DION LENON PREDIGER FEIL

IMPLEMETAÇÃO EM HARDWARE DA MÁQUINA VIRTUAL LUA

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Engenheiro Eletricista, do curso de Engenharia Elétrica da Universidade Federal do Pampa – Campus Alegrete

Orientador: Alessandro G. Girardi

ALEGRETE
2013

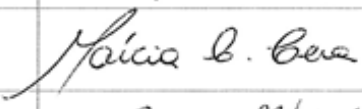
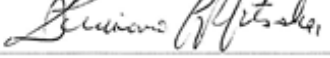
ERRATA

Folha	Linha	Onde se lê	Leia-se

Autoria: Dion Lenon Prediger Feil

Título: Implementação em Hardware da Máquina Virtual Lua

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Engenheiro Eletricista, do curso de Engenharia Elétrica da Universidade Federal do Pampa – Campus Alegrete.

Os componentes da banca de avaliação, abaixo listados, consideram este trabalho aprovado.				
	Nome	**Titulação**	**Assinatura**	**Instituição**
1	Alessandro Gonçalves Girardi	Dr.		Unipampa
2	Márcia Cristina Cera	Dra.		Unipampa
3	Luciano Lopes Pfitscher	Me.		Unipampa

Data da aprovação: 08 de março de 2013.

Dedico este trabalho à minha família e amigos pela força, apoio, incentivo, amor e carinho dedicados a mim durante todo o processo de execução deste trabalho, principalmente nos momentos mais difíceis dessa caminhada.

AGRADECIMENTOS

Agradeço em primeiro lugar a DEUS, por ter me dado força e coragem para enfrentar e superar os obstáculos encontrados. À minha família, que sempre acreditou em mim e me incentivou na busca por meus objetivos, mostrando que sou capaz de alcança-los. Aos meus amigos pelas palavras de carinho e apoio nas horas mais difíceis e pelos momentos de descontração. Ao orientador Alessandro Girardi, pela confiança, direção, amizade e paciência.

O ato da conquista de um grande sonho reflete o sucesso nas lutas diárias para alcançá-lo.

Kléber Novartes

RESUMO

O constante avanço dos processos de fabricação de circuitos integrados tem possibilitado o desenvolvimento de sistemas computacionais cada vez mais complexos. Atualmente vem se destacando os sistemas embarcados que ganham espaço e mercado em produtos de eletrônica de consumo. Este trabalho apresenta a proposta de uma arquitetura para um *Processador Dedicado* para a execução de um conjunto de instruções de forma nativa da linguagem Lua 5.1, sem a necessidade de uso da máquina virtual Lua. O Processador visa aplicações onde o requisito de área é fundamental, como sistemas embarcados ou sistemas integrados em um único circuito integrado - *System on a Chip* (SoC), e onde não seja necessário o uso de processadores potentes, ou seja, cujas aplicações são simples. Inicialmente, após uma revisão bibliográfica, onde são apresentadas características dos sistemas embarcados e da linguagem Lua, além de mostrar as instruções Lua e seus tipos formatos, um Programa Teste é apresentado a fim de se definir um conjunto de instruções a serem implementadas no Processador Dedicado. Após é feito a descrição organizacional do processador, onde são apresentados os elementos ou conjunto de elementos necessários para a implementação do conjunto de instruções definidas a partir do Programa Teste. Na sequência são apresentados os caminhos de dados para cada instrução definida anteriormente. Os resultados apresentados são resultados da síntese lógica do Processador. O software utilizado para a síntese lógica foi o Quartus II 9.1 Web Edition da Altera Corporation, e a síntese foi realizada para o dispositivo FPGA Cyclone II EP2C35F672C6. Por fim, no apêndice consta o código gerado a pós a compilação do código do Programa Teste, o qual foi descrito em linguagem Lua.

Palavras-chave: Máquina Virtual Lua, Instruções Lua, Sistemas Embarcados, Processador Dedicado, SystemVerilog, Hardware.

ABSTRACT

The constant advancement of the manufacturing processes of integrated circuits has enabled the development of computational systems increasingly complex. Currently has been highlighting the embedded systems that gain space and market in products of electronics consumer. This work presents the proposal an architecture for a *Dedicated Processor* for executing a set of instructions of form native of the language Lua 5.1, without the need use Lua virtual machine. The processor aims applications where the requirement of area is fundamental, as embedded systems or systems integrated on a single chip - System on a Chip (SoC), and where it is not necessary to use powerful processors, in other words, whose applications are simple. Initially, after a literature review, where is presented characteristics of embedded systems and the Lua language, besides showing the formats and types of Lua instructions, a test program is presented in order to define a set of instructions to be implemented in the dedicated processor. After is made the description organizational of processor, where is presents the element or group of elements required to implement the instruction set defined from the test program. Following are presented the data paths for each instruction defined previously. The results shown are the results of logic synthesis processor. The software used for logic synthesis was the Quartus II 9.1 Web Edition from Altera Corporation, and the synthesis was performed for the device Cyclone II FPGA EP2C35F672C6. Finally, in appendix is showed the code generated code after compiling test program, that was been described in language Lua.

Keywords: Lua Virtual Machine, Lua Instructions, Embedded Systems, Dedicated Processor, SystemVerilog, Hardware.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O constante avanço dos processos de fabricação de circuitos integrados tem possibilitado o desenvolvimento de sistemas computacionais cada vez mais complexos. Atualmente vem se destacando os sistemas embarcados que ganham espaço e mercado em produtos de eletrônica de consumo (SOCAL, 2003).

A maioria dos processadores fabricados atualmente são utilizados em sistemas embarcados. O uso de processadores em sistemas embarcados teve início antes do surgimento dos computadores pessoais, onde esses sistemas eram projetados para realizar funções de controle, existindo até hoje aplicações com essa finalidade. Porém, existem outras aplicações que demandam grande capacidade de processamento, como o processamento de sinais.

Na atualidade, é inevitável o uso de sistemas embarcados, pois estamos cercados por eles. Máquinas de lavar, eletrodomésticos, televisores, automóveis, são alguns exemplos de sistemas embarcados que usamos em nosso cotidiano, pois possuem algum tipo informação processada.

Esta presença cada vez mais marcante do uso de sistemas embarcados é devido ao baixo custo tecnológico, permitindo o aumento da capacidade do hardware e viabilizando a implementação de aplicações mais complexas. O conhecimento das necessidades humanas e o crescimento do mercado tem impulsionado o surgimento de novas aplicações (SANTOS, 2006).

Devido à necessidade dos usuários, que aumenta a cada dia, de softwares manuteníveis e portáteis, ultimamente tem-se buscado o desenvolvimento de softwares com tais características, mas que apresentem também uma rápida prototipação e a possibilidade de ter seu teste integrado. Algumas linguagens de programação tradicionais como C++ e Java podem ser utilizadas para atender tais propósitos, mas deixam a desejar quando comparadas com linguagens de mais alto nível como Scheme, Python ou Lua (SOCAL, 2003).

Neste cenário, a linguagem Lua destaca-se dentre outras pela simplicidade em que apresenta as características de rápida prototipação e possibilidade de teste integrado, além de ser portável, leve, compacta, projetada para expandir aplicações em geral e para ser usada como linguagem extensível, ou seja, unir partes de um programa feitas em mais de uma linguagem. Outra característica muito importante, apresentada pela linguagem é de ser um software livre, podendo ser usado para quaisquer propósitos, tanto acadêmico quanto comercial. Devido à sua eficiência, clareza e facilidade de aprendizado, passou a ser usada em diversos ramos da programação, como no desenvolvimento de jogos, controle de robôs, processamento de texto, etc (YANAGISAWA, 1993).

1.1. Motivação

Assim como no desenvolvimento de software, a complexidade do hardware impõe a reutilização de componentes tanto para lidar com esta complexidade como também para diminuir o tempo de projeto dos sistemas digitais (LEITE, 2006). Dessa maneira, tem-se como motivação deste trabalho o desenvolvimento de um *processador dedicado*, o qual poderá ser inserido em sistemas onde o requisito de área é fundamental, como sistemas embarcados ou sistemas integrados em um único circuito integrado - *System on a Chip* (SoC) (FIGUEIRÓ, 2011).

1.2. Objetivos

1.2.1. Objetivo Geral

O objetivo geral deste trabalho é o desenvolvimento de um *Processador Dedicado* para a execução de um conjunto de instruções de forma nativa da linguagem Lua 5.1, sem a necessidade de uso da máquina virtual Lua. O sistema atual de compilação de um código em Lua consiste basicamente de um compilador onde são gerados os *bytecodes*, que em seguida passam pelo interpretador (máquina virtual) para atingir o sistema alvo. Enquanto que o sistema proposto consiste de um processador, ao invés de uma máquina virtual, para fazer a execução dos *bytecodes* gerados pelo compilador.

1.2.2. Objetivo Específico

Visando a implementação de um *Processador Dedicado* em hardware da Máquina Virtual Lua com uso reduzido de área em FPGA e baixo consumo de potência, o objetivo específico é a implementação em hardware de um conjunto de instruções Lua definidas a partir de um

Programa Teste, o qual consiste na operação de multiplicação de duas matrizes iguais de dimensões 3x3. As instruções obtidas, e que serão implementadas em hardware através do processador dedicado, a partir do programa teste são apresentadas na TABELA 3.

Logo, a arquitetura proposta para o processador deverá executar 11 instruções de um total de 38 instruções reconhecidas e executas atualmente pela máquina virtual Lua 5.1. As 11 instruções que serão implementadas na arquitetura, serão descritas na seção 3.2.

1.3.Metodologia

Como o objetivo é o desenvolvimento de um processador dedicado destinado à execução das instruções da linguagem Lua, a metodologia utilizada foi a utilização de um programa teste capaz de explorar um conjunto destas instruções. Após definido o conjunto de instruções, implementou-se separadamente o caminho de dados para cada uma destas instruções. Na sequência, com a caminho de dados necessário para a execução de cada instrução definido, foi desenvolvida a organização do processador dedicado, a qual é apresentada na FIGURA 1. Com a organização da arquitetura do processador definida, partiu-se para a descrição em linguagem de hardware dos elementos de hardware necessários para a execução das instruções e a união destes elementos conforme a organização do processador mostrado na FIGURA 1. Por fim foram obtidos os resultados de síntese lógica do processador, utilizando-se do software Quartus II 9.1 Web Edition da Altera Corporation.

1.4.Estrutura do trabalho

Este trabalho está estruturado da seguinte forma:

No Capítulo 2 serão vistos conceitos importantes sobre processadores e sistemas embarcados, linguagem Lua, e a máquina virtual Lua. No Capitulo 3 será visto algumas instruções da Máquina Virtual Lua necessárias para execução de um *Programa Teste*, assim como, os elementos de hardware necessários para a implementação destas instruções. Por fim, o Capítulo 4 mostra os resultados obtidos de síntese lógica da arquitetura proposta para o *Processador Dedicado*.

2 FUNDAMENTAÇÃO TEÓRICA

2.1.Processadores e Sistemas Embarcados

De um modo geral, os sistemas embarcados são sistemas que manipulam dados dentro de outros sistemas ou produtos maiores. Porém, o termo *sistema embarcado* não possui uma definição universal e única, então abaixo são listadas algumas definições para este:

- Sistemas embarcados são sistemas onde hardware e software normalmente são integrados e seu projeto visa o desempenho de uma função especifica (SANTOS, 2006);
- Sistemas embarcados são sistemas de processamento de informações que estão embutidos em sistemas maiores e que normalmente não são visíveis ao usuário (SANTOS, 2006);
- Diferentemente de um computador de uso comum, que pode executar diversos programas e funções, os sistemas embarcados são dispositivos que se limitam a executar bem uma única tarefa ou uma gama de tarefas. Eles se fundem em nosso cotidiano, sem percebermos que eles existem. Em geral, eles são formados pelos mesmos componentes de um computador: processador, memória, interface e demais componentes (OLEQUES, 2009).

A importância dos sistemas embarcados pode ir desde um simples sistema para brinquedos, até uma máquina com centenas de processadores, destinada a controlar o tráfego aéreo.

Devido aos sistemas embarcados, os processadores simples passaram a ser os mais produzidos, não só pelo baixo custo, mas também por não ser necessário um processador potente para aplicações mais simples, como um controle remoto (OLEQUES, 2009).

2.1.1. Características de Sistemas Embarcados

A característica principal e comum a todos os sistemas embarcados é que estes são sistemas que manipulam dados dentro de outros sistemas ou produtos maiores. Porém outras características também estão presentes nos sistemas embarcados:

- Os sistemas embarcados interagem com o meio ambiente através da coleta de dados de sensores e modificam o mesmo utilizando atuadores;
- Devido estes sistemas realizarem funções críticas onde não possa haver falhas, os sistemas embarcados devem ser muito confiáveis. Para que o sistema seja confiante, é importante apresentar tais características: estabilidade, recuperação, disponibilidade e segurança;
- Muitos sistemas embarcados são móveis e alimentados por baterias. Assim, estes devem ser projetados para consumir o mínimo de energia e ser o mais leve possível;
- Para implementar uma função, o uso de recursos de hardware devem ser limitados ao que é realmente necessário, reduzindo o espaço ocupado, o consumo de energia e o custo do produto final (SANTOS, 2006).

2.2. Linguagem Lua

Lua é uma linguagem que foi projetada e implementada pelo Tecgraf, Grupo de Computação Gráfica da PUC-Rio. A primeira versão (1.0) é de Julho de 1993. A última versão é a versão 5.1 (CELES, 2008).

Lua surgiu a partir de outras duas linguagens, DEL (*Data-Entry Language*) e SOL (*Simple Object Language*), as quais foram desenvolvidas com a preocupação de adicionar flexibilidade a dois diferentes projetos (ambos programas utilizados para aplicações de engenharia na Petrobrás). Porém, as duas linguagens possuíam problemas. Com isso, Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes decidiram, então, que precisavam de uma linguagem que fosse completa, de configuração genérica, facilmente acoplável, a mais simples possível e que não tivesse uma sintaxe intimidante. Logo, como estavam deixando de utilizar a linguagem SOL, surgiu a ideia de colocar o nome da nova linguagem, que ali surgia, de Lua (YANAGISAWA, 1993).

A linguagem Lua é uma linguagem poderosa, leve e projetada para expandir aplicações, ou seja, para ser acoplada a programas maiores que precisam ler e executar programas descritos pelos usuários (CELES, 2008).

Conforme (LUA, 2008), Lua destaca-se pela sua simplicidade, portabilidade e rapidez, além de ser uma linguagem pequena e de propósito geral, ou seja, pode ser utilizada para escrever pequenos scripts, assim como sistemas complexos.

Ultimamente, Lua tem sido muito utilizada na indústria de jogos. Alguns jogos conhecidos como o *The Sims* e *Sim City*, são exemplos de jogos que utilizam a linguagem Lua. Além disso, Lua também esta sendo utilizada em softwares como *Adobe Photoshop Lightroom* e o middleware *Ginga* do sistema brasileiro de TV digital, além de ser altamente utilizada no desenvolvimento de aplicações web.

2.2.1. Características da Linguagem Lua

Desde a criação da linguagem, os objetivos nem sempre foram os mesmos, mas mesmo assim eles fizeram Lua obter qualidades interessantes se comparadas com outras linguagens (MACHADO, 2009). Tais características são descritas a seguir:

- **Simplicidade:** Esta característica foi obtida construindo uma linguagem com sintaxe simples, com poucas construções de linguagem e número reduzido de tipos de dados.
- **Eficiência:** Desde o início do projeto da linguagem Lua, buscou-se uma linguagem que fosse rápida, pois Lua trabalha com grandes quantidades de dados. Para isto, é necessário compilar e executar o código rapidamente. Logo, a solução encontrada foi criar um compilador rápido e inteligente, assim como uma máquina virtual também rápida. Tal máquina virtual foi projetada de forma mais eficiente que as máquinas habituais.
- **Portabilidade:** Devido a portabilidade ser um requisito importante e relevante, sempre se buscou atender esta característica. Para isto, foi necessário que o núcleo da linguagem fosse desenvolvido de uma forma que pudesse ser compilado em qualquer lugar e que executasse programas em Lua que suportasse o interpretador Lua.
- **Programas embarcáveis com baixo custo:** Devido a necessidade de se acoplar programas em Lua com programas maiores, criou-se uma API (*Aplication Pro-*

graming Interface - Interface de Programação de Aplicativos) de C simples e poderosa. Porém, o núcleo da linguagem precisou ser compacto, para que não fosse caro para os desenvolvedores do projeto embarcar programas em Lua (MACHADO, 2009).

2.2.2. Máquina Virtual Lua

Ao contrário das usuais e mais comuns máquinas virtuais que são baseadas em pilhas, a Máquina Virtual Lua (*Lua Virtual Machine*, ou LMV) é baseada em um conjunto de registradores. Uma vantagem com relação à linguagem intermediaria é que ela não precisa de instruções para pôr ou tirar elementos em uma pilha. Por exemplo, numa operação de adição de três endereços, é necessário apenas informar o número dos dois registradores que serão somados e o número do registrador que receberá o resultado da operação. Consequentemente, este processo leva a uma redução no número de instruções necessárias (MACHADO, 2009).

Segundo (MAN, 2006), atualmente, a máquina virtual Lua 5.1 reconhece e executa um total de 38 instruções. Tais instruções estão listadas na TABELA 1.

TABELA 1 - Instruções Lua e Opcodes.

Opcode	Instrução
0	MOVE
1	LOADK
2	LOADBOOL
3	LOADNIL
4	GETUPVAL
5	GETGLOBAL
6	GETTABLE
7	SETGLOBAL
8	SETUPVAL
9	SETTABLE
10	NEWTABLE
11	SELF
12	ADD
13	SUB
14	MUL
15	DIV

TABELA 1 - Instruções Lua e Opcodes.

Opcode	Instrução
16	MOD
17	POW
18	UNM
19	NOT
20	LEN
21	CONCAT
22	JMP
23	EQ
24	LT
25	LE
26	TEST
27	TESTSET
28	CALL
29	TAILCALL
30	RETURN
31	FORLOOP
32	FORPREP
33	TFORLOOP
34	SETLIST
35	CLOSE
36	CLOSURE
37	VARARG

As instruções Lua possuem um tamanho fixo de 32 bits. Elas são de três tipos, e são nomeadas como: iABC, iABx, iAsBx. A representação de cada uma pode ser visualizada na TABELA 2 (MAN, 2006).

TABELA 2 - Tipo e formato das instruções Lua.

Tipo	Formato			
iABC	B:9	C:9	A:8	Opcode:6
iABx	Bx:18		A:8	Opcode:6
iAsBx	sBx:18		A:8	Opcode:6

3 CONCEPÇÃO DA ARQUITETURA

Visando a implementação de um *Processador Dedicado* em hardware da Máquina Virtual Lua com uso reduzido de área em FPGA e baixo consumo de potência, nesta seção serão vistos o conjunto de instruções do processador dedicado e os elementos de hardware necessários para a implementação de algumas instruções da Máquina Virtual Lua necessárias para execução do *Programa Teste*. Segundo (FIGUEIRÓ, 2011), a reusabilidade de elementos funcionais na arquitetura permitirá que algumas operações utilizem o mesmo caminho de dados, resultando em uma maior economia de hardware.

O ALGORITMO 1 mostra a descrição em linguagem Lua da operação de multiplicação das duas matrizes.

```
1  -- Programa que multiplica duas matrizes iguais 3x3
2
3  local n = 1
4
5  local size = 3
6
7  local rows = size
8  local cols = size
9  local count = 1
10  local mx = {}
11
12  -- preenche a matriz com números em sequência
13
14  for i=0,(rows - 1) do
15          local row = {}
16          for j=0,(cols - 1) do
17          row[j] = count
18          count = count + 1
```

```
19            end
20            mx[i] = row
21  end
22
23  -- matrizes m1 e m2 são iguais
24
25   local m1 = mx
26   local m2 = mx
27
28  local m3 = {}
29
30  -- realiza a multiplicação das matrizes m1 e m2
31
32  for i=0,(rows-1) do
33     m3[i] = {}
34     for j=0,(cols-1) do
35        local rowj = 0
36        for k=0,(cols-1) do
37           rowj = rowj + m1[i][k] * m2[k][j]
38        end
39        m3[i][j] = rowj
40     end
41  end
```

ALGORITMO 1 - Código em linguagem Lua da multiplicação de duas matrizes iguais.

Após a compilação do código descrito em linguagem Lua, que está mostrado no ALGORTIMO 1, gerou-se o código com as instruções Lua necessárias para a execução da multiplicação das matrizes. Logo, as instruções necessárias são apresentadas na TABELA 3, as quais serão implementadas em hardware através do *Processador Dedicado.* A compilação do código em linguagem Lua foi feito através do software ChunkSpy. Este é uma ferramenta que desmonta um código descrito em linguagem Lua em uma listagem detalhada.

O código gerado com a compilação do código descrito em linguagem Lua é apresentado no APÊNDICE A.

TABELA 3 - Instruções Lua necessárias para a execução do Programa Teste.

Opcode	Instrução
0	MOVE
1	LOADK

TABELA 3 - Instruções Lua necessárias para a execução do Programa Teste.

Opcode	Instrução
6	GETTABLE
9	SETTABLE
10	NEWTABLE
12	ADD
13	SUB
14	MUL
30	RETURN
31	FORLOOP
32	FORPREP

Devido se tratar de um projeto digital, a descrição será realizada utilizando-se linguagens de descrição de hardware, comumente conhecidas como HDL's. Esta descrição utilizando HDL's será realizada não só como métodos para a descrição do comportamento do sistema, mas também como uma descrição estrutural dos componentes que o compõem.

A HDL *SystemVerilog* (SUTHERLAND, 2004), utilizada neste projeto, é uma nova linguagem que evoluiu da HDL Verilog e é utilizada em projetos orientados a hardware (LEITE, 2006). Assim, todos os elementos funcionais do processador dedicado foram descritos com esta linguagem, sendo que alguns códigos serão mostrados para a exemplificação do comportamento do bloco.

A FIGURA 1 apresenta a organização proposta para o processador dedicado, que será responsável pela execução das instruções MOVE, LOADK, ADD, SUB, MUL, FORPREP, FORLOOP, NEWTABLE, GETTABLE e SETTABLE.

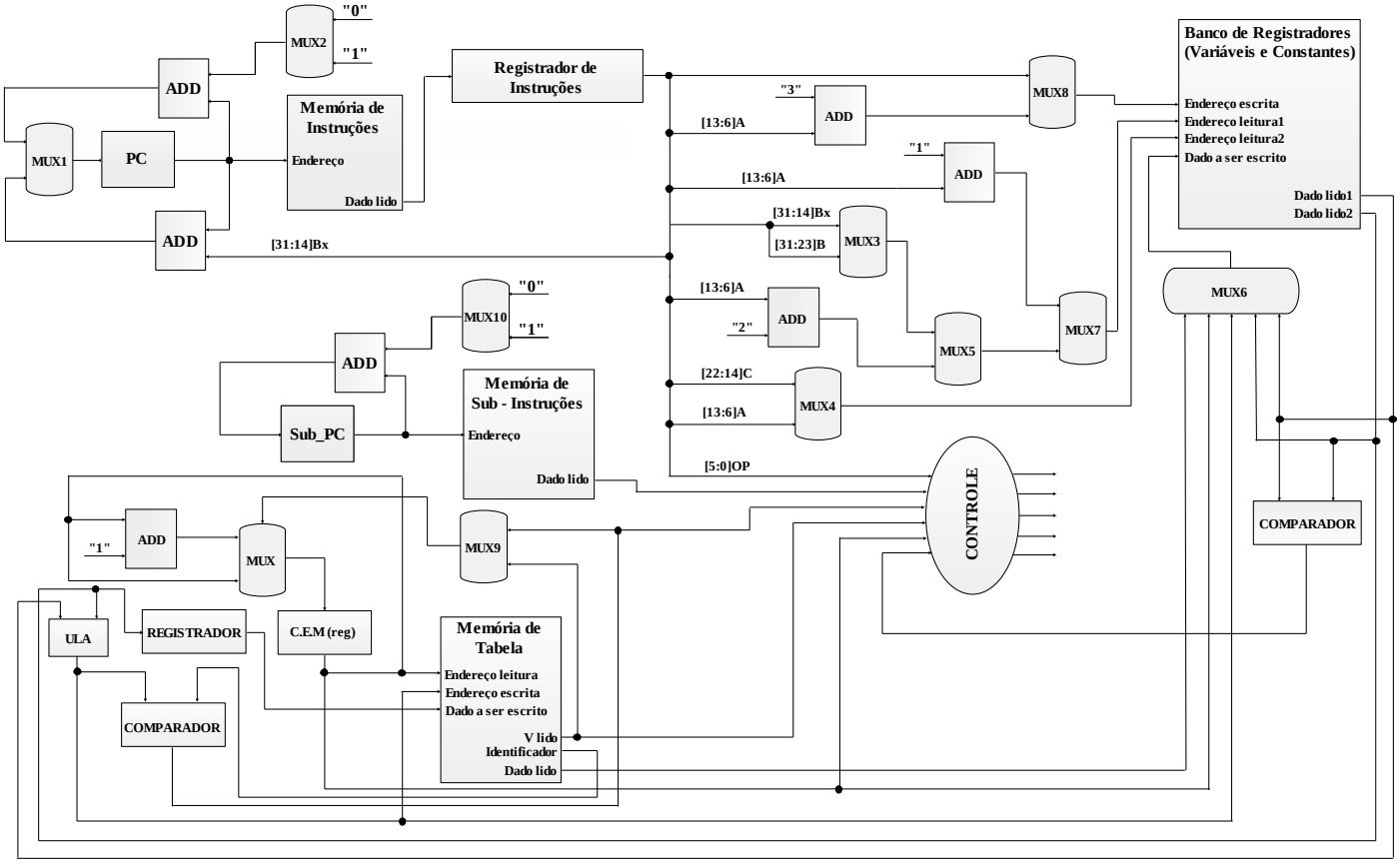


FIGURA 1 – Organização proposta final para a arquitetura do processador.

