

# Trabalho 3 - VLIW e Vetorial

SERGIO SIVONEI DE SANT'ANA FILHO GRR20242337  
EDUARDO KALUF GRR20241770

June 2, 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
1º	2º	3º	4º

## 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

### 2.3.1 Multi Operations

A primeira escolha de instrução foi a de 'multioperações' (MULTIOPS), a qual é do tipo misto (INC) tendo um registrador indicado e um imediato de 0

a 3.

Essa instrução funciona como um grupo de 4 instruções simples, sendo elas: NOT(logico), XOR, AND e XNOR. Todos operando sobre R0 (que é sempre o registrador destino) e Ra (por Ra entenda registrador descrito na instrução).

Ela segue o formato OPCODE nos 4 últimos bits e nos outros 4 ela divide em metade para indicar o Ra e outra para um numero de 0 a 3, o qual indica qual operação deve ser realizada, NOT(0), XOR(1), AND(2) e XNOR(3).

Formato Novo: Tipo Misto								
Tipo INC								
Bits	7	6	5	4	3	2	1	0
	Opcode				Ra		Imm Unsigned	

### 2.3.2 Move

A instrução move faz uso da logica de copiar o valor de um registrador para outro, coisa que é útil em casos com grande limitação (como no nosso caso). Isso se dá por, por exemplo, só podermos alterar diretamente o valor de R0 (usando as funções movh/movl). Essa função faz Ra receber Rb, sobrescrevendo o conteudo anterior de Ra. Tal lógica retira a necessidade de zerar um registrador para depois somar por exemplo.

### 2.3.3 Predicação

A fim de aproveitar que temos 16 instruções disponíveis por LANE, fizemos a implementação de um sistema de predicados em nosso processador. Temos 4 instruções responsáveis por controlar o valor do predicado (TRUE ou FALSE) e então todas as outras instruções possuem um espelho de si mesmas, a primeira funcionando quando o predicado é true e a segunda quando o mesmo é falso. Dessa forma o programador pode fazer o caminho verdadeiro de um if junto com o caminho falso, diminuindo a quantidade de dependencias no código. Vale a pena resaltar que uma das instruções é o STRUE, a ideia é que ela seja utilizada em conjunto com as instruções que executam o caminho verdadeiro quando nenhuma predicação está acontecendo, assim evitamos criar um terceiro espelho de cada instrução que funcione independente do valor do predicado As 4 instruções estão apresentadas abaixo juntamente com a tabela final de cada uma das LANES

**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
Controle				
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
Setters				
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
Free Slot				
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	$R[ra] = R[ra] \text{ OP } R[0] \mid !R[0]$ if PR
0011	R	or	Or	$R[ra] = R[ra] \mid R[rb]$ if PR
0100	R	not	Not	$R[ra] = ! R[rb]$ if PR
0101	R	slr	Shift Left Register	$R[ra] = R[ra] \ll R[rb]$ if PR
0110	R	srr	Shift Right Register	$R[ra] = R[ra] \gg R[rb]$ 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	$R[ra] = R[ra] \text{ OP } R[0] \mid !R[0]$ if !PR
1011	R	f-or	Or	$R[ra] = R[ra] \mid R[rb]$ if !PR
1100	R	f-not	Not	$R[ra] = ! R[rb]$ if !PR
1101	R	f-slr	Shift Left Register	$R[ra] = R[ra] \ll R[rb]$ if !PR
1110	R	f-srr	Shift Right Register	$R[ra] = R[ra] \gg R[rb]$ if !PR
<b>Free Slot</b>				
1111	R	nop	No Operation	

## 3 Implementação e Organização

Na implementação do nosso VLIW consumimos 4 instruções por vez, simulando uma palavra grande dessa maneira, elas são então passadas para os controles que já definem se a instrução deve ou não ser executada com base no predicado. Na ULA temos uma saída que retorna zero em todos os 8 bits, utilizada em casos em que a instrução não deve ser executada pelo predicado ou casos de nop. Para cada LANE existe uma UL que faz as lógicas necessárias a fim de retornar o que cada instrução precisa. O banco de registradores utiliza varios DEMUXes para abranger a escrita em qualquer registrador independente da LANE que envia o dado.

