UNIVERSIDADE FEDERAL DO PAMPA

ALIAN MOREIRA ENGROFF

PAMPIUM I - PROJETO E IMPLEMENTAÇÃO DE UM MICROCONTROLADOR DE 16 BITS EM ARQUITETURA RISC

Alegrete, RS
2014

ALIAN MOREIRA ENGROFF

PAMPIUM I - PROJETO E IMPLEMENTAÇÃO DE UM MICROCONTROLADOR DE 16 BITS EM ARQUITETURA RISC

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica, Área de Concentração em Microeletrônica Digital, da Universidade Federal do Pampa (Unipampa, RS), como requisito parcial para obtenção do grau de **Bacharel em Engenharia Elétrica**.

Orientador: Prof. Dr. Alessandro Girardi
Coorientador: Prof. Dr. Sidinei Ghissoni

Universidade Federal do Pampa – Unipampa
Curso de Engenharia Elétrica

Alegrete, RS
2014

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

E57p Engroff, Alian Moreira
PAMPIUM I - PROJETO E IMPLEMENTACÃO DE UM MICROCONTROLADOR DE 16 BITS EM ARQUITETURA RISC / Alian Moreira Engroff.
148 p.

Trabalho de Conclusão de Curso(Graduação)-- Universidade Federal do Pampa, ENGENHARIA ELÉTRICA, 2014.
"Orientação: Prof. Dr. Alessandro Girardi".

1. Microcontrolador 16 bits. 2. Eletrônica Digital. 3. Arquitetura e Organização de Computadores . 4. Projeto e Síntese de Arquiteturas. 5. Grupo de Arquitetura de Computadores e Microeletrônica. I. Título.

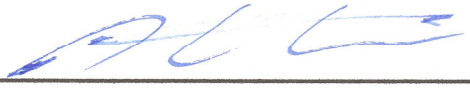
Alian Moreira Engroff

PAMPIUM I - PROJETO E IMPLEMENTAÇÃO DE UM MICROCONTROLADOR DE 16 BITS EM ARQUITETURA RISC

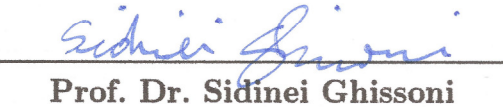
Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica, Área de Concentração em Microeletrônica Digital, da Universidade Federal do Pampa (Unipampa, RS), como requisito parcial para obtenção do grau de **Bacharel em Engenharia Elétrica**.

Trabalho de Conclusão defendido e aprovado em: Alegrete, RS, 2014:

Banca examinadora:



Prof. Dr. Alessandro Girardi
Orientador
UNIPAMPA



Prof. Dr. Sidinei Ghissoni
UNIPAMPA

Prof. Dr. Aline Vieira de Mello
UNIPAMPA

AGRADECIMENTOS

Agradeço a Deus por ter dado saúde e força a mim e minha família nesse caminho. Aos meu pais Artur e Balbina e meus irmãos Alexander e Bianca por serem os principais responsáveis por eu chegar até aqui. Agradeço a minha namorada Aline por me apoiar durante toda a minha jornada. Agradeço ao meu orientador professor Alessandro Girardi por todas as orientações durante a graduação e esse projeto, também por sua paciência ao me ajudar a corrigir meus erros de português. Agradeço ao meu coorientador professor Sidinei Ghissoni por me ajudar em várias questões essenciais para o desenvolvimento desse projeto. A todos os meus colegas e amigos por suas contribuições na minha jornada e por dividirem as dificuldades das disciplinas comigo. Agradeço ao Grupo de Arquitetura de Computadores e Microeletrônica(GAMA) através do CNPQ e a Universidade Federal do Pampa(UNIPAMPA) através da PRAEC por investirem em mim, financiando meus estudos. E a todas as pessoas que me ajudaram de forma direta ou indireta a conquistar mais esse objetivo.

"Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes".

(Marthin Luther King)

RESUMO

Este trabalho apresenta o desenvolvimento em SystemVerilog, simulação e síntese lógica de um microcontrolador de 16 bits com arquitetura RISC chamado PAMPIUM I. Os tópicos para o desenvolvimento da arquitetura que são detalhados neste trabalho são: Proposta de estrutura básica; Definição das instruções a serem implementadas; Especificação do tamanho da palavra de instrução e os formatos padrões; Especificação das instruções no padrão que serão implementadas; Levantamento do caminho de dados para as instruções dentro da arquitetura básica proposta; Especificação dos blocos funcionais para atenderem aos requisitos do caminho de dados; Implementação em linguagem de hardware e teste de validação funcional dos bloco que compõem a arquitetura do PAMPIUM I; Implementação do caminho de dados final e teste de validação das instruções wm ferramenta Quartus II da Altera; Validação das instruções em FPGA; Testes de aplicação em comunicação com chip ISP1362; Teste de aplicação no controle de uma porta eletrônica de segurança. O PAMPIUM I é um microcontrolador de 16 bits em arquitetura RISC, com 32 instruções implementadas, dois bancos de registradores com 32 registradores cada, $64kWords$ de 16 bits de memória de dados, $8kWords$ de 24 bits de memória de programa e duas portas IN/OUT de 16 bits. A síntese em FPGA resulta na utilização de 3336 elementos lógicos e 1536 registradores. As memórias de dados e de programa foram mapeados para a memória da FPGA que resultaram um total de 88 $kBytes$ da memória usada. A frequência máxima de clock é de $25.4MHz$, sendo que o caminho critico é dado pelas operações envolvendo a ULA. O microcontrolador foi aplicado no controle do ISP1362 e para o projeto de uma porta eletrônica de segurança. Os testes de validação demostraram o correto funcionamento das interfaces de entrada e saída na comunicação com ISP1362, bem como o correto funcionamento do PAMPIUM I como microcontrolador no controle de uma porta de segurança eletrônica.

Palavras-chave: Microcontrolador. SystemVerilog. Projeto. Desenvolvimento. RISC. Validação.

ABSTRACT

This work presents the development in SystemVerilog, simulation and logic synthesis of a 16-bit microcontroller with RISC architecture called PAMPIUM I. The topics for the development of architecture that are detailed in this work are: the basic structure; definition of instructions to be implemented; specification of the size of the instruction word and standard formats; specification of the instructions to be implemented; creation of the instruction data path within the basic architecture proposed; specification of functional blocks that meet the requirements of the data path; implementation in hardware description language of the functional test validation for each block that compose the architecture of the PAMPIUM I; implementation of the final data path and validation of test instructions in Quartus II tool; test applicaton for communication with an ISP1362 chip; and test application for controlling an electronic security door. The PAMPIUM I is a 16-bit microcontroller RISC architecture, with 32 implemented instructions, two banks of registers each with 32 registers, $64kWords$ of 16-bit of data memory, $8kWords$ of 24-bit of program memory and two in/out 16-bit ports. The FPGA synthesis results in using 3336 logic elements and 1536 registers. The FPGA synthesis resulted in using 3336 logic elements and 1536 registers. The data memory and program memory are mapped to the FPGA resulting a total of 88 $kBytes$ of used memory. The maximum clock frequency is $25.4MHz$,and the critical path is given by operations involving the ALU. The microcontroller was used to control the ISP1362 and to the design of an electronic security door. Validation tests demonstrated the correct operation of input and output interfaces to communicate with ISP1362, well as the correct functioning of PAMPIUM I like microcontroller to control a security door electronics.

Key-words: Microcontroller. SystemVerilog. Design. Development. RISC. Validation.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente os sistemas de computação vem sendo usados em uma ampla variedade de aplicações, e com os avanços da tecnologia essa demanda tende a aumentar. Dentro dos sistemas computacionais estão contidos os dispositivos microcontroladores que apresentam importância no desenvolvimento de equipamentos de processamento de dados. Assim há necessidade de implementar sistemas inteligentes que requerem baixo custo e pouca demanda por recursos energéticos e espaço, o que torna atraente o uso e criação de arquiteturas de microcontroladores. No projeto de um microcontrolador existem diferentes níveis e cada um dos níveis depende dos demais para apoio na concepção de uma máquina de qualidade. No nível conceitual, a atenção é dada às necessidades do usuário. Atualmente, a ênfase é dada à velocidade, versatilidade, negociabilidade, e vida útil (HENNESSY; PATTERSON, 2012)(RANDELL, 1987)(CARRO; WAGNER, 2003) .

Este trabalho apresenta o desenvolvimento em SystemVerilog de uma arquitetura de microcontrolador de 16-bits RISC para uso geral chamada PAMPIUM I. O objetivo principal é desenvolver uma arquitetura com instruções genéricas que permita a fácil comunicação e controle de vários periféricos, tornando possível sua utilização nas mais diversas aplicações, como desenvolvimento de equipamentos de comunicação e aquisição de dados.

A principal finalidade é desenvolver uma arquitetura genérica capaz de se comunicar e controlar diversos periféricos. Também é interesse gerar conhecimento no projeto de microcontroladores, visto que isso faz parte da linha de pesquisa na área de microeletrônica desenvolvida pelo Grupo de Arquitetura de Computadores e Microeletrônica(GAMA) da UNIPAMPA, campus Alegrete.

A metodologia utilizada neste projeto é baseado no IP-Process (LIMA et al., 2005b), adotada pelo programa Brazil-IP. Na metodologia IP-Process, procedimentos de verificação estão presentes em cada fase do projeto. Essa metodologia divide o projeto de um IP-core em quatro fases: concepção comportamental, arquitetura, projeto RTL e prototipação. Na fase de projeto comportamental, os requisitos funcionais e não-funcionais são listados em ordem para definir o escopo do projeto e critérios de aceitação. Na fase de arquitetura, os blocos e as conexões são estabelecidas, fornecendo a base para a implementação e verificação. No projeto RTL, a arquitetura é descrita em blocos sintetizáveis e inclusão de código para verificação funcional.

O trabalho está dividido em seis capítulos que abordam o desenvolvimento do projeto, sendo eles resumidamente apresentados abaixo:

No Capítulo 1 são apresentados os conceitos utilizados para o desenvolvimento e para as decisões de projeto.

O Capítulo 2 apresenta a metodologia e as ideias iniciais da arquitetura e organização do PAMPIUM I.

No Capítulo 3 é apresentada a especificação de cada instrução e o caminho de dados necessário para a execução das mesmas.

O Capítulo 4 apresenta a implementação do microcontrolador e os resultados da validação das instruções em software e em FPGA.

O Capítulo 5 apresenta os resultados da síntese em FPGA e os testes de aplicação, comunicação com ISP1362 e porta de segurança eletrônica.

Por último, o Capítulo 6 apresenta uma breve conclusão sobre os resultados obtidos e proposta para futuros trabalhos.

2 CONTEXTUALIZAÇÃO TEÓRICA

2.1 Introdução

Este capítulo visa descrever de forma detalhada os principais conceitos utilizados para o desenvolvimento do microcontrolador PAMPIUM I. Apresenta conceitos referentes a arquitetura e organização de microcontroladores, sistema embarcado, arquiteturas CISC e RISC. Apresenta também conceitos sobre as arquiteturas Von Neumann e Harvard, utilizadas com modelo de aprendizagem.

2.2 Arquitetura e Organização de Computadores

Arquitetura refere-se aos atributos do ponto de vista do programador e, portanto, tem impacto direto sobre a execução lógica de um programa. Logo, a arquitetura está diretamente associada ao tamanho dos registradores, número e tipos de instruções, tamanho da memória dados. Já a organização refere-se às unidades operacionais e suas interconexões. Desta forma, uma mesma arquitetura pode ser implementada por meio de diferentes organizações(HENNESSY; PATTERSON, 2012)(ORTEGA; ANGUITA; PRIETO, 2005).

Com isso, a arquitetura está diretamente relacionada ao tipo de aplicação que a mesma exercerá, atendendo aos requisitos de sistemas mais complexos ou mais simples. Como exemplo, um processador encontrado em um computador, que possui fonte de alimentação ligada a uma tomada, e portanto não há necessidade de se preocupar com consumo de energia da mesma forma que um computador alimentado por bateria. Essa aplicação permite que a arquitetura do microprocessador seja de alta velocidade com consumo de energia elevado. Como a aplicação de um computador pede que o processador execute diferentes tipos de programas, a arquitetura deve possuir um grande número de instruções complexas, para tratar com elevado desempenho variados tipos de dados. Já os sistemas embarcados necessitam de arquiteturas de baixo consumo de energia, com instruções mais simples e não exigem um desempenho tão rápido quanto de um processador de um computador.

2.3 Sistemas Embarcados

Sistemas embarcados englobam a capacidade de colocar um sistema computacional dentro de apenas um circuito, um sistema completo e independente mas preparado para executar um determinada tarefa. O usuário final não tem acesso ao programa instalado nesse sistema, apenas contato com sua interface externa. Diferente dos computadores convencionais, os quais são desenvolvidos para que diversas aplicações sejam executadas, os sistemas embarcados em geral são construídos para executar uma tarefa pré-definida, com pouca funcionalidade, reduzidas dimensões e baixo consumo de energia.(GERVINI et al., 2003)(GAJSKI et al., 1994).

Podem ser classificados pelo tipo de aplicações: Aplicação de propósito geral como as dos computadores, mas com reduzidas dimensões e com grande interação com o usuário; Aplicação em sistema de controle, em malha fechada com sistemas robustos dedicados a sistemas com muitos sensores; Aplicação em processamento de sistemas, em geral adquire sinais através de conversores ADs; Rede de comunicação no controle de chaveamento de sinais de comunicação.

Em geral os sistemas embarcados podem operar em dois tipos operações. Em modo reativo, respondendo a estímulos do usuário com botões e há eventos extremos. Mas enquanto não há eventos, o sistema fica em espera, não executando nenhuma função. Em modo real-time, onde existe a necessidade de executar cada tarefa em um determinado tempo específico, como coleta de dados, análise e resposta. Esses sistemas não dependem de estímulos externos para tomadas de decisão. O modo real-time é subdividido em dois tipos de execução em tempo real: Em real-time, as tarefas devem ser executadas em um determinado tempo, mas sem consequências graves caso ultrapassem esse tempo; Em verdadeiro real-time as tarefas devem ser executadas no tempo especificado, caso isso não ocorra, haverá consequência graves pela falha.

Quanto as características desejadas para um sistema embarcado são: Tamanho e peso sempre o mais reduzido o possível; Deve apresentar o menor consumo de energia possivel, aumentando sua autonomia; Ser robusto e resistente.

2.4 Arquiteturas CISC e RISC

As arquiteturas são divididas em dois conceitos básicos, que são definidas quanto ao número de instruções e a complexidade da mesma. As duas classificações são:

2.4.1 CISC

A denominação de CISC, do inglês *Complex Instruction Set Computer* é usado para designar arquiteturas computacionais com um conjunto complexo de instruções. Em geral estás arquiteturas possuem um grande número de instruções e tipos de dados, diversos modos de endereçamento, além de tipos variados de instruções e dados (JUNQUEIRA, 1993)(HWANG; RAMACHANDRAN; PURUSHOTHAMAN, 1993).

Esse tipo de arquitetura apresenta grande número de instruções e muito delas complexas, o que faz com que o hardware seja muito complexo, dificultando seu desenvolvimento. Os processadores baseados na computação de conjunto de instruções complexas contêm uma micro programação, ou seja, um conjunto de códigos de instruções que são gravados no processador, permitindo-lhes receber as instruções dos programas e executá-las, utilizando as instruções contidas na sua micro programação. Em geral essas arquiteturas são utilizadas em computadores e servidores. As aplicações necessitam de um processamento

rápido de cálculos complexos. Por serem mais complexas, demandam mais área de silício e um consumo maior de energia.

2.4.2 RISC

RISC, da sigla em inglês *Reduced Instruction Set Computer*, tem sido usada para designar máquinas com conjunto reduzido de instruções, sendo essas simples. A arquitetura RISC surgiu como alternativa ao projeto de arquiteturas CISC(JUNQUEIRA, 1993)(HWANG; RAMACHANDRAN; PURUSHOTHAMAN, 1993).

Entre as características básicas para uma arquitetura RISC estão: As operações são realizada entre registradores, há apenas operações de leitura e escrita com memória; Contém no máximo 100 instruções e 3 tipos de endereçamento. Essas características facilitando o desenvolvimento do controle da arquitetura. São utilizadas em equipamentos de baixo consumo de energia e que não necessitam de cálculos complexos.

2.5 Arquitetura Von Neumann

A Arquitetura de Von Neumann - de John Von Neumann - é uma arquitetura de computador que se caracteriza pela possibilidade de uma máquina digital armazenar seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas. Esta arquitetura é um projeto modelo de um computador digital de programa armazenado que utiliza uma unidade de processamento (CPU) e uma unidade de armazenamento ("memória") para comportar, respectivamente, instruções e dados, a qual pode ser observada na Fig.1.

A máquina proposta por Von Neumann reúne os seguintes componentes: uma memória, uma unidade aritmética e lógica, uma unidade central de processamento (CPU), composta por diversos registradores, e uma Unidade de Controle , cuja função é a mesma da tabela de controle da Máquina de Turing universal: buscar um programa na memória, instrução por instrução, e executá-lo sobre os dados de entrada(CARTER, 2002)(NEUMANN, 1992).

2.6 Arquitetura Harvard

A Arquitetura Harvard criada em 1945, baseia-se em um conceito mais recente que a Von Neumann, tendo em vista a necessidade de um microcontrolador mais rápido. É uma arquitetura de computador que se distingue das outras por possuir duas memórias diferentes e independentes em termos de barramento e ligação ao processador. Ela é baseada na separação dos barramentos de dados de memória de programa e de dados, permitindo que um processador possa acessar as duas simultaneamente, o que faz com que seja obtido um desempenho melhor do que a Arquitetura de Von Neumann, pois

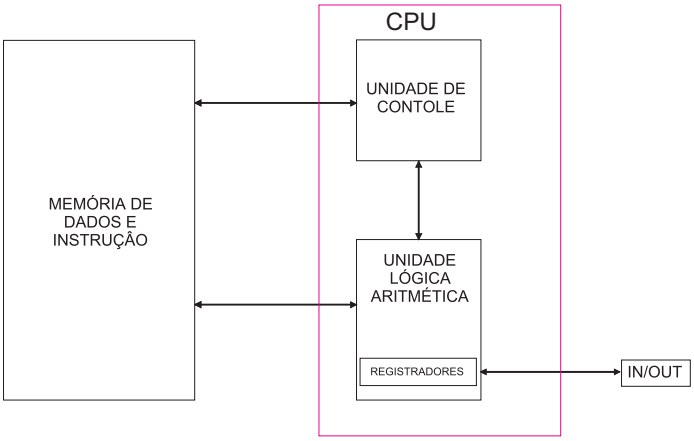


Fig. 1 – Organização da arquitetura Von Neumann.

pode buscar uma nova instrução enquanto executa outra. A estrutura da organização da arquitetura Harvard está expressa na Fig.2. Arquitetura Harvard é normalmente utilizada em qualquer sistema especializado ou para uso específico. É utilizada em processamento de sinais digitais(DSP)e em pequenos microcontroladores, tais como ARM(Advanced Risc Machines)(STALLINGS, 2001)(NEUMANN, 1992).

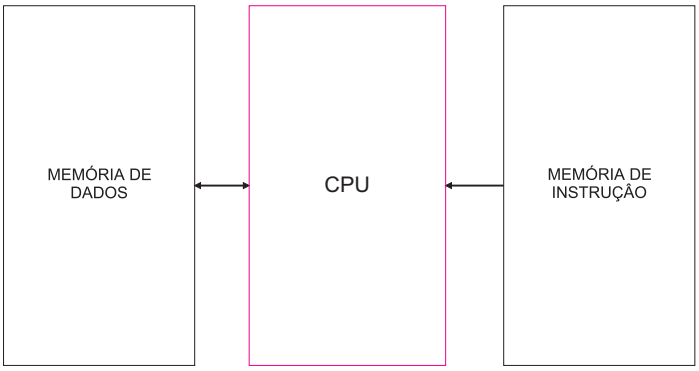


Fig. 2 – Organização da arquitetura Harvard.

3 METODOLOGIA E ORGANIZAÇÃO BÁSICA

Nesse capítulo serão apresentadas as metodologias utilizadas no desenvolvimento de um microcontrolador de arquitetura RISC de 16 bits. São apresentadas especificações básicas do microcontrolador, definidas de modo a compatibilizar a arquitetura com arquiteturas já consolidados no mercado, definindo o modo operante do microcontrolador, também são apresentadas as especificações básicas da memória de programa, do banco de registradores, da memória de dados e das instruções básicas a serem implementadas.

3.1 Metodologia

Este trabalho busca desenvolver uma arquitetura de 16 bits RISC baseada na arquitetura Harvard e Von Neumann, mesclando elementos da cada uma das arquiteturas. A separação da memória de dados da memória de programa está presente na Harvard e a utilização de registradores para os cálculos é encontrada na arquitetura Von Neumann. A escolha de uma arquitetura 16 bits se deve ao fato de a mesma apresentar um número de bits intermediário em relação à maioria das arquiteturas encontradas no mercado. As arquiteturas de 8 bits são utilizadas para projetos mais simples, enquanto arquiteturas de 32 bits são utilizadas em projetos que necessitam maior precisão de cálculo. Logo, uma arquitetura de 16 bits atende a uma maior quantidades de áreas, facilitando a conquista do objetivo de criar uma arquitetura de propósito geral de baixo custo, permitindo seu emprego nas mais diversas aplicações.

A arquitetura proposta é composta por uma memória de programa, memória de dados, dois bancos de registradores, uma unidade logica aritmética(ULA) e uma unidade de controle, sendo esses os blocos principais. Também há duas portas de comunicação externa do tipo IN/OUT. O esquema básico está expresso na Fig. 3.

Para o desenvolvimento dessa arquitetura é definido o tamanho da palavra de instrução, o número e tamanho dos registradores em cada banco, o método que é utilizado nas configurações internas e a direcionalidade das portas do microcontrolador. Também são definidas as funcionalidades que cada bloco exerce dentro da arquitetura, como também as instruções implementadas e sua segmentação dentro da palavra de instrução. Com as informações de projeto e o caminho de dados entre os blocos é especificada a estrutura organizacional do PAMPIUM I.

3.2 Modo Operante

O microcontrolador realiza todas as operações em um único ciclo, com exceção das instruções de condicionais que são executadas em um ciclo se a condição não é verdadeira e em 2 ciclos caso seja verdadeira, um ciclo para o teste condicional e outro para realizar o pulo do número de instruções.

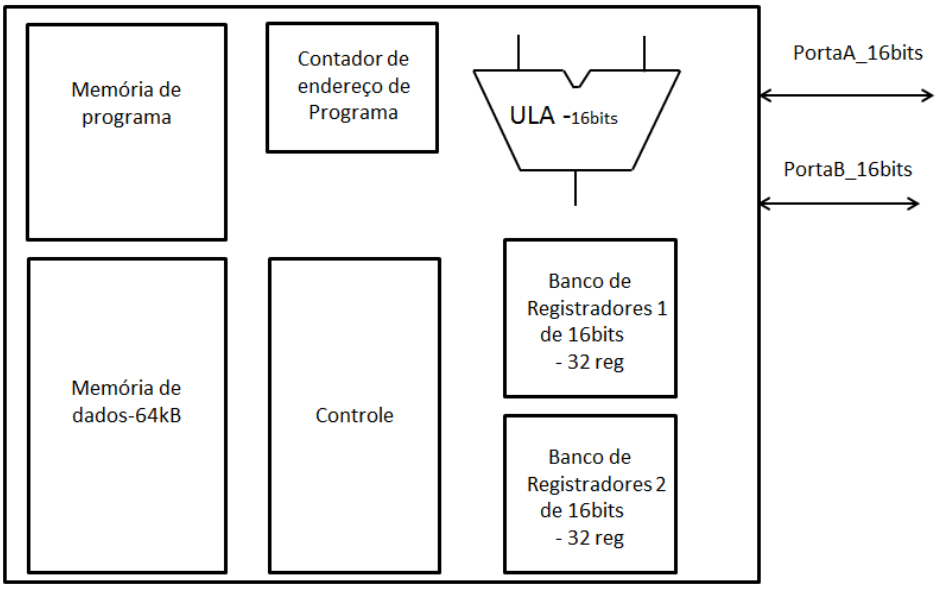


Fig. 3 – Estrutura organizacional básica do microcontrolador.

A comunicação externa é feita através de porta IN/OUT do microcontrolador. A direcionalidade de cada pino das portas é definido por um bit específico em um registrador específico contido no banco de registradores. O dado lido ou escrito na porta está contido em outro registrador específico.

Todas as operações aritméticas e lógicas são realizadas entre registradores e seus resultados serão salvos em registradores.

Os literais necessários para operações são gravados em registrador específico. Operações com a *ULA* geram flags utilizadas em operações condicionais. Tais flags também estão disponíveis em registrador específico. O endereço de acesso de leitura ou escrita à memória de dados é determinado por registrador específico possibilitando o acesso a um número maior de endereços de memória de dados. Logo, é necessário primeiramente definir o endereço de acesso em um registrador específico, e após executar a instrução de leitura ou escrita da memória.

As operações condicionais entre registradores analisam o resultado da operação de subtração entre dois registradores através das flags geradas pela *ULA* para determinar se a condição analisada foi satisfeita. Também há condicionais binárias que verificam um bit de um registrador para determinar se a condição foi satisfeita.

3.3 Memória de Programa

A memória de programa é composta basicamente por um bloco de memória com um tamanho de palavra de 24 bits e o contador de programa de 16 bits. O bloco de memória pode armazenar um número máximo de $64kWords$ de instruções. O contador de programa

tem a função de indicar qual instrução está sendo executada, além de incrementar o endereço de modo que no próximo ciclo seja executada a próxima instrução. O esquema básico da memória de programa pode ser visto na Fig.4.

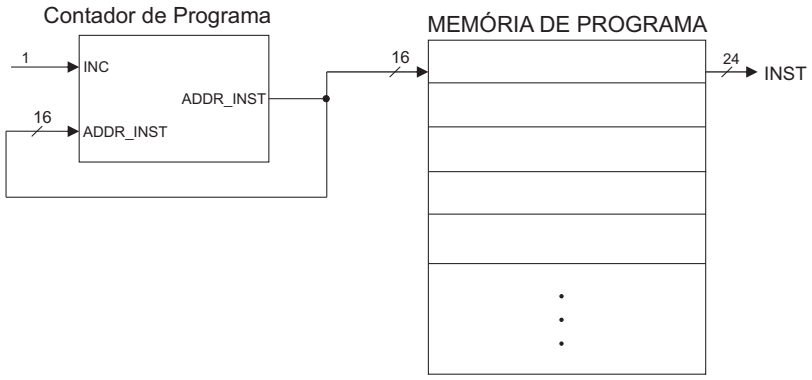


Fig. 4 – Esquemático Básico da memória de programa.

3.4 Banco de Registradores

O banco de registradores principal é composto por dois bancos. chamados *R0* e *R1*, com 32 registradores cada, sendo que cada registrador tem 16 bits. Para acesso a um determinado registrador é necessário informar o endereço do mesmo e em qual banco ele está localizado. Alguns registradores apresentam funcionalidade específica para determinadas instruções ou configurações de portas. O número de registradores e suas funcionalidades serão detalhadas no próximo capitulo. O esquema básico do banco de registradores pode ser observado na Fig.5.

3.5 Memória de dados

Um registrador de 16 bits é capaz de endereçar até 65.536 posições diferentes em uma memória. Por isso optou-se por utilizar um registrador para armazenamento do endereço da memória, pois utilizando o campo disponível da palavra de instrução permitiria apenas mapear 1024 endereços. O esquema básico é expresso na Fig. 6.

3.6 Unidade Lógica Aritmética - ULA

A *ULA* realiza as operações aritméticas e lógicas entre registradores, tais operações são:

- Aritméticas: Soma, Subtração, Multiplicação, Divisão e Resto da divisão.
- Lógicas: AND, OR, NOT e XOR.

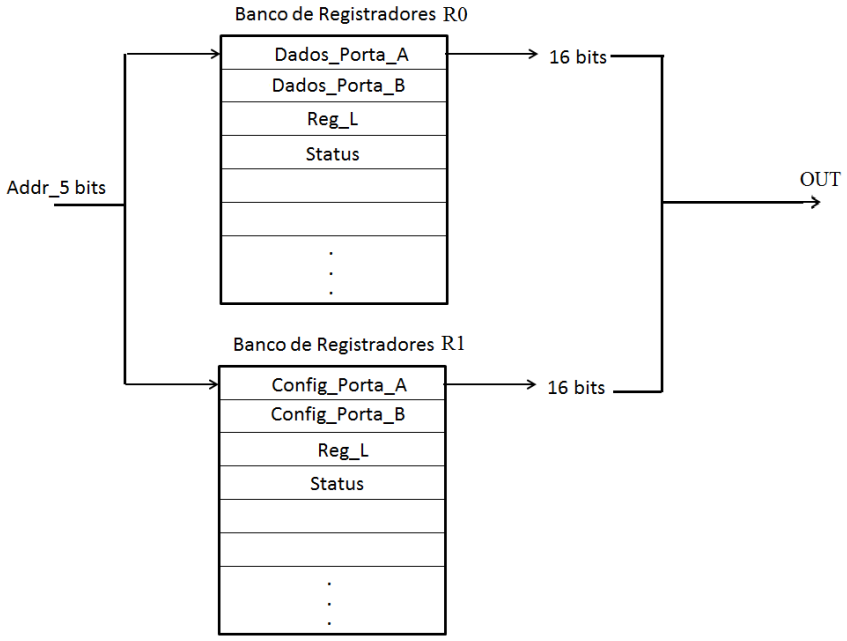


Fig. 5 – Esquemático Básico do banco de registradores.

Fig. 6 – Esquemático Básico da memória de dados.

- Operações sobre um determinado bit de um registrador;

As operações realizadas pela *ULA* geram flags, que tem a finalidade de indicar se o resultado da operação é zero, overflow ou negativo, também há flag BIT utilizada para operações condicionais. O esquema básico da *ULA* pode ser visto na Fig.7.

3.7 Instruções Básicas

As instruções básicas a serem implementadas no microcontrolador, que são possíveis de se implementar com as especificações anteriores são:

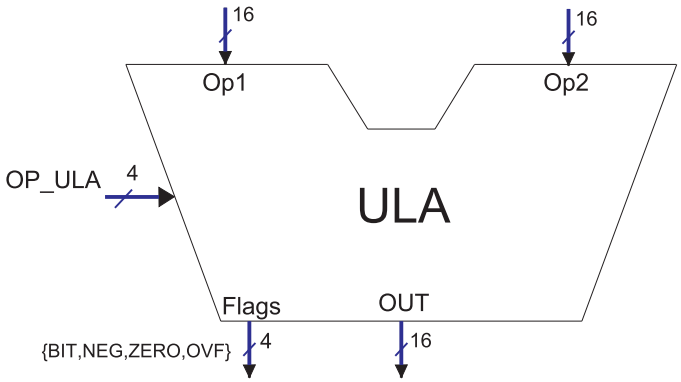


Fig. 7 – Esquemático Básico da *ULA*.