De maneira a compreender o funcionamento da organização, optou-se por mostrá-la em partes, cabendo à parte de descrição da organização do Processador detalhar cada elemento, funcionamento e papel dentro da arquitetura, e a parte de descrição do fluxo de dados, onde estará sendo realizada uma análise detalhada do caminho de dados de cada instrução.

3.1. Descrição organizacional do Processador

Nesta seção será visto detalhadamente cada elemento ou conjunto de elementos, de forma a especificar não só os sinais de entrada e saída, como também o seu comportamento em linguagem de descrição de hardware – SystemVerilog. Os elementos estarão divididos em Registradores (PC, Registrador de Instrução, Contador de Endereço de Memória, Banco de Registradores), Memórias (Memória de Instruções, Memória de Tabela), Comparador, Somadores, Subtratores, Multiplexadores, Unidade Lógica e Aritmética (ULA), Unidade de Controle.

3.1.1. Registradores

Um registrador é um conjunto de células de memória sensíveis à transição do sinal de clock, agrupadas de forma a armazenar uma dada palavra binária (UYEMURA, 2002). Neste

caso, os registradores são utilizados para formar outros blocos operativos, como por exemplo, PC (*Program Counter*), Banco de Registradores, Registradores de Instruções, Contadores de Endereço de Memória.

Esta seção apresentará os tipos de registradores usados na implementação da arquitetura do *Processador Dedicado*.

3.1.1.1.PC (*Program Counter*)

O *PC – Program Counter (Contador de Programa)* - é um registrador responsável por armazenar o endereço da próxima instrução a ser buscada na Memória de Instruções, para a futura execução. Este é incrementado a cada busca da instrução atual, ficando com o endereço da próxima instrução na memória.

O PC vai sendo incrementado a cada nova instrução para apontar sempre para a próxima instrução a ser executada, quando a execução do programa é de forma sequencial. Porém, quando é executada uma instrução de salto, o PC é alterado para o novo endereço de memória após o salto, ao invés de ser incrementado pra o endereço seguinte. (GOMES, 2011).

O bloco PC é mostrado na FIGURA 2. Neste bloco temos o sinal de clock *clk* utilizado como referência para a gravação dos dados, o sinal de controle *write*, o qual habilita a escrita no registrador, e a entrada e saída do PC, nomeadas por *reg_input* e *reg_output*, respectivamente.

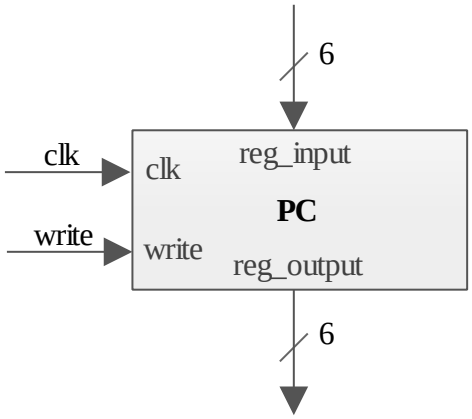


FIGURA 2 – Símbolo para diagrama de blocos de um PC.

O ALGORITMO 2 apresenta o código em System Verilog do registrador PC. Este registrador é parametrizável, ou seja, pode ser referenciado para qualquer tamanho de palavra.

```
1 module PC
2  #(parameter integer Data_Width =2)
3 (clk, write, reg_input, reg_output);
4  input logic clk;
5   input logic write;
6   input logic  [Data_Width -1:0] reg_input;
7   output logic [Data_Width -1:0] reg_output;
8
9  always_ff @ (posedge clk)
10        if (write)
12            reg_output <= reg_input;
13        else
14            reg_output <= 0;
15 endmodule
```

ALGORITMO 2 - Código parametrizável para o PC.

3.1.1.2.Registrador de Instruções

O *Registrador de Instruções* é um tipo de registrador que armazena a instrução atual que esta sendo executada.

O bloco *Registrador de Instruções* é mostrado na FIGURA 3. Neste bloco temos o sinal de clock *clk* utilizado como referência para a gravação dos dados, o sinal de controle *write*, o qual habilita a escrita no registrador, e a entrada e saída do PC, nomeadas por *reg_input* e *reg_output*, respectivamente.

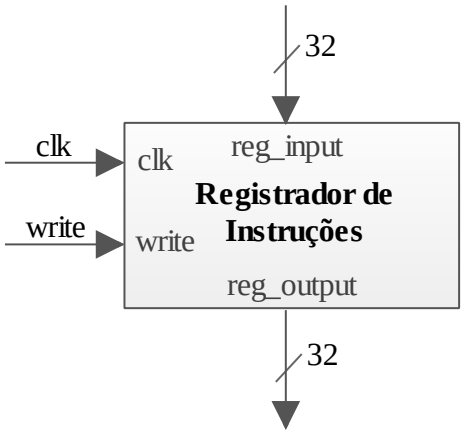


FIGURA 3 – Símbolo para diagrama de blocos de um Registrador de Instruções.

O ALGORITMO 3 apresenta o código em System Verilog do bloco *Registrador de Instruções*. Este registrador é parametrizável, ou seja, pode ser referenciado para qualquer tamanho de palavra.

```
1 module reg_instruc
2  #(parameter integer Data_Width =2)
3 (clk, write, reg_input, reg_output);
4  input logic clk;
5   input logic write;
6   input logic  [Data_Width -1:0] reg_input;
7   output logic [Data_Width -1:0] reg_output;
8
9  always_ff @ (posedge clk)
10        if (write)
12            reg_output <= reg_input;
13        else
14            reg_output <= 0;
15 endmodule
```

ALGORITMO 3 - Código parametrizável para um Registrador de Instruções.

3.1.1.3.Contador de Endereço de Memória

Um contador é um componente construído a partir de uma extensão de um registrador, que pode incrementar ou decrementar o próprio valor a cada ciclo de relógio, quando uma entrada de controle de habilitação de contagem estiver ativa. Incrementar significa adicionar 1, decrementar significa subtrair 1 (VAHID, 2008).

O bloco *Contador de Endereço de Memória* (C.E.M) é mostrado na FIGURA 4. Neste bloco temos o sinal de clock *clk* utilizado como referência para a gravação dos dados, o sinal de controle *write*, o qual habilita a contagem, o sinal de controle *rst* o qual reseta o contador, a entrada do C.E.M, nomeada por *in_CEM*, e a saída do C.E.M, nomeada por *out_CEM*.

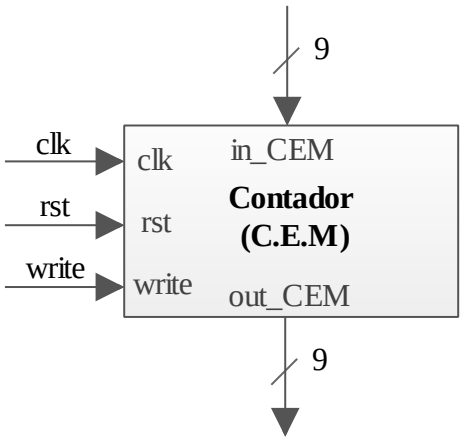


FIGURA 4 – Símbolo para diagrama de blocos de um Contador de Endereço de Memória.

O ALGORITMO 4 apresenta o código em System Verilog do bloco *Contador de Endereço de Memória.*

```
1 module contador_end_memoria
2 (
3  input logic clk,
4  input logic rst,
5  input logic write,
6  input logic [8:0] reg_input,
7   output logic [8:0]reg_outpu);
8
9 Reg_instruc #  (.Data_Width(9))  X1    (.clk (clk),
10                                        .write (write),
11                                        .rst (rst),
12                                        .reg_input (reg_input [8:0]),
13                                        .reg_output (reg_output [8:0]));
14
15 endmodule
```

ALGORITMO 4 - Código para um Contador de Endereço de Memória.

3.1.1.4.Banco de Registradores – Variáveis e Constantes

Um *Banco de Registradores* “MxN” é um componente de memória de blocos operacionais que propicia um acesso eficiente a um conjunto de “M” registradores, onde cada registrador tem uma largura de “N” bits (VAHID, 2008).

A FIGURA 5 apresenta o bloco *Banco de Registradores*. Neste bloco temos o sinal de clock *clk* utilizado como referência para a leitura e escrita dos dados, o sinal de controle *write*, o qual habilita a leitura e escrita dos dados no banco, os endereços de leitura 1 e leitura 2 dos dados do banco, nomeado por *endereço_leitura1 e endereço_leitura2*, respectivamente, o endereço de escrita dos dados, nomeado por *endereço_escrita*, a entrada de dados, nomeada por *dado*, e as saídas do banco, nomeadas por *dado_lido1 e dado_lido2*, respectivamente.

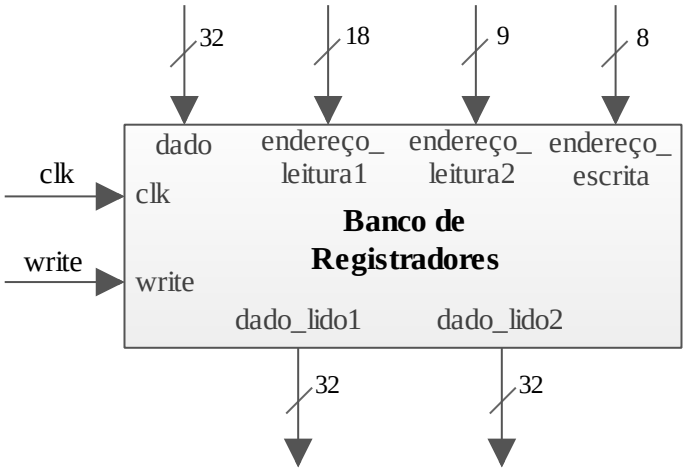


FIGURA 5 – Símbolo para diagrama de blocos de um Banco de Registradores.

O ALGORITMO 5 apresenta o código em System Verilog do bloco *Banco de Registradores*.

```
1 module banco
2 (
3
4  input [17:0]read_address1,
5  input [8:0]read_address2,
6  input [7:0]write_address,
7  input clk,
8  input write,
9  input [31:0]data,
10  output logic [31:0]dado_lido1,
11  output logic [31:0]dado_lido2);
12
13   reg [31:0] banco [0:255];
14
15
16 always_ff @ (posedge clk)
```

```
17        begin
18            if (write)
19                begin
20                    dado_lido1[31:0]<=banco[read_address1];
21                    dado_lido2[31:0]<=banco[read_address2];
22                end
23        end
24
25  always_ff @ (negedge clk)
26        begin
27            if (write)
28                begin
29                    banco[write_address]<=data[31:0];
30            end
31        end
32
33  endmodule
```

ALGORITMO 5 - Código para um Banco de registradores.

O banco de registradores é formado por 256 posições, ou seja, 256 endereços de leitura ou escrita de dados, conforme esta mostrado no ALGORITMO 5.

3.1.2. Memórias

Uma *Memória* "MxN" é um componente capaz de armazenar "M" itens de dados, de "N" bits cada um. Cada item de dados é conhecido como uma *palavra* (VAHID, 2008).

Esta seção apresentará os tipos de memórias usadas na implementação da arquitetura do *Processador Dedicado*.

3.1.2.1.Memória de Instruções

A *Memória de Instruções* é um tipo que memória que tem por função armazenar as instruções que serão executadas.

A FIGURA 6 apresenta o bloco *Memória de Instruções*. Neste bloco temos o sinal de clock *clk* utilizado como referência para a leitura dos dados, o sinal de controle *read*, o qual habilita a leitura dos dados da memória, o endereço de leitura da memória, nomeado por *read_address*, e a saída da memória, nomeada por *out_memory*.

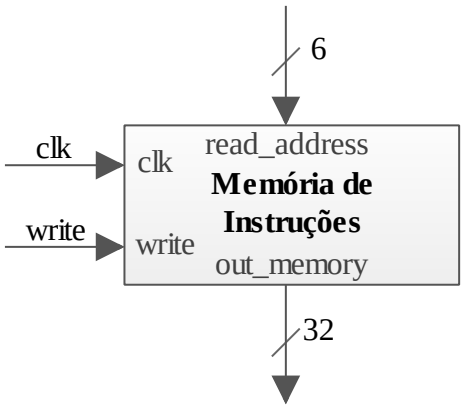


FIGURA 6 – Símbolo para diagrama de blocos de uma Memória de Instruções.

O código em System Verilog do bloco *Memória de Instruções* é apresentado no ALGORITMO 6.

```
1 module memory_instruction
2 (
3  input logic clk,
4  input logic read,
5 input logic [5:0]read_address,
6 output logic [31:0]out_memory);
7
8   reg [31:0] memory [0:63];
9
10 always_ff @ (posedge clk)
11         begin
12            if (read)
13                  begin
14                       out_memory[31:0] <= memory_instruction[read_address];
15                   end
16 endmodule
```

ALGORITMO 6 - Código para uma Memória de Instruções.

A memória de instruções é formada por 64 posições, ou seja, 64 endereços de leitura de dados, pois é uma memória do tipo ROM, ou seja, ela permite somente a leitura de dados, conforme mostrado no ALGORITMO 6.

3.1.2.2.Memória de Tabela

Em ciência da computação a tabela hash, também conhecida por tabela de espalhamento, é uma estrutura de dados especial, que associa chaves de pesquisa (*hash*) a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado (SEDGEWICK, 1998).

O bloco *Memória de Tabela* é mostrado na FIGURA 7. Neste bloco temos o sinal de clock *clk* utilizado como referência para a escrita e leitura dos dados, o sinal de controle *write*, o qual habilita a leitura e a escrita dos dados na memória, o endereço de leitura da memória, nomeado por *read_address*, o endereço de escrita da memória, nomeado por *write_address*, a entrada de dados, nomeada por *dado*, e as saídas da memória, nomeadas por e *dado_lido*, *identificador* e *v_lido*, respectivamente. A saída *identificador* é a chave de pesquisa para a saída *dado_lido*, ou seja, o dado que será liberado na saída é indicado pelo sinal *identificador*. A saída *v_lido* indica se o *identificador* naquela posição da memória está ocupado ou não por algum valor.

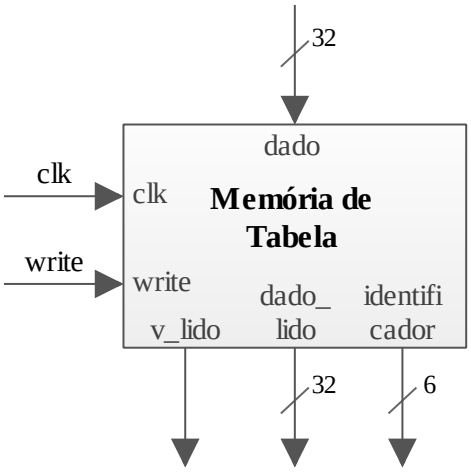


FIGURA 7 – Símbolo para diagrama de blocos de uma Memória de Tabela.

O ALGORITMO 7 apresenta o código em System Verilog do bloco *Memória de Tabela*.

```
1 module memory_table
2 (
3  input logic clk,
4  input logic write,
5  input logic [31:0]dado,
6  input logic [8:0]write_addres,
7  input logic [8:0]read_addres,
```

```
8  output logic [5:0]identificador,
9  output logic v_lido,
10  output logic [31:0]dado_lido);
11
12  wire [5:0]out_ID1;
13  wire [5:0]out_ID2;
14
15  reg [31:0] memory_table [0:63];
16  reg[5:0] hash [0:63];
17  reg[0:0] verification_bit [0:63];
18
19
20 always_ff @ (negedge clk)
21         begin
22           if (write)
23                 begin
24                     out_ID2[5:0] <= hash[write_address];
25                      memory_table[out_ID2] <= dado[31:0];
26                  end
27           end
28  always_ff @ (posedge clk)
29         begin
30           if (write)
31                 begin
32                     out_ID1[5:0] <= hash[read_address];
33                      dado_lido[31:0] <= memory_table[out_ID1];
34                       identificador[5:0] <= out_ID1[5:0];
35                       v_lido <= verification_bit[read_address];
36                  end
37           end
38  always_comb
39           begin
40              if (out_ID[5:0]==6'b0)
41                    begin
42                        verification_bit[read_address] <= 1;
43                     end
44              else
45                     begin
46                         verification_bit[read_address] <= 0;
```

```
47                  end
48              end
49 endmodule
```

ALGORITMO 7 - Código para uma Memória de Tabela.

Conforme mostrado no ALGORITMO 7, a memória de tabela é formada por 64 posições, ou seja, 64 endereços de leitura ou escrita de dados.

3.1.3. Comparador

Um comparador de igualdade é um componente de bloco operacional que compara duas entradas "A" e "B", produzindo na saída um sinal de controle igual a 1 quando as duas entradas são iguais (VAHID, 2008).

O bloco *Comparador* é apresentado na FIGURA 8. Este bloco é composto por duas entradas de dados "A" e "B" de 32 bits cada uma, e uma saída de 1 bit. Se as entradas forem iguais, a saída será igual a 1, caso contrário, a saída será igual a 0.

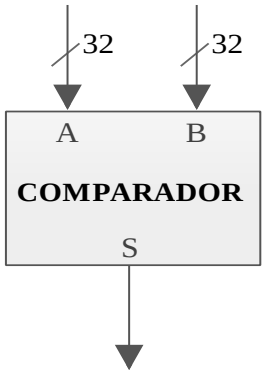


FIGURA 8 – Símbolo para diagrama de blocos de um Comparador.

O ALGORITMO 8 apresenta o código em System Verilog do bloco *Comparador*.

```
1 module comparador
2
3  input logic [31:0]A,
4   input logic [31:0]B,
5   output logic out);
6
7  always_comb
8         begin
9             if (A[31:0] == B[31:0])
```

```
10              out <= 1;
11        else
12              out <= 0;
13       end
14
15 endmodule
```

ALGORITMO 8 - Código para um Comparador.

3.1.4. Somadores

A adição de dois números binários é possivelmente a mais comum das operações que é realizada com dados em um sistema digital (VAHID, 2008). Em uma operação de adição, os dígitos são somados bit a bit, da direita para a esquerda, com *carries* (“vai-uns”) sendo passados para o próximo dígito à esquerda, como fosse feito manualmente (PATTERSON et al., 2005). Um somador binário de “n” bits é um circuito lógico que adiciona dois números “A” e “B” gerando uma soma “S” de “n” bits (VAHID, 2008).

O bloco *Somador* que será utilizado na arquitetura do *Processador* é apresentado na FIGURA 9. Este bloco é composto por duas entradas de dados “A” e “B” de 32 bits cada uma, e uma saída com tamanho de dados igual a 32 bits.

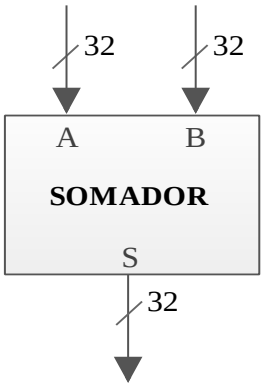


FIGURA 9 – Símbolo para diagrama de blocos de um Somador de 32 bits.

O código em System Verilog de um somador de 32 bits está mostrado no ALGORITMO 9.

```
1  module somador
2 (
3 input [31:0]A,
4 input [31:0]B,
```

```
5 output logic [31:0]S);
6
7 always_comb
8          begin
9               S[31:0] <= (A[31:0] + B[31:0]);
10          end
11 endmodule
```

ALGORITMO 9 - Código de um Somador de 32 bits.

Devido a necessidade de o PC ser incrementado a cada ciclo de instrução, será implementado um somador que realize a seguinte operação:

$$PC=PC+1 \quad (1)$$

Este somador é apresentado na FIGURA 10. Este bloco é composto por duas entradas de dados “A” e “1” de 18 bits cada uma, e uma saída ”S” com tamanho de dados igual a 18 bits.

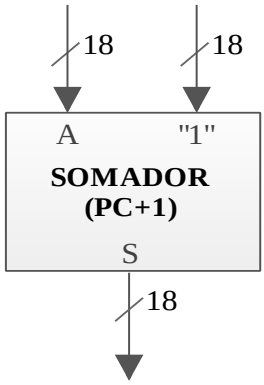


FIGURA 10 – Símbolo para diagrama de blocos de um Somador PC+1.

O código em System Verilog do somador PC+1 está mostrado no ALGORITMO 10.

```
1  module somador
2 (
3 input [17:0]A,
4 output logic [17:0]S);
5
6 always_comb
7          begin
8               S[31:0] <= (A[17:0] + 18'b000000000000000001);
9          end
10 endmodule
```

ALGORITMO 10 - Código do Somador .

Além do Somador PC+1, também será necessário implementar outros três somadores, que realizem as seguintes operações:

S=A[13:6]+1 (2)

S=A[13:6]+2 (3)

S=A[13:6]+3 (4)

O código em linguagem System Verilog para estes somadores é análogo ao código do ALGORITMO 10.

3.1.5. Subtratores

A subtração de números binários é realizada a partir de uma adição, ou seja, o operando é negado antes de ser somador. Esta negação é uma representação denominada de *complemento de dois*. O complemento de dois de um número binário envolve o complemento de cada bit e a soma de "1" ao bit menos significativo do valor complementado. Por exemplo, para negar 0101_2, basta complementar o número para obter 1010_2 e depois somar "1" ao bit menos significativo para obter 1011_2. Assim, para implementar uma operação *a* – *b*, basta complementar o operando *b*. (MURDOCA, 2000).

O bloco *Subtrator* que será utilizado na arquitetura do *Processador* é apresentado na FIGURA 11. Este bloco é composto por duas entradas de dados "A" e "B" de 32 bits cada uma, e uma saída com tamanho de dados igual a 32 bits.

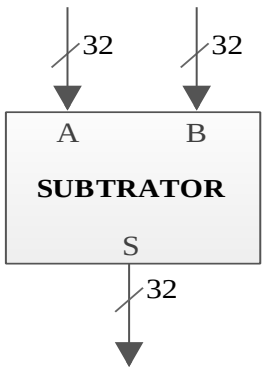


FIGURA 11 – Simbolo para diagrama de blocos de um subtrator de 32 bits.

O código em System Verilog do subtrator de 32 bits, utilizado na arquitetura do processador, está mostrado no ALGORITMO 11.

```
1 module subtrator
2 (
3 input [31:0]A,
4 input [31:0]B,
5output logic [31:0]S);
6
7 always_comb
8          begin
9               S[31:0] <= (A[31:0] - B[31:0]);
10          end
11 endmodule
```

ALGORITMO 11 - Código de um Subtrator de 32 bits.

