Universidade Federal do Paraná

CI1212 - Arquitetura de Computadores

PROCESSADOR SAGUI 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, 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, mostrado na imagem abaixo (Figura 1), formato das instruções (Figura 2):

Opcode	Tipo	Mnemonic	Nome	Operação
Controle				
0000	R	brzr	Branch On Zero Register	if (R[ra] == 0) PC = R[rb]
0001	I	brzi	Branch On Zero Immediate	if (**R[0]** == 0) PC = PC + Imm.
0010	R	jr	Jump Register	PC = R[rb]
0011	I	ji	Jump Immediate	PC = PC + Imm.
Dados				
0100	R	ld	Load	R[ra] = M[**R[rb]**]
0101	R	st	Store	M[**R[rb]**] = R[ra]
0110	R	movr	Move Register	R[ra] = R[rb]
0111	I	movh	Move High	R[0] = {Imm. + R[0](3:0)}
1000	I	movl	Move Low	R[0] = {R[0](7:4) + Imm.}
Aritmética				
1001	R	add	Add	R[ra] = R[ra] + R[rb]
1010	R	sub	Sub	R[ra] = R[ra] - R[rb]
Lógica				
1011	R	and	And	R[ra] = R[ra] & R[rb]
1100	R	or	Or	R[ra] = R[ra] | R[rb]
1101	R	not	Not	R[ra] = ! R[rb]
1110	R	slr	Shift Left Register	R[ra] = R[ra] << R[rb]
1111	R	srr	Shift Right Register	R[ra] = R[ra] >> R[rb]

Figura 1: A arquitetura Sagui

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 em papel de uma forma bem simplificado, ou seja, sendo um racunho (Figura 3). Mais tarde o desenvolvimento de alguns detalhes foram modificados

diretamente no Software Logisim (Figura 4):

