Pedro Spoljaric Gomes

Implementação do Compilador C-

São José dos Campos - Brasil

outubro de 2020

Pedro Spoljaric Gomes

Implementação do Compilador C-

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Compiladores.

Docente: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

outubro de 2020

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

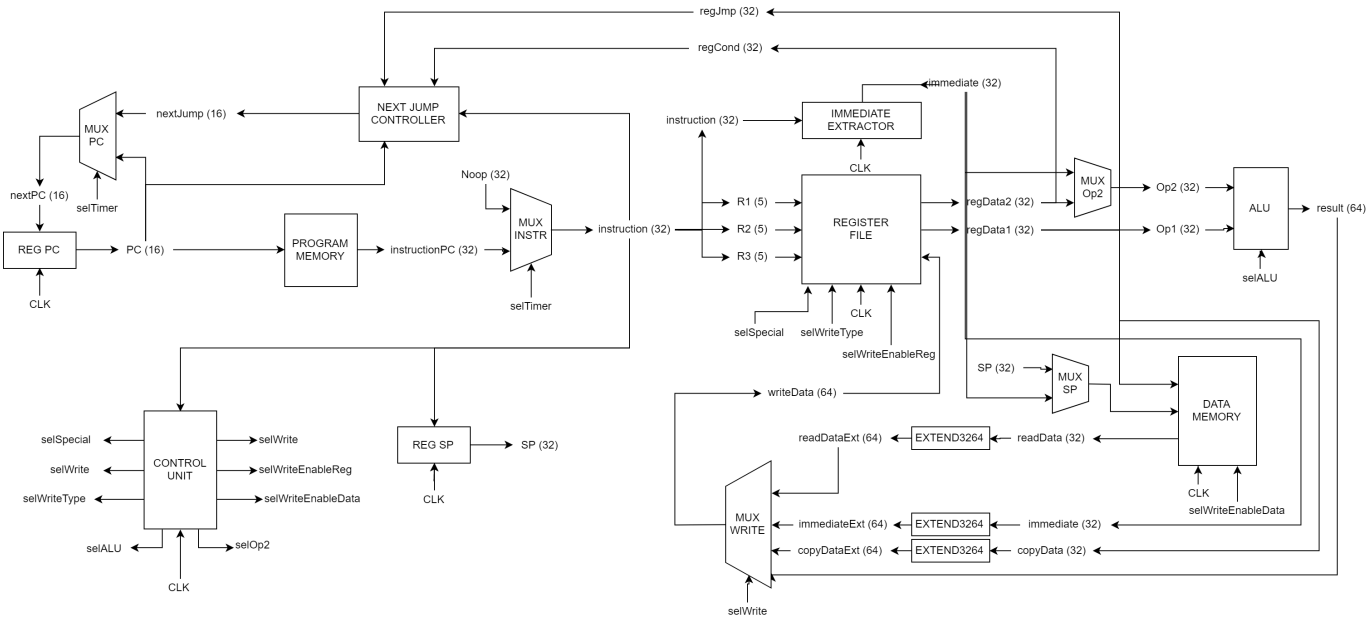
Em computação, um compilador é um programa que transforma um texto escrito em uma linguagem fonte específica em um texto escrito em uma linguagem alvo, também específica. Para uma melhor compreensão, pode-se dizer que esse processo é análogo à tradução de linguas verbais. Processadores não entendem linguagens além daquela que foram projetados para interpretar, que possui instruções no nível mais baixo possível de abstração, utilizando *bits*. A função de um compilador é ser capaz de ler um código numa linguagem que possui nível de abstração mais alto e transformá-lo no código equivalente da linguagem que o processador consegue interpretar, e é desse conceito que vem a importância dos compiladores; sem eles, tudo precisaria ser programado diretamente em código binário, o que tornaria inviável a evolução de sistemas computacionais. Os próximos capítulos tratarão de todo o processo de desenvolvimento de um compilador que transforma códigos escritos em linguagem C- em códigos binários feitos de instruções que podem ser interpretadas pelo processador desenvolvido na disciplina Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

2 Processador

2.1 Componentes do processador

Os principais componentes do processador são a unidade lógica aritmética, o banco de registradores, as memórias de dados e de programa, e diversos blocos intermediários que auxiliam na interligação entre eles e na manipulação das instruções. Para auxiliar na interpretação e na implementação da arquitetura do sistema, foi desenvolvido um esquemático, ilustrado na Figura 1 detalhado voltado para o entendimento da movimentação dos dados em cada etapa do processo. Em seguida, serão analisados com mais detalhes cada bloco individualmente, juntamente com os respectivos códigos implementados seguindo o mesmo padrão de nomes do esquemático detalhado.

Figura 1 – Esquemático detalhado.



Fonte: O Autor

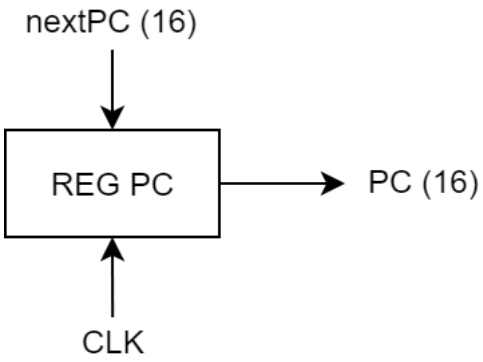
2.1.1 REG PC

O fio PC (Program Counter) de 16 bits serve como um indicador de qual instrução armazenada na memória de programa (PROGRAM MEMORY) deve ser executada no momento. O fio nextPC, também de 16 bits, indica qual será o valor de PC depois da próxima atualização.

O papel do bloco REG PC, ampliado na Figura 2 é simplesmente atualizar o valor do fio PC a cada descida de *clock* com o valor vigente de nextPC, que será especificado nas

próximas seções. A implementação deste bloco pode ser visualizada no código REG PC.

Figura 2 – Bloco REG PC



Fonte: O Autor

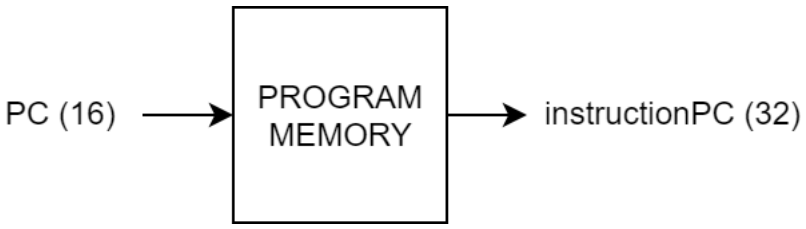
```
1  module regPC
2  #(parameter TAM_PC = 16)
3  (
4          input wire  [(TAM_PC-1):0] nextPC,
5          output wire [(TAM_PC-1):0] PC,
6          input wire  clk
7  );
8
9  reg [(TAM_PC-1):0] aux;
10 assign PC = aux;
11
12 always @ (negedge clk) aux = nextPC;
13
14 endmodule
```

2.1.2 PROGRAM MEMORY

É neste bloco, ampliado na Figura 3, que todas as instruções do programa que a ser executado ficam armazenadas. Além de armazenar as instruções, é função deste bloco atualizar o valor do fio instructionPC com o valor da instrução referente ao PC sempre que este for atualizado, ou seja, não depende diretamente do *clock*.

A implementação deste bloco, código PROGRAM MEMORY, mostra como a atualização de valor de instructionPC é realizada com base na atualização do fio PC. Para simulação, foi utilizada uma memória que comporta até 10 instruções, que são carregadas do arquivo "program.txt".

Figura 3 – Bloco PROGRAM MEMORY



Fonte: O Autor

```
1  module programMemory
2  #(parameter TAM_PC = 16)
3  (
4          input wire  [(TAM_PC-1):0] PC,
5          output wire [31:0] instructionPC
6  );
7
8  //reg [31:0] instructions[(2**TAM_PC)-1:0];
9  reg [31:0] instructions[9:0];
10
11 reg [31:0] regInstr;
12 assign instructionPC = regInstr;
13
14 initial begin
15         $readmemb("program.txt", instructions);
16 end
17
18 always @ (PC) regInstr = instructions[PC];
19
20 endmodule
```

2.1.3 REGISTER FILE

Este bloco, Figura 4, armazena informações nos 32 registradores de propósito geral e em alguns registradores especiais que guardam, por exemplo, o resultado de uma divisão ou multiplicação. Sendo assim, ele possui duas funções principais durante o ciclo de *clock*, uma durante a subida e outra durante a descida.

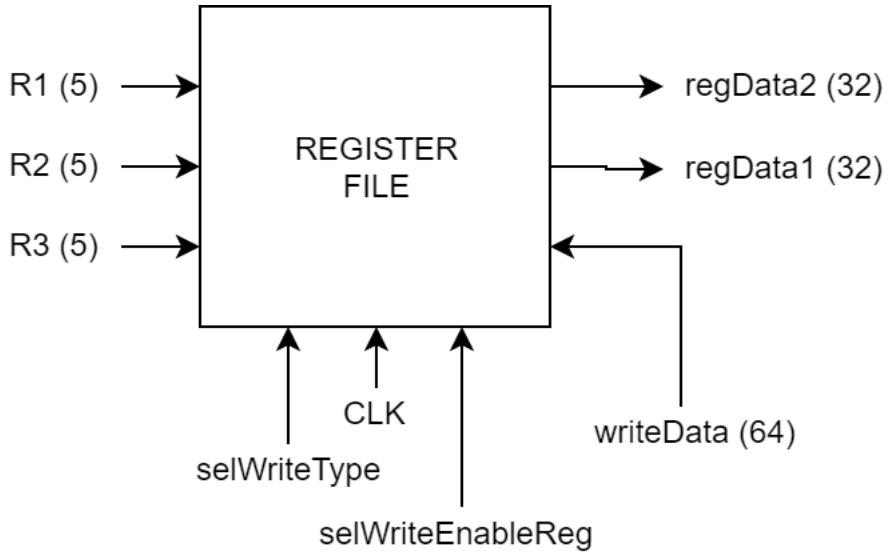
Na subida de clock, os valores de saída regData1 e regData2 são atualizados com base nos valores de entrada, respectivamente, R1 e R2, que indicam os índices dos registradores devem ter suas informações armazenadas expostas.

Na descida de *clock* é realizado, caso habilitado, o armazenamento do valor dos 32 bits menos significativos do fio writeData no registrador com índice especificado por R3. writeData possui 64 bits para que seja possível tratar exceções como os resultados de uma divisão ou multiplicação, que precisam ser armazenados em dois registradores de 32 bits cada.

A implementação do bloco representada pelo código REGISTER FILE mostra a separação das duas funcionalidades dele, a leitura de informações na subida de *clock* e a

escrita de informações, além do tratamento dos casos especiais de escrita usando o seletor selWriteType, na descida de *clock*.

Figura 4 – Bloco REGISTER FILE



Fonte: O Autor

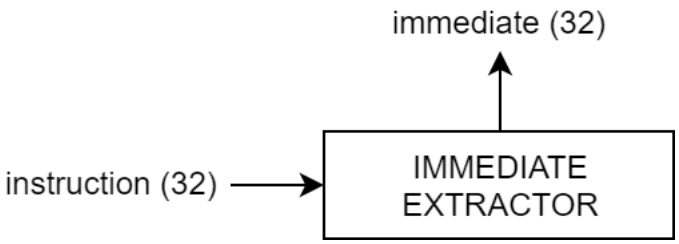
```
1  module regFile
2  (
3          input wire  [4:0]  R1,
4          input wire  [4:0]  R2,
5          input wire  [4:0]  R3,
6          output wire [31:0] regData1,
7          output wire [31:0] regData2,
8          input wire  [63:0] writeData,
9          input wire  [1:0]  selWriteType,
10         input wire                      selWriteEnableReg, clk
11 );
12
13 parameter REGULAR = 2'b00;
14 parameter MULT = 2'b01;
15 parameter DIV = 2'b10;
16
17 parameter MS = 0;
18 parameter LS = 1;
19 parameter QUO = 2;
20 parameter REM = 3;
21
22 reg [31:0] registers[31:0];
23 reg [31:0] special[3:0];
24
25 reg [31:0] aux1, aux2;
26 assign regData1 = aux1;
27 assign regData2 = aux2;
28
29 always @ (posedge clk) begin
30         aux1 <= registers[R1];
31         aux2 <= registers[R2];
32 end
```

```
33
34 always @ (negedge clk) begin
35         if (selWriteEnableReg) begin
36                 case (selWriteType)
37                         REGULAR: registers[R3] <= writeData[31:0];
38                         MULT: begin
39                                 special[MS] = writeData[63:32];
40                                 special[LS] = writeData[31:0];
41                         end
42                         DIV: begin
43                                 special[QUO] = writeData[63:32];
44                                 special[REM] = writeData[31:0];
45                         end
46                         default: registers[R3] <= writeData[31:0];
47                 endcase
48         end
49 end
50
51 endmodule
```

2.1.4 IMMEDIATE EXTRACTOR

Na Figura 5 foi ampliado o bloco IMMEDIATE EXTRACTOR, seu papel é extrair das instruções pertinentes vindas do fio instruction valores que não correspondem à endereços de registradores, ou seja, operandos imediatos que serão utilizados em operações aritméticas, corresponder à endereços da memória de dados, entre outros. Além disso, o bloco identifica o tamanho do imediato com base na instrução recebida e extende seu comprimento para um padrão de 32 *bits*. O código implementado, IMMEDIATE EXTRACTOR, mostra como é feita a identificação do tipo de instrução pelo *OPCODE* e como é tratada a extração do imediato em cada caso.

Figura 5 – Bloco IMMEDIATE EXTRACTOR



Fonte: O Autor

```
1 module immediateExtractor
2 (
3         input wire  [31:0] instruction,
4         output wire [31:0] immediate,
5         input wire clk
6 );
7
8 parameter SUM                   = 4'b0000;
9 parameter SUBTRACT      = 4'b0001;
```

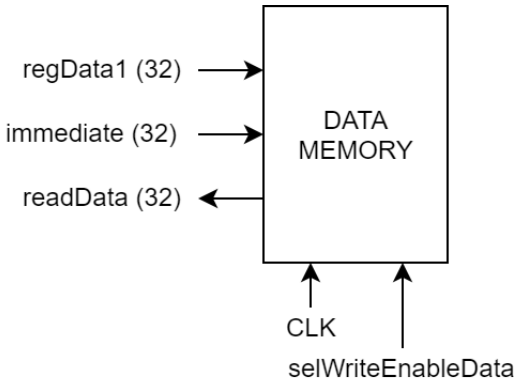
```
10 parameter MULTIPLY      = 4'b0010;
11 parameter DIVIDE                = 4'b0011;
12 parameter SHIFT         = 4'b0100;
13 parameter LOGIC         = 4'b0101;
14 parameter JUMP                  = 4'b0110;
15 parameter STACK         = 4'b0111;
16 parameter WRITE         = 4'b1000;
17 parameter COPY                  = 4'b1001;
18 parameter LOAD                  = 4'b1010;
19 parameter STORE         = 4'b1011;
20 parameter SLEEP         = 4'b1100;
21
22 reg [3:0] opcode;
23
24 reg  [31:0] auxImmediate;
25 assign immediate = auxImmediate;
26
27 always @ (posedge clk) begin
28         opcode = instruction[31:28];
29
30         case (opcode)
31                 SUM, SUBTRACT:          auxImmediate = instruction[0:0] == 1'b0 ? 32'b0 :
                        {15'b0, instruction[17:1]};
32                 MULTIPLY, DIVIDE: auxImmediate = instruction[0:0] == 1'b0 ? 32'b0 : {10'
                    b0, instruction[22:1]};
33                 SHIFT:                          auxImmediate = {10'b0, instruction
                    [17:1]};
34                 LOGIC:                          auxImmediate = {10'b0, instruction
                    [17:1]};
35                 JUMP:                                   auxImmediate = instruction[1:0]
                    == 2'b00 || instruction[1:0] == 2'b01 ? {6'b0, instruction[27:2]} :
                    32'b0;
36                 WRITE:                          auxImmediate = {9'b0, instruction[22:0]};
37                 LOAD, STORE:            auxImmediate = {10'b0, instruction[22:1]};
38                 SLEEP:                          auxImmediate = {4'b0, instruction[27:0]};
39                 default:                                auxImmediate = 32'b0;
40         endcase
41 end
42
43 endmodule
```

2.1.5 DATA MEMORY

A Figura 6 corresponde à ampliação do bloco DATA MEMORY, que armazena uma quantidade maior de informações que o bloco REGISTER FILE e que não passarão por movimentações com tanta frequência Seu funcionamento é parecido com o do bloco REGISTER FILE, passando por processos de leitura ou escrita dependendo do momento do ciclo de *clock*, subida ou descida. Diferente do bloco REGISTER FILE, este bloco não possui casos especiais de escrita, armazenando sempre dados de 32 *bits* e dependendo apenas da habilitação do seletor selWriteEnableData. Além disso, a definição do endereço de leitura ou escrita é feita pelo imediato de 32 *bits*, podendo abranger uma quantidade muito superior de endereços diferentes. O código DATA MEMORY se refere à implementação desse bloco e mostra a utilização de uma memória de capacidade muito superior ao

REGISTER FILE, além de maior simplicidade no processo de escrita.

Figura 6 – Bloco DATA MEMORY



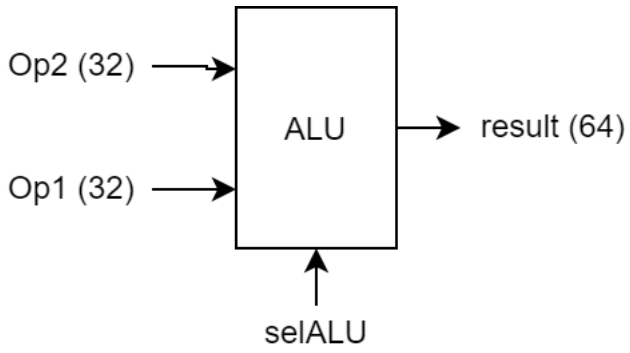
Fonte: O Autor

```
1  module dataMemory
2  (
3          input wire  [31:0] immediate,
4          input wire  [31:0] regData1,
5          output wire [31:0] readData,
6          input wire                      selWriteEnableData, clk
7  );
8
9  reg [31:0] data[(2**19):0]; // ~1MB
10
11 reg [31:0] aux1, aux2;
12 assign readData = aux2;
13
14 always @ (posedge clk) begin
15         aux1 <= data[immediate];
16 end
17
18 always @ (negedge clk) begin
19         if (selWriteEnableData)
20                 data[immediate] <= regData1;
21 end
22
23 endmodule
```

2.1.6 ALU

O bloco ALU foi ampliado na Figura 7, onde é possível observar que ele possui duas entradas de 32 *bits*, Op1 e Op2, além do seletor selALU, e uma saída de 64 *bits*, result. Este bloco realiza a operação aritmética selecionada por selALU entre os operandos vindos de Op1 e Op2, gerando um resultado de até 64 bits (variando de tamanho dependendo do tipo de operação) em result. O código ALU mostra todos os tipos de operação selecionáveis e o que cada um representa.