Table 7: **ULA: Controle**

Operação	OP_ULA
Add	000
Sub	001
Multi-op	010
Or	011
Not	100
Sll	101
Slr	110
Nop	111

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 Multi Operations

A instrução Multi Operations foi implementada dentro da Ula e pode ser pensada como uma extensão da mesma. A Ula passou a receber 2 bits de controle a mais, vindos da MULT-OPS, que definem qual operação fazer e então, com base no código do Op\_ula é decidido se será escolhida ou não. MULT-OPS opera fixamente sobre o R0, o resto do caminho é idêntico a qualquer outra operação Lógica/Aritmética

Table 11: Multi Operations: Controle

Operação	OP_PR
Greater	00
Less	01
Equal	10
Set True	11

### 3.0.2 Move

Para a Instrução Move, simplesmente retornamos o valor de RB para o banco de registradores, este, com base se MOVE foi settado ou não passa o valor para o RA

### 3.0.3 Predicação

O registrador de predicados recebe o valor zero ou um dependendo do resultado das comparações e é sempre atualizado na borda de descida, dessa maneira é possível enviar outras instruções não dependentes de predicado em conjunto de algum dos set predicates. O valor dele então é utilizado pelas unidades de controle, que identificam com base na instrução se um "hit" ocorreu ou não, settando os controles de maneira apropriada.

Table 12: Operações do Predicado: Controle

Operação	OP_ULA
	00
Xor	01
And	10
Xnor	11

## 4 Assembly

Após a implementação da arquitetura junto com as instruções criadas chegamos ao seguinte Assembly:

### Assembly Para Teste

```
;;zera os registradores e r0 = -1
movh -1
nop
sub r1 r1
sub r2 r2

movl -1
nop
nop
sub r3 r3

;;r1 = -1 e r0 = -12
movl 4
nop
nop
add r1 r0
```



```

;;r2 = 12 (tamanho dos vetores)
nop
nop
nop
sub r2 r0

;;salva valor 12 na memoria M[-1]
st r2 r1
nop
nop
nop

//zera ro e r0 = 100
movh 6
nop
nop
sub r1 r1

;;r1 = 100 (endereço de inicio dos vetores, r1 = A[0])
movh 0
nop
nop
add r1 r0

;;r0 = 1 e r1 = B[0]
movl 1
nop
nop
add r1 r2

;;r3 = 1
nop
nop
nop
add r3 r0

;;comeco do loop de preencher vetor A
;;3 x unrolling preenche de 4 em 4
;;r1-- e r2--
nop
nop
sub r1 r3
sub r2 r3

;;salva o valor da vez na posicao da vez
st r2 r1
nop
sub r1 r3
sub r2 r3

```

```

st r2 r1
nop
sub r1 r3
sub r2 r3

st r2 r1
nop
sub r1 r3
sub r2 r3

;;r0 recebe o not r2 (saira do loop quando r2 for 0)
st r2 r1
nop
not r0 r2
nop

;;branch para o comeco do loop
nop
brzi -5
nop
nop

;;r0 = -1
movh -1
nop
nop
nop

movl -1
nop
nop
nop

;;pega o valor 12 da memoria
ld r2 r0
nop
nop
nop

;;r1 = B[0]
movh 1
nop
nop
add r1 r2

;;r0 = 19 e r1 = R[0]
movl 3
nop
nop
add r1 r2

```

```

;;r3 = 20
movh 0
nop
nop
add r3 r0

;;r0 = 0 r3 = 32
movl 0
nop
nop
add r3 r2

;;r0 = 1
nop
nop
nop
not r0 r0

;;comeco do loop de preencher vetor B
;;r3--
;;r1--
nop
nop
sub r3 r0
sub r1 r0

;;salva o valor da vez no endereco certo
;;r2-- (iterador)
st r3 r1
nop
sub r2 r0
nop

;;r0 so sera 1 quando r2 for 0 (fim do loop)
nop
nop
nop
not r0 r2

;;branch de loop
nop
brzi -4
nop
nop

;;zera r3 e comeca a calcular valor de branch
movh 8
nop
nop

```

```

sub r3 r3

;;r0 = 130
movl 12
nop
nop
nop

;;r3 = endereco de branch
movh -1
nop
nop
add r3 r0

;;r0 = -2
movl -2
nop
nop
nop

;;salva valor de r3 na memoria
st r3 r0
nop
nop
nop

;;r0 = -1
movl -1
nop
nop
nop

;;r2 = 12
ld r2 r0
nop
nop
nop

;;r0 = -3
movl -3
nop
nop
nop

;;salva iterador na memoria
st r2 r0
nop
nop
nop