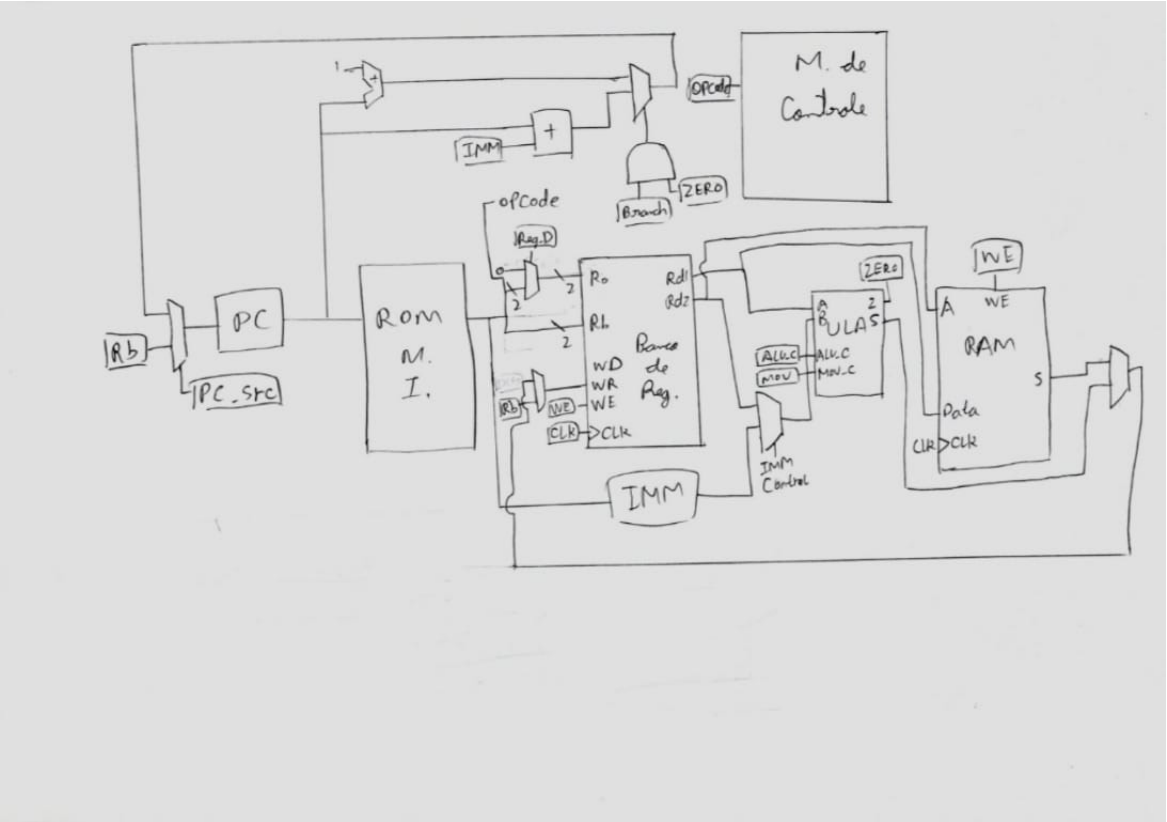


Figura 3: Datapath

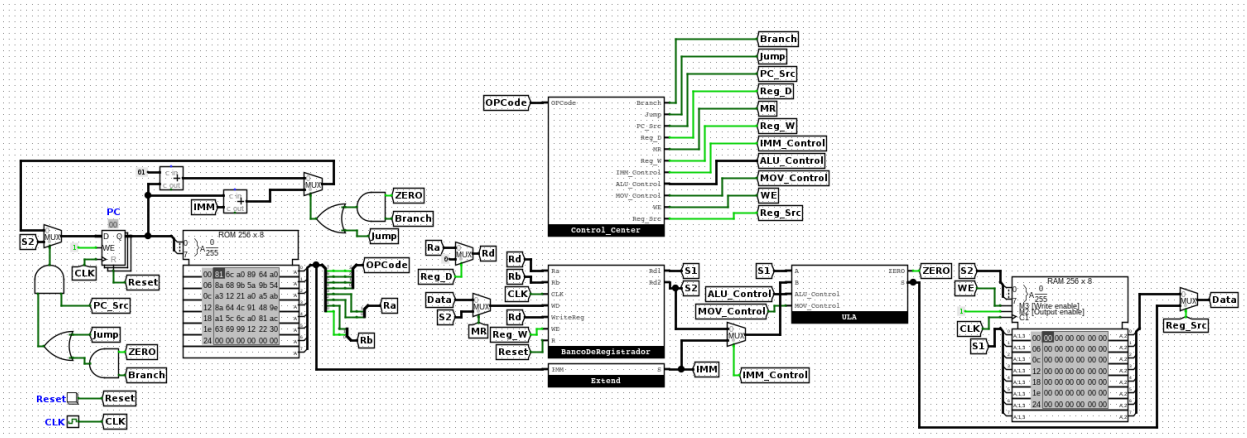


Figura 4: Datapath

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, seguinda da arquitetura Sagui (Figura 1). Alguns detalhes do planejamento da ULA, foi feito diretamento no Logisim.

O planejamento da ULA (figura 5):

