UNIVERSIDADE FEDERAL DO PARANÁ
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
ARQUITETURA DE COMPUTADORES

PEDRO TAKEO SHIMA
PIETRO COMIN

IMPLEMENTAÇÃO DE UMA ISA REDUX-V

Marco Zanata

CURITIBA
2025

SUMÁRIO

1 INTRODUÇÃO

Este relatório tem como objetivo descrever a implementação das arquiteturas Sagui, nas versões Vetorial e VLIW, conforme especificado no enunciado do trabalho. O projeto foi realizado em dupla e envolveu a elaboração dos datapaths, ULAs, a definição dos sinais de controle e a implementação prática no Logisim Evolution.

2 ARQUITETURA VLIW

2.1 Visão Geral e Datapath

A arquitetura VLIW consiste em um processador que recebe várias instruções simultaneamente. O objetivo dessa implementação é tentar aumentar o número de instruções por ciclo (IPC), podendo processar grupos de instruções independentes por ciclo de clock.

O fluxo das operações implementado para a arquitetura VLIW e suas operações na ULA segue o planejamento dos diagramas abaixo:

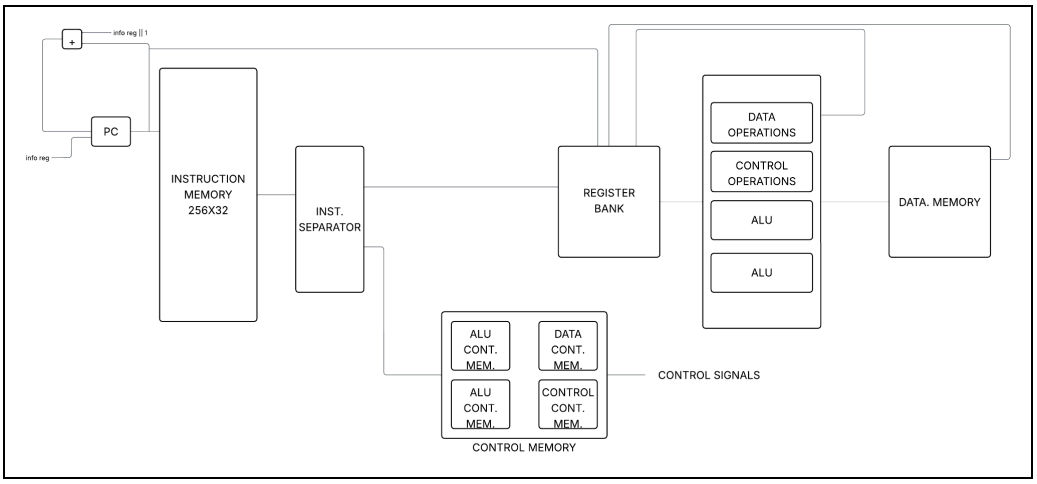

Diagrama 1: Fluxo de operações do processador VLIW

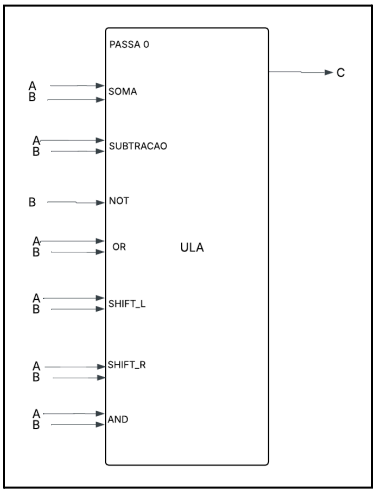

Diagrama 2: Operações da ULA

Como a implementação do processador teria um tamanho de instrução maior do que o tamanho dos dados, foi utilizado um sistema Harvard para a memória, separando a memória de dados e a memória de instruções.

2.2 Formato de Instruções e Unidade de Controle

As instruções no processador desenvolvido têm tamanho de 32 bits mas são separadas pelo componente “INSTRUCTION HANDLER”, que as separa em quatro instruções de 8 bits. A definição de quais instruções podem ser agrupadas para evitar problemas relacionados a dependências é o compilador, não o processador. Nos quatro slots de instruções temos:

- 1 slot para operações de DADOS
- 1 slot para operações de CONTROLE
- 2 slots para operações da ULA

A arquitetura tem dois possíveis formatos de instruções, sendo eles:

TIPO I

OPCODE	Imm

TIPO R

OPCODE	RA	RB

As operações da ULA são somente do tipo R, porém, as de dados e controle podem ser de ambos os tipos.

