

# Trabalho 3 - VLIW e Vetorial

SERGIO SIVONEI DE SANT'ANA FILHO GRR20242337  
EDUARDO KALUF GRR20241770

May 30, 2025

## 1 Apresentação

Ao decorrer da disciplina aprendemos inúmeras maneiras de realizar a implementação de um processador, o modo como operam, seus prós e contras e seus contextos históricos. Sendo assim, neste trabalho iremos nos aprofundar em duas dessas maneiras sendo elas o VLIW e o processador Vetorial, o objetivo é a partir de uma ISA já predefinida (SAGUI) criar o projeto de ambos os processadores utilizando **Logisim**, criar duas instruções adicionais para cada ISA e então realizar a execução de um programa assembly em cada um deles.

Para cada um dos processadores iremos utilizar uma ISA diferente que deriva do SAGUI, no caso do VLIW será o Saguí de rabo longo, já para o vetorial o Saguí em bando, além disso, o programa assembly teste consiste em fazer a soma de dois vetores com 12 posições e guardar os resultados na memória, com todos os vetores alinhados. Devido a maneira de como os processadores operam, cada um deles terá um programa Assembly diferente, porém que deverá apresentar o mesmo resultado.

## 2 VLIW

### 2.1 Arquitetura

VLIW significa "Very long instruction Word" que se traduz para "palavra de instrução muito grande" e a ideia do processador gira exatamente em torno disso, basicamente, iremos ter uma palavra grande que carrega mais de uma instrução dentro dela, no nosso caso, 4 instruções, dessa forma o processador irá executar-las em paralelo aumentando o IPC total. No VLIW a responsabilidade de evitar dependências e escritas simultâneas ficam completamente na mão do compilador, o qual utiliza diversas técnicas como loop unrolling e predicação a fim de diminuir a quantidade de NOPS necessários, fazendo com que o código seja executado de maneira mais otimizada.

## 2.2 ISA

A ISA que iremos utilizar para o nosso VLIW é a Sagui de Rabo Longo (SRB) com algumas modificações para operar com predicação. A SRB consiste em uma ISA parecida com a REDUX-V, sendo tão compacta e amigável quanto para quem está iniciando no mundo da arquitetura de computadores. Ela possui 8 bits com 4 registradores de propósito geral e será endereçada de palavra a palavra ou seja de 4 em 4 bytes, além de definir 4 "lanes" diferentes, esses "lanes" são os tipos de operação (Branch, aritmética, lógica e etc...) que cada posição da palavra do VLIW poderá realizar. O opcode possui 4 bits, fazendo com que possamos operar 16 instruções diferentes, devido as modificações para predicação as instruções serão apresentadas mais a frente já em suas lanes específicas.

Por padrão, os formatos das instruções e as lanes são definidas a seguir:

**Formato Padrão 1: Tipo I**

Tipo I							
Bits	7	6	5	4	3	2	1 0
	Opcode				Imm.		

**Formato Padrão 2: Tipo R**

Tipo R							
Bits	7	6	5	4	3	2	1 0
	Opcode				Ra		Rb

**Lanes:**

Dados	Controle	ULA	ULA
1 <sup>o</sup>	2 <sup>o</sup>	3 <sup>o</sup>	4 <sup>o</sup>

## 2.3 Motivações e Instruções

Para este processador gostaríamos de dar a maior quantidade de operações lógicas possíveis ao programador e uma facilidade maior para mexer com loops, adicionando outra instrução branch de controle. Além disso, para que a utilização de predicados seja viável, adaptamos a ISA de modo que ela possua um conjunto de instruções específicas para cada uma das lanes do VLIW e instruções para settar o predicado na parte de controle

Formato Novo: Tipo Misto

Tipo INC								
Bits	7	6	5	4	3	2	1	0
	Opcode				Ra		Imm Unsigned	

### 2.3.1 Multi Operations

### 2.3.2 Move

### 2.3.3 Predicação

Lane 1 Instruções: Dados

Opcode	Tipo	Mnemonic	Nome	Operação
Dados				
0000	R	ld	Load	$R[ra] = M[R[rb]]$ if PR
0001	R	st	Store	$M[R[rb]] = R[ra]$ if PR
0010	I	movh	Move High	$R[0] = Imm, R[0](3:0)$ if PR
0011	I	movl	Move Low	$R[0] = R[0](7:4), Imm$ if PR
0100	R	f-ld	!Load	$R[ra] = M[R[rb]]$ if !PR
0101	R	f-st	!Store	$M[R[rb]] = R[ra]$ if !PR
0110	I	f-movh	!Move High	$R[0] = Imm, R[0](3:0)$ if !PR
0111	I	f-movl	!Move Low	$R[0] = R[0](7:4), Imm$ if !PR
1000	EMPTY	EMPTY	EMPTY	EMPTY
1001	EMPTY	EMPTY	EMPTY	EMPTY
1010	EMPTY	EMPTY	EMPTY	EMPTY
1011	EMPTY	EMPTY	EMPTY	EMPTY
1100	EMPTY	EMPTY	EMPTY	EMPTY
1101	EMPTY	EMPTY	EMPTY	EMPTY
1110	EMPTY	EMPTY	EMPTY	EMPTY
Free Slot				
1111	R	nop	No Operation	