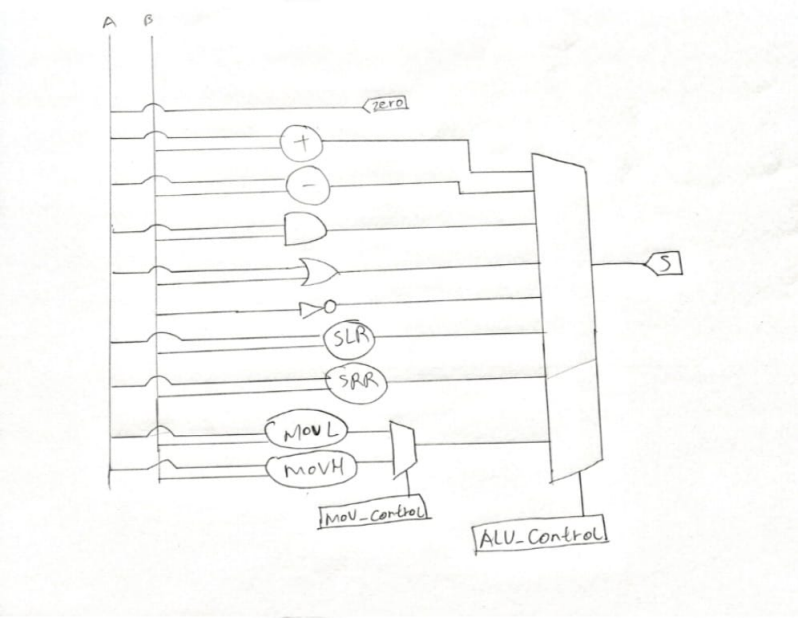


Figura 5: Circuito da ULA no Planejamento

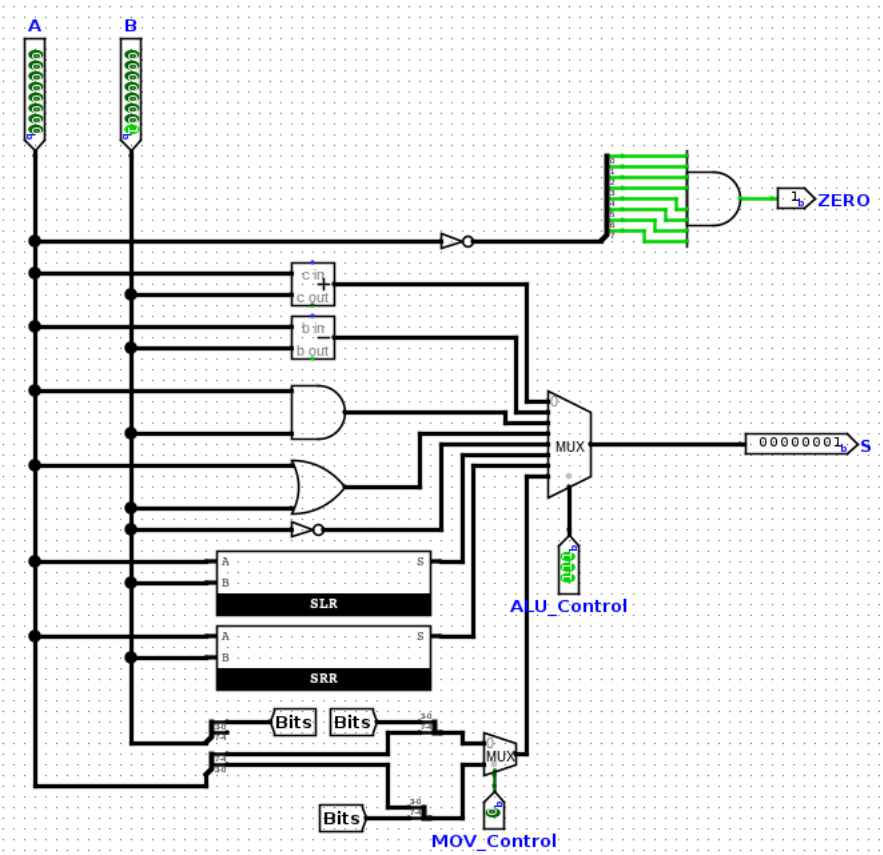
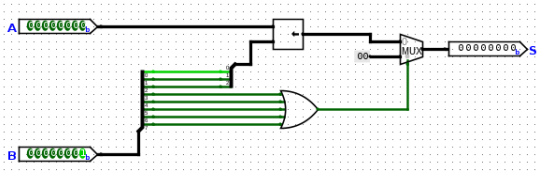


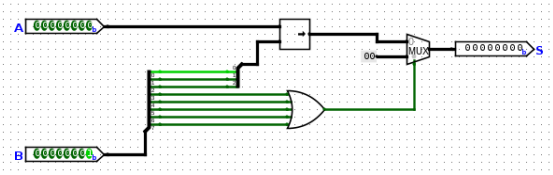
Figura 6: Circuito da ULA no Logisim

2.2.1 SLR e SRR

SLR = Shift Left Register (Figura 7a): SRR = Shift Right Register (Figura 7b):



(a) Shift Left Register



(b) Shift Right Register

Figura 7: SLR e SRR

2.2.2 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
000	ADD
001	SUB
010	AND
011	OR
100	NOT
101	SLR
110	SRR
111	MOVS

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 8).