Figura 7 – Bloco ALU



Fonte: O Autor

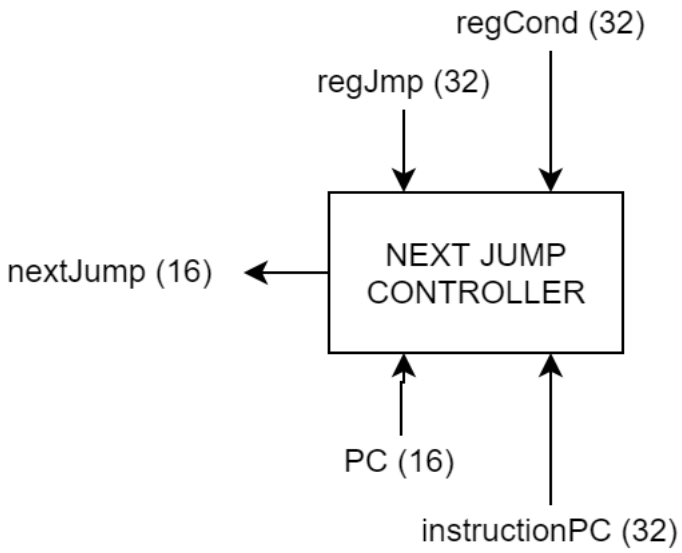
```
module alu (
        input  [31:0] op1,
        input  [31:0] op2,
        output [63:0] result,
        input  [3:0]  selALU
);

parameter SUM                = 4'b0000;
parameter SUBTRACT           = 4'b0001;
parameter MULTIPLY           = 4'b0010;
parameter DIVIDE             = 4'b0011;
parameter SHIFT_LEFT         = 4'b0100;
parameter SHIFT_RIGHT        = 4'b0101;
parameter BITWISE_NOT        = 4'b0110;
parameter BITWISE_AND        = 4'b0111;
parameter BITWISE_OR         = 4'b1000;
parameter BITWISE_XOR        = 4'b1001;
parameter LOGIC_EQUAL        = 4'b1010;
parameter LOGIC_DIFFERENT    = 4'b1011;
parameter LOGIC_LESS_THAN    = 4'b1100;
parameter LOGIC_GREATER_THAN = 4'b1101;

reg [31:0] aux1, aux2;

assign result = {aux2, aux1};

always @(op1 or op2) begin
        aux2 = 32'b0;
        case (selALU)
                SUM:                aux1 = op1 + op2;
                SUBTRACT:           aux1 = op1 - op2;
                MULTIPLY:           {aux2, aux1} = op1 * op2;
                DIVIDE: begin
                                    aux2 = op1 / op2;
                                    aux1 = op1 % op2;
                end
                SHIFT_LEFT:         aux1 = op1 << op2;
                SHIFT_RIGHT:        aux1 = op1 >> op2;
                BITWISE_NOT:        aux1 = ~op1;
                BITWISE_AND:        aux1 = op1 & op2;
                BITWISE_OR:         aux1 = op1 | op2;
                BITWISE_XOR:        aux1 = op1 ^ op2;
                LOGIC_EQUAL:        aux1 = {32{op1 == op2}};
```

```
44                  LOGIC_DIFFERENT:    aux1 = {32{op1 != op2}};
45                  LOGIC_LESS_THAN:    aux1 = {32{op1 < op2}};
46                  LOGIC_GREATER_THAN: aux1 = {32{op1 > op2}};
47          endcase
48  end
49
50  endmodule
```

2.1.7 NEXT JUMP CONTROLLER

A função deste bloco é definir o endereço da próxima instrução a ser executada; essa definição é feita com base no instructionPC, para interpretar o tipo de instrução, no PC atual, no caso de a próxima instrução ser relativa ao PC, regJmp, quando a próxima instrução é o valor absoluto contido em um registrador, e regCond, quando este desvio depende da condição contida em um registrador. O código NEXT JUMP CONTROLLER mostra como cada um desses tipos de instrução são interpretados e o que acontece em cada um deles.

Figura 8 – Bloco NEXT JUMP CONTROLLER



Fonte: O Autor

```
1   module nextJumpController
2   #(parameter TAM_PC = 16)
3   (
4           input wire  [31:0] instruction,
5           input wire  [(TAM_PC-1):0] PC,
6           input wire  [31:0] regJmp, regCond,
7           output wire [(TAM_PC-1):0] nextJump
8   );
9
10  reg [(TAM_PC-1):0] auxJump;
11  assign nextJump = auxJump;
12
```

```
13 reg [3:0] opcode;
14 reg [1:0] T;
15
16 always @ (regJmp or regCond or PC) begin
17
18         opcode = instruction[31:28];
19         T = instruction[1:0];
20
21         if (opcode == 4'b0110) begin
22                 case (T)
23                         2'b00: begin
24                                 auxJump = PC + instruction[(TAM_PC-1+2):2];
25                         end
26                         2'b01: begin
27                                 auxJump = instruction[(TAM_PC-1+2):2];
28                         end
29                         2'b10: begin // Jump para endereco contido no registrador
30                                 auxJump = regJmp[(TAM_PC-1):0];
31                         end
32                         2'b11: begin // Jump para endereco contido no registrador (
                               condicional)
33                                 auxJump = regCond == {32{1'b1}} ? regJmp[(TAM_PC-1):0] :
                                       PC + 1;
34                         end
35                         default: begin
36                                 auxJump = PC + 1;
37                         end
38                 endcase
39         end else begin
40                 auxJump = PC + 1;
41         end
42
43 end
44 endmodule
```

2.1.8 CONTROL UNIT

A função deste bloco é definir os valores de todos seletores a cada subida de *clock* de acordo com a Tabela 1. Os seletores definem coisas como as operações que serão realizadas e os tipos de dados que serão movimentados, por exemplo, o seletor selAlu define qual operação o bloco ALU deve realizar entre os operandos de entrada. O código CONTROL UNIT mostra como a definição dos seletores é feita.

Tabela 1 – Valores dos seletores por tipo de instrução.

Instrução	Subtipo	selTimer	selWrite EnableReg	selWrite EnableData	selOp2	selAlu	selWrite Type	selWrite	selSpecial	selSP
SUM	1	0	1	0	0	0000	00	10	00	0
	2	0	1	0	1	0000	00	10	00	0
SUBTRACT	1	0	1	0	0	0001	00	10	00	0
	2	0	1	0	1	0001	00	10	00	0
MULTIPLY	1	0	1	0	0	0010	01	10	00	0
	2	0	1	0	1	0010	01	10	00	0
DIVIDE	1	0	1	0	0	0011	01	10	00	0
	2	0	1	0	1	0011	10	10	00	0
SHIFT	1	0	1	0	1	0101	00	10	00	0
	2	0	1	0	1	0100	00	10	00	0
LOGIC	1	0	1	0	0	0110	00	10	00	0
	2	0	1	0	0	0111	00	10	00	0
	3	0	1	0	0	1000	00	10	00	0
	4	0	1	0	0	1001	00	10	00	0
	5	0	1	0	0	1010	00	10	00	0
	6	0	1	0	0	1011	00	10	00	0
	7	0	1	0	0	1100	00	10	00	0
	8	0	1	0	0	1101	00	10	00	0
JUMP	-	0	0	0	d	dddd	dd	dd	dd	0
STACK	1	0	0	1	d	dddd	00	00	00	1
	2	0	1	0	d	dddd	00	00	00	1
WRITE	-	0	1	0	d	dddd	dd	dd	dd	0
COPY	-	0	1	0	d	dddd	00	01	00	0
LOAD	-	0	1	0	d	dddd	00	00	00	0
STORE	-	0	0	1	d	dddd	dd	dd	dd	0
SLEEP	-	1	0	0	d	dddd	dd	dd	dd	0
INPUT	-	0	1	0	d	dddd	00	11	00	0

Fonte: O Autor

```
1  module controlUnit
2  (
3          input wire  [31:0] instruction,
4          output wire selTimer, selWriteEnableReg, selWriteEnableData, selOp2,
5          output wire [3:0] selALU,
6          output wire [1:0] selWriteType, selSpecial, selWrite,
7          output wire selSP,
8          input wire clk
9  );
10
11 parameter SUM                    = 4'b0000;
12 parameter SUBTRACT       = 4'b0001;
13 parameter MULTIPLY       = 4'b0010;
14 parameter DIVIDE                 = 4'b0011;
15 parameter SHIFT          = 4'b0100;
16 parameter LOGIC          = 4'b0101;
17 parameter JUMP                   = 4'b0110;
18 parameter STACK          = 4'b0111;
19 parameter WRITE          = 4'b1000;
20 parameter COPY                   = 4'b1001;
21 parameter LOAD                   = 4'b1010;
22 parameter STORE          = 4'b1011;
23 parameter SLEEP          = 4'b1100;
24 parameter INPUT          = 4'b1101;
25
26 reg regTimer, regWriteEnableReg, regWriteEnableData, regOp2, regSP;
27 reg [1:0] regWriteType, regWrite, regSpecial;
```

```
28 reg [3:0] regALU;
29
30 assign selTimer = regTimer;
31 assign selWriteEnableReg = regWriteEnableReg;
32 assign selWriteEnableData = regWriteEnableData;
33 assign selOp2 = regOp2;
34 assign selALU = regALU;
35 assign selWriteType = regWriteType;
36 assign selWrite = regWrite;
37 assign selSpecial = regSpecial;
38 assign selSP = regSP;
39
40 wire [3:0] opcode;
41
42 assign opcode = instruction[31:28];
43
44 always @ (posedge clk) begin
45
46         case (opcode)
47         SUM: begin
48             regTimer = 1'b0;
49             regWriteEnableReg = 1'b1;
50             regWriteEnableData = 1'b0;
51             regOp2 = instruction[0];
52             regALU = 4'b0000;
53             regWriteType = 2'b00;
54             regWrite = 2'b10;
55             regSpecial = 2'b00;
56                                 regSP = 1'b0;
57         end
58         SUBTRACT: begin
59             regTimer = 1'b0;
60             regWriteEnableReg = 1'b1;
61             regWriteEnableData = 1'b0;
62             regOp2 = instruction[0];
63             regALU = 4'b0001;
64             regWriteType = 2'b00;
65             regWrite = 2'b10;
66             regSpecial = 2'b00;
67                                 regSP = 1'b0;
68         end
69         MULTIPLY: begin
70             regTimer = 1'b0;
71             regWriteEnableReg = 1'b1;
72             regWriteEnableData = 1'b0;
73             regOp2 = instruction[0];
74             regALU = 4'b0010;
75             regWriteType = 2'b01;
76             regWrite = 2'b10;
77             regSpecial = 2'b00;
78                                 regSP = 1'b0;
79         end
80         DIVIDE: begin
81             regTimer = 1'b0;
82             regWriteEnableReg = 1'b1;
83             regWriteEnableData = 1'b0;
84             regOp2 = instruction[0];
85             regALU = 4'b0011;
86             regWriteType = 2'b10;
87             regWrite = 2'b10;
```

```
88              regSpecial = 2'b00;
89                                  regSP = 1'b0;
90          end
91          SHIFT: begin
92              regTimer = 1'b0;
93              regWriteEnableReg = 1'b1;
94              regWriteEnableData = 1'b0;
95              regOp2 = 1'b0;
96              regALU = instruction[0] == 1'b1 ? 4'b0100 : 4'b0101;
97              regWriteType = 2'b00;
98              regWrite = 2'b10;
99              regSpecial = 2'b00;
100                                 regSP = 1'b0;
101         end
102         LOGIC: begin
103             regTimer = 1'b0;
104             regWriteEnableReg = 1'b1;
105             regWriteEnableData = 1'b0;
106             regOp2 = 1'b0;
107             regALU = instruction[2:0] == 3'b000 ? 4'b0110 :
108                                                     instruction[2:0] == 3'b001 ? 4'
                                                        b0111 :
109                                                     instruction[2:0] == 3'b010 ? 4'
                                                        b1000 :
110                                                     instruction[2:0] == 3'b011 ? 4'
                                                        b1001 :
111                                                     instruction[2:0] == 3'b100 ? 4'
                                                        b1010 :
112                                                     instruction[2:0] == 3'b101 ? 4'
                                                        b1011 :
113                                                     instruction[2:0] == 3'b110 ? 4'
                                                        b1100 :
114                                                     4'b1101;
115             regWriteType = 2'b00;
116             regWrite = 2'b10;
117             regSpecial = 2'b00;
118                                 regSP = 1'b0;
119         end
120         JUMP: begin
121             regTimer = 1'b0;
122             regWriteEnableReg = 1'b0;
123             regWriteEnableData = 1'b0;
124             regOp2 = 1'b0;
125             regALU = 4'b0000;
126             regWriteType = 2'b00;
127             regWrite = 2'b10;
128             regSpecial = 2'b00;
129                                 regSP = 1'b0;
130         end
131         STACK: begin
132             regTimer = 1'b0;
133             regWriteEnableReg = 1'b0;
134             regWriteEnableData = instruction[0];
135             regOp2 = 1'b0;
136             regALU = 4'b0000;
137             regWriteType = 2'b00;
138             regWrite = 2'b10;
139             regSpecial = 2'b00;
140                                 regSP = 1'b1;
141         end
```

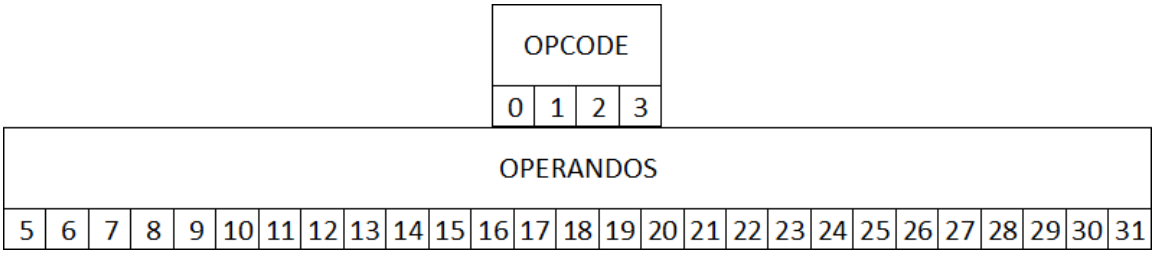
```
142         WRITE: begin
143             regTimer = 1'b0;
144             regWriteEnableReg = 1'b1;
145             regWriteEnableData = 1'b0;
146             regOp2 = 1'b0;
147             regALU = 4'b0000;
148             regWriteType = 2'b00;
149             regWrite = 2'b11;
150             regSpecial = 2'b00;
151                                 regSP = 1'b0;
152         end
153         COPY: begin
154             regTimer = 1'b0;
155             regWriteEnableReg = 1'b1;
156             regWriteEnableData = 1'b0;
157             regOp2 = 1'b0;
158             regALU = 4'b0000;
159             regWriteType = 2'b00;
160             regWrite = 2'b01;
161             regSpecial = instruction[1:0];
162                                 regSP = 1'b0;
163         end
164         LOAD: begin
165             regTimer = 1'b0;
166             regWriteEnableReg = 1'b1;
167             regWriteEnableData = 1'b0;
168             regOp2 = 1'b0;
169             regALU = 4'b0000;
170             regWriteType = 2'b00;
171             regWrite = 2'b00;
172             regSpecial = 2'b00;
173                                 regSP = 1'b0;
174         end
175         STORE: begin
176             regTimer = 1'b0;
177             regWriteEnableReg = 1'b0;
178             regWriteEnableData = 1'b1;
179             regOp2 = 1'b0;
180             regALU = 4'b0000;
181             regWriteType = 2'b00;
182             regWrite = 2'b10;
183             regSpecial = 2'b00;
184                                 regSP = 1'b0;
185         end
186         SLEEP: begin
187             regTimer = 1'b0;
188             regWriteEnableReg = 1'b0;
189             regWriteEnableData = 1'b0;
190             regOp2 = 1'b0;
191             regALU = 4'b0000;
192             regWriteType = 2'b00;
193             regWrite = 2'b10;
194             regSpecial = 2'b00;
195                                 regSP = 1'b0;
196         end
197                   INPUT: begin
198             regTimer = 1'b0;
199             regWriteEnableReg = 1'b1;
200             regWriteEnableData = 1'b0;
201             regOp2 = 1'b0;
```

```
202             regALU = 4'b0000;
203             regWriteType = 2'b00;
204             regWrite = 2'b11;
205             regSpecial = 2'b00;
206                                 regSP = 1'b0;
207         end
208         endcase
209
210 end
211
212 endmodule
```

2.2 Conjunto de instruções

As instruções possuem 32 *bits* de comprimento, sendo 4 *bits* reservados para a identificação da instrução (*OPCODE*), permitindo a existência de até 16 tipos de instrução diferentes, e 28 *bits* para operandos, como mostra a Figura 9.

Figura 9 – Formato da instrução



Fonte: O Autor

As instruções que poderão ser interpretadas pelo processador foram separadas em 14 grupos, cada grupo sendo identificado por um *OPCODE* diferente de acordo com a Tabela 2

Tabela 2 – Tipos de instrução

OPCODE	INSTRUÇÃO
0000	SOMA
0001	SUBTRAÇÃO
0010	MULTIPLICAÇÃO
0011	DIVISÃO
0100	DESLOCAMENTO
0101	LÓGICA
0110	*JUMP*
0111	PILHA
1000	*WRITE*
1001	*COPY*
1010	*LOAD*
1011	*STORE*
1101	*INPUT*

Fonte: O Autor

A seguir, serão apresentados os formatos dos 28 *bits* de operandos para cada tipo de instrução.

