

UNIVERSIDADE FEDERAL DE LAVRAS
Departamento de Ciência da Computação
Bacharelado/Mestrado em Ciência da Computação
Disciplina: **Arquitetura de Computadores II - GCC123/PCC507**
Trabalho Prático 1 – 2º Semestre de 2025
Professor: **Luiz Henrique A. Correia**
Data de entrega: 21/11/2025 – Valor: 15 pontos

Simulador Funcional do Processador UFLA-RISC

1 Introdução

Este trabalho visa a implementação de um simulador funcional para o processador RISC de 32 bits UFLA-RISC. Este processador possui 32 registradores de uso geral e 22 instruções. Outras instruções devem ser adicionadas pelos alunos para facilitar o desenvolvimento e execução de códigos, no **mínimo 8 instruções** que deverão ser compartilhadas para todos os grupos.

O desenvolvimento deste trabalho é a base para uma implementação de um simulador funcional de um processador, e contribui para o aprendizado dos conceitos empregados na operação de um processador.

Um dos requisitos fundamentais para se entender como implementar o simulador funcional do processador monociclo é a compreensão do termo “funcional”. Uma descrição funcional divide o processador nos blocos que existirão em uma implementação real. Portanto, o processador deverá conter quatro rotinas principais, uma para cada estágio a ser executado para cada instrução. É necessário que seja criada uma **unidade de controle** para a interação das unidades funcionais.

Dessa forma, cada instrução executada no UFLA-RISC possui quatro estágios para completar **IF, ID, EX/MEM e WB**:

- **IF (Instruction Fetch)**: busca da instrução na memória e armazenamento no registrador de instruções (IR); incrementa-se o contador de programa (PC) para a busca da próxima instrução.
- **ID (Instruction Decode)**: decodificação da instrução e busca dos operandos no banco de registradores.
- **EX/MEM (Execute and Memory)**: execução da instrução, definição dos códigos de condição, cálculo de endereço efetivo, resolução de branch.
- **WB (Write Back)**: escrita dos resultados no banco de registradores.

2 Implementação do Simulador

O processador UFLA-RISC deverá as seguintes características:

- i. Um barramento de dados de 32 bits e um barramento de endereço de 16 bits;
- ii. Contém 32 registradores de uso geral de 32 bits de largura;

- iii. instruções de 3 operandos;
- iv. A memória é endereçada à palavra, ou seja, cada endereço de memória deve se referir a **quatro bytes**. No total, o processador deverá possuir 64k (2^{16}) endereços. Então, a memória total do processador será de: 256KB (64K endereços x 4 bytes).
- v. O programa deve receber como entrada um arquivo binário com linhas de 32 bits que representam a codificação das instruções do UFLA-RISC. Para tal, é necessário o desenvolvimento de um **Interpretador de Instruções** no qual dado um conjunto de instruções em linguagem de montagem do UFLA-RISC, este seja **convertido para linguagem de máquina, em binário**.

O simulador pode ser implementado em qualquer linguagem, preferencialmente, Java, C++, PHP ou Python. O conjunto **mínimo** de instruções que devem ser usadas no trabalho são apresentadas a seguir.

2.1 Conjunto de Instruções

O conjunto de instruções inclui:

- Instruções aritméticas (ALU);
- Instruções com constantes e acesso à memória;
- Instruções de transferência de controle.

2.1.1 Instruções ALU

O UFLA-RISC possui as seguintes instruções de ALU:

Opcode	Operação
00000001	Adição de inteiros
00000010	Subtração de inteiros
00000011	Zero
00000100	Xor
00000101	Or
00000110	Not
00000111	And
00001000	Shift aritmético para a esquerda
00001001	Shift aritmético para a direita
00001010	Shift lógico à esquerda
00001011	Shift lógico à direita
00001100	Copia

Todas as instruções de ALU alteram os valores dos **flags do processador**: **neg**, **zero**, **carry** e **overflow**. As instruções lógicas, como **and** e **or**, zeram os flags **overflow** e **carry**. Veja a descrição das instruções no Apêndice B para maiores detalhes.

2.1.2 Constantes e Acesso à Memória

O processador UFLA-RISC possui somente instruções para a carga de constantes de 16 bits, para carga de uma posição de memória específica e para armazenar um valor numa posição de memória específica.

