Relatório Sagui

Andre Grassi de Jesus
Universidade Federal do Paraná – UFPR
Curitiba, Brasil
agj23@inf.ufpr.br

I. Introdução

Esse relatório consiste na exposição da implementação da arquitetura Sagui, bem como uma breve explicação à cerca dos seus componentes. Além disso serão apresentados dois programas de teste desenvolvidos para essa arquitetura.

O projeto foi criado utilizando o software de simulação de circuitos digitais *Logisim Evolution*.

A nomenclatura de entradas e a ideia geral do projeto foi baseada na arquitetura *RISC-V* apresentada no livro *Computer Organization and Design: the Hardware/Software Interface* [1].

II. Visão Geral

Para construir o circuito, foram utilizados, principalmente, os seguintes componentes:

- **Clock**.
- **PC**: consiste em um registrador de 8 bits.
- **ROM**: memória de instruções endereçada a 8 bits, guardando 8 bits em cada endereço.
- **Banco de registradores**: contém 4 registradores de 8 bits cada.
- **ULA**.
- **RAM**.
- **Ctrl**: circuito que atualiza os sinais de controle.
- **BranchCtrl**: circuito que controla qual será o próximo valor do PC.

Uma visão mais detalhada será apresentada a seguir

III. Sinais de Controle

Os sinais de controle são gerados por dois circuitos, com diferentes propósitos. Em relação ao circuito Crtl, os sinais são os seguintes:

- **RegWrite** (1 bit): indica se a escrita no banco de registradores deve ser habilitada ou não.
- **RamWrite** (1 bit): indica se habilita escrita na RAM.
- **WbOp** (2 bits): indica o que deve ser escrito no banco de registradores na etapa de *write back* (0: saída da ULA, 1: saída da RAM, 2: saída do Read Data 2, 3: é um caso *don't care*)
- **AluOp** (4 bits): indica qual operação deve ser realizada na ULA.
- **JorB** (1 bit): indica se a instrução é Jump ou Branch (1) ou se não é (0).
- **SelectR0** (1 bit): indica se deve ser selecionado obrigatoriamente o R[0] em uma das entradas do banco.
- **UseImm** (1 bit): indica o uso de imediato.

Já o BranchCtrl servirá para controlar qual será a operação que o PC irá sofrer. A sua saída, nomeada **PcOp** (Pc Operation) seleciona qual sinal será propagado por um MUX, em que temos PC + 4, PC = R[rb] e PC + imm. Para tanto, é necessário fornecer como entrada para esse circuito: o BranchOp (indica qual o tipo de Branch/Jump com base nos 2 primeiros bits do opcode), o sinal JorB e o sinal Zero (indica se a R[ra] ou R[0] tem o valor zero).

IV. ULA

A ULA recebe três entradas, que são os **operandos** (Operand1 e Operand2) e o sinal de controle **ALUOp** (ALU Operation) que escolhe qual operação será propagada à saída **Result**. As operações são as seguintes:

0) **Add**: soma operando 1 e 2.
1) **Sub**: subtrai o operando 2 do 1.
2) **And**: and entre operandos.
3) **Or**: or entre operandos.
4) **Not**: nega operando 2.
5) **Slr**: faz o shift left do operando 1, deslocando o número indicado pelos 3 bits menos significativos do operando 2.
6) **Srr**: faz o shift right do operando 1, deslocando o número indicado pelos 3 bits menos significativos do operando 2
7) **Movh**: concatena os 4 LSB (*Least significat bits*) do operando 2 com os 4 LSB do operando 1, colocando os bits do operando 2 na parte alta.
8) **Movl**: concatena, colocando os 4 LSB do operando 2 na parte baixa.

V. Programas de Teste

Para desenvolvimento dos programas de teste foi utilizado um "pseudo-assembly", semelhante ao RISC-V, mas com as instruções do Sagui.