3.1.6. Multiplexadores

Um multiplexador, também chamado de forma abreviada de “mux”, é um bloco construtivo usado em circuitos digitais. Um multiplexador Nx1 possui “N” entradas de dados e uma única saída, permitindo que apenas uma das entradas seja passada para a saída. Com isso, muitas vezes os multiplexadores são chamados de *seletores* (VAHID, 2008).

A seleção de qual entrada será passada para a saída é feita através de um sinal de controle. O sistema de seleção de entradas e saída de um multiplexador 2x1, ou seja, 2 entradas e uma saída é mostrado na TABELA 4. Para este exemplo, a seleção de entradas e saída é realizada através de um sinal de controle (“sel”) de 1 bit.

TABELA 4 – Sistema de seleção de um multiplexador 2x1.

Entradas	Seleção	Saídas
in_0	sel = 0	out = in_0
in_1	sel = 1	out = in_1

A FIGURA 12 mostra o bloco *multiplexador 2x1*.

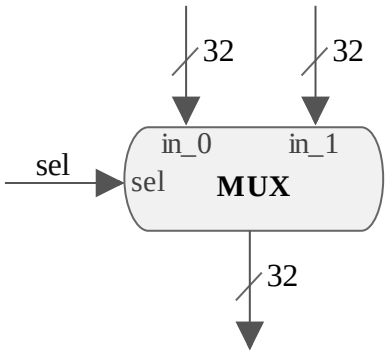


FIGURA 12 – Símbolo para diagrama de blocos de um multiplexador 2x1 de 32 bits.

O código em System Verilog de um multiplexador 2x1 está mostrado no ALGORITMO 12.

```
1 module mux_2x1
2
3   #(parameter integer Data_Width=2)
4             (in_0, in_1, sel ,out);
5              input       [Data_Width-1:0]   in_0;
6              input       [Data_Width-1:0]   in_1;
7              input                          sel;
8              output logic [Data_Width-1:0]   out ;
9                 always_comb
10                   case (sel)
11                           0   :   out <= in_0;
12                           1   :   out <= in_1;
13                   endcase
14 endmodule
```

ALGORITMO 12 - Código de um multiplexador 2x1 parametrizável.

Este multiplexador é descrito de forma parametrizável, ou seja, é possível adotar qualquer quantidade de bits às entradas e saída.

Na arquitetura do *Processador Dedicado* serão utilizados multiplexadores 2x1 e multiplexadores 5x1, ou seja, multiplexados de 2 entradas e uma saída e multiplexadores de 5 entradas e uma saída.

3.1.7. ULA (Unidade Lógica e Aritmética)

A Unidade Lógica e Aritmética é um componente de bloco operacional capaz de executar diversas operações lógicas e aritméticas, através de duas entradas de “n” bits, gerando uma

saída de dados de “n” bits (VAHID, 2008). Como a arquitetura proposta do Processador não necessita de operações lógicas, a ULA fica condicionada a apenas operações aritméticas de Soma, Subtração e Multiplicação.

A seleção de qual operação é realizada pela ULA é feita através de um sinal de controle (sel). O sistema de seleção da operação aritmética a ser realizada pela ULA é mostrado na TABELA 5.

TABELA 5 – Sistema de seleção da operação da ULA.

sel	Operação
00	Soma
01	Subtração
10	Multiplicação
11	-

A FIGURA 13 mostra o símbolo para diagrama de blocos de uma ULA.

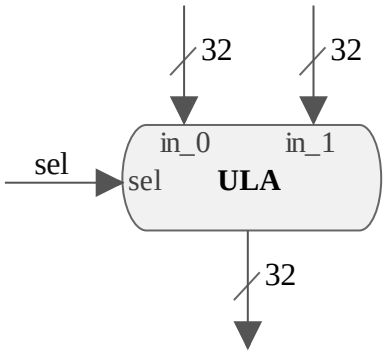


FIGURA 13 – Símbolo para diagrama de blocos de uma ULA.

O código em System Verilog de uma ULA, com tamanho de dados das entradas e saída igual a 32 bits, está mostrado no ALGORTIMO 13.

```
1 module ULA
2 (
3 input [31:0]a,
4 input [31:0]b,
5 input [1:0]sel_ULA,
6 output logic [31:0]out);
7
8         always_comb
9             begin
```

```
10          case(sel_ULA)
11                      2'b00:   out[31:0] <= (a[31:0] + b[31:00]);
12                      2'b01:   out[31:0] <= (a[31:0] - b[31:00]);
13                      2'b10:   out[31:0] <= (a[31:0] * b[31:00]);
14                      default:  out[31:0] <= 0;
15          endcase
16          end
17 endmodule
```

ALGORITMO 13 - Código de uma ULA de 32 bits.

3.1.8. Unidade de Controle

Em um circuito digital, os registradores armazenam bits. Logo, estes bits armazenados significam que o circuito tem memória, ou seja, tem *estado*, o que resulta nos chamados circuito sequenciais. Por outro lado, podemos usar esses estados para projetar circuitos que tenham determinados comportamentos ao longo do tempo. Um circuito sequencial que controla saídas booleanas com base em entradas booleanas e em um comportamento específico, é comumente referido como *Bloco de Controle* (VAHID, 2008).

O bloco *Controle* que será utilizado na arquitetura do *Processador* é apresentado na FIGURA 14. Este bloco é composto por 6 entradas e 20 saídas de dados.

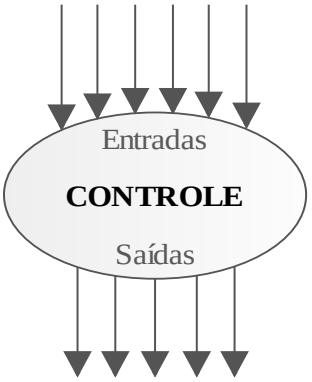


FIGURA 14 – Símbolo para diagrama de blocos de uma Unidade de Controle.

A seleção de quais sinais de saída estarão ativos para cada instrução na saída do controle é feita através do campo *Opcode* (bits [5:0]) de cada instrução. Este sistema de seleção é mostrado na TABELA 6.

TABELA 6 – Sistema de seleção dos sinais de saída do Controle.

Instrução	Opcodes						Write_ PC	Write_ RI	Write_ banco	Read_ banco	Write_ CEM	Clear_ CEM	Write_ Mem_ Instruc
	OP [5]	OP [4]	OP [3]	OP [2]	OP [1]	OP [0]							
MOVE	0	0	0	0	0	0	1	1	1	1	0	0	1
LOADK	0	0	0	0	0	1	1	1	1	1	0	0	1
ADD	0	0	1	1	0	0	1	1	1	1	0	0	1
SUB	0	0	1	1	0	1	1	1	1	1	0	0	1
MUL	0	0	1	1	1	0	1	1	1	1	0	0	1
GETTA-BLE	0	0	0	1	1	0	1	1	0	0	0	0	1
SETTABLE	0	0	1	0	0	1	1	1	0	0	0	0	1
NEWTA-BLE	0	0	1	0	1	0	1	1	0	0	0	0	1
FORPREP	1	0	0	0	0	0	1	1	1	1	0	0	1
FORLOOP	0	1	1	1	1	1	1	1	1	1	0	0	1

Instrução	Opcodes						Op ULA	Write_ Mem_ Tabela	Write_ regis-trador	mux1	mux2	mux3	mux4
	OP [5]	OP [4]	OP [3]	OP [2]	OP [1]	OP [0]							
MOVE	0	0	0	0	0	0	11	0	0	0	1	1	0
LOADK	0	0	0	0	0	1	11	0	0	0	1	0	0
ADD	0	0	1	1	0	0	00	0	0	0	1	1	0
SUB	0	0	1	1	0	1	01	0	0	0	1	1	0
MUL	0	0	1	1	1	0	10	0	0	0	1	1	0
GETTA-BLE	0	0	0	1	1	0	11	0	0	0	0	1	0
SETTABLE	0	0	1	0	0	1	11	0	0	0	0	0	1
NEWTA-BLE	0	0	1	0	1	0	11	0	0	0	0	0	0
FORPREP	1	0	0	0	0	0	11	0	0	1	0	0	1
FORLOOP	0	1	1	1	1	1	11	0	0	0	0	0	1

Instrução	Opcodes						mux5	mux6	mux7	mux8	mux9	mux10	--
	OP [5]	OP [4]	OP [3]	OP [2]	OP [1]	OP [0]							
MOVE	0	0	0	0	0	0	0	100	0	0	0	0	--
LOADK	0	0	0	0	0	1	0	100	0	0	0	0	--
ADD	0	0	1	1	0	0	0	010	0	0	0	0	--
SUB	0	0	1	1	0	1	0	010	0	0	0	0	--

Instrução	Opcodes						mux5	mux6	mux7	mux8	mux9	mux10	--
	OP [5]	OP [4]	OP [3]	OP [2]	OP [1]	OP [0]							
MUL	0	0	1	1	1	0	0	010	0	0	0	0	--
GETTA-BLE	0	0	0	1	1	0	1	111	0	0	0	0	--
SETTABLE	0	0	1	0	0	1	0	111	0	0	0	0	--
NEWTA-BLE	0	0	1	0	1	0	0	111	0	0	0	0	--
FORPREP	1	0	0	0	0	0	0	010	0	0	0	0	--
FORLOOP	0	1	1	1	1	1	1	010	1	0	0	0	--

Devido as instruções NEWTABLE, GETTABLE, SETTABLE e FORLOOP não poderem ser executadas em apenas um estado de controle, ou seja, em apenas um ciclo de clock, estas serão implementadas de usando mais de um estado da máquina de estados do bloco controle.

3.2. Conjunto de Instruções do Processador Dedicado

Devido ao conjunto de instruções influenciar na definição e implementação da arquitetura, ele é considerado um dos pontos centrais na arquitetura de um processador. Por exemplo, as operações realizadas pela unidade lógica e aritmética, o número e função dos registradores, o caminho de dados, o projeto da seção de controle, além de as operações básicas que acontecem dentro da seção de processamento depender das instruções que devem ser executadas. (STALLINGS, 2002).

Nesta seção será visto detalhadamente o caminho de dados de cada instrução necessária para executar o programa teste, as quais são apresentadas na TABELA 3. As instruções estarão divididas em MOVE, LOADK, ADD, SUB, MUL, GETTABLE, SETTABLE, NEWTABLE, FORPREP e FORLOOP, e serão descritas separadamente a seguir. Devido à instrução RETURN ser apenas uma instrução de retorno ao início do programa teste, não será necessário a implementação de um caminho de dados especial como para as demais instruções.

3.2.1. Instrução MOVE

A instrução MOVE copia o valor presente no registrador R(B) para o registrador R(A) (MAN, 2006). A TABELA 7 apresenta um resumo desta instrução.

TABELA 7 - Tipo e Operação da Instrução MOVE.

Instrução	Tipo	Operação
MOVE	iABC	R(A) = R(B)

Apesar de a instrução MOVE ser do tipo iABC, esta ocupa somente os campos [13:6]A, [31:23]B e [5:0]OPCODE, sendo que o campo [14:22]C não é necessário para a execução da instrução.

A FIGURA 15 apresenta o caminho de dados para a instrução MOVE.

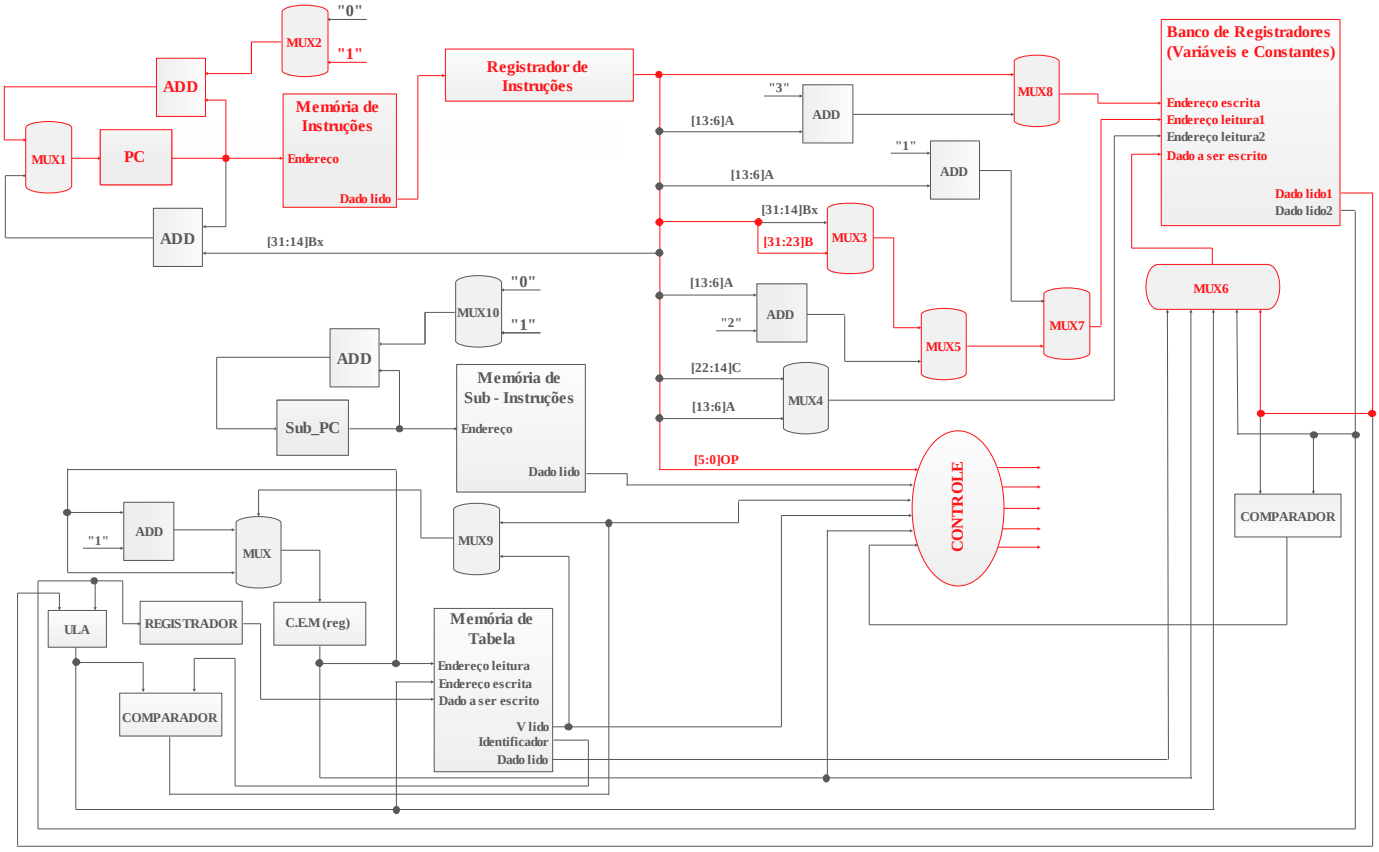


FIGURA 15 – Caminho de dados para a instrução MOVE.

Para a execução da instrução MOVE, conforme apresenta a FIGURA 15, o conteúdo de um registrador é escrito em outro registrador, onde o endereçamento do registrador de leitura e escrita é feito através dos campos [31:23]B e [13:6]A, respectivamente. Além disso, o PC é incrementado em “1” com a finalidade de apontar para a próxima instrução a ser executada.

3.2.2. Instrução LOADK

A instrução LOADK carrega o valor da constante Bx no registrador R(A) (MAN, 2006). A TABELA 8 apresenta um resumo desta instrução.

TABELA 8 - Tipo e Operação da Instrução LOADK.

Instrução	Tipo	Operação
LOADK	iABx	R(A) = Kst(Bx)

A FIGURA 16 apresenta o caminho de dados para a instrução LOADK.

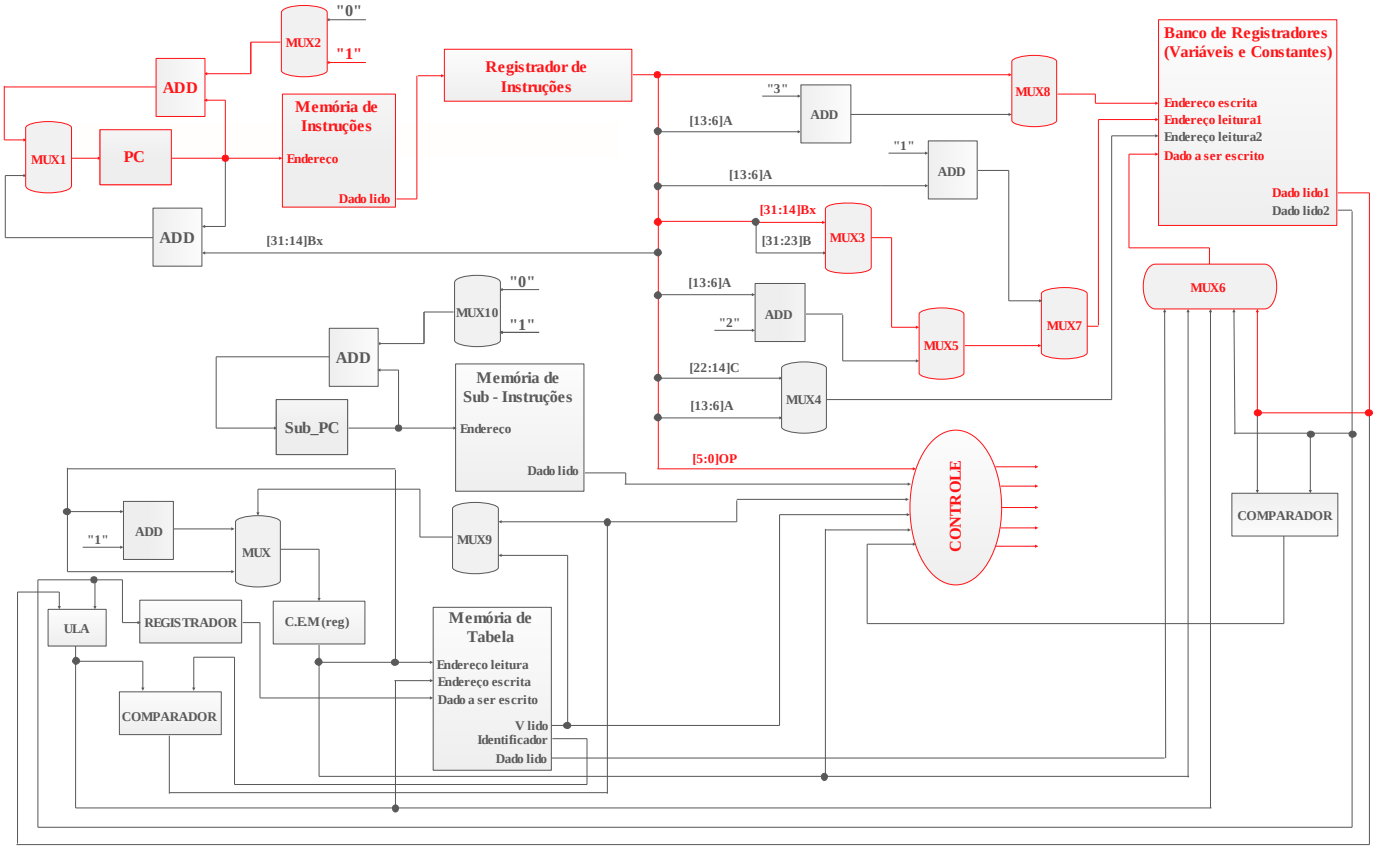


FIGURA 16 – Caminho de dados para a instrução LOADK.

Para a execução da instrução LOADK, conforme apresenta a FIGURA 16, o conteúdo de uma constante é escrito em um registrador, onde o endereçamento da constante e do registrador é feito através dos campos [31:14]Bx e [13:6]A, respectivamente. Além disso, o PC é incrementado em “1” com a finalidade de apontar para a próxima instrução a ser executada.

3.2.3. Instruções ADD, SUB e MUL

As instruções ADD, SUB e MUL realizam a operação de soma, subtração ou multiplicação, respectivamente, entre RK(B) e RK(C), os quais podem ser registradores ou constantes. O resultado da operação é escrito no registrador R(A) (MAN, 2006). A TABELA 9 apresenta um resumo destas instruções.

TABELA 9 - Tipo e Operação das Instruções ADD, SUB e MUL.

Instrução	Tipo	Operação
ADD	iABC	R(A) = RK(B) + RK(C)
SUB	iABC	R(A) = RK(B) - RK(C)
MUL	iABC	R(A) = RK(B) * RK(C)

A FIGURA 17 apresenta o caminho de dados para as instruções ADD, SUB e MUL.

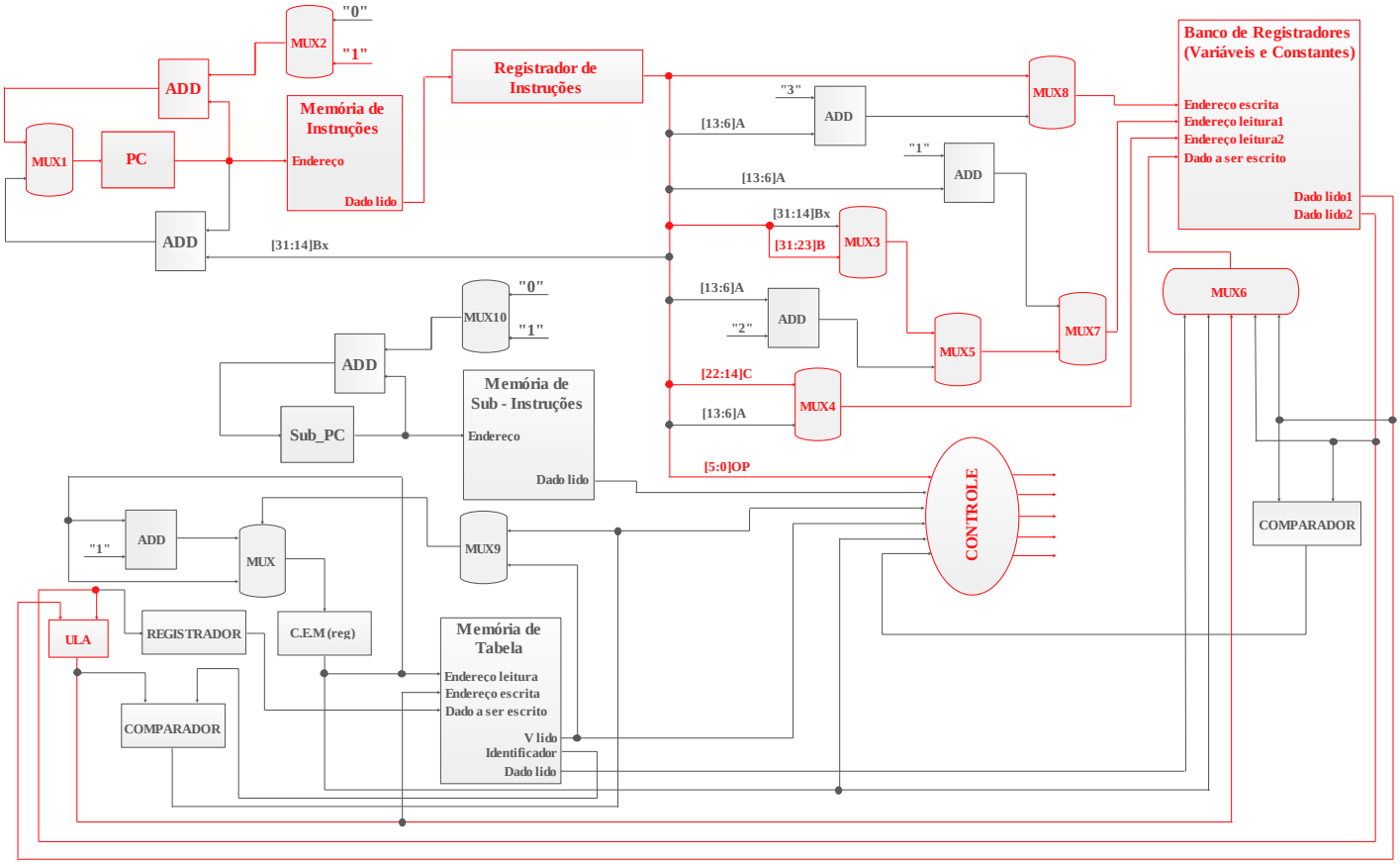


FIGURA 17 – Caminho de dados para as instruções ADD, SUB e MUL.

