Trabalho Final - Arquitetura de Computadores

João Meyer, Thiago Zilio, Tiago Augusto, Vitor Faria
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil

I. Introdução

Neste trabalho, exploramos a implementação e otimização de um processador Simple RISC com pipeline, visando demonstrar os benefícios do uso de pipelines para melhorar a eficiência do processador. A arquitetura Simple RISC foi escolhida por sua simplicidade e clareza, facilitando a compreensão de conceitos fundamentais de arquitetura de computadores.

Implementamos um pipeline de 5 estágios e avaliamos o impacto das técnicas de otimização, como forwarding e branch prediction, para gerenciar hazards e melhorar o desempenho. Utilizamos a função de cálculo da sequência de Fibonacci para validar nossa implementação e medir a eficácia das otimizações.

A. Introdução ao SimpleRISC

O SimpleRISC é uma arquitetura de processador didática, projetada para ensinar conceitos fundamentais de arquitetura de computadores de forma simplificada. Sua clareza e simplicidade permitem a implementação e experimentação de técnicas de otimização.

Inst.	Format	Inst.	Format
add	add rd, rs1, (rs2/imm)	lsl	lsl rd, rs1, (rs2/imm)
sub	sub rd, rs1, (rs2/imm)	lsr	lsr rd, rs1, (rs2/imm)
mul	mul rd, rs1, (rs2/imm)	asr	asr rd, rs1, (rs2/imm)
div	div rd, rs1, (rs2/imm)	nop	nop
mod	mod rd, rs1, (rs2/imm)	ld	ld rd, imm[rs1]
cmp	cmp rs1, (rs2/imm)	ld	ld rd, imm[rs1]
and	and rd, rs1, (rs2/imm)	st	st rd, imm[rs1]
or	or rd, rs1, (rs2/imm)	beq	beq offset
not	not rd, (rs2/imm)	bgt	bgt offset
mov	mov rd, (rs2/imm)	b	b offset
ret	ret	call	call offset

Tabela I
Instruções e seus formatos

B. Visão Geral da Arquitetura SimpleRISC

A arquitetura do SimpleRISC é caracterizada por um conjunto reduzido de instruções (RISC - Reduced Instruction Set Computer), que inclui operações aritméticas, lógicas, de desvio e de acesso à memória. Ela possui um conjunto de registradores de uso geral, uma unidade de controle simples e uma interface de memória básica. Os principais componentes do SimpleRISC incluem:

- **Conjunto de Instruções:** Um conjunto limitado de instruções que cobre as operações básicas necessárias para a execução de programas.
- **Registradores:** Um conjunto de registradores de uso geral utilizados para armazenar operandos e resultados intermediários.
- **Unidade de Controle:** Responsável pela decodificação das instruções e pela geração dos sinais de controle necessários para a execução.
- **Memória:** Interface para acesso à memória, incluindo operações de leitura e escrita.

II. Implementação do Projeto Microarquitetural

A implementação do pipeline no Simple RISC segue as etapas clássicas de um pipeline de 5 estágios:

1) **IF (Instrução Fetch)**: Captura a próxima instrução da memória. Neste estágio, a instrução é buscada da memória de instruções utilizando o contador de programa (PC), que é então incrementado para apontar para a próxima instrução.
2) **ID (Instrução Decode)**: Decodifica a instrução e lê os registradores. A instrução capturada é decodificada para entender qual operação deve ser realizada. Os operandos necessários são lidos dos registradores.
3) **EX (Execução)**: Realiza a operação aritmética ou lógica. Dependendo da instrução, a Unidade Lógica e Aritmética (ULA) executa a operação especificada, como adição, subtração, e comparações.
4) **MEM (Memória)**: Acessa a memória para operações de load e store. Para instruções que envolvem acesso à memória, como load e store, este estágio é responsável por ler ou escrever dados na memória.
5) **WB (Write Back)**: Escreve o resultado de volta no registrador. O resultado da operação é escrito de volta no banco de registradores, completando o ciclo de execução da instrução.

Cada estágio do pipeline tem funções e responsabilidades específicas que permitem que múltiplas instruções sejam processadas simultaneamente em diferentes estágios do pipeline. Isso aumenta significativamente a eficiência e a velocidade de processamento do processador.