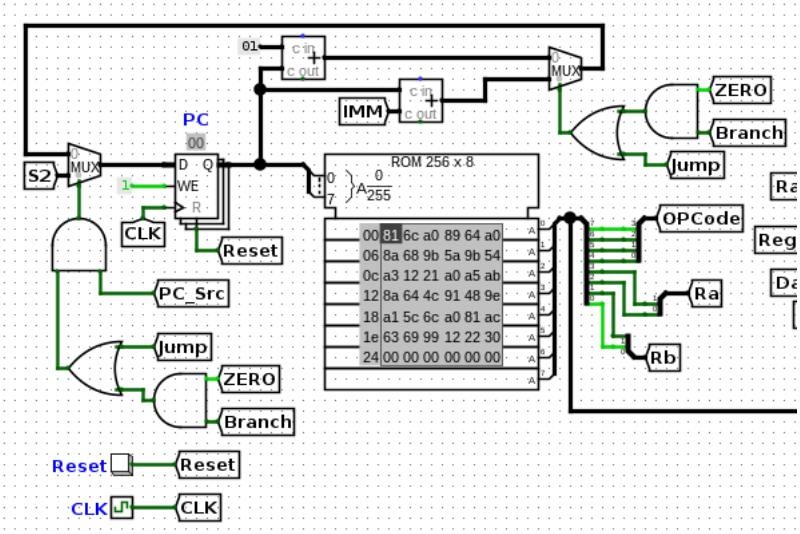


Figura 8: PC

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 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 esquerda 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 e Jump, fazem $PC = PC + imm$ ou $PC = R[rb]$, então necessitou utilizar um MUX para receber o PC novo vindo ou do $PC + 1$ ou dos casos citados anteriormente.

O controle principal do PC é o MUX, é o sinal do PC_Src (*PC source*), vindo da memória de controle, para controlar o PC vindo ou de $PC = PC + imm$, $PC = R[rb]$ ou $PC = PC + 1$.

Depois da memória de instruções é notável que possui vários distribuidores de sinais, como: *OPCode*, *Ra*, *Rb*, esses são os sinais que será necessário para memória de controle, banco de registradores.

Legenda:

- *OPCode* código da instrução
- *Ra* registrador A
- *Rb* registrador B

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 WriteReg (*Register Write*, um sinal de 1 bit vinda da memória de controle), para verificar se escreve ou não em algum registrador do banco de registradores (Figura 9).

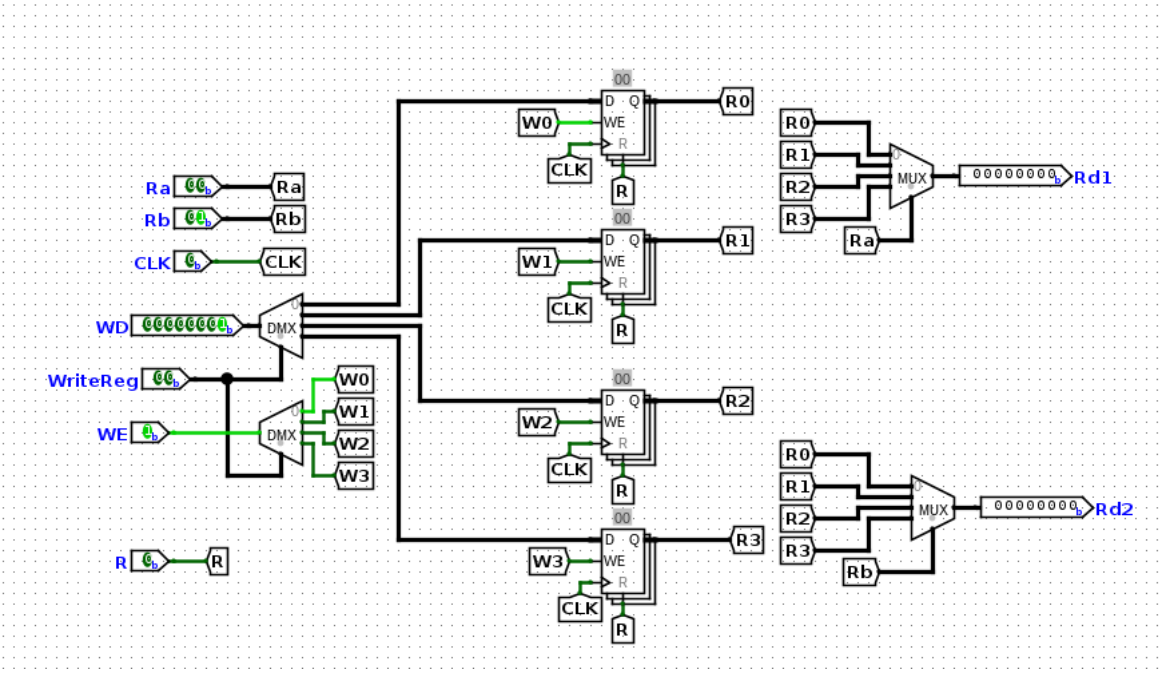


Figura 9: Banco de Registrador

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.