Para a execução das instruções ADD, SUB ou MUL, como é mostrado na FIGURA 17, é realizada uma operação aritmética de soma, subtração ou multiplicação entre os valores contidos em dois registradores e o resultado é escrito em outro registrador. O endereçamento dos registradores de leitura é feito pelos campos [31:23]B e [22:14]C, enquanto o endereçamento do registrador de escrita é feito pelo campo [13:6]A. As operações aritméticas são realizadas através de uma ULA, e a seleção da operação a ser executada é realizada através de um sinal de controle. Além disso, o PC é incrementado em “1” com a finalidade de apontar para a próxima instrução a ser executada.

3.2.4. Instrução GETTABLE

A instrução GETTABLE copia o valor de um elemento de tabela no registrador R(A). A tabela é referencia pelo registrador R(B), enquanto o índice para a tabela é dado por RK(C), que pode ser o valor do registrador R(C) ou uma constante (MAN, 2006). A TABELA 10 apresenta um resumo desta instrução.

TABELA 10 - Tipo e Operação da Instrução GETTABLE.

Instrução	Tipo	Operação
GETTABLE	iABC	R(A) = R(B)[RK(C)]

A FIGURA 18 apresenta o fluxograma para a implementação da instrução GETTABLE.

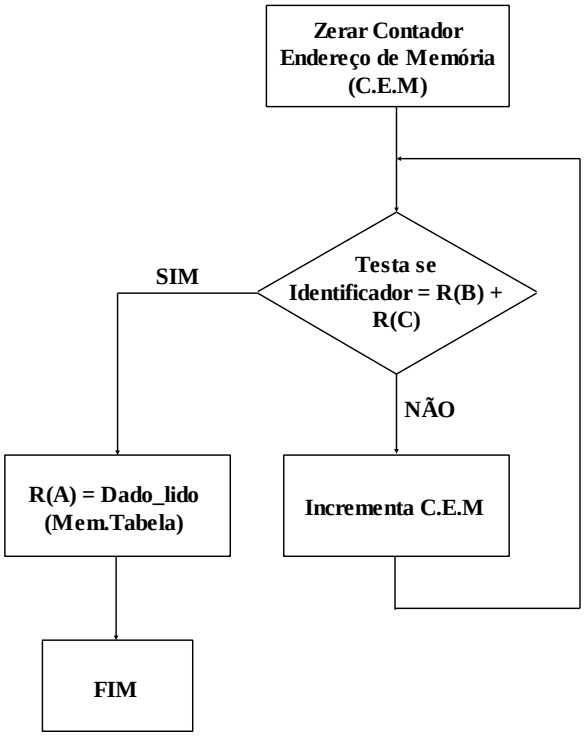


FIGURA 18 – Fluxograma da instrução GETTABLE.

Devido esta instrução não permitir a execução monociclo, optou-se por dividi-la em sub-instruções, sendo necessário também a implementação de uma Memória de Sub-intruções e um Sub_PC para a execução da instrução GETTABLE.

As FIGURAS 19, 20, 21 e 22 apresentam os caminhos de dados para a instrução GET-TABLE.

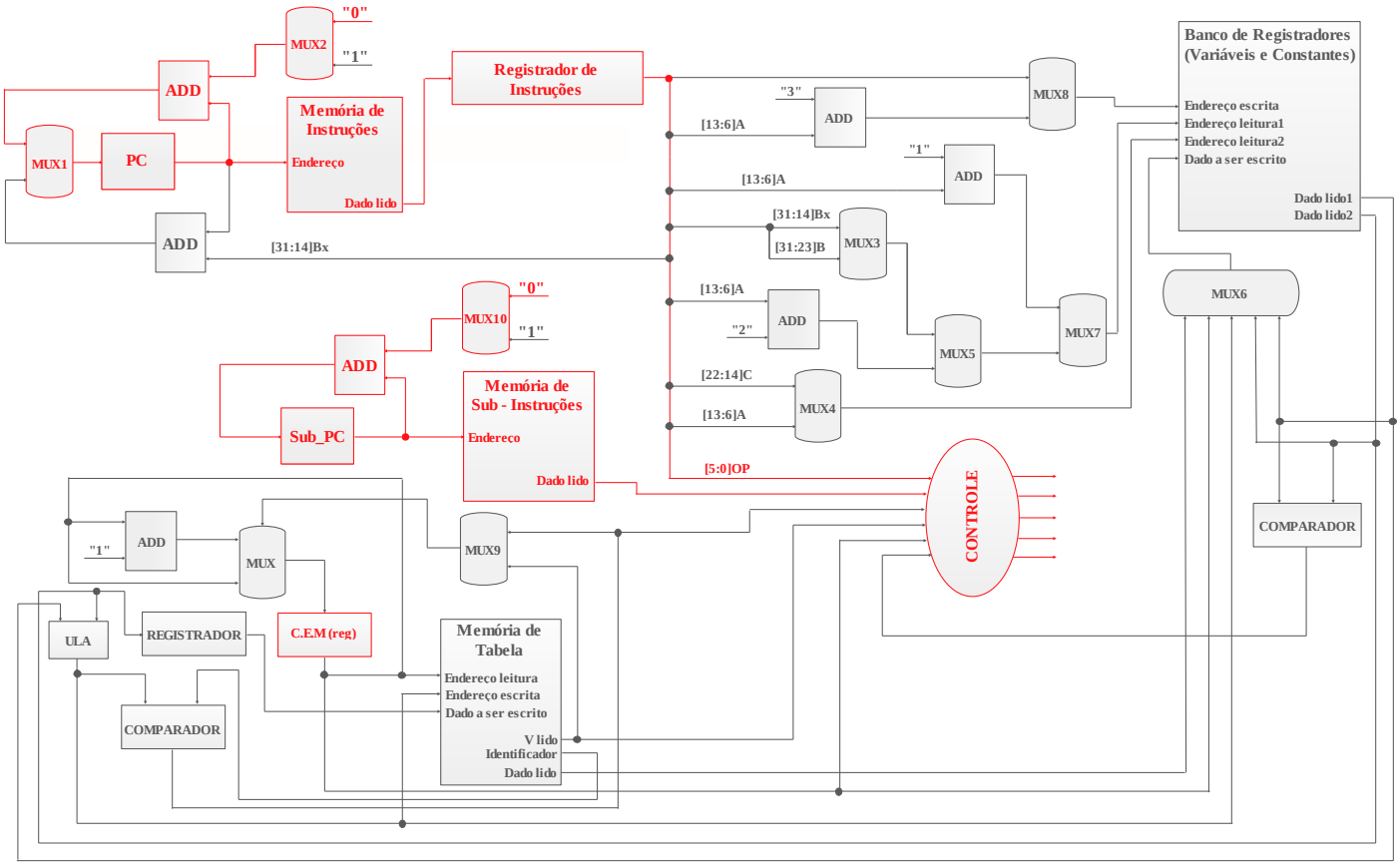


FIGURA 19 – Caminho de dados para a instrução GETTABLE.

O primeiro passo para a execução da instrução GETTABLE, como mostra FIGURA 19, é zerar o C.E.M, que é um registrador contador responsável por fazer a varredura da Memória de Tabela em busca de uma posição livre. Neste passo, o PC e o Sub_PC não são incrementados.

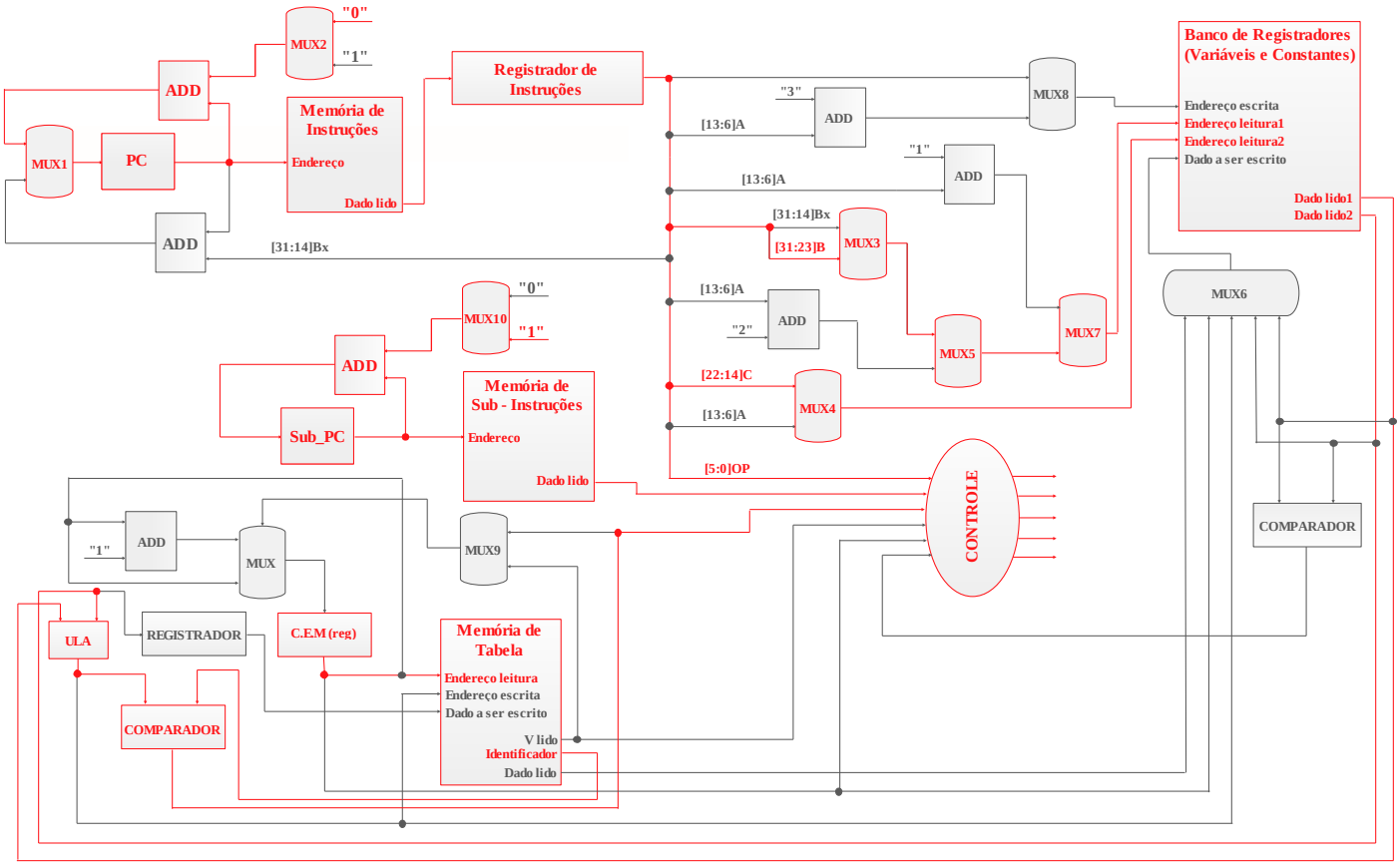


FIGURA 20 – Caminho de dados para a instrução GETTABLE.

Na segunda etapa, como mostra a FIGURA 20, é feito o teste se o campo “Identificador” da Memória de Tabela, endereçado pelo valor do C.E.M, é igual ao resultado da operação de adição entre os valores presentes em dois registradores, endereçados pelos campos [31:23]B e [22:14]C.

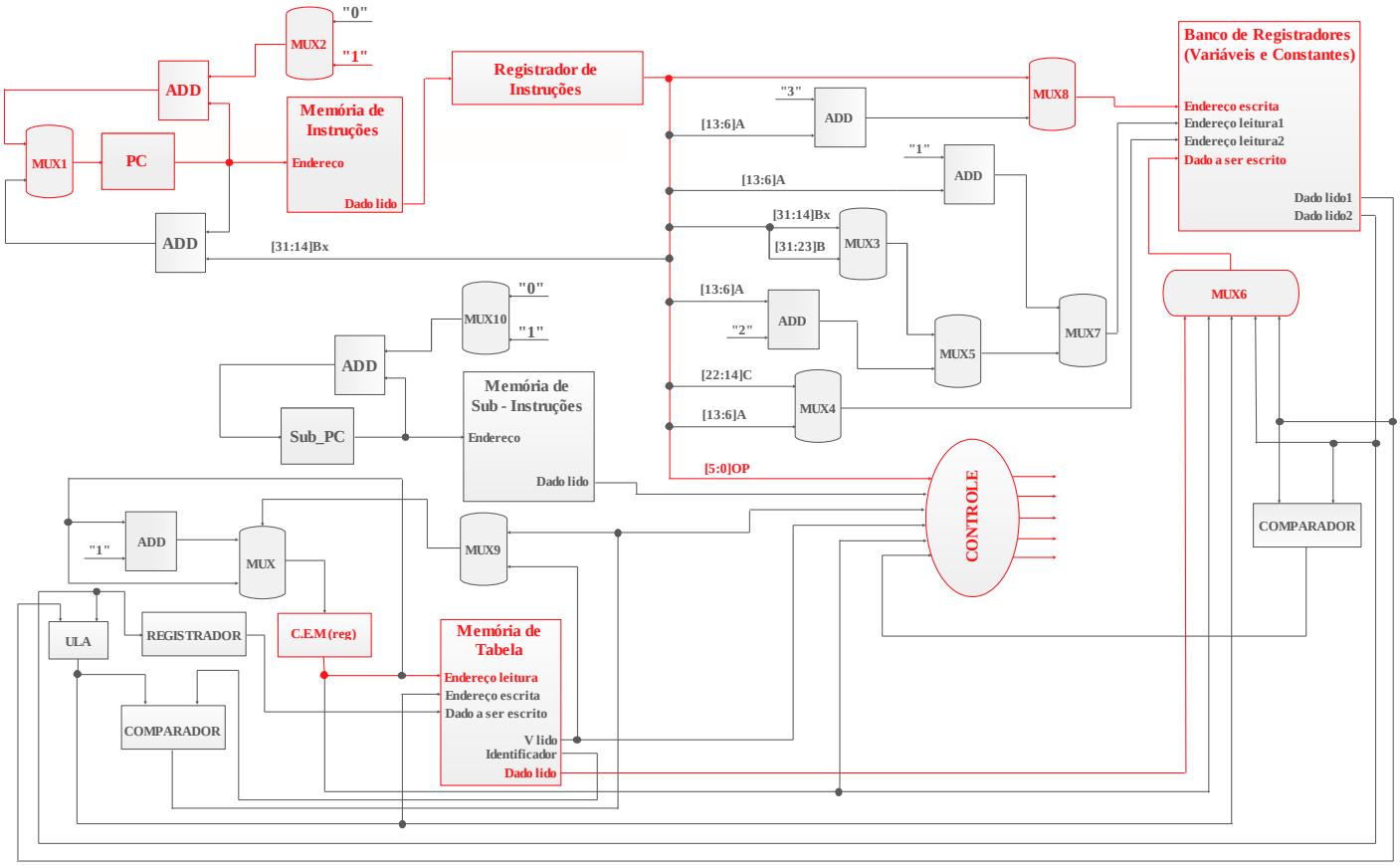


FIGURA 21 – Caminho de dados para a instrução GETTABLE.

Caso o teste mencionado anteriormente seja verdadeiro, no terceiro passo, conforme demonstrado na FIGURA 21, o dado lido da Memória de Tabela, endereçado pelo valor do C.E.M, é escrito no registrador endereçado pelo campo [13:6]A. Neste passo, o PC também é incrementado em ”1” com a finalidade de apontar para a próxima instrução a ser executada.

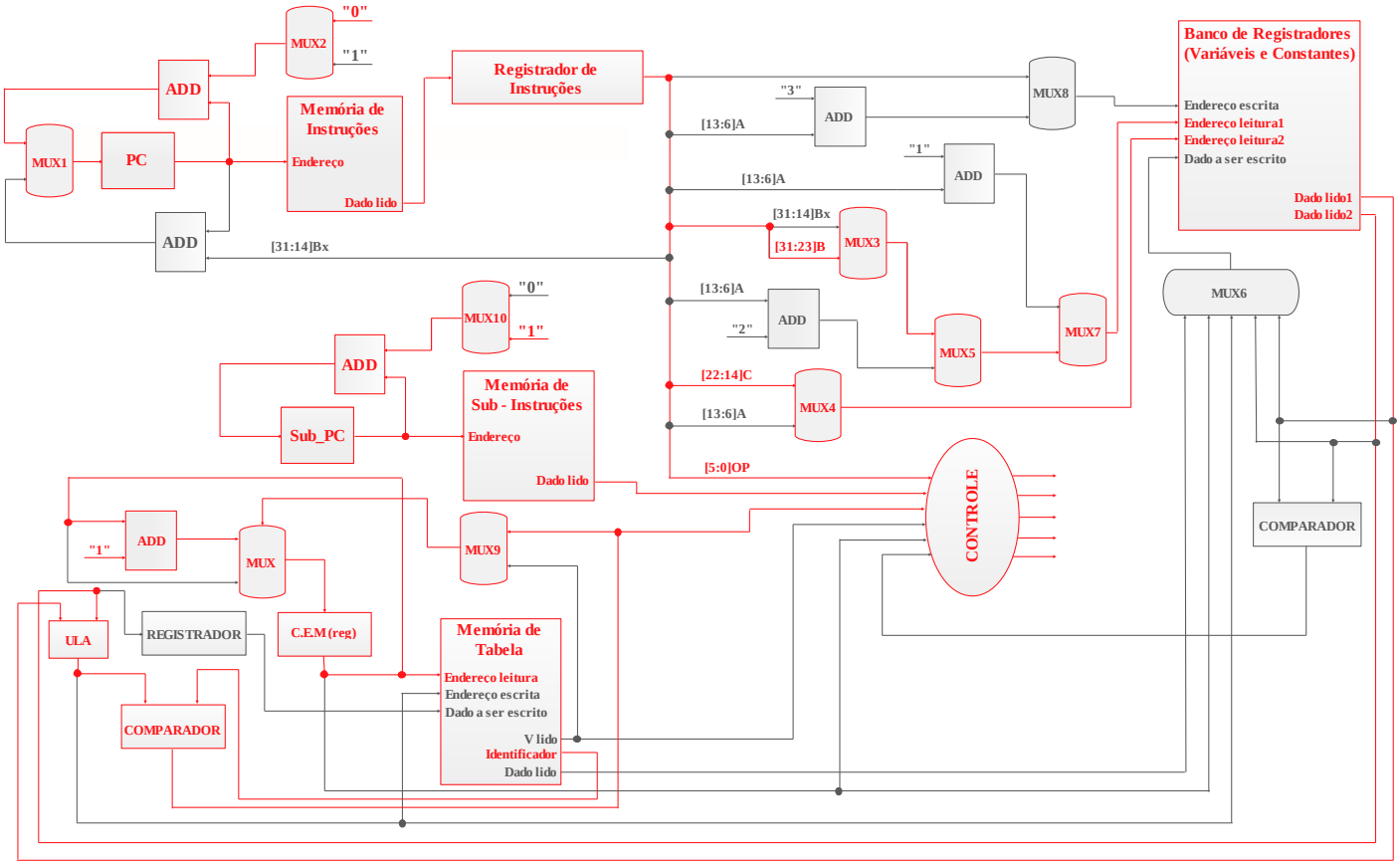


FIGURA 22 – Caminho de dados para a instrução GETTABLE.

Caso o teste mencionado anteriormente não seja verdadeiro, no quarto e último passo, conforme demonstrado na FIGURA 22, o C.E.M é incrementado e o teste, se o campo "Identificador" da Memória de Tabela, endereçado pelo valor do C.E.M, é igual ao resultado da operação de adição entre os valores presentes nos registradores, endereçados pelos campos [31:23]B e [22:14]C, é feito até que a condição seja satisfeita.

3.2.5. Instrução SETTABLE

A instrução SETTABLE copia o valor do registrador R(C) ou uma constante em um elemento de tabela. A tabela é referenciada pelo registrador R(A), enquanto o índice para a tabela é dado por RK(B), o qual pode ser o valor do registrador R(B) ou uma constante (MAN, 2006). A TABELA 11 apresenta um resumo desta instrução.

TABELA 11 - Tipo e Operação da Instrução SETTABLE.

Instrução	Tipo	Operação
SETTABLE	iABC	R(A)[RK(B)] = RK(C)

A FIGURA 23 apresenta o fluxograma para a implementação da instrução SETTABLE.

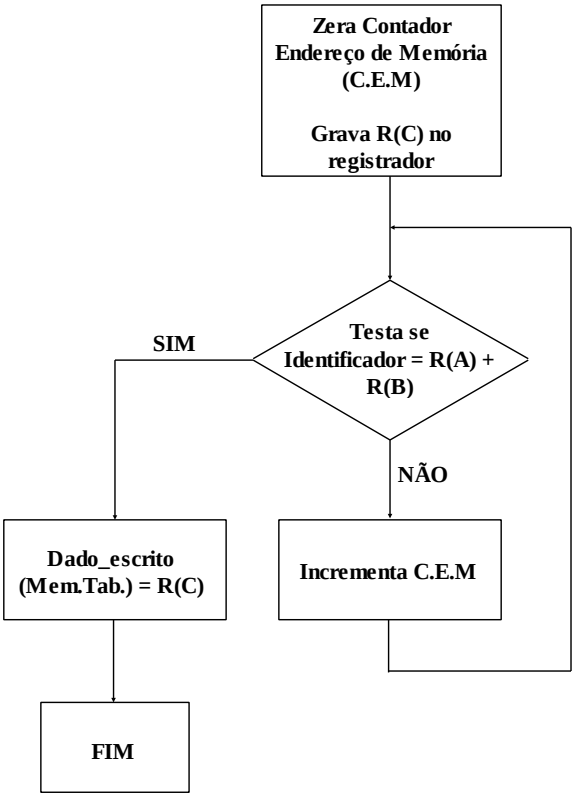


FIGURA 23 – Fluxograma da instrução SETTABLE.

Devido esta instrução não permitir a execução monociclo, optou-se por dividi-la em sub-instruções, sendo necessário também a implementação de uma Memória de Sub-intruções e um Sub_PC para a execução da instrução SETTABLE.

As FIGURAS 24, 25, 26 e 27 apresentam os caminhos de dados para a instrução SET-TABLE.