- Copiar o valor de um registrador para outro;
- Gravar um valor literal em registrador específico;
- Pular um número de instruções informado pelo usuário;
- Pular um número de instruções salvando posição para retorno.
- Retornar de pulo de instruções podendo salvar ou não um literal no registrador específico;
- Copiar o valor da memória de dados para um registrador;
- Copiar o valor do registrador para memória de dados;
- Determinar o valor de um bit específico, como um ou zero;
- Pular um número de instruções, na seguintes condições fora satisfeitas:

  A>B; A<B; A<=B; A>=B; A==B; A!=B;
- Pular um número de instruções se o bit “N” do registrador “A” é igual a “1” ou “0” lógico.
- Deslocar(shift) direita ou a esquerda os bits de um registrador;
- Realizar operações aritméticas e lógicas entre registradores:

  A*B; A/B; A%B; A+B; A-B; A && B, A || B; A^B; !A;
- Não realizar nada em um ciclo;
- Paralisar o contador de programa, o contador de programa é paralisado nessa instrução, utilizada para fins de teste de hardware não acessível a usuário final;
- Pular um número de instruções indicada pelo valor de um registrador;

4 INSTRUÇÕES E CAMINHO DE DADOS

As instruções são utilizadas como referência para a determinar o caminho de dados. Foram especificados três formatos de instruções que servirão de molde para a implementação das instruções. Portanto cada instrução foi implementada no formato de instrução mais adequado, determinando a funcionalidade de cada campo do formato utilizado. Assim, com a especificação e detalhamento das instruções, o caminho de dados e cada bloco do microcontrolador foram especificados de modo implementar as instruções.

4.1 Formato Base das Instruções

A memória de programa foi definida com um tamanho de palavra de 24 bits, sendo os 6 bits mais significativos dedicados ao código de programa. Portanto, é possível implementar até 64 instruções diferentes. Há também 2 bits após o código de operação que permitem indicar em qual banco de registradores estão contidos os registradores utilizados nas operações. Por fim, sobram 16 bits para especificação dependendo do tipo de instruções executadas. O esquema básico pode ser observado na Fig 8.

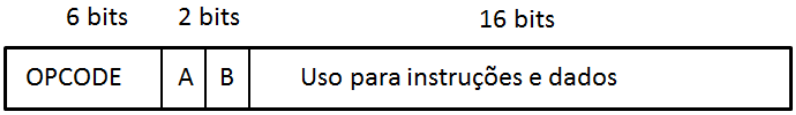


Fig. 8 – Formato básico de palavra de instrução.

A forma base para as instruções é subdividido em três formatos, chamados L, R e M, os quais são utilizados como base para as instruções a serem implementadas.

4.1.1 Formato L

Formato de instrução L visa atender às instruções que necessitam entrada de valores literais para sua execução.

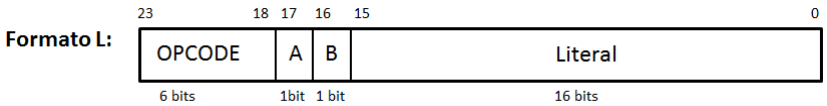


Fig. 9 – Formato de instruções L.

O formato é dividido em quatro campos distintos expressos na Fig.9: um campo para a definição do código de operação, dois campos A e B caso seja necessário informar em qual banco de registradores está localizado o registrador utilizado pela instrução e um campo de 16 bits para entrada de literais.

4.1.2 Formato R

Formato de instrução R visa às atender instruções que executam operações entre registradores.

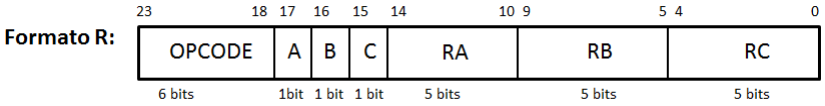


Fig. 10 – Formato de instruções R.

O formato é dividido em sete campos distintos expressos na Fig.10: um campo para a definição do código de operação, três campos A, B e C que indicam em qual banco de registradores estão localizados os registradores utilizados na operação e três campos RA, RB e RC que definem os endereços desse registradores.

4.1.3 Formato M

Formato de instrução M visa às atender instruções que executam operações entre registrador a memória de dados.

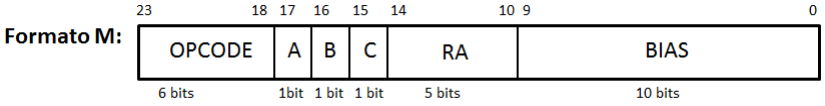


Fig. 11 – Formato de instruções M.

O formato é dividido em seis campos distintos expressos na Fig.11: um campo para a definição do código de operação, três campos A, B e C que indicam em qual banco de registradores estão localizados os registradores utilizados na operação, o campo RA que define o endereço do registrador utilizado na operação e o campo $BIAS$ que tem a função de complementar o endereço de acesso à memória de dados.

4.2 Conjunto de Instruções

Todas as instruções implementadas são baseadas nas necessidades descritas na seção 3.7, contabilizando 32 instruções que podem ser vistas na Tabela 1.

4.2.1 Instrução cópia entre registradores - COPY

A instrução **COPY** tem a função de copiar um valor contido em um registrador para outro. Utiliza o formato de instruções R expresso na Fig.10.

- O campo RA indica qual registrador terá seu valor copiado;

Tabela 1 – Instruções do PAMPIUM I.

OPCODE	Mnemônico	Descrição curta
000000b	**NOP**	Não faz nada
000001b	**END**	Paralisa programa
000010b	**COPY**	Cópia de valores entre registradores
000011b	**MOVL**	Atribui literal para *#REG_L*
000100b	**JUMP**	Pula instruções
000101b	**CALL**	Pula instruções e salva endereço de retorno
000110b	**RET**	Retorno para instrução
000111b	**RETL**	Retorno para instrução salvando literal
001000b	**RM**	Lê da memória de dados
001001b	**WM**	Escreve na memória de dados
001010b	**BL**	Pula se for maior
001011b	**BS**	Pula se for menor
001100b	**BLE**	Pula se for maior ou igual
001101b	**BSE**	Pula se for menor ou igual
001110b	**BE**	Pula se for igual
001111b	**BNE**	Pula se foi diferente
010000b	**BBCLEAR**	Pula se o bit for igual ”0”
010001b	**BBSET**	Pula se o bit for igual ”1”
010010b	**SHR**	Deslocamento a direita
010011b	**SHL**	Deslocamento a esqueda
010100b	**BSET**	Determina o bit como ”1”
010101b	**BCLEAR**	Determina o bit como ”0”
010110b	**MUL**	Multiplica o valor dos registradores
010111b	**DIV**	Divide o valor dos registradores
011000b	**REM**	Retorna o resto da divisão de dois registradores
011001b	**ADD**	Soma o valor dos registradores
011010b	**SUB**	Subtrai o valor dos registradores
011011b	**OR**	Faz a operação ”OU”lógica
011100b	**AND**	Faz a operação ”E”lógica
011101b	**XOR**	Faz a operação ”OU”exclusiva lógica
011110b	**NOT**	Faz a operação ”NOT”lógica
011111b	**ADDPC**	Pula instruções indicada por registrador

- O campo *RB* não apresenta função nessa instrução;

- O campo *RC* indica o registrador que receberá o valor copiado;

- O campo *A*, se em nível logico “0”, indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico “1” indica o banco *R1*;

- O campo *B* não apresenta funcionalidade nessa instrução;

- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- A descrição matemática da operação pode ser vista na Fig.12:

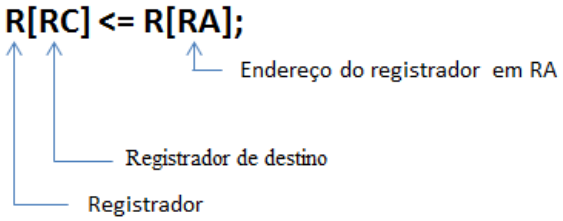


Fig. 12 – Descrição da Operação **COPY**.

- Exemplo de utilização m linguagem de montagem:

COPY *#01h(0),#10h(1);*

Copia o valor do registrador no endereço *#01h* contido no banco de registradores *R0* para o registrador no endereço *#10h* contido no banco de registradores *R1*.

4.2.2 Instrução grava literal - MOVL

A instrução **MOVL** tem a função de gravar um valor literal informado pelo usuário em um registrador específico, que será chamado de *#REG_L* e seu endereço é determinado pelo hardware, é necessário indicar em qual banco de registradores o mesmo está contido. Utiliza o formato de instruções *L* expresso na Fig.9.

- O campo *Literal* contém o valor literal de 16 bits a ser gravado no registrador *#REG_L*;

- O campo *A*, em nível lógico "0" indica que o registrador *#REG_L* está no banco de registradores *R0*, se nível lógico "1" o banco *R1*;

- O campo *B* não apresenta funcionalidade nessa instrução;

- A descrição matemática da operação pode ser vista na Fig.13:

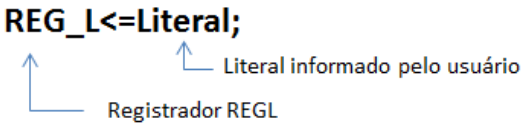


Fig. 13 – Descrição da Operação **MOVL**.

- Exemplo de utilização m linguagem de montagem:

MOVL *AAFFh(1);*

Atribui o valor hexadecimal *AAFFh* para o registrador *#REG_L* contido no banco de registradores *R1*.

4.2.3 Pulo de Instruções - JUMP

A instrução **JUMP** tem a função de pular um certo número de instruções. Esse pulo é feito pela soma do deslocamento mais o valor do contador de programa. Se for necessário subtrair o valor do contador de programa, o mesmo deve ser informado em complemento de dois. Utiliza o formato de instruções *L* expresso na Fig.9.

- O campo *Literal* contém o valor literal de 16 bits a ser somado ao contador de programa;
- Os campos *A* e *B* não apresentam funcionalidade nessa instrução;
- A descrição matemática da operação pode ser vista na Fig.14:

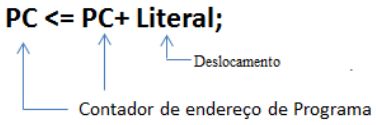


Fig. 14 – Descrição da Operação **JUMP**.

- Exemplo de utilização m linguagem de montagem:

JUMP *0FFAh;*

Soma o valor hexadecimal *0FFAh* informado pelo usuário com o valor do contador de programa.

4.2.4 Pulo de Instruções Salvando Posição de Retorno - CALL

A instrução **CALL** tem a função de pular um certo número de instruções especificado pelo usuário. Esse pulo é obtido pela soma do deslocamento mais o valor do contador de programa. Se for necessário subtrair o valor do contador de programa, o mesmo deve ser informado em complemento de dois. A posição de retorno é salva em uma memória pilha($STACK$) composta por dezesseis posições. O número de posições salvas é armazenado pela variável N e assim que for chamada a instrução essa variável é incrementada. Essa instrução utiliza o formato de instruções L expresso na Fig.9.

- O campo *Literal* contém o valor literal de 16 bits a ser somado ao contador de programa;
- Os campos *A* e *B* não apresentam funcionalidade nessa instrução;
- A descrição matemática da operação pode ser vista na Fig.15:

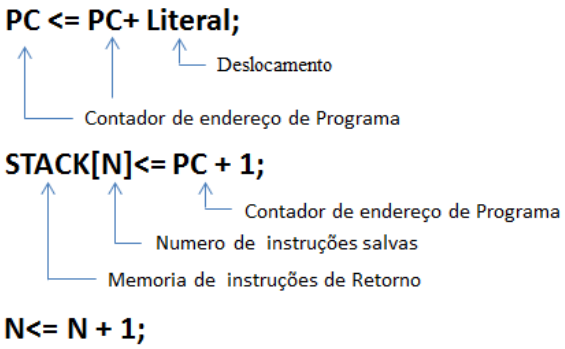


Fig. 15 – Descrição da Operação **CALL**.

- Exemplo de utilização m linguagem de montagem:

CALL *0009h;*

Soma o valor hexadecimal *0009h* informado pelo usuário com o valor do contador de programa, e o endereço de programa para retorno será salvo em *STACK*.

4.2.5 Pulo de um número de instruções definido por Registrador - ADDPC

A instrução **ADDPC** adiciona o valor contido em um registrador ao contador de programa. Utiliza o formato de instruções *M* expresso na Fig.11.

- O campo *RA* indica qual registrador terá seu valor soma o contador de programa;
- O campo *BIAS* não apresenta funcionalidade nessa instrução;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- Os campos *B* e *C* não apresentam funcionalidade nessa instrução;
- A descrição matemática da operação pode ser vista na Fig.16:
- Exemplo de utilização m linguagem de montagem:

ADDPC *#1Ah(1);*

Soma o valor do registrador no endereço *#1Ah* no banco de registradores *R1* ao endereço do contador de programa.

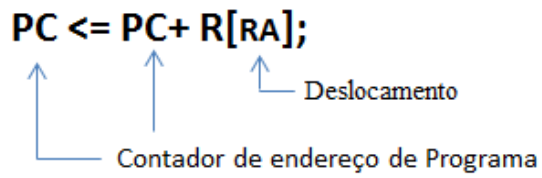


Fig. 16 – Descrição da Operação **ADDPC**.

4.2.6 Retorno Para Instrução - RET

A instrução **RET** tem a função de retornar para a última posição salva em *STACK*. Essa função também decrementa a variável *N* que armazena quantas instruções de retorno foram salvas. A instrução utiliza o formato de instruções *L* expresso na Fig.9.

- O campo *Literal* não apresenta função nessa instrução;
- Os campos *A* e *B* não apresentam funcionalidade nessa instrução;
- A descrição matemática da operação pode ser vista na Fig.17:

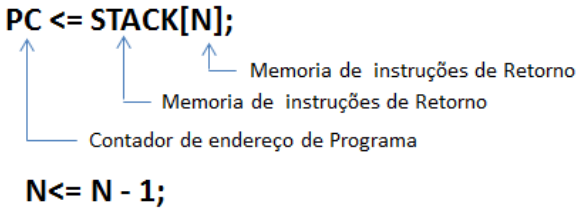


Fig. 17 – Descrição da Operação **RET**.

- Exemplo de utilização m linguagem de montagem:

RET;

Retorna para última instrução salva no *STACK* e decrementa a variável *N*.

4.2.7 Retorno de Instrução Salvando literal em #REG_L - RETL

A instrução **RETL** tem a função de retornar para a última posição salva em *STACK*. Essa operação também decrementa a variável *N* que armazena quantas instruções de retorno foram salvas e também atribui literal informado pelo usuário para *#REG_L*. A instrução utiliza o formato de instruções *L* expresso na Fig.9.

- O campo *Literal* contém o valor literal de 16 bits a ser gravado no registrador *#REG_L*;

- O campo A, se em nível logico "0", indica que o registrador RA está no banco de registradores $R0$. Se nível logico "1" indica o banco $R1$;

- O campo B não apresenta funcionalidade nessa instrução;

- A descrição matemática da operação pode ser vista na Fig.18:

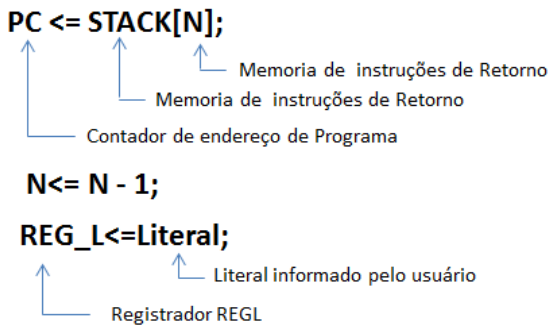


Fig. 18 – Descrição da Operação **RETL**.

- Exemplo de utilização m linguagem de montagem:

RETL *009Ah(1);*

Retorna para última instrução salva no $STACK$ e decrementa a variável N. Salva valor hexadecimal $009Ah$ no registrador $\#REG\_L$ contido no banco de registradores $R1$.

4.2.8 Copia o Valor da Memória de Dados para Registrador - RM

A instrução **RM** tem a função de copiar o valor contido em uma posição da memória de dados para um registrador. O endereço de acesso à memória é composto pelo registrador específico, que foi nomeado com $\#ADDR\_MEM$, mais um valor literal informado pelo usuário. O endereço de acesso à memória deve ser gravado previamente no registrador $\#ADDR\_MEM$. Essa instrução utiliza o formato de instruções M expresso na Fig.11;

- O campo RA indica qual registrador tem seu valor copiado;

- O campo $BIAS$ contém o valor literal a ser somado ao registrador $\#ADDR\_MEM$ para compor o endereço de acesso da memória de dados;

- O campos A, se em nível logico "0", indica que o registrador RA está no banco de registradores $R0$. Se nível logico "1" indica o banco $R1$;

- Os campo B e C não apresentam funcionalidade nessa instrução;

- A descrição matemática da operação pode ser vista na Fig.19:

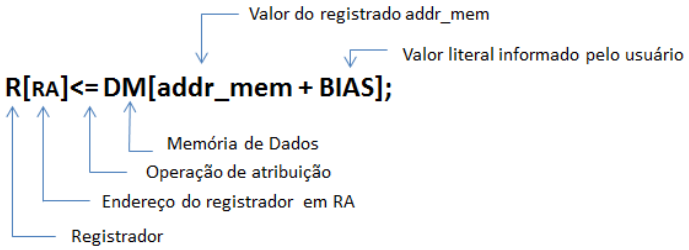


Fig. 19 – Descrição da Operação **RM**.

- Exemplo de utilização m linguagem de montagem:

RM *#1Ah(1),01Fh;*

Copia o valor do registrador no endereço composto pelo valor hexadecimal *01Fh* mais o valor salvo no registrador *#ADDR_MEM*, para o registrador no endereço *#1Ah* contido no banco de registradores *R1*.

4.2.9 Copia o Valor do Registrador para a Memória de Dados - RW

A instrução **RW** tem a função de copiar o valor contido em um registrador para a memória de dados. O endereço de acesso à memória de dados é composto pelo registrador *#ADDR_MEM* mais um valor literal informado pelo usuário. O endereço de acesso à memória deve ser gravado previamente no registrador *#ADDR_MEM*. A instrução utiliza o formato de instruções *M* expresso na Fig.11.

- O campo *RA* indica qual registrador tem seu valor copiado;
- O campo *BIAS* contém o valor literal a ser somado ao registrador *#ADDR_MEM* para compor o endereço de acesso da memória de dados;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- Os campos *B* e *C*, não apresentam funcionalidade nessa instrução;
- A descrição matemática da operação pode ser vista na Fig.20:

Fig. 20 – Descrição da Operação **RW**.

- Exemplo de utilização m linguagem de montagem:

RM *#1Ah(0),002h;*

Copia o valor do registrador no endereço *#1Ah* contido no banco de registradores *R0*, para a memória de dados no endereço composto pelo valor hexadecimal *002h* mais o valor salvo no registrador *#ADDR_MEM*.

4.2.10 Pulo condicional entre dois registradores se "A>B"- BL

A instrução **BL** verifica se o valor salvo em um registrador é maior que em outro registrador. Se a condição for satisfeita, pula um número de instruções definido pelo valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor comparado;
- O campo *RB* indica qual registrador terá seu valor utilizado para a comparação;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.21:

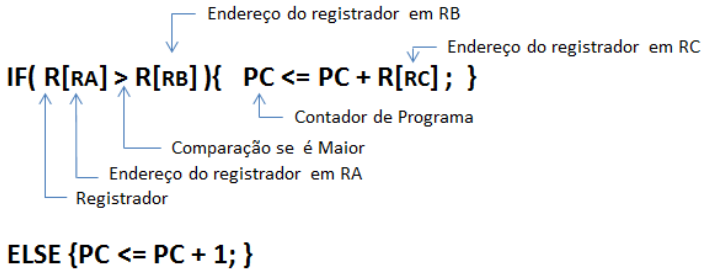


Fig. 21 – Descrição da Operação **BL**.

- Exemplo de utilização m linguagem de montagem:

***BL** #11h(0),#00h(1),#02h(1);*

Verifica se o valor do registrador no endereço *#11h* contido no banco de registradores *R0* é maior que o valor do registrador no endereço *#00h* contido no banco de registradores *R1*. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#02h* contido no banco de registradores *R1*. Caso contrário, executa a próxima instrução.

4.2.11 Pulo condicional entre dois registradores se "A<B"- BS

A instrução **BS** verifica se o valor salvo em um registrador é menor que em outro registrador. Se a condição, for satisfeita pula um número de instruções definido pelo valor salvo em um terceiro registrador. Caso contrário executa a próxima instrução. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor comparado;
- O campo *RB* indica qual registrador terá seu valor utilizado para a comparação;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.22:

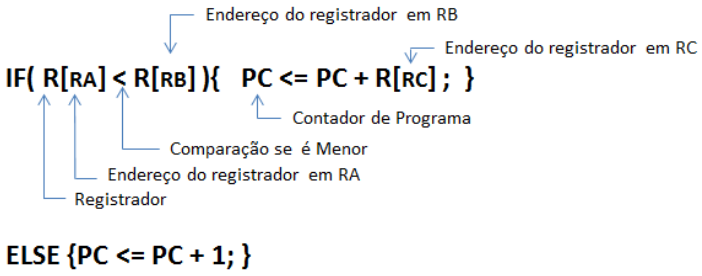


Fig. 22 – Descrição da Operação **BS**.

- Exemplo de utilização m linguagem de montagem:

BS *#12h(1),#02h(1),#01h(1);*

Verifica se o valor do registrador no endereço *#12h* contido no banco de registradores *R1* é menor que o valor do registrador no endereço *#02h* contido no banco de registradores *R1*. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#01* contido no banco de registradores *R1*. Caso contrário, executa a próxima instrução.

4.2.12 Pulo condicional entre dois registradores se "A>=B" - BLE

A instrução **BLE** verifica se o valor salvo em um registrador é maior ou igual ao que está em outro registrador. Se a condição for satisfeita, pula um número de instruções definido pelo valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor comparado;
- O campo *RB* indica qual registrador terá seu valor utilizado para a comparação;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.23:

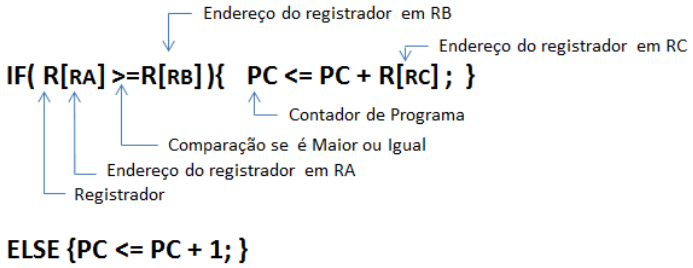


Fig. 23 – Descrição da Operação **BLE**.

- Exemplo de utilização m linguagem de montagem:

***BLE** #03h(1),#04h(1),#05h(1);*

Verifica se o valor do registrador no endereço *#03h* contido no banco de registradores *R1* é maior ou igual ao valor do registrador no endereço *#04h* contido no banco de registradores *R1*. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#05h* contido no banco de registradores *R1*. Caso contrário, executa a próxima instrução.

4.2.13 Pulo condicional entre dois registradores se "A<=B" - BSE

A instrução **BSE** verifica se o valor salvo em um registrador é menor ou igual ao que está em outro registrador. Se a condição for satisfeita, pula um número de instruções através do valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor comparado;
- O campo *RB* indica qual registrador terá seu valor utilizado para a comparação;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.24:

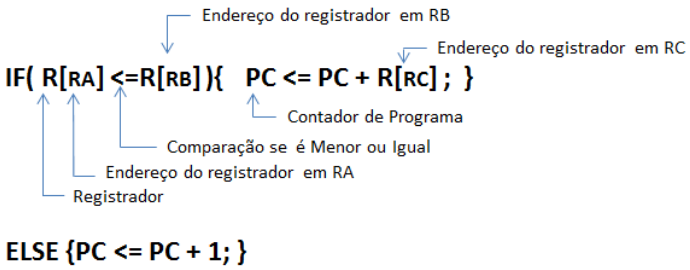


Fig. 24 – Descrição da Operação **BSE**.

- Exemplo de utilização m linguagem de montagem:

BSE *#06h(0),#07h(0),#08h(0);*

Verifica se o valor do registrador no endereço *#06h* contido no banco de registradores *R0* é menor ou igual ao valor do registrador no endereço *#07h* contido no banco de registradores *R0*. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#08h* contido no banco de registradores *R0*. Caso contrário, executa a próxima instrução.

4.2.14 Pulo condicional entre dois registradores se "A==B" - BE

A instrução **BE** verifica se o valor salvo em um registrador é igual ao que está em outro registrador. Se a condição for satisfeita, pula um número de instruções através do valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor comparado;
- O campo *RB* indica qual registrador terá seu valor utilizado para a comparação;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.25:

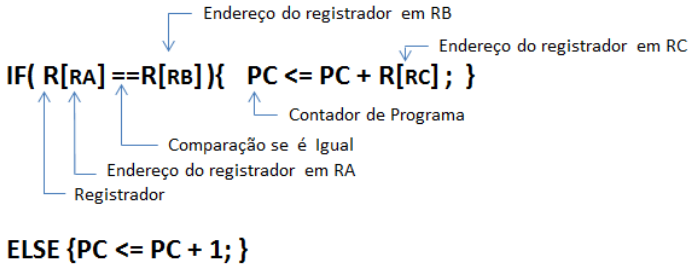


Fig. 25 – Descrição da Operação **BE**.

- Exemplo de utilização m linguagem de montagem:

***BE** #09h(0),#0Ah(1),#0Bh(0);*

Verifica se o valor do registrador no endereço *#09h* contido no banco de registradores *R0* é igual ao valor do registrador no endereço *#0Ah* contido no banco de registradores *R1*. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#0Bh* contido no banco de registradores *R0*. Caso contrário, executa a próxima instrução.

4.2.15 Pulo condicional entre dois registradores se "A!=B" - BNE

A instrução **BNE** verifica se o valor salvo em um registrador é diferente do que está em outro registrador. Se a condição for satisfeita, pula um número de instruções indicado pelo valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor comparado;
- O campo *RB* indica qual registrador terá seu valor utilizado para a comparação;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.26:

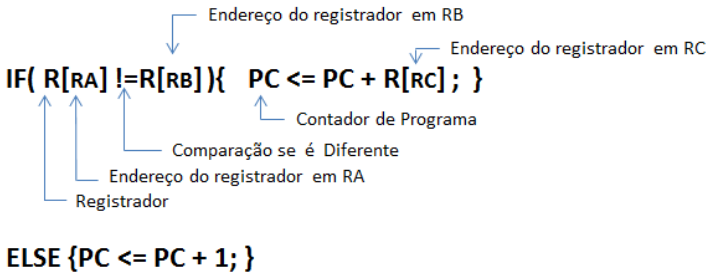


Fig. 26 – Descrição da Operação **BNE**.

- Exemplo de utilização m linguagem de montagem:

***BNE** #0Ch(0),#0Dh(1),#0Eh(1);*

Verifica se o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* é diferente do valor do registrador no endereço *#0Dh* contido no banco de registradores *R1*. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#0Eh* contido no banco de registradores *R1*. Caso contrário, executa a próxima instrução.

4.2.16 Pulo condicional se um determinado bit de um registrador é zero "A(N)==0" - BBCLEAR

A instrução **BBCLEAR** verifica se um determinado bit de um registrador é igual a zero. Se a condição for satisfeita, pula um número de instruções definido pelo valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu bit testado;
- O campo *RB* indica qual bit será testado;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, não apresenta funcionalidade nessa operação;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.27:

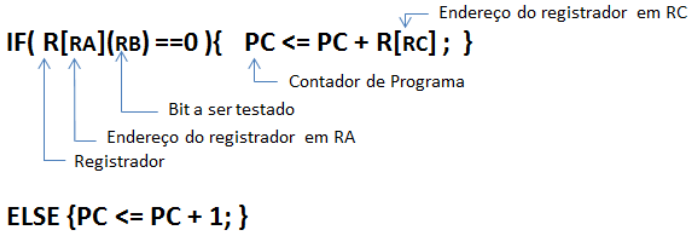


Fig. 27 – Descrição da Operação **BBCLEAR**.

- Exemplo de utilização m linguagem de montagem:

***BBCLEAR** #0Ch(0),8,#0Eh(1);*

Verifica se o valor do bit do registrador no endereço *#0Ch* contido no banco de registradores *R0* na posição *8* é zero. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#0Eh* contido no banco de registradores *R1*. Caso contrário, executa a próxima instrução.

4.2.17 Pulo condicional se um determinado bit de um registrador é um "A(N)==1" - BBSET

A instrução **BBSET** verifica se um determinado bit de um registrador é igual a um lógico. Se a condição for satisfeita, pula um número de instruções definido pelo valor salvo em um terceiro registrador. Caso contrário, executa a próxima instrução. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu bit testado;
- O campo *RB* indica qual bit será testado;
- O campo *RC* indica o registrador que contém o número de instruções a serem puladas;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, não apresenta funcionalidade nessa operação;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.28:

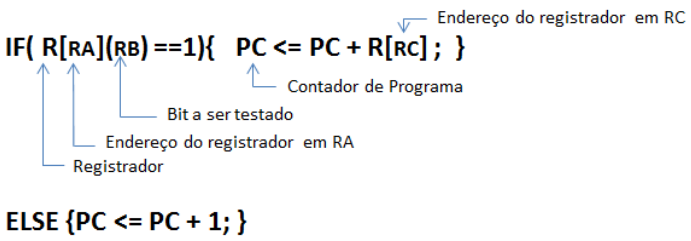


Fig. 28 – Descrição da Operação **BBSET**.

- Exemplo de utilização m linguagem de montagem:

***BBSET** #0Ch(0),8,#0Eh(1);*

Verifica se o valor do bit do registrador no endereço *#0Ch* contido no banco de registradores *R0* na posição *8* é um lógico. Se verdadeiro, pula o número de instruções definido pelo valor salvo no endereço *#0Eh* contido no banco de registradores *R1*. Caso contrário, executa a próxima instrução.

4.2.18 Deslocamento dos bits de um registrador à direita - SHR

A instrução **SHR** desloca o valor de um registrador "N" vezes à direita e o salva em outro registrador. Essa instrução utiliza o formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor deslocado;
- O campo *RB* indica quantas casas à direita será deslocada;
- O campo *RC* indica o registrador que receberá o valor deslocado;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, não apresenta funcionalidade nessa operação;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.29:

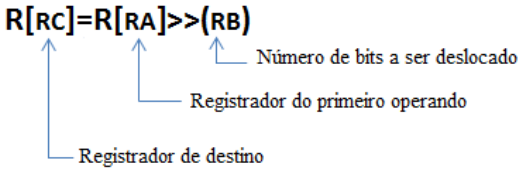


Fig. 29 – Descrição da Operação **SHR**.

- Exemplo de utilização m linguagem de montagem:

SHR *#0Ch(0),8,#0Eh(1);*

Desloca oito vezes à direita o valor binário do registrador no endereço *#0Ch* contido no banco de registradores *R0* e armazena o resultado no registrador no endereço *#0Eh* contido no banco de registradores *R1*.

4.2.19 Deslocamento dos bits de um registrador à esquerda - SHL

A instrução **SHL**, desloca o valor de um registrador ”N” vezes à esquerda e o salva em outro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu valor deslocado;
- O campo *RB* indica quantas casas à esquerda será deslocada;
- O campo *RC* indica o registrador que receberá o valor deslocado;
- O campo *A*, se em nível logico “0”, indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico “1” indica o banco *R1*;
- O campo *B*, não apresenta funcionalidade nessa operação;
- O campo *C*, se em nível logico “0”, indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico “1” indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.30:

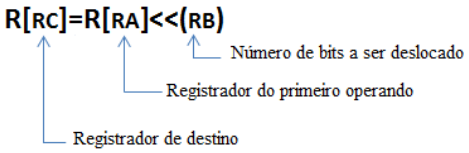


Fig. 30 – Descrição da Operação **SHL**.

- Exemplo de utilização m linguagem de montagem:

SHL *#0Ch(0),8,#0Eh(1);*

Desloca oito vezes à esquerda o valor binário do registrador no endereço *#0Ch* contido no banco de registradores *R0* e armazena o resultado no registrador no endereço *#0Eh* contido no banco de registradores *R1*.