Opcode	Operação
00001110	Carrega constante de 16 bits nos 2 bytes mais significativos
00001111	Carrega constante de 16 bits nos 2 bytes menos significativos
00010000	Carrega conteúdo de memória em registrador
00010001	Armazena conteúdo de registrador na memória

2.1.3 Transferência de Controle

O processador UFLA-RISC possui cinco instruções para a transferência de controle, codificadas de três maneiras diferentes. A primeira codificação é utilizada para instruções de desvios condicionais, a segunda codificação é utilizada para a instrução de desvios incondicionais e a terceira codificação também é utilizada para instruções de desvios incondicionais, porém o endereço de desvio é especificado por um registrador. As cinco instruções para transferência de controle são: `jump and link`, `jump register`, `jump se igual`, `jump se diferente` e `jump incondicional`.

Na instrução `jump and link`, o próximo PC ($PC + 1$) deve ser armazenado no registrador `r31` e o endereço do desvio armazenado em PC.

Na instrução `jump register`, a única operação a ser feita é armazenar o conteúdo do registrador `rc` no registrador que armazena o valor de PC.

Nas instruções de `Jump Condicional` (`jump se igual` ou `jump se diferente`) o endereço da próxima instrução a ser executada, caso o `Jump` não seja realizado, é o valor do PC incrementado de 1. Assim, no ciclo `EX/MEM`, esse endereço pode ser obtido diretamente do registrador que armazena o valor de PC, pois nesse ciclo o PC já foi incrementado de um. Se o `Jump` for tomado, o fluxo de instruções deve ser transferido para o endereço de memória especificado na instrução.

Na instrução `jump incondicional`, a única operação a ser feita é armazenar o endereço do desvio no registrador que armazena o valor de PC.

Instruções: `jump and link`, `jump register`, `jump se igual`, `jump se diferente`, `jump incondicional`

2.1.4 Condição de Término

Uma instrução onde todos os 32 bits são iguais a 1 será utilizada para indicar a instrução **HALT** (parada do sistema).

3 Documentação

A documentação será de grande importância na avaliação de todo o trabalho. Não é necessário que seja extensa, mas é imprescindível que seja completa, contendo:

- Resumo da máquina simulada;
- Decisões de implementação;
- Instruções projetadas pelo grupo com justificativas;
- Tutorial de uso do interpretador e simulador;
- Estruturas de representação do hardware (*Datapath*);
- Código fonte do simulador;
- Testes realizados.

3.1 Testes

Os testes pelo qual o processador passa ajuda no desenvolvimento da sua correção e robustez. Como o objetivo primeiro do projeto é prezar pela sua correção, é esperado que se faça uso de uma bateria de testes para auxílio na depuração. Como sugestão de testes úteis, temos:

- **Testes massivos:** testam o maior número de combinações de instruções possíveis;
- **Programas reais:** simulam situações práticas.

3.2 Entrega

No Campus Virtual devem ser enviados os arquivos contendo:

- Arquivo compactado contendo o código do uRISC-UFLA;
- Arquivo contendo o nome dos integrantes do grupo (sem número de matrícula ou enumeração, texto simples).
- Arquivos de testes.
- Documentação.
- Link do **Github** onde será feito o desenvolvimento (ver Apêndice C).

O trabalho será apresentado ao professor em data a ser estabelecida com os grupos após a entrega do código e documentação. Nesta etapa será verificada a contribuição de cada membro do grupo por meio do hisgitory do Github.

Apêndice A: Formato de Entrada/Saída do Simulador

A.1 Introdução

Esta seção define como deve ser o formato do arquivo de instruções para o processador UFLARISC e a saída do simulador. Convém salientar que o formato, que será descrito a seguir, deve ser adotado obrigatoriamente por todos os grupos.

Como especificado, a memória do processador possui 16 bits de endereçamento, sendo endereçada por palavra. Ou seja, a memória pode armazenar até 65.536 palavras de 32 bits cada.

A.2 Descrição do formato

O formato do arquivo de instruções para o processador UFLA-RISC é bem simples. Podemos defini-lo através de apenas três características:

- O arquivo, em modo texto, deve conter apenas uma instrução por linha.
- As instruções devem estar codificadas em binário.
- A palavra-chave `address` pode ser usada para definir a partir de qual posição de memória as instruções devem ser armazenadas.