2.2.1 Soma e Subtração

As instruções dos tipos **soma** e **subtração** foram divididas em dois subtipos diferenciados pela *flag* T. Quando T possui o valor 0, a instrução é do subtipo 1, Figura 10(a), e armazena no registrador (REG. OUT) o resultado da soma/subtração entre os conteúdos armazenados nos registradores (REG. IN1) e (REG. IN2) . Quando T possui o valor 1, a instrução é do subtipo 2, Figura 10(b), e armazena no registrador (REG. OUT) o resultado da soma/subtração entre o conteúdo armazenado no registrador (REG. IN1) e o imediado (IMED. IN2).

Figura 10 – Operandos das instruções **soma** e **subtração**

(a) Soma/subtração - subtipo 1

REG. OUT					REG. IN1					REG. IN2					NULL											T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

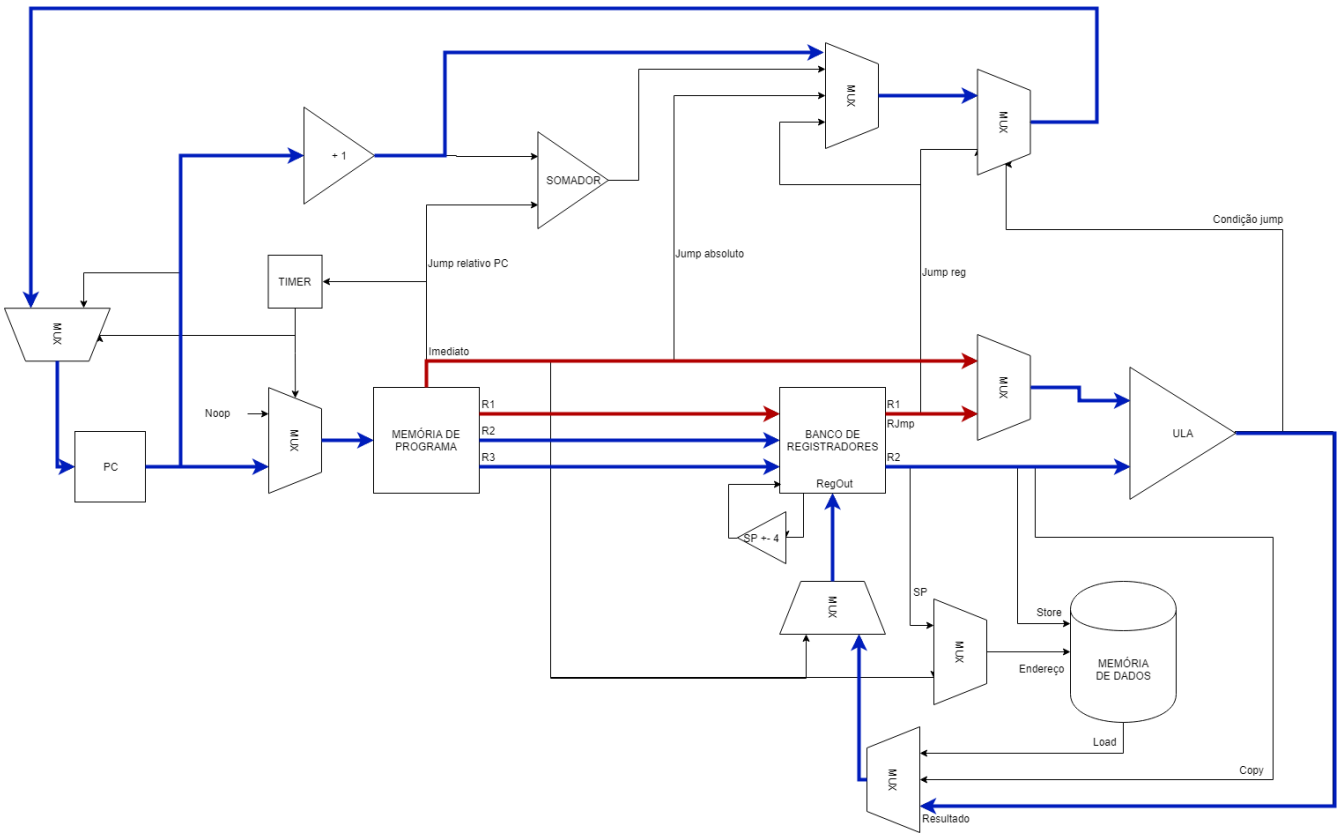
(b) Soma/subtração - subtipo 2

REG. OUT					REG. IN1					IMED. IN2																T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesses tipos de instrução são: **por registrador** para a saída (REG. OUT) e para as entradas (REG. IN1) e (REG. IN2), a segunda quando a *flag* assume o valor 0; e **imediato** para o segundo valor de entrada (IMED. IN2) quando a *flag* T assume o valor 1. O caminho de execução da instrução (*datapath*) pode ser observado na Figura 11, onde as setas em azul indicam caminhos tomados por todas as instruções desse tipo, enquanto as setas em vermelho indicam caminhos opcionais de acordo com a segunda entrada da operação.

Figura 11 – *Datapath* das instruções **soma** e **subtração**



Fonte: O Autor

2.2.2 Multiplicação e Divisão

As instruções dos tipos **multiplicação** e **divisão** também foram divididas em dois subtipos diferenciados pela *flag* T, que indica se a operação deve ser realizada entre os conteúdos de dois registradores, Figura 12(a), ou entre o conteúdo de um registrador e um valor imediato, Figura 12(b). A diferença entre esses tipos de insruções e as instruções de soma/subtração é que o resultado será armazenado em registradores específicos, reservados para aquele tipo de instrução; ou seja, os *bits* mais significativos do resultado de uma multiplicação são armazenados no registrador reservado **MS**, e os menos significativos no registrador **LS**. No caso da divisão, o quociente é armazenado no registrador **Quo** e o resto no registrador **Rem**.

Figura 12 – Operandos das instruções **multiplicação** e **divisão**

(a) Multiplicação/divisão - subtipo 1

REG. IN1					REG. IN2					NULL																T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

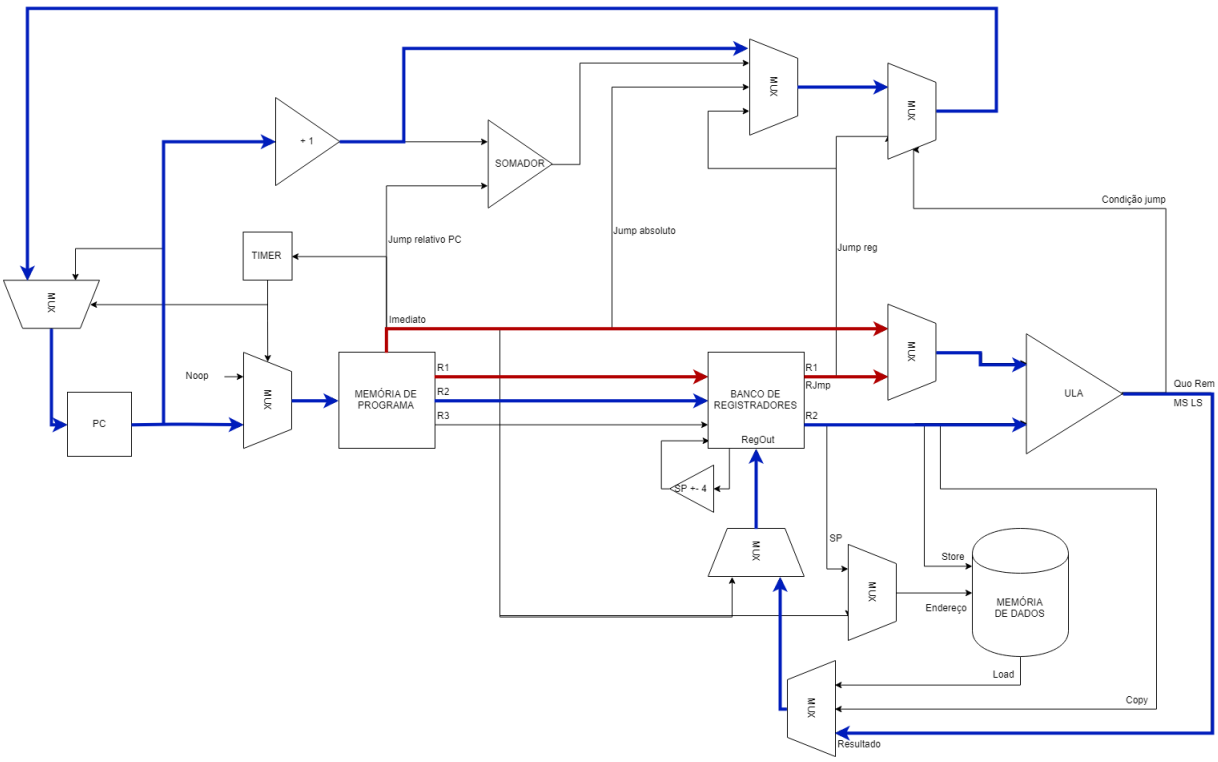
(b) Multiplicação/divisão - subtipo 2

REG. IN1					IMED. IN2																					T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesses tipos de instrução são: **por registrador** para as entradas (REG. IN1) e (REG. IN2), a segunda quando a *flag* assume o valor 0; e **imediato** para o segundo valor de entrada (IMED. IN2) quando a *flag* T assume o valor 1. O caminho de execução da instrução (*datapath*) pode ser observado na Figura 13, onde as setas em azul indicam caminhos tomados por todas as instruções desse tipo, enquanto as setas em vermelho indicam caminhos opcionais de acordo com a segunda entrada da operação.

Figura 13 – *Datapath* das instruções **multiplicação** e **divisão**



Fonte: O Autor

2.2.3 Deslocamento

Essa instrução, usando a nomenclatura da Figura 14, armazena no registrador (Reg. OUT) o conteúdo do registrador (REG. IN1) deslocado (IMED. IN2) *bits*. O deslocamento é para a direita quando a *flag* T assume o valor 0, e para a esquerda quando assumo o valor 1.

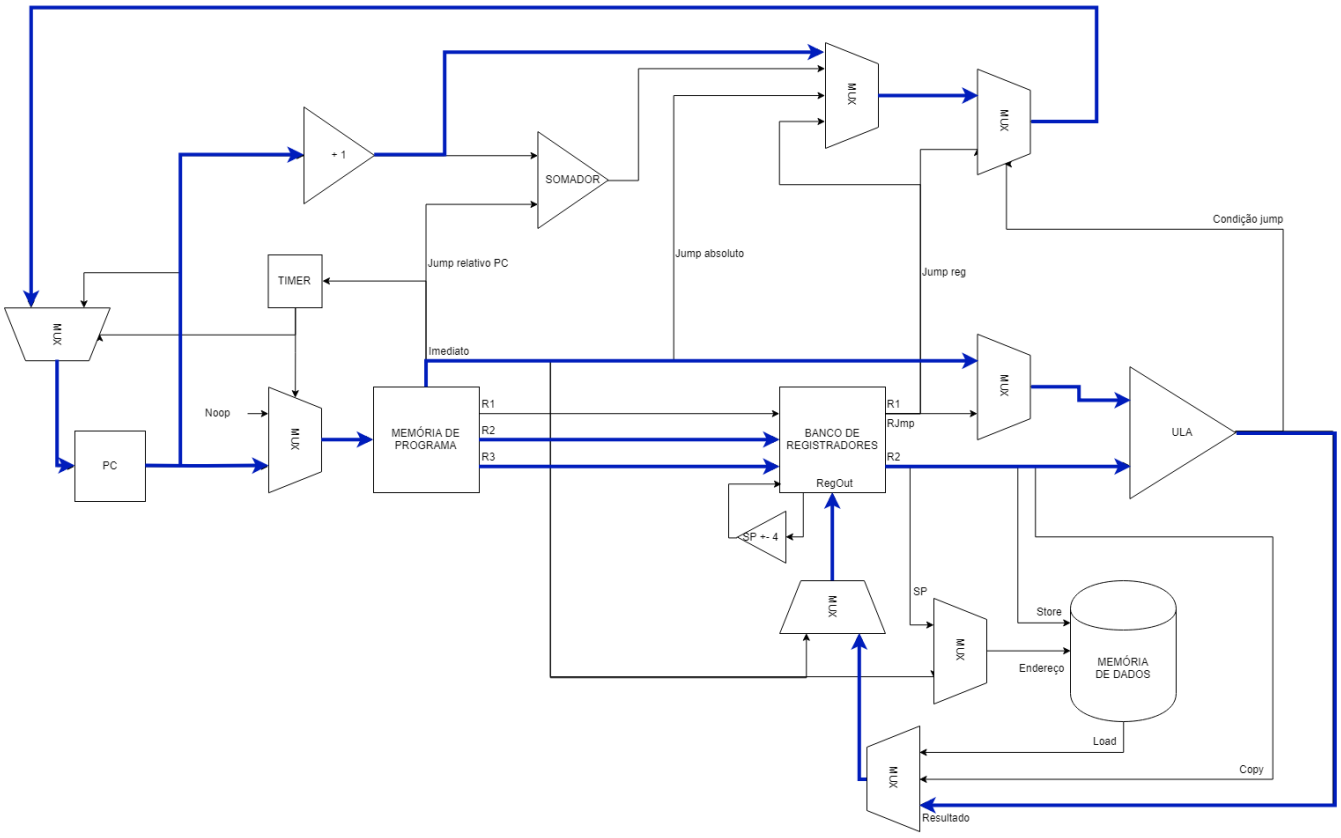
Figura 14 – Operandos das instruções de **deslocamento**

REG. OUT					REG. IN1					IME. IN2																T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a saída (REG. OUT) e para o primeiro valor de entrada (REG. IN), e **imediato** para o segundo valor de entrada (IMED. IN2). O caminho de execução da instrução, para ambos os sentidos de deslocamento, (*datapath*) pode ser observado na Figura 15.

Figura 15 – *Datapath* das instruções de **deslocamento**



Fonte: O Autor

2.2.4 Lógica

Esse tipo de instrução é dividido em 8 subtipos diferentes, identificados pela *flag* de 3 *bits* T de acordo com a Tabela 3. O resultado da operação lógica entre os registradores (REG. IN1) e (REG. IN2) é armazenado no registrador (REG. OUT), de acordo com a Figura 16(b), há uma exceção para a operação do tipo **NOT**, que utiliza apenas um registrador de entrada (REG. IN1), como mostra a Figura 16(a).

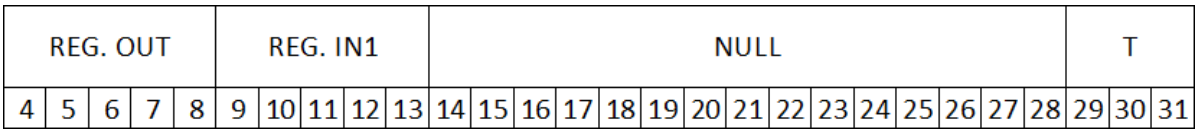
Tabela 3 – Subtipos de instrução lógica

T	OPERAÇÃO
000	*NOT*
001	*AND*
010	*OR*
011	*XOR*
100	IGUAL
101	DIFERENTE
110	MAIOR QUE
111	MENOR QUE

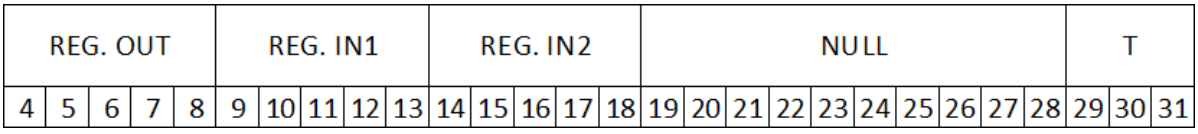
Fonte: O Autor

Figura 16 – Operandos das instruções do tipo **lógica**

(a) Lógica - tipo NOT



(b) Lógica - demais tipos

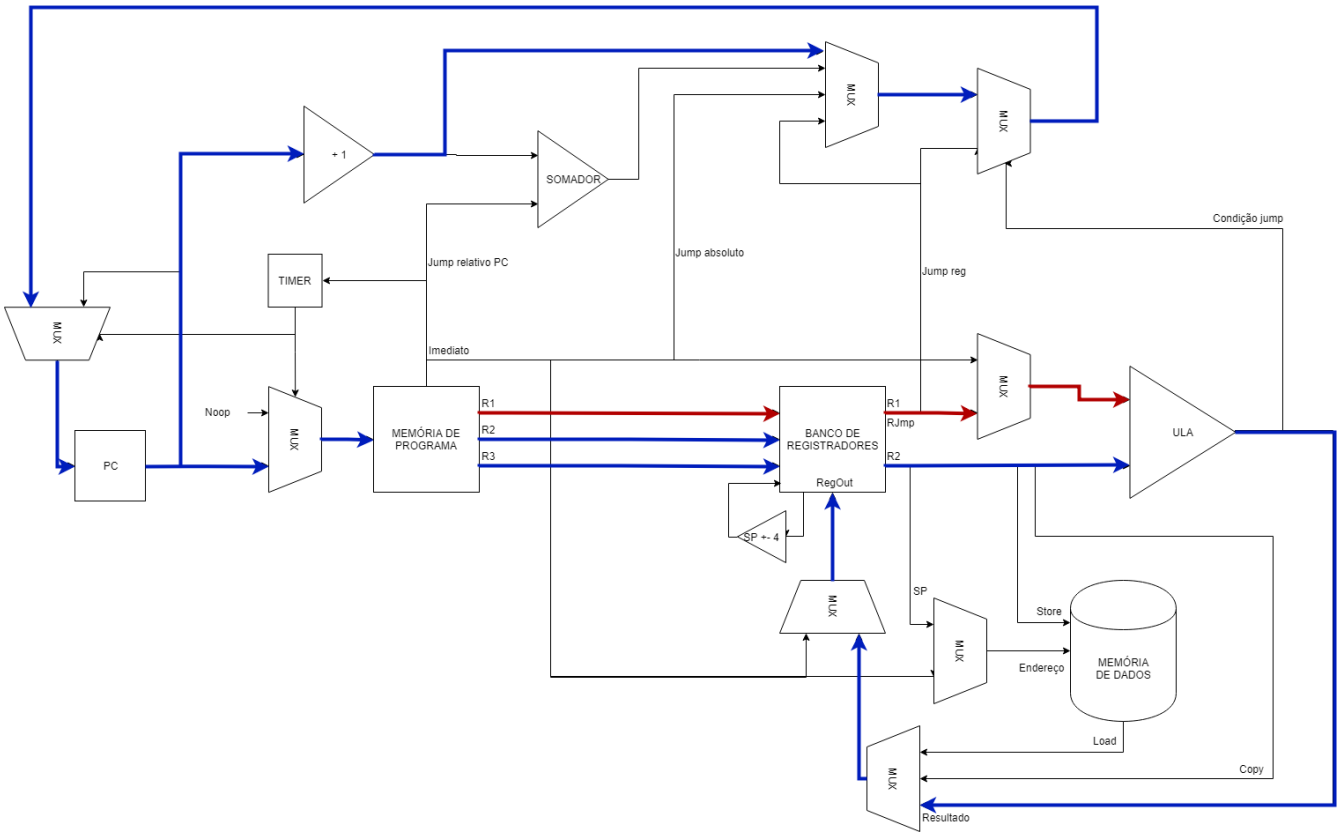


Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a saída (REG. OUT) e para o primeiro valor de entrada (REG. IN), além de para o segundo valor de entrada (REG. IN2) no caso de a operação não ser do subtipo NOT. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 17, onde as

setas em azul indicam o caminho tomado por qualquer subtipo de instrução lógica, e as setas em vermelho diferenciam a operação *NOT* das demais.