4.2.20 Set bit como 1 lógico- BSET

A instrução **BSET**, determina como 1 lógico o valor de um bit de um registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica qual registrador terá seu bit determinado com 1;
- O campo *RB* indica qual bit deve ser determinado como 1;

- O campo RC não é utilizado nessa operação;
- O campo A, se em nível logico "0", indica que o registrador RA está no banco de registradores $R0$. Se nível logico "1" indica o banco $R1$;
- O campo B não apresenta funcionalidade nessa operação;
- O campo C não apresenta funcionalidade nessa operação;
- A descrição matemática da operação pode ser vista na Fig.31:

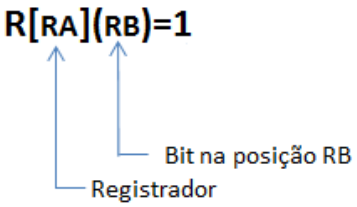


Fig. 31 – Descrição da Operação **BSET**.

- Exemplo de utilização m linguagem de montagem:

BSET *#0Ch(0),8;*

Determina como 1 o bit na posição 8 do registrador no endereço *#0Ch* contido no banco de registradores *R0* e armazena o resultado no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.21 Determina o valor do bit como 0 lógico - BCLEAR

A instrução **BCLEAR** determina como 0 lógico o valor de um bit de um registrador. Essa instrução se utiliza do formato de instruções R expresso na Fig.10.

- O campo RA indica qual registrador terá seu bit determinado com 0;
- O campo RB indica qual bit deve ser determinado como 0;
- O campo RC não apresenta funcionalidade nessa operação;
- O campo A, se em nível logico "0", indica que o registrador RA está no banco de registradores $R0$. Se nível logico "1" indica o banco $R1$;
- O campo B não apresenta funcionalidade nessa operação;
- O campo C não apresenta funcionalidade nessa operação;
- A descrição matemática da operação pode ser vista na Fig.32:

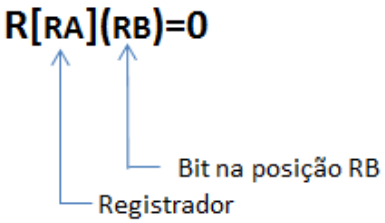


Fig. 32 – Descrição da Operação **BCLEAR**.

- Exemplo de utilização m linguagem de montagem:

BCLEAR *#0Ch(0),8;*

Determina como 0 o bit na posição 8 do registrador no endereço *#0Ch* contido no banco de registradores *R0* e armazena o resultado no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.22 Operação de Multiplicação - MULT

A instrução **MULT** multiplica o valor de dois registradores e armazena o resultado em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador a ser multiplicado;
- O campo *RB* indica o endereço do registrador a ser multiplicado;
- O campo *RC* indica o registrador que receberá o resultado da operação;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.33:
- Exemplo de utilização m linguagem de montagem:

MULT *#0Ch(0),#08h(1),#0Ch(0);*

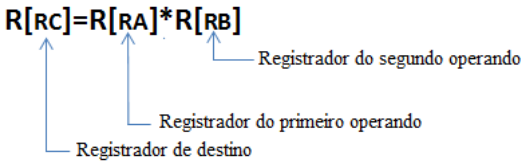


Fig. 33 – Descrição da Operação **MULT**.

Multiplica o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* com o valor do registrador no endereço *#08h* no banco de registradores *R1* e armazena o resultado no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.23 Operação de Divisão - DIV

A instrução **DIV** divide o valor de um registrador por outro e armazena o resultado em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador dividendo;
- O campo *RB* indica o endereço do registrador que contém o valor do divisor;
- O campo *RC* indica o registrador que receberá o resultado da operação;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.34:

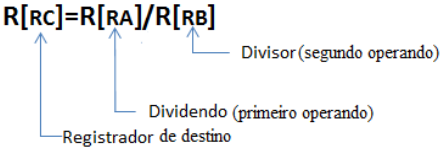


Fig. 34 – Descrição da Operação **DIV**.

- Exemplo de utilização m linguagem de montagem:

DIV *#0Ch(0),#08h(1),#0Ch(0);*

Divide o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* pelo valor do registrador no endereço *#08h* no banco de registradores *R1* e armazena o resultado no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.24 Operação de Resto da divisão - REM

A instrução **REM** divide o valor de um registrador por outro e armazena o resto da operação de divisão em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador dividendo;
- O campo *RB* indica o endereço do registrador que contém o valor do divisor;
- O campo *RC* indica o registrador que receberá o resto da operação de divisão;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*,se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.35:

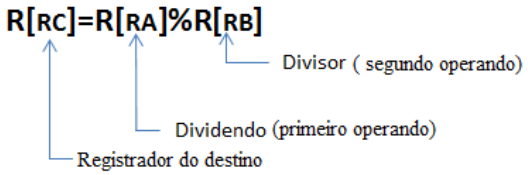


Fig. 35 – Descrição da Operação **REM**.

- Exemplo de utilização m linguagem de montagem:

REM *#0Ch(0),#08h(1),#0Ch(0);*

Divide o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* pelo valor do registrador no endereço *#08h* no banco de registradores *R1* e armazena o resto da operação de divisão no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.25 Operação de Soma - ADD

A instrução **ADD** soma o valor de um registrador com outro e armazena o resultado da operação em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador que terá o valor somado;
- O campo *RB* indica o endereço do registrador que terá o valor somado;
- O campo *RC* indica o registrador que receberá o resultado da operação;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.36:

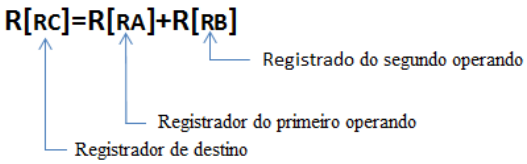


Fig. 36 – Descrição da Operação **ADD**.

- Exemplo de utilização m linguagem de montagem:

ADD *#0Ch(0),#08h(1),#0Ch(0);*

Soma o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* com valor do registrador no endereço *#08h* no banco de registradores *R1* e armazena o resultado da operação no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.26 Operação de Subtração - SUB

A instrução **SUB** subtrai o valor de um registrador pelo o de outro e armazena o resultado da operação em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador minuendo;

- O campo *RB* indica o endereço do registrador subtraendo;

- O campo *RC* indica o registrador que receberá o resultado da operação;

- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- A descrição matemática da operação pode ser vista na Fig.37:

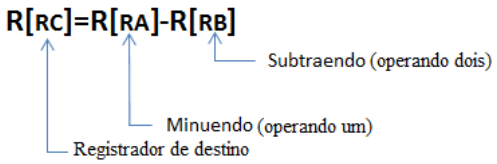


Fig. 37 – Descrição da Operação **SUB**.

- Exemplo de utilização m linguagem de montagem:

SUB *#0Ch(0),#08h(1),#0Ch(0);*

Utiliza o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* como minuendo e o valor do registrador no endereço *#08h* no banco de registradores *R1* como subtraendo e armazena o resultado da operação no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.27 Operação OU lógica - OR

A instrução **OR** realiza a operação lógica *OU* com o valor de um registrador pelo o de outro e armazena o resultado da operação em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador que terá o valor utilizado na operação;

- O campo *RB* indica o endereço do registrador que terá o valor utilizado na operação;

- O campo *RC* indica o registrador que receberá o resultado da operação;

- O campo *A*,se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- A descrição matemática da operação pode ser vista na Fig.38:

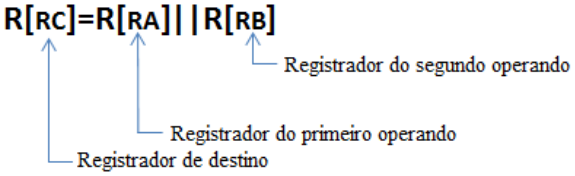


Fig. 38 – Descrição da Operação **OR**.

- Exemplo de utilização m linguagem de montagem:

OR *#0Ch(0),#08h(1),#0Ch(0);*

Realiza a operação lógica OR entre o registrador no endereço *#0Ch* contido no banco de registradores *R0* com o registrador no endereço *#08h* no banco de registradores *R1* e armazena o resultado da operação no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.28 Operação E lógica - AND

A instrução **AND** realiza a operação *E* lógica com o valor de um registrador pelo o de outro e armazena o resultado da operação em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador que terá o valor utilizado na operação;

- O campo *RB* indica o endereço do registrador que terá o valor utilizado na operação;

- O campo *RC* indica o registrador que receberá o resultado da operação;

- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- A descrição matemática da operação pode ser vista na Fig.39:

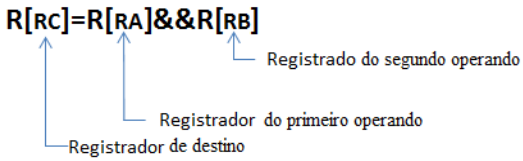


Fig. 39 – Descrição da Operação **AND**.

- Exemplo de utilização m linguagem de montagem:

AND *#0Ch(0),#08h(1),#0Ch(0);*

Realiza a operação lógica *E* entre o registrador no endereço *#0Ch* contido no banco de registradores *R0* com o registrador no endereço *#08h* no banco de registradores *R1* e armazena o resultado da operação no registrador no endereço em *#0Ch* contido no banco de registradores *R0*.

4.2.29 Operação OU lógica exclusiva - XOR

A instrução **XOR** realiza a operação *OU* lógica exclusiva com o valor de um registrador pelo o de outro e armazena o resultado da operação em um terceiro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador que terá o valor utilizado na operação;

- O campo *RB* indica o endereço do registrador que terá o valor utilizado na operação;

- O campo *RC* indica o registrador que receberá o resultado da operação;

- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *B*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;

- A descrição matemática da operação pode ser vista na Fig.40:

- Exemplo de utilização m linguagem de montagem:

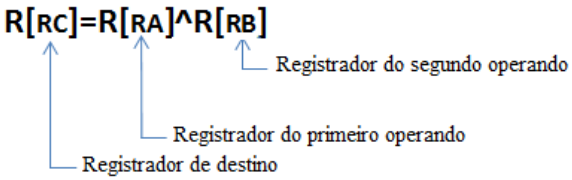


Fig. 40 – Descrição da Operação **XOR**.

XOR *#0Ch(0),#08h(1),#0Ch(0);*

Realiza a operação lógica *OU* exclusiva entre o registrador no endereço *#0Ch* contido no banco de registradores *R0* com o registrador no endereço *#08h* no banco de registradores *R1* e armazena o resultado da operação no registrador no endereço *#0Ch* contido no banco de registradores *R0*.

4.2.30 Operação lógica NOT - NOT

A instrução **NOT** realiza a operação *NOT* com o valor de um registrador e armazena o resultado da operação em um outro registrador. Essa instrução se utiliza do formato de instruções *R* expresso na Fig.10.

- O campo *RA* indica o endereço do registrador que terá o valor utilizado na operação;
- O campo *RB* não apresenta funcionalidade nessa operação;
- O campo *RC* indica o registrador que receberá o resultado da operação;
- O campo *A*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- O campo *B*, não apresenta funcionalidade nessa operação;
- O campo *C*, se em nível logico "0", indica que o registrador *RA* está no banco de registradores *R0*. Se nível logico "1" indica o banco *R1*;
- A descrição matemática da operação pode ser vista na Fig.41:

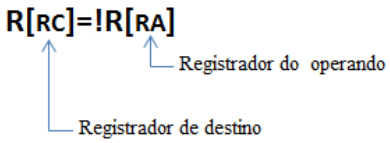


Fig. 41 – Descrição da Operação **NOT**.

- Exemplo de utilização m linguagem de montagem:

***NOT** #0Ch(0),#08h(1);*

Realiza a operação lógica *NOT* com o valor do registrador no endereço *#0Ch* contido no banco de registradores *R0* e armazena o resultado da operação no registrador no endereço *#08h* no banco de registradores *R1*.

4.2.31 Instrução não faz nada- NOP

Na instrução **NOP** nenhuma operação é realizada. Utiliza o formato de instruções *L* expresso na Fig.9.

- O campo *Literal*, não apresenta funcionalidade nessa operação;
- O campo *A*,não apresenta funcionalidade nessa operação;
- O campo *B* não apresenta funcionalidade nessa operação;
- Exemplo de utilização m linguagem de montagem:

***NOP** ;*

Não faz nada nesse ciclo de máquina.

4.2.32 Instrução fim de programa - END

A instrução **END**, paralisa o contador de programa. Esta operação é utilizada somente para fins de verificação. Deve ser usada com cautela, pois a execução desta instrução paralisa completamente o processador e a única maneira de retornar à operação normal é através de um reset. Se utiliza do formato de instruções *L* expresso na Fig.9.

- O campo *Literal*, não apresenta funcionalidade nessa operação;
- O campo *A*,não apresenta funcionalidade nessa operação;
- O campo *B* não apresenta funcionalidade nessa instrução;
- Exemplo de utilização m linguagem de montagem:

***END** ;*

Paralisa o contador de programa.

4.3 Caminho de Dados das Instruções

Para a especificação da organização do microcontrolador é necessário observar os requisitos de projeto e a estrutura básica definida. Partindo desses pontos, é analisado o caminho de dados requerido para a execução de cada uma das instruções.

4.3.1 Requisitos Básicos

Os requisitos de operação são:

- É necessário ler e escrever um valor em um registrador na mesma operação;
- O sentido dos pinos do microcontrolador como saída(OUT) ou entrada (IN) é definido através de um registrador específico;
- O valor dos pinos é definido por registrador específico;
- A leitura de qualquer registrador é assíncrona;
- A gravação de um registrador é síncrona;
- Serão armazenados apenas 16 valores consecutivos de endereço de retorno na pilha;
- As operações da *ULA* geram as seguintes flags: *ZERO*, caso o resultado seja zero; *NEG*, caso o resultado seja negativo; *OVF*, caso o resultado da operação não possa ser representado em 16 bits;
- Todas as flags geradas pela *ULA* são salvas em registrador específico;
- As instruções condicionais são executadas em dois ciclos caso a condição seja verdadeira: um ciclo para análise e outro ciclo para a execução do pulo das instruções;
- Todas as verificações das operações condicionais são feitas através da análise das flags geradas pela ULA;

4.3.2 Caminho de dados para NOP

A instrução **NOP**, no formato de instrução *L*, não apresenta caminho de dados, mas possibilita o incremento do contador de programa. O sistema é composto por um somador e um registrador de programa que armazena o endereço da instrução atual. O esquema pode ser observado na Fig.42.

4.3.3 Caminho de dados para END

A instrução **END** não apresenta caminho de dados, apenas é desabilitada a escrita no registrador do contador de programa.

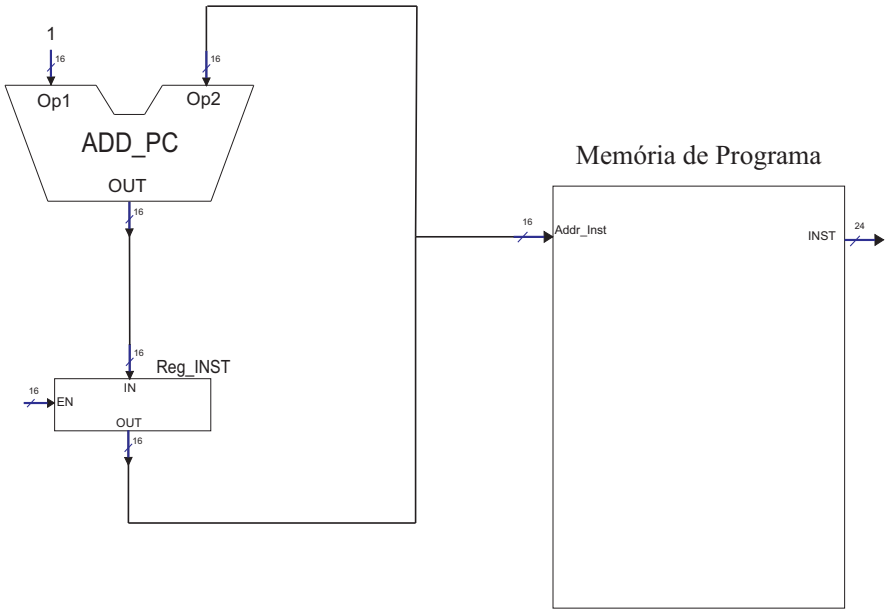


Fig. 42 – Caminho de dados da operação **NOP**.

4.3.4 Caminho de dados para COPY

A instrução **COPY**, no formato de instrução *R*, utiliza apenas o Banco de Registradores. Assim, a instrução requer o seguinte caminho de dados:

1. O campo *RA*, definido pelos bits 14 ao 10 da instrução, determina o endereço a ser lido no banco de registradores;

2. O campo *A*, definido pelo bit 17 da instrução, determina qual banco de registradores (*R0* ou *R1*) será acessado;

3. O valor do registrador lido é direcionado à entrada do dado a ser escrito no banco de registradores;

4. O campo *RC*, definido pelos bits 4 ao 0 da instrução determina o endereço do registrador a ser escrito;

5. O campo *C*, definido pelo bit 15 da instrução determina o banco de registradores(*R0* ou *R1*) que será gravado o valor da entrada de dados do banco de registradores;

6. É necessário habilitar a escrita do registrador;

Determinado o caminho de dados necessário para a instrução **COPY**, é possivel especificar o esquemático do mesmo, que é expresso na Fig.43.

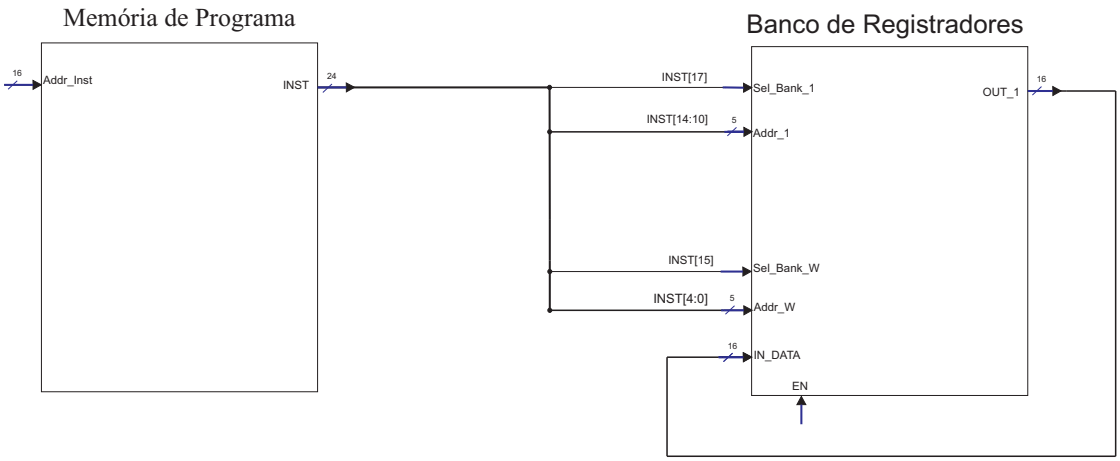


Fig. 43 – Caminho de dados da operação **COPY**.

4.3.5 Caminho de dados para MOVL

A instrução **MOVL**, no formato de instrução *L*, utiliza apenas o Banco de Registradores. Assim, a instrução requer o seguinte caminho de dados:

1. O campo *L*, definido pelos bits 15 ao 0 da instrução é direcionado a entrada de dados do banco de registradores;
2. O campo *A*, definido pelo bit 17 da instrução determina em qual banco de registradores será acessado (*R0* ou *R1*);
3. O endereço de escrita no banco de registradores é definido pelo bloco de controle;
4. É necessário habilitar a escrita do registrador;

Dependendo da instrução que está sendo executada, o endereço de escrita e a entrada de dados do banco de registradores podem ser definidos pela unidade de **CONTROLE** ou pela própria instrução. Na prática não é possível conectar dois barramentos a um mesmo pino de entrada. Assim, é necessário acrescentar um multiplexador(MUX) na entrada de endereço de escrita e entrada de dados. Como observado no caminho de dados, o sinal que seleciona o banco de registradores também é definido pelo campo *A* da instrução. Logo, é acrescentado um MUX a esse pino.
Em vermelho na Fig.44 está destacado o caminho de dados para a instrução **MOVL**.

4.3.6 Caminho de dados para JUMP

A instrução **JUMP**, utiliza o formato de instrução *L*. Está instrução incrementa o valor do contador de programa e necessita do seguinte caminho de dados:

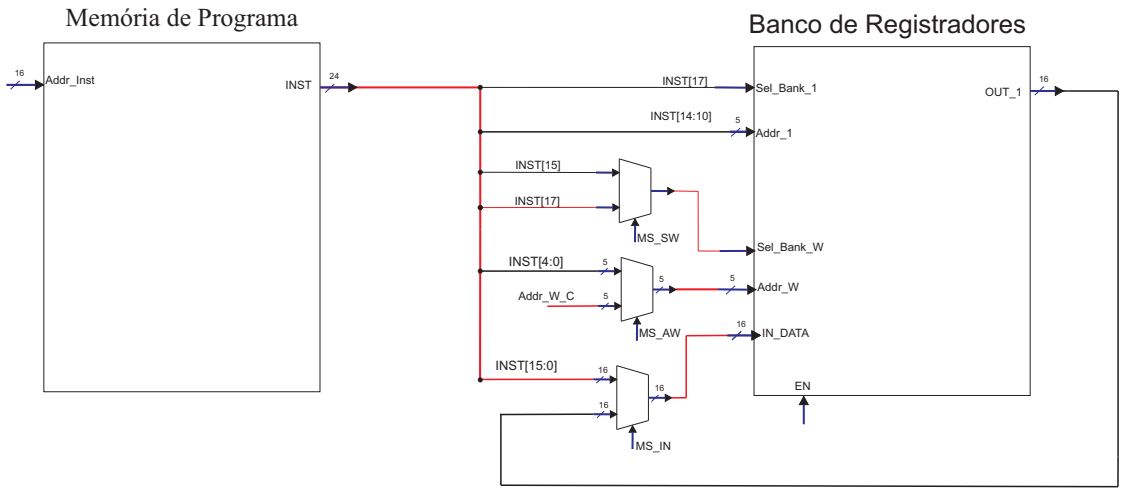


Fig. 44 – Caminho de dados da operação **MOVL**.

1. O campo *L*, composto pelos bits 15 ao 0 da instrução, é direcionado ao somador do contador de programa;

2. O valor da soma entre o endereço de instrução e o campo *L* é direcionado ao registrador do contador de programa;

3. É habilitada a escrita no registrador do contador de programa;

Como é necessário direcionar o valor do campo *L* ao somador do contador de programa e em outras operação será acrescentado apenas um ao endereço de programa, coloca-se um MUX na entrado *OP1* do somador. O caminho de dados para a instrução **JUMP** está destacado em vermelho na Fig.45.

4.3.7 Caminho de dados para CALL

A instrução **CALL** utiliza o formato de instrução *L*. Está instrução incrementa o valor do contador de programa com literal, guardando o endereço de retorno. Para a execução dessa operação o seguinte caminho de dados é necessário:

1. O campo *L*, composto pelos bits 15 ao 0 da instrução, é direcionado ao Somador do contador de programa;

2. O valor da soma entre o endereço de instrução e o campo *L* é direcionado ao registrado do contador de programa;

3. É habilitada a escrita no registrador do contador de programa;

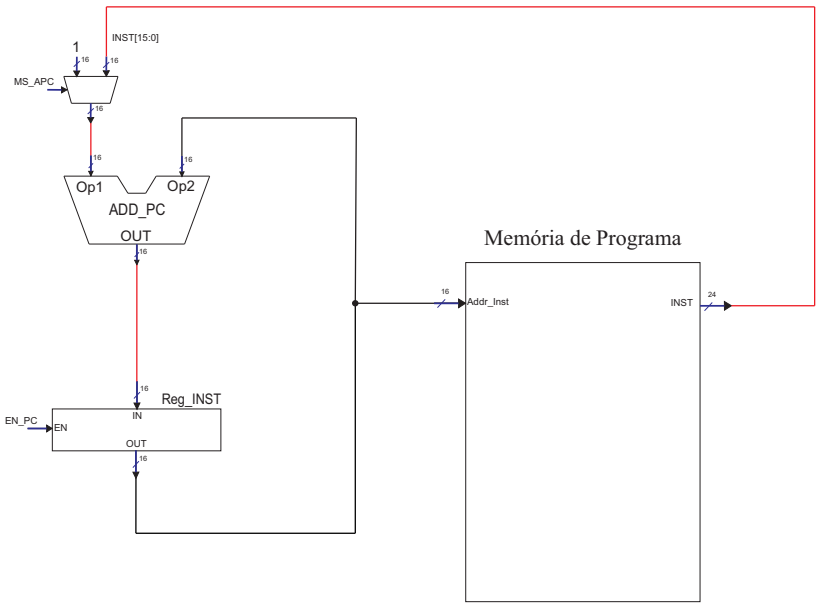


Fig. 45 – Caminho de dados da operação **JUMP**.

Em paralelo à execução do pulo do número de instrução é executado o fluxo de armazenagem do endereço de retorno dado por:

1. O valor do endereço de programa é direcionado a um somador para o incremento de 1, logo o resultado da operação é o endereço de retorno;
2. O endereço de retorno é armazenado em uma pilha($STACK$);
3. O número de endereços salvos é incrementado;

Em vista do caminho de dados necessário para o armazenamento do endereço de retorno, é preciso acrescentar à arquitetura: um somador para incrementar o endereço de programa, uma pilha para armazenar os endereços de retorno($STACK$), um somador para o incremento do número de posições de retorno salvas e um registrador para armazenar o número de posições salvas que indica em qual posição será armazenado um novo endereço de retorno. Assim, em vermelho está destacado o fluxo para o pulo de um número de instruções e em verde o fluxo de armazenagem do endereço de retorno na Fig.46;

4.3.8 Caminho de dados para ADDPC

A instrução **ADDPC** utiliza o formato de instrução M. Está incrementa o valor do contador de programa com um valor armazenado em um registrador. Para a execução dessa operação o seguinte caminho de dados é necessário:

1. O campo RA, composto pelos bits 14 ao 10 é direcionado ao pino de endereço de leitura do banco de registradores;

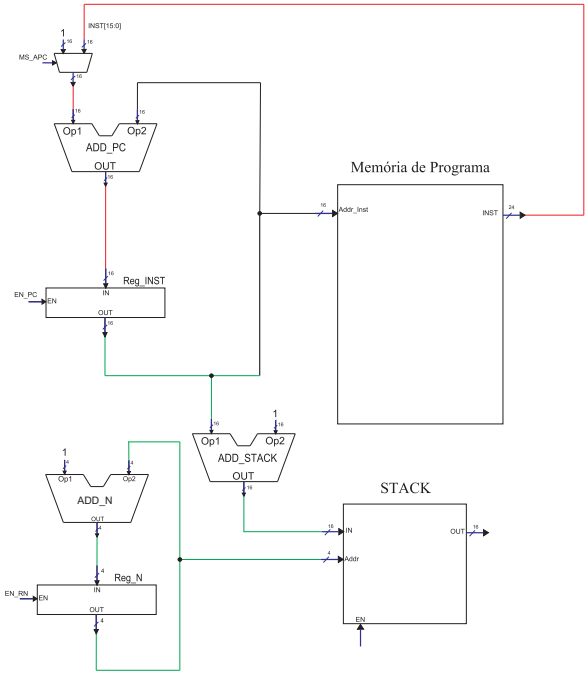


Fig. 46 – Caminho de dados da operação **CALL**.

2. O campo *A*, definido pelo bit 17 é direcionado ao pino do seletor do banco de registradores de leitura;

3. O valor lido é direcionado ao somador do contador de programa;

4. O resultado da soma é direcionador ao registrado de endereço de programa;

5. É habilitada a escrita no registrador de programa;

Com esse caminho de dados é necessário expandir o MUX da entrada do somador do endereço de programa. O caminho de dados para o pulo de um número de instruções está destacado em vermelho na Fig.47.

4.3.9 Caminho de dados para RET

A instrução **RET** utiliza o formato de instrução *L*. Está instrução atribui ao registrador de endereço de programa a última posição de retorno armazenada. Também decrementa o número de posições de retorno de instruções salvas. Tal instrução requer o seguinte caminho de dados:

1. Atribui como endereço ao bloco de pilha *STACK* o número de operações salvas menos 1. Pois o registrador de posições salvas determina o endereço que será salva uma nova posição;

2. O valor lido da pilha *STACK* é direcionado ao registrador de endereço de programa;

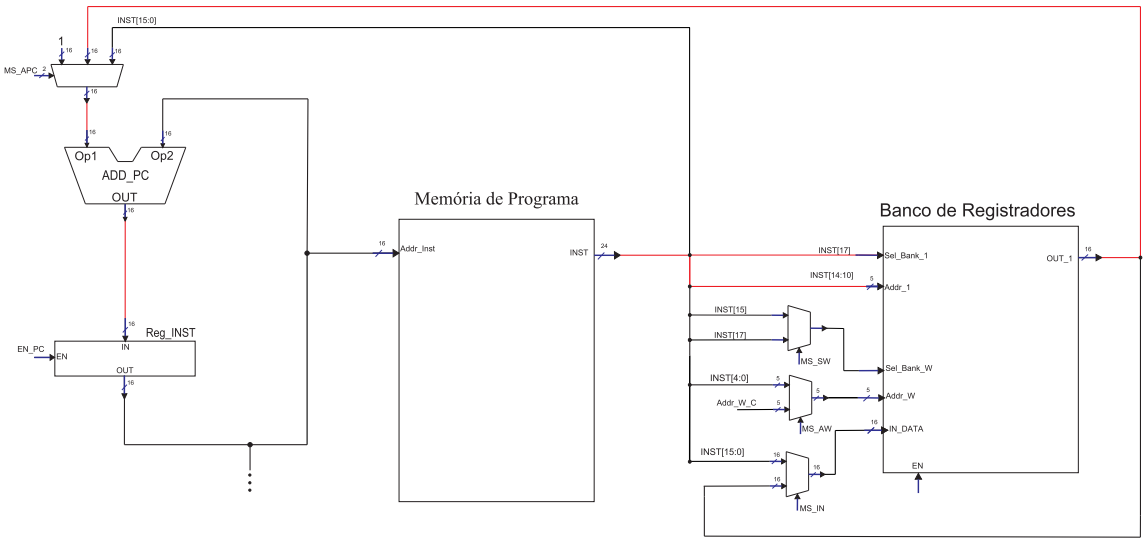


Fig. 47 – Caminho de dados da operação **ADDPC**.

3. É habilitada a escrita no registrador de endereço de programa;
4. É decrementado o valor do registrador do número de posições salvas e habilitada a escrita no mesmo;

O fluxo acima necessita a alteração do somador do número de posições de retorno salvas para um somador/subtrator. De modo a incrementar ou decrementar o valor do registrador. É necessário acrescentar um MUX para definir o valor atribuído ao endereço da pilha *STACK*, pois o mesmo é definido pelo registrador ou pela saída do somador de número de posições salvas. Também é acrescentado um MUX na entrada do registrador de endereço de programa, para selecionar entre o valor lido da pilha *STACK* e a saída do somador de endereço de programa.

O fluxo necessário para a execução dessa instrução está destacado em vermelho na Fig.48

4.3.10 Caminho de dados para RETL

A instrução **RETL** utiliza o formato de instrução *L*. E atribui ao registrador de endereço de programa o último endereço salvo na pilha *STACK*. Também decrementa o valor do registrador do número de posições de retorno salvas e armazena um valor literal no banco de registradores.

