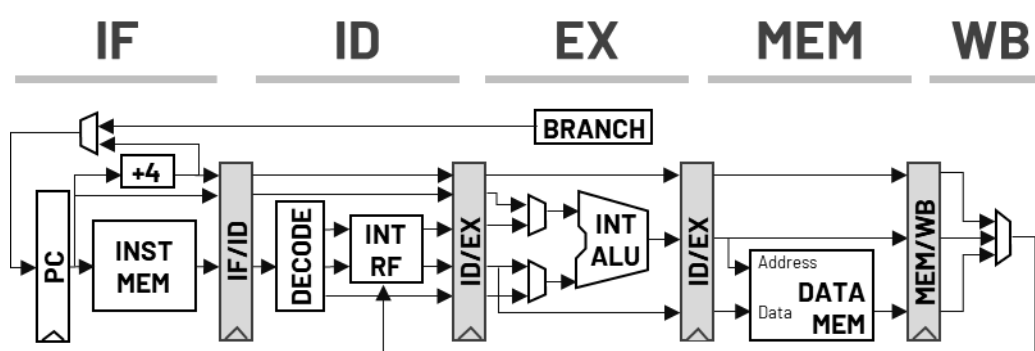


Figura 1 - Arquitetura do Processador.

Ao longo do trabalho irá ser necessário alterar a arquitetura em pipeline considerada. Para o efeito, deverá configurar o simulador de forma que este simule a arquitetura pretendida, sempre na opção 64 bits. Para tal, deverá clicar sobre o símbolo do processador do canto superior esquerdo da janela e seleccionar a opção desejada (ver Figura 2).

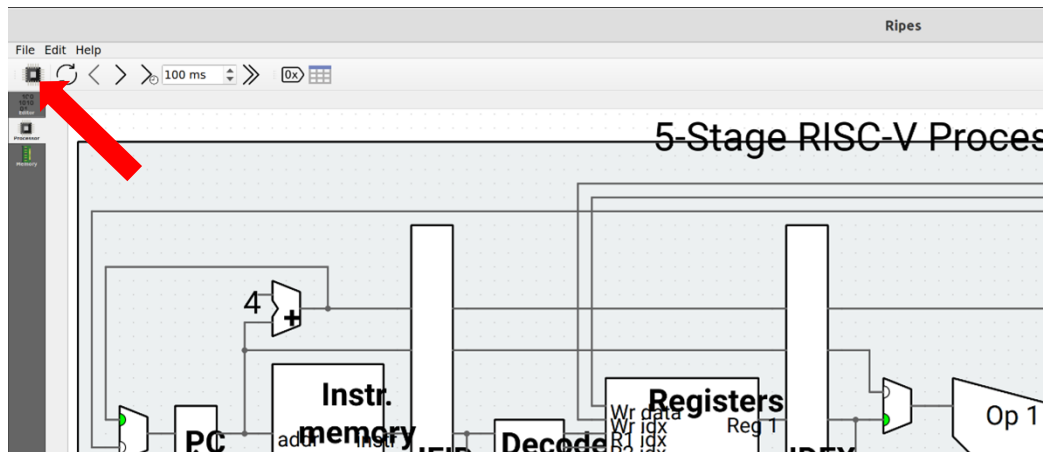


Figura 2 – Seleção da arquitetura do processador

Em qualquer das arquiteturas disponibilizadas, existe ainda a possibilidade de selecionar o Layout “Extended” para que consiga ver os detalhes da implementação do pipeline.

## Programa a Executar

Uma dada aplicação de processamento de sinal utiliza um determinado algoritmo, representado através do seguinte troço (em pseudo-código):

```
#define N 12

int a[N] = { ..... }
int b[N] = { ..... }
int x = 1;
int n = 0;
register int i = 0;           # a keyword “register” aloca a variável em registo

while (b[i]>0){
    x *= a[i] + a[N-1-i];
    n += (N-1-i) - i;
    i ++;
}
```

Este algoritmo processa dois vetores unidimensionais, constituídos por N elementos inteiros. A figura seguinte representa o código Assembly correspondente, utilizando o ISA RV32IM. Este código foi-lhe fornecido no ficheiro “L4.s”.

```
# Data section
.data
a:      .word 1, 5, 6, 6, 7, 2, 5, 2, 3, 2, 3, 4
b:      .word 4, 3, 1, 7, 2, 4, 9, -3, 5, 8, 1, 9
x:      .word 1
n:      .word 0

# Program section
.text
# NOTE: Upon start, the Global-Pointer (gp=x3) points to the beginning of .data section
addi x11, x3, 0      # x11 - address of the first element of vector a
addi x13, x11, 48     # x13 - address of the first element of vector b
addi x12, x13, -4     # x12 - address of the last element of vector a

        lw      x14, 100(x3)  # x14 - n - index distance accumulator
        lw      x15, 96(x3)   # x15 - x
        li      x16, 0        # x16 - i

while:   add     x20, x13, x16  # x20 = &b[i]
        lw      x21, 0(x20)    # x21 = b[i]
        blez    x21, end      # if b[i] <= 0 end the loop

        lw      x22, 0(x11)    # x22 = a[i]
        lw      x23, 0(x12)    # x23 = a[N-1-i]
        add     x22, x22, x23   # x22 = a[i] + a[N-1-i]
        mul     x15, x15, x22   # x15 = x15*x22 (x *= a[i] + a[N-1-i])

        sub     x22, x12, x11   # x22 = 4*((N-1-i)-i)
        sraiw   x22, x22, 2     # x22 = x22/4
        add     x14, x14, x22   # n += x22

        addi    x16, x16, 4     # i++

        addi    x11, x11, 4
        addi    x12, x12, -4
        jal     x0, while

end:     sw      x14, 100(x3)    # store n's final value
        sw      x15, 96(x3)    # store x's final value

        addi    a7, x0, 10
        ecall

# Expected result: M[x] = 1270080 = 136140h
#                  M[n] = 35      = 23h
```

Para fins meramente exemplificativos, o presente código considera um conjunto de dados pré-preenchidos nos vetores a processar, em que cada elemento foi representado com uma palavra de 32-bits (int).