A. Teste das Instruções

Nesse programa foi realizado um teste de todas as 16 instruções da arquitetura. O assembly é o seguinte:

```
#          "Pseudo-assembly" da arquitetura Sagui                #
#      ##############################################            #
#                                                                #
#          Organização dos comentários dos comandos:             #
#                                                                #
# mnem. # sign. da operação --> # instr. em bin. = instr. em hex. #


######## MOV ########

# Inicializa registrador 0 com 00000000
# Como em uma cpu real, teríamos lixo de memória nos registradores,
# o que é representado por "?"
movh 0000   # R[0] = {Imm + R[0](3:0)} = {0000 + ????} = 0000???? = ? --> 01110000 = 70
movl 0000   # R[0] = {R[0](7:4) + Imm.} = {0000 + 0000} = 00000000 = 0 --> 10000000 = 80

# Coloca o valor 112 no R[0]
movh 0111   # R[0] = {Imm + R[0](3:0)} = {0111 + 0000} = 01110000 = 112 --> 01110111 = 77

# Reseta R[0]
movh 0000   # R[0] = {Imm + R[0](3:0)} = {0000 + 0000} = 00000000 = 0 --> 01110000 = 70

# Coloca o valor 7 no R[0]
movl 0111   # R[0] = {R[0](7:4) + Imm.} = {0000 + 0111} = 00000111 = 7 --> 10000111 = 87

# Inicializa registradores 1, 2 e 3 movendo o valor de R[0] para eles
movr $1, $0 # R[1] = R[0] = 7 --> 01100100 = 64
movr $2, $1 # R[2] = R[1] = 7 --> 01101001 = 69
movr $3, $2 # R[3] = R[2] = 7 --> 01101110 = 6E


######## ADD ########

# Soma de positivo com positivo
add $2, $1  # R[2] = R[2] + R[1] = 7 + 7 = 14 --> 10011001 = 99

# Reseta R[0]
movl 0000   # R[0] = {R[0](7:4) + Imm.} = {0000 + 0000} = 00000000 = 0 --> 10000000 = 80

# Coloca o valor -16 no R[0]
movh 1111   # R[0] = {Imm + R[0](3:0)} = {1111 + 0000} = 11110000 = -16 --> 01111111 = 7F

# Soma de positivo com negativo
add $3, $0 # R[3] = R[3] + R[0] = 7 + (-16) = -9 --> 10011100 = 9C

# Soma de negativo com negativo
add $3, $0 # R[3] = R[3] + R[0] = -9 + (-16) = -25 --> 10011100 = 9C


######## SUB ########

# Subtração de positivo com positivo
sub $2, $1 # R[2] = R[2] - R[1] = 14 - 7 = 7 --> 10101001 = A9

# Subtração de positivo com negativo
sub $2, $3 # R[2] = R[2] - R[3] = 7 - (-25) = 32 --> 10101011 = AB

# Subtração de negativo com negativo
sub $3, $0  # R[3] = R[3] - R[0] = -25 - (-16) = -9 --> 10101100 = AC


######## AND ########

# And de valores diferentes
and $2, $1 # R[2] = R[2] & R[1] = 32 & 7 = 0 --> 10111001 = B9

# And de valores iguais
and $1, $1 # R[1] = R[1] & R[1] = 7 & 7 = 7 --> 10110101 = B5

######## OR ########

# Or de valores diferentes
or $1, $0 # R[1] = R[1] | R[0] = 7 | -16 = -9 --> 11000100 = C4

# Or de valores iguais
or $1, $1 # R[1] = R[1] | R[1] = -9 | -9 = -9 --> 11000101 = C5

######## NOT ########
not $1, $1 # R[1] = !R[1] = !(-9) = 8 --> 11010101 = D5

######## SHIFT ########
# R[1] = 1
movh 0000   # R[0] = {Imm + R[0](3:0)} = {0000 + 0000} = 00000000 = 0 --> 01110000 = 70
```

```
movl 0001   # R[0] = {R[0](7:4) + Imm.} = {0000 + 0001} = 00000001 = 1 --> 10000001 = 81
movr $1, $0 # R[1] = R[0] = 1 --> 01100100 = 64

# Multiplica R[1] por 2
slr $1, $0 # R[1] = R[1] << R[0] = 1 << 1 = 2 --> 11100100 = e4

# Divide R[1] por 2
srr $1, $0 # R[1] = R[1] >> R[0] = 2 >> 1 = 1 --> 11110100 = f4

# R[0] = 7 = 00000111
movl 0111  # R[0] = {R[0](7:4) + Imm.} = {0000 + 0111} = 00000111 = 7 --> 10000111 = 87

# Máximo de slr possível, causando overflow
slr $1, $0 # R[1] = R[1] << R[0] = 1 << 7 = -128 --> 11100100 = E4

# Máximo de srr possível, causando underflow
srr $1, $0 # R[1] = R[1] << R[0] = 1 << 7 = 1 --> 11110100 = F4


######## STORE ########
st $0, $1 # M[R[1]] = R[0] <=> M[1] = 7 --> 01010001 = 51


######## LOAD ########
ld $2, $1 # R[2] = M[R[1]] <=> R[2] = M[1] --> 01001001 = 49

######## BRZR ########
# Não entra no branch
brzr $1, $0 # if (R[1] == 0) PC = R[0] = 7 --> 00000100 = 4

# Entra no branch
# Reseta R[1]
movh 0000   # 01110000 = 70
movl 0000   # 10000000 = 80
movr $1, $0 # 01100100 = 64

# R[0] = 43
movl 1011   # 10001011 = 8B
movh 0010   # 01110010 = 72

brzr $1, $0 # if (R[1] == 0) PC = R[0] = 40 --> 00000100 = 4


######## BRZI ########
# Não entra no branch
brzi 0111   # if (R[0] == 0) PC = PC + Imm = PC + 7 --> 00010111 = 17

# Entra no branch
# Reseta R[1]
movh 0000   # 01110000 = 70
movl 0000   # 10000000 = 80
movr $1, $0 # 01100100 = 64

brzi 0111   # if (R[0] == 0) PC = PC + Imm = PC + 7 --> 00010111 = 17


######## JR ########
movl 0001   # R[0] = 00000001 --> 10000001 = 81
movh 0100   # R[0] = 00110100 --> 01110100 = 74
movr $2, $0 # R[2] = R[0] = 65 = 41 (hex) --> 01101000 = 68
jr $2       # PC = R[2] = 41 --> 00100010 = 22

######## JI ########
ji 0111 # PC = PC + Imm = PC + 7 --> 00110111 = 37
```