Figura 17 – *Datapath* das instruções do tipo **lógica**



Fonte: O Autor

2.2.5 *Jump*

As instruções desse tipo têm como objetivo desviar a contagem automática do registrador PC para acessar outras seções do programa. Essa operação é feita de quatro jeitos diferentes, identificados pela *flag* de 3 *bits* T de acordo com a Tabela 4. O primeiro tipo, "relativo ao PC por deslocamento", Figura 18(a), incrementa ou decrementa em (IMED. IN) o endereço armazenado no PC; ou seja, é relativo à sua posição atual. O segundo tipo, "absoluto direto", Figura 18(b), reposiciona o PC no endereço de programa (END. IN), independente de sua posição atual. O terceiro tipo, "absoluto por registrador", Figura 18(c), reposiciona o PC no endereço de programa armazenado no registrador (REG. JMP), independente de sua posição atual. O quarto tipo, "condicional absoluto por registrador", Figura 18(d), realiza a mesma operação que o terceiro tipo, porém somente sob a condição de que o conteúdo armazenado no registrador (REG. IN) represente o valor lógico **verdadeiro**.

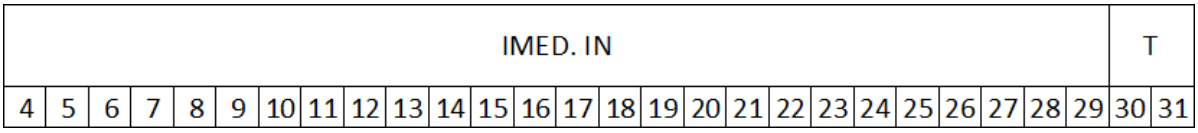
Tabela 4 – Subtipos de instrução *jump*

T	OPERAÇÃO
00	*Jump* relativo ao PC por deslocamento
01	*Jump* absoluto direto
10	Jump absoluto por registrador
11	*Jump* condicional absoluto por registrador

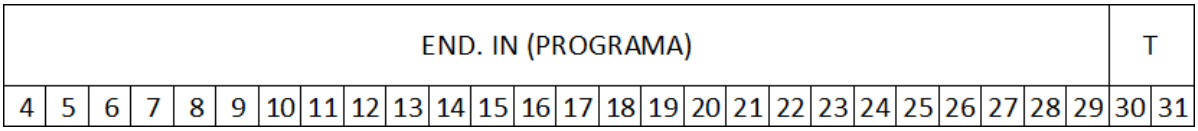
Fonte: O Autor

Figura 18 – Operandos das instruções do tipo ***jump***

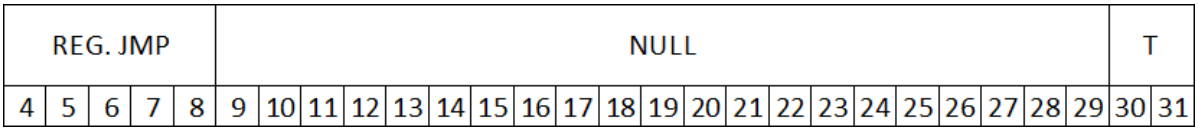
(a) Jump - relativo ao PC por deslocamento



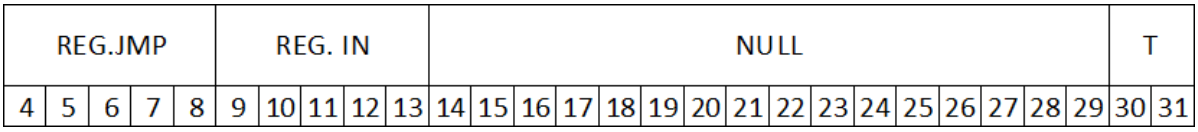
(b) *Jump* - absoluto direto



(c) *Jump* - absoluto por registrador



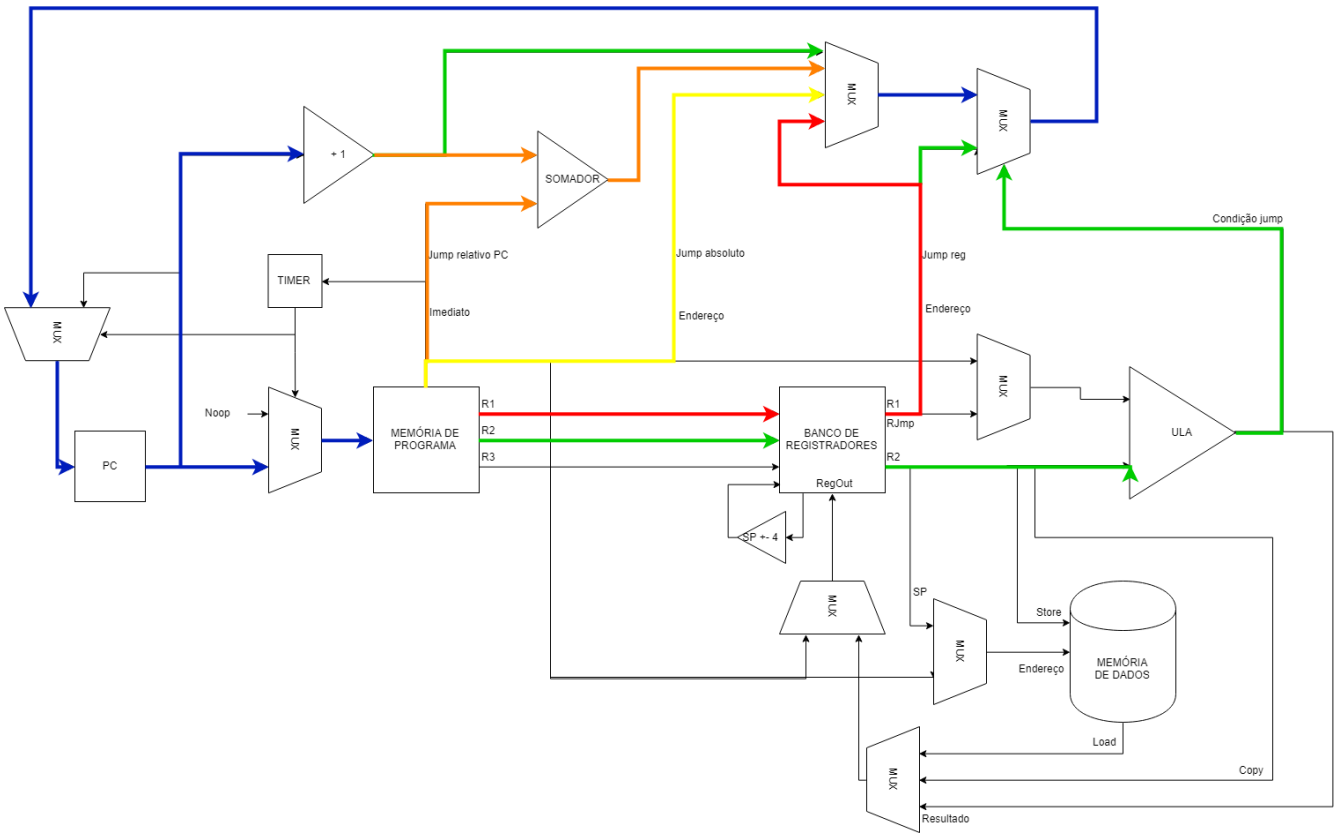
(d) *Jump* - condicional absoluto por registrador



Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para as entradas (REG. JMP) e (REG. IN), nos subtipos aos quais fazem parte, **imediado** para a entrada (IMED. IN) do primeiro subtipo, e **direto** para a entrada (END. IN) do segundo subtipo. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 19, onde as setas em azul indicam o caminho tomado por qualquer subtipo de instrução, as setas em laranja indicam o caminho tomado pelo primeiro subtipo, a em amarelo o caminho tomado pelo segundo subtipo, as em vermelho para o terceiro e as setas em verde indicam opções de caminhos tomados pelo quarto subtipo de instrução *jump*.

Figura 19 – *Datapath* das instruções do tipo ***jump***



Fonte: O Autor

2.2.6 Pilha

Esse tipo de instrução, usando a nomenclatura da Figura 20, realiza as operações de **empilhamento**, quando a *flag* T assume o valor 0, do conteúdo de um registrador (REG. IO) na pilha presente na memória principal; e **desempilhamento**, quando T assume o valor 1, do conteúdo presente no topo da pilha, armazenando-o no registrador (REG. IO).

Figura 20 – Operandos das instruções do tipo **pilha**

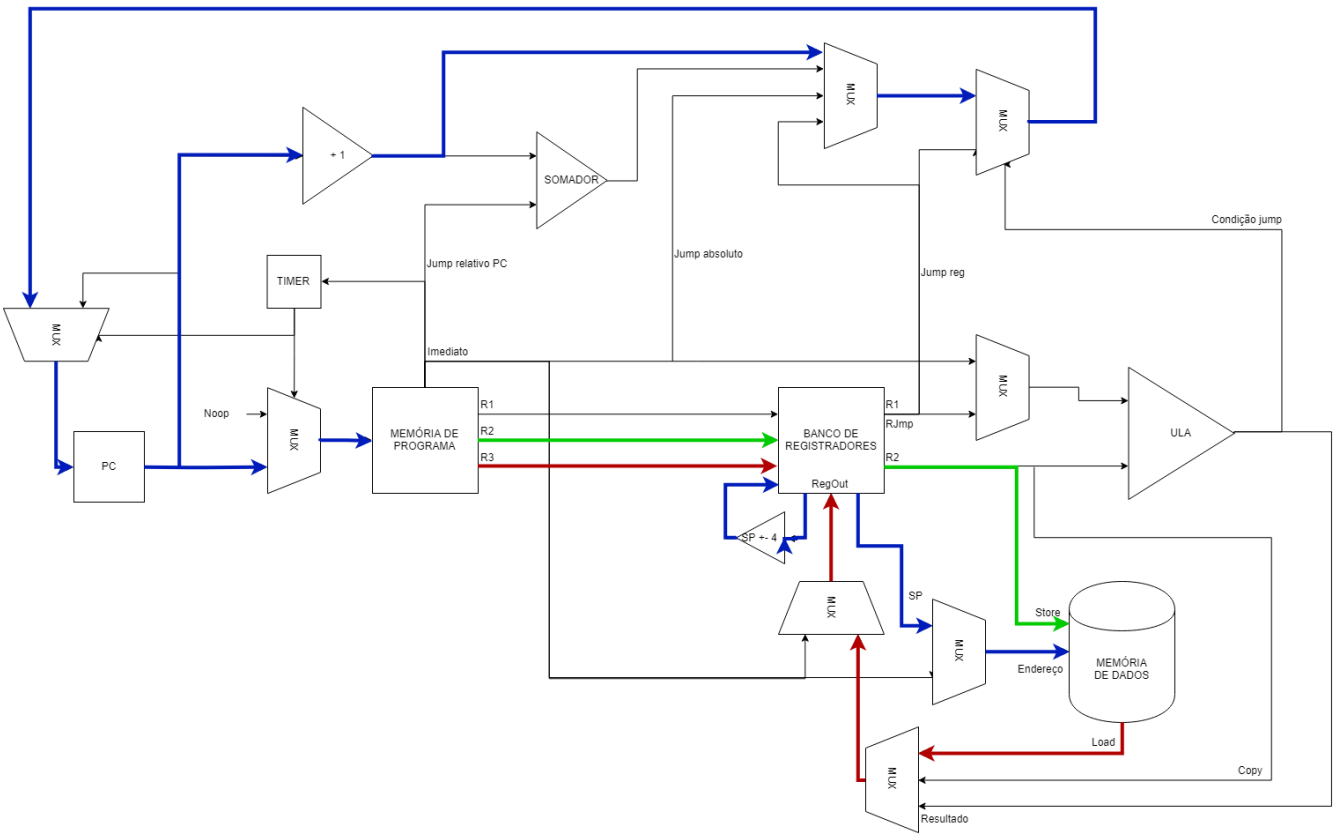
REG. IO					NULL																					T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a entrada/saída (REG. IO) e **por pilha** para referenciar o topo da pilha (SP) na memória de dados. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 21, onde as setas em azul indicam o caminho tomado tanto por empilhamento quando por desempilhamento de dados, as setas em verde indicam o caminho tomado

exclusivamente pelo empilhamento, e as setas em vermelho indicam o caminho tomado exclusivamente pelo desempilhamento.

Figura 21 – *Datapath* das instruções do tipo **pilha**



Fonte: O Autor

2.2.7 Write

Essa instrução armazena o valor (IMED. IN) no registrador (REG. OUT), Figura 22.

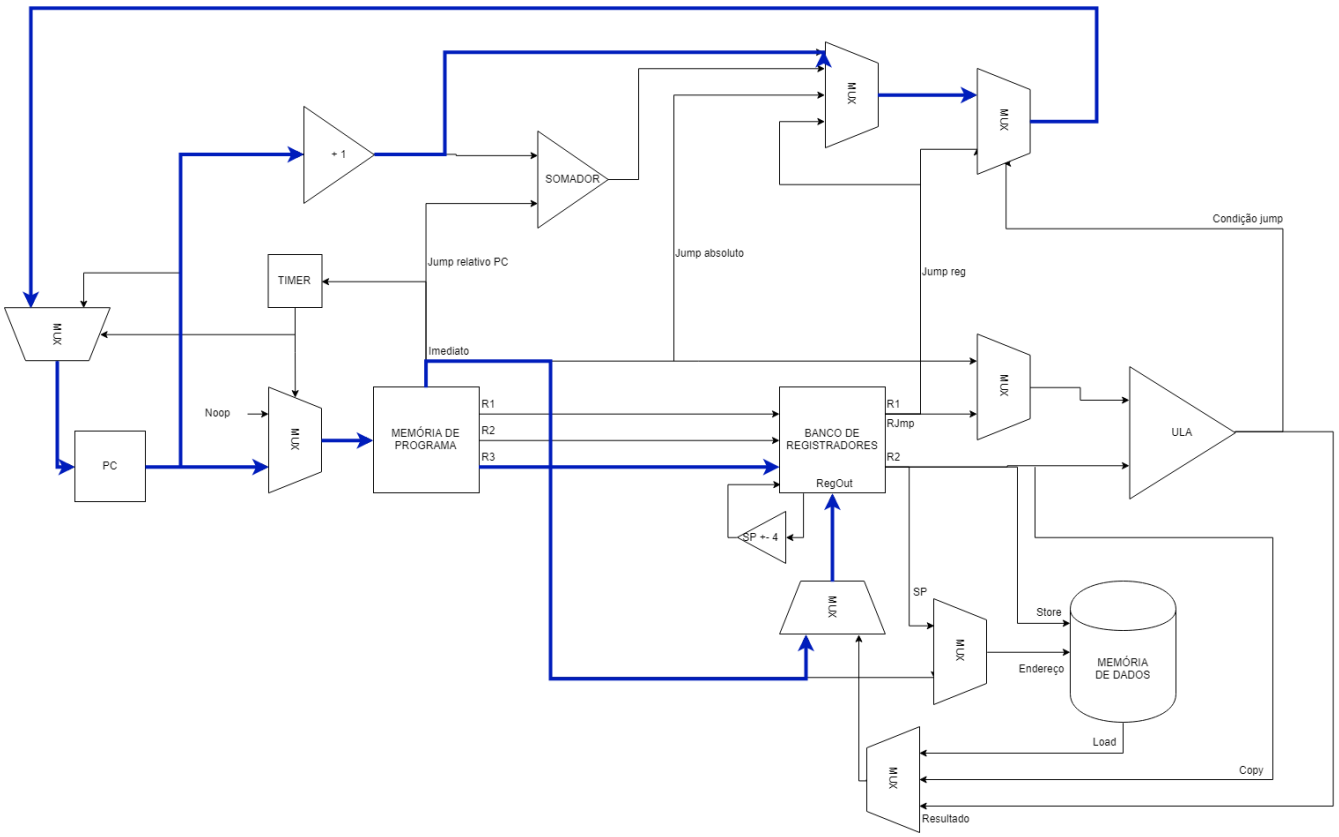
Figura 22 – Operandos da instrução ***write***

REG. OUT					IME. IN																					
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a saída (REG. OUT) e **imediato** para a entrada (IMED. IN) a ser armazenada no registrador. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 23.

Figura 23 – *Datapath* da instrução ***write***



Fonte: O Autor

2.2.8 Copy

Essa instrução copia o conteúdo do registrador (REG. IN) no registrador (REG. OUT), Figura 24. Essa instrução também é utilizada para alterar o valor da saída, copiando o valor de um registrador para o registrador de saída.