NOTA: De modo a facilitar a compreensão dos conceitos que irão ser discutidos ao longo deste trabalho, o código Assembly deste programa foi devidamente preparado de modo a evitar a utilização de pseudo-instruções. Em particular, a generalidade dos endereços das estruturas de dados acedidas estão indexados ao início da secção .data, cujo endereço é passado no início do programa através do registo x3 (Global-Pointer).

## Exercício 1

1. Configure o simulador de forma que este simule a arquitetura pipeline mais simples, a qual:
  - não realiza o adiamento de dados (*forwarding*) entre estágios do *pipeline* (embora possua um banco de registos transparente) e também não realiza a detecção de dependências de dados
  - os conflitos de controlo são resolvidos por predição de salto estática *not taken*.

Para tal, clique sobre o símbolo do processador do canto superior esquerdo e selecione a seguinte opção (selecionando também o Layout “Extended” para que consiga ver os detalhes da implementação do pipeline):

Select Processor → 5-Stage Processor w/o forwarding or hazard detection

Note que a unidade de controlo de salto está localizada no estágio de EX.

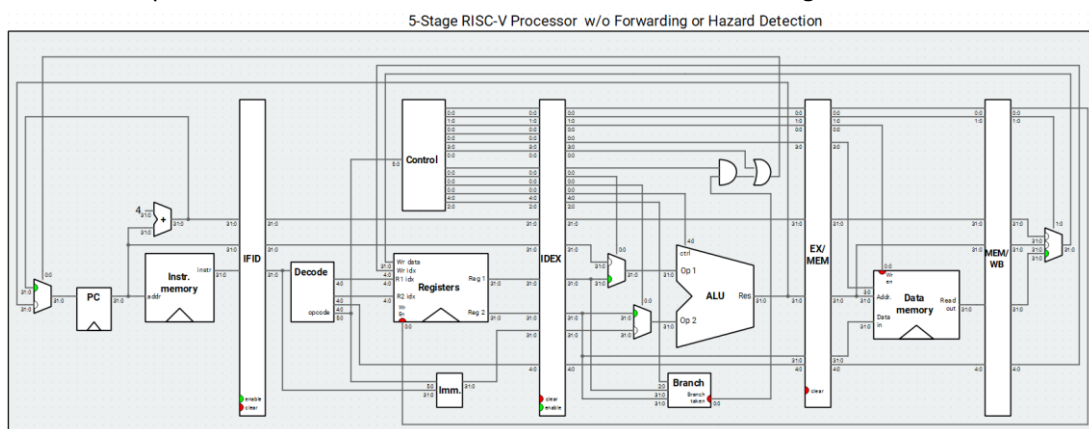


Figura 3 – Arquitetura do processador “5-stage Processor w/o forwarding or hazard detection”.

2. Carregue o programa L4.s que lhe foi fornecido.
3. Identifique os endereços de início dos vetores `a [ ]` e `b [ ]`. Deverá/poderá fazê-lo através de dois métodos:
  - Por inspeção do código  
*Sugestão: procure identificar os valores que deverão ser escritos nos registos x11 e x13 pelas duas primeiras instruções*
  - Por observação da zona de memória do simulador  
*Nota: não se esqueça de seleccionar a secção .data*
4. Coloque um *Break-Point* na primeira instrução do ciclo “while” e execute o programa até esse instante, pressionando o botão >>
5. Compare os valores nos registos x11 e x13 com os valores que identificou no ponto 3. São iguais? Porquê?
6. Tendo em consideração a arquitetura do processador, identifique todos os conflitos de dados, indicando a instrução que escreve e a instrução que lê e qual o operando que provoca o conflito.
7. Altere o código do programa de modo a garantir o correto funcionamento, assegurando a resolução de todos os conflitos de dados e de controlo.  
*Nota: resolva este exercício através da introdução de NOPs. Contudo, após resolver o exercício 9, poderá tentar minimizar o número de NOPs re-ordenando as instruções.*
8. Execute o programa completo, verificando se os resultados dos cálculos correspondem aos valores esperados (ver comentário final, no código fonte, ou compare o valor dos registos quando o código original é executado num processador de ciclo único).

- Nota: sugere-se a consulta do painel “Processador”

## Exercício 3

1. Utilizando o mesmo modelo de processador que considerou no exercício anterior ("5-Stage Processor"), modifique o programa L4.s utilizando técnicas de reordenação da sequência de instruções do programa de modo a minimizar o número de ciclo de relógios necessários à sua execução, otimizando assim o seu desempenho.
2. Quais foram as alterações que introduziu? Porquê?
3. Registe as seguintes estatísticas de execução: número de ciclos de relógio, número de instruções executadas, IPC.