A. Vantagens do uso de um pipeline em processadores

O uso de um pipeline em processadores apresenta várias vantagens:

- **Aumento da Taxa de Instruções por Ciclo**: Com o pipeline, diferentes estágios da execução de várias instruções podem ser realizados simultaneamente, aumentando a taxa de instruções processadas por ciclo.

– **Melhor Utilização de Recursos**: O pipeline permite a utilização contínua de todas as unidades do processador, minimizando os tempos ociosos.
– **Redução da Latência Média**: Embora a latência de uma única instrução possa não diminuir, a latência média para a execução de um conjunto de instruções é reduzida.
– **Escalabilidade**: O pipeline pode ser ajustado e escalado para processadores mais complexos e rápidos, permitindo a integração de técnicas avançadas como pipelining superescalar e execução fora de ordem.

Estas vantagens fazem do pipeline uma técnica essencial na arquitetura de processadores modernos, permitindo alcançar altas taxas de desempenho e eficiência.

III. Planejamento e Design da Microarquitetura

A. Especificações do Projeto

O projeto do pipeline para o processador Simple RISC foi desenvolvido com as seguintes especificações:

– **Arquitetura**: Pipeline de 5 estágios (IF, ID, EX, MEM, WB).
– **Instruções Suportadas**: Instruções aritméticas (ADD, SUB), instruções de memória (LW, SW), e instruções de desvio (BEQ, JUMP).
– **Técnicas de Otimização**: Forwarding para minimizar stalls, branch prediction para reduzir bolhas causadas por desvios.
– **Objetivo**: Melhorar a taxa de execução das instruções e otimizar o desempenho geral do processador.

B. Desenho do Pipeline em Blocos

O diagrama abaixo representa o pipeline de 5 estágios do Simple RISC, destacando os principais componentes e a interação entre eles.

C. Alocação de Recursos e Módulos

A implementação do pipeline requer a alocação eficiente de recursos e a integração de diversos módulos para garantir o funcionamento correto e eficiente do processador. Os principais recursos e módulos são:

– **Banco de Registradores**: Utilizado no estágio de decodificação para ler os operandos necessários.
– **Unidade Lógica e Aritmética (ULA)**: Executa operações aritméticas e lógicas no estágio de execução.
– **Memória de Dados**: Acessada no estágio de memória para operações de load e store.
– **Contador de Programa (PC)**: Controla a sequência de instruções a serem executadas.
– **Unidade de Controle**: Gera sinais de controle para coordenar a operação dos diferentes estágios do pipeline.
– **Técnicas de Forwarding**: Utilizadas para minimizar stalls, permitindo que dados sejam passados diretamente entre estágios do pipeline.
– **Branch Prediction**: Implementada para reduzir bolhas no pipeline causadas por instruções de desvio.

A alocação eficiente desses recursos e a correta implementação dos módulos são cruciais para maximizar o desempenho do pipeline e garantir a execução correta das instruções no processador Simple RISC.

IV. Hazards no Pipeline

Os principais desafios na implementação de um pipeline são os hazards, que podem ser de três tipos:

A. Data Hazards

Os data hazards ocorrem quando uma instrução depende do resultado de uma instrução anterior ainda em execução.

– **RAW (Read After Write)**: Leitura antes da escrita.
– **WAR (Write After Read)**: Escrita antes da leitura.
– **WAW (Write After Write)**: Escrita após escrita.

1) Forwarding: O forwarding (adiantamento) passa os resultados diretamente entre os estágios do pipeline, evitando a espera pela escrita e leitura dos registradores, mitigando os data hazards.

B. Control Hazards

Os control hazards ocorrem em instruções de desvio, como BEQ e BGT, que podem alterar o fluxo de execução.

1) Branch Prediction: O processador utiliza uma unidade de previsão de desvios de dois bits iniciada com valor falso. Para a implementação, foi utilizada uma máquina de estados, na qual é previsto “F” até o previsor errar duas vezes, momento em que o previsor troca seu “chute”. O mesmo ocorre para “V”.

C. Structural Hazards

Os structural hazards ocorrem quando múltiplas instruções competem pelo mesmo recurso.

1) Soluções: Duplicar recursos críticos ou utilizar técnicas de escalonamento para minimizar conflitos de recursos e evitar structural hazards.

Implementando estas técnicas, é possível gerenciar os hazards no pipeline, garantindo uma execução eficiente no processador Simple RISC.