O fluxo necessário para gravar o literal no banco de registradores é dado por:

1. O campo *L*, composto pelos bits 15 a 0 da instrução, é direcionado a entrada de dados do banco de registradores;

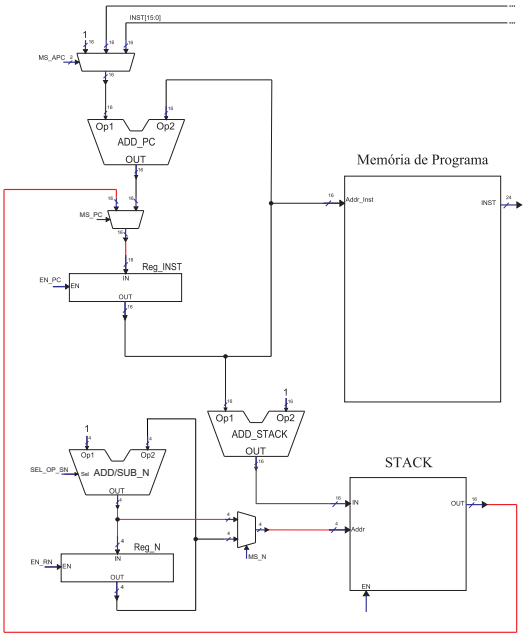


Fig. 48 – Caminho de dados da operação **RET**.

2. O campo *A*, definido pelo bit 17, é direcionado ao pino seletor do banco de registradores para escrita;
3. O bloco de controle define o endereço de escrita;
4. É habilitada a escrita no banco de registradores;

Para o retorno de instrução o seguinte fluxo é executado:

1. Atribui como endereço ao bloco de pilha *STACK* o número de operações salvas menos 1;
2. O valor lido da pilha *STACK* é direcionado ao registrador de endereço de programa;
3. É habilitada a escrita no registrador de endereço de programa;
4. É decrementado o valor do registrador do número de posições salvas e habilitada a escrita no mesmo;

Na Fig.49 está destacado em vermelho o fluxo para se gravar o literal no banco de registradores e em verde o fluxo para retorno de instrução.

4.3.11 Caminho de dados para RM

A instrução **RM** utiliza o formato de instrução *M*. Está instrução escreve o valor da memória de dados para um registrador. Tal instrução utiliza os blocos de memória

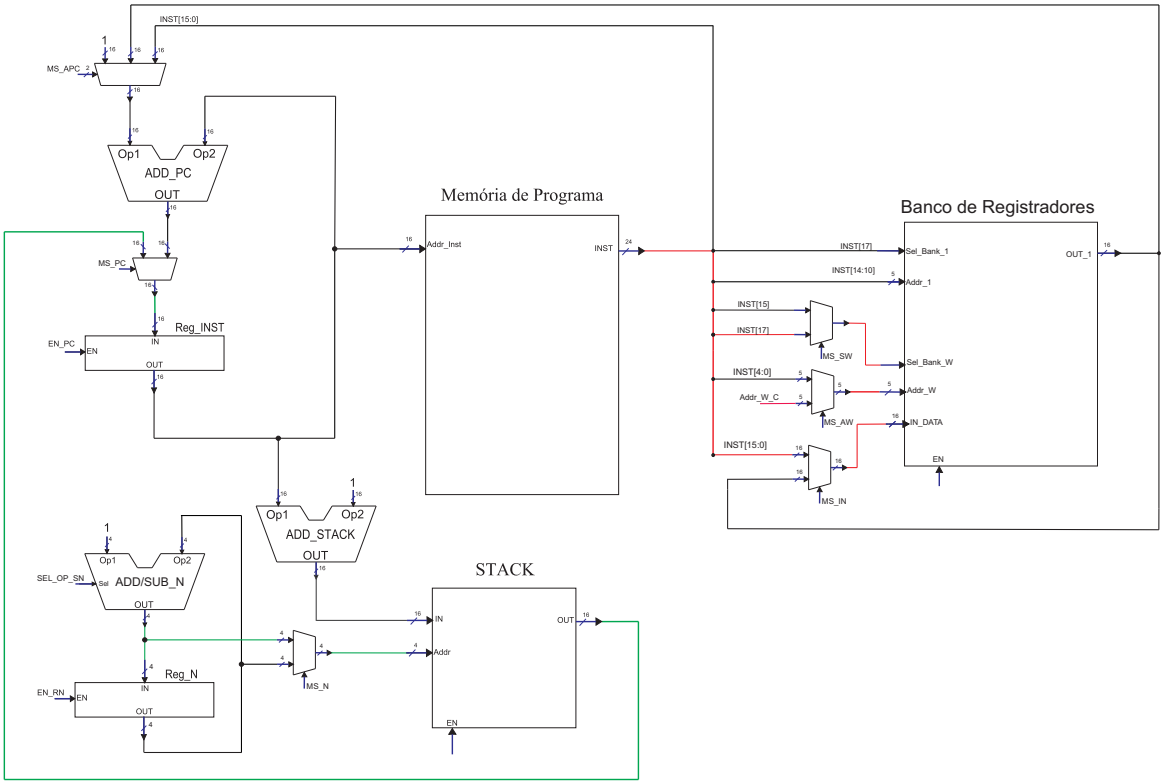


Fig. 49 – Caminho de dados da operação **RETL**.

de dados, banco de registradores e *ULA*. Assim, o fluxo necessário para a execução da instrução é:

1. O campo *BIAS*, composto pelos bits 9 ao 0 da instrução, é direcionado a entrada da ULA;

2. O campo *RA*, composto pelos bits 14 ao 10 da instrução, é direcionado ao pino de endereço de escrita do banco de registradores;

3. O campo *A*, definido pelo bit 17 da instrução, é direcionado ao pino seletor de banco de registradores de escrita;

4. O controle define o endereço do registrador de leitura para compor o endereço de acesso a memória de dados;

5. O controle define o banco de registradores que deve ser feita a leitura;

6. O valor do registrador lido é direcionado à entrada da *ULA*;

7. O resultado da soma na saída da *ULA* é levado ao pino de endereço da memória de dados;

8. O valor lido da memória de dados vai para a entrada de dados do banco de registradores;

9. É habilitada à escrita no banco de registradores;

O fluxo define que é necessário acrescentar MUXs às entradas de endereço de leitura e seletor de banco de registradores de leitura. Também devem ser expandidos os MUXs do pino de endereço de escrita e entrada de dados do banco de registradores.

Na Fig.50 está destacado em vermelho o caminho de dados para a execução da instrução.

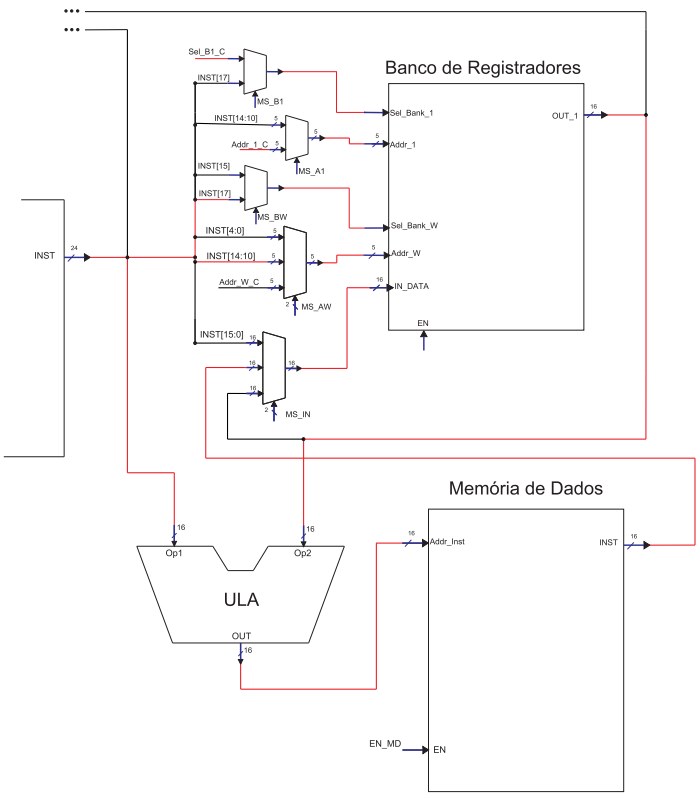


Fig. 50 – Fluxo de dados da operação **RM**.

4.3.12 Caminho de dados para WM

A instrução **WM** utiliza o formato de instrução *M*. Está instrução escreve o valor de um registrador para a memória de dados. Essa instrução apresenta o seguinte caminho de dados:

1. O campo *BIAS*, composto pelos bits 9 ao 0, é levado ao entrada da *ULA*;

2. O controle define o endereço do registrador a ser lido para compor o endereço de leitura da memória de dados;
3. O controle determina em qual banco de registradores está contido o registrador a ser lido;
4. O valor do registrador lido é direcionado à outra entrada da *ULA*;
5. O resultado da soma da *ULA* é levado à entrada de endereço da memória de dados;
6. O campo *RA*, composto pelos bits 14 ao 10 da instrução, é levado a um segundo pino de endereço de leitura da memória de dados;
7. O campo *A*, definido pelo bit 17 da instrução, é conectado ao pino seletor do banco de registradores de leitura para o endereço dois;
8. O valor lido é direcionado à entrada de dados da memória de dados;
9. É habilitada a escrita na memória de dados;

Como pode ser observado no caminho de dados, é necessário acrescentar mais uma estrutura de leitura de registradores para sua execução.

Assim, na Fig.51 está destacado em vermelho o caminho de dados para a instrução.

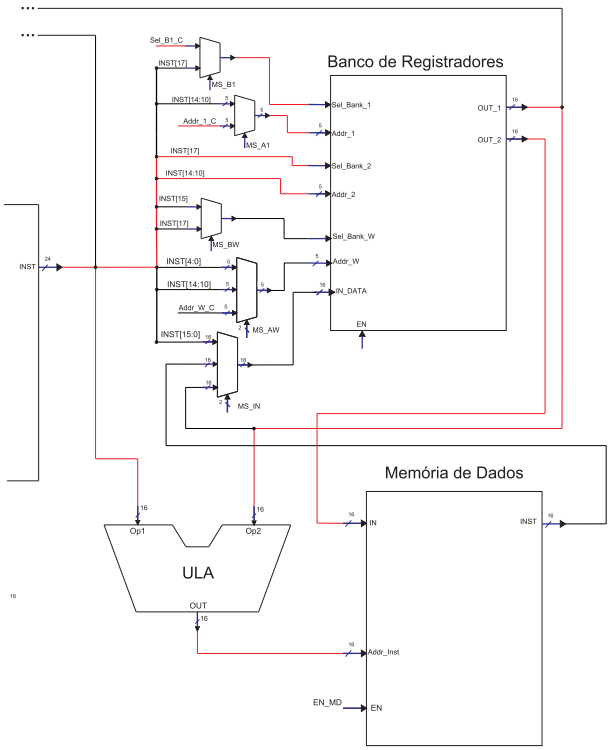


Fig. 51 – Caminho de dados da operação **WM**.

4.3.13 Caminho de dados para BL

A instrução **BL** utiliza o formato de instrução *R*. Se baseia nas flags NEG e ZERO da operação de subtração da *ULA* para determinar se o valor armazenado em um registrador é maior que em outro registrador. A instrução requer dois caminho de dados, um para a análise da condição e outro para o pulo da instrução. Se a condição for falsa deve-se apenas somar mais 1 ao contador de programa e será executada só primeiro caminho de dados. Se verdadeira deve paralisar o contador, para no próximo ciclo somar o valor do pulo ao endereço de programa atual. Assim, os fluxos necessários são:

Fase de verificação da condição:

1. O campo *RA*, compostos pelos bits 14 ao 10, é direcionado ao endereço de leitura 1 do banco de registradores;

2. O campo *RB*, compostos pelos bits 9 ao 5, é direcionado ao endereço de leitura 2 do banco de registradores;

3. O campo *A*, definido pelo bit 17 é direcionado ao seletor de banco do registrador de leitura 1;

4. O campo *B*, definido pelo bit 16 é direcionado ao seletor de banco do registrador de leitura 2;

5. O valor lido da saída 1 é direcionado a entrada da *ULA*;

6. O valor lido da saída 2 é direcionado a outra entrada da *ULA*;

7. As flags *NEG* e *ZERO* geradas através da operação de subtração são avaliadas para determinar se a condição é verdadeira ou falsa;

Observando o fluxo acima, é necessário acrescentar um MUX a entrada 1 da *ULA* para receber o valor lido da saída 2 do banco de registradores. É acrescentado um MUX ao pinos de entrada de leitura 2 e seleção de banco 2. A análise da condição pede que as duas flags *NEG* e *ZERO* estejam em nível lógico 0, assim haverá a paralisação do contador de programa. Logo, para uma resposta mais rápida será realizada uma operação lógica OU entre os dois sinais e direcionado ao pino que habilita a escrita no registrador de endereço de programa. Desta forma, é acrescentado um MUX no pino de de habilita escrita do registrador de endereço de programa.
O fluxo necessário para a realização da primeira etapa está destacado em vermelho na Fig.52. Para a segunda etapa caso a condição seja verdadeira, ocorrerá o seguinte caminho de dados.

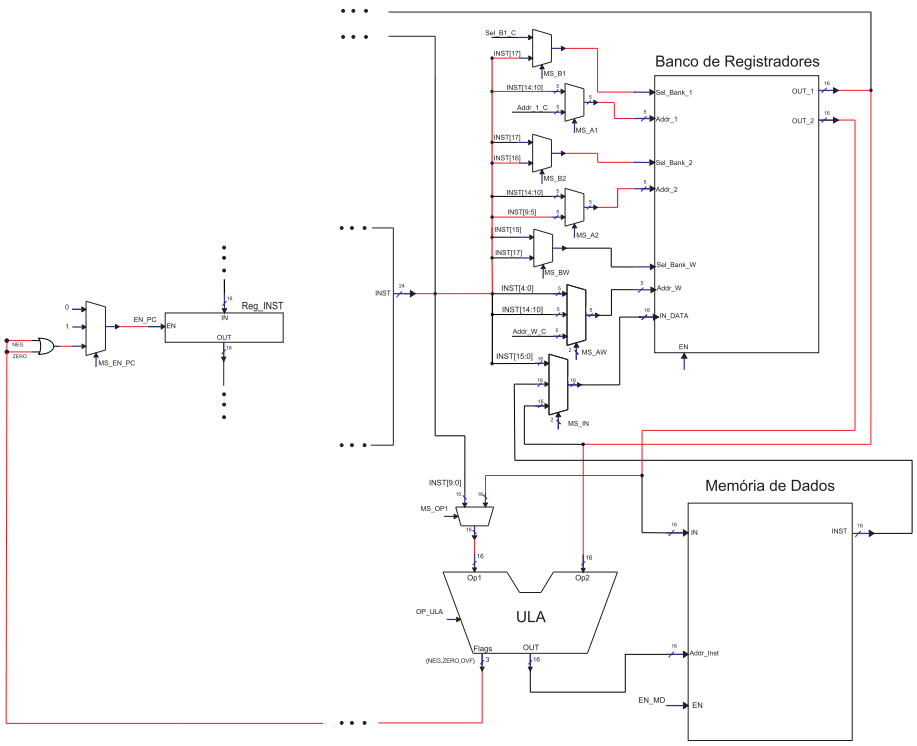


Fig. 52 – Caminho de dados da parte 1 da operação **BL**.

1. O campo *RC*, composto pelos bits 4 ao 0 da instrução, definem o endereço de leitura 1;

2. O campo *C*, definido pelo bit 15 determina em qual banco está contido o registrado de leitura 1;

3. O valor lido é direcionado à entrada do somador do contador de programa;

4. O resultado da soma é direcionado ao registrador de endereço de programa;

5. É habilitada a escrita no registrador;

O fluxo acima requer uma expansão do MUX do endereço e seletor de banco da leitura 1. Está destacado em vermelho na Fig.53 o fluxo necessário para a execução da segunda etapa.

4.3.14 Caminho de dados para as demais instruções condicionais entre registradores

As demais instruções condicionais entre registradores, sendo elas: **BS**, **BLE**, **BSE**, **BE** e **BNE**, apresentam o mesmo caminho de dados que a instrução **BL**. E analisam as flags *ZERO* e *NEG* geradas pela operação de subtração dos registradores do campo *RA* e *RB*. Mas a única alteração está na condição de bloqueio da escrita no registrador de

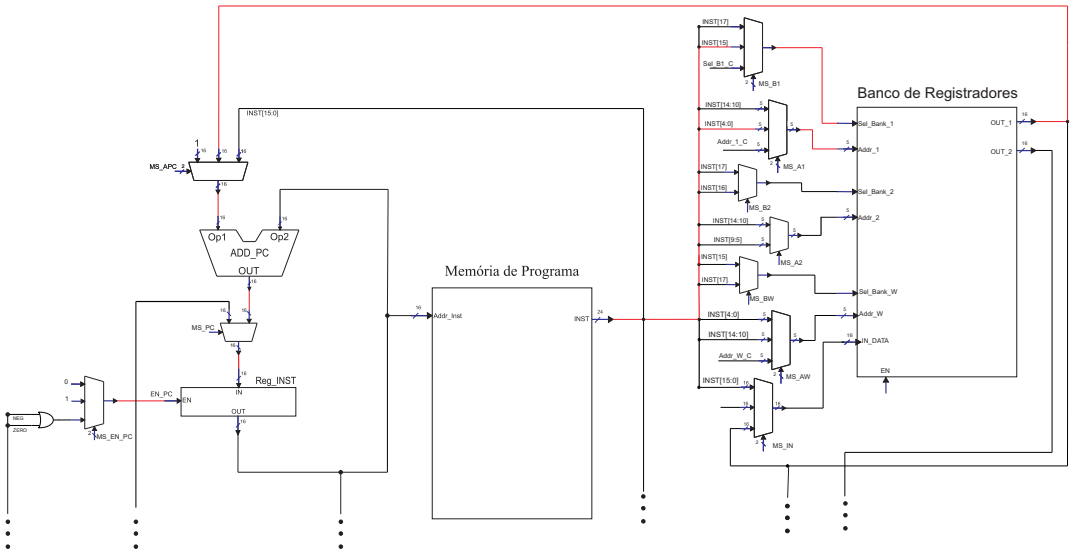


Fig. 53 – Caminho de dados da parte 2 da operação **BL**.

endereço de programa em comparação com a instrução *BL*. Assim, está descrito abaixo as condições para o bloqueio do registrador de endereço de programa, bem como, as alterações no MUX do pino de habilitação de escrita, gerada por cada uma das instruções citadas acima:

- Instrução **BS**: Pulo condicional se o valor de um registrador é menor que de outro, logo para haver o bloqueio, o resultado da operação deve ser negativo e não pode ser zero. Assim, uma operação ”OR” lógica entre a flag *ZERO* e a flag *NEG* barrada satisfaz a condição. A alteração na estrutura está apresentada na Fig.54;

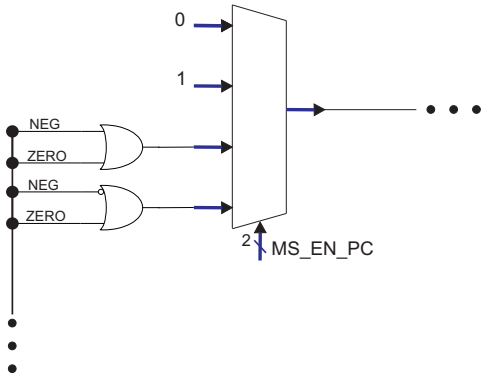


Fig. 54 – Alteração na estrutura para satisfazer a operação **BS**.

- Instrução **BLE**: Pulo condicional se o valor de um registrador é maior ou igual ao de outro registrador, logo para haver o bloqueio o resultado da operação tem que ser

zero ou positivo. Assim, uma operação ”E” lógica entre a flag *ZERO* barrada e a flag *NEG*, satisfaz a condição. A alteração na estrutura está apresentada na Fig.55;

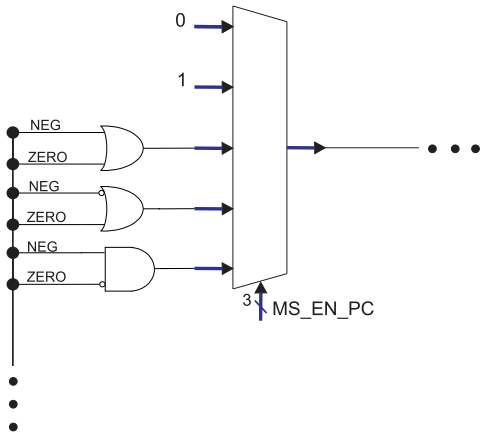


Fig. 55 – Alteração na estrutura para satisfazer a operação **BLE**.

- Instrução **BSE**: Pulo condicional se o valor de um registrador é menor ou igual ao de outro registrador, logo para haver o bloqueio o resultado da operação tem que ser zero ou negativo. Assim, uma operação ”OU” negada(NOR) lógica entre a flag *ZERO* e a flag *NEG*, satisfaz a condição. A alteração na estrutura está apresentada na Fig.56;

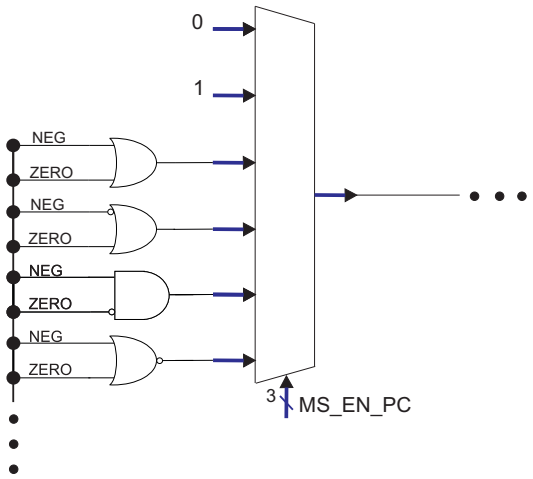


Fig. 56 – Alteração na estrutura para satisfazer a operação **BSE**.

- Instrução **BE**: Pulo condicional se o valor de um registrador é igual ao de outro registrador, logo para haver o bloqueio o resultado da operação tem que ser zero, para isso ser satisfeito a flag *ZERO* deve ser levada barrada para o pino que habilita a escrita no registrador de endereço de programa. A alteração na estrutura está apresentada na Fig.57;

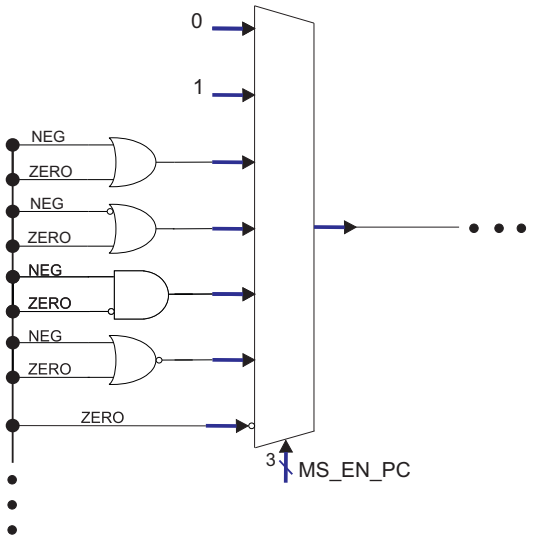


Fig. 57 – Alteração na estrutura para satisfazer a operação **BE**.

- Instrução **BNE**: Pulo condicional se o valor de um registrador é diferente ao de outro registrador, logo para haver o bloquei o resultado da operação tem que ser diferente de zero, para isso ser satisfeito a flag *ZERO* deve ser levada para o pino que habilita a escrita no registrador de endereço de programa. A alteração na estrutura está apresentada na Fig.58;

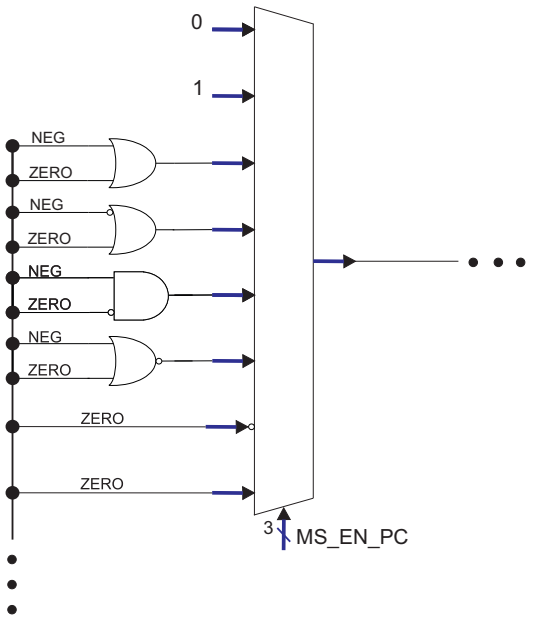


Fig. 58 – Alteração na estrutura para satisfazer a operação **BNE**.

4.3.15 Caminho de dados para BBCLEAR e BBSET

A instrução **BBCLEAR** utiliza o formato de instrução *R*. E realiza o pulo de um certo número de instrução caso o valor de um bit indicado pelo usuário seja zero. Para isso será acrescentado uma operação na *ULA*, denominada *BIT* que conduz o bit indicado ao pino de flags da *ULA*. Desta forma serão atendidos os requisitos indicados na seção 4.3.1. A nova flag será denominada *BIT*. Assim, o fluxo para execução da instrução está dividido em duas partes as quais são:

Verificação da condição:

1. O campo *RA*, composto pelos bits 14 ao 10, é direcionado à entrada de leitura 1;
2. O campo *A*, definido pelo bit 17, determina em qual banco de registradores deve ser lido o endereço 1;
3. O campo *RB*, composto pelos bits 9 ao 5 da instrução, é direcionado a entrada da *ULA*;
4. O valor lido 1 é levado a outra entrada da *ULA*;
5. A flag *BIT* resultante da operação da *ULA* é levada ao pino que habilita a escrita no registrador de endereço de programa;

Para satisfazer o fluxo acima é necessário expandir o MUX da entrada *OP1* da *ULA* para a entrada do campo *RB*. Também é expandido o MUX do pino que habilita a escrita do registrador de endereço de programa.

Assim, a primeira parte do fluxo para a execução dessa instrução está destacado em vermelho na Fig.59.

O fluxo para o pulo do número de instruções é o mesmo apresentado na seção 4.3.13 e pode ser observado na Fig.53.

A instrução **BBSET** utiliza o formato de instrução *R*. E realiza o pulo de um certo número de instrução caso o valor de um bit indicado pelo usuário seja um. A instrução segue o fluxo anterior e a única modificação na estrutura é a expansão do MUX do pino que habilita a escrita no registrador de endereço de programa. Pois é necessário levar a flag *BIT* barrada para desabilitar a escrita caso a condição seja verdadeira. A modificação na estrutura é apresentada na Fig.60.

4.3.16 Caminho de dados para SHR e SHL

As instruções **SHR** e **SHL** utilizam o formato de instrução *R*. E deslocam à direita ou à esquerda o valor de um registrador. O fluxo necessário para execução dessas instruções é:

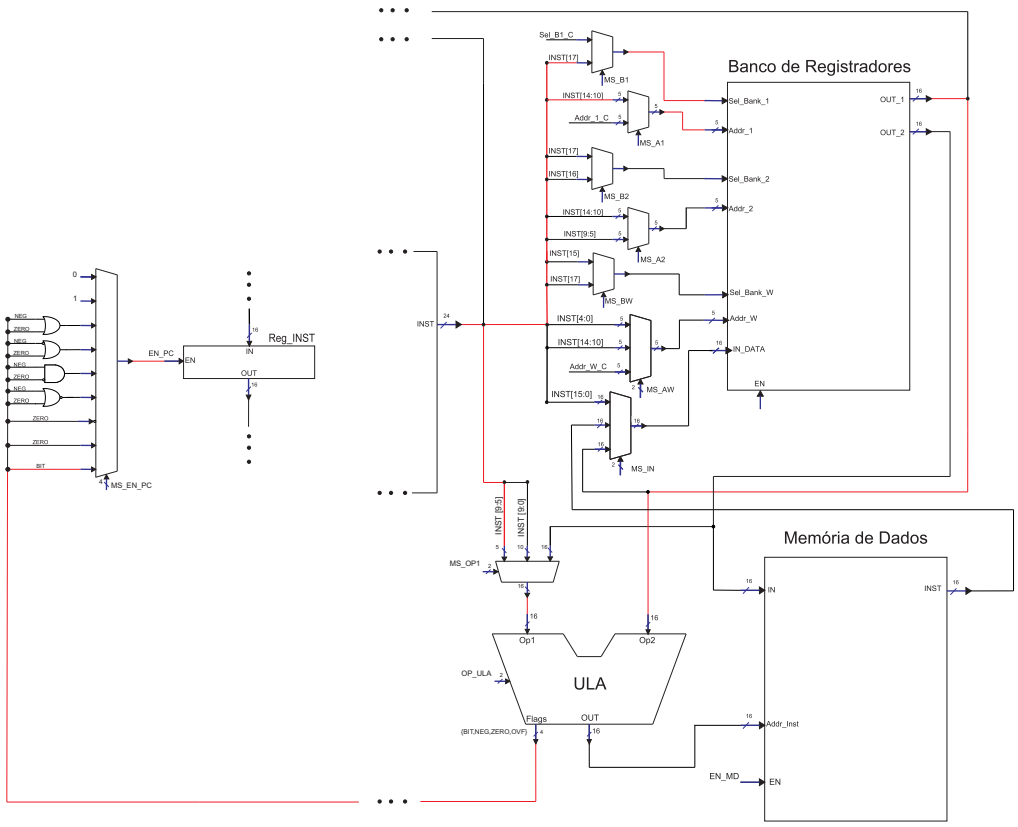


Fig. 59 – Alteração na estrutura para satisfazer a operação **BBCLEAR**.

1. O campo *RA*, composto pelos bits 14 ao 10, é direcionado ao endereço de leitura 1;
2. O campo *A*, definido pelo bit 17, indica em qual banco de registradores está contido o endereço lido 1;
3. O valor lido 1 é direcionado a entrado da *ULA*;
4. O campo *RB*, composto pelos bits 9 ao 5 da instrução, é direcionado à entrada da *ULA* para indicar quantas casas o valor deve ser deslocado;
5. O resultado da operação é direcionado à entrada de dados do banco de registradores;
6. O campo *RC*, composto pelos bits 4 ao 0, é direcionado ao pino do endereço de escrita do banco de registradores;
7. O campo *C*, definido pelo bit 15, indica em qual banco de registradores está contido o endereço de escrita.
8. É habilitada a escrita no banco de registradores;

O fluxo acima exige a expansão do MUX da entrada de dados do banco de registradores. O caminho de dados para a execução da instrução está destacado em vermelho na Fig.61.

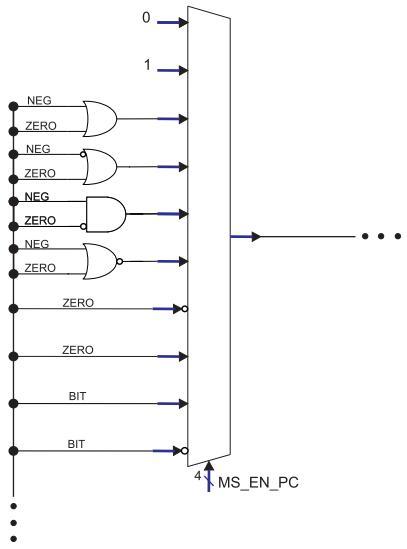


Fig. 60 – Alteração na estrutura para satisfazer a operação **BBSET**.

4.3.17 Caminho de dados para as instruções BSET e BCLEAR

As instruções **BSET** e **BCLEAR** utilizam o formato de instruções *R*. E determinam o valor de um certo bit como 1 lógico ou como 0 lógico. O fluxo requerido para essa instrução é o mesmo apresentado na seção 4.3.16, na Fig.61 está destacado o fluxo para essa instrução.