A sintaxe é a seguinte:

`address end`, onde `end` é um endereço de memória codificado em binário. Um pequeno exemplo ilustra a utilidade desta palavra-chave.

Observar que, se a diretiva `address` não for usada para definir onde as instruções devem ser armazenadas, será considerado, por convenção, que as instruções serão colocadas a partir da posição zero de memória.

A.3 Exemplo de um arquivo de instruções

A seguir é apresentado um arquivo de instruções que segue o formato descrito no item 2.

```
address 0
00011000100000000000110001000000
0001100010000011000110001000000
0001100010000010000110001000000
0001100010000011000110001000000
```

```
address 1000001
00011000100000000000110001000000
0001100010000011000110001000000
0001100010000010000110001000000
0001100010000011000110001000000
```

Veja o conteúdo de memória após o processador carregar o arquivo de instruções acima:

Endereço de Memória	Conteúdo
0 (Posição 0)	0001100010000000 0001100010000000
10 (Posição 1)	0001100010000011 0001100010000000
10 (Posição 2)	0001100010000010 0001100010000000
11 (Posição 3)	0001100010000011 0001100010000000
1000001 (Posição 65)	0001100010000000 0011000100000000
1000010 (Posição 66)	00011000100000110 0011000100000000
1000011 (Posição 67)	00011000100000100 0011000100000000
1000100 (Posição 68)	00011000100000110 0011000100000000

A.4 Formato da Saída

Para verificar a corretude do simulador, o mesmo deverá apresentar na saída as modificações ocorridas nos registradores e nas posições de memória a cada ciclo de clock.

Considere o exemplo abaixo onde é executada uma instrução de carga do registrador.

LOAD R1,A;

- No primeiro ciclo (IF) carrega-se a instrução no registrador IR, logo após este ciclo IR deve ter como conteúdo a instrução LOAD R1,A e PC deve ter como conteúdo a posição de memória da próxima instrução a ser executada.
- No segundo ciclo (ID) a instrução é decodificada e não há alterações nos conteúdos de registradores/memória.
- No terceiro ciclo (EX/MEM) a instrução é executada.
- No quarto ciclo (WB) a instrução LOAD escreve o resultado em R1, logo após este ciclo o registrador R1 deve ter o valor da posição A da memória.

Agora considere um exemplo onde é executada uma instrução ADD

ADD R3, R2, R1

- No primeiro ciclo (IF) carrega-se a instrução no registrador IR, logo após este ciclo IR deve ter como conteúdo a instrução ADD R3,R2,R1 e PC devem ter como conteúdo a posição de memória da próxima instrução a ser executada.
- No segundo ciclo (ID) a instrução é decodificada e os registradores R1 e R2 são lidos, logo neste ciclo não há alterações nos conteúdos de registradores/memória.
- No terceiro ciclo (EX/MEM) a instrução é executada.
- No quarto ciclo (WB) a instrução ADD escreve o resultado em R3, logo após este ciclo o registrador R3 deve ter como valor a soma dos conteúdos dos registradores R1 e R2.

Apêndice B: Descrição detalhada das instruções

B.1 SOMA INTEIRA

`add rc, ra, rb`

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00000001	<code>add r3, r2, r1</code>	$r3 = r2 + r1$
ra	23–16	00000010		
rb	15–8	00000001		
rc	7–0	00000011		

Descrição: Soma (aritmética) o conteúdo de ra e de rb e coloca o resultado em rc.

Flags afetados: neg, zero, carry e overflow.

B.2 SUBTRAÇÃO INTEIRA

`sub rc, ra, rb`

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00000010	<code>sub r3, r2, r1</code>	$r3 = r2 - r1$
ra	23–16	00000010		
rb	15–8	00000001		
rc	7–0	00000011		

Descrição: Subtrai o conteúdo de rb do conteúdo de ra e coloca o resultado em rc.

Flags afetados: neg, zero, carry e overflow.

B.3 ZERA

`zeros rc`

Campo	Bits	Valor
OPCODE	31–24	00000011
rc	7–0	00000001

Descrição: Zera o conteúdo de rc.

Flags afetados: neg = 0, zero = 1, carry = 0, overflow = 0.

B.4 XOR LÓGICO

`xor rc, ra, rb`

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00000100	<code>xor r3, r2, r7</code>	$r3 = r2 \wedge r7$
ra	23–16	00000010		
rb	15–8	00000111		
rc	7–0	00000011		

Descrição: Efetua xor lógico bit a bit de ra e rb e coloca resultado em rc.

