Universidade Federal do Paraná

CI1212 - Arquitetura de Computadores

PROCESSADOR SAGUI EM BANDO 8-Bits

Aluno: **Ruibin Mei**
Professor: **Marco Zanata**

Curitiba, 2024

Sumário

1 DESCRIÇÃO DO TRABALHO

No trabalho desenvolvido, utiliza-se um software chamado *"Logisim-Evolution"* para implementar uma microarquitetura monociclo seguindo as instruções da arquitetura Sagui SIMD (*Single Instruction Multiple Data*), com objetivo de testar se o circuito lógico foi elaborada de forma correta. Testando a soma dos elementos de dois vetores da memória RAM.

2 IMPLEMENTAÇÃO DO CIRCUITO LÓGICO

A implementação do circuito lógico foi guiado pela arquitetura Sagui SIMD, mostrado na imagem abaixo (Figura 1), formato das instruções (Figura 2):

Sagui em Bando (Vetorial)

Vector Architecture				
Opcode	**Tipo**	**Menemonico**	**Nome**	**Operação**
Scalar - SPE				
0000	R	ld	Load	SR[ra] = M[SR[rb]]
0001	R	st	Store	M[SR[rb]] = SR[ra]
0010	I	movh	Move High	SR[1] = {Imm., SR[1](3:0)}
0011	I	movl	Move Low	SR[1] = {SR[1](7:4), Imm.}
0100	R	add	Add	SR[ra] = SR[ra] + SR[rb]
0101	R	sub	Sub	SR[ra] = SR[ra] - SR[rb]
0110	R	and	And	SR[ra] = SR[ra] & SR[rb]
0111	R	brzr	Branch On Zero Register	if (SR[ra] == 0) PC = SR[rb]
Vector - VPE				
1000	R	ld	Load	VR[ra] = M[VR[rb]]
1001	R	st	Store	M[VR[rb]] = VR[ra]
1010	I	movh	Move High	VR[1] = {Imm., VR[1](3:0)}
1011	I	movl	Move Low	VR[1] = {VR[1](7:4), Imm.}
1100	R	add	Add	VR[ra] = VR[ra] + VR[rb]
1101	R	sub	Sub	VR[ra] = VR[ra] - VR[rb]
1110	R	and	And	VR[ra] = VR[ra] & VR[rb]
1111	R	or	Or	VR[ra] = VR[ra] | VR[rb]
			4x Vector PE	**Scalar PE**
SR -> Scalar register			4 Regs por PE - Sendo o 1º é o ID {0,1,2,3} do PE ele é hardwired (não muda) - Os outros 3x são de propósito geral (GP)	4 Regs - Sendo o 1º igual a ZERO hardwired (não muda) - Outros 3x de propósito geral (GP)
VR -> Vectorial register			VR0 = {0,1,2,3} dependendo do PE	SR0 = 0
			1 Memória por PE Considere que o vetor estará distribuído entre as memórias	1 Memória exclusiva
Em um dado ciclo apenas 1 dos PEs atuam, ou VPE ou SPE. Não existem instruções que atuem de forma escalar e vetorial ao mesmo tempo. Enquanto um atua o outro recebe NOP.				

Figura 1: A arquitetura Sagui SIMD

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

Tipo I							
7	6	5	4	3	2	1	0
opcode				Imm			

Figura 2: Formato das Instruções

2.1 DataPath

Inicialmente foi projetado o circuito baseado na versão 1 do Sagui (Sagui Monociclo), ou seja, sendo uma base para o circuito atual. Mais tarde o desenvolvimento de alguns detalhes foram modificados diretamente no circuito (Figura 3):