**Lane 2 Instruções: Controle**

Opcode	Tipo	Mnemonic	Nome	Operação
<b>Controle</b>				
0000	R	brzr	Branch On Zero Register	if (R[ra] == 0) PC = R[rb] if PR
0001	I	brzi	Branch On Zero Immediate	if (R[0] == 0) PC = PC + Imm if PR
0010	R	jr	Jump Register	PC = R[rb] if PR
0011	R	mov	Move	R[ra] = R[rb] if PR
0100	R	f-brzr	!Branch On Zero Register	if (R[ra] == 0) PC = R[rb] if !PR
0101	I	f-brzi	!Branch On Zero Immediate	if (R[0] == 0) PC = PC + Imm if !PR
0110	R	f-jr	!Jump Register	PC = R[rb] if !PR
0111	R	f-mov	!Move	R[ra] = R[rb] if !PR
<b>Setters</b>				
1000	R	sgt	Set Greater Than	if R[ra] > R[rb] R[pr] := 1; R[pr] := 0
1001	R	slt	Set Less Than	if R[ra] < R[rb] R[pr] := 1; R[pr] := 0
1010	R	seq	Set Equal	if R[ra] = R[rb] R[pr] := 1; R[pr] := 0
1011	R	strue	Set True	R[pr] := 1
1100	EMPTY	EMPTY	EMPTY	EMPTY
1101	EMPTY	EMPTY	EMPTY	EMPTY
1110	EMPTY	EMPTY	EMPTY	EMPTY
<b>Free Slot</b>				
1111	R	nop	No Operation	

**Lanes 3 e 4 Instruções: Aritmética e Lógica**

Opcode	Tipo	Mnemonic	Nome	Operação
<b>Aritmética e Lógica</b>				
0000	R	add	Add	R[ra] = R[ra] + R[rb] if PR
0001	R	sub	Sub	R[ra] = R[ra] - R[rb] if PR
0010	M	mult-op	Multi Operations	R[ra] = R[ra] OP R[0] — !R[0] if PR
0011	R	or	Or	R[ra] = R[ra] — R[rb] if PR
0100	R	not	Not	R[ra] = ! R[rb] if PR
0101	R	slr	Shift Left Register	R[ra] = R[ra] ; R[rb] if PR
0110	R	srr	Shift Right Register	R[ra] = R[ra] ; R[rb] if PR
0111	EMPTY	EMPTY	EMPTY	EMPTY
1000	R	f-add	Add	R[ra] = R[ra] + R[rb] if !PR
1001	R	f-sub	Sub	R[ra] = R[ra] - R[rb] if !PR
1010	M	f-mult-op	Multi Operations	R[ra] = R[ra] OP R[0] — !R[0] if !PR
1011	R	f-or	Or	R[ra] = R[ra] — R[rb] if !PR
1100	R	f-not	Not	R[ra] = ! R[rb] if !PR
1101	R	f-slr	Shift Left Register	R[ra] = R[ra] ; R[rb] if !PR
1110	R	f-srr	Shift Right Register	R[ra] = R[ra] ; R[rb] if !PR
<b>Free Slot</b>				
1111	R	nop	No Operation	

### 3 Implementação e Organização

Lane 1 Controle: Dados						
Mnemonic	OPCODE	MOVL	MOVH	ST	LD	TRUE
ld	0000	0	0	0	1	1
st	0001	0	0	1	0	1
movh	0010	0	1	0	0	1
movl	0011	1	0	0	0	1
f-ld	0100	0	0	0	1	0
f-st	0101	0	0	1	0	0
f-movh	0110	0	1	0	0	0
f-movl	0111	1	0	0	0	0
empty	1000	x	x	x	x	x
empty	1001	x	x	x	x	x
empty	1010	x	x	x	x	x
empty	1011	x	x	x	x	x
empty	1100	x	x	x	x	x
empty	1101	x	x	x	x	x
empty	1110	x	x	x	x	x
nop	1111	0	0	0	0	x

Lane 2 Controle: Branches								
Mnemonic	OPCODE	BOZR	JR	BOZI	MOVE	TRUE	PR_OP	W_PR
brzr	0000	1	0	0	0	1	xx	0
brzi	0001	0	0	1	0	1	xx	0
jr	0010	0	1	0	0	1	xx	0
mov	0011	0	0	0	1	1	xx	0
f-brzr	0100	1	0	0	0	0	xx	0
f-brzi	0101	0	0	1	0	0	xx	0
f-jr	0110	0	1	0	0	0	xx	0
f-mov	0111	0	0	0	1	0	xx	0
sgt	1000	0	0	0	0	x	00	1
slt	1001	0	0	0	0	x	10	1
seq	1010	0	0	0	0	x	01	1
strue	1011	0	0	0	0	x	11	1
empty	1100	x	x	x	x	x	xx	x
empty	1101	x	x	x	x	x	xx	x
empty	1110	x	x	x	x	x	xx	x
nop	1111	0	0	0	0	x	xx	0

**Lane 3 e 4 Controle: Aritmética e Lógica**

Mnemonic	OPCODE	OP_ULA	WE	TRUE
add	0000	000	1	1
sub	0001	001	1	1
mult-op	0010	010	1	1
or	0011	011	1	1
not	0100	100	1	1
slr	0101	101	1	1
srr	0110	110	1	1
empty	0111	xxx	x	x
f-add	1000	000	1	0
f-sub	1001	001	1	0
f-mult-op	1010	010	1	0
f-or	1011	011	1	0
f-not	1100	100	1	0
f-slrr	1101	101	1	0
f-srr	1110	110	1	0
nop	1111	111	0	x

### 3.0.1 Instrução 1

### 3.0.2 Instrução 2

### 3.0.3 Predicação

## 4 Resultado