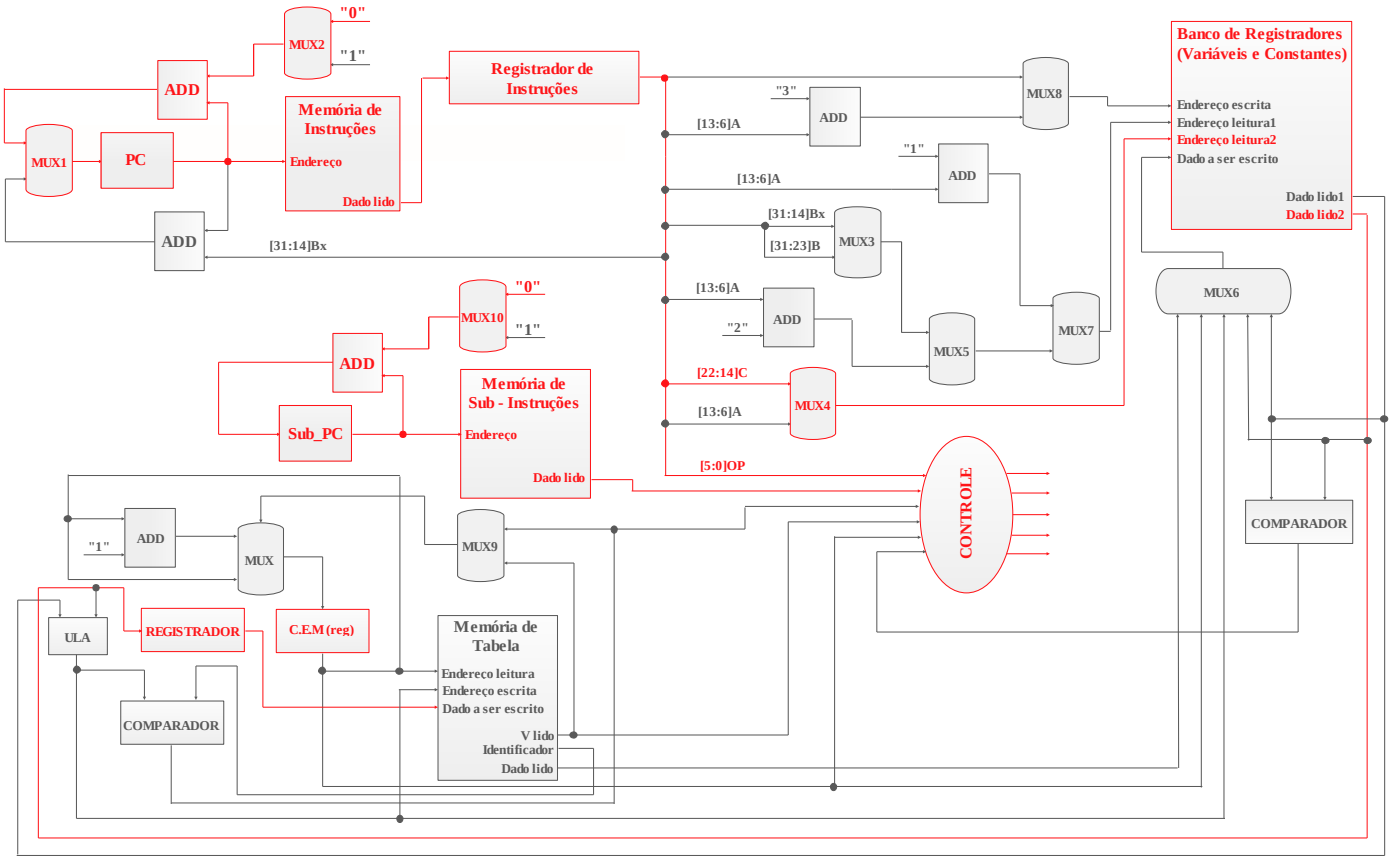


FIGURA 24 – Caminho de dados para a instrução SETTABLE.

O primeiro passo para a execução da instrução SETTABLE, como mostra a FIGURA 24, é zerar o C.E.M e escrever o valor contido em um registrador do Banco de Registradores, o qual é endereçado pelo campo [22:14]C, em um registrador temporário. Neste passo, o PC e o Sub_PC não são incrementados.

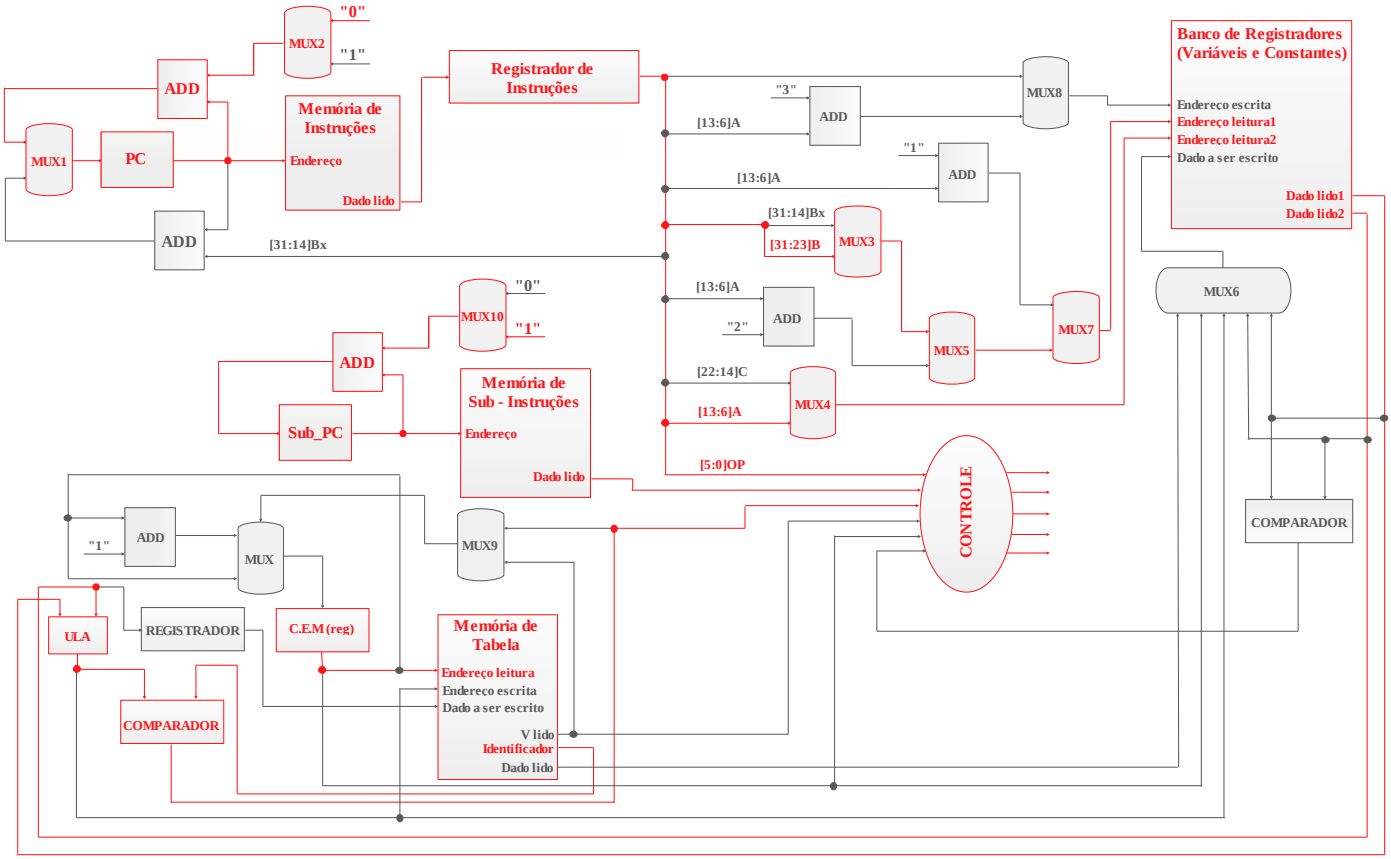


FIGURA 25 – Caminho de dados para a instrução SETTABLE.

Na segunda etapa, como mostra a FIGURA 25, é feito o teste se o campo “Identificador” da Memória de Tabela, endereçado pelo valor do C.E.M, é igual ao resultado da operação de adição entre os valores presentes em dois registradores, endereçados pelos campos [31:23]B e [13:6]A.

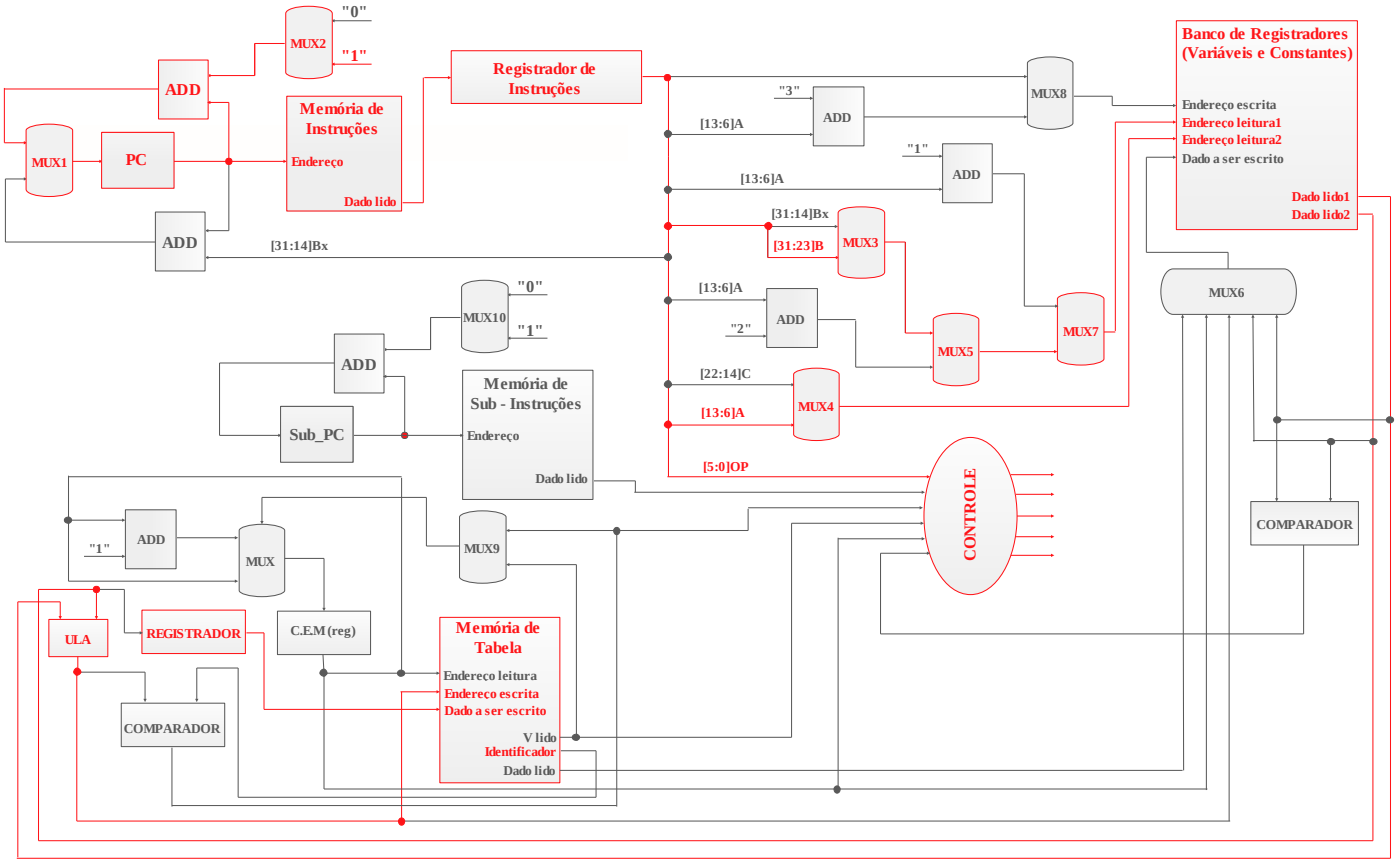


FIGURA 26 – Caminho de dados para a instrução SETTABLE.

Caso o teste mencionado anteriormente seja verdadeiro, no terceiro passo, conforme demonstrado na FIGURA 26, o valor escrito no registrador temporário é escrito na Memória de Tabela. O endereço de escrita da Memória de Tabela é dado pelo resultado da operação de adição entre os valores presentes em dois registradores, endereçados pelos campos [31:23]B e [13:6]A. Neste passo, o PC também é incrementado em ”1” com a finalidade de apontar para a próxima instrução a ser executada.

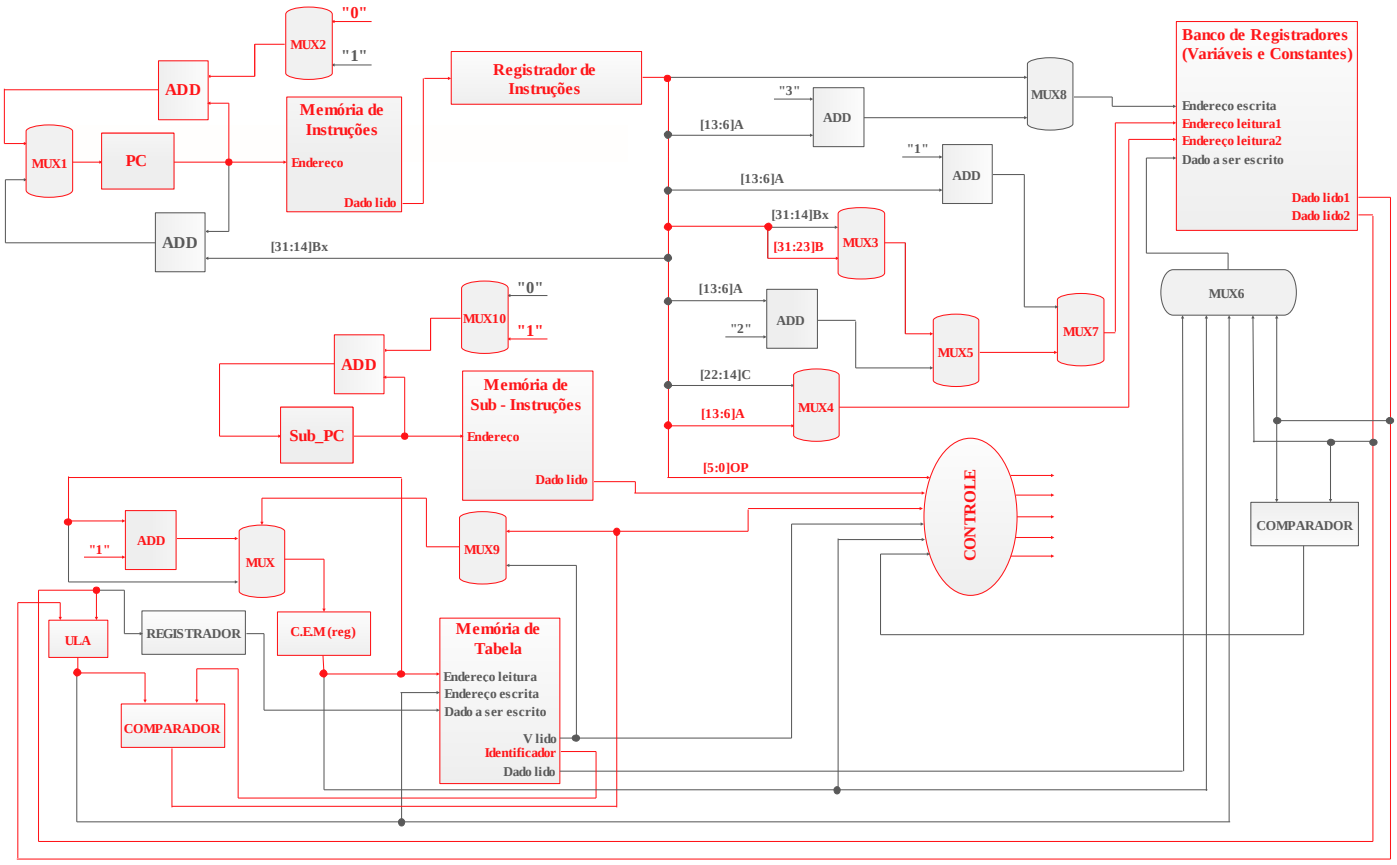


FIGURA 27 – Caminho de dados para a instrução SETTABLE.

Caso o teste mencionado anteriormente não seja verdadeiro, no quarto e último passo, conforme demonstrado na FIGURA 27, o C.E.M é incrementado e o teste, se o campo "Identificador" da Memória de Tabela, endereçado pelo valor do C.E.M, é igual ao resultado da operação de adição entre os valores presentes nos registradores, endereçados pelos campos [31:23]B e [14:6]A, é feito até que a condição seja satisfeita.

3.2.6. Instrução NEWTABLE

A instrução NEWTABLE cria uma nova tabela vazia no registrador R(A). Os campos B e C são os tamanhos das informações da parte array e da parte hash da tabela, respectivamente (MAN, 2006). A TABELA 12 apresenta um resumo desta instrução.

TABELA 12 - Tipo e Operação da Instrução NEWTABLE.

Instrução	Tipo	Operação
NEWTABLE	iABC	R(A) = {}(size = B,C)

A FIGURA 28 apresenta o fluxograma para implementação da instrução NEWTABLE.

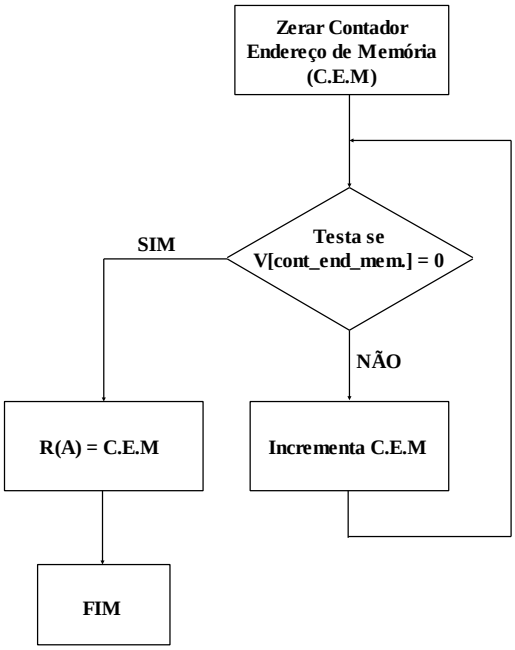


FIGURA 28 – Fluxograma da instrução NEWTABLE.

Devido esta instrução não permitir a execução monociclo, optou-se por dividi-la em sub-instruções, sendo necessário também a implementação de uma Memória de Sub-intruções e um Sub_PC para a execução da instrução NEWTABLE.

As FIGURAS 29, 30, 31 e 32 apresentam os caminhos de dados para a instrução NEW-TABLE.

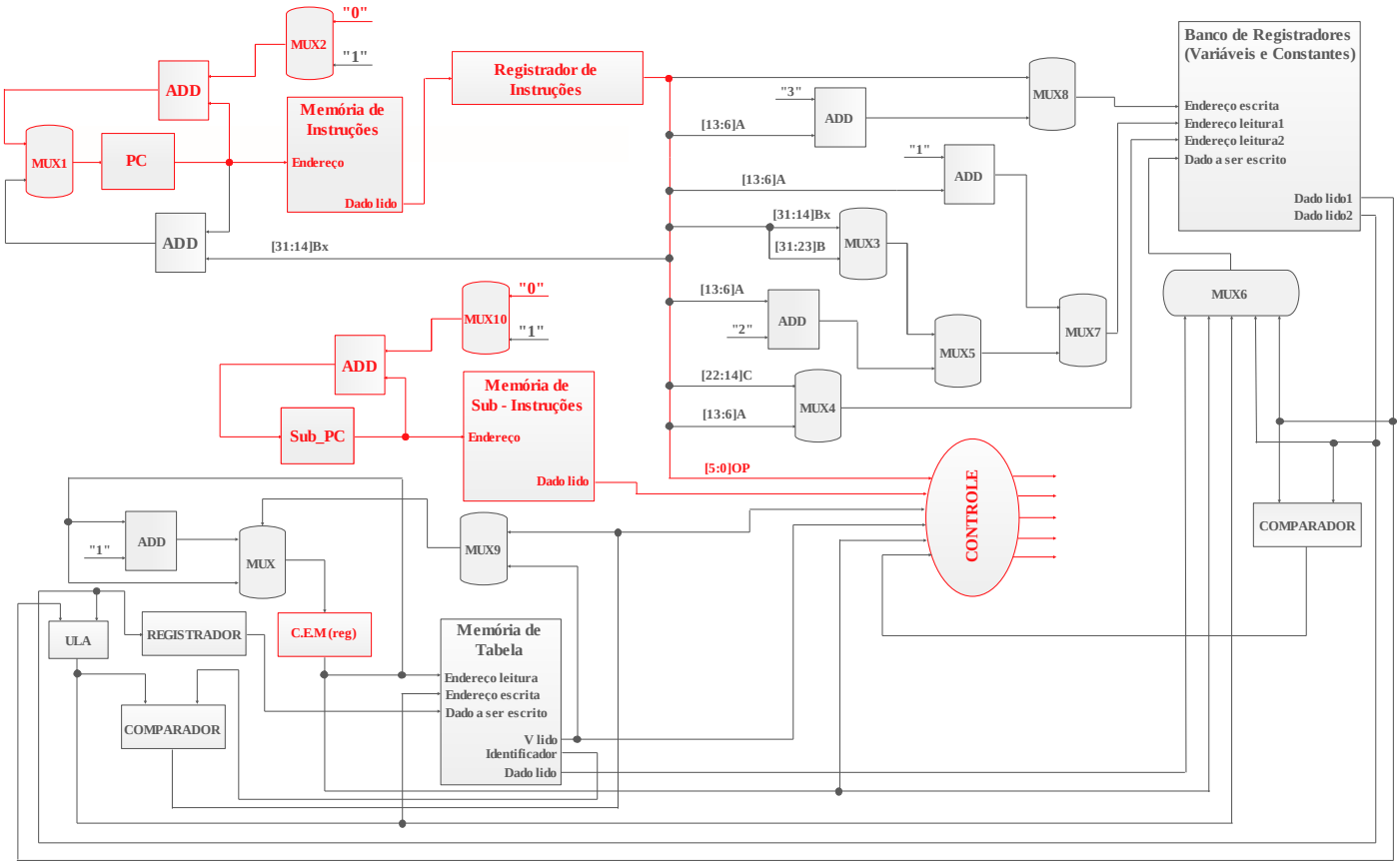


FIGURA 29 – Caminho de dados para a instrução NEWTABLE.

O primeiro passo para a execução da instrução NEWTABLE é zerar o C.E.M, como mostra a FIGURA 29.

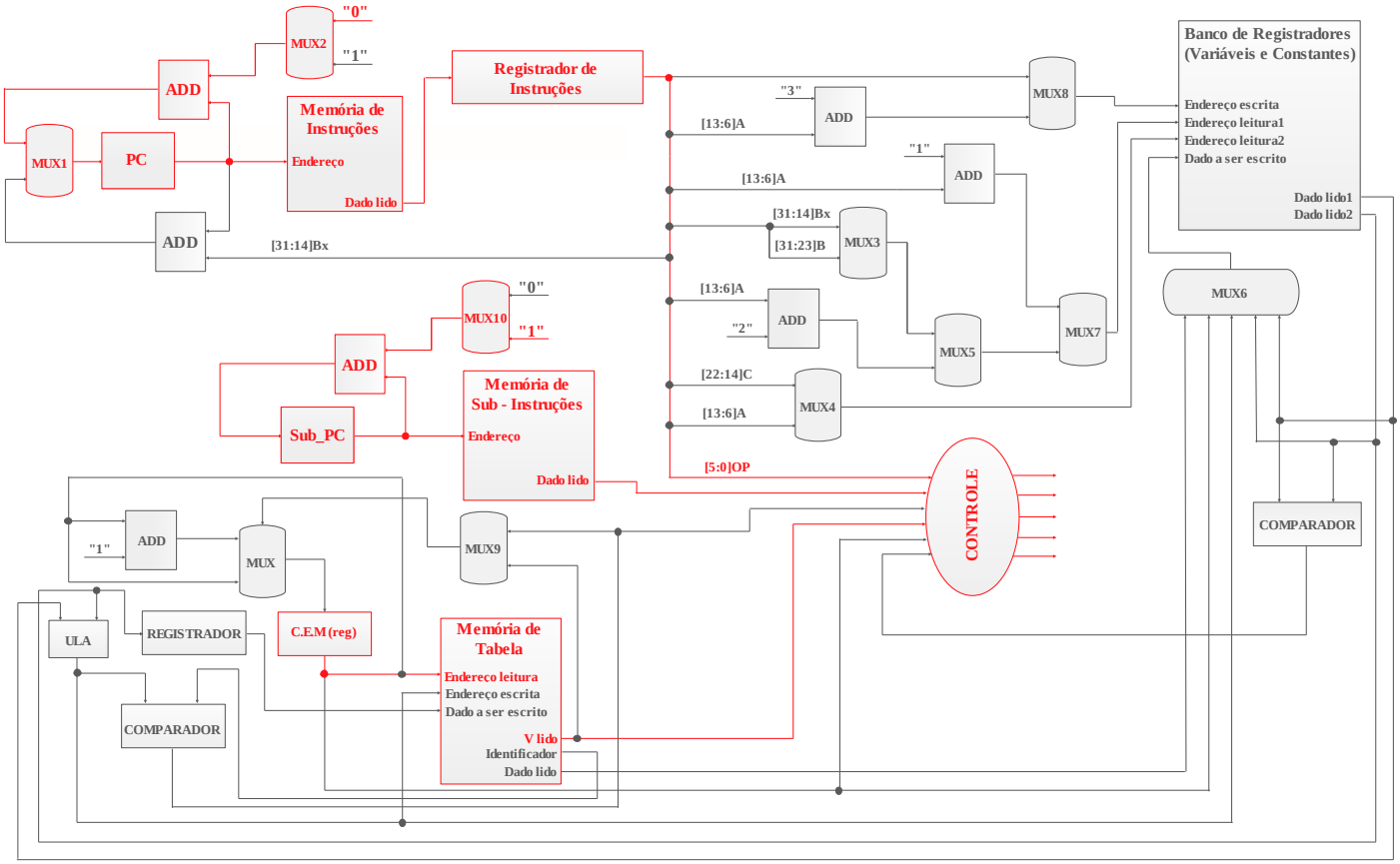


FIGURA 30 – Caminho de dados para a instrução NEWTABLE.