4.3.18 Caminho de dados para as instruções Aritméticas e Lógicas

As instruções aritméticas **MULT**, **DIV**, **REM**, **ADD**, **SUB**, **OR**, **AND** e **XOR** apresentam o mesmo caminho de dados que é apresentado a baixo:

1. O campo *RA*, compostos pelos bits 14 ao 10, é direcionado ao endereço de leitura 1 do banco de registradores;

2. O campo *RB*, compostos pelos bits 9 ao 5, é direcionado ao endereço de leitura 2 do banco de registradores;

3. O campo *A*, definido pelo bit 17 é direcionado ao seletor do banco do registradores de leitura 1;

4. O campo *B*, definido pelo bit 16 é direcionado ao seletor do banco do registradores de leitura 2;

5. O valor lido da saída 1 é direcionado à entrada da *ULA*;

6. O valor lido da saída 2 é direcionado à outra entrada da *ULA*;

7. O resultado da operação é direcionado à entrada de dados do banco de registradores;

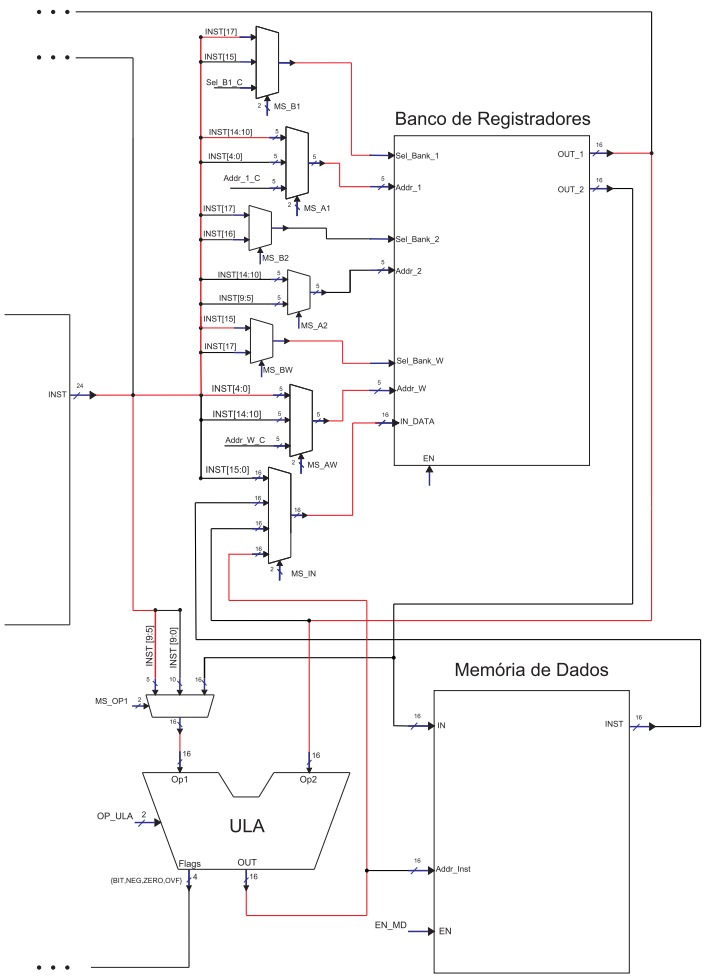


Fig. 61 – Caminho de dados para as operações **SHR** e **SHL**.

8. O campo *RC*, composto pelos bits 4 ao 0, é direcionado ao pino do endereço de escrita do banco de registradores;

9. O campo *C*, definido pelo bit 15, indica em qual banco de registradores está contido o endereço de escrita.

10. É habilitada a escrita no banco de registradores;

O fluxo para a operação **NOT** é o mesmo acima, com a diferença de não ser executado o fluxo do dado lido 2. A única modificação na estrutura é o aumento no número de bits que são necessários para indicar qual operação que está sendo executada pela *ULA*. A qual totaliza 14 operações executadas, sendo necessários 4 bits para indicar a operação que deve ser executada.

Por fim o fluxo para executar cada uma das operações citadas nessa seção está destacado em vermelho na Fig.62;

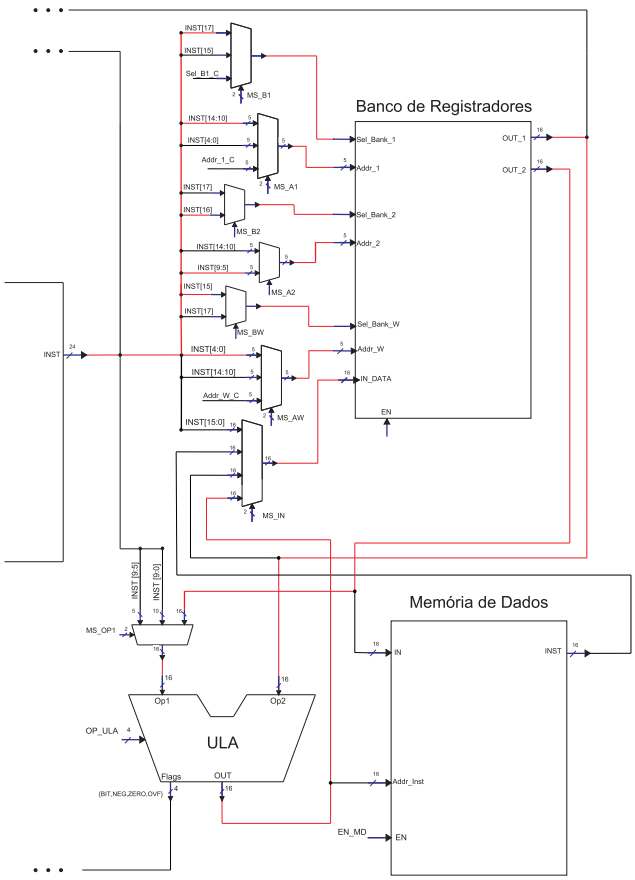


Fig. 62 – Caminho de dados para as operações aritmética e lógica.

4.4 Organização de Blocos Complexos

Este seção apresenta a especificação da estrutura organizacional dos blocos mais complexos como a *ULA*, Banco de Registradores e a unidade de *CONTROLE* para atender o caminho de dados determinado pela seção anterior.

4.4.1 Especificação do Banco de Registradores

O banco de registradores é composto por dois bancos denominados *R0* e *R1*. Para sua elaboração é necessário atentar aos requisitos básicos e os definidos pelo caminho de dados. Todos os requisitos necessários são:

- Apresentar dois bancos de registradores com 32 registradores de 16 bits cada;
- Apresentar duas portas IN/OUT as quais estão conectadas ao registradores do banco *R0*;
- A direcionalidade das portas IN/OUT é definida por registradores da banco *R1*;

- A leitura de todos os registradores é assíncrona, logo o endereço de leitura é levando diretamente aos bancos de registradores. Assim, a seleção de qual banco de registradores o dado é lido, é feito através de um MUX;
- A escrita é síncrona e para ser realizada é preciso ser habilitada pelo seletor de banco de registradores e pelo pino de habilitar escrita;
- Todas as flags geradas pelas operações da *ULA* são salvas em registrador específico e a escrita no registrador é habilitada quando essa operações são realizadas;
- Dois registradores devem ser lidos ao mesmo tempo de cada um dos banco de registradores;

Definido as especificações do Banco de registradores a estrutura que as satisfazem é expressa pela Fig.63;

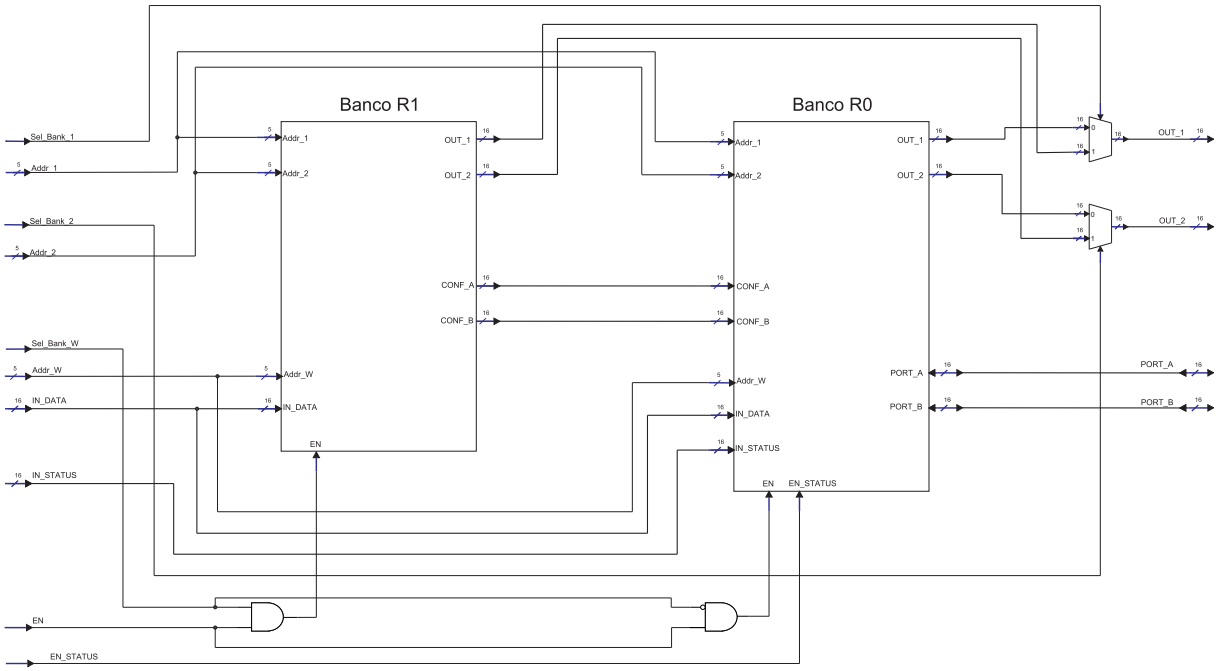


Fig. 63 – Organização do Banco de Registradores

4.4.2 Especificação da ULA

A unidade lógica aritmética deve executar 14 operações diferentes. A *ULA* é divida em blocos que realizaram as operações e a suas execuções são feita de maneira assíncrona. Assim, é selecionado qual bloco receberá os valores de entrada da *ULA* para evitar o funcionamento desnecessário dos demais. O resultado de saída é selecionado entre as saídas dos blocos. Então os blocos ficam divididos das seguinte maneira:

- Bloco *ADD/SUB*: Realiza as operações de soma e subtração;
- Bloco *DIV/REM*: Realiza as operações de divisão e resto;
- Bloco *BIT SET/CLR*: Realiza as operações **BSET** e **BCLEAR**;
- Bloco *SHIFT R/L*: Realiza as operações de deslocamento a esquerda e deslocamento a direita;
- Bloco *MULT*: Realiza a operação de multiplicação;
- Bloco *OP_LOGIC*: Realiza as operações lógicas **AND**, **OR**, **XOR** e **NOT**;
- Bloco *BIT*: Utilizado nas operações de pulo condicional binário;

Além dos blocos de operações a *ULA* apresenta mais 2 blocos, um para a detecção de resposta zero e um decodificador para indicar qual operação deve ser executada pelos blocos com mais de uma função. O esquema da *ULA* pode ser observado na Fig.64.

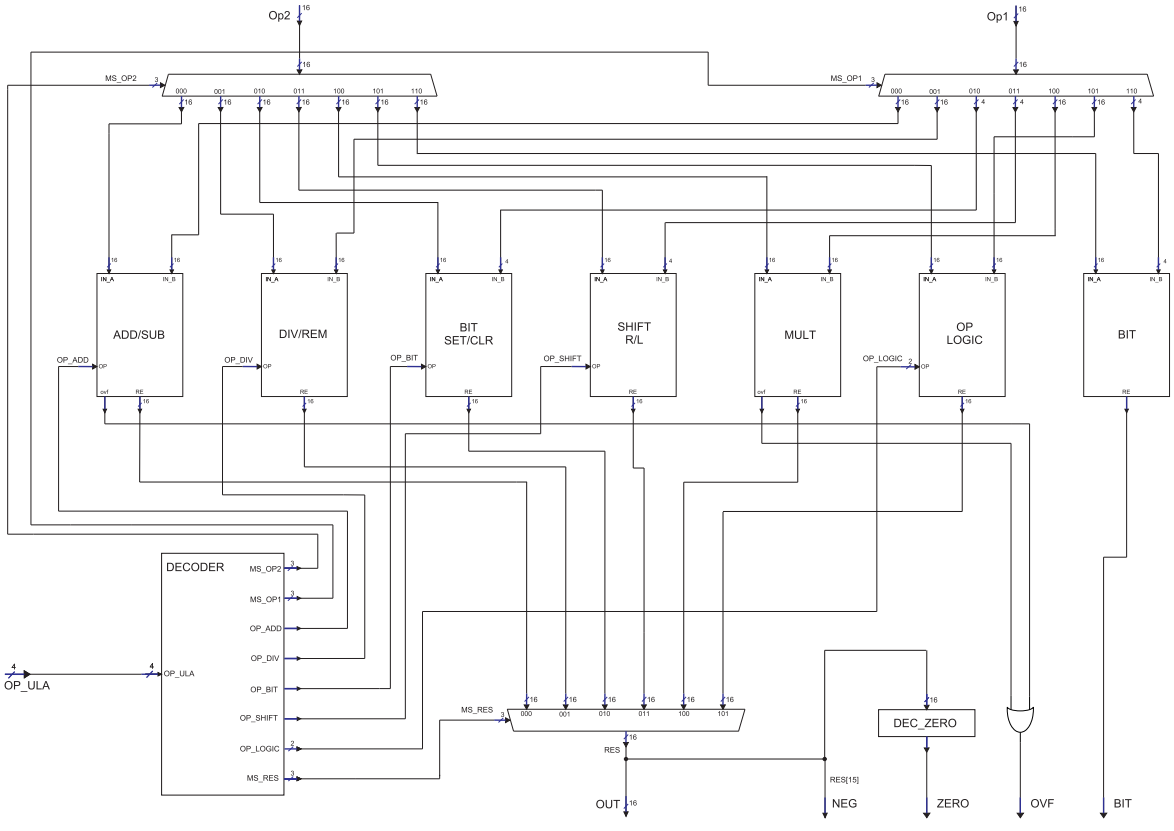


Fig. 64 – Organização da Unidade Lógica Aritmética

4.4.3 Unidade de CONTROLE

A unidade de CONTROLE tem uma função vital dentro do microcontrolador, que é controlar o caminho de dados que deve ser executado, dependendo da instrução executada e as flags geradas por essas operações. Desta forma, o código da instrução é direcionado a ele, como também as flags geradas pelas operações da *ULA*. Como saída há os pinos de seleção dos MUX, os pinos habilita escrita e seleção de operação da *ULA*. Então o controle totaliza 6 pinos de entrada e 22 pinos de saída, a funcionalidade de cada sinal está contida na Tabela 2. O bloco de controle é apresentado na Fig.65.

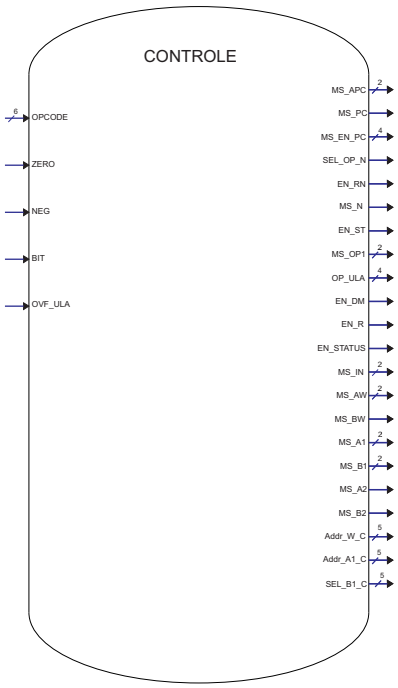


Fig. 65 – Unidade de *CONTROLE*

4.5 Estrutura Organizacional do PAMPIUM I

Através da análise dos caminhos de dados requiridos pelas instruções implementadas, foram definidas as especificações de cada bloco, as entradas e saídas e os MUXs necessários para seleção de dados. Desta forma, foi gerado o caminho de dados final do microcontrolador PAMPIUM I. A unidade de CONTROLE e os sinais de controle não foram colocados no estrutura organizacional final, mas fica implícito a sua conexão aos demais blocos.

A estrutura organizacional final do PAMPIUM I está expressa na Fig. 66.

Tabela 2 – Sinais de Controle.

Sinal	**Descrição Funcional**
MS_APC	Seletor do dado de entrada do somador do endereço de programa
MS_PC	Seletor do dado de entrada do registrador do endereço de programa
MS_EN_PC	Seletor do sinal de habilita escrita do registrado de endereço de programa
SEL_OP_N	Seleciona o tipo de operação executada pelo somador de número de posições de retorno salvas.
EN_RN	Habilita escrita no registrador do número de posições de retorno salvas.
MS_N	Seletor do endereço do STACK.
EN_ST	Habilita escrita no STACK.
MS_OP1	Seletor do dado de entrada no pino OP1 da *ULA*.
OP_ULA	Seleciona a operação executada pela *ULA*.
EN_DM	Habilita escrita na Memória de Dados.
EN_R	Habilita a escrita no Banco de Registradores.
EN_STATUS	Habilita escrita na registrador STATUS.
MS_IN	Seletor do dado de entrada de dados do Banco de Registradores.
MS_AW	Seletor do endereço da entrada de escrita do Banco de Registradores.
MS_BW	Seletor do sinal que indicará o Banco de Registrador a ser realizada a escrita.
MS_A1	Seletor do endereço de leitura 1.
MS_B1	Seletor do sinal que indicará o Banco de Registradores do endereço de leitura 1.
MS_A2	Seletor do endereço de leitura 2.
MS_B2	Seletor do sinal que indicará o Banco de Registradores do endereço de leitura 2.
Addr_W_C	Endereço de escrita do Banco de Registradores indicado pela Unidade de Controle.
Addr_A1_C	Endereço de leitura 1 do Banco de Registradores indicado pela Unidade de Controle.
Addr_B1_C	Banco de Registradores da entrada de leitura 1 indicado pela Unidade de Controle.

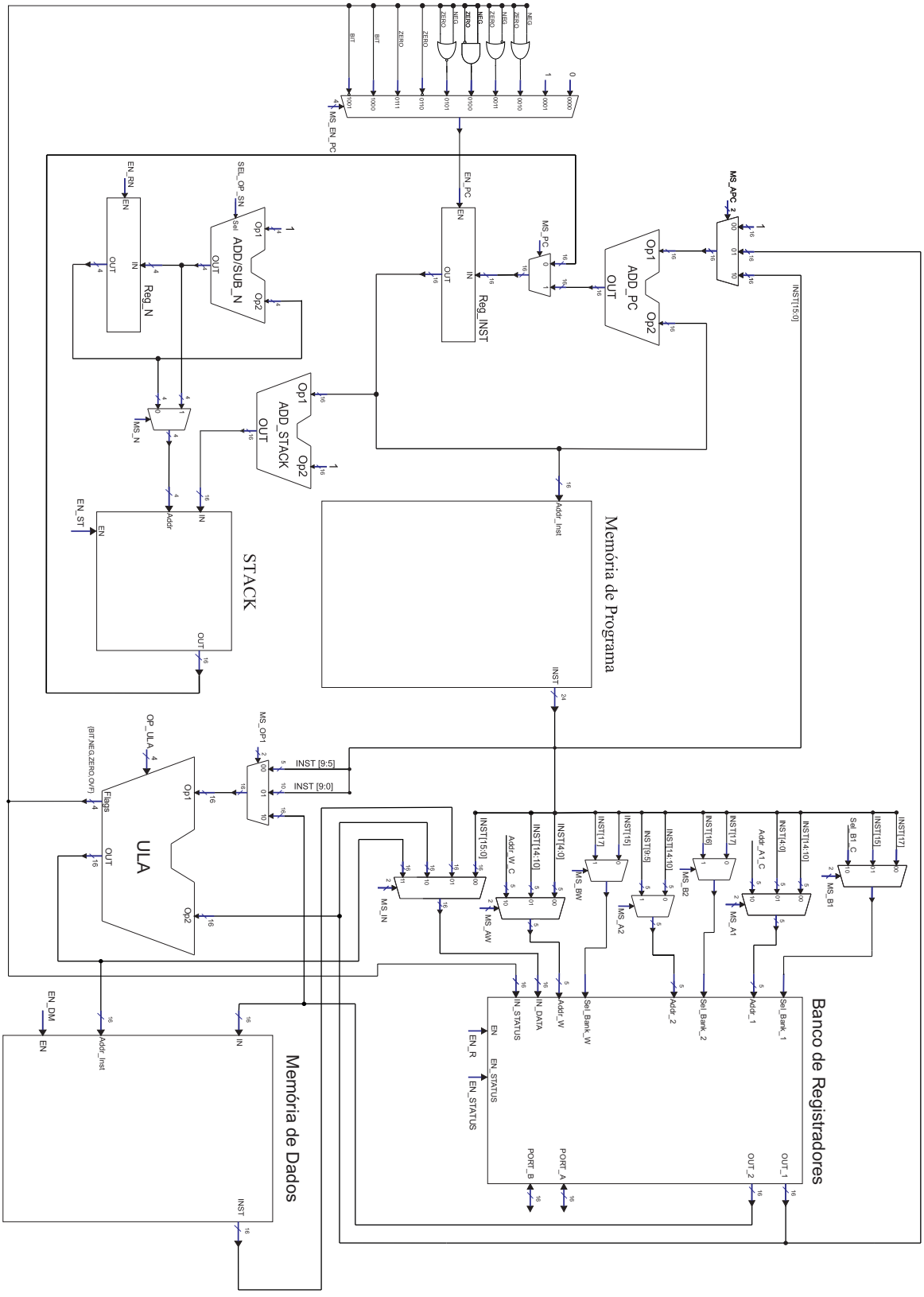


Fig. 66 – Caminho de dados completo do PAMPIUM I

5 IMPLEMENTAÇÃO

Este capítulo apresenta o implementação em linguagem HDL SystemVerilog da estrutura organizacional apresentada no capítulo anterior. Mostra também detalhadamente as decisões de projeto e os testes de validação via software das instruções implementadas.

5.1 Implementação em SystemVerilog

SystemVerilog é um linguagem de descrição de hardware, verificação e síntese. É derivada da linguagem de hardware Verilog, desenvolvida pela IEEE em 1995 e com uma nova versão em 2001. A linguagem Verilog não trazia muitas vantagens na área de testabilidade computacional. Assim, foi criado o SystemVerilog que engloba os elementos de descrição de hardware com os sistemas de verificação de linguagem VERA e Testbuider(SUTHERLAND SIMON DAVIDMANN, 2006). Systemverilog foi escolhida para o desenvolvimento desse projeto por apresentar características que facilitam o desenvolvimento de hardware, como reduzido código em comparação com outras linguagens HDL e fácil testabilidade.

Para o implementação em linguagem HDL SystemVerilog foi utilizado a metodologia IP-Process. Na metodologia IP-Process, procedimentos de verificação estão presentes em cada fase do projeto. Essa metodologia divide o projeto de um IP-core em quatro fases: concepção comportamental, arquitetura, implementação RTL e prototipação. Na fase de projeto comportamental, requisitos funcionais e não-funcionais são listados em ordem para definir o escopo do projeto e os critérios de aceitação. Na fase de arquitetura os blocos e as ligações entre eles são estabelecidas, fornecendo a base para a implementação e verificação. No desenho RTL, a arquitetura é descrita em blocos sintetizáveis com a inclusão de código de verificação funcional e ultima fase o prototipação é testado eletricamente(LIMA et al., 2005b). Na Fig.67 pode ser visto o fluxo de projeto definido pela metodologia IP-Process. Esse metodologia foi adotada no projeto por especificar bem o fluxo necessário para o desenvolvimento correto de uma arquitetura, os passos que devem ser adotados e quando e quais partes devem ser testadas, assim garantindo que o resultado final seja o esperado e reduzindo a possibilidade de falhas.

A primeira parte da metodologia é a especificação da estrutura organizacional do projeto que foi apresentado na seção 4.5. Após, cada bloco foi descrito em SystemVerilog de maneira comportamental e validado utilizando a ferramenta QUARTUS II da Altera.

5.2 Detalhes da descrição da ULA

A Unidade Lógica Aritmética é composta por 12 blocos de operações e mais um decoder. Todos os sub-blocos foram descritos de maneira comportamental. Os detalhes de sua organização podem ser obtidos na seção 4.4.2 e Fig.64. Para sua implementação foi definido o padrão de seleção de operação encontrada na Tabela 3:

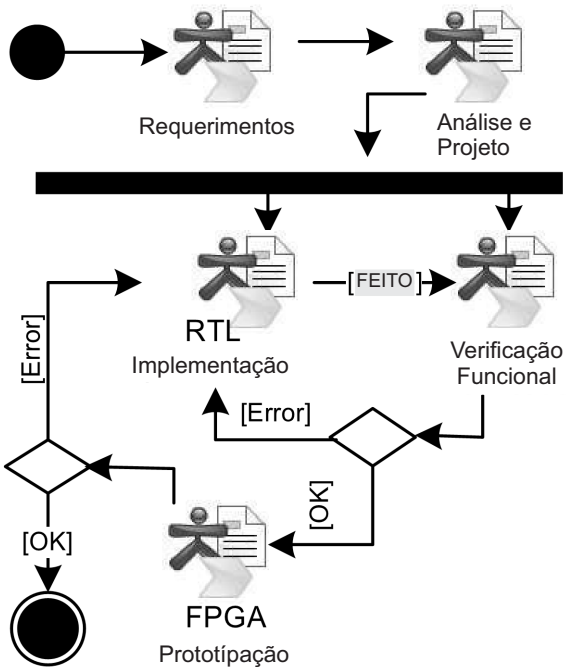


Fig. 67 – Fluxo de projeto para a metodologia IP-Process. Fonte: (LIMA et al., 2005a)

A descrição em SystemVerilog do bloco pode ser encontrada no apêndice A.

5.3 Detalhes da descrição do Banco de registradores

O banco de registradores do PAMPIUM é dividido em duas partes, chamadas *R0* e *R1* e sua especificação da sua organização é apresentada na seção 4.4.1 e Fig.63. E para sua implementação foram determinadas as características:

- Para ser possível a leitura e escrita de um registrador em um único ciclo, a leitura do valor é assíncrona e a escrita será realizada na borda de descida do sinal de clock;
- Na escrita o valor de entrada é atribuído à entrada de dados de todos os registradores. A seleção de qual será escrito é feita através do sinal de habilita escrita no endereço indicado;
- Os bits do registrador no endereço *02h* do banco de registradores *R0* indicam o valor dos pinos IN/OUT da *PORT_A*;
- Os bits do registrador no endereço *03h* do banco de registradores *R0* indicam o valor dos pinos IN/OUT da *PORT_B*;
- Os bits do registrador no endereço *02h* do banco de registradores *R1* indicam a direcionalidade dos pinos da *PORT_A*. Logo, se o bit 3 do registrador estiver em

Tabela 3 – Códigos de operação da *ULA* .

CÓDIGO(*OP_ULA*)	**OPERAÇÃO**	**DESCRIÇÃO**
0000b	**ADD**	Soma o valor da entrada *OP1* com o valor da entrada *OP2*
0001b	**SUB**	Subtrai o valor da entrada *OP1* do valor da entrada *OP2*
0010b	**DIV**	Divide o valor da entrada *OP2* pelo valor da entrada *OP1*
0011b	**REM**	Divide o valor da entrada *OP2* pelo valor da entrada *OP1*, e fornece como resultado o resto da operação de divisão
0100b	**BCLEAR**	Determina o valor de um bit da entrada *OP2* na posição indicada pelo valor da entrada *OP1* como 0 lógico
0101b	**BSET**	Determina o valor de um bit da entrada *OP2* na posição indicada pelo valor da entrada *OP1* como 1 lógico
0110b	**SHR**	Desloca à direita o valor da entrada *OP2*, o número de vezes indicado pela entrada *OP1*
0111b	**SHL**	Desloca à esquerda o valor da entrada *OP2*, o número de vezes indicado pela entrada *OP1*
1000b	**MULT**	Multiplica o valor da entrada *OP1* pelo da entrada *OP2*
1001b	**AND**	Realiza a operação "E" lógica entre os valores das entradas *OP1* e *OP2*
1010b	**OR**	Realiza a operação "OU" lógica entre os valores das entradas *OP1* e *OP2*
1011b	**XOR**	Realiza a operação "OU" exclusiva entre os valores das entradas *OP1* e *OP2*
1100b	**NOT**	Barra o valor da entrada *OP2*
1101b	**BIT**	Direciona o bit da entrada *OP2* da posição definida pela entrada *OP1* para a flag BIT
111xb		As entradas dos blocos de operação e a saída são colocados em estado de alta impedância

nível lógico 1, o pino 3 da *PORT_A* estará configurado como entrada(IN). Se em nível lógico 0 o pino estará configurado como saída(OUT);

- Os bits do registrador no endereço *03h* do banco de registradores *R1*, indicam a direcionalidade dos pinos da *PORT_B*.
- Os sinais de flags serão gravado no registrador de endereço *1Fh* no banco de registradores *R0*.

A descrição do bloco em SystemVerilog pode ser encontrada no apêndice B.

5.4 Detalhes da descrição da Unidade de CONTROLE

A unidade de **CONTROLE** deve determinar os sinais que formarão o caminho de dados, como explicado anteriormente. Além dos sinais, o controle deve determinar o endereço do registrador que receberá o valor dos literais de entrada (*#REG_L*), como também, o registrador que contém o valor que compõem o endereço de acesso à memória de programa (*#ADDR_MEM*).

Os sinais de controle e os endereços são apresentados a seguir. A instrução **NOP** é utilizada como referência, para a qual todos os sinais de controle são apresentados. Para as demais instruções, são destacados somente aqueles que forem diferentes.

5.4.1 Sinais para a instrução NOP

Como apresentado nas seção 4.3.2, na instrução **NOP** nenhuma operação é realizada, apenas será somado mais um no contador de programa. Para os demais blocos serão desabilitados os sinais de escrita. Os demais sinais de controle possuem valor irrelevante. Porém, foi escolhido deixá-los em 0 para minimizar o número de transições destes sinais. A *ULA* é configurada para ficar em alta impedância. O código da instrução foi definido como "***000000b***" e os sinais de controle são apresentados na Tabela 4.

Tabela 4 – Sinais de controle para instrução **NOP**.

