

Relatório da disciplina de Arquitetura e Organização de Computadores

Eduardo Kurek¹, Márcio Gabriel Pereira de Campos²

Universidade Tecnológica Federal do Paraná – UTFPR

COCIC – Coordenação do Curso de Bacharelado em Ciência da Computação

Campo Mourão, Paraná, Brasil

¹eduardokurek@alunos.utfpr.edu.br

²marcioc@alunos.utfpr.edu.br

• Introdução

O trabalho trata na implementação de componentes essenciais em um caminho de dados monociclo, feito no simulador Logisim.

• ULA

A Unidade Lógica e Aritmética (ULA) é um componente responsável por efetuar operações lógicas e aritméticas. Ela conta com duas entradas (A e B), de 32 bits cada, uma terceira entrada de 4 bits (OPCODE), que indica qual operação será efetuada, e uma quarta entrada de 5 bits (SHAMT), que indica quantos bits serão deslocados em uma operação de deslocamento. A ULA tem ao todo 3 saídas: overflow, com 1 bit, e indica se a soma ou subtração deram overflow, de acordo com a tabela verdade representada pelo circuito da figura abaixo ; result, com 32 bits e que indica o retorno da operação; zero, com 1 bit, e retorna 1, caso todos os bits do result sejam 0, ou 0 caso contrário.

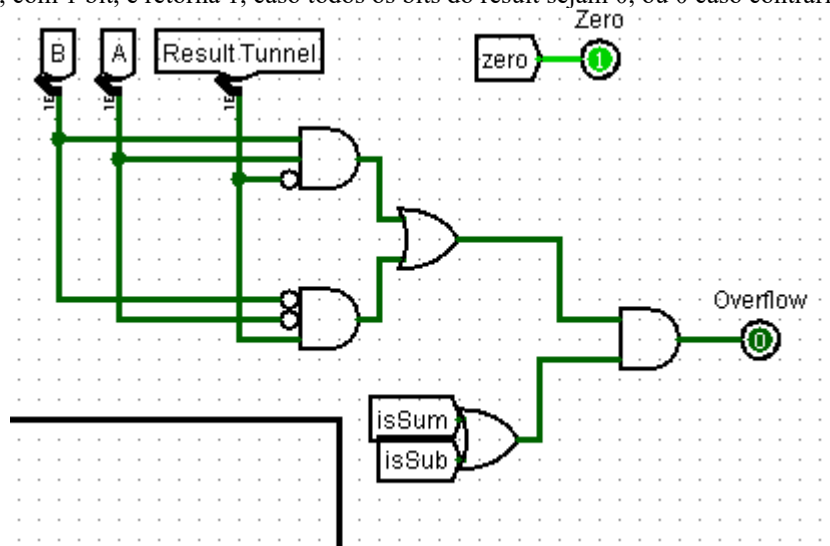


Figura: Circuito verificador de Overflow

Entradas

As entradas foram nomeadas como: Entrada A, Entrada B, Operação ALU e SHAMT

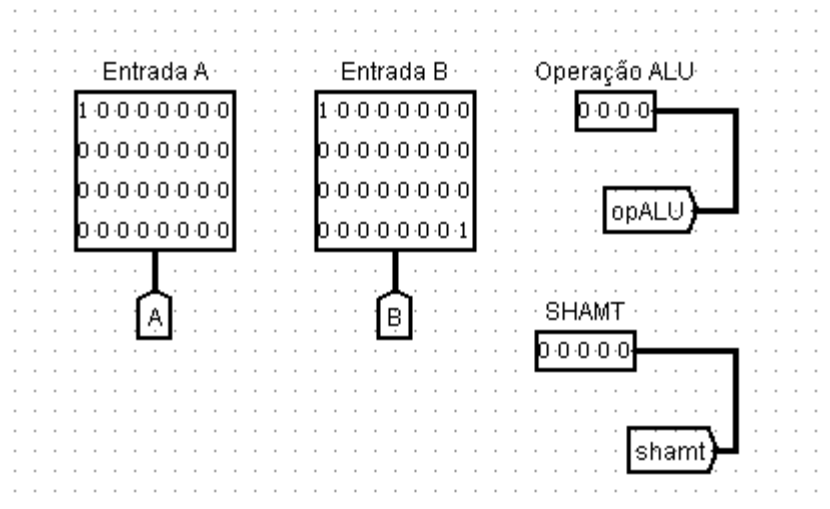


Figura: Operadores de entrada

Operações

As operações foram implementadas seguindo a ordem descrita na figura abaixo:

10100 - AND = 20

10101 - OR = 21

10111 - NOR = 23

10110 - XOR = 22

11010 - SLT = 26

10000 - ADD = 16

10010 - SUB = 18

00000 - SLL = 0

00010 - SRL = 2

Figura: Código de operações

As operações escolhidas têm esses códigos devido ao campo **funct** das instruções do tipo **R**. O campo funct tem 6 bits, porém para utilização das operações acima, é necessário olhar apenas para os bits: 5, 3, 2, 1, 0, pois o bit 4 é sempre 0 em todas as operações acima, ou seja, ele não é necessário para diferenciá-las. O código das operações é composto então pelos seguintes bits de funct que estarão na cor azul:

funct de AND: 100100

Operação AND, OR, NOR, XOR

As operações AND, OR, NOR, XOR, foram implementadas de forma simples, apenas utilizando as portas lógicas já existentes no Logisim:

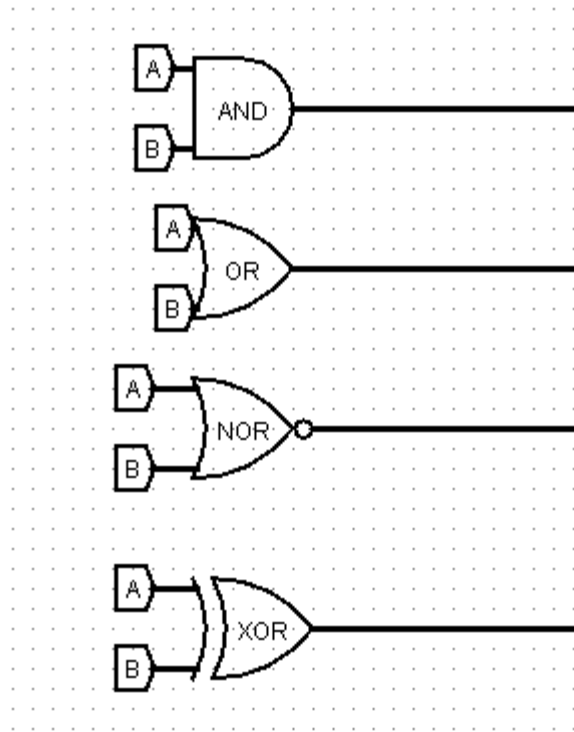


Figura: Implementação AND, OR, NOR, XOR

Operação SLT

A operação SLT foi implementada utilizando um comparador, comparando o valor de A com B, e em

seguida preenchendo os outros 31 bits com 0, pois o único valor retornado no comparador é o do primeiro bit, sendo 1 para quando for menor, e 0 para quando não for.

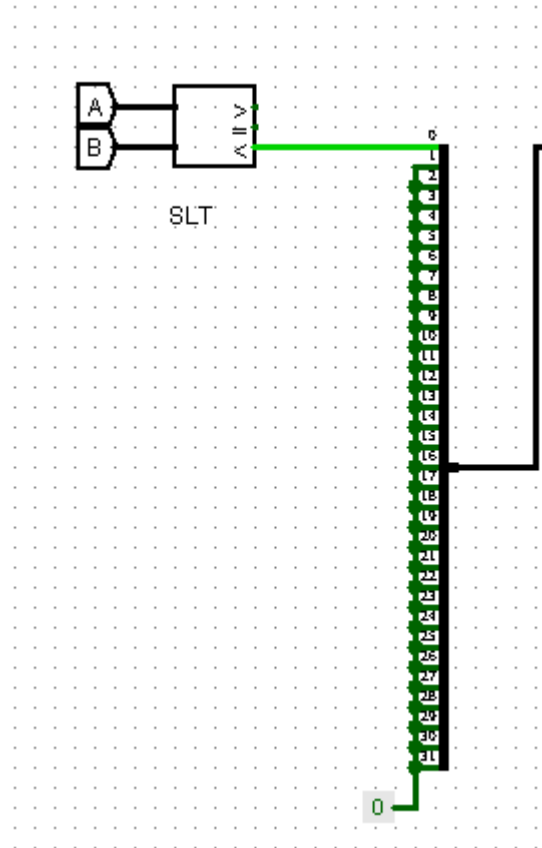


Figura: Implementação SLT

Operação ADD e SUB

Para as operações ADD e SUB, foram utilizados o Somador e Subtrator que o Logisim oferece

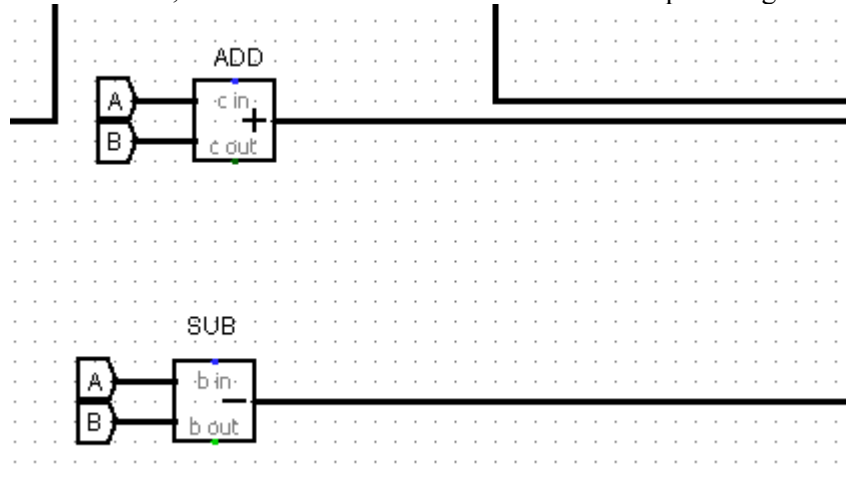


Figura: Implementação ADD e SUB

Operações SLL e SLR

Para implementar as operações SLL e SLR, foram utilizados Deslocadores (para esquerda ou direita) oferecidos pelo Logisim, utilizando como entrada o SHAMT, para informar quantos bits serão deslocados,