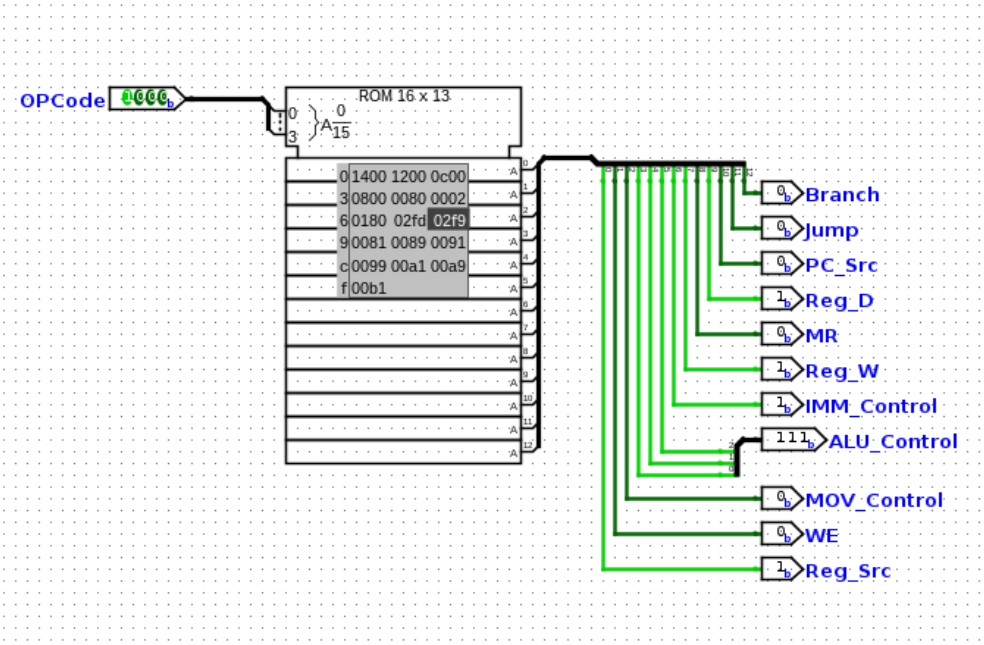


Figura 10: Memória de Controle

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

OPCode	Branch	Jump	PC_Src	Reg_D	MR	Reg_W	IMM_Control	ALU_Control	MOV_Control	WE	Reg_Src	Hexadecimal
"0000"	1	0	1	0	0	0	0	xxx	x	0	x	1400
"0001"	1	0	0	1	0	0	0	xxx	x	0	x	1200
"0010"	0	1	1	x	0	0	0	xxx	x	0	x	c00
"0011"	0	1	0	x	0	0	0	xxx	x	0	x	800
"0100"	0	0	0	0	0	1	0	"000"	x	0	0	80
"0101"	0	0	0	0	0	0	0	"000"	x	1	x	2
"0110"	0	0	0	x	1	1	0	xxx	x	0	x	180
"0111"	0	0	0	1	0	1	1	"111"	1	0	1	2fd
"1000"	0	0	0	1	0	1	1	"111"	0	0	1	2f9
"1001"	0	0	0	0	0	1	0	"000"	x	0	1	81
"1010"	0	0	0	0	0	1	0	"001"	x	0	1	89
"1011"	0	0	0	0	0	1	0	"010"	x	0	1	91
"1100"	0	0	0	0	0	1	0	"011"	x	0	1	99
"1101"	0	0	0	0	0	1	0	"100"	x	0	1	a1
"1110"	0	0	0	0	0	1	0	"101"	x	0	1	a9
"1111"	0	0	0	0	0	1	0	"110"	x	0	1	b1

Figura 11: Sinais da ROM

- 1 bit de controle *Branch*
- 1 bit de controle *Jump*
- 1 bit de controle *PC_Src*
- 1 bit da controle *Reg_D* (Registrador destino)
- 1 bit de controle *MR* (Memory Read)
- 1 bit de controle *Reg_W* (Register Write)
- 1 bit de controle *IMM_Control* (controle do MUX das operações de imediato do *EXTEND*)
- 3 bits de controles *ALU_Control* (Controle das operações da ULA)
- 1 bit de controle *MOV_Control* (Controle do move high e do move low)
- 1 bit de controle *WE* (Controle do Write Enable)
- 1 bit de controle *Reg_Src* (Controle do tipo de dados passado para banco de registrador)