MS_APC	*MS_PC*	*MS_EN_PC*	*SEL_OP_N*	*EN_RN*	*MS_N*
00b	1b	0001b	0b	0b	0b
EN_ST	*MS_OP1*	*OP_ULA*	*EN_DM*	*EN_R*	*EN_STATUS*
0b	00b	1111b	0b	0b	0b
MS_IN	*MS_AW*	*MS_BW*	*MS_A1*	*MS_B1*	*MS_A2*
00b	00b	0b	00b	00b	0b
MS_B2	*Addr_W_C*	*Addr_A1_C*	*Sel_B1_C*		
0b	00000b	00000b	0b		

5.4.2 Sinais para a instrução END

A instrução **END** requer que seja desabilitada a escrita no registrador de endereço de programa. O código da instrução foi definido como "***000001b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$MS\_EN\_PC = 0000\text{b};$$

5.4.3 Sinais para a instrução COPY

O caminho de dados para a instrução **COPY** pode ser visto na seção 4.3.4. O código da instrução foi definido como "***000010b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$EN\_R = 1\text{b};$$
$$MS\_IN = 10\text{b};$$

5.4.4 Sinais para a instrução MOVL

O caminho de dados para a instrução **MOVL** pode ser visto na seção 4.3.5. E como é o controle que define o endereço de escrita dos literais ($\#REG\_L$), foi escolhido como o endereço *#01h*. O código da instrução foi definido como "***000011b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$EN\_R\text{=}1\text{b};$$
$$MS\_AW = 10\text{b};$$
$$MS\_BW\text{=}1\text{b};$$
$$Addr\_W\_C\text{=}00001\text{b};$$

5.4.5 Sinais para a instrução JUMP

O caminho de dados para a instrução **JUMP** pode ser visto na seção 4.3.6. O código da instrução foi definido como "***000100b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$MS\_APC\text{=}10\text{b};$$

5.4.6 Sinais para a instrução CALL

O caminho de dados para a instrução **CALL** pode ser visto na seção 4.3.7. O código da instrução foi definido como "***000101b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$MS\_APC\text{=}10\text{b};$$
$$EN\_RN\text{=}1\text{b};$$
$$EN\_ST\text{=}1\text{b};$$

5.4.7 Sinais para a instrução ADDPC

O caminho de dados para a instrução **ADDPC** pode ser visto na seção 4.3.8. O código da instrução foi definido como "***011111b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$MS\_APC = 01\text{b};$$

5.4.8 Sinais para a instrução RET

O caminho de dados para a instrução **RET** pode ser visto na seção 4.3.9. O código da instrução foi definido como "***000110b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

MS_PC=0b;
SEL_OP_N=1b;
EN_RN=1b;
MS_N=1b;

5.4.9 Sinais para a instrução RETL

O caminho de dados para a instrução **RETL** pode ser visto na seção 4.3.10. O código da instrução foi definido como "***000111b***" e necessita a seguinte alteração nos sinais de controle em relação à referência.
Retorno de instrução:

MS_PC=0b;
SEL_OP_N=1b;
EN_RN=1b;
MS_N=1b;

Salva Literal:

EN_R=1b;
MS_AW = 10b;
MS_BW=1b;
$Addr\_W\_C$=00001b;

5.4.10 Sinais para a instrução RM

O caminho de dados para a instrução **RM** pode ser visto na seção 4.3.11. O registrador escolhido para indicar o endereço de acesso à memória de dados($\#ADDR\_MEM$), foi *#1Fh* do banco de registradores *R1*. O código da instrução foi definido como "***001000b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

EN_R=1b;
EN_STATUS=1b;
MS_IN=01b;
MS_AW=01b;
MS_BW=1b;
MS_A1=10b;
MS_B1=10b;
$Addr\_A1\_C$=11111b;
Sel_B1_C=1b;

5.4.11 Sinais para a instrução WM

O caminho de dados para a instrução **WM** pode ser visto na seção 4.3.12. O código da instrução foi definido como "***001001b***" e necessita a seguinte alteração nos sinais de controle em relação à referência:

$$EN\_STATUS=1b;$$
$$EN\_DM=1b;$$
$$MS\_AW=01b;$$
$$MS\_BW=1b;$$
$$MS\_A1=10b;$$
$$MS\_B1=10b;$$
$$Addr\_A1\_C=11111b;$$
$$Sel\_B1\_C=1b;$$

5.4.12 Sinais para a instrução BL

O caminho de dados para a instrução **BL** pode ser visto na seção 4.3.13. O código da instrução foi definido como "***001010b***" e necessita a seguinte alteração nos sinais de controle em relação à referência.
Análise da condição:

$$MS\_EN\_PC=0010b;$$
$$EN\_STATUS=1b;$$
$$MS\_OP1=10b;$$
$$OP\_ULA=0001b;$$
$$MS\_A2=1b;$$
$$MS\_B2=1b;$$

Caso condição verdadeira :

$$MS\_APC=01b;$$
$$MS\_A1=01b;$$
$$MS\_B1=01b;$$

5.4.13 Sinais para as demais instruções condicionais entre registradores

Os caminhos de dados para as instruções **BS**, **BLE**, **BSE**, **BE** e **BNE** são iguais ao visto na seção 4.3.13. A única alteração de sinal de controle referente a seção 5.4.12 é no MUX *MS_EN_PC* na fase de verificação da condição. Assim, os códigos de instruções e as mudanças de sinais para essas instruções são apresentados na Tabela 5.

Tabela 5 – Sinais para as instrução condicionais entre registradores.

Instrução	**Código de instrução**	**Sinal** *MS_EN_PC*
BS	***001011b***	0011b
BLE	***001100b***	0100b
BSE	***001101b***	0101b
BE	***001110b***	0110b
BNE	***001111b***	0111b

5.4.14 Sinais para as instruções BBCLEAR e BBSET

Os caminhos de dados para as instruções **BBCLEAR** e **BBSET** podem ser vistos na seção 4.3.15. Seus códigos de instrução são respectivamente "***010000b***" e "***010001b***" e necessitam a seguinte alteração nos sinais de controle em relação à referência.
Instrução **BBCLEAR**:

$$MS\_EN\_PC\text{=}1000b;$$
$$EN\_STATUS\text{=}1b;$$
$$MS\_OP1\text{=}01b;$$
$$OP\_ULA\text{=}1101b;$$

Caso condição verdadeira :

$$MS\_APC\text{=}01b;$$
$$MS\_A1\text{=}01b;$$
$$MS\_B1\text{=}01b;$$

Instrução **BBSET** só apresenta mudança no MUX *MS_EN_PC* em relação à instrução **BBCLEAR**, assim o sinal assume o seguinte valor:

$$MS\_EN\_PC\text{=}1001b;$$

5.4.15 Sinais para as instruções SHR e SHL

Os caminhos de dados para as instruções **SHR** e **SHL** podem ser vistos na seção 4.3.16. Seus códigos de instrução são respectivamente "***010010b***" e "***010011b***" e necessitam a seguinte alteração nos sinais de controle em relação à referência.
Instrução **SHR**:

$$EN\_R\text{=}1b;$$
$$EN\_STATUS\text{=}1b;$$
$$MS\_IN\text{=}11b;$$

MS_OP1=01b;
OP_ULA=0110b;

A instrução **SHL** apresenta apenas mudança no sinal OP_ULA em relação à instrução **SHR**, assim o valor o sinal fica:

OP_ULA=0111b;

5.4.16 Sinais para as instruções BSET e BCLEAR

As instruções **BSET** e **BCLEAR** foram definidas com os códigos de instrução "***010100b***" e "***01010b***" respectivamente, e apresentam as mesmas alterações nos sinais de referência que as das instruções **SHL** e **SHR**. Com a diferença no sinal OP_ULA, que irá ter os valores "0101b" e "0100b", respectivamente.

5.4.17 Sinais para as instruções aritméticas e lógicas

Os caminhos de dados para as instruções **MULT**, **DIV**, **REM**, **ADD**, **SUB**, **OR**, **AND** e **XOR** são vistos na seção 4.3.18.
A instrução **MULT** foi definida com o código de instrução "***010110b***" e as alterações nos sinais em relação à referência são:

EN_R=1b;
EN_STATUS=1b;
MS_A2=1b;
MS_B2=1b;
MS_IN=11b;
MS_OP1=10b;
OP_ULA=1000b;

As demais instruções apresentam apenas mudança no sinal OP_ULA referente a instrução **MULT**. Assim, os códigos de instruções e mudança de sinais para essas instruções são apresentados na Tabela 6.

5.5 Validação do Conjunto de Instruções

Todos os blocos projetados foram organizados como apresentado na seção 4.5. Com isso, todas as instruções implementadas foram verificadas via Software Quartus II. As instruções também foram testadas em Kit DE2 da Altera e para isso foram adicionadas

Tabela 6 – Sinais para as instrução aritméticas e lógicas.

Instrução	**Código de instrução**	**Sinal** *OP_ULA*
DIV	***010111b***	0010b
REM	***011000b***	0011b
ADD	***011001b***	0000b
SUB	***011010b***	0001b
OR	***011011b***	1010b
AND	***011100b***	1001b
XOR	***011101b***	1011b
NOT	***011110b***	1100b

estruturas de validação ao projeto, logo permitindo acompanhar os sinais e valores de registradores. Essa estruturas são retiradas após a validação final. A estrutura organizacional com os blocos para teste estão expressas na Fig.68.

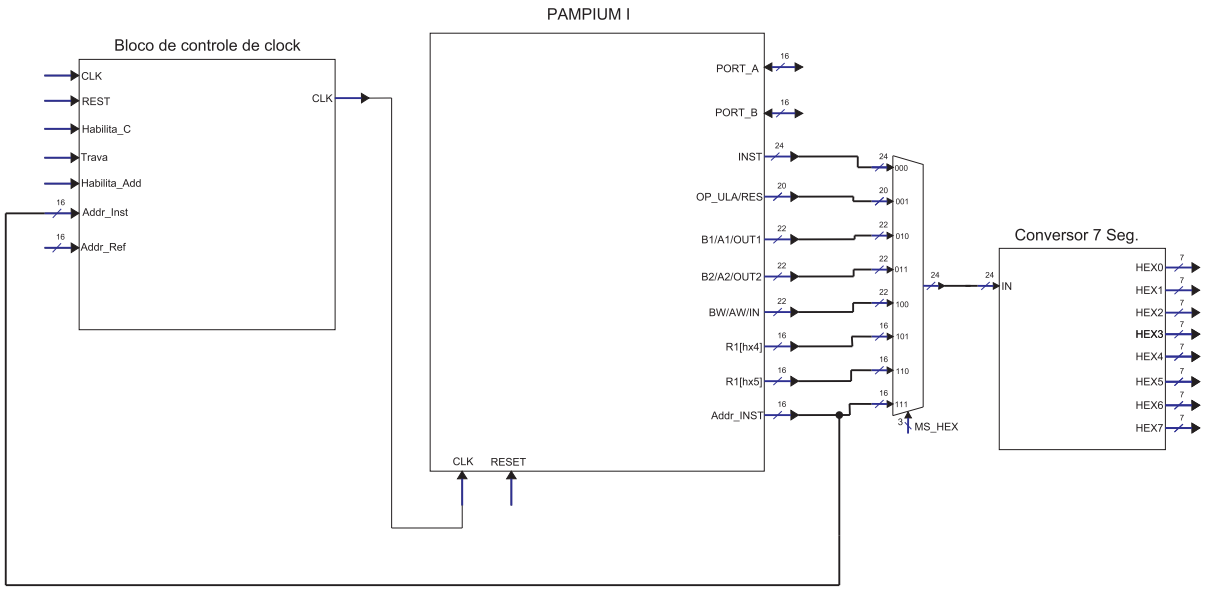


Fig. 68 – Arquitetura com estruturas de teste implementada em FPGA

Nesta seção são apresentados os testes de validação via software das instruções, como também o detalhamento dos blocos para testes acrescentados e os resultados obtidos para a validação das instruções em FPGA.

5.5.1 Validação via Software das Instruções

Com a descrição em SystemVerilg, cada instrução foi verificada via software Quartus II, utilizando o *Vector Waveform* do software Quartus II. Não serão apresentados os testes de validação das demais instruções, apenas o teste e o resultado para a instrução de multiplicação. O algoritmo executado na validação foi:

0:**NOP**;
1: **MOVL** 16h(1);
2: **MOVL** 3h(0);
3: **MULT** #01h(1),#01h(0),#00h(0);
4: **END**;

Na Fig.69 é apresentado o resultado da simulação da multiplicação de dois literais *16h* e *3h*. No primeiro ciclo de clock o literal *16h* é atribuído ao registrador *#REG_L* no banco de registradores *R1*. No segundo ciclo de clock o valor *3h* é atribuído ao registrador de literais *#REG_L* no banco de registradores *R0*. No próximo ciclo de clock é executado a multiplicação entre os dois registradores de literais e o resultado é armazenado no registrador *#00h* no banco de registradores *R0*. A Fig.69 apresenta mais 4 sinais: o "*IN_INST*" que indica o endereço da próxima instrução a ser executada; o sinal "*ADDR_INST*" indica a instrução que está sendo executada no presente ciclo; dois sinais "*RES_OUT*" e "*OP_ULA_OUT*" que apresentam respectivamente o resultado da operação executada pela *ULA* e qual operação está sendo executada.

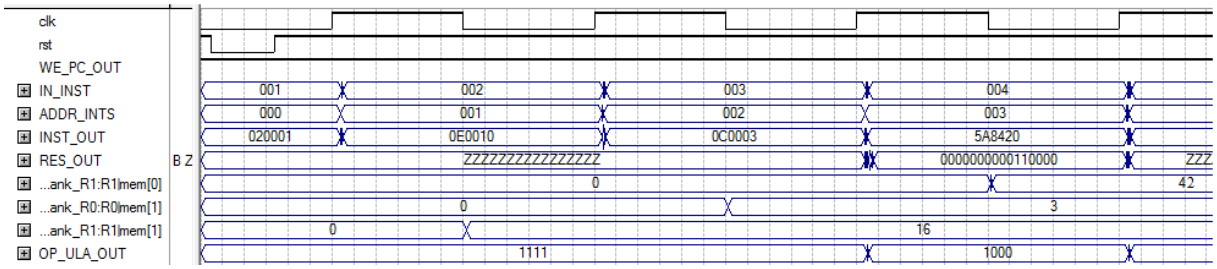


Fig. 69 – Resultado da simulação da instrução de multiplicação no software Quartus II

5.5.2 Estrutura de teste: Bloco de controle de clock

O controle de Clock permite através de um comando a passagem de um único ciclo de clock e também retém o clock quando um certo endereço de programa é atingido. A estrutura do bloco acrescentado é apresentada na Fig. 70.

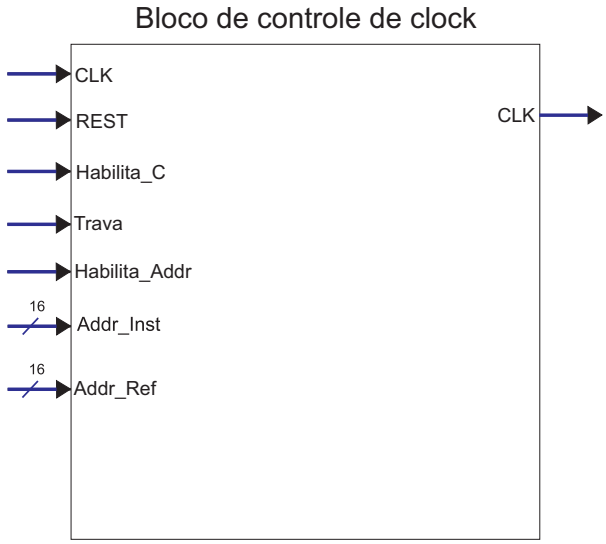


Fig. 70 – Estrutura acrescentada para testes: Bloco de controle de clock

Para permitir a passagem de um ciclo de clock, basta apenas levar o pino *Habilita_C* ao nível lógico 1. No entanto para permitir mais uma passagem de um ciclo, é necessário levar o pino "*Trava*" ao nível lógico 1 e depois a zero. Para levar o programa até uma determinada instrução basta colocar seu endereço no pino "*Addr_Ref*", e levar o pino "*Habilita_Addr*" a nível lógico 1.

5.5.3 Estrutura de teste: Bloco conversor 7 segmento

Este bloco converte um valor hexadecimal para ser apresentado na sequência de leds do display de 7 segmentos.

5.5.4 Estrutura de teste: Mux seletor do Display de 7 segmentos

Tem a função de selecionar qual sinal interno será mostrado no display de 7 segmentos.

5.5.5 Mapeamento das estruturas de teste para a FPGA

FPGA (Field-programmable gate array) é um grande conjunto de células lógicas idênticas. Cada célula lógica apresenta um certo número de funções que podem ser selecionadas. Assim, a célula exercerá a função a qual foi especificada. Essas células são interligadas através de *switches*(chaves) programáveis, que formam uma matriz. Resumindo, FPGA é um *array* de unidades lógicas em uma matriz de interconexões, que podem ser programadas pelo usuário formando circuitos integrados complexos(BROWN, 1996).

Para o teste em FPGA das instruções, os mesmos procedimentos da simulação foram adotados. No entanto, para esses testes foram utilizadas as estruturas adicionais de validação. Os sinais levados ao display de 7 segmentos foram:

- *INST*: A instrução que está sendo executada;
- *OP_ULA e RES*: Mostra a operação que está sendo executada pela ULA e o seu resultado;
- *OUT_MS_B1, OUT_MS_A1 e OUT1*: Mostra o banco de registradores que está sendo acessado, o endereço e o valor lido 1;
- *OUT_MS_B2, OUT_MS_A2 e OUT2*: Mostra o banco de registradores que está sendo acessado, o endereço e o valor lido 2;
- *OUT_MS_BW, OUT_MS_AW e OUT_MS_IN*: Mostra o banco de registradores que está sendo acessado, o endereço e o valor a ser escrito;
- *R1[04h]*: Mostra valor armazenado no registrador *#04h* no banco de registradores *R1*;
- *R1[05h]*: Mostra valor armazenado no registrador *#05h* no banco de registradores *R1*;
- *ADDR_INST*: Endereço da instrução que está sendo executada;

A estrutura de blocos utilizada para teste em FPGA está expressa na Fig.68.

O Kit de desenvolvimento utilizado para os teste foi o Kit DE2 da Altera com modelo de FPGA *EP2C35F672C6*. Os pinos *HEX* foram mapeados para os 8 displays de 7 segmentos contidos no Kit. O seletor de sinal para os displays (*MS_HEX*) foi mapeado para as chaves *SW0*, *SW1* e *SW2*. Os pinos de endereço de referência(*Addr_Ref*) foram mapeados para as chaves da *SW4* até à *SW15*, o restantes dos pinos foi conectado ao *ground*. O pino *Habilita_Addr* foi mapeado para a chave *SW17*. O pino de *RESET* foi mapeado para a chave *KEY0*. Os pinos *Habilita_C* e *Trava* foram mapeados para as chaves *KEY2* e *KEY3* respectivamente. O mapeamento para para o Kit DE2 está expresso na Fig.71. O clock utilizado para a validação das instruções foi de $10MHz$, pois devido à adição das estruturas de validação o clock máximo estimado foi de $16,7MHz$.

Os códigos testes foram descritos no formato binário atendendo aos padrões de instruções e pré-carregados na memória de programa.

5.5.6 Validação da instrução de multiplicação

O código utilizado para o teste foi o mesmo utilizando na simulação no programa Quartus II, com alteração no valor do primeiro literal. O código em Assembly foi traduzido para binário da seguinte forma:

- **NOP**: utiliza o formato de instrução *L* expresso na Fig.9 e traduzido para binário fica:

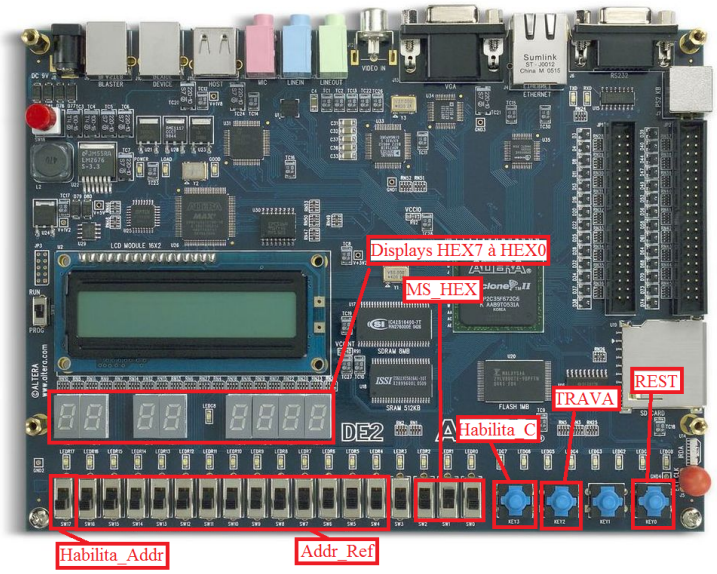


Fig. 71 – Mapeamento das estruturas de controle para Kit DE2 da Altera

000000_00_0000000000000000b;

- **MOVL** *10h*(1): utiliza o formato de instruções *L* expresso na Fig.9 e traduzido para binário fica:

000011_10_0000000000010000b;

- **MOVL** *3h*(0): utiliza o formato de instruções *L* expresso na Fig.9 e traduzido para binário fica:

000011_00_0000000000000011b;

- **MULT** *#1h*(1),*#1h*(0),*#0h*(0); utiliza o formato de instrução *R* expresso na Fig.10 e traduzido para binário fica:

010110_100_00001_00001_00000b;

- **END**: utiliza o formato de instrução *L* expresso na Fig.9 e traduzido para binário fica:

000001_00_0000000000000000b;

Após, a tradução do código o mesmo foi pré-carregado para memória de instrução do PAMPIUM I. O PAMPIUM I com as estruturas de teste foi sintetizado e carregado em FPGA. O resultado da validação para a instrução de multiplicação está detalhado abaixo:

Na Fig.72 pode ser vista a palavra de instrução da multiplicação no displays de 7 segmentos traduzida para hexadecimal "*5A0420h*".

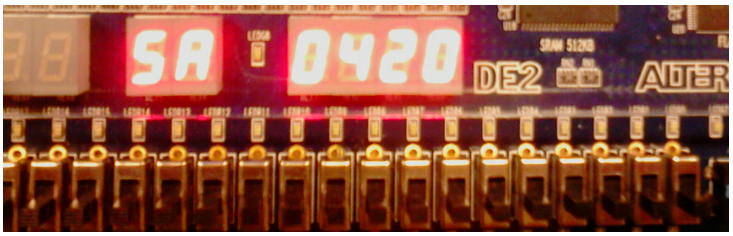


Fig. 72 – Palavra de instrução executada em teste de validação em FPGA

Na Fig.73 pode ser vista a operação executada pela *ULA*, correspondente à multiplicação "*8h*" e o resultado da operação em hexadecimal "*30h*".

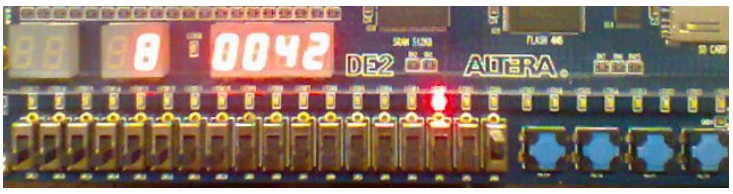


Fig. 73 – Operação e Resultado da *ULA* para a instrução executada em teste de validação em FPGA

Na Fig.74 pode ser visto o registrador lido 1, que está no banco de registradores *R1* no endereço hexadecimal "*#01h*" e seu valor de saída($OUT1$) é de "*0010h*".

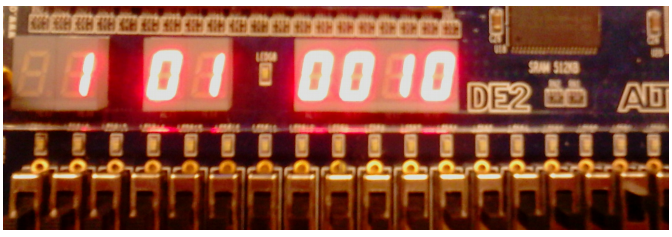


Fig. 74 – Registrador lido 1 da instrução executada em teste de validação em FPGA.

Na Fig.75 pode ser visto o registrador lido 2, que está no banco de registradores *R0* no endereço hexadecimal "*#01h*" e seu valor de saída($OUT1$) é de "*0003h*"; .

Na Fig.76 pode ser visto o registrador a ser escrito, que está no banco de registradores *R0* no endereço hexadecimal "*#00h*" e o valor de de entrada (IN) é de "*0030h*".

Por fim, na Fig.77 é apresentado o endereço de programa executado(*003h*).

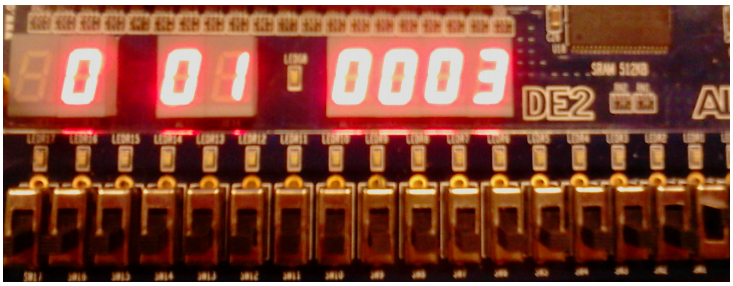


Fig. 75 – Registrador lido 2 da instrução executada em teste de validação em FPGA.

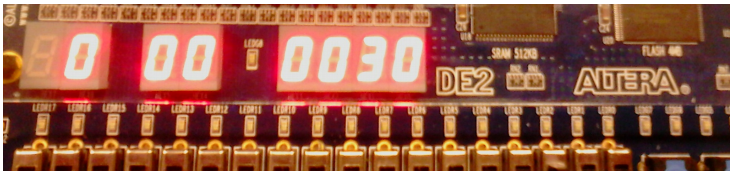


Fig. 76 – Registrador a ser escrito na instrução executada em teste de validação em FPGA.

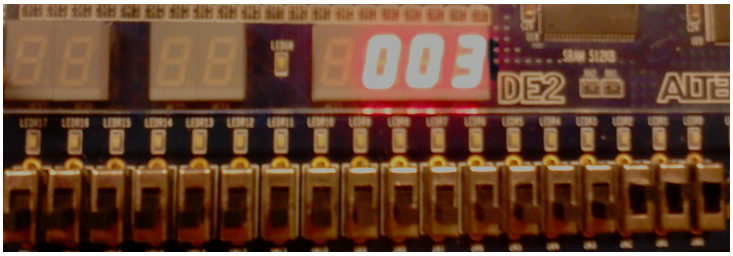


Fig. 77 – Endereço da instrução executada em teste de validação em FPGA.

6 PROTOTIPAÇÃO

Feitos todos os testes de validação de instrução via software e em FPGA, o PAMPIUM I foi sintetizado em FPGA para teste de funcionamento no controle de periféricos. Neste capítulo serão apresentados os dados da síntese sem as estruturas de testes, com as memórias mapeadas para a memória da FPGA. Também serão apresentados os dados técnicos gerais da arquitetura implementa e dois testes em aplicação prática, o primeiro de comunicação com ISP1362 e outro com a de aplicação em um porta de segurança eletrônica.

6.1 Dados da Síntese em FPGA

A implementação em Kit FPGA de desenvolvimento da Altera modelo EP2C35F672C6 resultou na utilização de 3336 elementos lógicos e 1536 registradores. As memórias de dados e de programa foram mapeados para a memória da FPGA que resultaram um total de $88kBytes$ da memória usada. A frequência máxima de clock foi de $25.4MHz$, sendo que o caminho crítico é dado pelas operações envolvendo a ULA.

6.2 Características Técnicas

O PAMPIUM I é um microcontrolador de 16 bits em arquitetura RISC. Com 32 instruções implementadas e 2 bancos de registradores com 32 registradores de 16 bits cada. Possui uma memória de dados de $64kBytes$, 2 portas IN/OUT e memória de programa de $8kWords$.

O banco de registradores possui registradores de uso específico e alguns são utilizados por algumas instruções como a instrução **MOVL** que armazena valores literais nos registradores *#01h* dos bancos de registradores *R0* e *R1* e as instruções **RM** e **WM** que utilizam o registrador *#1Fh* do banco de registradores *R1* para compor o endereço de acesso à memória de dados; O registrador de status no endereço *#1Fh* do banco de registradores *R0* é atualizado com os valores das flags da ULA toda vez que há uma operação envolvendo a mesma. Também há os registradores de configuração *#02h* e *#03h* do banco de registradores *R1*, que respectivamente configuram os pinos das portas *PORT_A* e *PORT_B* como saída(OUT) ou entrada (IN). Por fim os registradores conectados diretamente aos pinos das portas *PORT_A* e *PORT_B*, *#02h* e *#03h* no banco de registradores *R0*, que armazenam os valores dos pinos de suas respectivas portas.

As instruções implementadas com sua respectivas descrições são apresentadas na Tabela.1.

6.3 Montador de Programa

Para os testes de funcionamento é necessário desenvolver algoritmos que executem no PAMPIUM I. Elaborar esses algoritmos em binário ou os escrever em Assembly e traduzi-los para binário manualmente se torna muito desgastante. Pesando nisso foi desenvolvido um montador de programa que traduz o código Assembly para binário. O montador foi desenvolvido em linguagem MATLAB e se encontra no apêndice C.

O montador foi desenvolvido com uma função em MATLAB, onde a entrada dessa função é um arquivo do formato "TXT". Esse arquivo "TXT" deve ser escrito com linguagem Assembly sugerida na seção 4.2. É possivel também utilizar o comando "Function" e logo em seguida "nome da linha" para definir um *"label"* de linha. Esses *"label"* é utilizado nas instruções **CALL** e **JUMP** para facilitar a indicação para onde esse instruções devem pular.

6.4 Aplicação do teste 1- Comunicação com ISP1362

O ISP1362 é um single-chip controlador integrado USB (Universal Serial Bus) On-The-Go(OTG) com Philips Slave Host Controller e Philips ISP1181B Device Controller. O controlador USB OTG é compatível com a On-The-Go do USB 2.0. Os controladores de Host e Device são compatíveis com a Universal Serial Bus Specification Rev.2.0, com suporte à transferência de dados em velocidade maxima(12 Mbit/s) e baixa(1,5 Mbit/s). O ISP1362 tem duas portas USB: porta 1 e porta 2. A porta 1 pode ser configurada para funcionar como uma porta Host, Device ou OTG. A porta 2 só pode ser usado como Host(PHILIPS, 2008).

O ISP1362 foi escolhido para testes de controle de periférico, pois se encontra embarcado no Kit DE2 da Altera utilizados para síntese do PAMPIUM I. O teste de comunicação como ISP1362 tem a finalidade de verificar o funcionamento de PAMPIUM I no controle de um periférico e de suas portas IN/OUT.

O ISP1362 possui um barramento de dados IN/OUT de 16 bits utilizado para enviar e receber dados de seu registradores. Para ler ou escrever de um registrador é necessário condicionar alguns sinais de controle. Assim, para o controle desse periférico, como indicado em (PHILIPS, 2008) é necessário primeiramente executar um teste de comunicação extensiva com ISP1362. Este teste consiste em escrever um valor em um registrador do ISP1362 e lê-lo novamente, comparar o valor lido como o valor escrito. Caso o valor seja igual é incrementado um acerto. Caso haja 100% de acerto, o microcontrolador está pronto para controlar o ISP1362.

6.4.1 Leitura e escrita de um registrador do ISP1362

A leitura ou escrita de um registrador do ISP1362 segue duas fases: a fase de comando, onde é indicado o endereço a ser escrito ou lido e a fase de dado onde o dado é lido ou escrito.

Os sinais de controle para cada uma das fases pode ser visto na Tabela 7.

Tabela 7 – Fases para comunicação com ISP1362.

Fase	**Sinais de controle**				
	CS	**RD**	**WR**	**A1**	**A0**
Inicial	1	1	1	0	1
Comando	0	1	0	0	1
Escrita	0	1	0	0	1
Leitura	0	0	1	0	1

6.4.2 Algoritmo de comunicação

O algoritmo de teste de comunicação poder ser encontrado em apêndice D, e consiste nos seguintes passos:

1. Inicializar duas variáveis, ”CONT” e ”HITS”, uma que conterá o valor a ser escrito e outra para contar o número de correspondências;
2. Executar a fase de comando, para definir o endereço a ser escrito;
3. Escrever o valor do registrador ”CONT” no ISP1362;
4. Executar a fase de comando novamente, indicando o endereço de leitura;
5. Ler o valor do registrador;
6. Comparar o valor lido do registrador com o valor do contador;
7. Verificar se os valores forem iguais a variável ”HITS”é incrementada;
8. Incrementar a variável ”CONT”;
9. Direcionar ao passo 2 o programa, executando 65535 leituras e escritas;
10. Paralisar o programa depois que todas as leituras e escritas forem realizadas.

Com a execução do programa, foi constatado 100% de correspondência entre o valor escrito e o valor lido. Isso mostra que a interface da saída e entrada das porta

IN/OUT do PAMPIUM I funcionam de maneira correta. Também com esse teste é possivel constatar que o PAMPIUM I está preparado para atuar como microcontrolador em sistemas embarcados de processamento e comunicação de dados.