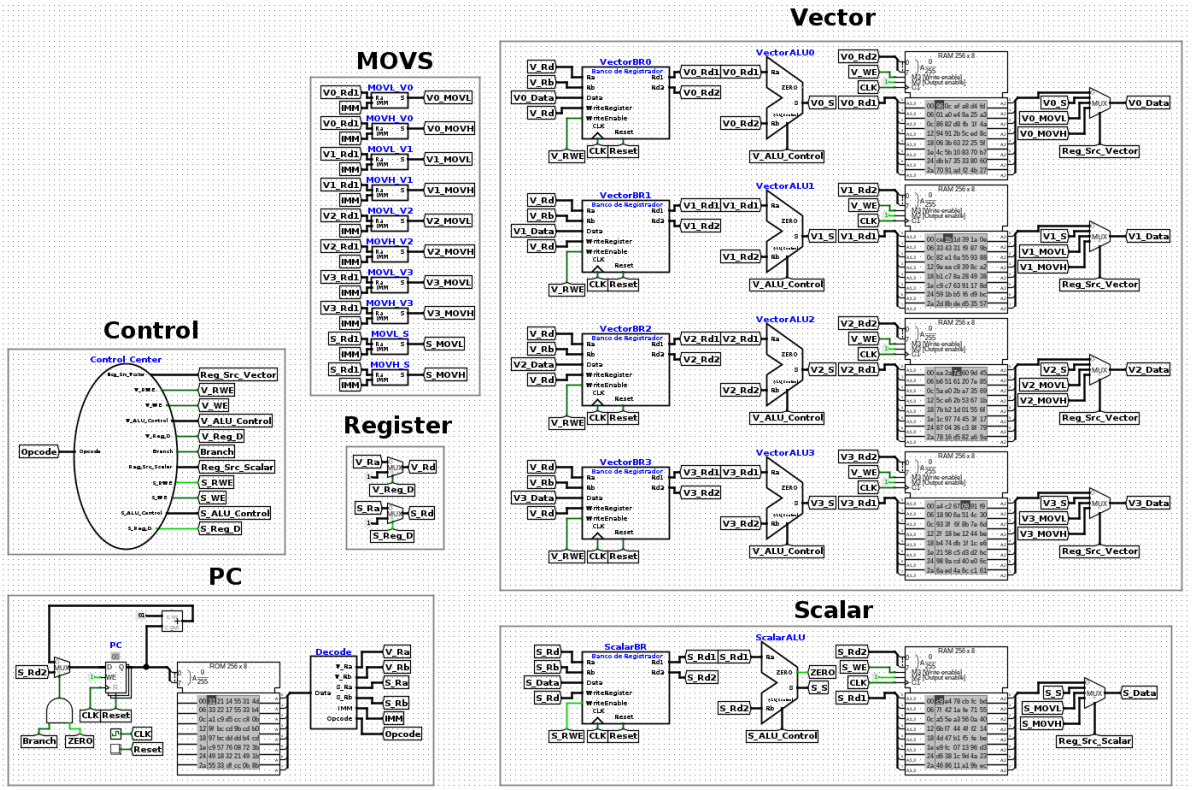

Figura 3: Datapath

2.2 UNIDADE LÓGICA E ARITMÉTICA

A ALU (*arithmetic logic unit*), conhecida também como ULA (*Unidade lógica e aritmética*), foi a primeira parte feita neste trabalho. No nome ficará notável que fará operações lógicas e aritméticas do processador, seguida da arquitetura Sagui (Figura 4).