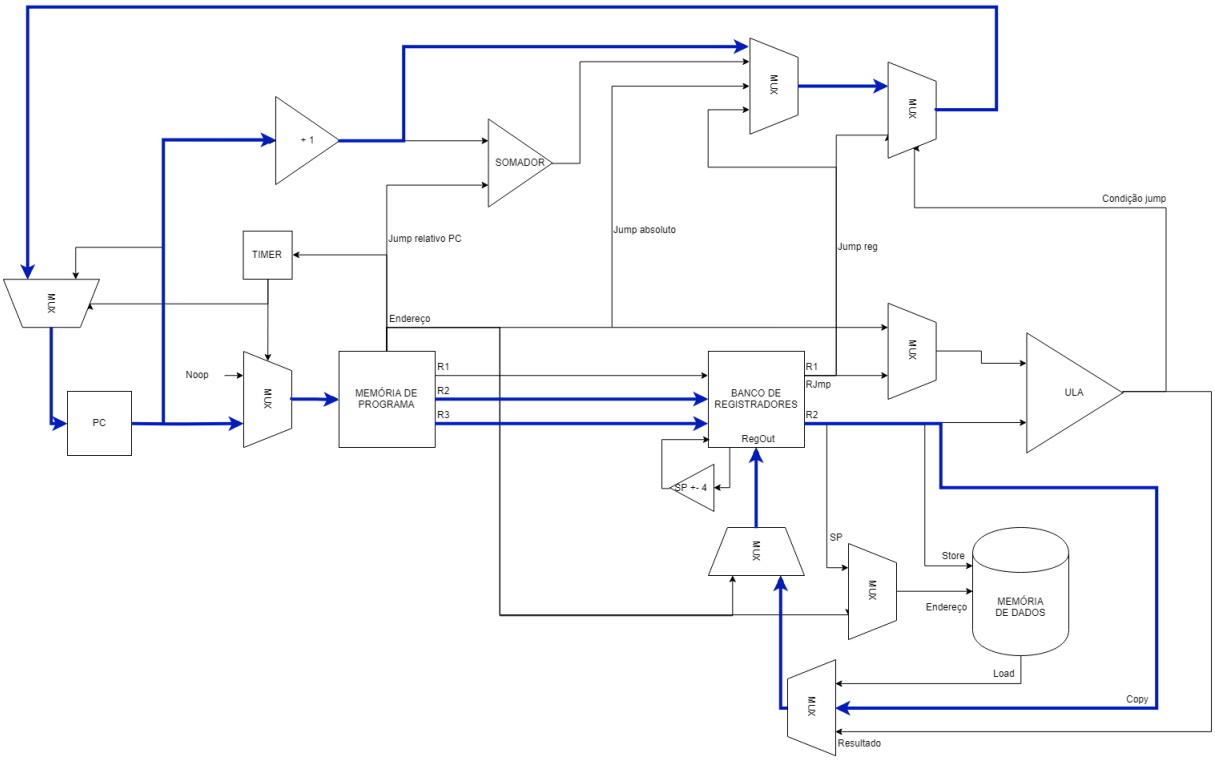
Figura 24 – Operandos da instrução ***copy***

REG. OUT					REG. IN					NULL																
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

O modo de endereçamento utilizado nesse tipo de instrução é: **por registrador**, tanto para para a entrada (REG. IN) quanto para a saída (REG. OUT). O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 25.

Figura 25 – *Datapath* da instrução ***copy***



Fonte: O Autor

2.2.9 Load e Store

Esses tipos de instrução realizam movimentações na memória de dados. Load, Figura 26(a), carrega no registrador (REG. OUT) o conteúdo presente no endereço (END. IN) da memória de dados, enquanto Store, Figura 26(b), armazena o conteúdo do registrador (REG. IN) no endereço (END. OUT) da memória da dados.

Figura 26 – Operandos das instruções ***load*** e ***store***

(a) Load

REG. OUT					END. IN (DADOS)																			
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

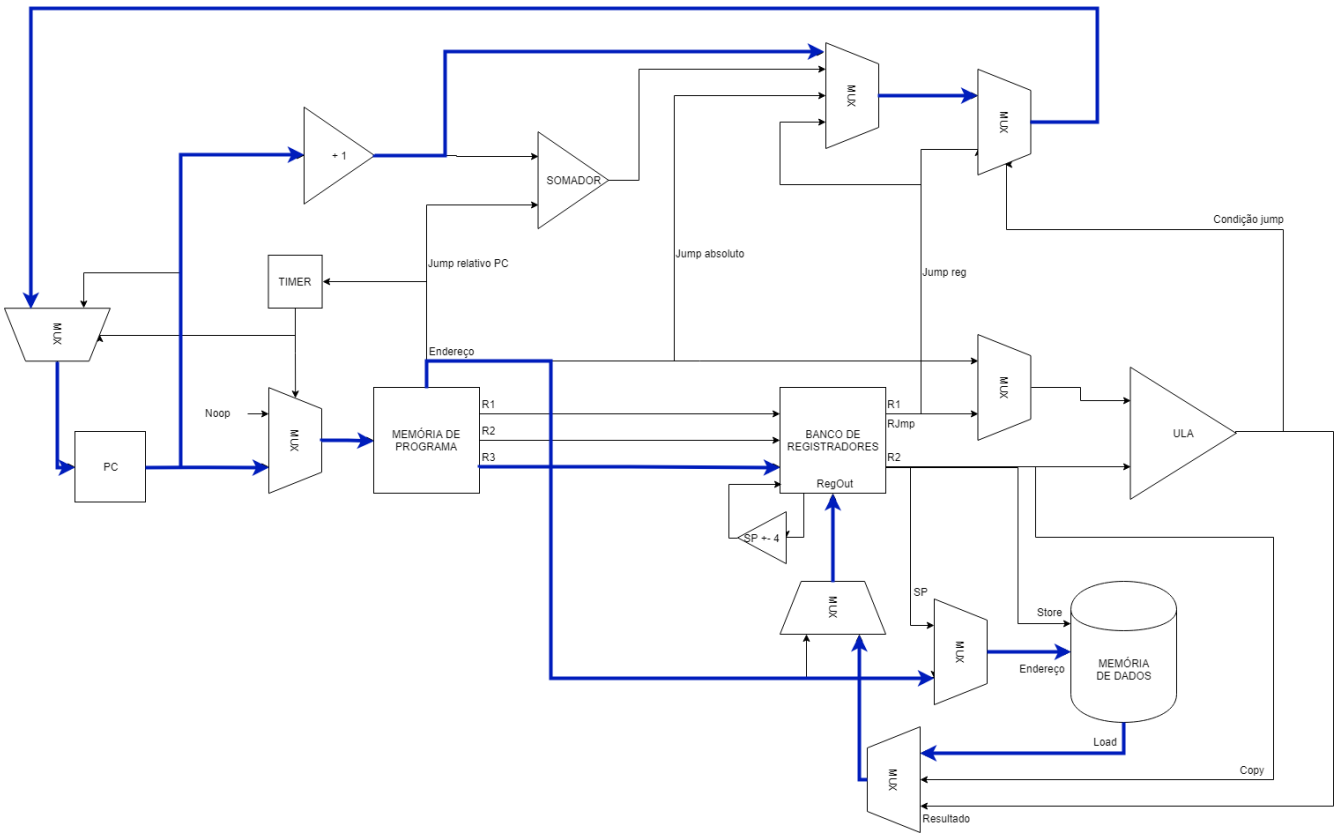
(b) Store

REG. IN					END. OUT (DADOS)																			
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fonte: O Autor

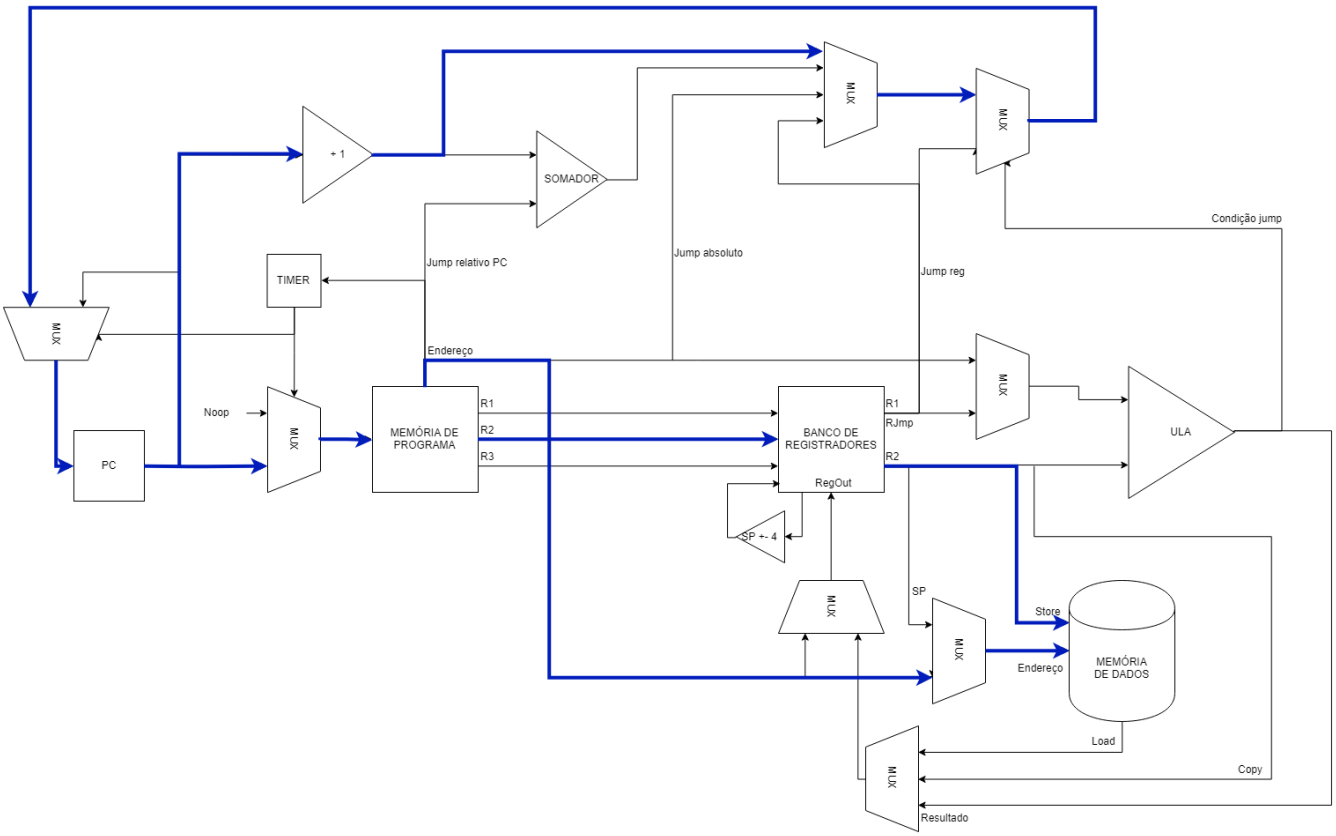
Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para o registrador de saída (REG. OUT), no caso da instrução store, e para o registrador de entrada (REG. IN), no caso da instrução load; e **direto** para a entrada (END. IN), no caso da instrução store, e para a saída (END. OUT), no caso da instrução load. O caminho de execução da instrução do tipo load pode ser observado na Figura 27, enquanto o da instrução do tipo store pode ser observado na Figura 28.

Figura 27 – *Datapath* da instrução ***load***



Fonte: O Autor

Figura 28 – *Datapath* da instrução ***store***



Fonte: O Autor

2.2.10 Input

A instrução *input* faz com que o PC pare de se movimentar até que a entrada seja fornecida. Para confirmar a inserção da entrada, o usuário deve alternar duas vezes o estado do switch de entrada (ativar e desativar). O formato dessa instrução pode ser visualizado na Figura 29.

Figura 29 – Operandos da instrução ***input***

REG. OUT					INPUT																			
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fonte: O Autor

O único modo de endereçamento utilizado nesse tipo de instrução é por registrador para indicar o registrador que receberá o valor de entrada.

2.3 Organização da memória

O processador possui três componentes de armazenamento de dados, como mostrado anteriormente nesse capítulo, *Program Memory*, *Register File* e *Data Memory*. O componente *Program Memory* armazena as instruções em código binário que foram carregadas no processador, e o acesso às instruções é feito usando o valor armazenado em PC. O componente *Register File* possui registradores que armazenam dados que estão sendo manipulados no momento e constantemente pelo processador. Por último, o componente *Data Memory* é capaz de armazenar uma quantidade relativamente grande de informações e é usado para guardar dados que não serão usados constantemente em operações do programa; além disso, é nesse componente em que a pilha apresentada anteriormente se encontra, empilhando dados em endereços de forma decrescente a partir do último endereço da memória. Todos esses componentes armazenam dados em espaços de 32 *bits*, que equivale ao tamanho das instruções, dos registradores e dos *slots* de memória presentes no bloco *Data Memory* .

3 Compilador

O processo de compilação é dividido em duas fases, de análise e de síntese, que serão explicadas a seguir.

3.1 Fase de análise

A fase de análise é responsável pela validação do código fonte para certificar que ele está dentro dos padrões da linguagem a ser traduzida, que no caso é C-. Essa fase é dividida em três etapas, análise léxia, análise sintática e análise semântica, que serão apresentadas com mais detalhes no decorrer dessa seção.

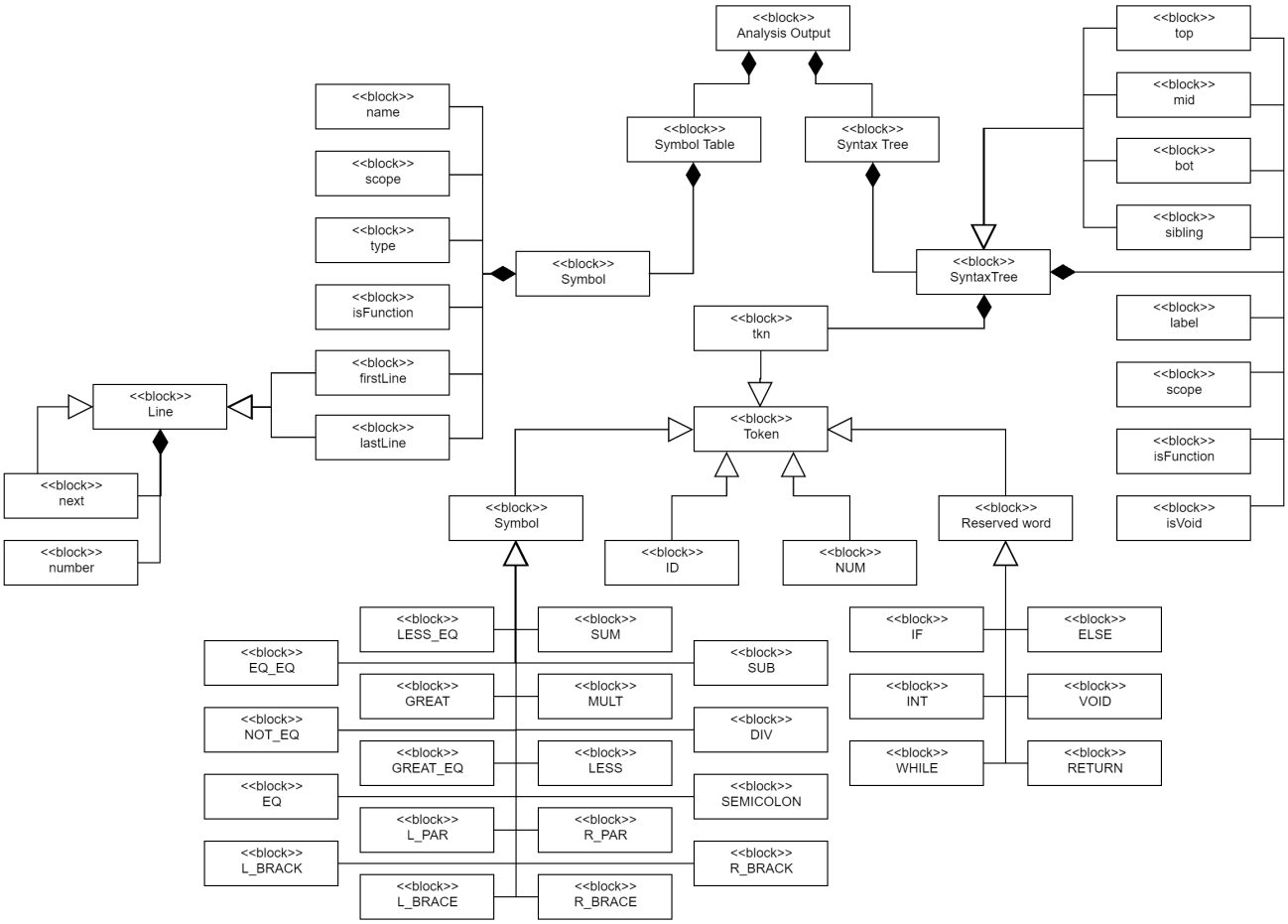
3.1.1 Modelagem

Antes de explicar detalhadamente cada etapa da fase de análise, serão apresentados os diagramas de blocos e de atividades, que foram modelados para consulta e mais fácil compreensão dessa fase.

3.1.1.1 Diagrama de blocos

Como ilustra o diagrama de blocos na Figura 30, os dados gerados pela etapa de análise são uma tabela de símbolos e uma árvore sintática. Cada nó da árvore sintática possui até três nós filhos, *top*, *mid* e *bot*, e um nó irmão, *sibling*, além dos dados sobre o próprio nó, que são o lexema (*label*), o escopo (*scope*), se é uma função (*isFunction*), o tipo (*isVoid*) e o *token* (*tkn*) ao qual a palavra equivale. Cada identificador presente na tabela de símbolos possui um nome (*name*), um escopo (*scope*), um tipo (*type*), se é uma função (*isFUnction*) e uma lista encadeada de linhas do código fonte onde o identificador aparece.