6.5 Aplicação do teste 2- Porta Eletrônica de Segurança

Foi elaborado uma aplicação com o intuito de demostrar que o PAMPIUM I exerce seu papel de microcontrolador. A aplicação escolhida foi a de uma porta de segurança eletrônica, na qual uma porta é aberta caso o usuário entre como seu nome e sua senha corretamente. As funcionalidades implementadas no algoritmo foram:

- A entrada de usuário e senha abre a porta.
- Se usuário for administrador, é possível adicionar usuários novos, editar senha de usuários, excluir senha de usuários, verificar quantos usuários estão salvos e abrir a porta;
- Cada erro de senha acarretará um tempo de espera de 15 segundos para uma nova entrada;
- No quarto erro o usuário deverá espera 1 minuto para uma nova tentativa;

O fluxo da execução do programa pode ser visto na Fig.78.

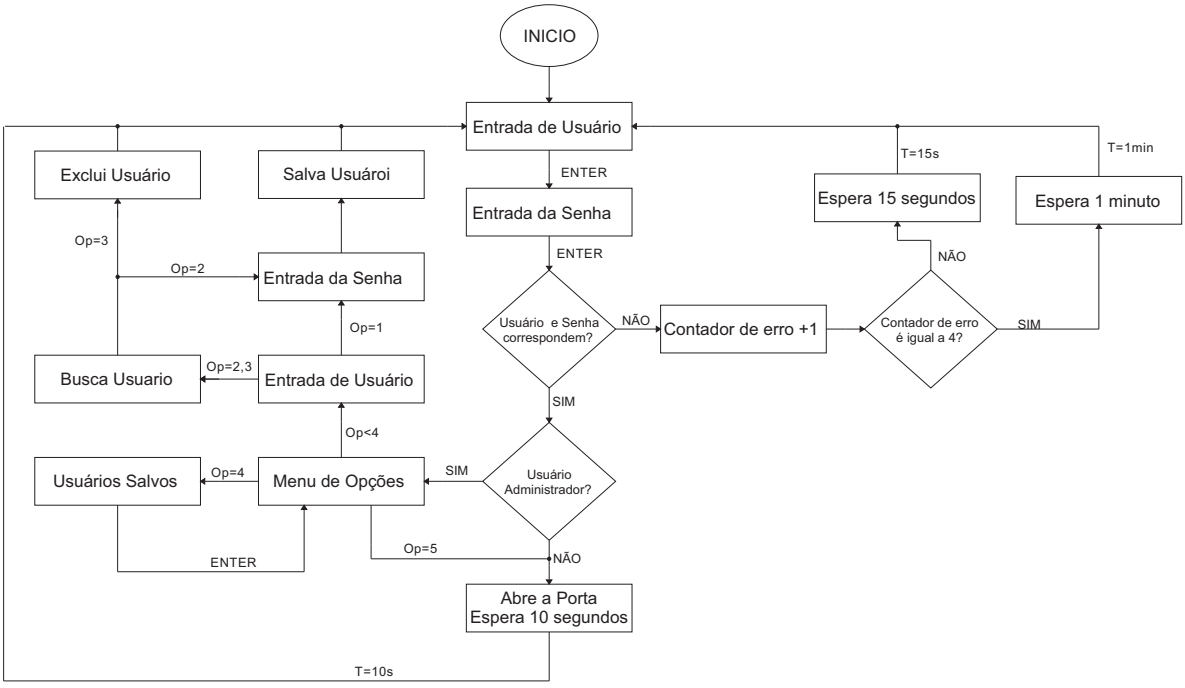


Fig. 78 – Fluxo da execução do programa da porta de segurança.

6.5.1 Aspectos do projeto

Para o desenvolvimento do projeto é necessário definir quais pinos são utilizados, a frequência de operação, tipo de comunicação com os periféricos e as funções a serem implementadas. Como é utilizados um display LCD e um porta de comunicação PS/2 para teclado, foram desenvolvidas as rotinas específicas para cada um dos periféricos. As definições de projeto para o PAMPIUM I e os demais periféricos foram:

- Para conexão como os pinos de controle, e dados do display LCD foram utilizados os pinos PORT_A;
- Para comunicação PS/2 foram utilizados os pinos "0"e "1"da porta PORT_B;
- Devido à taxa de transmissão de informações da comunicação PS/2 e à quantidade de registradores a serem utilizados na rotina de delay, a frequência de operação foi definida como $1MHz$;
- Os pinos de comando de abertura da porta foram ligados aos leds verdes, o alerta de erro de senha foi conectado ao leds vermelho.
- Foi implementa uma rotina de delay de mili-segundos, programável;
- Foram implementadas rotinas para o LCD: Inicializar, Escrever caractere; Apagar um caractere; Limpar LCD; Determinar linha e coluna à escrever;
- Foram implementadas rotinas de PS/2 para teclado: Inicializar; Lêr caractere simples(1 byte), Lêr caractere composto(2 bytes);
- Foram implementadas rotinas de conversão de caractere para formato ASCII.
- A demais rotinas que foram implementadas no algoritmo compreendem os estados apresentados no fluxo de programa na Fig.78.

6.5.2 Resultados da aplicação

Para o desenvolvimento do programa foram necessárias 2875 linhas de código Assembly, sendo que 46% dessa linhas foram dedicadas a escrever mensagens no LCD. O programa principal utilizou 206 linhas de código e as demais linhas são divididas entre as funções descritas acima. O código principal e todas as funções implementadas se encontram no apêndice E.

Na Fig.79 pode ser vista a aplicação implementada em FPGA. O usuário e senha são compostos pelo mínimo de 4 caracteres e o máximo de 8, podendo ser utilizados todos os caracteres do alfabeto, mais os numéricos para compor o usuário ou senha. O usuário administrador por padrão é "GAMA" e a senha é "GAMAUNIP". Como é possível

excluir qualquer usuário, se o administrador for excluído o primeiro usuário adicionado será definido como administrador. Se o primeiro for excluído o segundo será definido como administrador e assim por diante. Em todos os menus, se apertada a tecla "ESC", o programa será redirecionado ao menu inicial, se em 1 minuto não for apertada nenhuma tecla, o programa também será redirecionado ao menu inicial.

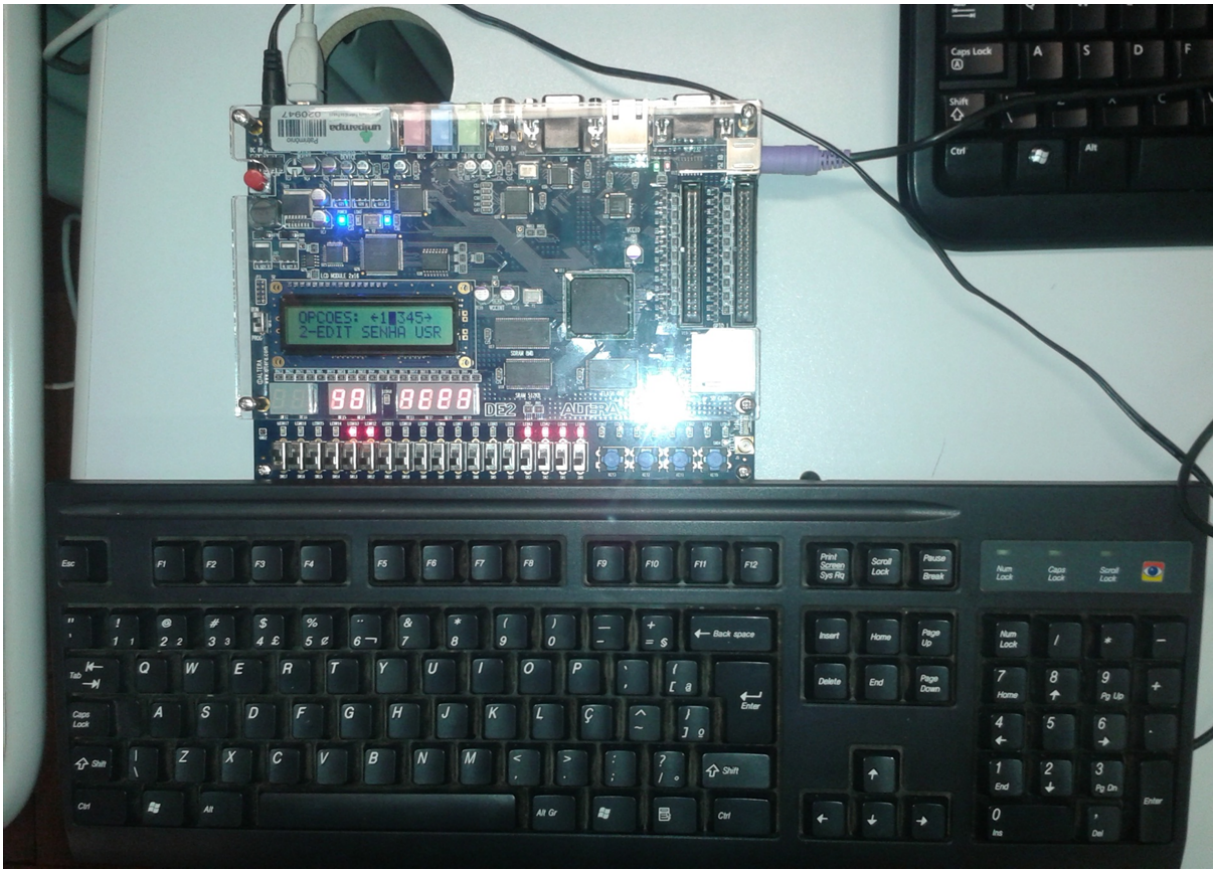


Fig. 79 – Foto da aplicação da porta de segurança implementada em FPGA.

Na Fig.79 pode-se acompanhar o funcionamento da aplicação. Como a aplicação funcionou de maneira correta é possivel afirma o correto funcionamento do PAMPIUM I na execução do programa e também no controle dos periféricos utilizados. Com isso foi cumprido o objetivo inicial da elaboração de um microcontrolador de 16 bits em arquitetura RISC de propósito geral para controlar os mais variados periféricos.

CONCLUSÃO

Este trabalho apresentou o desenvolvimento da arquitetura de um novo microcontrolador de 16 bits com arquitetura RISC chamado PAMPIUM I. Na fase atual do projeto obteve-se resultados positivos quanto à criação de uma arquitetura funcional, a qual foi validada através de testes via software e síntese em FPGA.

A arquitetura foi implementada em SystemVerilog utilizando uma descrição comportamental e sintetizada para FPGA. Os resultados da simulação no Quartus II e a implementação em FPGA demonstraram o correto funcionamento de todas as 32 instruções implementadas. O teste de aplicação, no qual o PAMPIUM I fui utilizado para realizar uma rotina de comunicação com o chip ISP1362, apresentou resultado positivo de 100% de correspondência na comunicação, afirmando o correto funcionamento das interfaces de comunicação externa do microcontrolador. O PAMPIUM I também foi utilizando no controle de uma porta eletrônica de segurança. Essa aplicação demonstrou que o microcontrolador concebido está pronto para ser utilizado nos mais diversos equipamentos. Partes dos resultados já foram publicados no 29º Simpósio Sul de Microeletrônica (ENGROFF ALIAN M., 2014).

Com isso podemos afirmar que os objetivos iniciais foram alcançados, na elaboração de uma arquitetura de propósito geral. As motivações do projeto foram satisfeitas pois o desenvolvimento do PAMPIUM I abre portas para trabalhos futuros que são apresentados a seguir.

Trabalhos Futuros

O PAMPIUM I é a primeira versão do que pode levar a vários outros trabalhos de pesquisa e desenvolvimento. Podem ser acrescentadas novas funcionalidades, também adaptá-lo para aplicações específicas. As sugestões e planos de trabalhos futuros estão seguir:

- Desenvolvimento de compilador para o PAMPIUM I, que permita utilizar linguagens de alto nível, otimizando o algoritmo. Isso permite utilizar vários testes de desempenho no PAMPIUM I possibilitando a comparação com arquiteturas do estado da arte.

- Desenvolver simulador para verificação dos correto funcionamento dos programas criados para o PAMPIUM I.

- Acrescentar estruturas de interrupção. Acrescentar interrupções geradas por eventos externos ou internos do PAMPIUM, fazendo com que o programa seja desviado para uma linha específica de programa;

- Acrescentar estruturas de contadores e geradores de PWM(Pulse-Width Modulation).

- Acrescentar estruturas de Pipeline trazendo ao o PAMPIUM melhor eficiência e rapidez;
- Adaptação da estrutura organizacional do microcontrolador para atividades de ensino de disciplinas afins;
- Adaptar a estrutura organizacional do PAMPIUM para uma atividade específica;
- Adicionar o PAMPIUM a outros projetos a fim de controlar periféricos na aquisição de dados utilizados nesses projetos(ex: Aquisição de imagem para testes de arquiteturas de filtros digitais);
- Adaptar o PAMPIUM para o tornar um microcontrolador com instruções reconfiguráveis, possibilitando aos usuários criar suas próprias instruções com os blocos internos disponíveis.
- Aperfeiçoar a arquitetura existente, melhorando seu caminho de dados para a torná-la mais rápida.
- Desenvolver interface de gravação na memória de programa, possibilitando gravar o programa em FPGA sem a necessidade de recompilar e carregar a arquitetura no mesmo.
- Fazer a síntese lógica e física do PAMPIUM I, gerando um protótipo de chip do microcontrolador PAMPIUM I.

REFERÊNCIAS

BROWN, S. FPGA architectural research: a survey. **Design Test of Computers, IEEE**, v. 13, n. 4, p. 9–15, Winter 1996. ISSN 0740-7475.

CARRO, L.; WAGNER, F. R. Sistemas computacionais embarcados. **Jornadas de atualização em informática. Campinas: UNICAMP**, 2003.

CARTER, N. P. Arquitetura de Computadores. **ATMED EDITORA S.A**, 2002.

ENGROFF ALIAN M., D. S. A. O. G. A. PAMPIUM I - a new 16-bit RISC microcontroller suitable for low power applications. **SIM- 29° Simpósio Sul de Microeletrônica**, 2014.

GAJSKI, D. D. et al. Specification and design of embedded systems. {Prentice Hall PTR}, 1994.

GERVINI, A. I. et al. Avaliação de desempenho, área e potência de mecanismos de comunicação em sistemas embarcados. **SEMISH'03–XXX Seminário Integrado de Software e Hardware**, 2003.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. [S.l.]: Elsevier, 2012.

HWANG, K.; RAMACHANDRAN, A.; PURUSHOTHAMAN, R. **Advanced computer architecture: parallelism, scalability, programmability**. [S.l.]: McGraw-Hill New York, 1993.

JUNQUEIRA, A. A. Risco: microprocessador RISC CMOS de 32 bits. **CPGCC da UFRGS**, Porto Alegre, RS, 1993.

LIMA, M. et al. IP-PROCESS: using a process to teach IP-core development. p. 27–28, June 2005.

LIMA, M. et al. **IP-PROCESS: A Development Process for Soft IP-core with Prototyping in FPGA**. [S.l.: s.n.], 2005.

NEUMANN, J. V. John von neumann. **Collected Works**, v. 6, p. 219–237, 1992.

ORTEGA, J.; ANGUITA, M.; PRIETO, A. Arquitectura de computadores. **Thomson editores Spain Parainfo SA**, 2005.

PHILIPS, S. **ISP1362: Single-chip Universal Serial Bus On-The-Go controller**. [S.l.: s.n.], 2008.

RANDELL, B. **Design fault tolerance**. [S.l.]: Springer, 1987.

STALLINGS, W. **Computer organization and architecture: designing for performance**. [S.l.]: Pearson Education India, 2001.

SUTHERLAND SIMON DAVIDMANN, P. F. S. **SystemVerilog for Design:A Guide to Using SystemVerilog for Hardware Design and Modeling**. Second. New York, NY 10013, USA: Springer Science+Business Media,LLC, 2006.

Apêndices

APÊNDICE A – DESCRIÇÃO EM SYSTEMVERILGO DA ULA

```
module ULA(input wire [3:0] sel_op, input wire [15:0] OP1, OP2, output wire[15:0]RES, output wire NEG,ZERO,OVF,BIT);
wire [15:0] A, B,C,D,E,F,G,H,I,J,A1,B1,E1,F1,G1,H1, RES_A, RE_B, RES_C, RES_D, RES_E, RES_F, RES_G, RES_H, RES_I, RES_END;
wire [3:0] C1,D1,J1;
logic op_a,op_b,op_c,op_d,ovf_add,ovf_mult;


DMUX DMUX(sel_op,OP1,A,B,C,D,E,F,G,H,I,J);//Demux da entrada OP2
DMUX1 DMUX1(sel_op,OP2,A1,B1,E1,F1,G1,H1,C1,D1,J1);//Demux da entrada OP2
DECODER DC(sel_op,op_a,op_b,op_c,op_d);// Decoder da ULA
ADD ADD(A,A1,op_a,ovf_add,RES_A);// somador subtrator
DIV DIV(op_b,B,B1,RES_B);// divisor e resto
BIT_SC BIT_SC(op_c,C,C1,RES_C);// Bir set e bit clear
SHIFTER SHIFT(op_d,D,D1,RES_D);// deslocador binario
MULT MULT(E,E1,ovf_mult,RES_E);// multiplicador
P_AND P_AND(F,F1,RES_F);// operacao and
P_OR P_OR(G,G1,RES_G);// operacao or
P_XOR P_XOR(H,H1,RES_H);// operacao xor
P_NOT P_NOT(I,RES_I);// operacao not
BIT_D BIT_D(J,J1,BIT);// gerador da flag bit
MUX_ULA MUX(sel_op ,RES_A ,RES_B, RES_C, RES_D, RES_E, RES_F, RES_G, RES_H, RES_I, RES_END);// mux de saida
DEC_ZERO DEC(RES_END,ZERO);// detector de zero
always_comb begin
NEG=RES_END[15];// flag neg
OVF=ovf_mult || ovf_add;// flag overflow
RES=RES_END; // resposta de saida;
end


endmodule
```

APÊNDICE B – DESCRIÇÃO EM SYSTEMVERILOG DO BANCO DE REGISTRADORES

```
module BANK_REG (input wire clk, rst, we, we_status, sel_bank_write, sel_bank_1,
sel_bank_2,
input wire[4:0] address_1, address_2, address_write,
input wire[15:0] IN, IN_STATUS,
output wire[15:0] OUT_1, OUT_2,
inout wire[15:0] PORT_A,PORT_B);

wire [15:0] OUT_R0_1,OUT_R0_2,OUT_R1_1,OUT_R1_2,CONF_A,CONF_B;

Bank_R0 R0(clk, rst,(we && !sel_bank_write), we_status, address_write, address_1, ad-
dress_2,IN, IN_STATUS, OUT_R0_1, OUT_R0_2, CONF_A, CONF_B, PORT_A, PORT_B);
Bank_R1 R1(clk, rst,(we && sel_bank_write), address_write, address_1, address_2, IN,
OUT_R1_1, OUT_R1_2 , CONF_A, CONF_B);

assign OUT_1=(sel_bank_1)? OUT_R1_1:OUT_R0_1;
assign OUT_2=(sel_bank_2)? OUT_R1_2:OUT_R0_2;

endmodule
```

APÊNDICE C – PROGRAMA EM MATLAB DO MONTADOR DE PROGRAMA

```
function Comp_Assembly(kkkk)
fid = fopen('resultado/exp.txt', 'wt');
Arq=importdata(kkkk,'%s');

[b c]=size(Arq);
coment=0;
jvn=0;
for i=1:1:b
jvn=0;
x=Arq(i,:);
x=char(x);
x=strtrim(x);
x=deblank(x);
fprintf('Analizando linha %d, instrução=%s /n', i, x);
n=find(isspace(x));

[n1 n2]=size(x);
cont=1;
pw=' ';
for j=1:1:n2
if ((x(j) =' ') && (x(j) =';'))
pw(cont)=x(j);
cont=cont+1;
else
break;
end
end
switch pw
case '//'
coment=coment+1;
jvn=1;
case 'FUNCTION'
INST=dec2bin(0,6);
A=dec2bin(0,1);
B=dec2bin(0,1);
num=dec2bin(0,16);
R1=";
R2=";
R3='// NÂO FAZ NADA';
case 'NOP'
INST=dec2bin(0,6);
A=dec2bin(0,1);
B=dec2bin(0,1);
num=dec2bin(0,16);
R1=";
R2=";
R3='// NÂO FAZ NADA';
case 'FIM'
INST=dec2bin(1,6);
A=dec2bin(0,1);
B=dec2bin(0,1);
num=dec2bin(0,16);
R1=' ';
R2=' ';
R3='// FIM DE PROGRAMA';
case 'COPY'
INST=dec2bin(2,6);

[R1 j]=Reg(j);

[A j]=Ban(j);

[R2 j]=Reg(j);

[B j]=Ban(j);
num=dec2bin(0,16);
num(1)=B;
B=dec2bin(0,1);
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(0,5);
num(12:16)=dec2bin(hex2dec(R2),5);
R1=sprintf('R%s[hx%s]', A, R1);
R3=sprintf('// Copia, R%s[hx%s]=', num(1), R2);
R2=";
case 'MOVL'
INST=dec2bin(3,6);

[R1 j]=Reg(j);

[A j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
num=dec2bin(hex2dec(R1),16);
R1=sprintf('hx%s;',R1);
R3=sprintf('// MOVL, R%s[REGL]=', A);
R2=' ';
case 'JUMP'
INST=dec2bin(4,6);

[R1 j]=Reg(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
A=dec2bin(0,1);
num=dec2bin(hex2dec(R1),16);
R1=sprintf('hx%s;', R1);
R3=sprintf('// JUMP, PC=PC+1');
R2=' ';
case 'CALL'
INST=dec2bin(5,6);

[R1 j]=RegF(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
A=dec2bin(0,1);
num=dec2bin(hex2dec(R1),16);
R1=sprintf('hx%s; ',R1);
R3=sprintf('// CALL, PC=PC+1');
R2='SP[N]=PC+1; N=N+1;';
case 'RET'
INST=dec2bin(6,6);
A=dec2bin(0,1);
B=dec2bin(0,1);
num=dec2bin(0,16);
R1=";
R2=";
R3='// Retorno de instrução, PC=SP[N-1]; N=N-1;';
case 'RETL'
INST=dec2bin(7,6);

[R1 j]=Reg(j);

[A j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
num=dec2bin(hex2dec(R1),16);
R2=sprintf('hx%s;',R1);
R3='// Retorno de instrução com literal, PC=SP[N-1]; N=N-1;';
R1=sprintf(' R%s[REGL]=', A);
case 'RM'
INST=dec2bin(8,6);

[R1 j]=Reg(j);

[A j]=Ban(j);

[R2 j]=Reg(j);
B=dec2bin(0,1);
num=dec2bin(0,16);
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:16)=dec2bin(hex2dec(R2),10);
R3=sprintf('// RM, R%s[hx%s]=', A, R1);
R1=sprintf('DM[addr_mem + hx%s];', R2);
R2=";
case 'WM'
INST=dec2bin(9,6);

[R1 j]=Reg(j);

[A j]=Ban(j);

[R2 j]=Reg(j);
B=dec2bin(0,1);
num=dec2bin(0,16);
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:16)=dec2bin(hex2dec(R2),10);
R1=sprintf('R%s[hx%s];', A, R1);
R3=sprintf('// WM ,DM[addr_mem + hx%s]=',R2);
```

```
R2='';
case 'BL'
INST=dec2bin(10,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s]>', A,
R1);
R1=sprintf('R%s[hx%s]',B,R2);
R2=sprintf('{PC= PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BS'
INST=dec2bin(11,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s]<', A,
R1);
R1=sprintf('R%s[hx%s]', B, R2);
R2=sprintf('{PC=PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BLE'
INST=dec2bin(12,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s]>=',
A, R1);
R1=sprintf('R%s[hx%s]', B, R2);
R2=sprintf('{PC=PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BSE'
INST=dec2bin(13,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s]<=',
A, R1);
R1=sprintf('R%s[hx%s]', B, R2);
R2=sprintf('{PC=PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BE'
INST=dec2bin(14,6); [R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s]==',
A, R1);
R1=sprintf('R%s[hx%s]', B, R2);
R2=sprintf('{PC=PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BNE'
INST=dec2bin(15,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s] =', A,
R1);
R1=sprintf('R%s[hx%s]', B, R2);
R2=sprintf('{PC=PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BBCLEAR'
INST=dec2bin(16,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(str2num(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s](%s)==0',
A, R1, R2);
R2=' ';
R1=sprintf('{PC=PC +
R%s[hx%s]} {PC=PC+1}', C, R4);
case 'BBSET'
INST=dec2bin(17,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg1(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(str2num(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R4=R3;
R3=sprintf('// if R%s[hx%s]
(%s)==1', A, R1 ,R2);
R2=' ';
```

```
R1=sprintf('{PC=PC + R%s[hx%s]} {PC= PC+1}', C, R4);
case 'SHR'
INST=dec2bin(18,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg1(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
R4=R3;
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(str2num(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C, R4);
R2=sprintf('%s;',R2);
R1=sprintf('R%s[hx%s]<<', A, R1);
case 'SHL'
INST=dec2bin(19,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg1(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
R4=R3;
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(str2num(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C, R3);
R2=sprintf('%s;',R2);
R1=sprintf('R%s[hx%s]>>', A, R1);
case 'BSET'
INST=dec2bin(20,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg1(j); num=dec2bin(0,16);
B=dec2bin(0,1);
R4=R3;
num(1)=A;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(str2num(R2),5);
num(12:16)=dec2bin(hex2dec(R1),5);
R3=sprintf('// R%s[hx%s](%s)= 1', A, R1, R2);
R1=' ';
R2=' ';
case 'BCLEAR'
INST=dec2bin(21,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
R4=R3;
num(1)=A;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(str2num(R2),5);
num(12:16)=dec2bin(hex2dec(R1),5);
R3=sprintf('// R%s[hx%s](%s)= 0', A, R1, R2);
R1=' ';
R2=' ';
case 'MULT'
INST=dec2bin(22,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C, R3);
R1=sprintf('R%s[hx%s]*', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'DIV'
INST=dec2bin(23,6); [R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=',C,R3);
R1=sprintf('R%s[hx%s]/', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'REM'
INST=dec2bin(24,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C; num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C, R3);
R1=sprintf('R%s[hx%s]%%', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'ADD'
INST=dec2bin(25,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C, R3);
R1=sprintf('R%s[hx%s]+', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'SUB'
INST=dec2bin(26,6); [R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C,
```

```
R3);
R1=sprintf('R%s[hx%s]-', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'OR'
INST=dec2bin(27,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C,
R3);
R1=sprintf('R%s[hx%s]||', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'AND'
INST=dec2bin(28,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C,
R3);
R1=sprintf('R%s[hx%s]&&', A,
R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'XOR'
INST=dec2bin(29,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R2 j]=Reg(j);
[B j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(hex2dec(R2),5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C,
R3);
R1=sprintf('R%s[hx%s]^', A, R1);
R2=sprintf('R%s[hx%s];', B, R2);
case 'NOT'
INST=dec2bin(30,6);
[R1 j]=Reg(j);
[A j]=Ban(j);
[R3 j]=Reg(j);
[C j]=Ban(j);
num=dec2bin(0,16);
B=dec2bin(0,1);
num(1)=C;
num(2:6)=dec2bin(hex2dec(R1),5);
num(7:11)=dec2bin(0,5);
num(12:16)=dec2bin(hex2dec(R3),5);
R3=sprintf('// R%s[hx%s]=', C,
R3);
R1=sprintf('!R%s[hx%s]', A, R1);
R2=' ';
otherwise
fprintf(fid,'Erro linha %d=> Ex-
pressão desconhecida "%s"',(i-1),
pw);
break;
end
if(jvn==0)
fprintf(fid,' %d: data <= 24"b%s
_%s%s%s _%s _%s _%s _%s; %s%s%s
/n', (i-coment-1), INST(1:4),
INST(5:6), A, B, num(1:4),
num(5:8), num(9:12), num(13:16),
R3, R1, R2);
end
end
clear ;
open('resultado/exp.txt');
function [R,k]=Reg(j)
if (j =n2)
cont=1;
for k=j+1:1:n2
if ((x(k) ='(') && (x(k) =';'))
if(x(k) =',')&&(x(k) =' ')
Rn(cont)=x(k);
cont=cont+1;
end
else
break;
end
end
else
Rn=0;
end
R=Rn;
end
function [R,k]=Reg1(j)
Nz=0;
if (j =n2)
cont=1;
for k=j+1:1:n2
if ((x(k) ='(') && (x(k) =';'))
if(x(k) =',')
if(x(k) =' ')
Rn(cont)=x(k);
cont=cont+1;
end
else
if(Nz==1)
break;
end
Nz=1;
end
else
break;
end
end
else
Rn=0;
end
R=Rn;
end
function [B1,k]=Ban(j)
if(j =n2)
cont=1;
for k=j+1:1:n2
if ((x(k) =')'))
if (x(k) =' ')
A1(cont)=x(k);
cont=cont+1;
end
else
break;
end
end
else
A1=0;
end
B1=A1;
end
function [R,k]=RegF(j)
if (j =n2)
cont=1;
for k=j+1:1:n2
if ((x(k) ='(') && (x(k) =';'))
if(x(k) =',')&&(x(k) =' ')
Rn(cont)=x(k);
cont=cont+1;
end
else
```

```
break;
end
end
Rnt=' ';

[nk nv]=size(Rn);
Rnt1=[Rn(1) Rn(2)];
if 'hx'==Rnt1
for jn=3:1:nv
Rnt(jn-2)= Rn(jn);
end
Rn=Rnt;
else
comentt=0;
for nv=1:1:b
x=Arq(nv,:);
x=char(x);
x=strtrim(x);
x=deblank(x);
n=find(isspace(x));

[n1 n2]=size(x);
cont=1;
pw=' ';
for j=1:1:n2
if ((x(j) =' ') && (x(j) =';'))
pw(cont)=x(j);
cont=cont+1;
else
break;
end
end
switch pw
case '//'
comentt=comentt+1;
case 'FUNCTION'
cont=1;
Rnv=' ';
for k=j+1:1:n2
if ((x(k) ='(') && (x(k) =';'))
if(x(k) =',')&&(x(k) =' ')
Rnv(cont)=x(k);
cont=cont+1;
end
else
break;
end
end
switch Rn
case Rnv
KV=nv-i;
if (KV<0)
KV=65535-abs(KV)+1+comentt;
Rn=dec2hex(KV);
break;
else
Rn=dec2hex(KV-comentt);
break;
end
end
end
end
end
else
Rn=0;
end
R=Rn;
end
end
```

APÊNDICE D – PROGRAMA DE COMUNICAÇÃO COM ISP1362

```
NOP;
MOVL 003D(1);
COPY 01(1),3(0); configura sinais iniciais isp
MOVL 0(0); REGL=0
COPY 1(0),0(0); R0[0]=0;
COPY 1(0),6(0); R0[6]=0
MOVL FFFF(0); REGL=FFFF
COPY 1(0),0(1); R1[0]=FFFF;
MOVL 2(0); REGL=2;
COPY 1(0),5(1); R1[5]=2;
MOVL 25(0); REGL=25
COPY 1(0),4(0); R0[4]=25;
MOVL 1(0); REGL=1;
COPY 1(0),5(0); R0[5]=1;
JUMP 3; PULA 3 INTRUÇÕES
BE 0(0),0(1),4(0); IF(R0[0]==R1[0])
ADD 0(0),5(0),0(0); R0[0]=R0[0]+R0[5];
NOP;
MOVL A8(0); MOVE ENDEREÇO DE TESTE PARA
COPY 1(0),2(0); COPIA VALOR PARA PORT_A
NOP;
MOVL 0031(0);
COPY 01(0),3(0); CONDICIONA SINAIS PARA COMANDO ISP
BSET 3(0),3; SET WR
BSET 3(0),2; SET CS
COPY 0(0),1(0);
COPY 1(0),2(0); COPIA VALOR PARA PORT_A
MOVL 0030(0);
COPY 01(0),3(0); CONDICIONA SINAIS PARA COMANDO ISP
BSET 3(0),2; SET CS
BSET 3(0),3; SET WR
BSET 3(0),0; SET A0
MOVL 28(0); MOVE ENDEREÇO DE TESTE PARA
COPY 1(0),2(0); COPIA VALOR PARA PORT_A
MOVL 0031(0);
COPY 01(0),3(0); CONDICIONA SINAIS PARA COMANDO ISP
BSET 3(0),3; SET WR
BSET 3(0),2; SET CS
MOVL FFFF(1);
COPY 01(1), 2(1);
MOVL 0028(0);
COPY 01(0),3(0); CONDICIONA SINAIS PARA COMANDO ISP
NOP;
COPY 02(0),04(1);
BSET 3(0),4; SET RD
BSET 3(0),2; SET CS
BSET 3(0),0; SET A0
MOVL 0000(1);
COPY 01(1), 2(1);
BNE 0(0),4(1),5(1);
ADD 6(0),5(0),6(0);
JUMP FFDC;
NOP;
COPY 6(0),4(0);
AND;
```