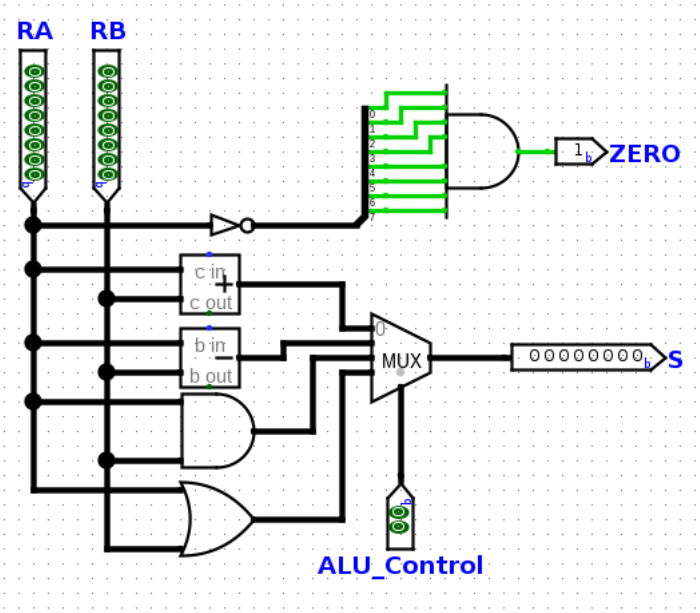

Figura 4: Circuito da ULA no Logisim

2.2.1 ALU Control

A função ALU Control, que na verdade é a função/entrada, ou seja, ele será o seletor do MUX, para indicar qual operação será feito na ULA.

Os sinais de controle:

ALUControl	Function
00	ADD
01	SUB
10	AND
11	OR

2.3 MEMÓRIA DE INSTRUÇÕES

A memória de instruções é uma das partes mais importantes do processador, pois é ela que manda sinais para o processador, o que é feito em todo seu processamento. Ela trabalha junto com um PC ou IP, conhecido como *Program Counter* ou *Instruction Pointer*, sendo um registrador que mantém o endereço da próxima instrução a ser executada, em outras palavras, ela controla o fluxo do programa (Figura 5).

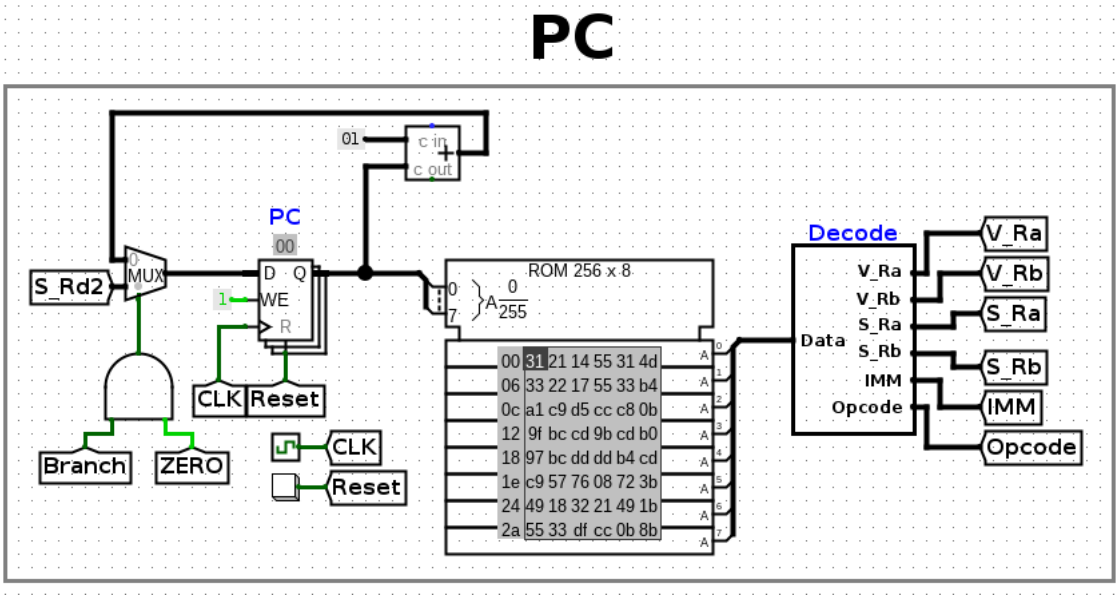

Figura 5: PC

Na implementação do todo o controle do fluxo, foi analisado o que acontece em cada uma das situações das instruções que ocorre. O PC elaborado, é um registrador junto com uma ROM (*Read-Only Memory*, memória que pode ser apenas para leitura) e CLK (*Clock*) usada para atualizar o registrador e sincronização do circuito todo.