```

```

;;comeco do loop de soma
;;2xUnrolling (preenche 3 posicoes por vez)
;;r0 = -1
movl -1
nop
nop
nop

;;r2 = 12 (tamanho vetor)
ld r2 r0
nop
nop
nop

;; r1--
nop
nop
add r1 r0
nop

;;baixa valor da vez de A
;;r1 = B da vez
ld r0 r1
nop
nop
add r1 r2

;;baixa valor da vez de B
;;r1 = R da vez (R[x])
ld r3 r1
nop
nop
add r1 r2

;;r3 = soma de A[x] e B[x]
movh -1
nop
nop
add r3 r0

;;salva a soma em R
;;r1 = B da vez
st r3 r1
nop
nop
sub r1 r2

;;r0 = -1
;;r1 = A da vez
movl -1

```

```

nop
nop
sub r1 r2

;;r1 -- (proxima posicao)
nop
nop
nop
add r1 r0

;;baixa valor da vez de A
;;r1 = B da vez
ld r0 r1
nop
nop
add r1 r2

;;baixa valor da vez de B
;;r1 = R da vez (R[x])
ld r3 r1
nop
nop
add r1 r2

;;r3 = soma de A[x] e B[x]
movh -1
nop
nop
add r3 r0

;;salva a soma em R
;;r1 = B da vez
st r3 r1
nop
nop
sub r1 r2

;;r0 = -1
;;r1 = A da vez
movl -1
nop
nop
sub r1 r2

;;r1 -- (proxima posicao)
nop
nop
nop
add r1 r0

```

```

;;baixa valor da vez de A
;;r1 = B da vez
ld r0 r1
nop
nop
add r1 r2

;;baixa valor da vez de B
;;r1 = R da vez (R[x])
ld r3 r1
nop
nop
add r1 r2

;;r3 = soma de A[x] e B[x]
movh -1
nop
nop
add r3 r0

;;salva a soma em R
;;r1 = B da vez
st r3 r1
nop
nop
sub r1 r2

;;r0 = -2
movl -2
nop
nop
sub r1 r2

;;baixa o valor de branch da memoria em r3
ld r3 r0
nop
nop
nop

;;r0 = -3
movl -3
nop
nop
nop

;;baixa iterador da memoria em r2
ld r2 r0
nop
nop
nop

```

```

;;r2 = r2 - 3 (iterador - 3, pois faz 3 por loop)
nop
nop
nop
add r2 r0

;;salva o iterador na memoria
;;r0 recebe !iterador (so saira do loop quando iterador for 0)
st r2 r0
nop
nop
not r0 r2

;;branch para comeco do loop
movh -1
brzr r0 r3
nop
nop

;; zera r0
nop
nop
nop
sub r0 r0

;;HALT
nop
brzi 0
nop
nop

```

Além deste código, um teste para mostrar a eficiência dos predicados também foi criado:

#### Assembly Para de Predicados

```

;;vai executar as proximas instrucoes ate o proximo branch
nop
strue
nop
nop

;;zera r0 usando xor do mult-ops
nop
nop
mult-op r0 1
sub r3 r3

;;r0 = 10 e zera r1 e r2 com um and 0

```



```

movl 10
nop
mult-op r1 2
mult-op r2 2

;;r1 e r3 recebe o valor de 10
;;r2 = 1
nop
mov r1 r0
add r3 r0
not r2 r2

;; deve executar as instrucoes normais
;; predicado se r1 for maior que r2 executa
;; as instrucoes normais ate o proximo set
;; caso seja falso, executa as f-.
;; zera r0 com xor
nop
sgt r1 r2
mult-op r0 1
nop

;;deve executar r1 = r1 - r2
nop
nop
sub r1 r2
nop

;;r2 nao deve receber valor de r1
nop
f-mov r2 r1
nop
nop

;; deve executar as instrucoes 'f-'
;; se r1 igual a r2
nop
seq r1 r2
nop
nop

;; deve guardar o valor 10 no endereco 10 da memoria
;; deve pular para instrucao strue
;; r2 deve ser igual a 10
;; nao deve zerar r1
f-st r3 r3
f-brzi 2
f-add r2 r1
sub r1 r1

```

```
;;nao deve fazer o load
f-ld r3 r3
nop
nop
nop

;;seta novamente para true
;;ira executar normalmente as instrucoes a partir daqui
nop
strue
nop
nop

;;halt
nop
brzi 0
nop
nop
```

## 5 Vetorial

## 6 Conclusão