2.3 Sinais de Controle

A memória de controle dessa implementação é separada em três componentes diferentes, providenciando um sistema mais modularizado para cada tipo de operação e oferecendo independência entre as instruções.

Sinais de Controle (ULA)

OPCODE	MNEMONICO	ALUOP	REG_W
1000	ADD	001	1
1001	SUB	010	1
1010	AND	111	1
1011	OR	100	1
1100	NOT	011	1
1101	SLR	101	1
1110	SRR	110	1

1111	NOP	000	0

Sinais de Controle (CONTROLE)

OPCODE	MNEMONICO	REG_PC	IMM_PC	SRC_IMM
0000	BRZR	1	0	0
0001	BRZI	0	1	1
0010	JR	1	0	0
1111	NOP	0	0	0

Sinais de Controle (DADOS)

OPCODE	MNEMONICO	SVPC	MEM_T_REG	MEM_W	H_or_L	SRC_IMM	REG_W
0011	SVPC	1	0	0	0	0	1
0100	LD	0	1	0	0	0	1
0101	ST	0	0	1	0	0	0
0110	MOVH	0	0	0	0	1	1
0111	MOVL	0	0	0	1	1	1
1111	NOP	0	0	0	0	0	0

2.4 Novas Instruções

Como explicitado no enunciado, foram criadas duas novas instruções para a ISA VLIW. O principal foco na criação das novas instruções era aumentar a modularidade do programa para que houvesse menos volatilidade quando fosse necessário editá-lo. As novas funções são:

- SAVE PC (SVPC): Salva o valor de PC em um registrador especificado. Essa instrução facilita o cálculo de endereços de pulo para operações de JUMP e BRANCH

- AND (AND): Para a instrução lógica livre, foi implementada uma operação de AND. Apesar da simplicidade, pode ser útil em arquiteturas com tamanho

limitado de instrução, podendo criar números maiores de forma similar às instruções MOVL e MOVH.

3 ARQUITETURA VETORIAL

3.1 Visão e ideia geral

Diferente da arquitetura VLIW, a vetorial consiste em um processador com múltiplas unidades de processamento (processing elements), uma escalar e quatro vetoriais; assim sendo definido os dois grandes blocos do processador, sendo eles escalar e vetorial.

O processador pode ser melhor compreendido a partir dessas duas áreas, onde somente uma atua por instrução, que equivale a um ciclo do clock. O prefixo s. ou v. das instruções determina qual delas irá atuar, enquanto a outra recebe NOP, anulando os sinais de controle da não atuante.

3.2 Datapath

O seguinte diagrama representa, de forma geral, o fluxo de dados que é percorrido no processador vetorial. Vale ressaltar que este é simplificado e sua forma original contém circuitos lógicos adicionais.