A ideia da implementação, foi iniciado primeiramente para que execute a próxima instrução, ou seja, fazendo $PC = PC + 1$ para próxima instrução o que pode ser notado na parte superior do circuito, foi utilizado um somador e uma constante 1 para fazer $PC + 1$. Depois disso, como outras operações segundo a Figura 1, do tipo Branch, que faz $PC = R[rb]$, então necessitou utilizar um MUX para receber o PC novo vindo ou do $PC + 1$ ou do caso citado anteriormente.

O controle principal do PC é o MUX, é o sinal da porta lógica AND, comparando se a instrução é do tipo Branch e a saída da ULA para se o R[ra] é igual a 0 ou não, para controlar o PC vindo de $PC = R[rb]$ ou $PC = PC + 1$.

Depois da memória de instruções é notável que possui vários distribuidores de sinais, como: Opcode, IMM, S_Ra, S_Rb, V_Ra, V_Rb, esses são os sinais que será necessário para memória de controle, banco de registradores.

Legenda:

- *Opcode* código da instrução
- IMM imediato
- S_Ra registrador A do escalar
- S_Rb registrador B do escalar
- V_Ra registrador A do vetorial
- V_Rb registrador B do vetorial

2.4 BANCO DE REGISTRADOR

O Banco de registrador é uma peça importante para que podemos olhar nele alguns dados armazenados que é utilizado no processador quando algum programa é executado. Nele possui as entradas dos registradores, do dado, clock , seletores e um WriteEnable (*RegisterEnable*, um sinal de 1 bit vinda da memória de controle), para verificar se escreve ou não em algum registrador do banco de registradores (Figura 6).