V. Proposta de Otimização para o Pipeline

Como proposta de otimização para o pipeline, foi utilizado um previsor de desvios. Em um pipeline, várias instruções são processadas simultaneamente em diferentes estágios, e as instruções de desvio podem interromper esse fluxo, pois determinam quais instruções serão executadas a seguir. O previsor de desvios tenta prever se um desvio será tomado sem aguardar a avaliação completa da condição.

A previsão é baseada em um contador de 2 bits, que mede a ”confiança”na previsão. O contador pode ter quatro combinações possíveis, refletindo a confiança em um desvio ser tomado (”taken”) ou não tomado (”not taken”). Após a execução da instrução de desvio, o resultado real atualiza o contador: fortalece a previsão correta e enfraquece a previsão errada. Se a previsão estiver incorreta, as instruções executadas com base nela são descartadas e o pipeline é corrigido.

VI. Componentes do Processador

Nesta seção, apresentamos imagens e descrições dos principais componentes do processador Simple RISC, incluindo a Branch Unit, Control Unit, Data Lock, Forwarding Unit, Calculate Immediate e ULA.

A. Diagrama Geral da Arquitetura

O diagrama geral mostra a interação entre os principais componentes do processador, incluindo:

- Buffers de Pipeline (Pipeline Registers);
- Unidade de Branch (Branch Unit);
- Unidade de Forwarding (Forwarding Unit);
- Unidade de Controle de Pipeline (Pipeline Control Unit);
- Unidade de Controle de Dados (Data Hazard Control Unit);
- Unidade de Controle de Estrutura (Structural Hazard Control Unit);

E também, Unidade Lógica e Aritmética (ULA), o Banco de Registradores, a Unidade de Controle e a Memória.

1) Diagrama com e sem Previsor de Desvios: O previsor de desvios melhora o desempenho do pipeline ao tentar prever a direção dos desvios antes de sua execução completa.

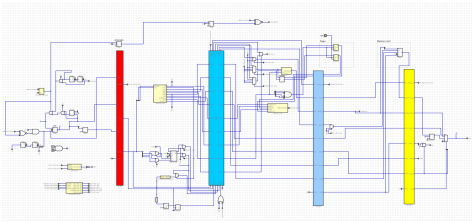

Figura 1. Diagrama com previsor de desvios.

No diagrama acima, o previsor tenta prever se um desvio será tomado, otimizando o fluxo de execução e reduzindo ciclos perdidos. No entanto, nossa implementação com o previsor apresentou problemas em um dos testes realizados.

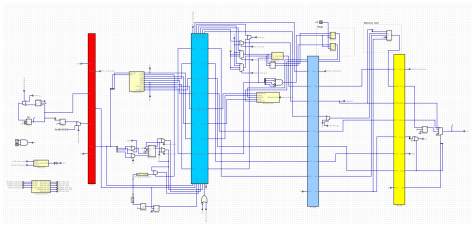

Figura 2. Diagrama sem previsor de desvios.

Sem um previsor, o pipeline deve esperar pela confirmação do desvio, o que pode causar atrasos e reduzir o desempenho.

B. Branch Unit

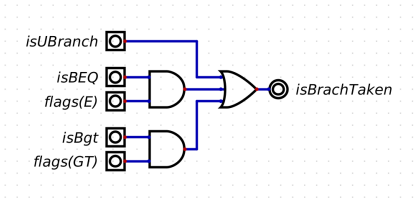

Figura 3. Diagrama da Branch Unit.

A Branch Unit é responsável pelo gerenciamento das instruções de desvio, garantindo que o processador possa alterar o fluxo de controle de acordo com as condições especificadas. Esta unidade lida com cálculos de endereço de desvio e a decisão de qual instrução buscar em seguida.