e o input B

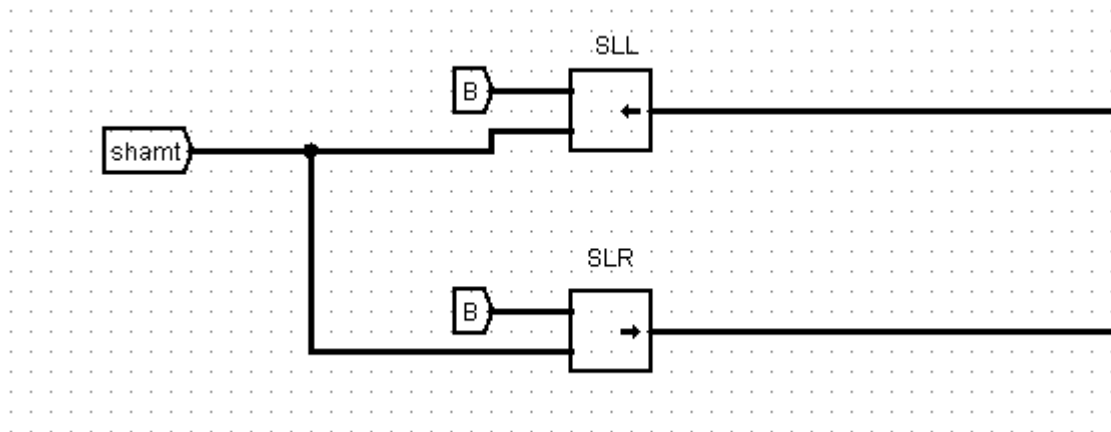


Figura: Implementação SLL e SLR

Saídas

Como dito anteriormente, a ULA conta com 3 saídas: Result, Zero e Overflow

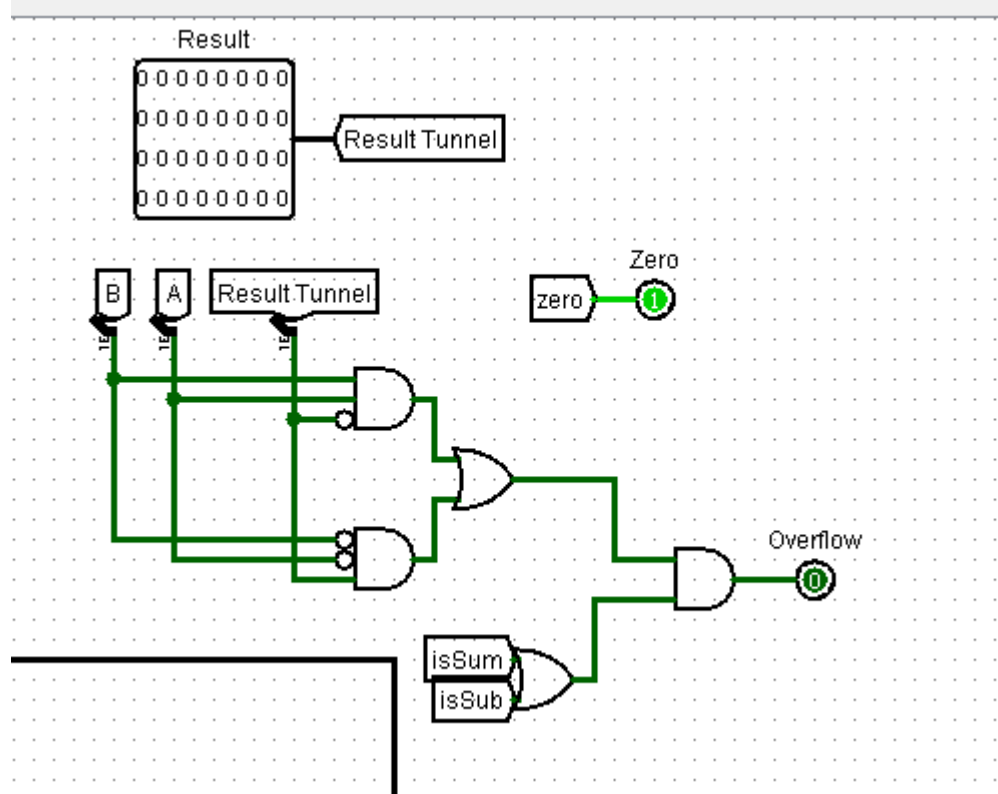


Figura: Saídas Result, Zero e Overflow

Overflow

O Overflow retorna 1 apenas se houve overflow, e se a operação escolhida é uma soma ou uma subtração, como apresentada nas figuras abaixo:

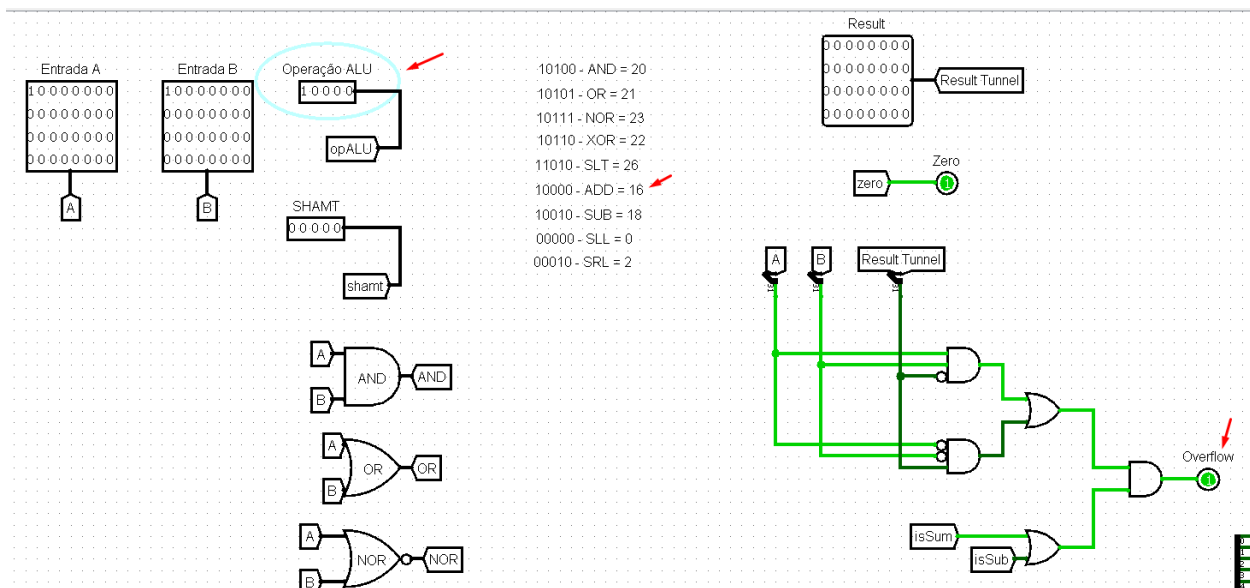


Figura: Overflow com soma

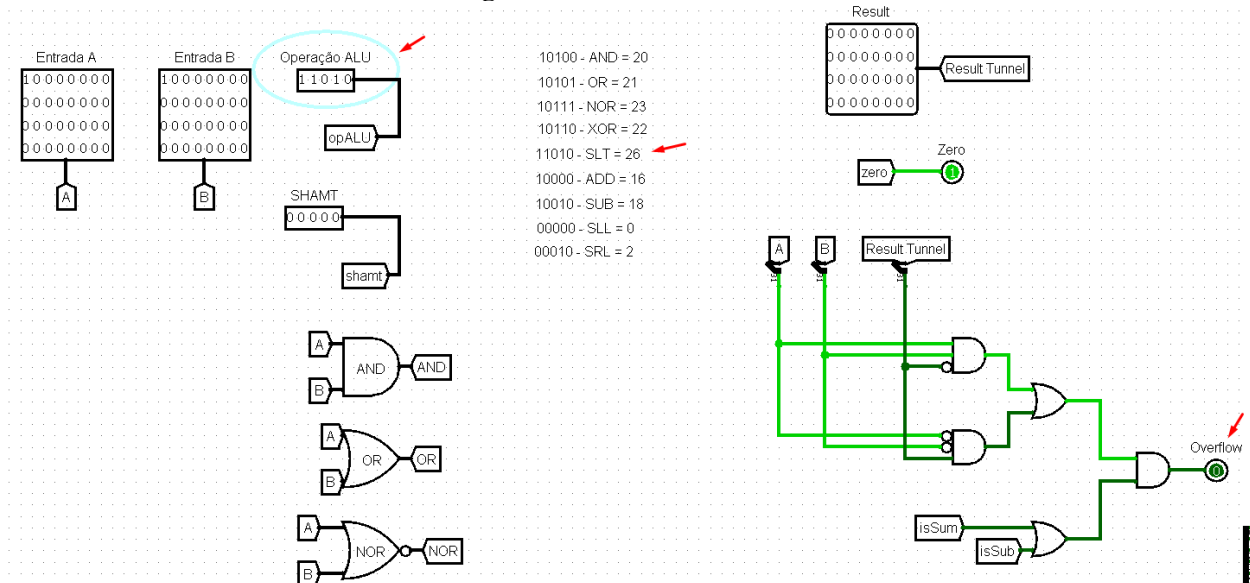


Figura: Overflow quando não é soma

Escolha da operação

A escolha da operação é feita através do Operação ALU, e controlada por um Multiplexador de acordo com os valores na tabela de operações.