B. Soma de Vetores

Nesse programa são inicializados dois vetores e eles são somados. Para melhor elaboração do assembly, foi desenvolvido um código simplificado em C:

```c
#include <stdio.h>

#define TAM 10

int main () {
    // Vetores teste de 10 posições
    int vet_1[TAM] = {0, -45, 25, 62, 2, 46, 73, 32, 11, 1};
    int vet_2[TAM] = {-5, 23, 65, 21, -24, -33, -1, 24, 63, 10};
    int vet_r[TAM];

    // Contador
    int cont = TAM - 1;
    int cont_ant = cont;
```

```
SOMA:
    if (cont_ant == 0)
        goto FIM;

    // Soma posição i dos vetores e guarda em vet_r
    *(vet_r + cont) = *(vet_1 + cont) + *(vet_2 + cont);

    cont_ant = cont;
    cont = cont - 1;

    goto SOMA;

FIM:
    return 0;
}
```

Já o código transformado no assembly é o seguinte:

```
# Reseta todos os registradores
movl 0 # 10000000 = 80
movh 0 # 01110000 = 70
movr $1, $0 # 01100100 =  64
movr $2, $0 # 01101000 = 68
movr $3, $0 # 01101100 = 6C

# R[0] guarda o tamanho dos vetores
movl 1010 # 10001010 = 8A
movr $1, $0 # 01100100 = 64

# Endereço 0 da memória guarda o tamanho
movl 0 # 10000000 = 80
st $1, $0 # M[0] = 10 --> # 01010100 = 54

# R[1] recebe o endereço do vetor 1
movl 1 # 10000001 = 81
movr $1, $0 # 01100100 = 64

# R[2] recebe o endereço do vetor 2
# Endereço do vetor 2 = Endereço do vetor 1 + 10
movl 1010 # 10001010 = 8A
movr $2, $0 # 01101000 = 68
add $2, $1 # 10011001 = 99

# Inicializa vetor 1 na memória
# R[3] recebe 1 para incremento do endereço
movl 1 # 10000001 = 81
movr $3, $0 # 01101100 = 6C

# vet_1[0] = 0 = 0 (hex)
movl 0 # 10000000 = 80
st $0, $1 # 01010001 = 51

# vet_1[1] = -45 = d3
add $1, $3 # R[1] = 1 + 1 = 2 --> 10010111 = 97
movl 0011 # 10000011 = 83
movh 1101 # 01111101 = 7D
st $0, $1 # 01010001 = 51

# vet_1[2] = 25 = 19
add $1, $3 # R[1] = 2 + 1 = 3 --> 10010111 = 97
movl 1001 # 10001001 = 89
movh 0001 # 01110001 = 71
st $0, $1 # 01010001 = 51

# vet_1[3] = 62 = 3E
add $1, $3 # R[1] = 3 + 1 = 4 --> 10010111 = 97
movl 1110 # 10001110 = 8E
movh 0011 # 01110011 = 73
st $0, $1 # 01010001 = 51

# vet_1[4] = 2 = 2
add $1, $3 # R[1] = 4 + 1 = 5 --> 10010111 = 97
movl 0010 # 10000010 = 82
movh 0000 # 01110000 = 70
st $0, $1 # 01010001 = 51

# vet_1[5] = 46 = 2E
add $1, $3 # R[1] = 5 + 1 = 6 --> 10010111 = 97
movl 1110 # 10001110 = 8E
movh 0010 # 01110010 = 72
st $0, $1 # 01010001 = 51

# vet_1[6] = 73 = 49
```

```
add $1, $3 # R[1] = 6 + 1 = 7 --> 10010111 = 97
movl 1001 # 10001001 = 89