Na segunda etapa, como mostra a FIGURA 30, é feito o teste se o campo “V” da Memória de Tabela, endereçado pelo valor do C.E.M, é igual a “0” ou igual a “1”.

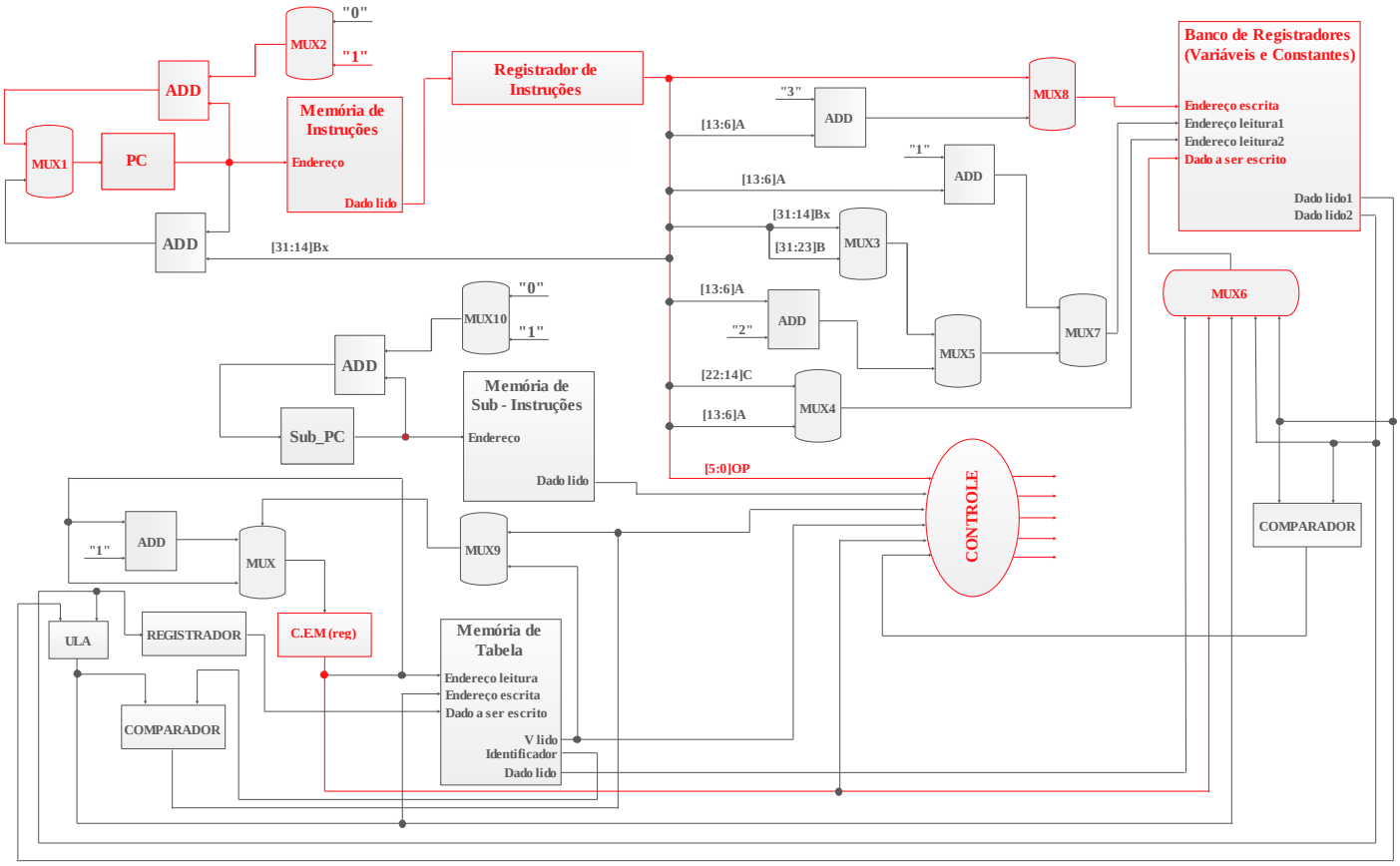


FIGURA 31 – Caminho de dados para a instrução NEWTABLE.

Caso o teste mencionado anteriormente seja verdadeiro, no terceiro passo, conforme demonstrado na FIGURA 31, o valor do C.E.M é escrito no registrador endereçado pelo campo [13:6]A. Neste passo, o PC também é incrementado em ”1” com a finalidade de apontar para a próxima instrução a ser executada.

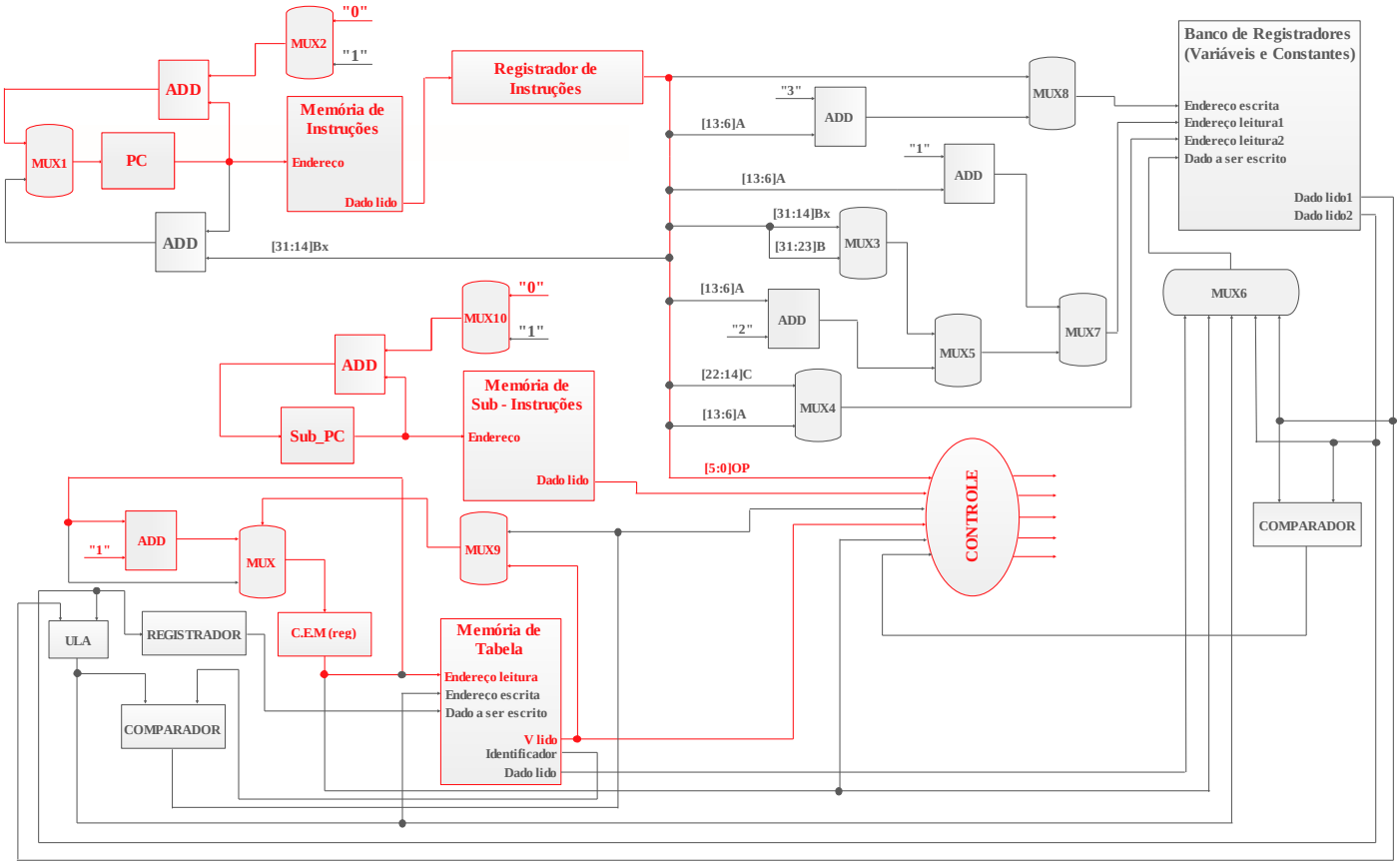


FIGURA 32 – Caminho de dados para a instrução NEWTABLE.

Caso o teste mencionado anteriormente não seja verdadeiro, no quarto e último passo, conforme demonstrado na FIGURA 32, o C.E.M é incrementado e o teste, se o campo “V” da Memória de Tabela, endereçado pelo valor do C.E.M, é igual a “0” ou igual a “1”, é feito até que a condição seja satisfeita.

3.2.7. Instrução FORPREP

A instrução FORPREP inicializa um laço numérico “for” (MAN, 2006). A TABELA 13 apresenta um resumo desta instrução.

TABELA 13 - Tipo e Operação da Instrução FORPREP.

Instrução	Tipo	Operação
FORPREP	iAsBx	R(A) = R(A) - R(A+2); PC = PC + sBx

A FIGURA 33 apresenta o caminho de dados para a instrução FORPREP.

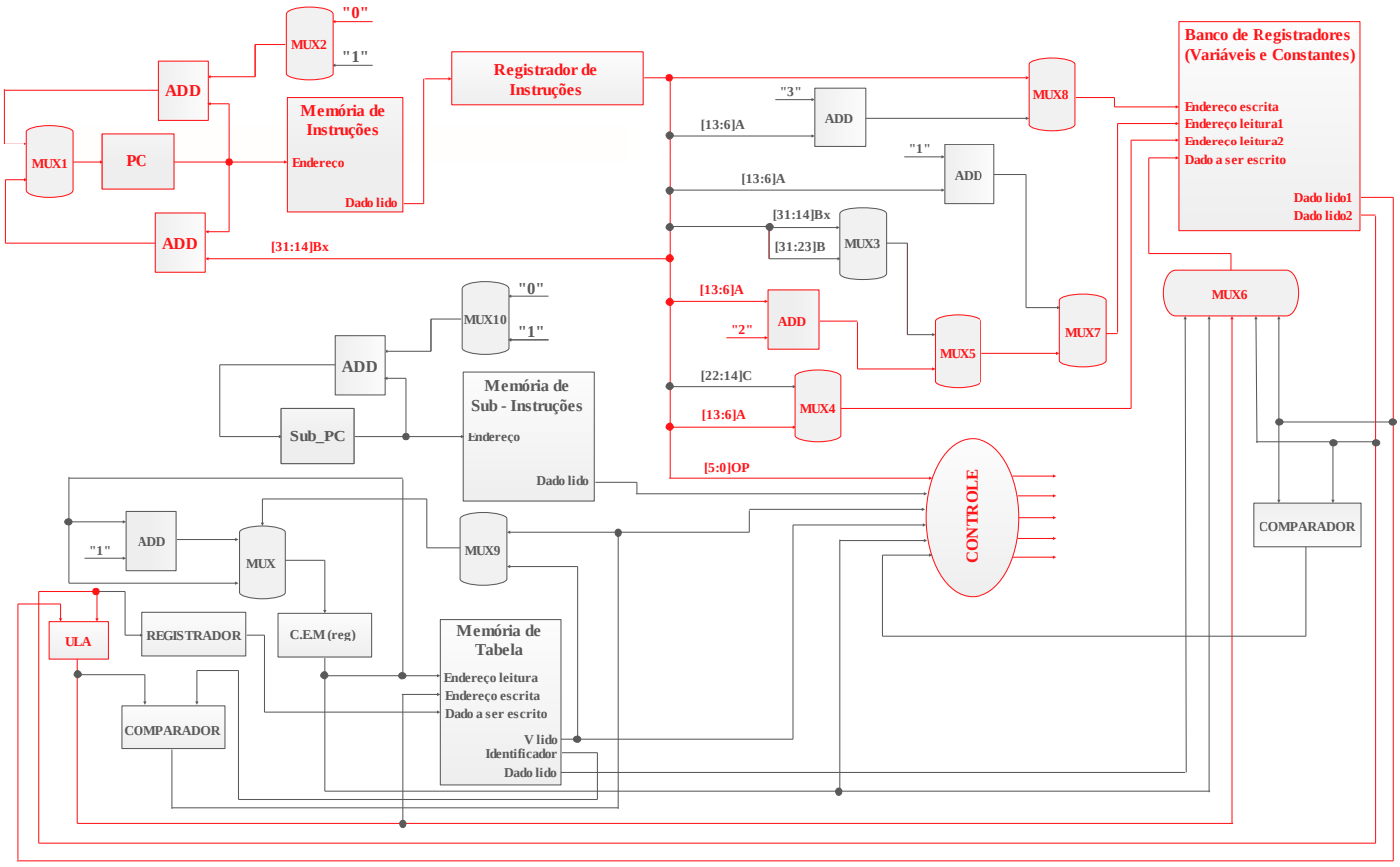


FIGURA 33 – Caminho de dados para a instrução FORPREP.

Para a execução da instrução FORPREP, conforme apresenta a FIGURA 33, o conteúdo presente em dois registradores, os quais são endereçados pelos campos [13:6]A e ([13:6]A + 2), são subtraídos e escritos em um registrador endereçado também pelo campo [13:6]A. Além disso, o PC é incrementado com o valor dado pelo campo [31:14]Bx, executando assim um salto incondicional.

3.2.8. Instrução FORLOOP

A instrução FORLOOP realiza uma iteração de um laço numérico “for” (MAN, 2006). A TABELA 14 apresenta um resumo desta instrução.

TABELA 14 - Tipo e Operação da Instrução FORLOOP.

Instrução	Tipo	Operação
FORLOOP	iAsBx	R(A) = R(A) + R(A+2); if R(A) <?= R(A+1) then { PC = PC + sBx; R(A+3) = R(A) }

A FIGURA 34 apresenta o fluxograma para implementação da instrução FORLOOP.

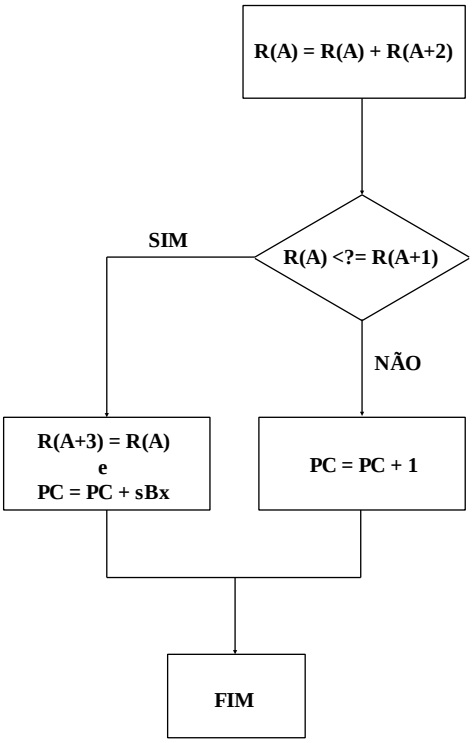


FIGURA 34 – Fluxograma da instrução FORLOOP.

Devido esta instrução não permitir a execução monociclo, optou-se por dividi-la em sub-instruções, sendo necessário também a implementação de uma Memória de Sub-intruções e um Sub_PC para a execução da instrução FORLOOP.

As FIGURAS 35, 36, 37 e 38 apresentam os caminhos de dados para a instrução FOR-LOOP.

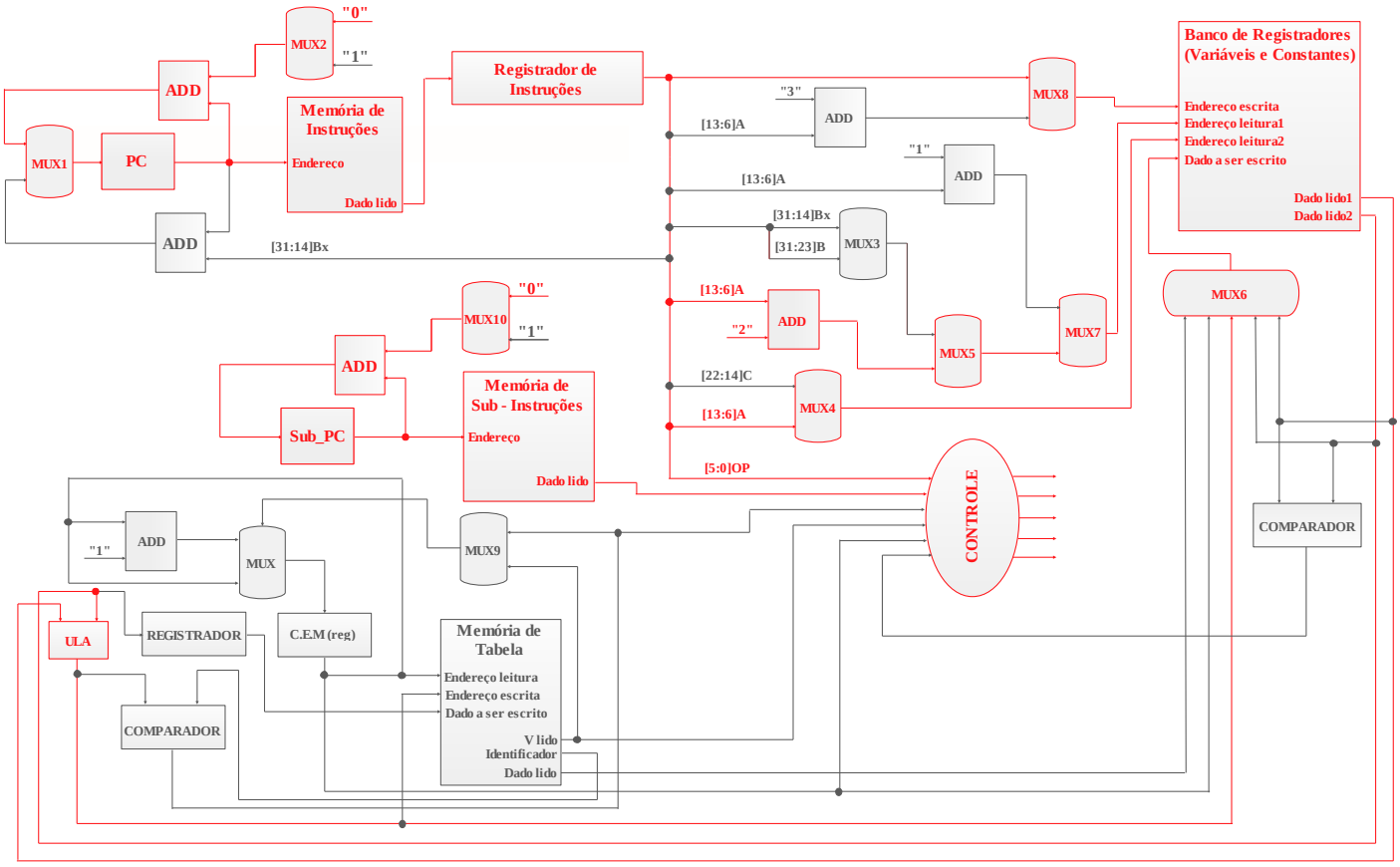


FIGURA 35 – Caminho de dados para a instrução FORLOOP.

Para a execução da instrução FORLOOP, como é mostrado na FIGURA 35, o primeiro passo consiste em fazer a soma dos conteúdos presentes em dois registradores, os quais são endereçados pelos campos [13:6]A e ([13:6]A + 2), e escrever o resultado em um registrador endereçado também pelo campo [13:6]A.

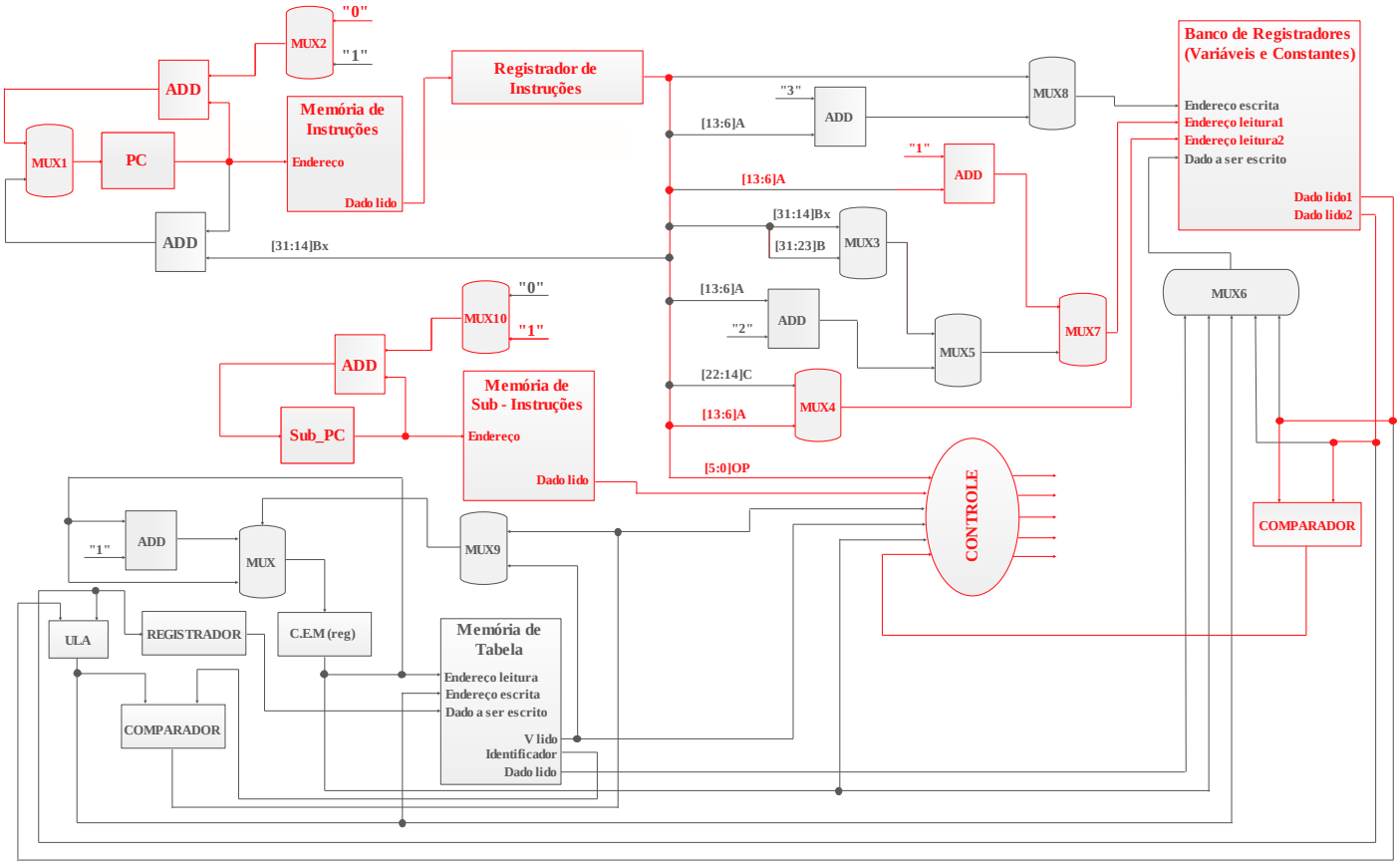


FIGURA 36 – Caminho de dados para a instrução FORLOOP.