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:

```
ji 2          //(jump para inst. 3)  00110010   32
ji 0          //(NOP)                00110000   30
movh 1        //(r0 = 00010000)      01110001   71
sub r0 r0     //(r0 = 0)             10100000   a0
movl 255      //(r0 = 0f)            10001111   8f
movr r1 r0    //(r1 = 0f)            01100100   64
sub r0 r0     //(r0 = 0)             10100000   a0
movh 255      //(r0 = f0)            01111111   7f
add r1 r0     //(r1 = ff)            10010100   94
sub r2 r1     //(r2 = 1)             10101001   a9
and r1 r2     //(r1 = 1)             10110110   b6
or r1 r0      //(r1 = f1)            11000100   c4
not r2 r2     //(r2 = fe)            11011010   da
brzr r0 r1    //(branch se r0 == 0)  00000001   01
brzi r0 1     //(branch se r0 == 0)  00010001   11
sub r0 r0     //(r0 = 0)             10100000   a0
movl 12       //(r0 = c)             10001100   8c
add r3 r0     //(r3 = c)             10011100   9c
add r3 r0     //(r3 = 18)            10011100   9c
sub r0 r0     //(r0 = 0)             10100000   a0
brzr r0 r3    //(jump para inst.24)  00000011   03
not r3 r3     //dentroBranch         11011010   da
brzr r0 r0    //dentroBranch         00000000   00
not r2 r2     //dentroBranch         11011010   da
movl 1        //(r0 = 1)             10000001   81
slr r2 r0     //(r2 = fc)            11101000   e8
st r1 r0      //(M[R[r0]] == R[r2])  01010100   58
srr r2 r0     //(r2 = 7e)            11111000   f8
load r2 r0    //(R[r2] == M[R[r0]])  01000100   44
jr r0         //(jump R[r0])         00100000   20
```

O algoritmo em assembly que faz o armazenamento dos dados na memória RAM e faz a soma dos elementos de cada vetor:

```
// Aqui o bloco inteiro faz inicializao dos registradores
// r0 = 1, r1 = 9, r2 = 11, r3 = 1
movl 1     // r0 = 1             10000001   81
movr r3 r0 // r3 = 1             01101100   6c
sub r0 r0  // r0 = 0 (reset)     10100000   a0
movl 9     // r0 = 9             10001001   89
movr r1 r0 // r1 = 9             01100100   64
sub r0 r0  // r0 = 0 (reset)     10100000   a0
movl 10    // r0 = 10            10001010   8a
movr r2 r0 // r2 = 10            01101000   68
add r2 r3  // r2 = 11            10011011   9b

// Aqui o bloco  um loop que armazena os valores na memria RAM
st r2 r2   // M[R[r2]] = R[r2]   01011010   5a
add r2 r3  // r2 += 1            10011011   9b
st r1 r0   // M[R[r0]] = R[r1]   01010100   54
sub r0 r3  // r0 -= 1            10100011   a3
brzi 2     // PC + 2             00010010   12
jr r0 r1   // Jump r1 (9)        00100001   21
// Aqui   o bloco de reset
sub r0 r0  // r0 = 0 (reset)     10100000   a0
sub r1 r1  // r1 = 0 (reset)     10100101   a5
sub r2 r3  // r2 = 20            10101011   ab

// Aqui   o bloco de deixar os registradores com dados para utilizao
movl 10    // r0 = 10            10001010   8a
movr r1 r0 // r1 = 10            01100100   64

// Aqui inicia o loop da soma dos elementos
// Aqui   o bloco que faz a soma dos elementos e armazena nas posies do vetor 1
ld r3 r0   // R[r3] = M[R[r0]]   01001100   4c
add r0 r1  // r0 += 10           10010001   91
ld r2 r0   // R[r2] = M[R[r0]]   01001000   48
add r3 r2  // r3 += r2           10011110   9e
sub r0 r1  // r0 -= 10           10100001   a1
st r3 r0   // M[R[r0]] = R[r3]   01011100   5c

// Aqui   o bloco que faz o reset dos valores dos registradores
movr r3 r0 // r3 = r0            01101100   6c
sub r0 r0  // r0 = 0 (reset)     10100000   a0
movl 1     // r0 = 1             10000001   81
sub r3 r0  // r3 = r3 - 1        10101100   ac
movr r0 r3 // r0 = r3            01100011   63
movr r2 r1 // r2 = 10            01101001   69
add r2 r1  // r2 = 20            10011001   99
brzi 2     // PC + 2             00010010   12
jr r0 r2   // Jump r2 (20)       00100010   22
ji 0       //  NOP               00110000   30
```