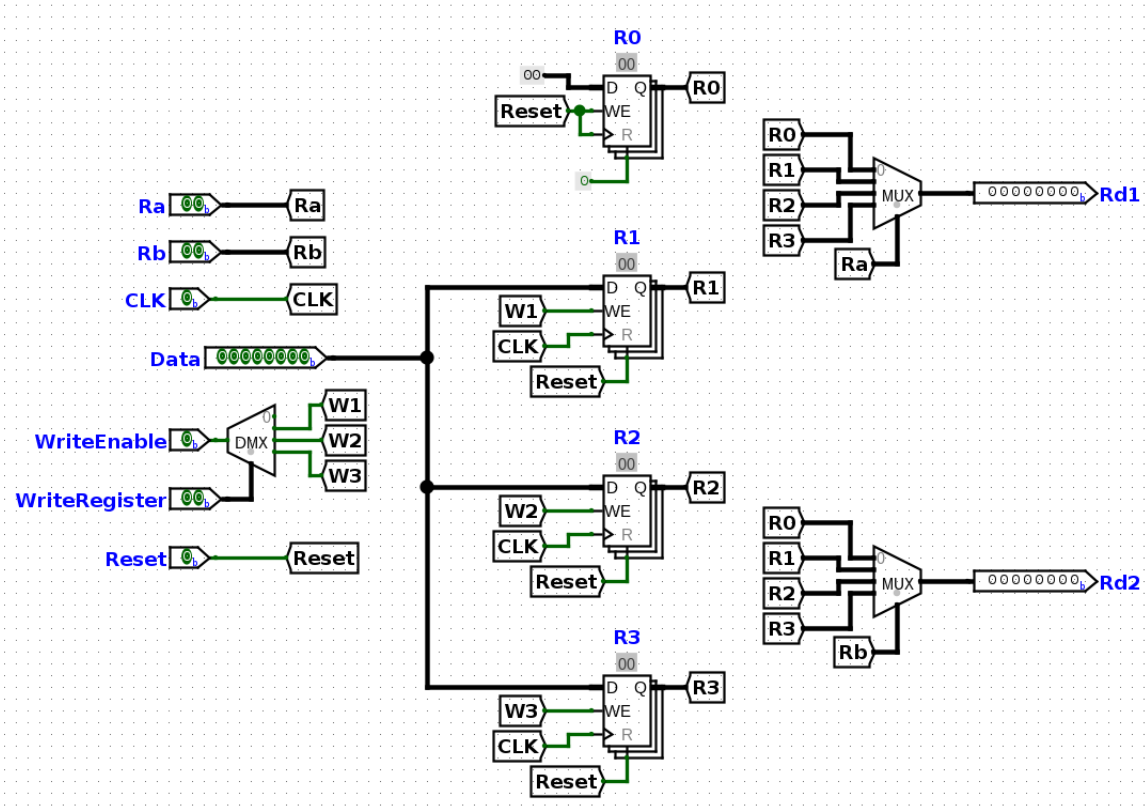

Figura 6: Banco de Registrador do Escalar

Fica notável que o MUX seleciona apenas os registradores 1, 2 e 3, pois isso é devido as instruções da arquitetura Sagui SIMD descrito no trabalho, na qual, o registrador 0 deve ser um registrador hardwired (não muda o valor, ou valor constante) e o seu write enable sempre estará desabilitado.

E isso acontece também para a parte vetorial do processador, ou seja, no registrador 0 dos bancos de registradores dos PEs (*Processing Elements*), como especificado no trabalho, as constantes são respectivamente 0, 1, 2 e 3. Mostrado como exemplo o R0 do banco de registrador vetorial 1 (Figura 7):

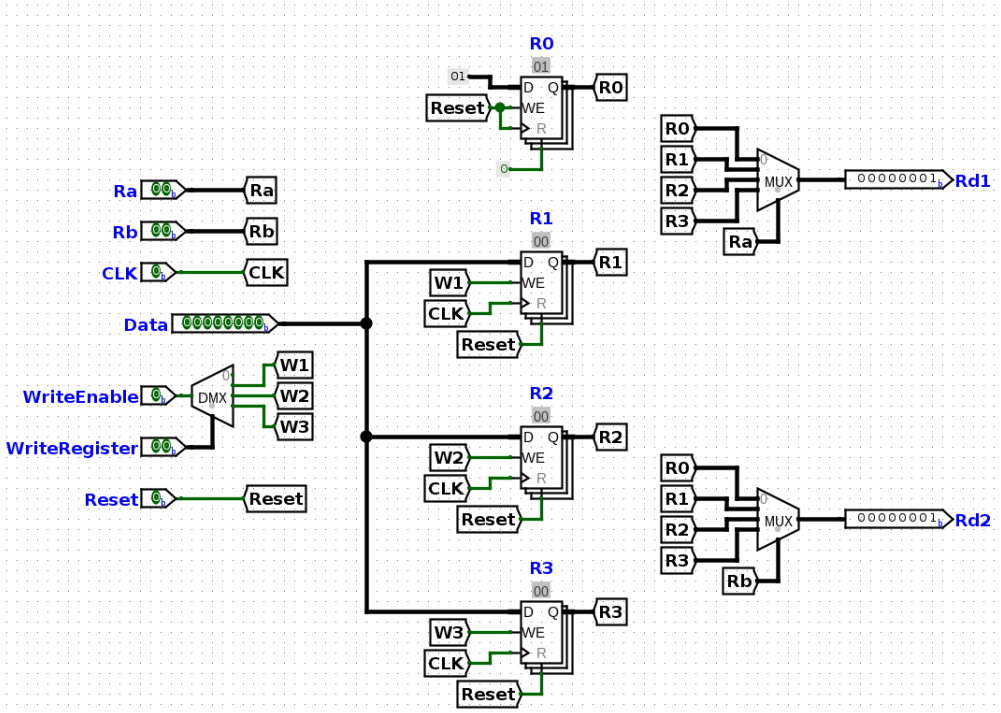

Figura 7: Banco de Registrador do Vetorial 1

2.5 MEMÓRIA DE CONTROLE

Com todas os componentes citadas acima, a memória de controle também é uma parte importante do processador, sem ela como no nome já indica, não terá controle de cada uma das componentes.