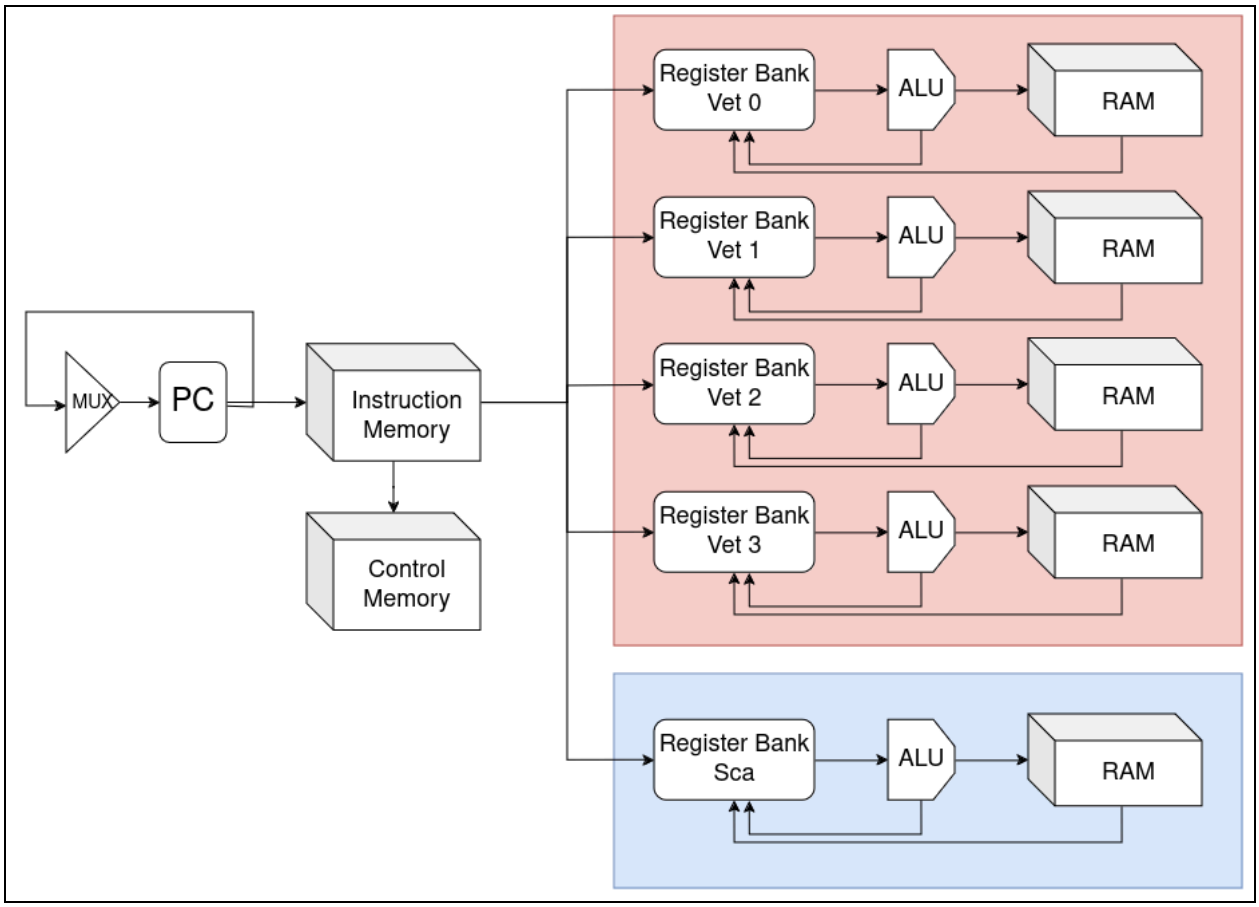

Diagrama 3: Fluxo de operações do processador vetorial

É importante destacar os dois segmentos principais do processador, o vermelho que representa a parte vetorial e o azul que é a parte escalar do mesmo. O componente que cuida de qual deles receberá a instrução enquanto o outro NOP, é a memória de controle.

A seguir segue um fluxograma da implementação da ULA (Unidade Lógica Aritmética), que foi projetada contendo somente quatro instruções: adição, subtração, AND e shift; contudo a implementação principal foi projetada para ter um bit de seleção a mais no MUX, facilitando a adição de mais quatro novas funcionalidades na ULA, por instância um XOR ou multiplicação.