Na segunda etapa, como mostra a FIGURA 36, é feito o teste se o o valor contido no registrador, endereçado pelo campo [13:6]A, é menor ou igual ao valor contido em outro registrador, endereçado pelo campo ([13:6]A + 2).

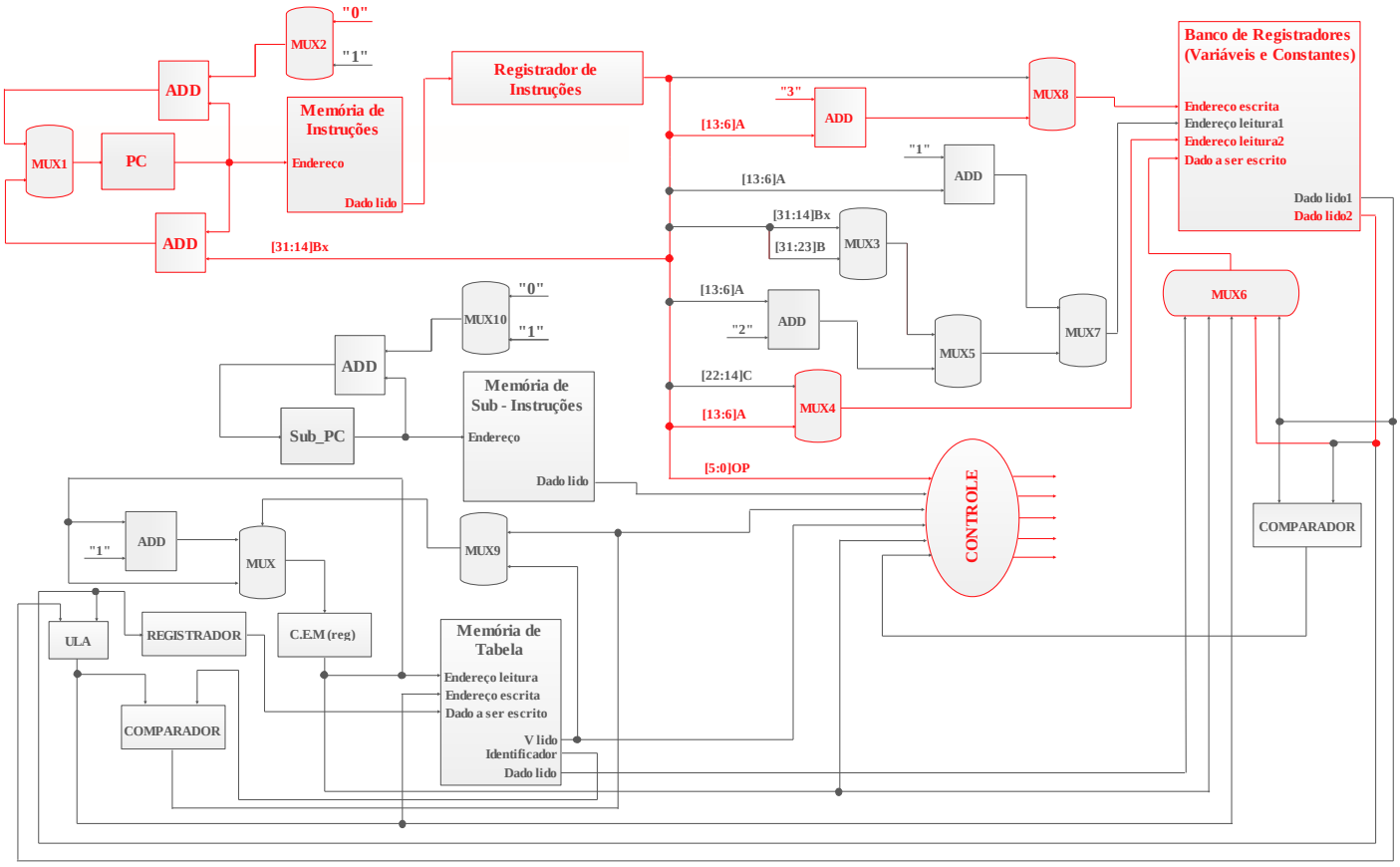


FIGURA 37 – Caminho de dados para a instrução FORLOOP.

Caso o teste mencionado anteriormente seja verdadeiro, no terceiro passo, conforme demonstrado na FIGURA 37, o valor do registrador, endereçado pelo campo [13:6]A, é escrito no registrador endereçado pelo campo ([13:6]A + 3). Além disso, o PC é incrementado com o valor dado pelo campo [31:14]Bx, executando assim um salto incondicional.

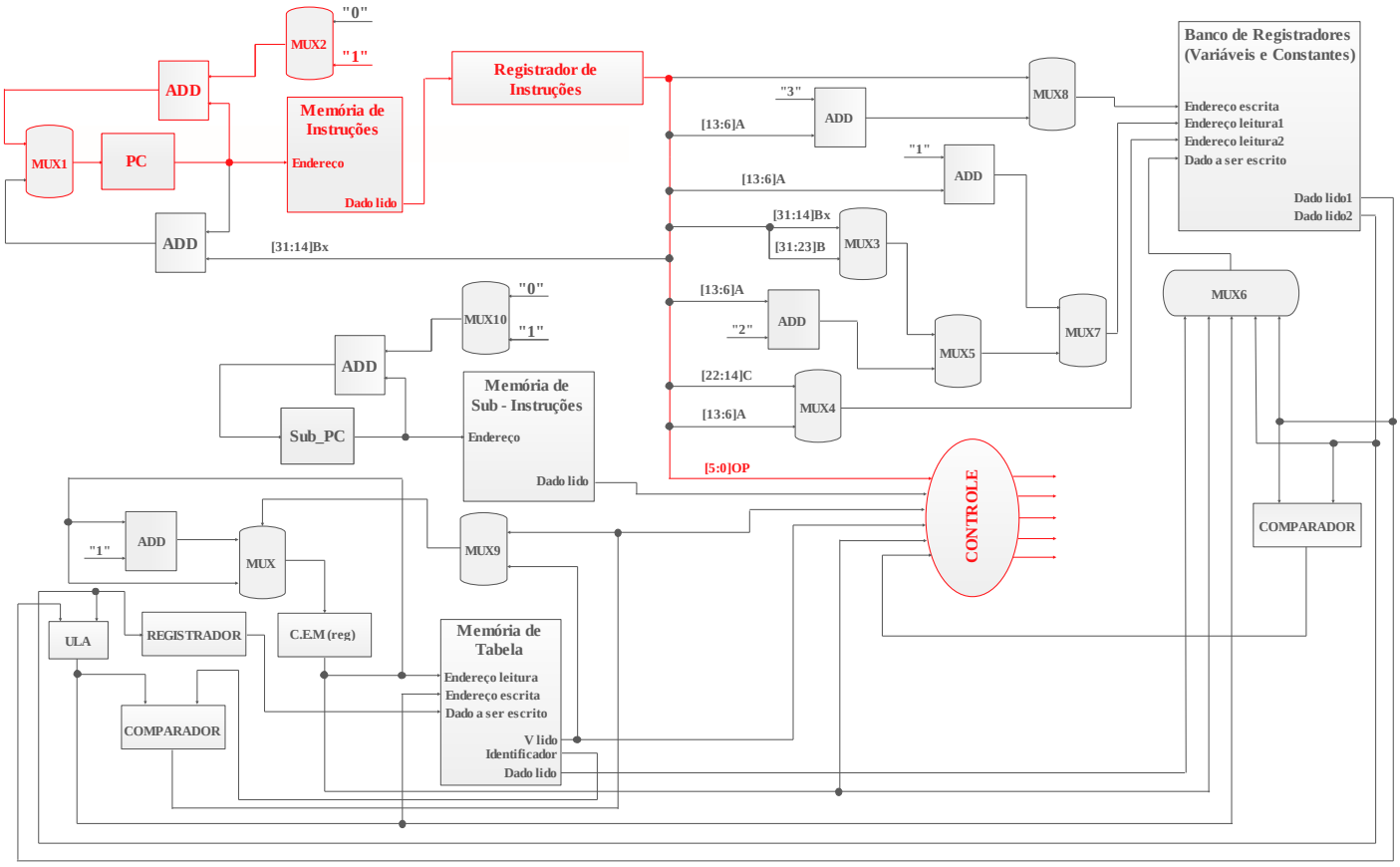


FIGURA 38 – Caminho de dados para a instrução FORLOOP.

Caso o teste mencionado anteriormente não seja verdadeiro, no quarto e último passo, conforme demonstrado na FIGURA 38, o PC é incrementado em ”1” com a finalidade de apontar para a próxima instrução a ser executada.

4 RESULTADOS

Após a proposta da arquitetura e sua descrição, torna-se necessário obter parâmetros para validá-la, como área utilizada, consumo de energia e frequência de operação. Normalmente estes parâmetros são obtidos via ferramentas de síntese, que transcrevem o código em HDL para um nível mais baixo de abstração, como o nível de portas lógicas.

Neste capítulo serão apresentados os resultados obtidos de síntese logica, consumo de energia e temporização da arquitetura proposta para o Processador Dedicado.

4.1.Síntese Lógica

Síntese é um processo automatizado de tradução e otimização onde uma especificação mais abstrata é transformada numa especificação menos abstrata. Por exemplo, a síntese lógica consiste na tradução de um código HDL em RTL em sua equivalente *netlist* de células digitais no nível de portas lógicas. Assim, o sistema é descrito como uma rede de portas e *flip-flops* e seu comportamento é especificado por equações lógicas (LEITE, 2006).

Os aplicativos de desenvolvimento de projetos que realizam esta tarefa de conversão de forma automática são conhecidos como ferramentas de síntese lógica (FIGUEIRÓ, 2011). O software utilizado para a síntese lógica do Processador foi o Quartus II 9.1 Web Edition da Altera Corporation. A síntese foi realizada para o dispositivo FPGA Cyclone II EP2C35F672C6.

Assim, o software Quartus II sintetiza a descrição do circuito, a qual foi realizada em SystemVerilog, para uma *netlist* que define os elementos lógicos (LE's) necessários e as conexões entre os elementos.

No fluxo da compilação, cuja interface está mostrada na FIGURA 39, são apresentadas as etapas de *Analisys* e *Synthesys*, *Fitter*, *Assembler* e *Classic Timing Analisys*. Segundo (FIGUEIRÓ, 2011), estas etapas são responsáveis pela síntese do projeto, posição e roteamento

dos LE's para o respectivo FPGA, geração de arquivos para programação do dispositivo e análises de temporização.

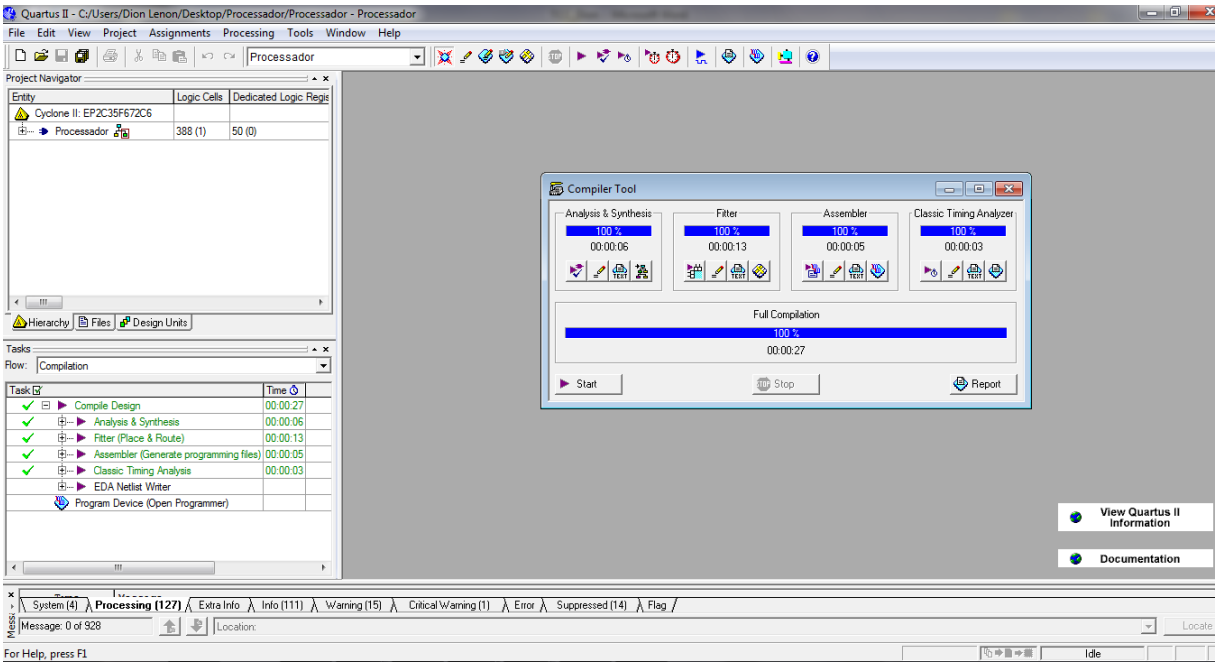


FIGURA 39 - Interface para o processo de compilação no Quartus II.

Após o término do processo de compilação, os resultados de síntese lógica são mostrados na própria interface para serem analisados. Na FIGURA 40 está mostrado o sumário com os resultados de síntese lógica para a entidade *Top-Level* do Processador no dispositivo Cyclone II EP2C35F672C6.

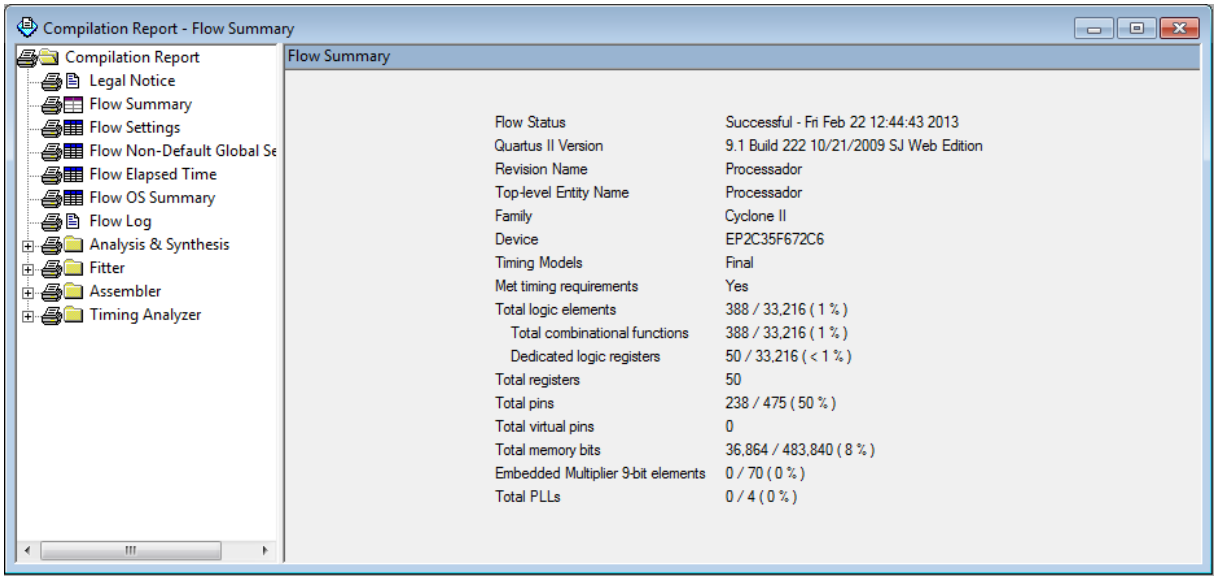


FIGURA 40- Sumários de resultados de síntese lógica do Quartus II.

Observa-se na FIGURA 40 que a Processador ocupou 388 elementos lógicos, não ultrapassando 1% da capacidade total do FPGA.

Além da síntese lógica, o Quartus II também dispõe de um visualizador gráfico do *netlist* sintetizado, onde se dá destaque ao *State Machine Viewer* e *RTL Viewer*. O primeiro representa os estados e as transições da máquina de estado e o segundo uma visão a nível RTL (*Register Transfer Level*) que representa todas as ligações em um nível mais baixo de abstração. Para a exemplificação destes visualizadores, está mostrada na FIGURA 41 a máquina de estados gerada do controle para as instruções *move* e *loadk*, na FIGURA 42 uma visão do RTL destas duas instruções.

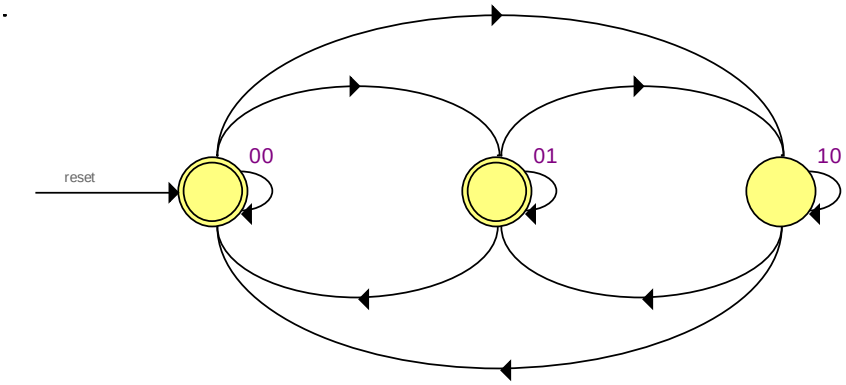


FIGURA 41 - Visualização gráfica da máquina de estados das instruções *move* e *loadk*.

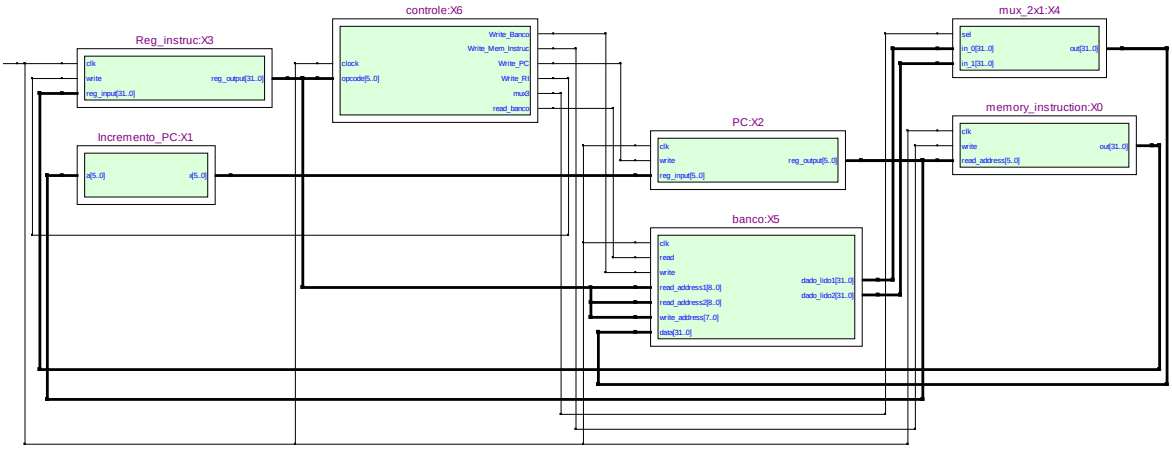


FIGURA 42 - Visualização gráfica do RTL das instruções *move* e *loadk*.

4.2. Consumo de energia e temporização

O Quartus II possui também ferramentas de estimativa de energia consumida e frequência de operação, denominadas *PowerPlay Power Analyzer Tool* e *Classic Timing Analyzer tool*, respectivamente, as quais serão analisadas a seguir.

4.2.1. Classic Timing Analyzer tool

No analisador de temporização *Timing Analyser tool*, cuja interface está mostrada na FIGURA 43, é estimado os tempos de percurso entre os sinais de entrada e saída do circuito, de forma a obter a frequência máxima de operação considerando as restrições temporais dos sinais.

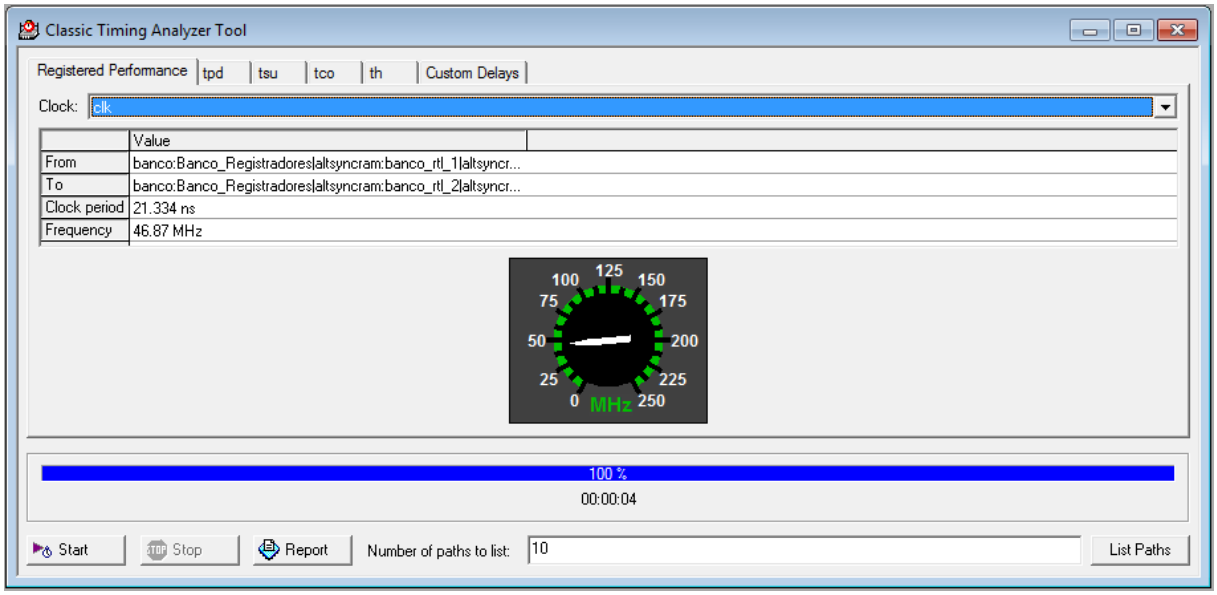


FIGURA 43 - Interface do analisador de temporização.

Com esta analise, observa-se na FIGURA 43 um período de *clock* de 21.334 *n*s e uma frequência de operação de 46.87 M*Hz* para o circuito sintetizado.

4.2.2. PowerPlay Power Analyzer

Com o uso da ferramenta de estimativa de potência *Power Analyzer* é possível obter uma estimativa do consumo de energia do circuito sintetizado.

Dentre os fatores que interferem na dissipação de potência em FPGA's citados em (HOLANDA, 2007), o número de elementos lógicos e arquiteturais tem uma forte influência sobre a dissipação de potência. Logo, quanto maior o número de elementos, maior será a tendência de dissipar energia.

Logo, os resultados de estimativa de potência estão mostrados na FIGURA 44.

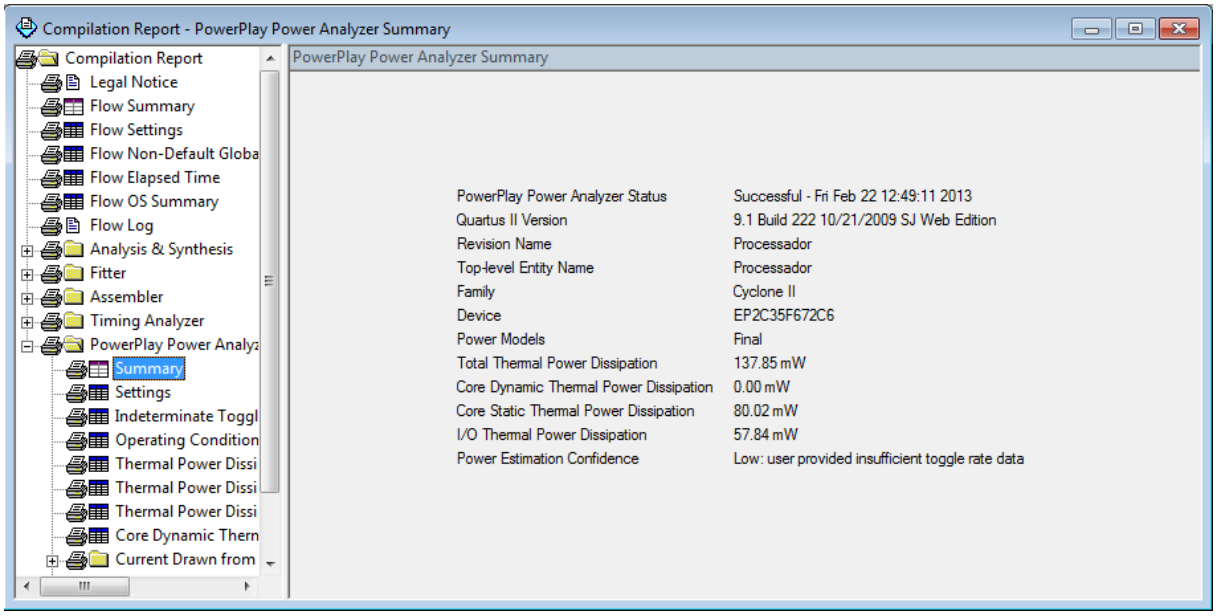


FIGURA 44 – Sumário de resultados da ferramenta *Power Play Power Analyzer.*

Observa-se na FIGURA 44, que o consumo total foi de 137.85 *m*W.