movh 0100 # 01110100 = 74
st $0, $1 # 01010001 = 51

# vet_1[7] = 32 = 20
add $1, $3 # R[1] = 7 + 1 = 8 --> 10010111 = 97
movl 0000 # 10000000 = 80
movh 0010 # 01110010 = 72
st $0, $1 # 01010001 = 51

# vet_1[8] = 11 = 0B
add $1, $3 # R[1] = 8 + 1 = 9 --> 10010111 = 97
movl 1011 # 10001011 = 8B
movh 0000 # 01110000 = 70
st $0, $1 # 01010001 = 51

# vet_1[9] = 1 = 1
add $1, $3 # R[1] = 9 + 1 = 10 --> 10010111 = 97
movl 0001 # 10000001 = 81
movh 0000 # 01110000 = 70
st $0, $1 # 01010001 = 51

# Inicializa vetor 2 na memória
# vet_2[0] = -5 = FB
movl 1011 # 10001011 = 8B
movh 1111 # 01111111 = 7F
st $0, $2 # 01010010 = 52

# vet_2[1] = 23 = 17
add $2, $3 # R[2] = 11 + 1 = 12 --> 10011011 = 9B
movl 0111 # 10000111 = 87
movh 0001 # 01110001 = 71
st $0, $2 # 01010010 = 52

# vet_2[2] = 65 = 41
add $2, $3 # R[2] = 12 + 1 = 13 --> 10011011 = 9B
movl 0001 # 10000001 = 81
movh 0100 # 01110100 = 74
st $0, $2 # 01010010 = 52

# vet_2[3] = 21 = 15
add $2, $3 # R[2] = 13 + 1 = 14 --> 10011011 = 9B
movl 0101 # 10000101 = 85
movh 0001 # 01110001 = 71
st $0, $2 # 01010010 = 52

# vet_2[4] = -24 = E8
add $2, $3 # R[2] = 14 + 1 = 15 --> 10011011 = 9B
movl 1000 # 10001000 = 88
movh 1110 # 01110001 = 7E
st $0, $2 # 01010010 = 52

# vet_2[5] = -33 = DF
add $2, $3 # R[2] = 15 + 1 = 16 --> 10011011 = 9B
movl 1111 # 10001111 = 8F
movh 1101 # 01111101 = 7D
st $0, $2 # 01010010 = 52

# vet_2[6] = -1 = FF
add $2, $3 # R[2] = 16 + 1 = 17 --> 10011011 = 9B
movl 1111 # 10001111 = 8F
movh 1111 # 01111111 = 7F
st $0, $2 # 01010010 = 52

# vet_2[7] = 24 = 18
add $2, $3 # R[2] = 17 + 1 = 18 --> 10011011 = 9B
movl 1000 # 10001000 = 88
movh 0001 # 01110001 = 71
st $0, $2 # 01010010 = 52

# vet_2[8] = 63 = 3F
add $2, $3 # R[2] = 18 + 1 = 19 --> 10011011 = 9B
movl 1111 # 10001111 = 8F
movh 0011 # 01110011 = 73
st $0, $2 # 01010010 = 52

# vet_2[9] = 10 = 0A
add $2, $3 # R[2] = 19 + 1 = 20 --> 10011011 = 9B
movl 1010 # 10001010 = 8A
movh 0000 # 01110000 = 70
st $0, $2 # 01010010 = 52
```

```
# Soma dos vetores
# R[1] recebe o endereço do vetor 1 = 1
movl 1 # 10000001 = 81
movr $1, $0 # 01100100 = 64

# Guarda o endereço do vetor 1 na posição 18 (hex) da RAM
movl 1000 # 10001000 = 88
movh 0001 # 01110001 = 71
st $1, $0 # 01010100 = 54

# R[2] recebe o endereço do vetor 2 = 0b
# Endereço do vetor 2 = Endereço do vetor 1 + 10
movl 1010 # 10001010 = 8A
movh 0000 # 01110000 = 70
movr $2, $0 # 01101000 = 68
add $2, $1 # 10011001 = 99