Flags afetados: neg, zero, carry = 0, overflow = 0.

B.5 OR LÓGICO

or rc, ra, rb

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00000101	or r3, r2, r7	r3 = r2 r7
ra	23–16	00000010		
rb	15–8	00000111		
rc	7–0	00000011		

Descrição: Efetua or lógico bit a bit de ra e rb e coloca resultado em rc.
Flags afetados: neg, zero, carry = 0, overflow = 0.

B.6 NOT ra

passnota rc, ra

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00000110	passnota r4, r3	rc = !ra
ra	23–16	00000011		
rc	7–0	00000100		

Descrição: Faz conteúdo de rc valer o complemento do conteúdo de ra.

Flags afetados: neg, zero, carry = 0, overflow = 0.

B.7 AND LÓGICO

and rc, ra, rb

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00000111	and r3, r2, r7	r3 = r2 & r7
ra	23–16	00000010		
rb	15–8	00000111		
rc	7–0	00000011		

Descrição: Efetua and lógico bit a bit de ra e rb e coloca resultado em rc.

Flags afetados: neg, zero, carry = 0, overflow = 0.

B.8 SHIFT ARITMÉTICO PARA A ESQUERDA asl rc, ra, rb

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00001000	asl r3, r2, r5	rc = ra << rb
ra	23–16	00000010		
rb	15–8	00000101		
rc	7–0	00000011		

Descrição: Coloca cada bit ra_j em rc_{i+1} e preenche com 0 a posição rc_j .

O registrador rb indica quantos bits serão deslocados.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

B.9 SHIFT ARITMÉTICO PARA A DIREITA asr rc, ra, rb

Campo	Bits	Código	Exemplo	Operação
OPCODE	31–24	00001001	asr r3, r2, r5	$rc = ra \gg rb$
ra	23–16	00000010		
rb	15–8	00000101		
rc	7–0	00000011		

Descrição: Coloca cada bit ra_i em rc_{i-1} e preenche com o valor do bit ra_{31} as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

B.10 SHIFT LÓGICO PARA A ESQUERDA lsl rc, ra, rb

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00001010	lsl r3, r2, r1	$rc = ra \ll rb$
ra	23–16	00000010		
rb	15–8	00000001		
rc	7–0	00000011		

Descrição: Coloca cada bit ra_i em rc_{i+1} e preenche com 0 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados.

Flags afetados: neg, zero, carry = 0, overflow = 0.

B.11 SHIFT LÓGICO PARA A DIREITA lsr rc, ra, rb

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00001011	lsr r3, r2, r1	$rc = ra \gg rb$
ra	23–16	00000010		
rb	15–8	00000001		
rc	7–0	00000011		

Descrição: Coloca cada bit ra_i em rc_{i-1} e preenche com 0 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados.

Flags afetados: neg, zero, carry = 0, overflow = 0.

B.12 COPIA ra passa rc, ra

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00001100	passa r4, r3	$rc = ra$
ra	23–16	00000011		
rc	7–0	00000100		

Descrição: Faz conteúdo de rc igual ao conteúdo de ra.

Flags afetados: neg, zero, carry = 0, overflow = 0.

B.13 CARREGA CONSTANTE NOS 2 BYTES MAIS SIGNIFICATIVOS

`lcl rc, Const16`

Campo	Bits	Exemplo	Operação
OPCODE	31–24	<code>lcl r3, Const16</code>	$rc = (Const16 \ll 16) (rc \& 0x0000ffff)$

Descrição: Carrega nos 2 bytes mais significativos de rc o conteúdo de Const16.
Flags afetados: Nenhum.

B.14 CARREGA CONSTANTE NOS 2 BYTES MENOS SIGNIFICATIVOS

`lcl rc, Const16`

Campo	Bits	Exemplo	Operação
OPCODE	31–24	<code>lcl r3, Const16</code>	$rc = Const16 (rc \& 0xffff0000)$

Descrição: Carrega nos 2 bytes menos significativos de rc o conteúdo de Const16.
Flags afetados: Nenhum.

B.15 LOAD WORD

`load rc, ra`

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00010000	<code>load r3, r6</code>	$rc = \text{memória}[ra]$

Descrição: Carrega no registrador rc o conteúdo da memória endereçada pelo registrador ra.
Flags afetados: Nenhum.