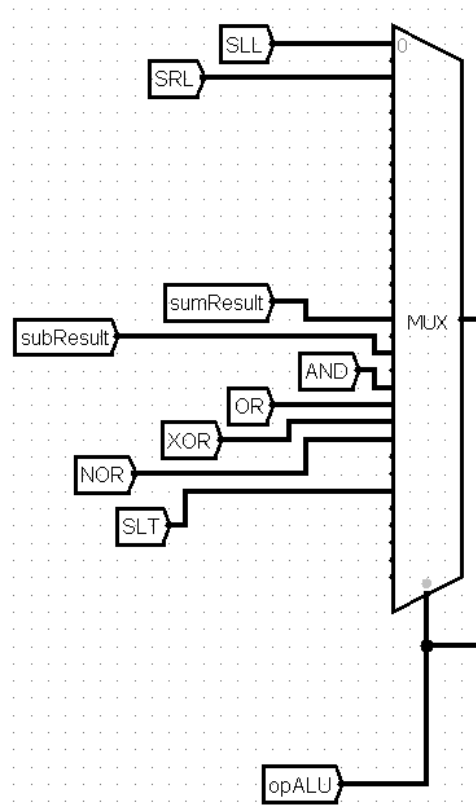


Figura: Multiplexador para controle de operações

- ## Banco de Registradores

O Banco de Registradores é um componente responsável por lidar com escritas e leituras nos registradores de uma arquitetura, em MIPS, existem 32 registradores presentes no banco:

- \$0: Sempre tem o valor zero;
- \$at: Usado pelo montador como um registrador auxiliar;
- \$v0 - \$v1: Valores de retornos dos procedimentos;
- \$a0 - \$a3: Argumentos que podem ser usados pelos procedimentos;
- \$t0 - \$t9: Registradores que possuem valores temporários;
- \$s1 - \$s7: Registradores com valores que devem ser preservados entre chamadas de funções;
- \$k0 - \$k1: Reservados pelo SO para tratamento de exceções;
- \$gp: *Global Pointer*, usado para guardar o endereço da região de dados da *heap*;
- \$sp: *Stack Pointer*, usado para guardar o endereço da região de dados da *stack*;
- \$fp: *Frame Pointer*, usado para acesso relativo dos dados dentro de procedimentos;
- \$ra: *Return Address*, guarda o endereço de retorno do procedimento;

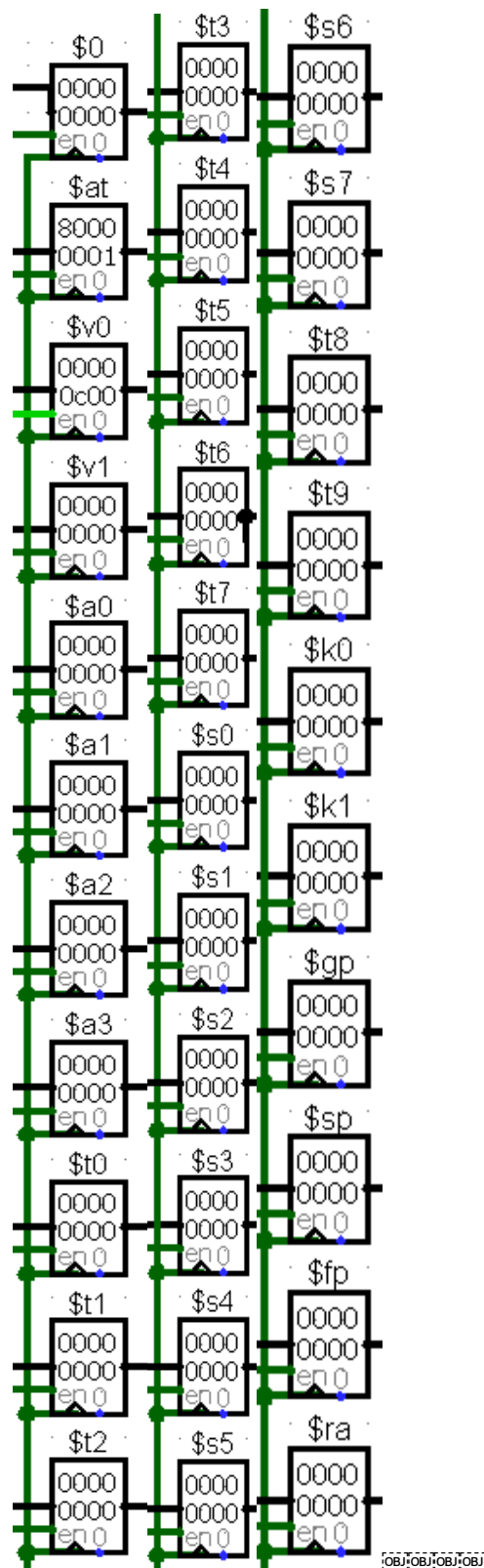


Figura: Registradores dentro do banco;

O Banco de Registradores deve possuir entradas para controle de escrita, dentre as quais, são as seguintes:

WD: Dado a ser escrito;

WR: Registrador a ser escrito o valor em WD;

RegWrite: Ativa ou desativa a escrita no registrador WR;

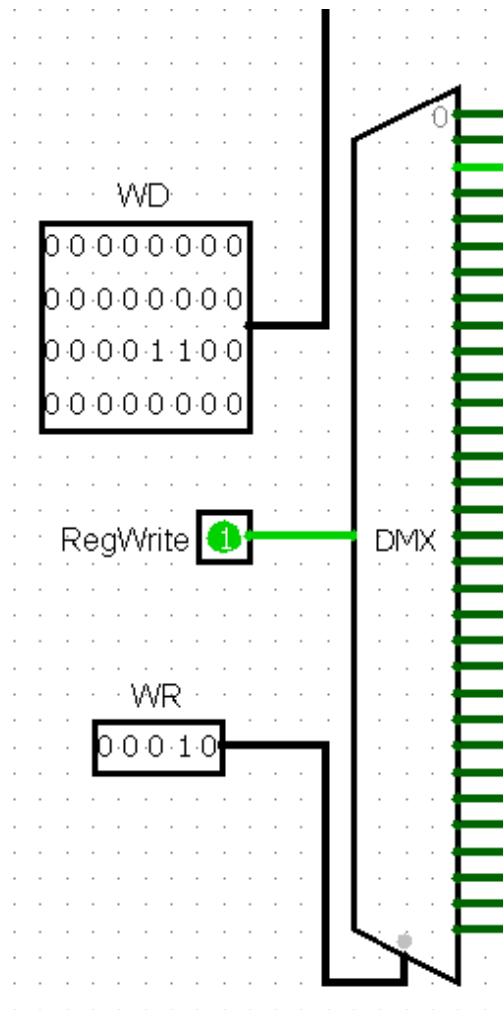


Figura: Entradas para controle de escrita;

Também existem entradas para controle de leitura dos registradores:

LR1: Registrador a ser lido;

LR2: Registrador a ser lido;

LD1: Dado lido por LR1;

LD2: Dado lido por LR2;

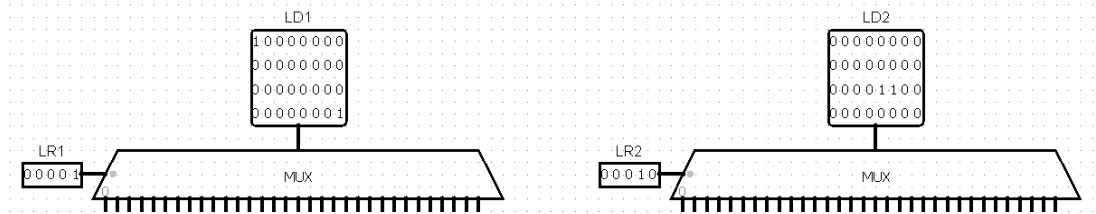


Figura: Entradas para controle de leitura;

A escrita nos registradores funciona da seguinte forma: todos os registradores do banco (exceto o registrador \$0) recebem o valor de WD como entrada, mas apenas o registrador que está selecionado por WR será escrito. Essa distinção de qual registrador deve ser escrito é feita usando um Demultiplexador.

Já a leitura é realizada de forma semelhante, todos os valores dos registradores vão para LD1 e LD2, mas apenas aquele cujo endereço está em LR1 e LR2 respectivamente, são enviados às saídas. Esse processo usa um Multiplexador para tomar essa decisão.

● Decodificador

Este circuito recebe apenas uma entrada: uma instrução de 32 bits. A partir dela, os bits de retorno serão os seguintes:

Para o tipo R, é retornado:

- OPCODE - 6 bits
- RS - 5 bits
- RT - 5 bits
- RD - 5 bits
- SHAMT - 5 bits
- FUNCT - 6 bits

As instruções do tipo J retornam:

- OPCODE - 6 bits
- ADDR - 26 bits

Por último, as instruções do tipo I retornam:

- OPCODE - 6 bits
- RS - 5 bits
- RT - 5 bits
- IMM - 16 bits

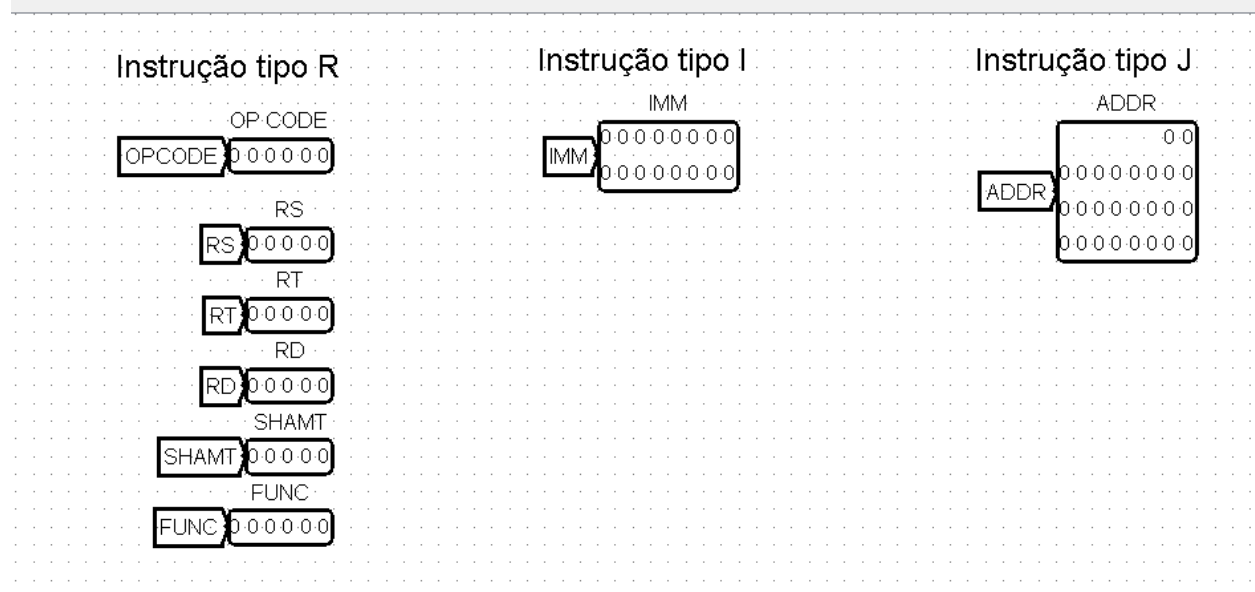


Figura: Saídas do Decodificador

Na implementação do circuito, utilizou-se demultiplexadores para retornar os valores baseados no tipo de instrução (R, J ou I), e a partir das verificações, as saídas são atribuídas como podem ser vistas nas imagens abaixo:

Figura: Demultiplexadores para identificar tipo de instrução

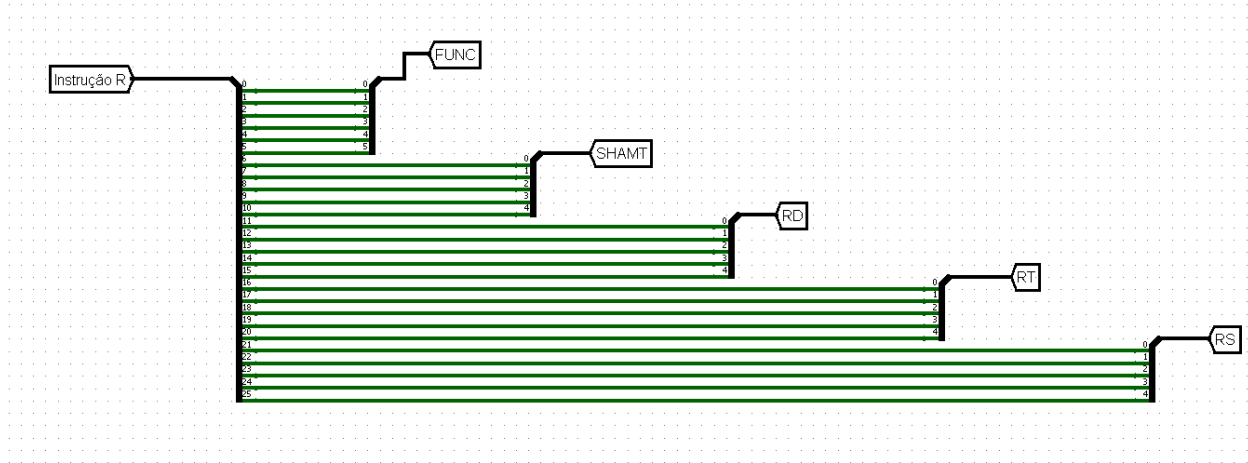


Figura: Saídas de instrução R

Para as instruções do tipo I e J, os campos IMM e ADDR são obtidos a partir do fio da entrada de instrução, da seguinte forma:

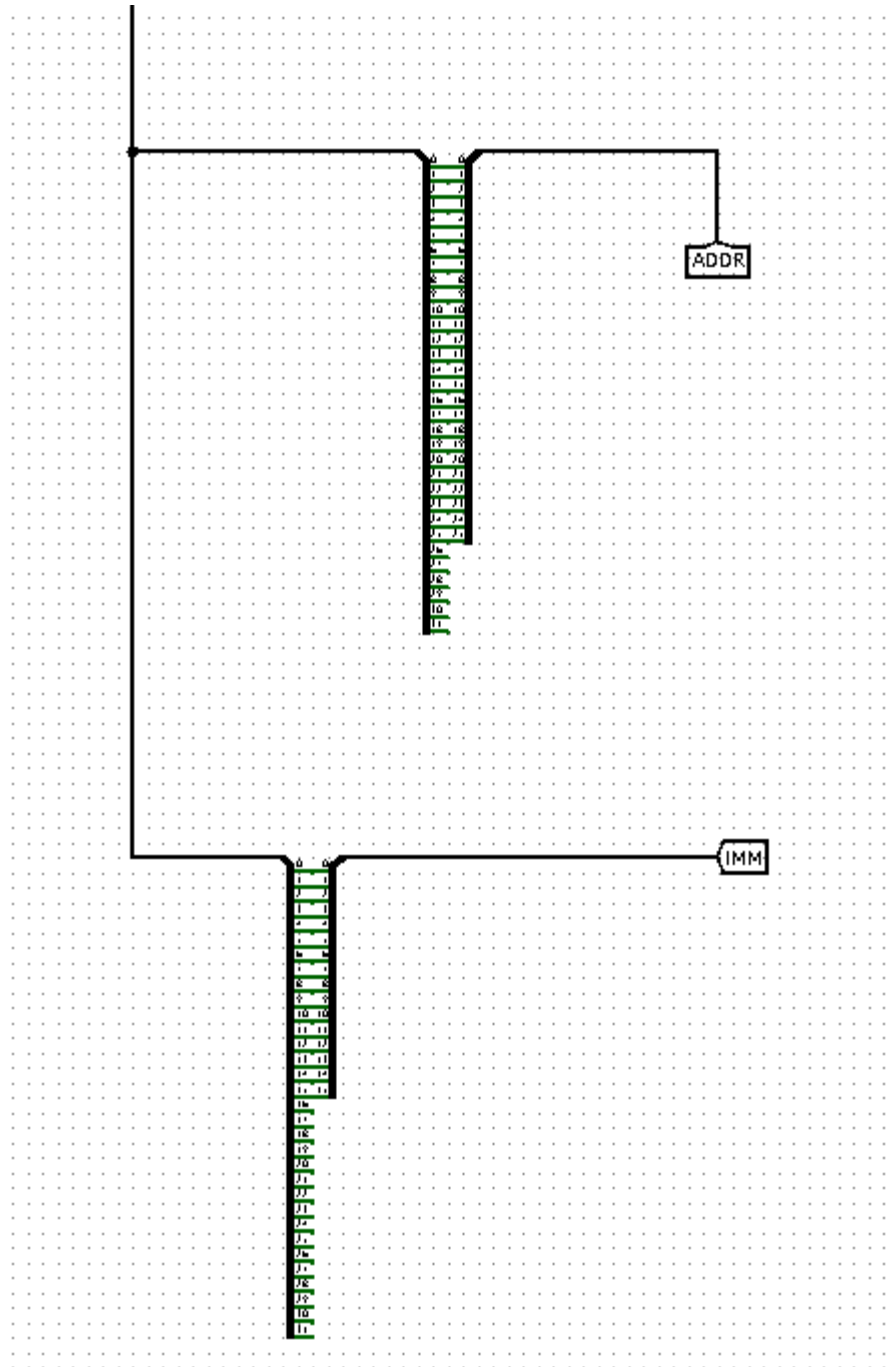


Figura: Circuito para obter IMM e ADDR

- **Extensor de sinal**

O Circuito Extensor de sinal recebe como entrada um valor de 16 bits, e complementa o sinal para 32 bits, estendendo o valor do bit mais significativo para manter o sinal do número de entrada.