# Guarda o endereço do vetor 2 na posição 19 (hex) da RAM
movl 1001 # 10001001 = 89
movh 0001 # 01110001 = 71
st $2, $0 # 01011000 = 58

# Guarda o endereço do vetor resultado na posição 1a da RAM
# Endereço vai começar no 20
# R[3] recebe o endereco do vetor resultado
movl 0000 # 10000000 = 80
movh 0010 # 01110010 = 72
movr $3, $0 # R[3] = 20 --> 01101100 = 6C
movl 1010 # 10001010 = 8A
movh 0001 # 01110001 = 71
st $3, $0 # M[1A] = 20 --> 01011100 = 5C

# Guarda contador na posição 1b da RAM
movl 1001 # 10001001 = 89 Cont vale 9
movh 0000 # 01110000 = 70
movr $3, $0 # R[3] = 10 --> 01101100 = 6C
movl 1011 # 10001011 = 8B
movh 0001 # 01110001 = 71
st $3, $0 # M[1B] = 10 --> 01011100 = 5C

# R[0] indica se a soma acabou, 0 = acabou, diferente de 0 = não acabou
brzr $0, $3 # 00000011 = 03
# Pega o endereço do vetor 1
movl 1000 # 10001000 = 88
movh 0001 # 01110001 = 71
ld $1, $0 # 01000100 = 44

# Pega o endereço do vetor 2
movl 1001 # 10001001 = 89
movh 0001 # 01110001 = 71
ld $2, $0 # 01001000 = 48

# Pega o endereço do vetor resultado
movl 1010 # 10001010 = 8A
movh 0001 # 01110001 = 71
ld $3, $0 # 01001100 = 4C

# Pega o contador
movl 1011 # 10001011 = 8B
movh 0001 # 01110001 = 71
ld $0, $0 # 01000000 = 40

# Obtém o índice correto dos vetores
add $1, $0 # 10010100 = 94
add $2, $0 # 10011000 = 98
add $3, $0 # 10011100 = 9C

# Pega vetor 1
ld $1, $1 # 01000101 = 45

# Pegar vetor 2
ld $2, $2 # 01001010 = 4A

# Soma os vetores
add $1, $2 # 10010110 = 96

# Guarda no endereço indicado por $3 o resultado
st $1, $3 # 01010111 = 57

# Pega o contador
movl 1011 # 10001011 = 8B
movh 0001 # 01110001 = 71
ld $1, $0 # 01000100 = 44
```

```
# Decrementa o contador
movl 0001 # 10000001 = 81
movh 0000 # 01110000 = 70
sub $1, $0 # 10100100 = A4

# Pega o contador (antigo)
movl 1011 # 10001011 = 8B
movh 0001 # 01110001 = 71
ld $2, $0 # 01001000 = 48

# Armazena novo valor do contador
movl 1011 # 10001011 = 8B
movh 0001 # 01110001 = 71
st $1, $0 # 01010100 = 54

# Guarda o valor antigo do contador em R[1]
movr $1, $2 # 01100110 = 66

# Guarda salto do jump em R[2]
movl 1101 # 10001101 = 8D
movh 1011 # 01111011 = 7B
movr $2, $0 # 01101000 = 68

# Guarda salto do branch em R[3]
movl 0110 # 10000110 = 86
movh 1110 # 01111110 = 7E
movr $3, $0 # 01101100 = 6C

movr $0, $1 # 01100001 = 61

# Volta à condição do laço
jr $2 # 00100010 = 22

# Resultado do vet_r
# [fb, ea, 5a, 53, ea, 0d, 48, 38, 4a, 0b]
# [-5, -22, 90, 83, -22, 13, 72, 56, 74, 11]
```