C. Control Unit

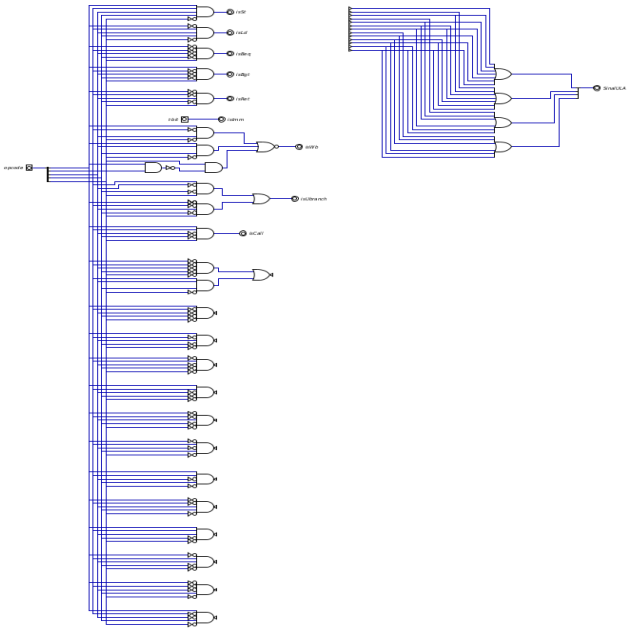

Figura 4. Diagrama da Control Unit.

A unidade de controle em um processador pipeline coordena e gerencia as operações entre os diferentes estágios do pipeline, decodificando instruções e gerando sinais de controle. Ela também lida com situações de hazard, sincroniza o pipeline e atualiza o contador de programa para garantir a execução eficiente das instruções.

D. Data Lock

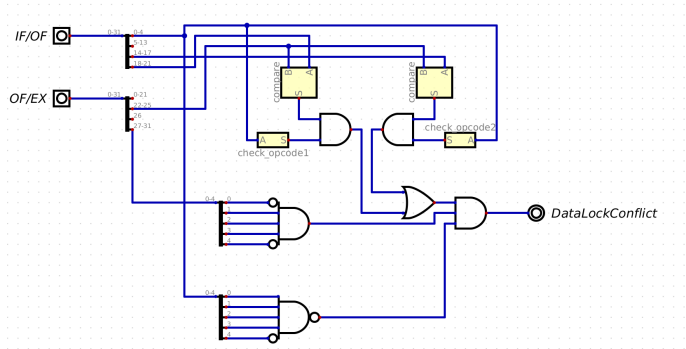

Figura 5. Diagrama do Data Lock.

O Data Lock é responsável pela proteção e sincronização dos dados durante as operações de acesso à memória. Ele assegura que múltiplos acessos à memória não causem inconsistências ou corrupções nos dados.

E. Forwarding Unit

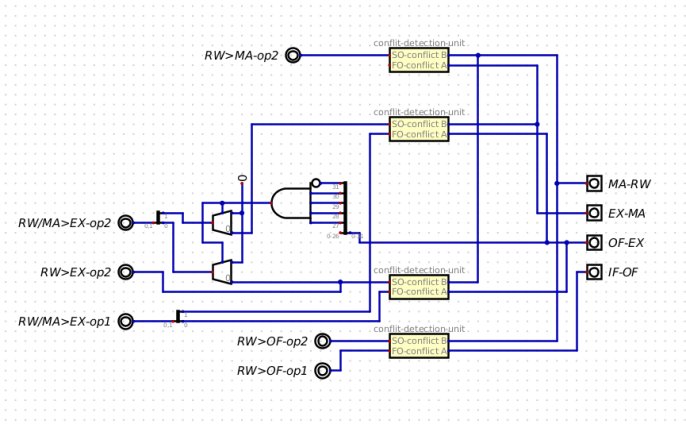

Figura 6. Diagrama da Forwarding Unit.

A Forwarding Unit minimiza stalls no pipeline permitindo que os dados produzidos em estágios anteriores sejam passados diretamente para estágios posteriores, evitando a necessidade de espera para a escrita e leitura de dados do banco de registradores.

F. Calculate Immediate

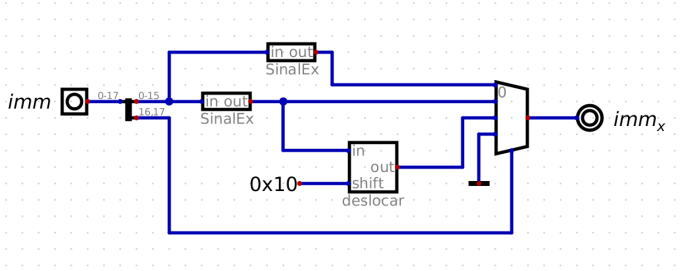

Figura 7. Diagrama do Calculate Immediate.

O módulo Calculate Immediate realiza operações de cálculo com valores imediatos incluídos nas instruções. Ele é responsável por gerar resultados baseados em valores constantes diretamente embutidos nas instruções.