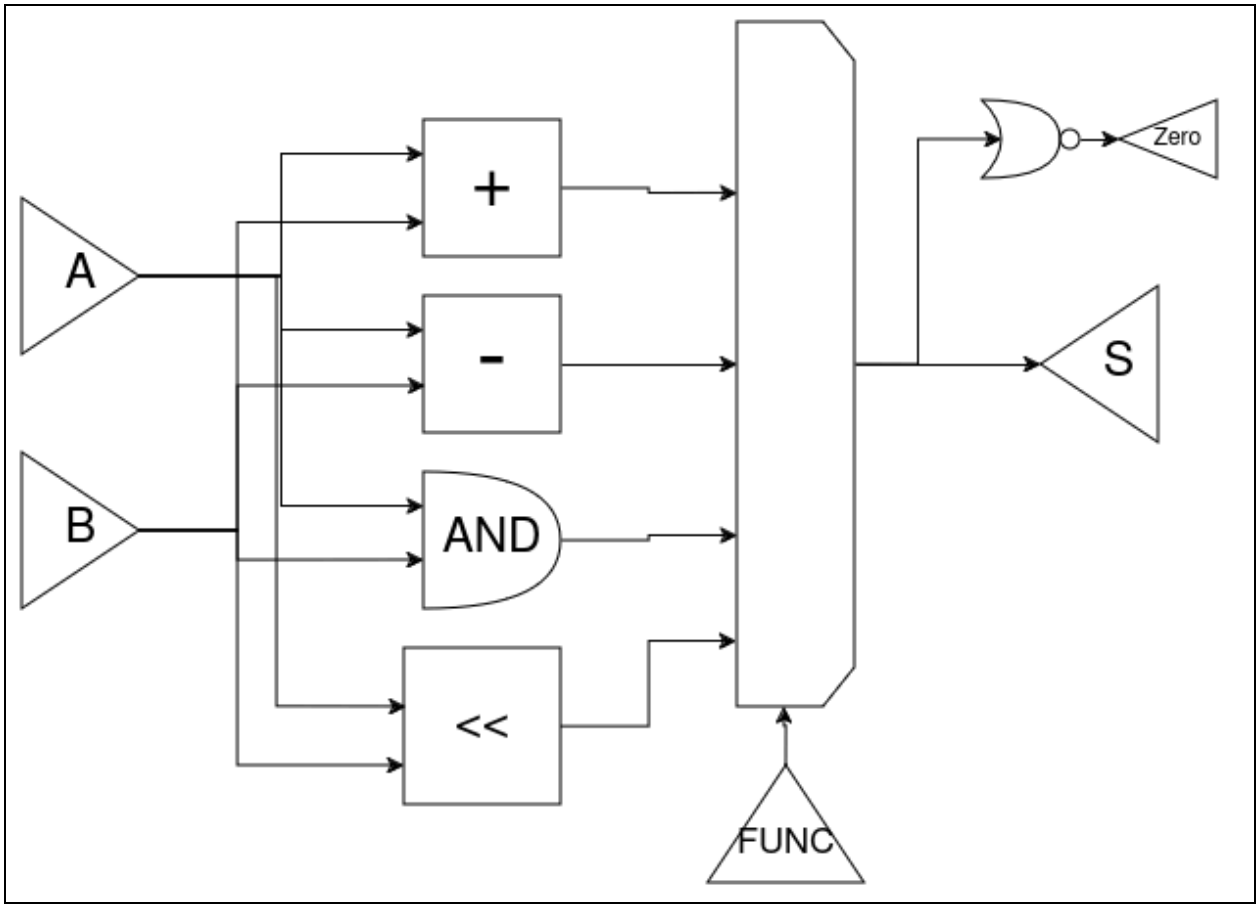

Diagrama 4: Projeto da ULA vetorial

3.2 Formato de Instruções

Os formatos de instrução seguem o mesmo padrão do VLIW após o Instruction Handler, sendo do tipo I para uso de 4 bits de imediato ou R para operação com os dois registradores de escolha, sendo 2 bits para cada um; para ambos o tamanho de opcode é 4 bits.

TIPO I

OPCODE	Imm

TIPO R

OPCODE	RA	RB

3.3 Sinais de Controle

Os sinais de controle vão estar armazenados em uma memória ROM (Read Only Memory), que como o próprio nome sugere, vão controlar os componentes para o correto funcionamento dos mesmos. O fator que decide quais sinais de controle vão ser ativados é o opcode.

OPCODE	branch	mv	mv_sel	writeEn	ramW	RB_src	ALU_src	func
0000	0	0	0	1	0	1	0	000
0001	0	0	0	0	1	0	0	000
0010	0	1	1	1	0	0	1	000
0011	0	1	0	1	0	0	0	000
0100	0	0	0	1	0	0	0	000
0101	0	0	0	1	0	0	0	001
0110	0	0	0	1	0	0	0	010
0111	1	0	0	0	0	0	0	000
1000	0	0	0	1	0	1	0	000
1001	0	0	0	0	1	0	0	000
1010	0	1	1	1	0	0	0	000
1011	0	1	0	1	0	0	0	000
1100	0	0	0	1	0	0	0	000
1101	0	0	0	1	0	0	0	001
1110	0	0	0	1	0	0	0	011
1111	0	0	0	1	0	0	0	000

3.4 Novas Instruções

Foram implementadas duas novas instruções, s.sll e sv.lr nos opcodes 1110 e 1111, respectivamente. s.sll (scalar shift logical left) é uma instrução do tipo R que faz r[b] shifts para a esquerda em r[a], enquanto sv.lr (scalar to vetorial load register) faz com que r[a] receba o conteúdo do registrador escalar r[b].