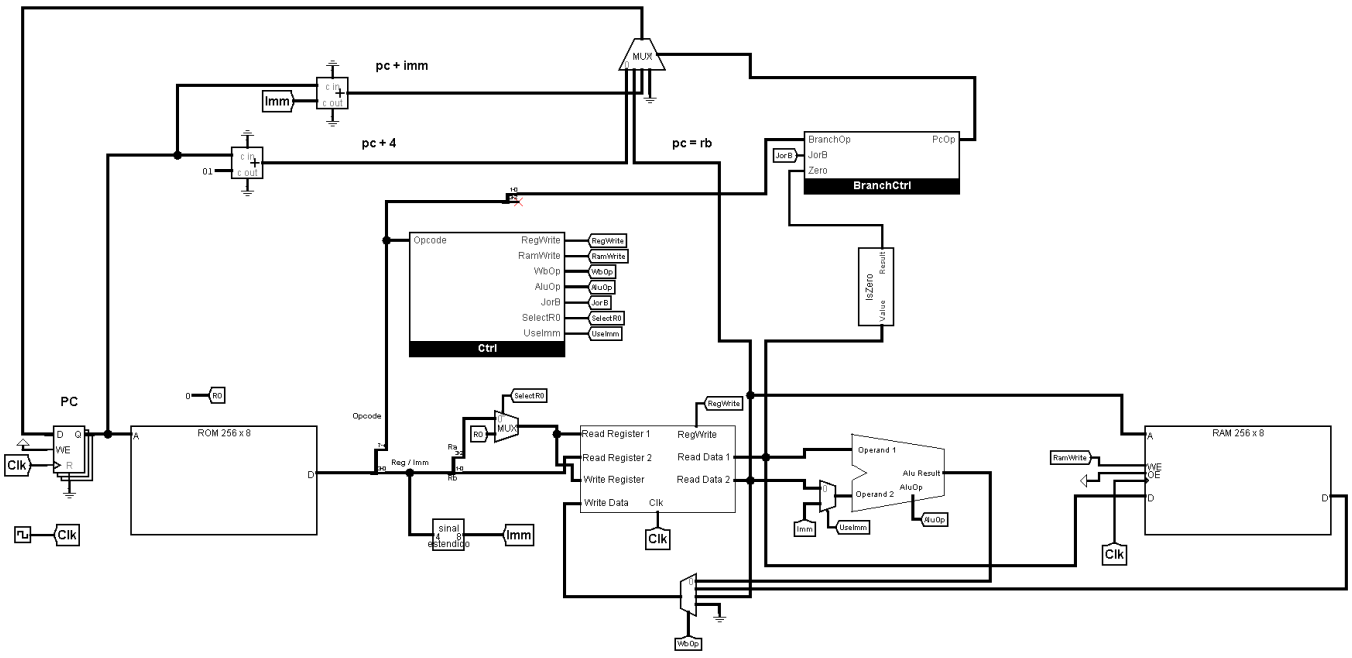

Figura 1. Imagem do processador.