G. ULA (Unidade Lógica e Aritmética)

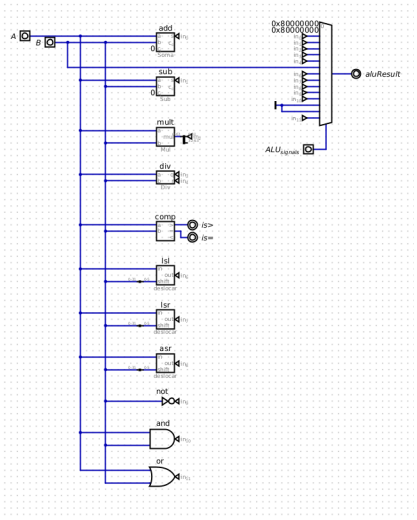

Figura 8. Diagrama da Unidade Lógica e Aritmética (ULA).

A ULA executa operações aritméticas e lógicas fundamentais, como adição, subtração, e comparações. É um dos componentes principais do pipeline e desempenha um papel central na execução das instruções.

H. Previsor de desvios

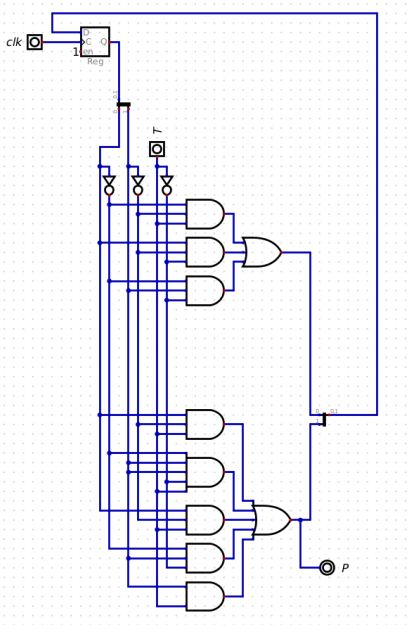

Figura 9. Diagrama do Previsor de Desvios.

VII. Simulação e Testes

Para validar a implementação do pipeline no Simple RISC, utilizamos a função de cálculo da sequência de Fibonacci como um exemplo de aplicação prática. O código de exemplo a seguir ilustra a implementação dessa função em Assembly, demonstrando a geração dos primeiros 20 números da sequência de Fibonacci. Também foi testada uma implementação do quickSort com os números 3, 7, 9, 11, 12 e 15.

```
main:
    mov r1, 0          # r1 = base address 0
    mov r2, 20         # r2 = size 20
    mov r3, 1          # r3 = first element 1
    mov r4, 1          # r4 = second element 1
    call fibonacci     # Jump to fibonacci function
    nop                # HALT equivalent

fibonacci:
    st r3, 0[r1]       # vector[0] = r3
    st r4, 4[r1]       # vector[1] = r4

    mov r5, 2          # r5 = 2

loop:
    cmp r5, r2         # Compare r5 and r2
    bgt end            # if r5 >= r2, jump to end

body:
    sub r9, r5, 1     # r9 = r5 - 1
    lsl r10, r9, 2    # r10 = r9 * 4
    add r9, r1, r10   # r9 = address of vector[i-1]
    ld r6, 0[r9]      # r6 = vector[i-1]

    sub r9, r5, 2     # r9 = r5 - 2
    lsl r10, r9, 2    # r10 = r9 * 4
    add r9, r1, r10   # r9 = address of vector[i-2]
    ld r7, 0[r9]      # r7 = vector[i-2]

    add r8, r6, r7    # r8 = r6 + r7

    lsl r9, r5, 2     # r9 = r5 * 4
    add r9, r1, r9    # r9 = address of vector[i]
    st r8, 0[r9]      # vector[i] = r8

    add r5, r5, 1     # r5 = r5 + 1
    b loop            # Jump to loop

end:
    ret               # Return from the function
```

Figura 10. Código Assembly para cálculo da sequência de Fibonacci no Simple RISC.

```
.main:
    mov r0, 0
    mov r1, 12    @ replace 12 with the number to be sorted
    st r1, 0[r0]
    mov r1, 7    @ replace 7 with the number to be sorted
    st r1, 4[r0]
    mov r1, 11  @ replace 11 with the number to be sorted
    st r1, 8[r0]
    mov r1, 9   @ replace 9 with the number to be sorted
    st r1, 12[r0]
    mov r1, 3   @ replace 3 with the number to be sorted
    st r1, 16[r0]
    mov r1, 15  @ replace 15 with the number to be sorted
    st r1, 20[r0]
    mov r2, 0          @ Starting address of the array
    mov r3, 5    @ REPLACE 5 WITH N-1, where, N is the number of numbers
          being sorted
    mul r3, r3, 4
    add r4, r2, r3  @ Ending address of the array
    call .quicksort
    .terminate:
        b .terminate
```