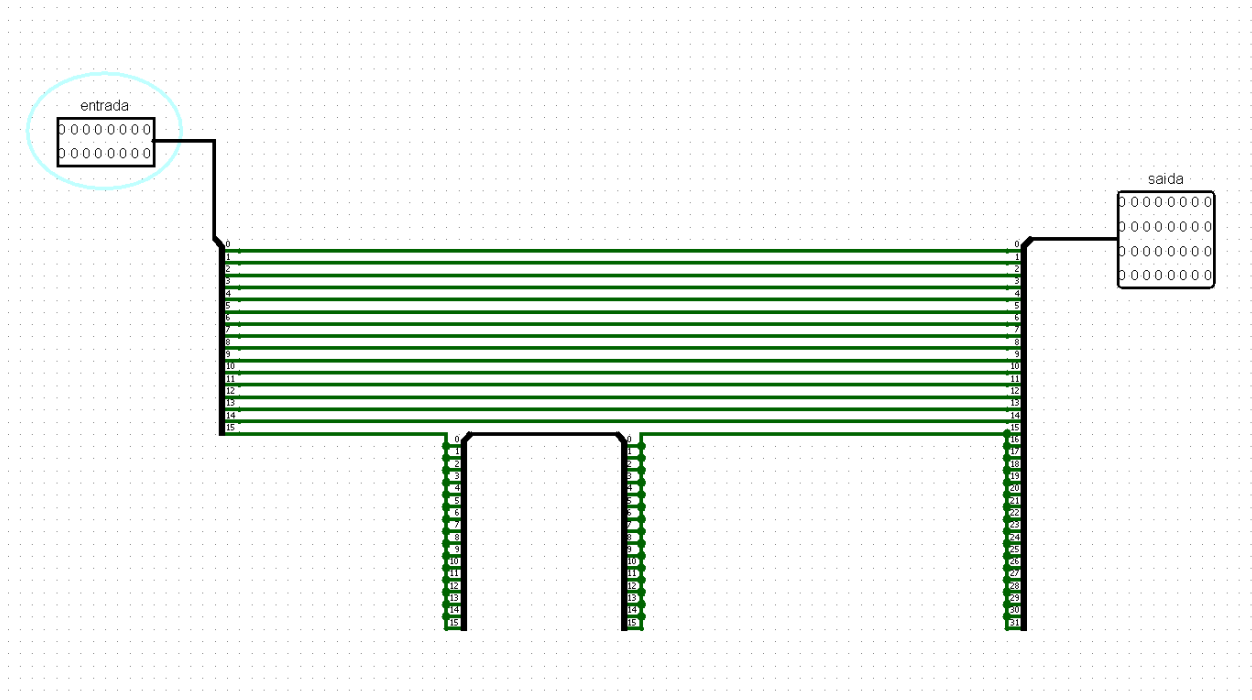


Figura: Circuito Extensor de sinal

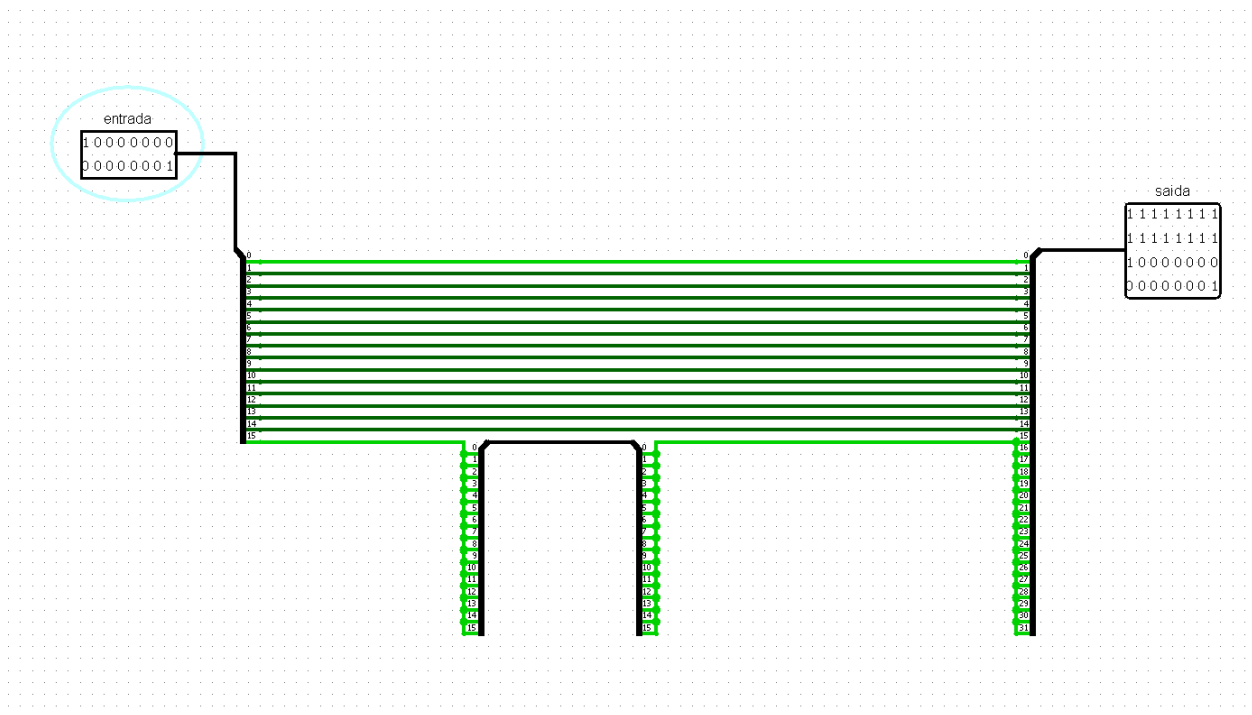


Figura: Extensor de sinal com sinal 1 estendido

- **Unidade de Controle**

A Unidade de Controle (UC) é responsável por determinar o caminho em que os dados devem tomar, ligando ou desligando *flags* que dizem de qual caminho o dado deve entrar, e qual caminho ele deve sair.

- **AluOp:** Determina se a ULA deve realizar a operação presente no campo funct da instrução (para instruções do tipo R), ou se deve realizar soma (para as instruções ADDI, LW e SW), ou se deve realizar subtração (para as instruções BEQ e BNE);
- **RegDst:** Escolhe entre o registrador RD e RT para escrita no banco de registradores;
- **WriteReg:** Ativa ou desativa a escrita no banco de registradores;
- **AluSrc:** Ativado quando a segunda entrada da ULA for o campo imediato;
- **MemToReg:** Escreve o valor da memória em um registrador;
- **MemRead:** Ativa a leitura de dados na memória através do endereço calculado;
- **MemWrite:** Ativa a escrita de dados na memória;
- **Branch:** Ativado quando a instrução for um BEQ ou BNE;
- **Jump:** Ativado quando a instrução for um J ou JAL;
- **Link:** Determina que o registrador \$ra recebe PC + 4;



AluOp é 00 quando a instrução é do tipo R. Nesse caso, $BEQ + BNE = 0$, ocasionando o bit 1 de AluOp ser igual a 0. $!(BEQ + BNE) \text{ AND } !R = 0$, pois R é 1, então o bit 0 de AluOp é igual a 0 também; AluOp deve ser 01 quando a instrução for ADDI, LW, ou SW. Os fios R, BEQ e BNE são 0 nesse caso, portanto, o bit 0 de AluOp será igual a 1;

AluOp deve ser 10 quando a instrução for BEQ ou BNE. O bit 1 de AluOp é igual a BEQ + BNE. Já o bit 0 será igual a 0 pois !(BEQ + BNE) é igual a 0;
 AluOp não é relevante em nenhum outro caso, podendo ser qualquer valor, pois a ULA não será utilizada;

- **Unidade de Controle da ULA**

A Unidade de Controle da ULA (UCA) é um componente responsável por definir qual operação será realizada na ULA. As entradas desse componente são: **funct**, **OpCode** e **ALUOp**, já as saídas são: **Op** e **JumpRegister**;

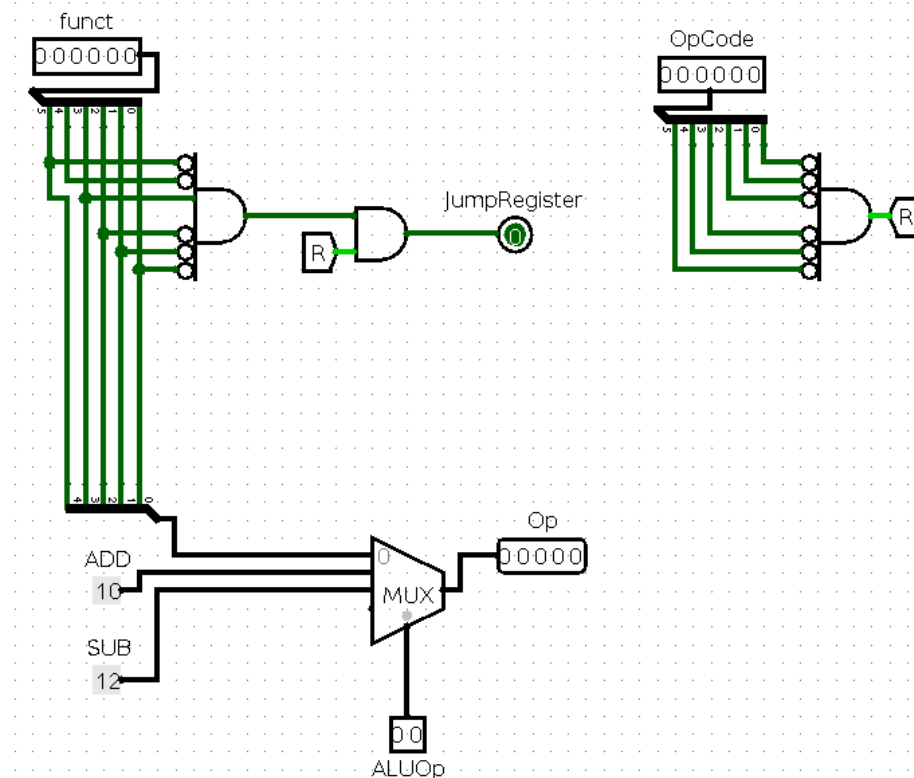


Figura: Circuito da Unidade de Controle da ULA

A saída de Op é determinada por um MUX que seleciona entre os 3 valores disponíveis: funct, ADD (0x10), e SUB (0x12). ALUOp é um valor determinado pela Unidade de Controle (UC).

A saída JumpRegister recebe 1 caso funct seja igual a 8 (código funct da instrução JR) e OpCode for igual a 0. Essa saída deve ser feita nesta UCA, pois ela é uma instrução do tipo R, o que torna impossível detectá-la na UC.

Como discutido anteriormente no tópico referente à ULA, a operação que vai de funct para Op ignora o bit 4 de funct, pois ele não é necessário para diferenciar as operações.

- **Endereçamento pseudodireto**

No MIPS, a instrução jump utiliza do endereçamento pseudodireto em relação ao PC. ou seja, a instrução recebe o endereço de desvio / 4 no campo addr. E para alterar o valor de PC corretamente, é necessário retornar o endereço em addr para o real, fazendo dois *shifts* à esquerda, depois, addr é concatenado com os 6 bits mais significativos de PC.

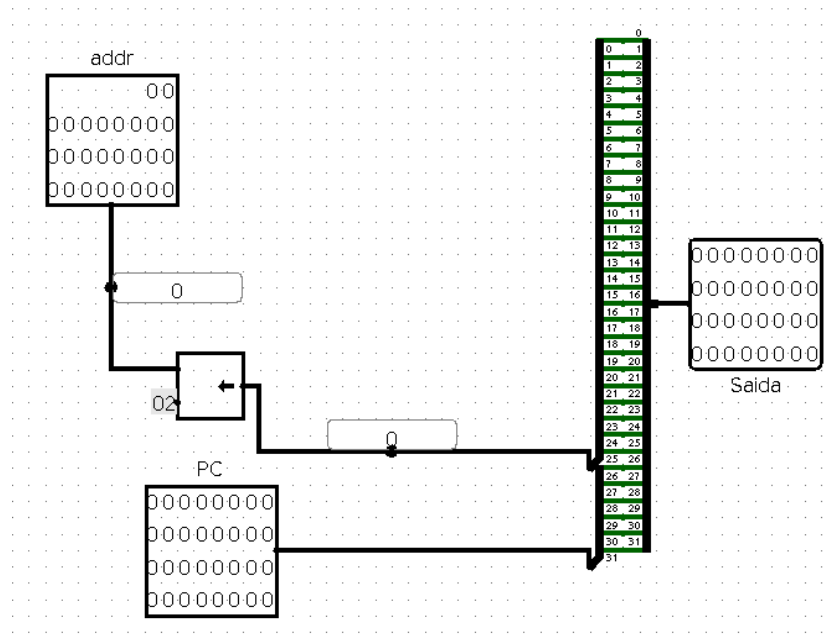


Figura: circuito que realiza o endereçamento pseudodireto em relação ao PC

- **Caminho de dados:**

O caminho de dados se inicia com o registrador PC, que tem uma conexão para buscar a próxima instrução a ser executada na memória de instruções:

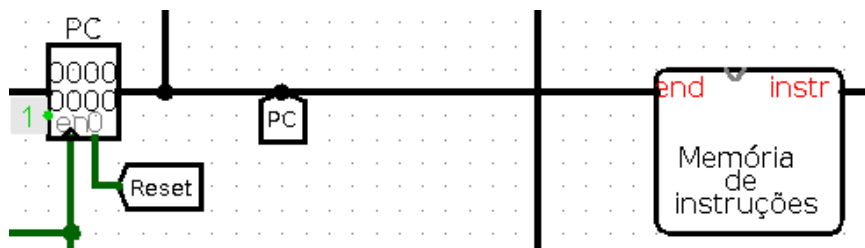


Figura: Caminho que leva PC à Memória de instruções

Depois, a instrução deve ser decodificada e suas partes devem ser separadas pelo decodificador:

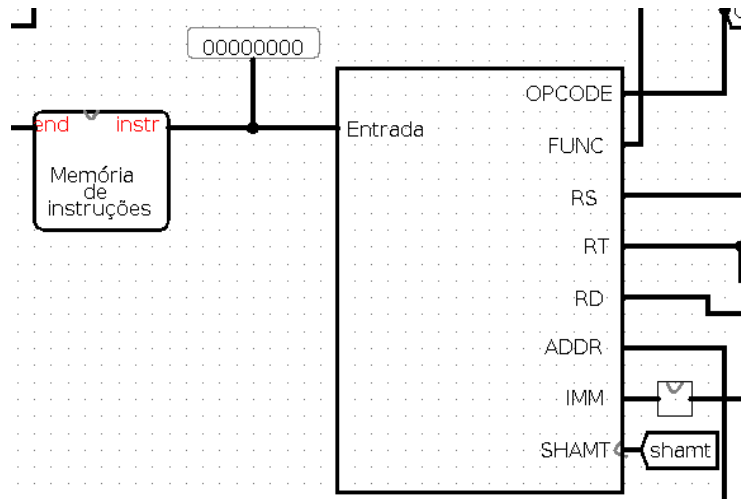


Figura: Caminho que leva a instrução até o decodificador

A Unidade de Controle e Unidade de Controle da ULA recebem o OpCode e Func, e setam todas as flags do caminho:

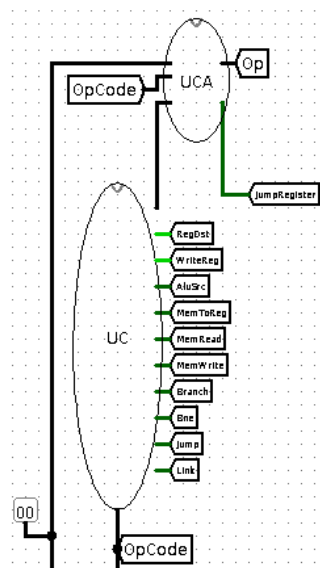


Figura: UC e UCA sentando todas as flags do caminho de dados

O decodificador encaminha os campos RS, RT, RD para o banco de registradores, ADDR para o cálculo de endereço, IMM para a segunda entrada da ULA, e SHAMT como entrada da ULA.

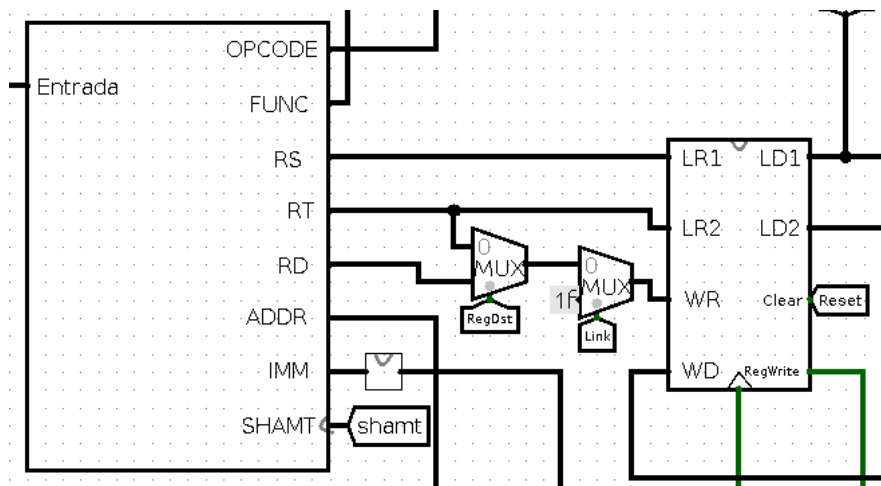


Figura: Decodificador encaminhando as suas saídas

Na ULA, a flag AluSrc decide se o segundo campo de entrada será o LD2, ou o campo IMM da instrução, a saída da ULA é direcionada para a entrada da memória e também para o banco de registradores.

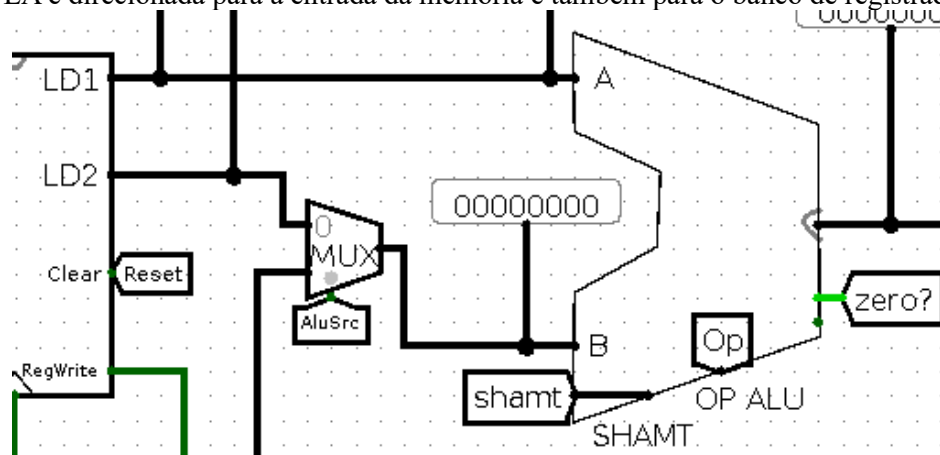


Figura: Entradas da ULA

O resultado da ULA pode ser direcionado à memória, caso a instrução tenha acesso à ela, ou ao banco de registradores. A saída da memória também retorna ao banco de registradores para escrita, MemWrite, MemRead, MemToReg e Link controlam esse fluxo

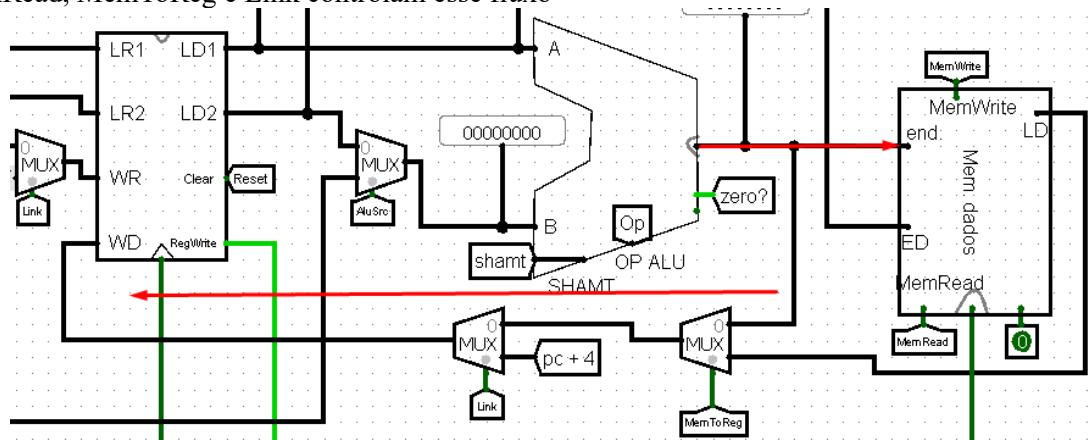


Figura: Caminho de WriteBack dos dados

Se a instrução for um Branch ou Jump, o PC receberá valores diferentes de $PC + 4$, e esses valores devem ser calculados. As flags Bne e Branch têm o propósito de fazer o cálculo de endereço para os branches, somando $PC + 4 \ll 2$ com IMM.

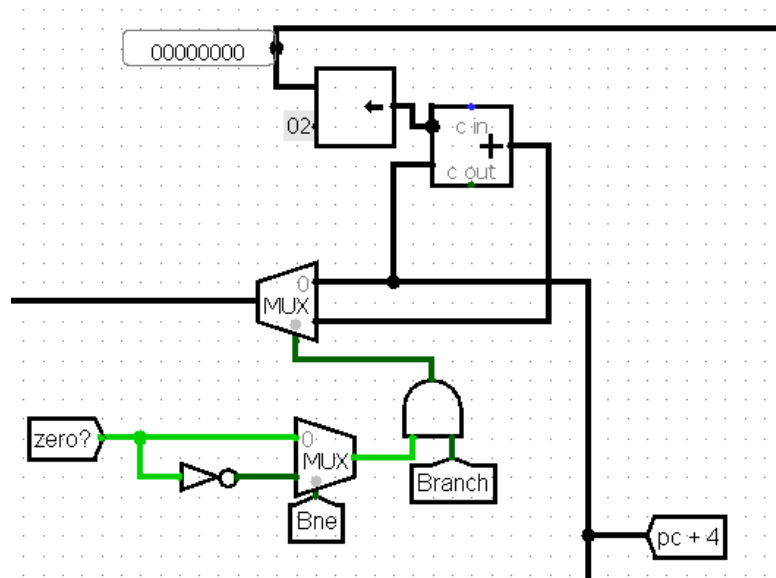


Figura: Circuito que faz o cálculo de endereço para instruções do tipo Branch

Antes do novo endereço calculado entrar em PC novamente, ele passa por 2 filtros, o de JumpRegister, que define PC = LD1, e o Jump que define PC como o cálculo de endereçamento pseudodireto.