Figura 30 – Diagrama de blocos da fase de análise

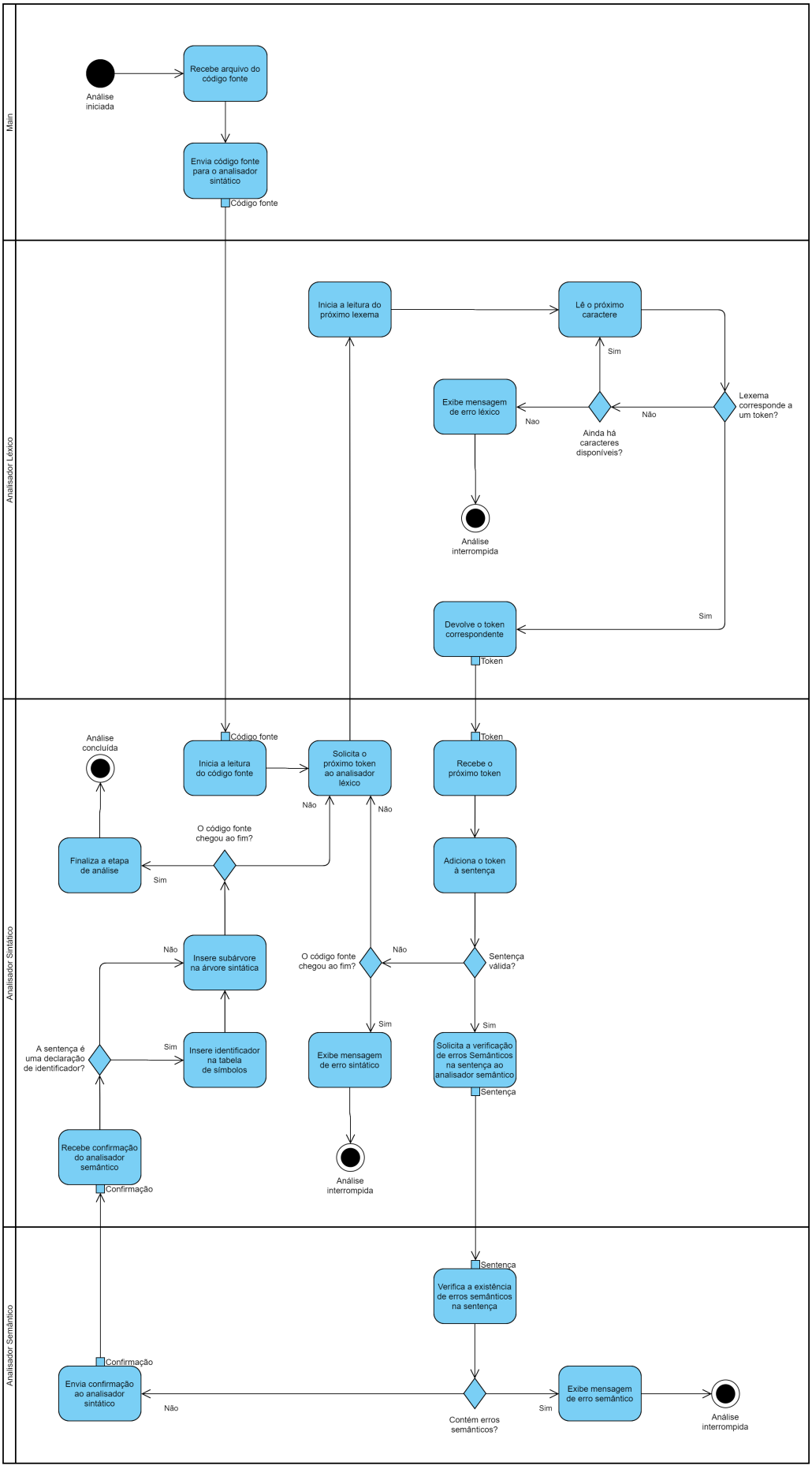


Fonte: O Autor

3.1.1.2 Diagrama de atividades

Como ilustra o diagrama de atividades na Figura 31, o processo de análise é iniciado com a recepção do código fonte e envio do mesmo para o analisador sintático, que solicita *tokens* ao analisador léxico para formar e validar sentenças, interrompendo a análise e exibindo uma mensagem de erro sintático caso uma sentença inválida seja encontrada, e enviando sentenças válidas para o analisador semântico. Quando um token é solicitado ao analisador léxico pelo analisador sintático, o próximo lexema do código fonte é lido e validado; se o lexema for válido, ele é retornado para o analisador sintático, caso contrário, a análise é interrompida e uma mensagem de erro léxico é exibida. Quando uma sentença é recebida pelo analisador semântico, é verificada a presença de erros semânticos nela e, se a sentença for válida, uma confirmação é enviada ao analisador sintático; se não for válida, o analisador semântico interrompe a análise e exibe uma mensagem de erro semântico.

Figura 31 – Diagrama de atividades da fase de análise



Fonte: O Autor

3.1.2 Análise léxica

A etapa de análise léxica é responsável pela verificação individual de cada lexema do código fonte, para que seja certificado que aquele lexema corresponde a um token pertencente à linguagem. No caso da linguagem C-, os lexemas válidos são:

- Palavras reservadas: *else*, *if*, *int*, *return*, *void*, *while*
- Símbolos: + - * / < <= > >= == != = ; , [ ] { } /* */
- Números
- Identificadores: lexemas que não são palavras reservadas, começam com uma letra e possuem apenas letras

O analisador léxico foi desenvolvido usando a ferramenta *Flex* e o código foi disponibilizado junto a este relatório.

3.1.3 Análise sintática

A etapa de análise sintática é responsável pela verificação das cadeias de lexemas, ou sentenças. Para realizar isso, foi utilizada a ferramenta *Bison* em conjunto com a gramática livre de contexto da linguagem C-, Figura 32.

Figura 32 – Gramática livre de contexto da linguagem C-

programa → declaração-lista
declaração-lista → declaração-lista declaração | declaração
declaração → var-declaração | fun-declaração
var-declaração → tipo-especificador ID ; | tipo-especificador ID [NUM] ;
tipo-especificador → int | void
fun-declaração → tipo-especificador ID (params) composto-decl
params → param-lista | void
param-lista → param-lista, param | param
param → tipo-especificador ID | tipo-especificador ID [ ]
composto-decl → { local-declarações statement-lista } | {local-declarações} | {statement-lista} | { }
local-declarações → local-declarações var-declaração | var-declaração
statement-lista → statement-lista statement | statement
statement → expressão-decl | composto-decl | seleção-decl | iteração-decl | retorno-decl
expressão-decl → expressão ; | ;
seleção-decl → if (expressão) statement | if (expressão) statement else statement
iteração-decl → while (expressão) statement
retorno-decl → return ; | return expressão;
expressão → var = expressão | simples-expressão
var → ID | ID [expressão]
simples-expressão → soma-expressão relacional soma-expressão | soma-expressão
relacional → <= | < | > | >= | == | !=
soma-expressão → soma-expressão soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → * | /
fator → (expressão) | var | ativação | NUM
ativação → ID (arg-lista) | ID ( )
arg-lista → arg-lista , expressão | expressão

Fonte: (1)

3.1.4 Análise semântica

A análise semântica é responsável por verificar se o todo tem significado. O analisador semântico deve reconhecer incoerências que não seriam identificadas ao observar os lexemas indivudalmente ou mesmo as sentenças formadas. A verificação dessas incoerências é feita juntamente com a análise sintática, dentro da sintaxe da ferramenta *Bison*, conforme as sentenças vão sendo formadas. Para realizar as verificações, é montada e utilizada a tabela de símbolos apresentada anteriormente neste capítulo. No projeto, os erros semânticos verificados pelo analisador semântico são:

- Variável não declarada no escopo;
- Atribuição inválida (atribuir o retorno de uma função void a uma variável);
- Declaração inválida (declarar variável como void);
- Declaração inválida (variável já declarada no escopo);
- Chamada de função não declarada;
- Função main não declarada;
- Declaração inválida (nome de variável já utilizado como nome de função).

3.2 Fase de síntese

A fase de síntese do processo de compilação é responsável pela tradução do código na linguagem fonte no código equivalente na linguagem alvo. No caso, o objetivo da compilação é traduzir códigos em linguagem C- em códigos binários, respeitando o conjunto de instruções do processador. Essa fase é dividida em três etapas que serão discutidas no decorrer desse capítulo, sendo elas, geração de código intermediário, geração se código *assembly* e geração de código binário.

3.2.1 Modelagem

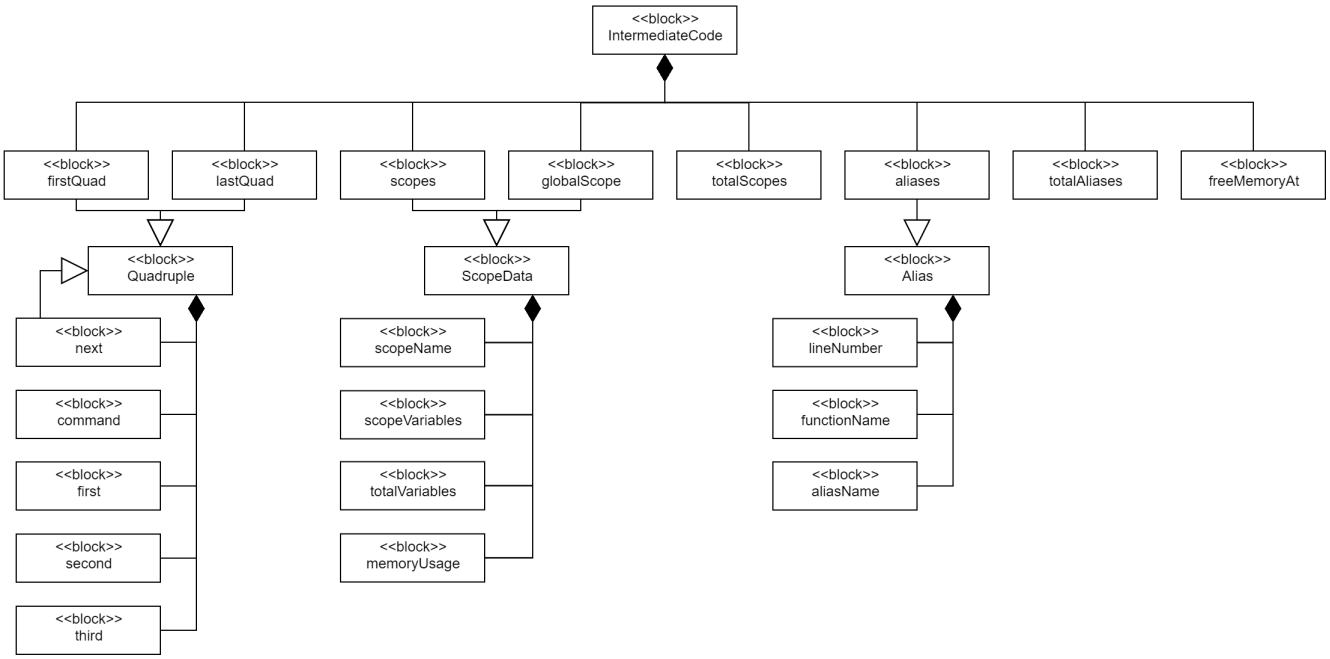
Antes de serem explicadas as três etapas de síntese com detalhes, serão apresentados os diagramas de atividades e de blocos da fase de síntese para se obter uma compreensão em alto nível de abstração do que é feito nessa fase.

3.2.1.1 Diagramas de blocos

O diagrama de blocos da etapa de geração de código intermediário, Figura 33, mostra que o código intermediário é formado por uma lista encadeada de quádruplas (*firstQuad*, *lastQuad*), um vetor de escopos (*scopes*), apelidos de linhas (*aliases*) e um

marcador de posição de memória livre (*freeMemoryAt*). Cada quádrupla é formada por um comando (*command*) e até três operandos (*first*, *second*, *third*). Cada escopo é formado por um nome (*scopeName*), um vetor de variáveis (*scopeVariables*) e um total de memória utilizada (*memoryUsage*). Cada apelido de linha é formado por um nome de identificação (*aliasName*), um número de linha (*lineNumber*) e, possivelmente, um nome de função (*functionName*).

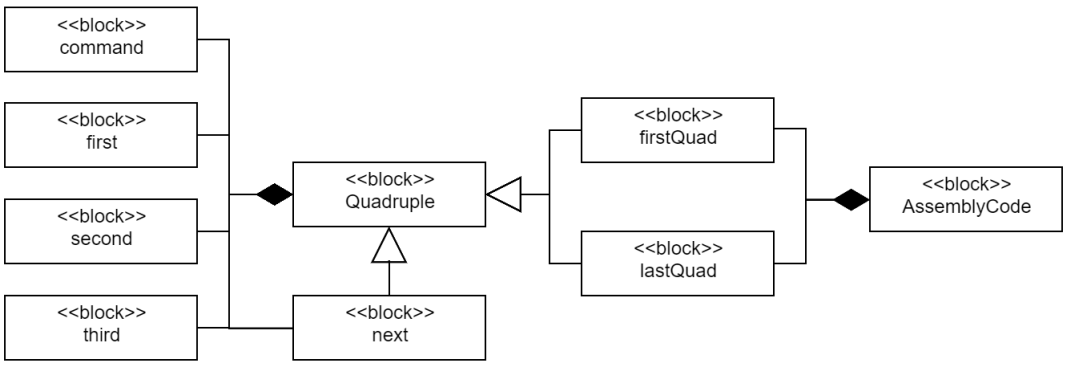
Figura 33 – Diagrama de blocos do gerador de código intermediário



Fonte: O Autor

O diagrama de blocos da geração de código assembly, ilustrado pela Figura 34, mostra que a estrutura do código assembly possui uma lista encadeada de quádruplas exatamente como o código intermediário, mas nada além disso, dado que o gerenciamento de memória e a nomeação de linhas são feitos durante a geração de código intermediário.

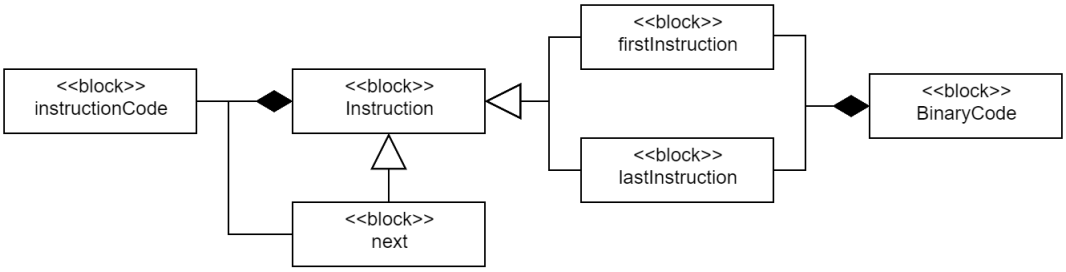
Figura 34 – Diagrama de blocos do gerador de código assembly



Fonte: O Autor

O diagrama de blocos da geração de código intermediário, Figura 35, mostra que este possui apenas uma lista encadeada de instruções, que são vetores de caracteres de 32 *bits* de comprimento.

Figura 35 – Diagrama de blocos do gerador de código binário

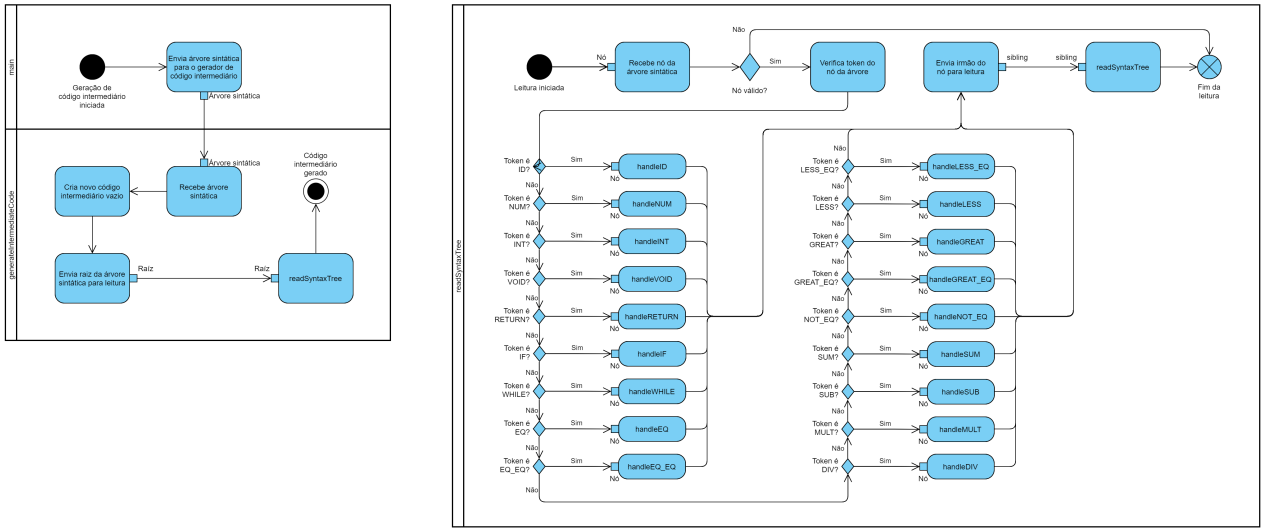


Fonte: O Autor

3.2.1.2 Diagramas de atividades

O diagrama de atividades ilustrado na Figura 36 mostra o processo de geração de código intermediário. No início, a árvore sintátia é recebida pelo gerador e redirecionada para a função recursiva *readSyntaxTree*, que realiza a leitura de um nó da árvore. Essa função, por sua vez, identifica o token daquele nó e redireciona para uma das funções que tratam com cada tipo de token diferente. Depois disso, o nó irmão é enviado recursivamente para leitura. Vale ressaltar que todos os nós têm seus filhos enviados para leitura dentro das funções de tratamento.