Figura 11. Parte 1 do Código Assembly para o QuickSort no Simple RISC.

```
.partition:
    mov r3, r4      @ address of pivot(the last element)
    sub r4, r4, 4   @ address of j (the second last element)

    ld r5, 0[r2]    @ access the value of ith element (first element)
    ld r6, 0[r3]    @ access the value of the pivot element
    ld r7, 0[r4]    @ access the value of jth element (second last
          element)
    .loop:
        cmp r2, r4  @ if i>j then swap pivot with i
        bgt .swap
        cmp r6, r5
        bgt .continue_i     @ if value of ith element < pivot, move to (i
              +1)th elemet
        cmp r7, r6
        bgt .continue_j     @ if value of jth element > pivot, move to (j
              -1)th elemet
        st r5, 0[r4]        @ swap elements and move forward
        st r7, 0[r2]
        add r2, r2, 4       @ increment i
        ld r5, 0[r2]        @ load the next ((i+1)th) value
        sub r4, r4, 4       @ decrement j
        ld r7, 0[r4]        @ load the next ((j-1)th) value
        b .loop
        .continue_i:
            add r2, r2, 4
            ld r5, 0[r2]
            b .loop
        .continue_j:
            sub r4, r4, 4
            ld r7, 0[r4]
            b .loop
    .swap:
        st r5, 0[r3]
        st r6, 0[r2]
        ret

.quicksort:
    cmp r2, r4      @ check the number of elements to be sorted
    bgt .return     @ return if no element is to be sorted
    sub sp, sp, 16      @ make the activation block for recursive calls
    st r2, 0[sp]       @ store the initial address, final address and
          return address in the stack
    st r4, 8[sp]
    st ra, 12[sp]

    call .partition
    st r2, 4[sp]    @ store the address of the pivot element received
          from the partition function in the stack

    ld r2, 0[sp]
    call .quicksort     @ sort the subarray containing elements having
          value less than the pivot

    ld r2, 4[sp]
    add r2, r2, 4       @ load the initial address of the second subarray
          to be sorted in r2
    ld r4, 8[sp]        @ load back the final address of the second
          subarray to be sorted in r4
    call .quicksort     @ sort the subarray containing elements having
          value more than the pivot

    ld ra, 12[sp]       @ retrieve the return address
    add sp, sp, 16      @ delete the activation block
    .return:
        ret
```

Figura 12. Parte 2 do Código Assembly para o QuickSort no Simple RISC.

VIII. Conclusão

A implementação do pipeline no Simple RISC demonstrou uma significativa melhoria no desempenho, reduzindo o número de ciclos necessários para calcular a sequência de Fibonacci. A utilização de técnicas como forwarding e branch prediction foi eficaz na mitigação de hazards e na otimização do fluxo de execução.

Apesar da complexidade adicional na gestão de hazards, os benefícios do pipeline são claros, evidenciando a importância dessa técnica para o aumento da eficiência e do desempenho dos processadores. Este trabalho confirma que uma abordagem cuidadosa na implementação e otimização de pipelines pode levar a avanços significativos na arquitetura de processadores.

IX. Referências

1) Sarangi, S. R. (n.d.). *Appendices*. [Livro Eletrônico]. Disponível em: https://www.cse.iitd.ac.in/~srsarangi/advbook/chapters/appendices.pdf
2) Cornell University. (2019). *CS 3410: Computer System Organization and Programming*. [Slides]. Disponível em: https://www.cs.cornell.edu/courses/cs3410/2019sp/schedule/\\slides/06-cpu-pre.pdf
3) Tandon, A. (n.d.). *Implementation of a 5-stage RISC processor*. Disponível em: https://github.com/anchittandon/Implementation-of-a-5-stage-RISC-processor/tree/master/Test-files
4) Patterson, D., e Waterman, A. (2019). *Guia Prático RISC-V*. [Livro Eletrônico]. Disponível em: http://riscvbook.com/portuguese/guia-pratico-risc-v-1.0.0.pdf