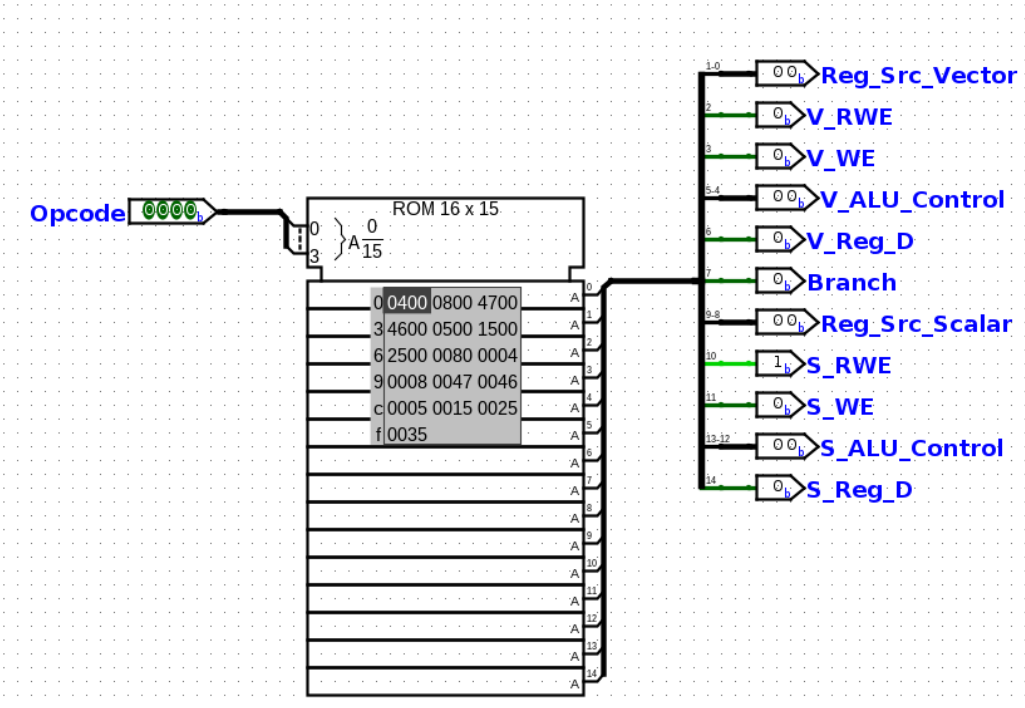

Figura 8: Memória de Controle

Na memória de controle ela recebe a entrada do opcode: E foi usada uma ROM para memória de controle.

		1	2	1	1	2	1	1	2	1	1	2	
Inst	Opcode	S_Reg_D	S_ALU_Control	S_WE	S_RWE	Reg_Src_Scalar	Branch	V_Reg_D	V_ALU_Control	V_WE	V_RWE	Reg_Src_Vector	Hexadecimal
Load	"0000"	0	"00"	0	1	"00"	0	x	xx	x	x	xx	400
Store	"0001"	0	"00"	1	0	xx	0	x	xx	x	x	xx	800
Move High	"0010"	1	xx	0	1	"11"	0	x	xx	x	x	xx	4700
Move Low	"0011"	1	xx	0	1	"10"	0	x	xx	x	x	xx	4600
Add	"0100"	0	"00"	0	1	"01"	0	x	xx	x	x	xx	500
Sub	"0101"	0	"01"	0	1	"01"	0	x	xx	x	x	xx	1500
And	"0110"	0	"10"	0	1	"01"	0	x	xx	x	x	xx	2500
Brzr	"0111"	0	xx	0	0	xx	1	x	xx	x	x	xx	80
Load	"1000"	x	xx	x	x	xx	x	0	"00"	0	1	"00"	4
Store	"1001"	x	xx	x	x	xx	x	0	"00"	1	0	xx	8
Move High	"1010"	x	xx	x	x	xx	x	1	xx	0	1	"11"	47
Move Low	"1011"	x	xx	x	x	xx	x	1	xx	0	1	"10"	46
Add	"1100"	x	xx	x	x	xx	x	0	"00"	0	1	"01"	5
Sub	"1101"	x	xx	x	x	xx	x	0	"01"	0	1	"01"	15
And	"1110"	x	xx	x	x	xx	x	0	"10"	0	1	"01"	25
Or	"1111"	x	xx	x	x	xx	x	0	"11"	0	1	"01"	35