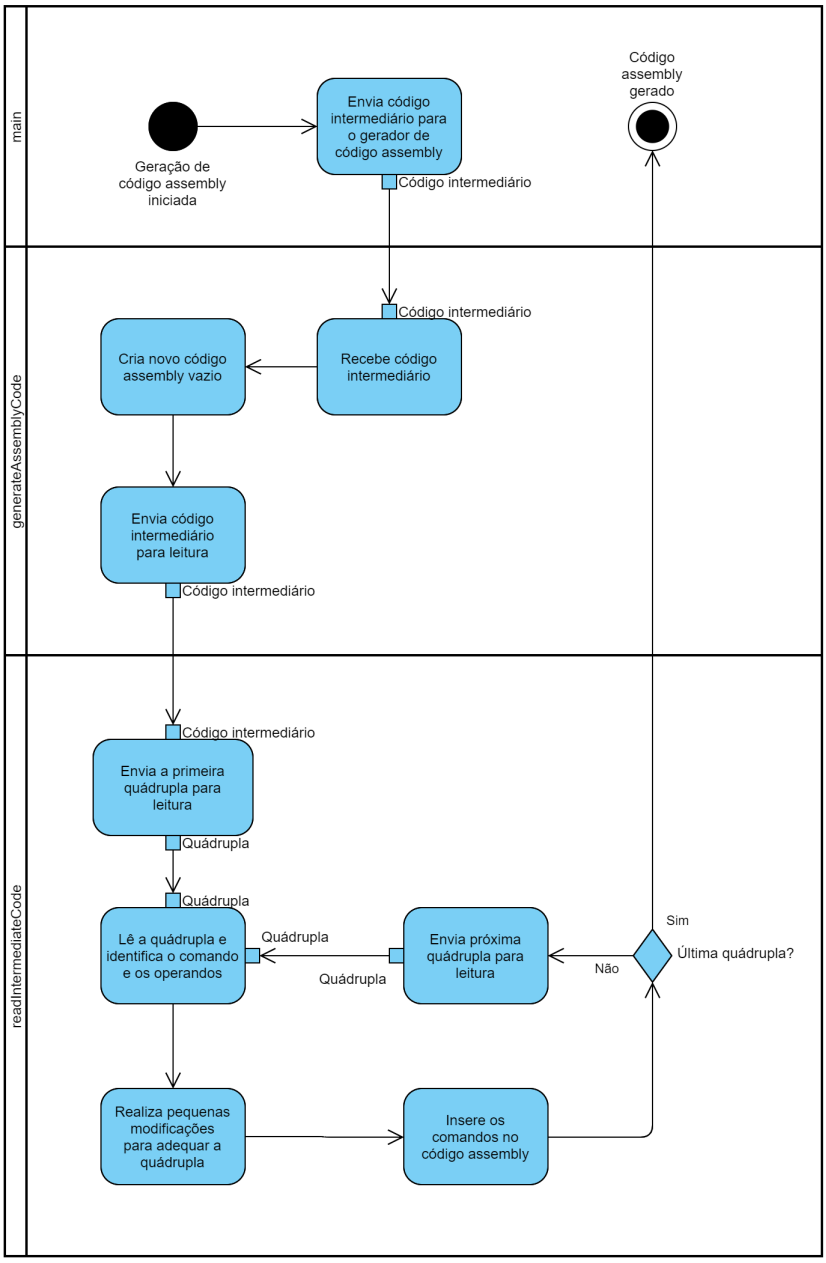
Figura 36 – Diagrama de atividades do gerador de código intermediário



Fonte: O Autor

A Figura 37 ilustra o diagrama do gerador de código *assembly*, mostrando que todas as quádruplas do códito intermediário são lidas e que são realizadas apenas algumas correções nelas antes de serem inseridas no código assembly.

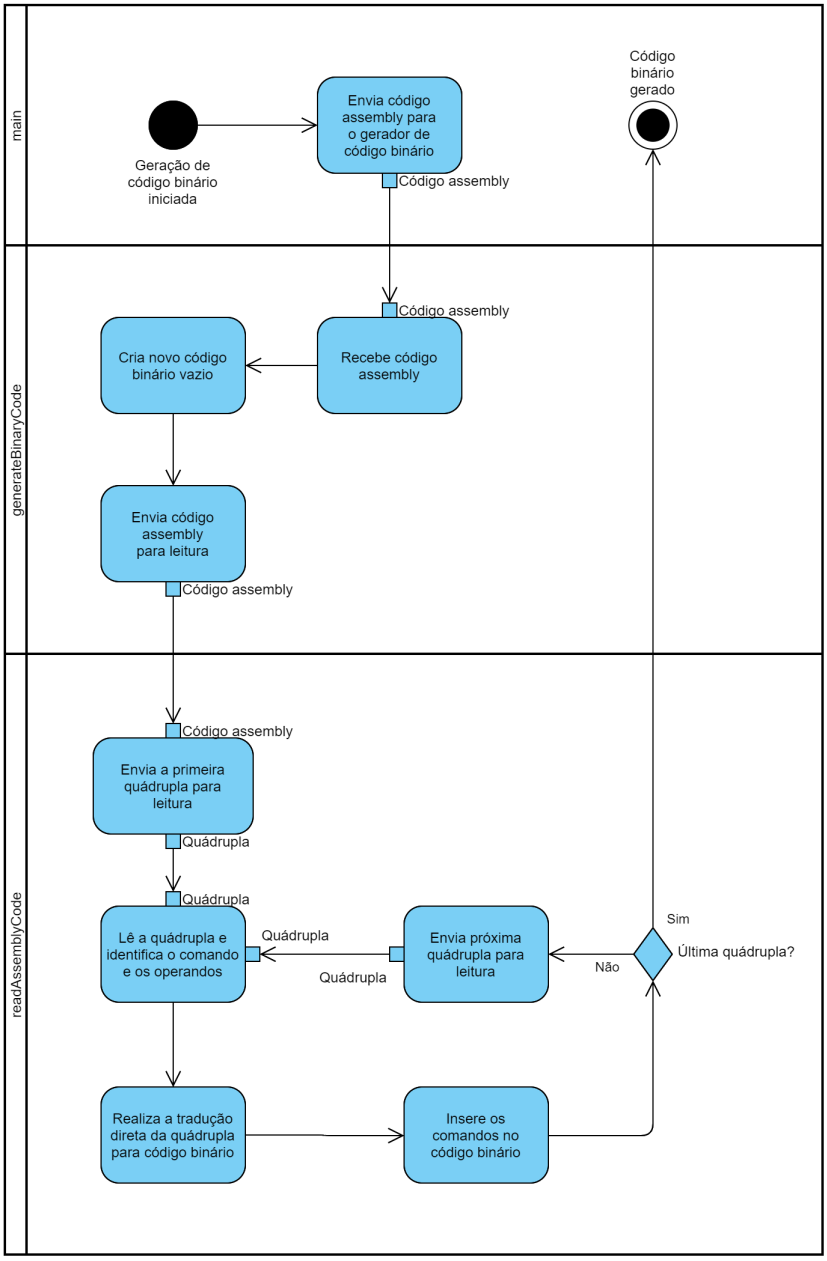
Figura 37 – Diagrama de atividades do gerador de código *assembly*



Fonte: O Autor

O diagrama de atividades do gerados de código binário, Figura 38, mostra que todas as quádruplas do código *assembly* são lidas e que é feita uma tradução direta delas para código binário.

Figura 38 – Diagrama de atividades do gerador de código binário



Fonte: O Autor

3.2.2 Geração do código intermediário

O gerador de código intermediário recebe a árvore sintática gerada na fase de análise e faz a leitura dos nós. Cada tipo de nó recebe um tratamento diferente e tem como objetivo gerar uma sequência específica de quádruplas que equivale ao trecho de código na linguagem fonte. As quádruplas geradas fazem parte do conjunto presente na Tabela 5

Tabela 5 – Tipos de quádrupla

Quádrupla	Descrição	Quádrupla	Descrição
SUM	soma	*JUMPIF*	desvio se
SUB	subtração	*WRITE*	atribuição de número
MULT	multiplicação	*LOAD*	carregamento da memória
DIV	disisão	*STORE*	armazenamento na memória
EQ_EQ	comparação igual	*EQ*	atribuição de valor
NOT_EQ	comparação diferente	*PUSH*	empilhamento
GREAT	comparação maior que	*POP*	desempilhamento
LESS	comparação menor que	*INPUT*	atribuição do valor da entrada
NOT	negação	*OUTPUT*	exibição da saída
JUMP	desvio		

Fonte: O Autor

Para atingir esse objetivo, foi utilizada como base a estrutura de dados ilustrada na Figura 39, onde é possível observar que o código intermediário possui uma lista de quádruplas (*Quadruple*), escopos, (*ScopeData*), apelidos de linhas (*Alias*) e uma marcação de posição de memória livre (*freeMemoryAt*), como foi mostrado no diagrama de blocos no início dessa seção.

Figura 39 – Estrutura do gerador de código intermediário

```
typedef struct Quadruple {
    char *command;
    char *first;
    char *second;
    char *third;
    struct Quadruple *next;
} Quadruple;

typedef struct {
    int lineNumber;
    char *functionName;
    char *aliasName;
} Alias;

typedef struct {
    char *scopeName;
    char **scopeVariables;

    int totalVariables;

    int memoryUsage;
} ScopeData;
```

```
typedef struct {
    Quadruple *firstQuad;
    Quadruple *lastQuad;
    int lastLine;

    ScopeData **scopes;
    int totalScopes;

    ScopeData *globalScope;

    Alias **aliases;
    int totalAliases;

    int freeMemoryAt;
} IntermediateCode;
```

Fonte: O Autor

A função principal da geração de código intermediário se chama *readSyntaxTree*, conforme ilustra a Figura 40. Essa função recebe a raíz da árvore sintática gerada na fase de síntese e redireciona o nó para uma função de tratamento específica de acordo com seu token, além de realizar uma chamada recursiva para leitura do nó irmão (*node->sibling*). Cada função de tratamento faz uma manipulação diferente do nó, mas todas redirecionam seus filhos recursivamente para leitura.

Figura 40 – Função principal do gerador de código intermediário

```
char *readSyntaxTree(SyntaxTree *node) {
    if (node) {
        switch (node->tkn) {
            case ID: {
                char *result = handleID(node);
                if (strlen(result) > 0) return result;
            } break;
            case NUM: return handleNUM(node); break;
            case INT: handleINT(node); break;
            case VOID: handleVOID(node); break;
            case RETURN: handleRETURN(node); break;
            case IF: handleIF(node); break;
            case WHILE: handleWHILE(node); break;
            case EQ: handleEQ(node); break;
            case EQ_EQ: return handleEQ_EQ(node); break;
            case LESS_EQ: return handleLESS_EQ(node); break;
            case LESS: return handleLESS(node); break;
            case GREAT: return handleGREAT(node); break;
            case GREAT_EQ: return handleGREAT_EQ(node); break;
            case NOT_EQ: return handleNOT_EQ(node); break;
            case SUM: return handleSUM(node); break;
            case SUB: return handleSUB(node); break;
            case MULT: return handleMULT(node); break;
            case DIV: return handleDIV(node); break;
        }

        readTreeDiscardReturn(node->sibling);
    }
    return "";
}
```

Fonte: O Autor

3.2.3 Geração do código assembly

O gerador de código *assembly* faz a leitura do código intermediário a fim de fazer as modificações necessárias para adequar as quádruplas à linguagem alvo, ou seja, transformá-

las em instruções definidas na arquitetura do processador. As quádruplas adequadas ao processador com suas respectivas descrições estão na Tabela 6.

Tabela 6 – Tipos de quádrupla

Quádrupla	Descrição
SUM0	soma - subtipo 1
SUM1	soma - subtipo 2
SUB0	subtração - subtipo 1
SUB1	subtração - subtipo 2
MULT0	multiplicação - subtipo 1
MULT1	multiplicação - subtipo 2
DIV0	divisão - subtipo 1
DIV1	divisão - subtipo 2
LOGIC0	lógica - *NOT*
LOGIC4	lógica - IGUAL
LOGIC5	lógica - DIFERENTE
LOGIC6	lógica - MAIOR QUE
LOGIC7	lógica - MENOR QUE
JUMP1	jump - absoluto direto
JUMP2	jump - absoluto por registrador
JUMP3	jump - condicional absoluto por registrador
STACK0	pilha - empilhamento
STACK1	pilha - desempilhamento
WRITE	atribuição de número em registrador
COPY	cópia do valor de um registrador em outro
LOAD	carregamento de valor da memória em um registrador
STORE	armazenamento de valor de um registrador na memória
INPUT	atribuição do valor da entrada em registrador

Fonte: O Autor

O armazenamento das quádruplas adaptadas foi feito usando a estrutura de dados ilustrada na Figura 41. Como o gerenciamento de memória e nomeação das linhas foi feito na geração do código intermediário, o código *assembly* possui apenas uma lista encadeada

de quádruplas.

Figura 41 – Estrutura do gerador de código assembly

```c
typedef struct Quadruple {
    char *command;
    char *first;
    char *second;
    char *third;
    struct Quadruple *next;
} Quadruple;

typedef struct {
    Quadruple *firstQuad;
    Quadruple *lastQuad;
} AssemblyCode;
```

Fonte: O Autor

A principal função do gerador de código *assembly* se chama *readIntermediateCode* e um trecho dela pode ser observado na Figura 42. Essa função faz a leitura das quádruplas do código intermediário, fazendo as modificações e adições necessárias para gerar quádruplas adaptadas à linguagem alvo, como pode ser observado no tratamento da quádrupla *SUM*.

Figura 42 – Função principal do gerador de código *assembly*

```c
void readIntermediateCode() {
    Quadruple *currentQuad = intermediateCode->firstQuad;
    int currentIntermediateCodeLine = 1;

    while (currentQuad) {
        char *command = currentQuad->command;
        char *first = currentQuad->first;
        char *second = currentQuad->second;
        char *third = currentQuad->third;

        int isRegister[4] = {0, strIsRegister(first), strIsRegister(second), strIsRegister(third)};
        if (strIsAlias(first)) first = intToStr(aliasToLineNumber(first));
        if (strIsAlias(second)) second = intToStr(aliasToLineNumber(second));
        if (strIsAlias(third)) third = intToStr(aliasToLineNumber(third));

        if (sameString(command, "SUM")) {
            if (isRegister[2] && isRegister[3]) insertNewAssemblyQuad("SUM0", first, second, third);
            else if (isRegister[2] && !isRegister[3]) insertNewAssemblyQuad("SUM1", first, second, third);
            else if (!isRegister[2] && isRegister[3]) insertNewAssemblyQuad("SUM1", first, third, second);
            else if (!isRegister[2] && !isRegister[3]) {
                insertNewAssemblyQuad("WRITE", first, second, "-");
                insertNewAssemblyQuad("SUM1", first, first, third);
                incrementAliasesAfter(currentIntermediateCodeLine);
            }
        }
        else if (sameString(command, "SUB")) {
```

Fonte: O Autor

3.2.4 Geração do código binário

A transformação das quádruplas do código *assembly* nas instruções do código binário é feita diretamente, de modo cada linha do código *assembly* corresponde a uma linha do código binário, ou seja, existe uma equivalência entre cada tipo de quádrupla e cada tipo de instrução. Para armazenar o código binário, foi utilizada a estrutura de dados ilustrada na Figura 43, onde é possível observar que o código binário é composto apenas de uma lista de instruções.

Figura 43 – Estrutura do gerador de código binário

```
typedef struct Instruction {
    char *instructionCode;
    struct Instruction *next;
} Instruction;

typedef struct {
    Instruction *firstInstruction;
    Instruction *lastInstruction;
} BinaryCode;
```

Fonte: O Autor

A principal função do gerador de código binário se chama *readAssemblyCode*, que, como mostra o trecho de código na Figura 44, recebe o código assembly para leitura das quádruplas e transforma todos os componentes de cada quádrupla em código binário, como acontece com o exemplo mostrado das instruções do tipo *SUM0* e *SUM1*.

Figura 44 – Função principal do gerados de código binário

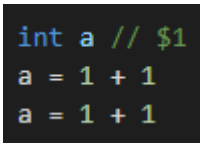
```
void readAssemblyCode() {
    Quadruple *currentQuad = assemblyCode->firstQuad;

    while (currentQuad) {
        char *command = currentQuad->command;
        char *first = currentQuad->first;
        char *second = currentQuad->second;
        char *third = currentQuad->third;

        int isRegister[4] = {0, strIsRegister(first), strIsRegister(second), strIsRegister(third)};

        if (strIsRegister(first)) first = registerNameToBinary(first);
        if (strIsRegister(second)) second = registerNameToBinary(second);
        if (strIsRegister(third)) third = registerNameToBinary(third);
        char *opcode = commandToOpCode(command);
        char *instruction = (char*) malloc(33 * sizeof(char));
        strcat(instruction, opcode);

        if (strStartsWith(command, "SUM")) {
            strcat(instruction, first);
            strcat(instruction, second);
            if (isRegister[3]) {
                strcat(instruction, second);
                strcat(instruction, decimalToExtendedBinary(0, sizeBetweenBits(19, 30)));
            }
            else strcat(instruction, decimalStringToExtendedBinary(third, sizeBetweenBits(14, 30)));
            if (sameString(command, "SUM0")) strcat(instruction, "0");
            else strcat(instruction, "1");
        }
        if (strStartsWith(command, "SUB")) {
```

Fonte: O Autor

3.2.5 Gerenciamento de memória

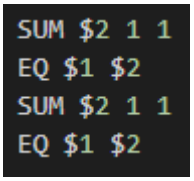
O uso de registradores do banco de registradores é feito de forma incremental conforme novas variáveis precisam ser alocadas, de modo que registradores usados por variáveis auxiliares são imediatamente liberados após o uso das mesmas. Por exemplo, o código escrito na linguagem fonte da Figura 45 gera o código intermediário da Figura 46, através desses códigos é possível observar que o registrador auxiliar *$2* é liberado logo após ter sido atribuído a $1, o que possibilita que ele seja reutilizado na segunda soma.

Figura 45 – Exemplo de gerenciamento de registradores - código fonte



Fonte: O Autor

Figura 46 – Exemplo de gerenciamento de registradores - código intermediário



Fonte: O Autor

A memória principal é gerenciada através das instruções *PUSH*, *POP*, *LOAD* e *STORE*. O principal objetivo do uso dessas instruções é liberar os registradores para serem usados na manipulação de dados do contexto atual. As instruções *LOAD* e *STORE* são usadas quando vetores são alocados e quando posições de vetores são lidas ou atribuídas. As instruções *PUSH* e *POP* são usadas quando argumentos precisam ser enviados para funções e recebidos por elas, quando resultados são retornados de funções e toda vez que ocorre a troca de escopo numa chamada de função, o que empilha todas as variáveis do escopo para serem resgatada quandos o escopo for acessado novamente. Além disso, sempre que se chega ao fim d eum escopo, todo espaço alocado por ele na memória é liberado. O controle das posições de memória é feito pelo atributo *freeMemoryAt* do código intermediário, que indica até que ponto a memória já foi utilizada.