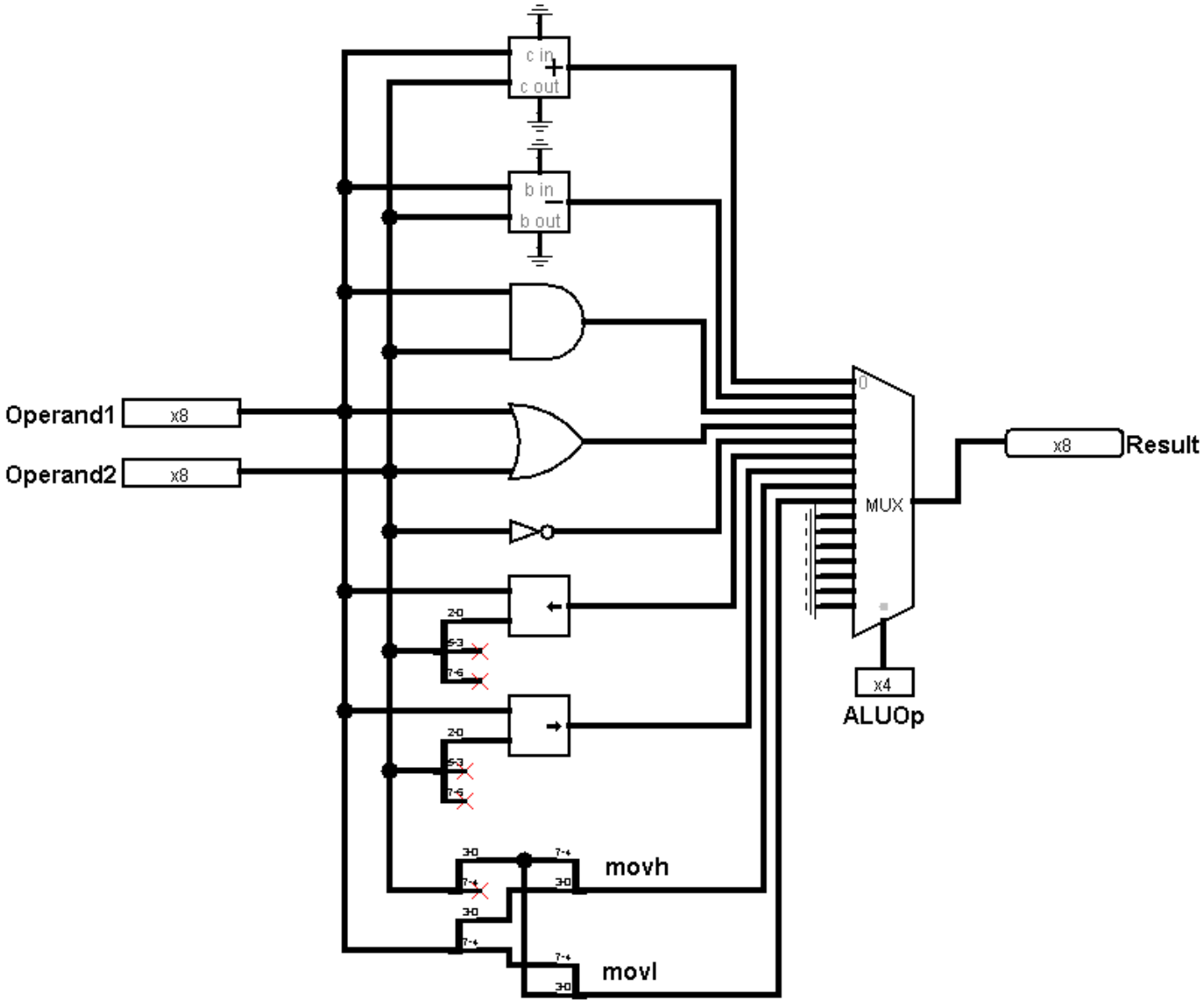

Figura 2. Imagem da ULA.

VI. Conclusão

Foi desenvolvido o projeto do processador com êxito, bem como foram elaborados programas de teste que podem ser transformados em uma memória ROM e executados no *Logisim Evolution*.

Referências

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2004.