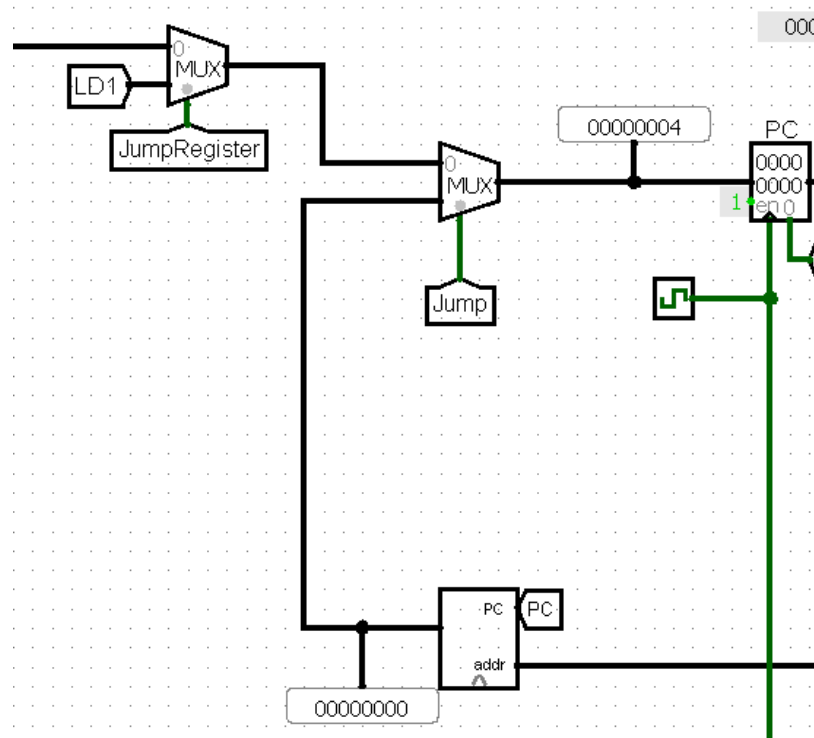


Figura: Filtro para instruções Jump e JumpRegister em PC

Código para execução

O Código escolhido para a execução no simulador foi uma implementação do BubbleSort, conhecido algoritmo de ordenação.

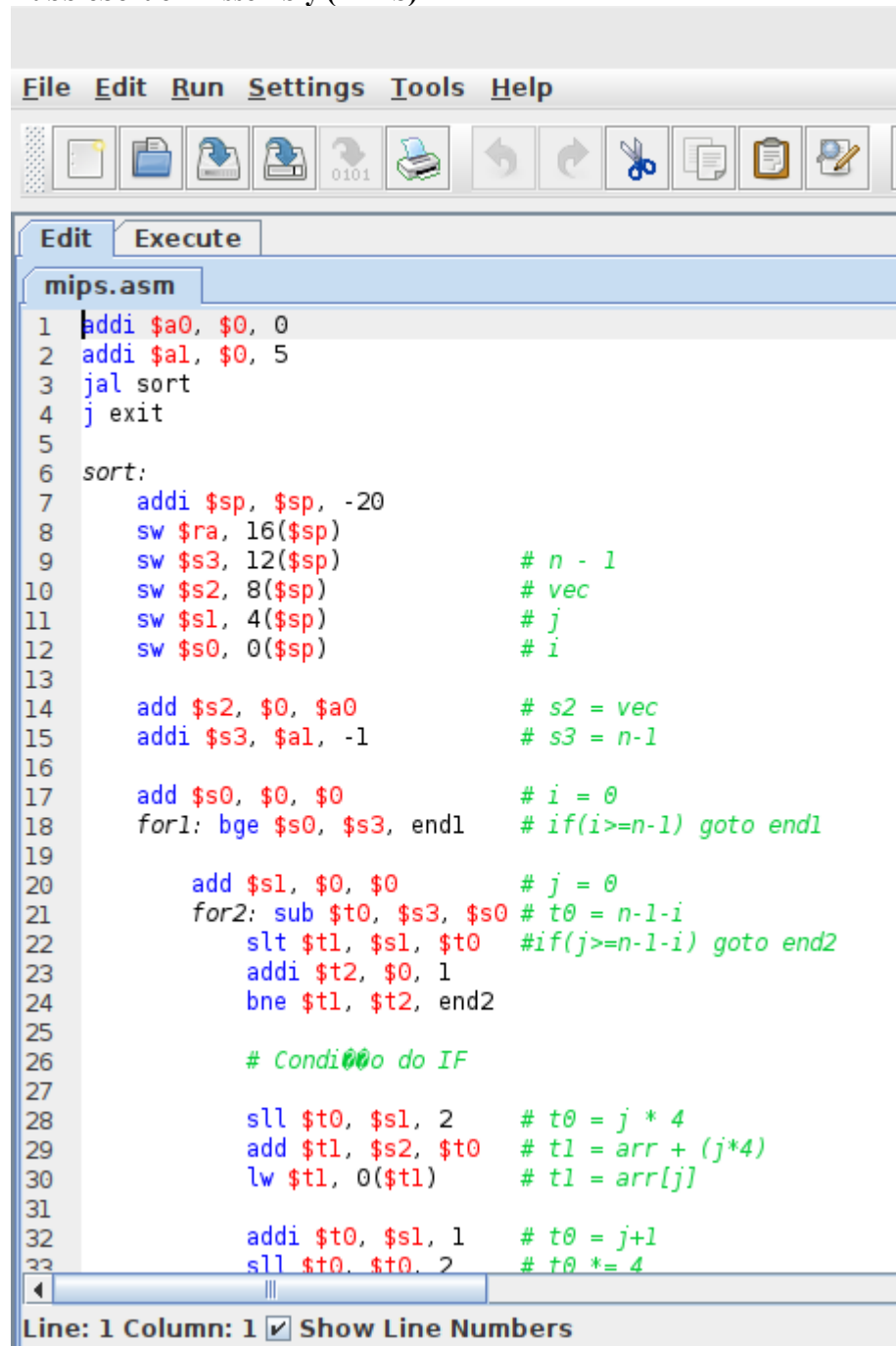
BubbleSort em C

```
void swap(int vec[], int k){
    int temp = vec[k];
    vec[k] = vec[k+1];
    vec[k+1] = temp;
}

void sort(int vec[], int n){
    for(int i = 0; i < n-1; i++)
        for(int j = 0; j < n-i-1; j++)
            if(vec[j] > vec[j+1])
                swap(vec, j);
}

int vec[5] = {3, 2, 1, 4, 5};
sort(vec, 5);
```

BubbleSort em Assembly (MIPS)

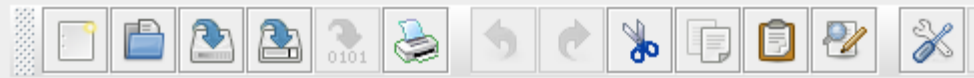


The screenshot shows a MIPS assembly editor window with a menu bar (File, Edit, Run, Settings, Tools, Help) and a toolbar. The editor is in 'Edit' mode and shows a file named 'mips.asm'. The code is a MIPS assembly implementation of BubbleSort. It starts with 'addi \$a0, \$0, 0' and 'addi \$a1, \$0, 5', then jumps to a 'sort' label. The 'sort' label begins with 'addi \$sp, \$sp, -20' to allocate stack space, followed by storing registers \$ra, \$s3, \$s2, \$s1, and \$s0. It then sets up pointers for 'vec' and 'n-1'. The main loop 'for1' checks if 'i' is greater than or equal to 'n-1'. If not, it enters an inner loop 'for2' where 'j' is set to 'n-1-i'. The inner loop compares and swaps elements at 'arr[j]' and 'arr[j+1]'. Comments in green provide explanations for the operations.

```
1 addi $a0, $0, 0
2 addi $a1, $0, 5
3 jal sort
4 j exit
5
6 sort:
7     addi $sp, $sp, -20
8     sw $ra, 16($sp)
9     sw $s3, 12($sp)           # n - 1
10    sw $s2, 8($sp)           # vec
11    sw $s1, 4($sp)           # j
12    sw $s0, 0($sp)           # i
13
14    add $s2, $0, $a0          # s2 = vec
15    addi $s3, $a1, -1         # s3 = n-1
16
17    add $s0, $0, $0           # i = 0
18    for1: bge $s0, $s3, end1   # if(i>=n-1) goto end1
19
20    add $s1, $0, $0           # j = 0
21    for2: sub $t0, $s3, $s0    # t0 = n-1-i
22    slt $t1, $s1, $t0         # if(j>=n-1-i) goto end2
23    addi $t2, $0, 1
24    bne $t1, $t2, end2
25
26    # Condição do IF
27
28    sll $t0, $s1, 2           # t0 = j * 4
29    add $t1, $s2, $t0         # t1 = arr + (j*4)
30    lw $t1, 0($t1)           # t1 = arr[j]
31
32    addi $t0, $s1, 1          # t0 = j+1
33    sll $t0, $t0, 2           # t0 *= 4
```

Line: 1 Column: 1 ☒ Show Line Numbers

File Edit Run Settings Tools Help



Edit Execute

mips.asm

```
25
26      # Condição do IF
27
28      sll $t0, $s1, 2      # t0 = j * 4
29      add $t1, $s2, $t0    # t1 = arr + (j*4)
30      lw $t1, 0($t1)      # t1 = arr[j]
31
32      addi $t0, $s1, 1     # t0 = j+1
33      sll $t0, $t0, 2      # t0 *= 4
34      add $t2, $s2, $t0    # t2 = arr + (j+1)*4
35      lw $t2, 0($t2)      # t2 = arr[j+1]
36
37      if: ble $t1, $t2, endif
38          add $a1, $s1, $0 # a1 = j
39          jal swap
40      endif:
41
42      # Fim do IF
43
44      addi $s1, $s1, 1     # j++
45      j for2
46  end2:
47
48      addi, $s0, $s0, 1     # i++
49      j for1
50  end1:
51      lw $s0, 0($sp)
52      lw $s1, 4($sp)
53      lw $s2, 8($sp)
54      lw $s3, 12($sp)
55      lw $ra, 16($sp)
56      addi $sp, $sp, 20
57      ir $ra
```

```

38         add $a1, $s1, $0 # a1 = j
39         jal swap
40     endif:
41
42     # Fim do IF
43
44     addi $s1, $s1, 1    # j++
45     j for2
46 end2:
47
48     addi, $s0, $s0, 1    # i++
49     j for1
50 end1:
51     lw $s0, 0($sp)
52     lw $s1, 4($sp)
53     lw $s2, 8($sp)
54     lw $s3, 12($sp)
55     lw $ra, 16($sp)
56     addi $sp, $sp, 20
57     jr $ra
58
59 swap:
60     sll $t1, $a1, 2
61     add $t1, $a0, $t1
62     lw $t0, 0($t1)
63     lw $t2, 4($t1)
64     sw $t2, 0($t1)
65     sw $t0, 4($t1)
66     jr $ra
67
68 exit:
69