4 Exemplos

Foram compilados três códigos fonte em linguagem C- para mostrar os resultados do compilador, mostrando o código fonte, o código intermediário, o código *assembly* e, finalmente, o código binário. Os dois primeiros exemplos, *Sort* e *GCD*, mostram o que é feito com todas as estruturas da linguagem C-, incluindo recursão e passagem de vetor como parâmetro de função.

4.1 Exemplo 1 - Sort

4.1.1 Sort - Código fonte

```
1  /* p */
2
3  int vet[ 10 ];
4
5  int minloc ( int a[], int low, int high )
6  {       int i; int x; int k;
7          k = low;
8          x = a[low];
9          i = low + 1;
10         while (i < high){
11                 if (a[i] < x){
12                         x = a[i];
13                         k = i;
14                 }
15                 i = i + 1;
16         }
17         return k;
18 }
19
20 void sort( int a[], int low, int high)
21 {       int i; int k;
22         i = low;
23         while (i < high-1){
24                 int t;
25                 k = minloc(a,i,high);
26                 t = a[k];
27                 a[k] = a[i];
28                 a[i] = t;
29                 i = i + 1;
30         }
31 }
32
33 void main(void)
34 {
35         int i;
36         i = 0;
37         while (i < 10){
38                 i = i + 1;
39         }
```

```
40          sort(vet,0,10);
41          i = 0;
42          while (i < 10){
43                  output(vet[i]);
44                  i = i + 1;
45          }
46  }
```

4.1.2 Sort - Código intermediário

```
1   WRITE $0 0 -
2   POP $3 - -
3   POP $2 - -
4   POP $1 - -
5   POP $4 - -
6   EQ $7 $2 -
7   SUM $8 $1 $2
8   LOAD $8 $8 -
9   EQ $6 $8 -
10  SUM $8 $2 1
11  EQ $5 $8 -
12  LESS $8 $5 $3
13  WRITE $9 #2 -
14  JUMPIF $8 $9 -
15  JUMP #5 - -
16  SUM $8 $1 $5
17  LOAD $8 $8 -
18  LESS $8 $8 $6
19  WRITE $9 #3 -
20  JUMPIF $8 $9 -
21  JUMP #4 - -
22  SUM $9 $1 $5
23  LOAD $9 $9 -
24  EQ $6 $9 -
25  EQ $7 $5 -
26  SUM $8 $5 1
27  EQ $5 $8 -
28  JUMP #1 - -
29  PUSH $7 - -
30  JUMP $4 - -
31  POP $3 - -
32  POP $2 - -
33  POP $1 - -
34  POP $4 - -
35  EQ $5 $2 -
36  SUB $7 $3 1
37  LESS $7 $5 $7
38  WRITE $8 #8 -
39  JUMPIF $7 $8 -
40  JUMP #10 - -
41  PUSH $1 - -
42  PUSH $2 - -
43  PUSH $3 - -
44  PUSH $4 - -
45  PUSH $5 - -
46  PUSH $6 - -
47  PUSH $7 - -
48  WRITE $8 #9 -
49  PUSH $8 - -
```

```
50  PUSH $1 - -
51  PUSH $5 - -
52  PUSH $3 - -
53  JUMP #0 - -
54  POP $7 - -
55  POP $6 - -
56  POP $5 - -
57  POP $4 - -
58  POP $3 - -
59  POP $2 - -
60  POP $1 - -
61  POP $8 - -
62  EQ $6 $8 -
63  SUM $8 $1 $6
64  LOAD $8 $8 -
65  EQ $7 $8 -
66  SUM $8 $1 $5
67  LOAD $8 $8 -
68  SUM $9 $1 $6
69  STORE $8 $9 -
70  SUM $8 $1 $5
71  STORE $7 $8 -
72  SUM $8 $5 1
73  EQ $5 $8 -
74  JUMP #7 - -
75  JUMP $4 - -
76  EQ $1 0 -
77  WRITE $2 10 -
78  LESS $2 $1 $2
79  WRITE $3 #13 -
80  JUMPIF $2 $3 -
81  JUMP #14 - -
82  SUM $2 $1 1
83  EQ $1 $2 -
84  JUMP #12 - -
85  PUSH $1 - -
86  WRITE $2 #15 -
87  PUSH $2 - -
88  PUSH $0 - -
89  WRITE $3 0 -
90  PUSH $3 - -
91  WRITE $3 10 -
92  PUSH $3 - -
93  JUMP #6 - -
94  POP $1 - -
95  EQ $1 0 -
96  WRITE $2 10 -
97  LESS $2 $1 $2
98  WRITE $3 #17 -
99  JUMPIF $2 $3 -
100 JUMP #18 - -
101 SUM $2 $0 $1
102 LOAD $2 $2 -
103 OUTPUT $2 - -
104 SUM $2 $1 1
105 EQ $1 $2 -
106 JUMP #16 - -
```

4.1.3 Sort - Código *assembly*

```
1  WRITE $0 0 -
2  STACK1 $3 - -
3  STACK1 $2 - -
4  STACK1 $1 - -
5  STACK1 $4 - -
6  COPY $7 $2 -
7  SUM0 $8 $1 $2
8  LOAD $8 $8 -
9  COPY $6 $8 -
10 SUM1 $8 $2 1
11 COPY $5 $8 -
12 LOGIC7 $8 $5 $3
13 WRITE $9 15 -
14 JUMP3 $9 $8 -
15 JUMP1 28 - -
16 SUM0 $8 $1 $5
17 LOAD $8 $8 -
18 LOGIC7 $8 $8 $6
19 WRITE $9 21 -
20 JUMP3 $9 $8 -
21 JUMP1 25 - -
22 SUM0 $9 $1 $5
23 LOAD $9 $9 -
24 COPY $6 $9 -
25 COPY $7 $5 -
26 SUM1 $8 $5 1
27 COPY $5 $8 -
28 JUMP1 11 - -
29 STACK0 $7 - -
30 JUMP2 $4 - -
31 STACK1 $3 - -
32 STACK1 $2 - -
33 STACK1 $1 - -
34 STACK1 $4 - -
35 COPY $5 $2 -
36 SUB1 $7 $3 1
37 LOGIC7 $7 $5 $7
38 WRITE $8 40 -
39 JUMP3 $8 $7 -
40 JUMP1 74 - -
41 STACK0 $1 - -
42 STACK0 $2 - -
43 STACK0 $3 - -
44 STACK0 $4 - -
45 STACK0 $5 - -
46 STACK0 $6 - -
47 STACK0 $7 - -
48 WRITE $8 53 -
49 STACK0 $8 - -
50 STACK0 $1 - -
51 STACK0 $5 - -
52 STACK0 $3 - -
53 JUMP1 1 - -
54 STACK1 $7 - -
55 STACK1 $6 - -
56 STACK1 $5 - -
57 STACK1 $4 - -
58 STACK1 $3 - -
```

```
STACK1 $2 - -
STACK1 $1 - -
STACK1 $8 - -
COPY $6 $8 -
SUM0 $8 $1 $6
LOAD $8 $8 -
COPY $7 $8 -
SUM0 $8 $1 $5
LOAD $8 $8 -
SUM0 $9 $1 $6
STORE $8 $9 -
SUM0 $8 $1 $5
STORE $7 $8 -
SUM1 $8 $5 1
COPY $5 $8 -
JUMP1 35 - -
JUMP2 $4 - -
WRITE $1 0 -
WRITE $2 10 -
LOGIC7 $2 $1 $2
WRITE $3 81 -
JUMP3 $3 $2 -
JUMP1 84 - -
SUM1 $2 $1 1
COPY $1 $2 -
JUMP1 76 - -
STACK0 $1 - -
WRITE $2 93 -
STACK0 $2 - -
STACK0 $0 - -
WRITE $3 0 -
STACK0 $3 - -
WRITE $3 10 -
STACK0 $3 - -
JUMP1 30 - -
STACK1 $1 - -
WRITE $1 0 -
WRITE $2 10 -
LOGIC7 $2 $1 $2
WRITE $3 100 -
JUMP3 $3 $2 -
JUMP1 106 - -
SUM0 $2 $0 $1
LOAD $2 $2 -
COPY $31 $2 -
SUM1 $2 $1 1
COPY $1 $2 -
JUMP1 95 - -
```

4.1.4 Sort - Código binário

```
10000000000000000000000000000000
01110001100000000000000000000001
01110001000000000000000000000001
01110000100000000000000000000001
01110010000000000000000000000001
10010011100010000000000000000000
00000100000001000010000000000000
10100100001000000000000000000000
```

```
10010011001000000000000000000000
00000100000010000000000000000011
10010010101000000000000000000000
01010100000101000110000000000111
10000100100000000000000000001111
01100100101000000000000000000011
01100000000000000000000000110001
00000100000001000010000000000000
10100100001000000000000000000000
01010100001000001100000000000111
10000100100000000000000000010101
01100100101000000000000000000011
01100000000000000000000001100101
00000100100001000010000000000000
10100100101001000000000000000000
10010011001001000000000000000000
10010011100101000000000000000000
00000100000101000000000000000011
10010010101000000000000000000000
01100000000000000000000000101101
01110011100000000000000000000000
01100010000000000000000000000010
01110001100000000000000000000001
01110001000000000000000000000001
01110000100000000000000000000001
01110010000000000000000000000001
10010010100010000000000000000000
00010011100011000000000000000011
01010011100101001110000000000111
10000100000000000000000000101000
01100100000111000000000000000011
01100000000000000000000100101001
01110000100000000000000000000000
01110001000000000000000000000000
01110001100000000000000000000000
01110010000000000000000000000000
01110010100000000000000000000000
01110011000000000000000000000000
01110011100000000000000000000000
10000100000000000000000000110101
01110100000000000000000000000000
01110000100000000000000000000000
01110010100000000000000000000000
01110001100000000000000000000000
01100000000000000000000000000101
01110011100000000000000000000001
01110011000000000000000000000001
01110010100000000000000000000001
01110010000000000000000000000001
01110001100000000000000000000001
01110001000000000000000000000001
01110000100000000000000000000001
01110100000000000000000000000001
10010011001000000000000000000000
00000100000001000010000000000000
10100100001000000000000000000000
10010011101000000000000000000000
00000100000001000010000000000000
10100100001000000000000000000000
00000100100001000010000000000000
```

```
10110100001001000000000000000000
00000100000001000010000000000000
10110011101000000000000000000000
00000100000101000000000000000011
10010010101000000000000000000000
01100000000000000000000010001101
01100010000000000000000000000010
10000000100000000000000000000000
10000001000000000000000000001010
01010001000001000100000000000111
10000001100000000000000001010001
01100001100010000000000000000011
01100000000000000000000101010001
00000001000001000000000000000011
10010000100010000000000000000000
01100000000000000000000100110001
01110000100000000000000000000000
10000001000000000000000001011101
01110001000000000000000000000000
01110000000000000000000000000000
10000001100000000000000000000000
01110001100000000000000000000000
10000001100000000000000000001010
01110001100000000000000000000000
01100000000000000000000001111001
01110000100000000000000000000001
10000000100000000000000000000000
10000001000000000000000000001010
01010001000001000100000000000111
10000001100000000000000001100100
01100001100010000000000000000011
01100000000000000000000110101001
00000001000000000000000000000000
10100001000010000000000000000000
10011111111010100000000000000000
00000001000001000000000000000011
10010000100010000000000000000000
01100000000000000000000101111101
```

4.2 Exemplo 2 - GCD

4.2.1 GCD - Código fonte

```
1  int gcd(int u, int v)
2  {
3      if (v == 0)
4          return u;
5      else
6          return gcd(v, u - u / v * v);
7  }
8  void main(void)
9  {
10     int x;
11     int y;
12     x = input();
13     y = input();
14     output(gcd(x, y));
15 }
```

4.2.2 GCD - Código intermediário

```
POP $1 - -
POP $0 - -
POP $2 - -
WRITE $3 0 -
EQ_EQ $3 $1 $3
WRITE $4 #2 -
JUMPIF $3 $4 -
PUSH $0 - -
PUSH $1 - -
PUSH $2 - -
PUSH $2 - -
WRITE $4 #1 -
PUSH $4 - -
PUSH $1 - -
DIV $5 $0 $1
MULT $5 $5 $1
SUB $5 $0 $5
PUSH $5 - -
JUMP #0 - -
POP $2 - -
POP $2 - -
POP $1 - -
POP $0 - -
POP $4 - -
PUSH $4 - -
JUMP #3 - -
PUSH $0 - -
JUMP $2 - -
INPUT $2 - -
EQ $0 $2 -
INPUT $2 - -
EQ $1 $2 -
PUSH $0 - -
PUSH $1 - -
WRITE $2 #5 -
PUSH $2 - -
PUSH $0 - -
PUSH $1 - -
JUMP #0 - -
POP $1 - -
POP $0 - -
POP $2 - -
OUTPUT $2 - -
```

4.2.3 GCD - Código *assembly*

```
STACK1 $1 - -
STACK1 $0 - -
STACK1 $2 - -
WRITE $3 0 -
LOGIC4 $3 $1 $3
WRITE $4 26 -
JUMP3 $4 $3 -
STACK0 $0 - -
STACK0 $1 - -
STACK0 $2 - -
STACK0 $2 - -
WRITE $4 19 -
```

```
STACK0 $4 - -
STACK0 $1 - -
DIV0 $0 $1 -
COPY $5 $29 -
MULT0 $5 $1 -
COPY $5 $28 -
SUB0 $5 $0 $5
STACK0 $5 - -
JUMP1 0 - -
STACK1 $2 - -
STACK1 $2 - -
STACK1 $1 - -
STACK1 $0 - -
STACK1 $4 - -
STACK0 $4 - -
JUMP1 29 - -
STACK0 $0 - -
JUMP2 $2 - -
INPUT $2 - -
COPY $0 $2 -
INPUT $2 - -
COPY $1 $2 -
STACK0 $0 - -
STACK0 $1 - -
WRITE $2 41 -
STACK0 $2 - -
STACK0 $0 - -
STACK0 $1 - -
JUMP1 0 - -
STACK1 $1 - -
STACK1 $0 - -
STACK1 $2 - -
COPY $31 $2 -
```

4.2.4 GCD - Código binário

```
01110000100000000000000000000001
01110000000000000000000000000001
01110001000000000000000000000001
10000001100000000000000000000000
01010001100001000110000000000100
10000010000000000000000000011010
01100010000011000000000000000011
01110000000000000000000000000000
01110000100000000000000000000000
01110001000000000000000000000000
01110001000000000000000000000000
10000010000000000000000000010011
01110010000000000000000000000000
01110000100000000000000000000000
00110000000001000000000000000000
10010010111101000000000000000000
00100010100001000000000000000000
10010010111100000000000000000000
00010010100000000000000000000000
01110010100000000000000000000000
01100000000000000000000000000001
01110001000000000000000000000001
01110001000000000000000000000001
```

```
24 01110000100000000000000000000001
25 01110000000000000000000000000001
26 01110010000000000000000000000001
27 01110010000000000000000000000000
28 01100000000000000000000001110101
29 01110000000000000000000000000000
30 01100001000000000000000000000010
31 11000001000000000000000000000000
32 10010000000010000000000000000000
33 11000001000000000000000000000000
34 10010000100010000000000000000000
35 01110000000000000000000000000000
36 01110000100000000000000000000000
37 10000001000000000000000000101001
38 01110001000000000000000000000000
39 01110000000000000000000000000000
40 01110000100000000000000000000000
41 01100000000000000000000000000001
42 01110000100000000000000000000001
43 01110000000000000000000000000001
44 01110001000000000000000000000001
45 10011111110101000000000000000000
```

4.3 Exemplo 3 - Sintético para analisar relação entre os códigos

4.3.1 Sintético - Código fonte

```
1 void main() {
2     int a;
3     a = 1 + 1;
4 }
```

4.3.2 Sintético - Código intermediário

No código intermediário gerado é possível observar que a partir da linha 3 do código fonte foram geradas duas quádruplas, uma para atribuir o valor da soma a uma variável auxiliar (*$1*) e outra para atribuir o resultado da soma para a variável *a* (*$0*).

```
1 SUM $1 1 1
2 EQ $0 $1 -
```

4.3.3 Sintético - Código *assembly*

No código *assembly* gerado é possível observar que, além da modificação das duas quádruplas existentes, houve a adição de uma quádrupla para auxiliar a operação de soma. Isso acontece porque a arquitetura da linguagem alvo não permite uma operação de soma usando 2 números como operandos, então um deles foi atribuído a um registrador auxiliar antes de a soma ser realizada.

```
1 WRITE $1 1 -
2 SUM1 $1 $1 1
3 COPY $0 $1 -
```

4.3.4 Sintético - Código binário

No código binário gerado é possível observar que é feita uma tradução direta do código *assembly* usando as instruções equivalentes às quádruplas.

```
1 10000000100000000000000000000001
2 00000000100001000000000000000011
3 10010000000001000000000000000000
```

5 Considerações Finais

O objetivo do curso é desenvolver um sistema computacional completo que seja capaz de resolver problemas computacionais básicos. A primeira etapa foi a definição dos conceitos básicos de *design* e estrutura do processador, e assim foram implementados todos os componentes deste que pode ser chamado de alicerce de todo o sistema. Nessa etapa, foi desenvolvido o compilador que traduz códigos em linguagem C- para a linguagem de máquina do processador, o que viabiliza a criação de algoritmos para o sistema em uma linguagem com alto nível de abstração. A maior dificuldade encontrada no desenvolvimento do projeto foi a implementação do gerador de código intermediário, pois este acabou ficando com a maior parte do processo de tradução, tendo responsabilidades como gerenciamento de memória e nomeação de linhas, enquanto o gerador de código *assembly* realiza apenas algumas correções para adequar as quádruplas ao conjunto de instruções do processador.

Referências

1 LOUDEN, K. C. *Compiladores - Princípios e Práticas*. [S.l.: s.n.], 2004. Citado na página 41.