# Simulador de execução fora de ordem com scoreboard - MO401 - 2s2025

## Objetivo Geral

Neste trabalho vamos desenvolver um simulador de execução fora de ordem com a técnica de scoreboarding. O simulador deve receber como entrada dois arquivos:

1. Programa a ser executado, em assembly do RISC-V, contendo apenas instruções de um determinado subconjunto (especificado abaixo). Um exemplo de programa é:

```
fld f1, 0(x1)
fld f5, 0(x1)
fdiv f2, f4, f5
```

2. Configuração das Unidades Funcionais, indicando a quantidade de unidades de cada tipo, e o número de ciclos necessários para completar a execução naquela unidade. Um exemplo de configuração é:

```
int 2 1
mult 2 4
add 1 2
div 1 10
```

Este exemplo indica que há duas unidades de processamento de operações inteiras (int) que levam 1 ciclo de execução, duas unidades para multiplicação em ponto flutuante (mult) que levam 4 ciclos, uma unidade de soma em ponto flutuante que leva 2 ciclos, e uma unidade de divisão em ponto flutuante que leva 10 ciclos para completar sua execução.

A saída do simulador deverá ser uma tabela indicando o número do ciclo em que cada instrução cada uma das etapas de sua execução com scoreboard: Issue, Read (Leitura de Operandos), Execute (completar execução), Write (escrita de resultados no banco de registradores).

| Instruction/Cicle | Issue | Read | Execute | Write |
|---|---|---|---|---|
| fld f1, 0(x1) | 1 | 2 | 3 | 4 |
| fld f5, 0(x1) | 2 | 3 | 4 | 5 |
| fdiv f2, f4, f5 | 3 | 6 | 16 | 17 |

## Instruções suportadas

O simulador deverá aceitar programas com as seguintes instruções RISC-V: `fld` (usa unidade de inteiros), `fsd` (usa unidade de inteiros), `fadd` (usa somador de ponto flutuante), `fsub` (usa somador de ponto flutuante), `fmul` (usa multiplicador de ponto flutuante), e `fdiv` (usa divisor de ponto flutuante). Assuma que

as instruções de acesso à memória sempre completam dentro do número de ciclos indicado para a unidade de instruções inteiras. Assuma que é possível fazer escrita de múltiplos resultados em um mesmo ciclo.

## Estruturas e simulação

Para a implementação do simulador, será necessária a criação de estruturas de dados correspondentes às tabelas do scoreboard:

1. Status das Instruções, indicando o progresso na instrução nas etapas do scoreboard (você pode usar esta estrutura para gerar o resultado final)
2. Status dos registradores, indicando para cada registrador inteiro x0-x31 e ponto flutuante f0-f31 se/qual unidade funcional irá produzir o valor para atualizar este registrador
3. Status das unidades funcionais, incluindo campos `busy`; `operation`; números dos registradores destino e fontes `Fi`, `Fj`, `Fk`; unidades funcionais `Qj`, `Qk` que irão produzir valores de `Fj`, `Fk`, e indicadores `Rj`, `Rk` de que os valores estão prontos para leitura.

A cada ciclo, as condições de issue, leitura, execução, e escrita devem ser verificadas para as instruções relevantes. As estruturas de dados devem ser então atualizadas conforme o algoritmo de scoreboarding. A tabela C.58 do livro texto sumariza estas condições e atualizações.

Não é necessário realizar a implementação funcional das instruções; basta identificar as dependências e fluxo de dados para obter a simulação de ciclos de execução.

## Entradas e testes

Para a seguinte configuração:

```
int 1 1
mult 2 4
add 1 2
div 1 10
```

e o seguinte código de entrada:

```
fld  f1, 100(x7)
fmul f2, f2, f4
fadd f2, f1, f3
fld  f9, 0(x3)
fdiv f3, f1, f7
fsub f6, f3, f4
fmul f7, f1, f2
fadd f4, f5, f2
fsd  f1, 50(x11)
```

o resultado esperado é:

| Instruction/Cicle | Issue | Read | Execute | Write |
|---|---|---|---|---|

| Instruction/Cicle | Issue | Read | Execute | Write |
|---|---|---|---|---|
| fld f1, 100(x7) | 1 | 2 | 3 | 4 |
| fmul f2, f2, f4 | 2 | 3 | 7 | 8 |
| fadd f2, f1, f3 | 9 | 10 | 12 | 13 |
| fld f9, 0(x3) | 10 | 11 | 12 | 13 |
| fdiv f3, f1, f7 | 11 | 12 | 22 | 23 |
| fsub f6, f3, f4 | 14 | 24 | 26 | 27 |
| fmul f7, f1, f2 | 15 | 16 | 20 | 21 |
| fadd f4, f5, f2 | 28 | 29 | 31 | 32 |
| fsd f1, 50(x11) | 29 | 30 | 31 | 32 |

## Entrega e aproveitamento

Teste o seu simulador com diferentes entradas e configurações de unidades funcionais. Inclua seus testes na entrega. Para entrega, inclua seu código, instruções mínimas para build (se necessário) e execução, juntamente com suas entradas de teste e respectivas saídas. Inclua todos os arquivos em um único .zip e envie pelo Classroom.

O trabalho entregue até a data limite dará bônus de até 2 pontos na nota total da prova 1, de acordo com a nota do trabalho. Só serão consideradas submissões com soluções que implementem as regras completas de scoreboarding (ainda que contendo eventuais pequenos erros).

## Parser

O código abaixo implementa um parser (provavelmente quebrado) das instruções especificadas. Use por conta e risco

```
"""
Zero Effort Parser (rv-zep)
This was made in about 15 minutes with the help of Chat-GPT.
I don't trust it. You shoudn't trust it either.
"""


# Define opcode constants
OPCODES = {
    'fld': 0,
    'fsd': 1,
    'fadd': 2,
    'fsub': 3,
    'fmul': 4,
    'fdiv': 5
}
```

```
# Define register prefix constants
REG_PREFIXES = {
    'x': 'int',
    'f': 'float'
}

def parse_file(filename):
    instructions = []
    with open(filename, 'r') as f:
        for line in f:
            fields = line.strip().replace(',', ' ').split()
            opcode = fields[0].lower()
            if opcode not in OPCODES:
                raise ValueError(f'Invalid opcode: {opcode}')
            opcode = OPCODES[opcode]
            rs1, rs2, rd, imm = 0, 0, 0, None  # Set imm to None by default
            rs1_type, rs2_type, rd_type = None, None, None
            if opcode == 0:  # fld format: "instruction rd imm(rs1)"
                rd = int(fields[1][1:])
                rd_type = REG_PREFIXES[fields[1][0].lower()]
                rs1_imm = fields[2].split('(')
                imm = int(rs1_imm[0])
                rs1 = int(rs1_imm[1][1:-1])
                rs1_type = REG_PREFIXES[rs1_imm[1][0:1].lower()]
            elif opcode == 1:  # fsd format: "instruction rs2 imm(rs1)"
                rs2 = int(fields[1][1:])
                rs2_type = REG_PREFIXES[fields[1][0].lower()]
                rs1_imm = fields[2].split('(')
                imm = int(rs1_imm[0])
                rs1 = int(rs1_imm[1][1:-1])
                rs1_type = REG_PREFIXES[rs1_imm[1][0:1].lower()]
            else:  # Other instructions format: "instruction rd rs1 rs2"
                rd = int(fields[1][1:])
                rd_type = REG_PREFIXES[fields[1][0].lower()]
                rs1 = int(fields[2][1:])
                rs1_type = REG_PREFIXES[fields[2][0].lower()]
                if len(fields) > 3:
                    rs2 = int(fields[3][1:])
                    rs2_type = REG_PREFIXES[fields[3][0].lower()]
                else:
                    rs2 = 0
                    rs2_type = None
            instructions.append({
                'opcode': opcode,
                'rs1': rs1,
                'rs1_type': rs1_type,
                'rs2': rs2,
                'rs2_type': rs2_type,
                'rd': rd,
                'rd_type': rd_type,
                'imm': imm
            })
    return instructions
```

```
# Example usage
instructions = parse_file('example.s')
for i in instructions:
    print(i)
```