```

BubbleSort em Hexadecimal

A instrução está no formato gerado pela memória de instruções:

v2.0 raw

```

23bd1fff 08100000 1048574*0 20040000 20050005 0c100004 08100031 23bdfec
afb0010 afb3000c afb20008 afb10004 afb00000 00049020 20b3ffff 00008020
0213082a 10200014 00008820 02704022 0228482a 200a0001 152a000d 00114080
02484820 8d290000 22280001 00084080 02485020 8d4a0000 0149082a 10200002
02202820 0c10002a 22310001 08100010 22100001 0810000d 8fb00000 8fb10004
8fb20008 8fb3000c 8fbf0010 23bd0014 03e00008 00054880 00894820 8d280000
8d2a0004 ad2a0000 ad280004 03e00008

```


Para executá-lo, deve-se adicionar valores nas 5 primeiras posições da memória de dados:

[illegible]

Após adicionados os números na Memória de Dados, deve-se carregar a instrução na Memória de Instruções:

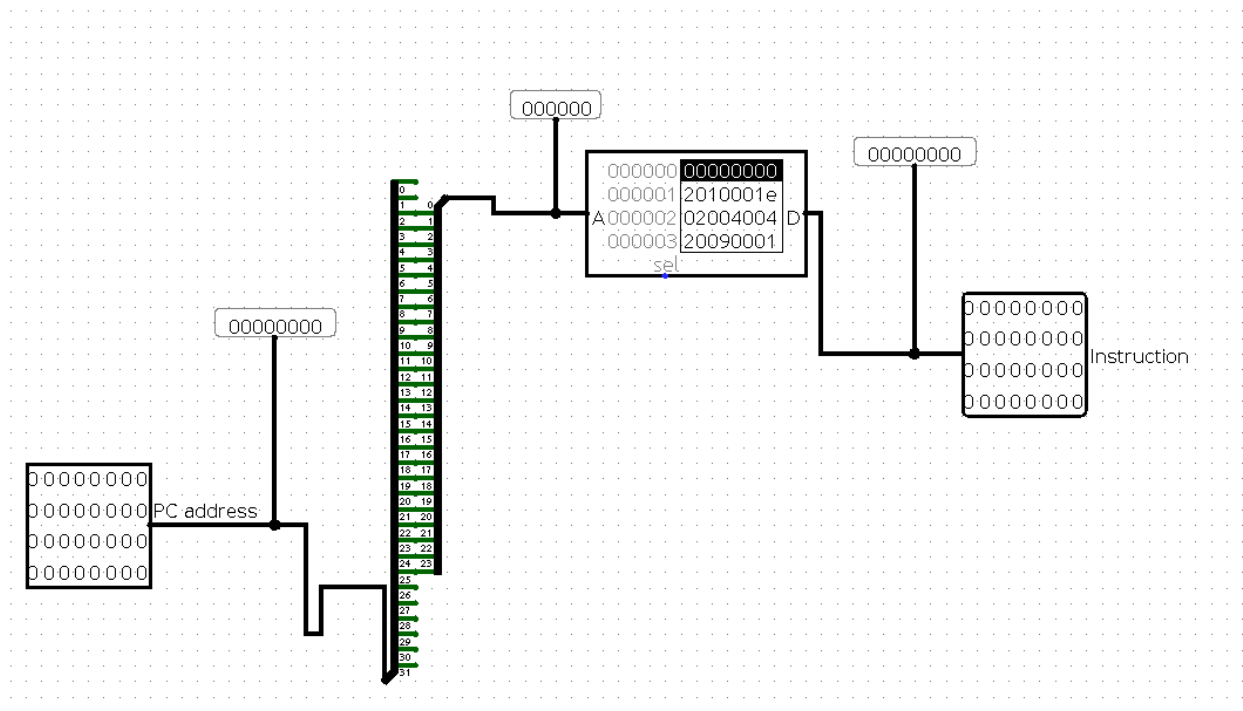


Figura: Memória de Instruções

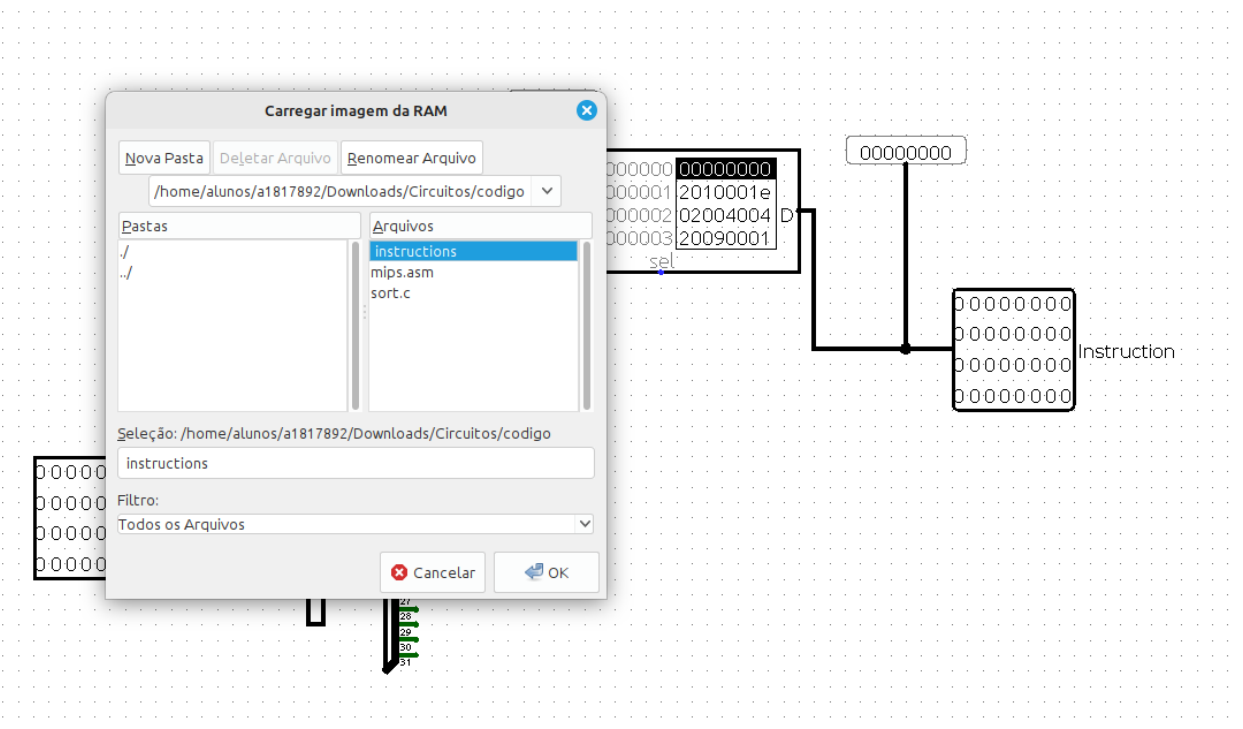


Figura: Carregar instruções na memória de instruções

Notas:

- A primeira instrução (0x23bd1fff) serve para inicializar \$sp com um valor alto, nesse caso, 0x1fff.

- A segunda instrução (0x08100000) é responsável por fazer um Jump para a posição da instrução inicial do algoritmo do BubbleSort (0x00400000). Foi escolhido adicionar esse Jump no início para sincronizar as posições na memória de instruções do Mars e do Datapath implementado nesse trabalho. Assim, com a sincronização dos dois, todos os Branches que funcionam no Mars, também funcionam no Datapath, sem a necessidade de alteração na posição de memória de destino.
- A segunda instrução (1048574*0), é uma versão simplificada para adicionar zeros nas instruções entre a primeira e a instrução onde começa o BubbleSort (0x00400000).
- A partir da quarta instrução (0x20040000), começa a implementação do BubbleSort e suas subfunções.
- As duas primeiras instruções do código servem para carregar os argumentos para o sort:
addi \$a0, \$0, 0 # vec[] começa na posição 0x0 da memória de dados;
addi \$a1, \$0, 5 # o vetor possui 5 elementos;

Carregadas as instruções e os dados, basta rodar a simulação, e ao final, os valores na memória de dados devem estar organizados em ordem crescente.

Pontuação por instrução

Tipo		Pontos	Máximo	Máximo
Instruções	Instruções R (exceto JR)	4	20	70
	Instruções aritméticas com imediato	5	15	
	Desvios condicionais	7	14	
	Desvios incondicionais	8	24	
	Operações de memória	8	16	
Código	Estrutura condicional	3	6	30
	Laço de repetição	5	10	
	Laços de repetição aninhados	8	8	
	Chamada de função folha	10	10	
	Chamada de função não-folha	12	12	

Figura: Tabela de Pontuação

Baseado na tabela de pontuação acima, o código desenvolvido para esse trabalho obtém os seguintes pontos:

Instruções - 76 pontos

- Instruções R (Exceto JR) - 12 pontos
 - add \$t1, \$a0, \$t1 - 4 pontos
 - sub \$t0, \$s3, \$s0 - 4 pontos
 - slt \$t1, \$s1, \$t0 - 4 pontos
- Instruções aritméticas com imediato - 10 pontos
 - sll \$t0, \$s1, 2 - 5 pontos
 - addi \$t0, \$s1, 1 - 5 pontos
- Operações de memória - 16 pontos
 - Load Word - 8 pontos
 - lw \$s0, 0(\$sp)
 - Store Word - 8 pontos
 - sw \$ra, 16(\$sp)
- Desvios Incondicionais - 24 pontos
 - Jump And Link - 8 pontos
 - jal sort
 - Jump Register - 8 pontos
 - jr \$ra
 - Jump - 8 pontos
 - j for1
- Desvios Condicionais - 14 pontos
 - Branch not Equals - 7 pontos
 - bne \$t1, \$t2, end2
 - Branch Equals - 7 pontos
 - bge \$s0, \$s3, end1, que é convertido para um sll e um beq

Código - 43 pontos

- Estrutura condicional - 3 pontos
 - if(vec[j] > vec[j + 1]){ - 3 pontos
- Laço de repetição - 10 pontos
 - for(int i = 0; i < n-1; i++){ - 5 pontos
 - for(int j = 0; j < n-i-1; j++){ - 5 pontos
- Laço de repetição aninhado - 8 pontos
 - for(int i = 0; i < n-1; i++){
for(int j = 0; j < n-i-1; j++){
- Chamada de função folha - 10 pontos
 - swap(int vec[], int k) - 10 pontos
- Chamada de função não folha - 12 pontos
 - sort(vec, 5); - 12 pontos

Referências

[1] Patterson, David A. Hennessy, John L. Organização e Projeto de Computadores. Disponível em: Minha Biblioteca, (5a. edição). Grupo GEN, 2017.