Figura 9: Sinais da ROM

- 1 bit de controle *S_Reg_D* (Registrador destino do escalar)
- 2 bits de controle *S_ALU_Control* (Controle da ULA escalar)
- 1 bit de controle *S_WE* (Write Enable da RAM escalar)
- 1 bit de controle *S_RWE* (Write Enable do Banco de Registradore do escalar)
- 2 bits de controles *Reg_Src_Scalar* (Escolha do dado de saída do escalar)
- 1 bit de controle *Branch*
- 1 bit de controle *V_Reg_D* (Registrador destino do vetorial)
- 2 bits de controle *V_ALU_Control* (Controle da ULA vetorial)
- 1 bit de controle *V_WE* (Write Enable da RAM vetorial)
- 1 bit de controle *V_RWE* (Write Enable do Banco de Registradore do vetorial)
- 2 bits de controles *Reg_Src_Vector* (Escolha do dado de saída do vetorial)

3 ASSEMBLY

O algoritmo em assembly que faz teste de cada instrução, com suas operações em binário e hexadecimal para ROM da memória de instruções:

```
// Bloco de teste do escalar
movh    1          // r1 = 16            00100001   21
movl    1          // r1 = 17            00110001   31
sub     01 01      // r1 = 0             01010101   55
movl    10         // r1 = 10            00111010   3a
add     10 01      // r2 = 10            01001001   49
and     11 01      // r3 = 0             01101101   6d
st      01 00      // M[0] = 10          00010100   14
ld      11 00      // r3 = 10            00001100   c
brzr    00 01      // jump -> 10         01110001   71
// linha dentro do if que deve ser saltado
add     10 10      // r2 = 20 (teste)    01001010   4a

// Bloco de teste do vetorial
movh    1          // r1 = 16            10100001   a1
movl    1          // r1 = 17            10110001   b1
sub     01 01      // r1 = 0             11010101   d5
movl    3          // r1 = 3             10110010   b3
add     01 00      // r1 = 3 + {0,1,2,3} 11000100   c4
or      11 01      // r3 = r1|r3         11111101   fd
and     11 00      // r3 = r0 && r3      11101100   ec
st      01 00      // M[R[r0]] = R[r1]   10010100   94
load    10 00      // R[r2] = M[R[r0]]   10001000   88

// Aqui faz o HALT
add     01 01      // r1 = 20            01000101   45
brzr    00 01      // jump -> 20         01110001   71
```