4.3.Simulação Temporal

O Programa Teste, que está mostrado no ALGORITMO 1, realiza a multiplicação de duas matrizes iguais *"M1"* e *"M2"* de tamanhos 3x3. Estas matrizes são apresentadas a seguir.

$$M1=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$M2=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Logo, o resultado da multiplicação destas matrizes é apresentado na matriz *"Mx"*, mostrada abaixo.

$$Mx = \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$$

Com isso, de maneira a testar a funcionalidade do Processador, serão apresentados os sultados de simulação temporal para a multiplicação das matrizes. Os resultados esperados para a simulação estão mostrados na matriz Mx. As FIGURAS 45, 46 e 47 apresentam os resultados da multiplicação das matrizes M1 e M2 para a simulação da arquitetura do Processador.

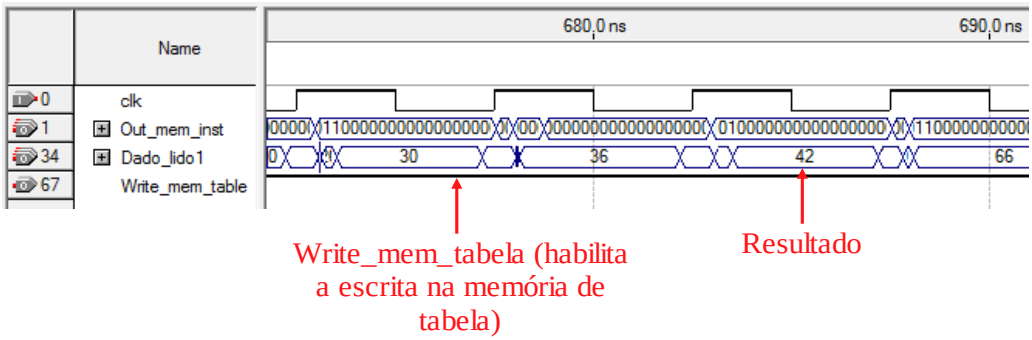


FIGURA 45 – Resultados da multiplicação das matrizes M1 e M2.

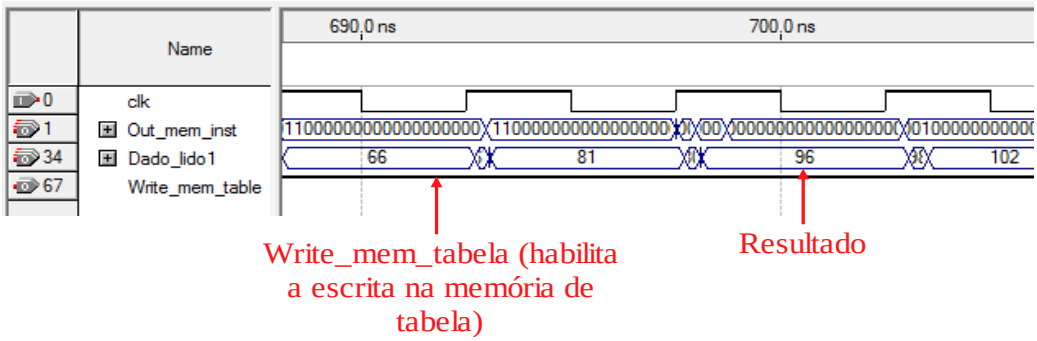


FIGURA 46 – Resultados da multiplicação das matrizes M1 e M2.

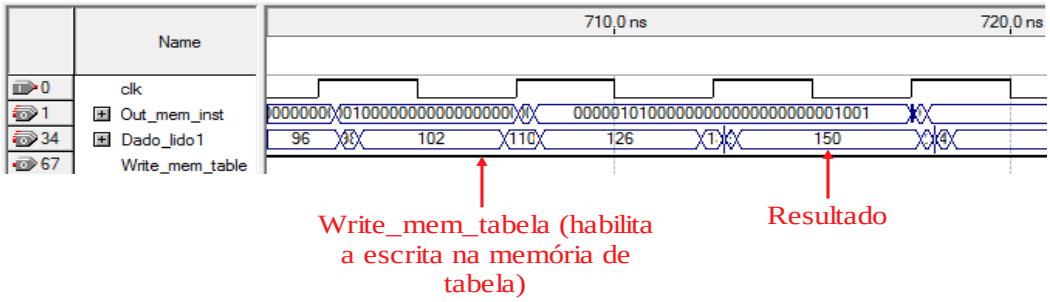


FIGURA 47 – Resultados da multiplicação das matrizes M1 e M2.

Nestas simulações o sinal *Write_mem_table* está em “1”, indicando que a memória de tabela está ativa para a escrita de dados em seu interior. A variável *Dado_lido1* apresenta os resultados da multiplicação das matrizes M1 e M2, resultados estes já esperados, conforme apresentados na matriz Mx.

CONSIDERAÇÕES FINAIS

Neste trabalho foi proposto uma alternativa de arquitetura de um Processador Dedicado para a execução de um conjunto de instruções de forma nativa da linguagem Lua 5.1, sem a necessidade de uso da máquina virtual Lua. A arquitetura foi descrita em SystemVerilog e sintetizada através do software Quartus II 9.1 Web Edition. Este software foi o responsável pela síntese, ou seja, pela transcrição da linguagem HDL em um nível RTL.

A arquitetura em questão passou pelas fases de concepção, de forma a ter uma visão sistêmica de sua funcionalidade, como também pela fase de definição da arquitetura, onde cada instrução e elemento utilizado foi detalhado quanto aos sinais de entrada e saída e interação no sistema.

Para o controle das unidades funcionais da arquitetura, utilizaram-se máquinas de estados, onde, em cada estado, sinais eram atribuídos de acordo com a entrada.

O Processador proposto neste trabalho ocupou 388 elementos lógicos de um total de 33.216, não ultrapassando 1% da capacidade total do FPGA para o dispositivo Cyclone II EP2C35F672C6. Quanto ao consumo de potência e frequência de operação, o processador apresentou um consumo total de 137.85 *m*W e uma frequência de 46.87 MHz para o circuito sintetizado. Esta frequência de operação corresponde a um período de clock de 21.334 ns.

Como não foram feitas comparações com trabalhos similares, como o picoJava (implementação em hardware da máquina virtual Java), a arquitetura do Processador foi validada em partes, ficando como trabalho futuro esta comparação. Além desta comparação, ficam ainda como trabalhos futuros, a comparação de ganho de se implementar em hardware a Máquina Virtual Lua, comparado com sua execução em software, e uma possível implementação do Processador utilizando técnicas de Pipeline, obtendo, com essa técnica, uma maior velocidade de execução das instruções, através da paralelização das etapas.

REFERÊNCIAS BIBLIOGRÁFICAS

CELES, Waldemar; FIGUEIREDO, Luiz Henrique de; IERUSALIMSCHY, Roberto. **A Linguagem Lua e suas Aplicações em Jogos**. Disponível em: <http://www.lua.org/doc/wjogos04.pdf>. Acesso em: 20 Set. 2011.

FIGUEIRÓ, Iuri Castro. **Arquitetura de uma unidade aritmética em ponto flutuante padrão IEEE754 32 bits**. Trabalho de conclusão de curso em Engenharia Elétrica, Universidade Federal do Pampa – Campus Alegrete, Alegrete/RS, 2011.

GOMES, Evandro Luís Brandão. **Introdução aos Sistemas Computacionais – Eletrônica Digital II, Capítulo 1.** Disponível em: <http://pt.scribd.com/doc/91124692/36/%E2%80%93-O-Contador-de-Programa-PC-%E2%80%93-Program-Counter>. Acesso em: 22 Dez. 2012.

HOLANDA, José Arnoldo Mascagni de, **Projeto de um estimador de potência para o processador NIOS II da Altera**. Dissertação (Ciências de Computação e Matemática Computacional), Instituto de Ciências Matemáticas e de Computação – ICMC – USP, São Carlos/SP, 2007.

LEITE, Risério Dourado, **Implementação de um Módulo Controlador de Vídeo na forma "Intellectual Property"**. Dissertação (Engenharia Elétrica), Universidade Federal de Minas Gerais, Belo Horizonte/MG, 2006.

LUA, **Conceitos Básicos e API C**. In: Pontifícia Universidade Católica do Rio de Janeiro. Agosto de 2008. Disponível em: < http://www.keplerproject.org/docs/apostila_lua_2008.pdf >. Acesso em: 20 Set. 2011.

MACHADO, Alex de Magalhães. **Um estudo sobre possíveis técnicas de otimização de código para bytecode Lua**. Trabalho de conclusão de em Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis/SC, 2009.

MAN, Kein-Hong. **A No-Frills Introduction to Lua 5.1 VM Instructions.** Disponível em: < http://luaforge.net/docman >. Acesso em: 02 Out. 2011.

MURDOCA, Miles; VINCENT, Heuring. **Introdução a Arquitetura de Computadores**. Campus, 2001.

OLEQUES, Ernesto Schimidt, **Sistemas Embarcados**. Disponível em: < http://pt.scribd.com/doc/18503391/SISTEMAS-EMBARCADOS>. Acesso em: 08 jan. 2013.

PATTERSON, DAVID A.; HENNESY, Jonh L. **Computer Architecture: A Quantitative Approach.** Morgan Kaufmann, 2003.

SANTOS, Danillo Moura, **PROJETO DE SISTEMAS EMBARCADOS: Um estudo de caso baseado em microcontrolador e seguindo AOSD.** Monografia em Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis/SC, 2006.

SEDGEWICK, Robert. **Algorithms in C++**. Addison – Wesley, 1998.

SOCAL, Francisco Roberto Peixoto. **Ferramenta para Desenvolvimento de Sistemas Embarcados Utilizando Linguagem de Alto Nível.** Projeto de Diplomação em Engenharia de Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre/RS, 2003.

STALLINGS, William – **Arquitetura e Organização de Computadores – Projeto para Desempenho**. Prentice Hall, 2002.

SUTHERLAND S., DAVIDMAN S, FLAKE P, **SystemVerilog For Design – A Guide to Using SystemVerilog for Hardware Design and Modeling**. Kluwe Academic Plubishers, 2004.

UYEMURA, Jonh P., **Sistemas digitais – Uma abordagem integrada.** Thomson, 2000.

VAHID, Frank – **Sistemas digitais – Projeto, Otimização e HDLs**. Bookman, 2008.

YANAGISAWA, Rodrigo. **A Linguagem Lua**. Disponível em: <www.geomatica.eng.uerj.br>. Acesso em: 25. Set. 2011.

APÊNDICE A. EXEMPLO DE COMPILAÇÃO DE CÓDIGO DESCRITO EM LINGUAGEM LUA

O ALGORITMO 1 mostra a descrição em linguagem Lua do Programa teste, o qual realiza a operação de multiplicação de duas matrizes.

```
1  -- Programa que multiplica duas matrizes iguais 3x3
2
3  local n = 1
4
5  local size = 3
6
7  local rows = size
8  local cols = size
9  local count = 1
10  local mx = {}
11
12  -- preenche a matriz com números em sequência
13
14  for i=0,(rows - 1) do
15            local row = {}
16            for j=0,(cols - 1) do
17            row[j] = count
18            count = count + 1
19            end
20            mx[i] = row
21  end
22
23  -- matrizes m1 e m2 são iguais
24
25   local m1 = mx
26   local m2 = mx
27
28  local m3 = {}
29
30  -- realiza a multiplicação das matrizes m1 e m2
31
32  for i=0,(rows-1) do
```

```
33     m3[i] = {}
34     for j=0,(cols-1) do
35        local rowj = 0
36        for k=0,(cols-1) do
37           rowj = rowj + m1[i][k] * m2[k][j]
38        end
39        m3[i][j] = rowj
40     end
41  end
```

ALGORITMO 1 - Código em linguagem Lua do Programa teste.

Após a compilação do código, que é mostrado no ALGORITMO 1, através do software ChunkSpy gerou-se uma listagem detalhada que é apresentada abaixo. Esta listagem apresenta, além de outras informações, as instruções Lua necessárias para a execução da multiplicação das matrizes.

```
Pos   Hex Data          Description or Code
-------------------------------------------------------------------------
0000                    ** source chunk: matrix_mult.lua
                        ** global header start **
0000  1B4C7561           header signature: "\27Lua"
0004  51                 version (major:minor hex digits)
005  00                  format (0=official)
0006  01                 endianness (1=little endian)
0007  04                 size of int (bytes)
0008  04                 size of size_t (bytes)
0009  04                 size of Instruction (bytes)
000A  08                 size of number (bytes)
000B  00                 integral (1=integral)
                        * number type: double
                        * x86 standard (32-bit, little endian, doubles)
                        ** global header end **

000C                     ** function [0] definition (level 1)
                         ** start of function **
000C  10000000           string size (16)
0010  6D61747269785F6D+   "matrix_m"
```

```
0018  756C742E6C756100      "ult.lua\0"
                            source name: matrix_mult.lua
0020  00000000              line defined (0)
0024  00000000              last line defined (0)
0028  00                    nups (0)
0029  00                    numparams (0)
002A  02                    is_vararg (2)
002B  18                    maxstacksize (24)
                            * code:
002C  32000000              sizecode (50)
0030  01000000              [01] loadk      0   0       ; 1
0034  41400000              [02] loadk      1   1       ; 3
0038  80008000              [03] move       2   1
003C  C0008000              [04] move       3   1
0040  01010000              [05] loadk      4   0       ; 1
0044  4A010000              [06] newtable   5   0   0   ; array=0, hash=0
0048  81810000              [07] loadk      6   2       ; 0
004C  CD014001                  [08] sub        7   2   256 ; 1
0050  01020000              [09] loadk      8   0       ; 1
0054  A0010280              [10] forprep    6   9       ; to [20]
0058  8A020000              [11] newtable   10  0   0   ; array=0, hash=0
005C  C1820000              [12] loadk      11  2       ; 0
0060  0D03C001              [13] sub        12  3   256 ; 1
0064  41030000              [14] loadk      13  0       ; 1
0068  E0420080              [15] forprep    11  2       ; to [18]
006C  89020107              [16] settable   10  14  4
0070  0C014002              [17] add        4   4   256 ; 1
0074  DF02FF7F              [18] forloop    11  -3      ; to [16] if loop
0078  49818204              [19] settable   5   9   10
007C  9F41FD7F              [20] forloop    6   -10     ; to [11] if loop
0080  80018002              [21] move       6   5
0084  C0018002              [22] move       7   5
0088  0A020000              [23] newtable   8   0   0   ; array=0, hash=0
008C  41820000              [24] loadk      9   2       ; 0
0090  8D024001              [25] sub        10  2   256 ; 1
0094  C1020000              [26] loadk      11  0       ; 1
0098  60020580              [27] forprep    9   21      ; to [49]
```

```
009C  4A030000          [28] newtable   13  0   0    ; array=0, hash=0
00A0  09420306          [29] settable   8   12  13
00A4  41830000          [30] loadk      13  2        ; 0
00A8  8D03C001          [31] sub        14  3   256  ; 1
00AC  C1030000          [32] loadk      15  0        ; 1
00B0  60430380          [33] forprep    13  14       ; to [48]
00B4  41840000          [34] loadk      17  2        ; 0
00B8  81840000          [35] loadk      18  2        ; 0
00BC  CD04C001          [36] sub        19  3   256  ; 1
00C0  01050000          [37] loadk      20  0        ; 1
00C4  A0440180          [38] forprep    18  6        ; to [45]
00C8  86050303          [39] gettable   22  6   12
00CC  8645050B          [40] gettable   22  22  21
00D0  C6458503          [41] gettable   23  7   21
00D4  C605840B          [42] gettable   23  23  16
00D8  8EC5050B          [43] mul        22  22  23
00DC  4C848508          [44] add        17  17  22
00E0  9F04FE7F          [45] forloop    18  -7       ; to [39] if loop
00E4  86040304          [46] gettable   18  8   12
00E8  89440408          [47] settable   18  16  17
00EC  5F03FC7F          [48] forloop    13  -15      ; to [34] if loop
00F0  5F42FA7F          [49] forloop    9   -22      ; to [28] if loop
00F4  1E008000          [50] return     0   1
                        * constants:
00F8  03000000          sizek (3)
00FC  03                const type 3
00FD  000000000000F03F  const [0]: (1)
0105  03                const type 3
0106  0000000000000840  const [1]: (3)
010E  03                const type 3
010F  0000000000000000  const [2]: (0)
                        * functions:
0117  00000000          sizep (0)
                        * lines:
011B  32000000          sizelineinfo (50)
                        [pc] (line)
011F  08000000          [01] (8)
```

```
0123  0A000000                [02] (10)
0127  0C000000                [03] (12)
012B  0D000000               [04] (13)
012F  0E000000               [05] (14)
...
01DF  23000000               [49] (35)
01E3  2C000000               [50] (44)
                             * locals:
01E7  1F000000               sizelocvars (31)
01EB  02000000               string size (2)
01EF  6E00                   "n\0"
                              local [0]: n
01F1  01000000               startpc (1)
01F5  31000000               endpc   (49)
01F9  05000000               string size (5)
01FD  73697A6500             "size\0"
                              local [1]: size
0202  02000000               startpc (2)
0206  31000000               endpc   (49)
020A  05000000               string size (5)
020E  726F777300             "rows\0"
                              local [2]: rows
0213  03000000               startpc (3)
0217  31000000               endpc   (49)
021B  05000000               string size (5)
021F  636F6C7300             "cols\0"
                              local [3]: cols
0224  04000000               startpc (4)
0228  31000000               endpc   (49)
022C  06000000               string size (6)
0230  636F756E7400           "count\0"
                              local [4]: count
0236  05000000               startpc (5)
023A  31000000               endpc   (49)
023E  03000000               string size (3)
0242  6D7800                 "mx\0"
                              local [5]: mx
```

```
0245  06000000                startpc (6)
0249  31000000                endpc   (49)
024D  0C000000                string size (12)
0251  28666F7220696E64+       "(for ind"
0259  65782900                "ex)\0"
                               local [6]: (for index)
025D  09000000                startpc (9)
0261  14000000                endpc   (20)
0265  0C000000                string size (12)
0269  28666F72206C696D+       "(for lim"
0271  69742900                 "it)\0"
                               local [7]: (for limit)
0275  09000000                startpc (9)
0279  14000000                endpc   (20)
027D  0B000000                string size (11)
0281  28666F7220737465+       "(for ste"
0289  702900                  "p)\0"
                               local [8]: (for step)
028C  09000000                startpc (9)
0290  14000000                endpc   (20)
0294  02000000                string size (2)
0298  6900                    "i\0"
                               local [9]: i
029A  0A000000                startpc (10)
029E  13000000                endpc   (19)
02A2  04000000                string size (4)
02A6  726F7700                "row\0"
                               local [10]: row
02AA  0B000000                startpc (11)
02AE  13000000                endpc   (19)
02B2  0C000000                string size (12)
02B6  28666F7220696E64+       "(for ind"
02BE  65782900                "ex)\0"
                               local [11]: (for index)
02C2  0E000000                startpc (14)
02C6  12000000                endpc   (18)
02CA  0C000000                string size (12)
```

```
02CE  28666F72206C696D+      "(for lim"
02D6  69742900               "it)\0"
                              local [12]: (for limit)
02DA  0E000000               startpc (14)
02DE  12000000               endpc   (18)
02E2  0B000000               string size (11)
02E6  28666F7220737465+      "(for ste"
02EE  702900                 "p)\0"
                              local [13]: (for step)
02F1  0E000000               startpc (14)
02F5  12000000               endpc   (18)
02F9  02000000               string size (2)
02FD  6A00                   "j\0"
                              local [14]: j
02FF  0F000000               startpc (15)
0303  11000000               endpc   (17)
0307  03000000               string size (3)
030B  6D3100                 "m1\0"
                              local [15]: m1
030E  15000000               startpc (21)
0312  31000000               endpc   (49)
0316  03000000               string size (3)
031A  6D3200                 "m2\0"
                              local [16]: m2
031D  16000000               startpc (22)
0321  31000000               endpc   (49)
0325  03000000               string size (3)
0329  6D3300                 "m3\0"
                              local [17]: m3
032C  17000000               startpc (23)
0330  31000000               endpc   (49)
0334  0C000000               string size (12)
0338  28666F7220696E64+      "(for ind"
0340  65782900               "ex)\0"
                              local [18]: (for index)
0344  1A000000               startpc (26)
0348  31000000               endpc   (49)
```

```
034C  0C000000                string size (12)
0350  28666F72206C696D+      "(for lim"
0358  69742900                "it)\0"
                               local [19]: (for limit)
035C  1A000000                startpc (26)
0360  31000000                endpc   (49)
0364  0B000000                string size (11)
0368  28666F7220737465+       "(for ste"
0370  702900                  "p)\0"
                               local [20]: (for step)
0373  1A000000                startpc (26)
0377  31000000                endpc   (49)
037B  02000000                string size (2)
037F  6900                    "i\0"
                               local [21]: i
0381  1B000000                startpc (27)
0385  30000000                endpc   (48)
0389  0C000000                string size (12)
038D  28666F7220696E64+       "(for ind"
0395  65782900                "ex)\0"
                               local [22]: (for index)
0399  20000000                startpc (32)
039D  30000000                endpc   (48)
03A1  0C000000                string size (12)
03A5  28666F72206C696D+       "(for lim"
03AD  69742900                "it)\0"
                               local [23]: (for limit)
03B1  20000000                startpc (32)
03B5  30000000                endpc   (48)
03B9  0B000000                string size (11)
03BD  28666F7220737465+       "(for ste"
03C5  702900                  "p)\0"
                               local [24]: (for step)
03C8  20000000                startpc (32)
03CC  30000000                endpc   (48)
03D0  02000000                string size (2)
03D4  6A00                    "j\0"
```

```
                                local [25]: j
03D6  21000000                  startpc (33)
03DA  2F000000                  endpc   (47)
03DE  05000000                  string size (5)
03E2  726F776A00                "rowj\0"
                                local [26]: rowj
03E7  22000000                  startpc (34)
03EB  2F000000                  endpc   (47)
03EF  0C000000                  string size (12)
03F3  28666F7220696E64+         "(for ind"
03FB  65782900                  "ex)\0"
                                local [27]: (for index)
03FF  25000000                  startpc (37)
0403  2D000000                  endpc   (45)
0407  0C000000                  string size (12)
040B  28666F72206C696D+         "(for lim"
0413  69742900                  "it)\0"
                                local [28]: (for limit)
0417  25000000                  startpc (37)
041B  2D000000                  endpc   (45)
041F  0B000000                  string size (11)
0423  28666F7220737465+         "(for ste"
042B  702900                    "p)\0"
                                local [29]: (for step)
042E  25000000                  startpc (37)
0432  2D000000                  endpc   (45)
0436  02000000                  string size (2)
043A  6B00                      "k\0"
                                local [30]: k
043C  26000000                  startpc (38)
0440  2C000000                  endpc   (44)
                                * upvalues:
0444  00000000                  sizeupvalues (0)
                                ** end of function **

0448                            ** end of chunk **
```