B.16 STORE WORD

`store rc, ra`

Campo	Bits	Código	Exemplo	Operação
OPCODE	31–24	00010001	<code>store r3, r6</code>	$\text{memória}[rc] = ra$

Descrição: Carrega na posição da memória endereçada pelo registrador rc o conteúdo do registrador ra.
Flags afetados: Nenhum.

B.17 JUMP AND LINK

jal end

Campo	Bits	Operação
OPCODE	31–24	jal 100
Endereço	23–0	r31 = PC; PC = end

Descrição: Realiza chamadas a procedimentos guardando o endereço do PC atual no registrador r31 (para o retorno após o procedimento) e colocando em PC o valor do endereço do desvio (primeira instrução do procedimento).

Flags afetados: Nenhum.

B.18 JUMP REGISTER

jr rc

Campo	Bits	Descrição	Exemplo	Operação
OPCODE	31–24	00010011	jr r31	PC = r31
ra	23–16	—		
rb	15–8	—		
rc	7–0	00011111		

Descrição: Armazena o conteúdo do registrador rc em PC.

Flags afetados: Nenhum.

B.19 JUMP SE IGUAL

beq ra, rb, end

Campo	Bits	Código	Exemplo	Operação
OPCODE	31–24	00010100	beq r7, r8, 34	PC = end (se ra == rb)
ra	23–16	00000111		
rb	15–8	00001000		
end	7–0	00100010		

Descrição: Realiza desvio de fluxo condicional. Compara o conteúdo dos registradores ra e rb e se forem iguais o registrador PC recebe o endereço de memória para o qual o fluxo de execução deve ser transferido.

Flags afetados: Nenhum.

B.20 JUMP SE DIFERENTE

bne ra, rb, end

Campo	Bits	Código	Exemplo	Operação
OPCODE	31–24	00010101	bne r7, r8, 34	PC = end (se ra != rb)
ra	23–16	00000111		
rb	15–8	00001000		
end	7–0	00100010		

Descrição: Realiza desvio de fluxo condicional. Compara o conteúdo dos registradores ra e rb e se forem diferentes o registrador PC recebe o endereço de memória para o qual o fluxo de execução deve ser transferido.

Flags afetados: Nenhum.

B.21 JUMP INCONDICIONAL

j Destino

Campo	Bits	Exemplo	Operação
OPCODE	31–24	00010110	PC = Endereço

Endereço 23–0 j loop

Descrição: Realiza desvio de fluxo incondicional. O endereço da próxima instrução a ser executada será o endereço de memória presente na instrução.

Flags afetados: Nenhum.

Entrega

Apêndice C: Organização do Projeto no GitHub

C.1 Instruções Gerais

Todos os grupos devem utilizar o GitHub como plataforma de versionamento e colaboração para o desenvolvimento do simulador funcional do processador UFLA-RISC. A seguir estão as diretrizes obrigatórias para organização, rastreabilidade e entrega do trabalho.

- Criar um repositório com o nome: `ufla-risc-simulador-nome-do-grupo`.
- Utilizar branches individuais para cada membro.
- Realizar commits frequentes e com mensagens claras.
- Utilizar Pull Requests (PRs) para integrar código à branch principal.
- Criar Issues para cada tarefa e vinculá-las aos PRs.

C.2 Estrutura Recomendada de Diretórios

```
ufla-risc-simulador/
|- src/
|   |- interpretador/
|   |- simulador/
|   \- testes/
|- binarios/
|- exemplos/
|- docs/
|- README.md
\-.gitignore
```

C.3 Modelo de README.md

```
# Simulador Funcional do Processador UFLA-RISC
```

Este projeto implementa um simulador funcional para o processador RISC de 32 bits UFLA-RISC.

```
## Integrantes do Grupo
```

- Nome 1
- Nome 2
- Nome 3

```
## Como Executar
```

1. Clone o repositório:

```
git clone https://github.com/usuario/ufla-risc-simulador.git
```

2. Compile o simulador:

```
cd src/simulador
python main.py
```

3. Execute com um arquivo binário:

```
python main.py binarios/programa.bin
```

```
## Testes
```

```
python -m unittest discover
```

```
## Licença
```

Projeto acadêmico sem licença comercial.