O algoritmo em assembly que faz o armazenamento dos dados na memória RAM e faz a soma dos elementos de cada vetor:
A instrução com S na frente são instruções do tipo escalar e as instruções com V na frente são do tipo vetorial.

```
// Aqui inicializa os dados no processador escalar para fazer o loop
S_movl 1          // S_R1 = 1           00110001      31
S_movh 1          // S_R1 = 17          00100001      21
S_st   01 00      // M[S_R0] = 17       00010100      14
S_sub  01 01      // S_R1 = 0           01010101      55
S_movl 1          // S_R1 = 1           00110001      31
S_add  11 01      // S_R3 = 1           01001101      4d
S_movl 3          // S_R1 = 3           00110011      33
S_movh 2          // S_R1 = 35          00100010      22
S_st   01 11      // M[S_R3] = S_R1     00010111      17
S_sub  01 01      // S_R1 = 0           01010101      55
S_movl 4          // S_R1 = 4           00110100      33

// Aqui inicializa os valores dos registradores vetoriais
V_movl 4          // V_R1 = 4           10110100      b4
V_movh 1          // V_R1 = 20          10100001      a1
V_add  10 01      // V_R2 = 20          11001001      c9
V_sub  01 01      // V_R1 = 0           11010101      d5
V_add  11 00      // V_R3 = 0..         11001100      cc
V_add  10 00      // V_R2 = 20..        11001000      c8

// Aqui e o primeiro loop
// Loop para inicializar os valores dos vetores na RAM
S_ld   10 11      // S_R2 = M[S_R3]     00001011      b
V_st   11 11      // M[V_R3] = V_R3     10011111      9f
V_movl 12         // V_R1 = 12          10111100      bc
V_add  11 01      // V_R3 = 12..        11001101      cd
V_st    10 11     // M[V_R3] = V_R2     10011011      9b
V_add  11 01      // V_R3 = 24..        11001101      cd
V_movl  0         // V_R1 = 0           10110000      b0
V_st   01 11      // M[V_R3] = V_R1     10010111      97
V_movl 12         // V_R1 = 12          10111100      bc
V_sub  11 01      // V_R3 = 12..        11011101      dd
V_sub  11 01      // V_R3 = 0..         11011101      dd
V_movl 4          // V_R1 = 4           10110100      b4
V_add  11 01      // V_R3 = 4..         11001101      cd
V_add  10 01      // V_R2 = 24..        11001001      c9
S_sub  01 11      // S_R1=S_R1-S_R3     01010111      57
S_brzr 01 10      // Se R1=0,jump r2    01110110      76
S_ld   10 00      // S_R2 = M[S_R0]     00001000      8
S_brzr 00 10      // Senao jump r2      01110010      72

// Aqui e fora depois do loop
// Faz a inicializacao dos valores no processador escalar para outro loop
S_movl 11         // S_R1 = 11          00111011      3b
S_add  10 01      // S_R2 = 46          01001001      49
```

```
S_st   10 00      // M[S_R0] = S_R2     00011000      18
S_movl 2          // S_R1 = 2           00110010      32
S_movh 1          // S_R1 = 18          00100001      21
S_add  10 01      // S_R2 = 58          01001001      49
S_st   10 11      // M[S_R3] = S_R2     00011011      1b
S_sub  01 01      // S_R1 = 0           01010101      55
S_movl 3          // S_R1 = 3           00110011      33

// Inicializacao dos valores do processador vetorial
V_sub  11 11      // V_R3 = 0           11011111      df
V_add  11 00      // V_R3 = 0..         11001100      cc

// Aqui comeca o loop para fazer soma entre o Vetor A e Vetor B e armazena em R
S_ld   10 11      // S_R2 = M[S_R3]     00001011      b
V_ld   10 11      // V_R2 = M[V_R3]     10001011      8b
V_movl 12         // V_R1 = 12          10111100      bc
V_add  11 01      // V_R3 = 12..        11001101      cd
V_ld   01 11      // V_R1 = M[V_R3]     10000111      87
V_add  10 01      // V_R2 += V_R1       11001001      c9
V_sub  01 01      // V_R1 = 0           11010101      d5
V_movl 12         // V_R1 = 12          10111100      bc
V_add  11 01      // V_R3 = 24..        11001101      cd
V_st   10 11      // M[V_R3] = V_R2     10011011      9b
V_sub  11 01      // V_R3 = 12..        11011101      dd
V_sub  11 01      // V_R3 = 0..         11011101      dd
V_movl 4          // V_R1 = 4           10110100      b4
V_add  11 01      // V_R3 = 4..         11001101      cd
S_sub  01 11      // S_R1 -= S_R3       01010111      57
S_brzr 01 10      // Se R1=0,jump r2    01110110      76
S_ld   10 00      // S_R2 = M[S_R0]     00001000      8
S_brzr 00 10      //Senao jump r2       01110010      72

// HALT
S_brzr 00 10      // Halt               01110010      72
```