4 CÓDIGOS ASSEMBLY

4.1 Código VLIW

```
movl 15   ;; primeiro loop zera três espaços a mais (indica o fim mais claramente)
add r2 r0
brzr r2 r3   ;; COMEÇO PRIMEIRO LOOP (ZERA espaço de memória)
movl 1
sub r2 r0   ;; i--
st r1 r2   ;; zera na posição do primeiro vetor
movl 12
add r2 r0   ;; desloca 12
st r1 r2   ;; zera na posição do segundo vetor
add r2 r0   ;; desloca 12
st r1 r2   ;; zera na posição do terceiro vetor
sub r2 r0
sub r2 r0  ;; retira deslocamento
movl 2 ;; endereço de retorno para jr
svpc r3 r3  ;; salva PC para retorno - 2
add r3 r0
add r3 r0
jr r0, r0   ;; FIM PRIMEIRO LOOP
movl 12
sub r1 r1
sub r3 r3
add r2 r0    ;; reg reiniciados
movh 0    ;; COMEÇO SEGUNDO LOOP (carrega A e B)
movl 2    ;; ajusta PC de branch em r3
add r3 r0
brzr r2 r3
movl 1
sub r2 r0
st r2 r2    ;; carrega em A
movl 12    ;; desloca
add r1 r2
add r2 r0
movl 4
movh 1
add r1 r0
```

```
st r1 r2     ;; carrega em B
sub r1 r1
movh 0
movl 12
sub r2 r0
movl 4
movh 1
svpc r3 r3
jr r0, r0   ;; FIM SEGUNDO LOOP
movl 0
movl 12
sub r1 r1
sub r3 r3
add r2 r0    ;; reg reiniciados
movh 0     ;; COMEÇO TERCEIRO LOOP (soma A e B)
movl 2
add r3 r0
brzr r2 r3
sub r3 r3
movl 1
sub r2 r0
ld r3 r2    ;; recebe valor de A[i]
movl 12
add r2 r0
ld r1 r2    ;; recebe valor de B[i]
add r1 r3   ;; A[i] + B[i]
add r2 r0
st r1 r2    ;; guarda no terceiro vetor
sub r3 r3
sub r1 r1
sub r2 r0
sub r2 r0
movl 13
movh 2
svpc r3 r3
jr r0, r0 ;; FIM TERCEIRO LOOP
movl 2
sub r2 r2
svpc r2 r2
add r2 r0
jr r2 r2 ;; HALT
```

4.2 Código Vetorial

```
v.add r2, r0    ; vr[r2] = {0, 1, 2, 3}
s.movl 4        ; sr[r1] = X4
s.add r2, r1    ; sr[r2] = 4

init_A:
        s.movl 3
        s.sub r3, r1
        s.movh 1
        s.movl 4        ; sr[r1] = 20

        s.brzr r3, r1   ; se r3 == 3 pula para o init_B

        s.movh 0
        s.movl 3        ; manipulacao do laco

        s.add r3, r1    ; sr[r3] recupera seu valor original

        v.st r2, r3     ; M[ r3 ] = r2

        s.movh 0
        s.movl 1        ; sr[r1] = 1

        s.add r3, r1    ; sr[r3] = sr[r3] + 1
        sv.lr r3, r3    ; vr[r3] = sr[r3]

        sv.lr r1, r2    ; vr[r1] = 4

        v.add r2, r1    ; vr[r2] = vr[r2] + 4

        s.movl 3        ; sr[r3] = 3

        s.brzr r0, r1   ; pula para init_A

v.movh 0
v.movl 8
v.add r2, r1            ; vr[r2] = {20, 21, 22, 23}
s.movh 0
s.movl 3
s.add r3, r1            ; sr[r3] = 3

init_B:
        s.movh 0
        s.movl 6
        s.sub r3, r1
```

```
        s.movh 3
        s.movl 0        ; sr[r1] = fim do loop (45)

        s.brzr r3, r1   ; se r3 == 6 pula para o soma_AB

        s.movh 0
        s.movl 6        ; manipulacao do laco

        s.add r3, r1    ; sr[r3] recupera seu valor original

        v.st r2, r3     ; M[ r3 ] = r2

        s.movh 0
        s.movl 1        ; sr[r1] = 1

        s.add r3, r1    ; sr[r3] = sr[r3] + 1
        sv.lr r3, r3    ; vr[r3] = sr[r3]

        sv.lr r1, r2    ; vr[r1] = 4

        sv.add r2, r1   ; vr[r2] = vr[r2] + 4

        s.movh 1
        s.movl 10       ; sr[r3] = init_B (26)

        s.brzr r0, r1   ; pula para init_B

soma_AB:
        v.movh 0
        v.movl 0
        v.ld r2, r1
        v.movl 3
        v.ld r3, r1
        v.add r2, r3
        v.movl 6
        v.st r2, r1

        v.movh 0
        v.movl 1
        v.ld r2, r1
        v.movl 4
        v.ld r3, r1
        v.add r2, r3
        v.movl 7
        v.st r2, r1
```

```
        v.movh 0
        v.movl 2
        v.ld r2, r1
        v.movl 5
        v.ld r3, r1
        v.add r2, r3
        v.movl 8
        v.st r2, r1

s.movh 4
s.movl 7
s.brzr r0, r1
```