APÊNDICE E – PROGRAMA DA PORTA DE SEGURANÇA

E.1 Programa principal

```
NOP ;// ALGORITMO FINAL
CALL INICILIZA_LCD; // INCIALIZA O LCD
NOP ;
CALL INICIALIZA_TCL;// INCIALIZA O TECLADO
NOP ;
COPY 0(0),1E(1);
MOVL 341C(0); // CARREGA CARACTER GA PARA MEMORIA
WM 1(0),0;
MOVL 3A1C(0); // CARREGA CARACTER MA PARA MEMORIA
WM 1(0),1;
MOVL 341C(0); // CARREGA CARACTER GA PARA MEMORIA
WM 1(0),4;
MOVL 3A1C(0); // CARREGA CARACTER MA PARA MEMORIA
WM 1(0),5;
MOVL 3C31(0); // CARREGA CARACTER UN PARA MEMORIA
WM 1(0),6;
MOVL 434D(0); // CARREGA CARACTER IP PARA MEMORIA
WM 1(0),7;
NOP ; // INICIO
BCLEAR 3(0),7;
BCLEAR 3(0),6;
BCLEAR 3(0),13;
BCLEAR 3(0),12;
BSET 3(0),11;
BSET 3(0),10;
BSET 3(0),9;
BSET 3(0),8;
COPY 0(0),15(1); // ZERA VALOR DO ESC
CALL LIMPA_LCD;// LIMPA LCD
NOP ;
CALL M1 ;// ESCREVE MENSAGEM DE EM VINDO E PEDE USUARIO
NOP ;
CALL F_USER; // CHAMA FUNÇÃO DE CAPTURA DE USUARIO
NOP ;
MOVL 2(0);
COPY 1(0),0(1);
BE 15(1),0(0),0(1);// VERIFICA SE O ESC FOI APERTADO
JUMP FFED;
NOP ; // TERMINO DO PASSO 1
CALL LIMPA_LCD;// LIMPA LCD
NOP ;
CALL M2 ;// CHAMA FUNÇÃO QUE ESCREVE A MENSAGEM DE SENHA
NOP ;
CALL F_SENHA; // CHAMA FUNÇÃO DE CAPTURA DE SENHA
NOP ;
MOVL 2(0);
COPY 1(0),0(1);
BE 15(1),0(0),0(1);// VERIFICA SE O ESC FOI APERTADO
JUMP FFE2;
NOP ; // TERMINO DO PASSO 2
CALL F_CPUSR; // CHAMA FUNÇÃO QUE VERIFICA A CORRESPONDENCIA DE USUARIO
NOP ; // TERINO DO PASSO 3
MOVL 12(1);
COPY 1(1),0(1);
BE 1(0),0(0),0(1); // VERIFICA SE HOUVE CORRESPONDENCIA
CALL LIMPA_LCD;// CHAMA FUNÇÃO LIMPA LCD
MOVL 1(1);
ADD A(0),1(1),A(0);// INCREMENTA NUMERO DE ERROS
MOVL 4(1)
MOVL 6(0);
BLE A(0),1(1),1(0);
CALL M5 ;// CHAMA FUNÇÃO DE MENSAGEM DE USUARIO OU SENHA INCORRETO
MOVL 1388(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FFD1;// VOLTA PARA O INICIO
CALL M51;// CHAMA FUNÇÃO DE MENSAGEM DE 4 ERRO E AGURADA 1 MIN
COPY 0(0),A(0);
MOVL EA60(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FFCB;// VOLTA PARA O INICIO;
NOP ;// FINAL DO PASSO
COPY 0(0),A(0);
MOVL F(0);
COPY 1(0),0(1);
BE 17(0),0(0),0(1);// VERIFICA SE É O USUARIO ADMINISTRADOR
CALL LIMPA_LCD ; // LIMPA LCD
BSET 3(0),7;
BSET 3(0),6;
BSET 3(0),13;
BSET 3(0),12;
BCLEAR 3(0),11;
BCLEAR 3(0),10;
BCLEAR 3(0),9;
BCLEAR 3(0),8;
CALL M6 ;// CHAMA FUNÇÃO MENSAGEM PORTA ABERTA
MOVL 2710(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FFB8;
NOP ;// FINAL DO PASSO 6
CALL MENU ;// CHAMA FUNÇÃO MENU
NOP ;
MOVL 2(0);
COPY 1(0),0(1);
MOVL 5(0);
BNE 5(0),1(0),0(1);
CALL FFEB;
MOVL 2(0);
COPY 1(0),0(1);
BE 15(1),0(0),0(1); // VERIFICO
```

```
SE FOI APERTADO ESC
JUMP FFAC;
NOP ;
CALL LIMPA_LCD;
MOVL 3(0);
COPY 1(0),0(1);
MOVL 4(0);
BE 5(0),1(0),0(1);
CALL M8 ; // CHAMA FUNÇÃO MENSAGEM ENTRADA DE USUARIO
CALL F_USER;// CHAMA FUNÇÃO PARA ENTRADA DE USUARIO
NOP ;
MOVL 2(0);
BE 15(1),0(0),1(0); // VERIFICO SE FOI APERTADO ESC
JUMP FFA0;
NOP ;
MOVL 11(0);// CARREGA O VALOR DO PULO
COPY 1(0),0(1);
MOVL 1(0);// VERIFICA SE É A OPÇÃO 10
BNE 5(0),1(0),0(1);//
CALL LIMPA_LCD; // LIMPA LCD
CALL M9; // CHAMA FUNÇÃO MENSAGEM DE SENHA
CALL F_SENHA;// CHAMA FUNÇÃO DE CAPTURA DE SENHA
MOVL 2(0);
BE 15(1),0(0),1(0); // VERIFICO SE FOI APERTADO ESC
JUMP FF95;
NOP ;
CALL S_USER;// CHAMA FUNÇÃO QUE SALVA USUARIO
COPY 13(0),17(0);
CALL S_SENHA;// CHAMA FUNÇÃO QUE SALVA SENHA
CALL LIMPA_LCD;
CALL M12 ;// CHAMA MENSAGEM DE USUARIO SALVO
MOVL BB8(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FF8B;
NOP ;
MOVL 1A(0);// CARREGA O VALOR DO PULO
COPY 1(0),0(1);
MOVL 2(0);// VERIFICA SE É A OPÇÃO 2
BNE 5(0),1(0),0(1);//
CALL F_BUSC;// CHAMA FUNÇÃO QUE BUSCA USUARIO
MOVL 7(1);
COPY 1(1),0(1);
BE 1(0),0(0),0(1);// VERIFICA SE DEU CORRESPONDENCIA
CALL LIMPA_LCD;
CALL M13;// USUARIO NÃO ENCONTRADO;
MOVL BB8(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FF7C;
CALL LIMPA_LCD; // LIMPA LCD
CALL M9; // CHAMA FUNÇÃO MENSAGEM DE SENHA
CALL F_SENHA;// CHAMA FUNÇÃO DE CAPTURA DE SENHA
MOVL 2(0);
BE 15(1),0(0),1(0); // VERIFICO SE FOI APERTADO ESC
JUMP FF76;
COPY 6(1),17(0);
CALL S_SENHA;// CHAMA FUNÇÃO QUE SALVA SENHA
CALL LIMPA_LCD;
CALL M12 ;// CHAMA MENSAGEM DE USUARIO SALVO
COPY 0(0),A(0);
MOVL BB8(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FF6D;
NOP ;
MOVL 12(0);// CARREGA O VALOR DO PULO
COPY 1(0),0(1);
MOVL 3(0);// VERIFICA SE É A OPÇÃO 3
BNE 5(0),1(0),0(1);//
CALL F_BUSC;// CHAMA FUNÇÃO QUE BUSCA USUARIO
MOVL 7(1);
COPY 1(1),0(1);
BE 1(0),0(0),0(1);// VERIFICA SE DEU CORRESPONDENCIA
CALL LIMPA_LCD;
CALL M13;// USUARIO NÃO ENCONTRADO;
MOVL 2710(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FF5E;
CALL EX_USER;// CHAMA FUNÇÃO QUE EXCLUI USUARIO
CALL LIMPA_LCD;
CALL M11 ;// CHAMA MENSAGEM DE USUARIO EXCLUIDO
MOVL 1388(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
JUMP FF57;
NOP ;
MOVL 7(0);// CARREGA O VALOR DO PULO
COPY 1(0),0(1);
MOVL 4(0);// VERIFICA SE É A OPÇÃO 2
BNE 5(0),1(0),0(1);//
CALL USR_SALV ;// CHAMA FUNÇÃO QUE MOSTRA USUARIOS SALVOS
MOVL 2(0);
BE 15(1),0(0),1(0); // VERIFICO SE FOI APERTADO ESC
JUMP FF4E;
NOP ;
JUMP FF96 ;// VOLTA PARA MENU
NOP ;// VOLTA PARA FUNÇÃO MENU
MOVL 2(0);// CARREGA O VALOR DO PULO
COPY 1(0),0(1);
MOVL 5(0);// VERIFICA SE É A OPÇÃO 2
BNE 5(0),1(0),0(1);//
JUMP FF81; //
JUMP FF45 ;
```

E.2 Inicialização do LCD

```
FUNCTION INICILIZA_LCD
NOP ;
MOVL 0(0); // ATRIBUIR ZERO A REGL
COPY 1(0),2(1); // CONFIGURA PORTA "A"COMO SAIDA
COPY 1(0),2(0); // ZERA VALOR DA PORTA "A"
COPY 1(0),3(1); // CONFIGURA PORTA "B"COMO SAIDA
COPY 1(0),3(0); // ZERA VALOR DA PORTA "B"
BSET 3(0),0; // LIGA DISPLAY
BSET 3(0),1;// DESABILITA BLON
MOVL 0E(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BSET 2(0),5;//SET BIT DE DADO DB5 DO DISPLAY
BSET 2(0),4;//SET BIT DE DADO DB4 DO DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNCAO ms
MOVL 04(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BSET 2(0),3;// SET BIT BD3 DO DISPLAY
BSET 2(0),2;// SET BIT BD2 DO DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
MOVL 0(0);
COPY 1(0),2(0);// ZERA SAIDA
BSET 2(0),1;// SET BIT DB1 DO DISPLAY
BSET 2(0),2;// SET BIT DB2 DO DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BSET 2(0),3; //SET BITS DB3 DO DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCO ms
MOVL 0(0);
COPY 1(0),2(0);// ZERA SAIDA
```

```
BSET 2(0),0;// SET SAIDA DB0 DO DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAOO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAOO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNCAO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZACAO DA FUNCAO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
NOP ; // RETORNO DA FUNACO ms
RET ; // RETORNO DE FUNCAO
```

E.3 Inicialização do PS2

```
NOP ;
BSET 3(0),15;// LIMPA VALOR DO PINO PORTB 15
BSET 3(0),14;// LIMPA VALOR DO PINO PORTB 14
BSET 3(1),15;// DETERMINA PINO PORTB 15 COMO ENTRADA
BSET 3(1),14; // DETERMINA PINO PORTB 14 COMO ENTRADA
RET ;
```

E.4 Função Escreve Mensagem no LCD

```
COPY 0(0),1C(0);// SET ENDEREÇO ZERO
COPY 0(0),1B(0);// SET COLUNA
CALL COLUNA;
NOP ;
MOVL 0042(0);// CODIGO DO CARACTER "b"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0045(0);// CODIGO DO CARACTER "E"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 004D(0);// CODIGO DO CARACTER "T"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0020(0);// CODIGO DO CARACTER
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0056(0);// CODIGO DO CARACTER "V"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0049(0);// CODIGO DO CARACTER "I"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 004E(0);// CODIGO DO CARACTER "N"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0044(0);// CODIGO DO CARACTER "D"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 004F(0);// CODIGO DO CARACTER "O"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0021(0);// CODIGO DO CARACTER "T"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CA-
```

```
RACTER
NOP ;
MOVL 2(0);
COPY 1(0),1C(0);// SET ENDEREÇO ZERO
CALL COLUNA;
MOVL 0055(0);// CODIGO DO CARACTER "u"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0053(0);// CODIGO DO CARACTER "S"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0055(0);// CODIGO DO CARACTER "U"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0041(0);// CODIGO DO CARACTER "A"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0052(0);// CODIGO DO CARACTER "R"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 0049(0);// CODIGO DO CARACTER "I"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 004F(0);// CODIGO DO CARACTER "O"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
MOVL 003D(0);// CODIGO DO CARACTER "T"
COPY 1(0),1A(0); // ATRIBUI A REGISTRADOR DE CARACTER
CALL ESCREVE; // CHAMA FUNCAO DE ESCRITA DE CARACTER
NOP ;
RET ;
```

E.5 Função de Captura do Nome de Usuário

```
FUNCTION F_USER
NOP ;// INCIO DA FUNÇÃO CAPTURA DE USUARIO
COPY 0(0),4(0);
MOVL 2(0);
COPY 1(0),1C(0);// CARREGA VALOR DA LINHA
MOVL 8(0);
COPY 1(0),1B(0);// CARREGA COLUNA
CALL COLUNA;
MOVL 4(0);
COPY 1(0),4(1);
NOP ;
MOVL 3(1);
NOP ;
COPY 0(0),16(1);// INCIA CONTADOR DE CARACTER
COPY 0(0),15(1);// TECLA ESC
COPY 0(0),17(1);// ZERA VALOR DO CARACTER 7 E 8
COPY 0(0),18(1);// ZERA VALOR DO CARACTER 5 E 6
COPY 0(0),19(1);// ZERA VALOR DO CARACTER 3 E 4
COPY 0(0),1A(1);// ZERA VALOR DO CARACTER 1 E 2 NOP
;
CALL TCL; // CHAMA FUNCAO
NOP ;
MOVL 28(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
MOVL 3(0);
COPY 1(0),0(1);// VALOR DO PULO DO
MOVL 5A(0); // CARREGA VALOR DO ENTER
BNE 5(1),1(0),0(1); // VERIFICA SE A TECLA APERTA É ENTER
BS 16(1),4(1),0(1);// VERIFICA SE É MAIOR QUE 4
RET ; // SE FORAM MAIS DO QUE 4 CARACTERES PODE SAIR PRO FIM
NOP ;
MOVL B(0);
COPY 1(0),0(1);//VALOR DO PULO
MOVL 66(0); // CARREGA VALOR DO BACKSOACE
BNE 5(1),1(0),0(1); // VERIFICA SE É BACKSPACE
BE 16(1),0(0),0(1);// VERIFICA SE É ZERO
CALL ZERA_USR;// CHAMA FUNÇÃO QUE ZERA USER
MOVL 1(0);// MOVE UM PARA SUBTRAIR
SUB 16(1),1(0),16(1);// SUBTRAIR 1 DO NUMERO DE CARACTER
MOVL 8(0);// DETERMINA INICIO DA POSIÇÃO
ADD 1(0),16(1),1B(0);// DETERMINA O INDICE
MOVL 2(0);// MOVE VALOR DE PARA SEGUNDA COLUNA
COPY 1(0),1C(0);// DEFINE A LINHA 2 COMO ENDERÇO
CALL APAGA; // CHAMA FUNCAO APAGA CARACTER
JUMP FFE6;
NOP ;
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
```

```
MOVL 76(0);// CARREGA VALOR DO ESC
BNE 5(1),1(0),0(1);// VERIFICA SE É O VALOR DO ESC
MOVL 1(0);// CARREGA VALOR DO ESC
COPY 1(0),15(1);// INDICA O APERTO DO ESC
RET ; // SAI PARA O FIM
NOP ;
CALL CONV;// CHAMA FUNCAO
NOP ;
COPY 1(0),1A(0);// ARMAZENA VALOR PARA SER ESCRITO
MOVL 2(0);
COPY 1(0),0(1);
MOVL 1(0);
BNE 1A(0),1(0),0(1); // VERIFICA SE É VALOR DESCARTAVEL
JUMP FFD6; // SE FOR VOLTA PARA INICIO
MOVL 2(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
MOVL 7(0),
BSE 16(1),1(0),0(1);// VERIFICA SE É MENOR OU IGUAL A 7 CARACTER
JUMP FFD1;
CALL ESCREVE; // CHAMA FUNÇÃO PARA ESCREVER
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
BNE 16(1),0(0),0(1); // VERIFICA SE É O PRIMEIRO CARACTER
COPY 5(1),17(1); // COPIO VALOR PARA REGISTRADOR
SHL 17(1),8,17(1);// DESLOCA O VALRO DO 8 VESES
JUMP 2F; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 1(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 2
ADD 17(1),5(1),17(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP 29;// VOLTA PARA O INICIO
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 2(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 3
COPY 5(1),18(1); // COPIO VALOR PARA REGISTRADOR
SHL 18(1),8,18(1);// DESLOCA O VALRO DO 8 VESES
JUMP 22; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
MOVL 3(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 4
ADD 18(1),5(1),18(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP 1C;// VOLTA PARA O INICIO
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
MOVL 4(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 5
COPY 5(1),19(1); // COPIO VALOR PARA REGISTRADOR
SHL 19(1),8,19(1);// DESLOCA O VALRO DO 8 VESES
JUMP 15; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 5(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 6
ADD 19(1),5(1),19(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP F;// VOLTA PARA O INICIO
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 6(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 7
COPY 5(1),1A(1); // COPIO VALOR PARA REGISTRADOR
SHL 1A(1),8,1A(1);// DESLOCA O VALOR DO 8 VESES
JUMP 8; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 7(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 8
ADD 1A(1),5(1),1A(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP 2;// VOLTA PARA O INICIO
NOP ;
MOVL 1(0);
ADD 16(1),1(0),16(1); // INCREMENTA VALOR DA ELETRA
JUMP FF98;// VOLTA PARA O INICIO
```

E.6 Função de Captura de Senha

```
FUNCTION F_SENHA
NOP ;// INCIO DA FUNÇÃO CAPTURA DE USUARIO
COPY 0(0),4(0);
MOVL 2(0);
COPY 1(0),1C(0);// CARREGA VALOR DA LINHA
MOVL 8(0);
COPY 1(0),1B(0);// CARREGA COLUNA
CALL COLUNA;
MOVL 4(0);
COPY 1(0),4(1);
NOP ;
MOVL 3(1);
NOP ;
COPY 0(0),16(1);// INCIA CONTADOR DE CARACTER
COPY 0(0),15(1);// TECLA ESC
COPY 0(0),1E(1);// ZERA VALOR DO CARACTER 7 E 8
COPY 0(0),1D(1);// ZERA VALOR DO CARACTER 5 E 6
COPY 0(0),1C(1);// ZERA VA-
```

```
LOR DO CARACTER 3 E 4
COPY 0(0),1B(1);// ZERA VALOR DO CARACTER 1 E 2
NOP ;
CALL TCL; // CHAMA FUNCAO
NOP ;
MOVL 28(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNCAO ms
MOVL 3(0);
COPY 1(0),0(1);// VALOR DO PULO DO
MOVL 5A(0); // CARREGA VALOR DO ENTER
BNE 5(1),1(0),0(1); // VERIFICA SE A TECLA APERTA É ENTER
BS 16(1),4(1),0(1);// VERIFICA SE É MAIOR QUE 4
RET ; // SE FORAM MAIS DO QUE 4 CARACTERES PODE SAIR PRO FIM
NOP ;
MOVL B(0);
COPY 1(0),0(1);//VALOR DO PULO
MOVL 66(0); // CARREGA VALOR DO BACKSOACE
BNE 5(1),1(0),0(1); // VERIFICA SE É BACKSPACE
BE 16(1),0(0),0(1);// VERIFICA SE É ZERO
CALL ZERA_SENHA; // FUNCAO QUE APAGA SENHA
MOVL 1(0);// MOVE UM PARA SUBTRAIR
SUB 16(1),1(0),16(1);// SUBTRAIR 1 DO NUMERO DE CARACTER
MOVL 8(0);// DETERMINA INICIO DA POSIÇÃO
ADD 1(0),16(1),1B(0);// DETERMINA O INDICE
MOVL 2(0);// MOVE VALOR DE PARA SEGUNDA COLUNA
COPY 1(0),1C(0);// DEFINE A LINHA 2 COMO ENDERÇO
CALL APAGA; // CHAMA FUNCAO APAGA CARACTER
JUMP FFE6;
NOP ;
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
MOVL 76(0);// CARREGA VALOR DO ESC
BNE 5(1),1(0),0(1);// VERIFICA SE É O VALOR DO ESC
MOVL 1(0);// CARREGA VALOR DO ESC
COPY 1(0),15(1);// INDICA O APERTO DO ESC
RET ; // SAI PARA O FIM
NOP ;
CALL CONV;// CHAMA FUNCAO
NOP ;
COPY 1(0),1A(0);// ARMAZENA VALOR PARA SER ESCRITO
MOVL 2(0);
COPY 1(0),0(1);
MOVL 1(0);
BNE 1A(0),1(0),0(1); // VERIFICA SE É VALOR DESCARTAVEL
JUMP FFD6; // SE FOR VOLTA PARA INICIO
MOVL 2(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
MOVL 7(0),
BSE 16(1),1(0),0(1);// VERIFICA SE É MENOR OU IGUAL A 7 CARACTER
JUMP FFD1;
MOVL 2A(0);
COPY 1(0),1A(0);
CALL ESCREVE; // CHAMA FUNÇÃO PARA ESCREVER
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
BNE 16(1),0(0),0(1); // VERIFICA SE É O PRIMEIRO CARACTER
COPY 5(1),1B(1); // COPIO VALOR PARA REGISTRADOR
SHL 1B(1),8,1B(1);// DESLOCA O VALRO DO 8 VESES
JUMP 2F; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 1(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 2
ADD 1B(1),5(1),1B(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP 29;// VOLTA PARA O INICIO
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 2(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 3
COPY 5(1),1C(1); // COPIO VALOR PARA REGISTRADOR
SHL 1C(1),8,1C(1);// DESLOCA O VALRO DO 8 VESES
JUMP 22; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 3(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 4
ADD 1C(1),5(1),1C(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP 1C;// VOLTA PARA O INICIO
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 4(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 5
COPY 5(1),1D(1); // COPIO VALOR PARA REGISTRADOR
SHL 1D(1),8,1D(1);// DESLOCA O VALRO DO 8 VESES
JUMP 15; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 5(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 6
ADD 1D(1),5(1),1D(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP F;// VOLTA PARA O INICIO
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
MOVL 6(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 7
COPY 5(1),1E(1); // COPIO VALOR PARA REGISTRADOR
SHL 1E(1),8,1E(1);// DESLOCA O VALOR DO 8 VESES
JUMP 8; // PULA PARA INICIO
MOVL 3(0);
COPY 1(0),0(1);// CARREGA VALRO DO PULO
```

```
MOVL 7(0);
BNE 16(1),1(0),0(1);// VERIFICA SE É O CARACTER 8
ADD 1E(1),5(1),1E(1);// DETERMINA O VALOR DO SEGUNDO CARACTER
JUMP 2;// VOLTA PARA O INICIO
NOP ;
MOVL 1(0);
ADD 16(1),1(0),16(1); // INCREMENTA VALOR DA LETRA
JUMP FF96;// VOLTA PARA O INICIO
```

E.7 Função que Verifica Correspondência do Usuário

```
FUNCTION F_CPUSR
NOP ; // FUNCAO BUSCA DE CORRESPONDÊNCIA
COPY 0(0),17(0);// INCIALIZA CONTADOR DE USUARIOS BUSCADOS
NOP ;
MOVL 34(0);
COPY 1(0),0(1);// CARREGA TAMANHO DO PULO
BL 17(0),13(0),0(1);//VERIFICA SE TODOS OS USUARIOS FORAM OBSERVADOS
COPY 0(0),16(0);// INCIALIZA CONTADOR DE ACERTOS
MOVL 8(0);
MULT 17(0),1(0),1F(1);// MULTIPLICO PELO INDICE PARA COMPARA BUSCA DE COMPARAÇÃO
MOVL 1(1);
MOVL 3(0);
COPY 1(0),0(1);// CAREGA VALOR DO PULO
RM 1(0),0;// LE A PRIMEIRA POSIÇÃO
BNE 17(1),1(0),0(1);//VERIFICA SE É IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;// PULA DUAS CASAS
JUMP 26;
RM 1(0),1;// LE PROXIMO CARACTER
BNE 18(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP 21;
RM 1(0),2;// LE PROXIMO CARACTER
BNE 19(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP 1C;
RM 1(0),3;// LE PROXIMO CARACTER
BNE 1A(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP 17;
RM 1(0),4;// LE PROXIMO CARACTER
BNE 1B(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP 12;
RM 1(0),5;// LE PROXIMO CARACTER
BNE 1C(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP D;
RM 1(0),6;// LE PROXIMO CARACTER
BNE 1D(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP 8;
RM 1(0),7;// LE PROXIMO CARACTER
BNE 1E(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
MOVL 8(0);
BNE 16(0),1(0),0(1);// VEFICA SE O NUMERO DE ACERTOS FOI ALCANÇADO
RETL 0(0);
NOP ;
MOVL 1(0);
ADD 17(0),1(0),17(0);//INCREMENTA PARA O PROXIMO USUARIOS
JUMP FFCB;
RETL 1(0); // RETORNA ERRO
```

E.8 Função Menu de Opções

```
FUNCTION MENU
NOP ;// FUNCAO MENU INTERATIVO
MOVL 1(1);
CALL LIMPA_LCD;
MOVL 1(0);
COPY 1(0),5(0);// DETERMINA PRIMEIRA FUNÇAO
CALL M7;// CHAMA FUNÇAO QUE OPEÇÕES
CALL M71;// ESCREVE OPÇÃO 1
MOVL 5(0);
COPY 1(0),6(0);
MOVL 1(0);
COPY 1(0),4(0);
NOP ;
CALL TCL; // CHAMA FUNÇAO DO TECLADO
NOP ;
MOVL 28(0);// INICIALIZACAO DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI
```

```
CONTAR
CALL fMS; // CHAMA FUNCAO ms
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
MOVL 7602(0);// CARREGA VALOR DO ESC
BNE 5(1),1(0),0(1);// VERIFICA SE É O VALOR DO ESC
MOVL 1(0);// CARREGA VALOR DO ESC
COPY 1(0),15(1);// INDICA O APERTO DO ESC
RET ; // SAI PARA O FIM
NOP ;
MOVL 2(0);
COPY 1(0),0(1);// CARREGA PULO
MOVL 5A03(0);
BNE 5(1),1(0),0(1);// VERIFICA SE É ENTER
RET ;
MOVL 5(0);
COPY 1(0),0(1);// CARREGA PULO
MOVL 6051(0);// ESQUERDA
BNE 5(1),1(0),0(1);// VERIFICA SE É PARA DIREITA
BE 5(0),6(0),0(1);// VERIFICA SE A OPÇÃO É 5
ADD 5(0),1(1),5(0);// ADICIONA MAIS UM
CALL LIMPA_LCD;
CALL M7;// CHAMA MENU OPÇÕES
NOP ;
MOVL 5(0);
COPY 1(0),0(1);// CARREGA PULO
MOVL 602D(0);
BNE 5(1),1(0),0(1);//VERIFICA SE É SETA A ESQUERDADA
BE 5(0),1(1),0(1);// VERIFICA SE A OPÇÃO É 5
SUB 5(0),1(1),5(0);// ADICIONA MAIS UM
CALL LIMPA_LCD;
CALL M7;// CHAMA MENU OPÇÕES
NOP ;
MOVL 3(0);
COPY 1(0),0(1);
MOVL 1(0);
BNE 5(0),1(0),0(1); // VERIFICO SE É OPÇÃO 1
CALL M71; // CHAMA OPÇÃO 1
JUMP 18;
MOVL 3(0);
COPY 1(0),0(1);
MOVL 2(0);
BNE 5(0),1(0),0(1); // VERIFICO SE É OPÇÃO 1
CALL M72; // CHAMA OPÇÃO 1
JUMP 12;
MOVL 3(0);
COPY 1(0),0(1);
MOVL 3(0);
BNE 5(0),1(0),0(1); // VERIFICO SE É OPÇÃO 1
CALL M73; // CHAMA OPÇÃO 1
JUMP C;
MOVL 3(0);
COPY 1(0),0(1);
MOVL 4(0);
BNE 5(0),1(0),0(1); // VERIFICO SE É OPÇÃO 1
CALL M74; // CHAMA OPÇÃO 1
JUMP 6;
MOVL 3(0);
COPY 1(0),0(1);
MOVL 5(0);
BNE 5(0),1(0),0(1); // VERIFICO SE É OPÇÃO 1
CALL M75; // CHAMA OPÇÃO 1
JUMP FFBC;
```

E.9 Função que Salva um Novo Usuário

```
FUNCTION S_USER
NOP ; // FUNCAO SALVA USUARIO
MOVL 1(0);
ADD 13(0),1(0),13(0);// INCREMETA USUARIO
MOVL 8(0);
MULT 13(0),1(0),1F(1);// DETERMINA ENDEREÇO
WM 17(1),0;// SALVA PRIMEIRO CARCTER DO USUARIO
WM 18(1),1;// SALVA PRIMEIRO CARCTER DO USUARIO
WM 19(1),2;// SALVA PRIMEIRO CARCTER DO USUARIO
WM 1A(1),3;// SALVA PRIMEIRO CARCTER DO USUARIO
RET ;
```

E.10 Função que Salva Senha

```
FUNCTION S_SENHA
NOP ;// FUNCAO SALVA SENHA
```

```
MOVL 8(0);
MULT 17(0),1(0),1F(1);// DETERMINA ENDEREÇO
WM 1B(1),4;// SALVA PRIMEIRO CARACTER DA SENHA
WM 1C(1),5;
WM 1D(1),6;
WM 1E(1),7;
RET ;
```

E.11 Função de Busca de Usuário

```
FUNCTION F_BUSC
NOP ; // FUNCAO BUSCA DE CORRESPONDÊNCIA
COPY 0(0),17(0);// INCIALIZA CONTADOR DE USUARIOS BUSCADOS
MOVL 22(0);
COPY 1(0),0(1);// CARREGA TAMANHO DO PULO
BL 17(0),13(0),0(1);//VERIFICA SE TODOS OS USUARIOS FORAM OBSERVADOS
COPY 0(0),16(0);// INCIALIZA CONTADOR DE ACERTOS
MOVL 8(0);
MULT 17(0),1(0),1F(1);// MULTIPLICO PELO INDICE PARA COMPARA BUSCA DE COMPARAÇÃO
MOVL 1(1);
MOVL 3(0);
COPY 1(0),0(1);// CAREGA VALOR DO PULO
RM 1(0),0;// LE A PRIMEIRA POSIÇÃO
BNE 17(1),1(0),0(1);//VERIFICA SE É IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;// PULA DUAS CASAS
JUMP 14;
RM 1(0),1;// LE PROXIMO CARACTER
BNE 18(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP F;
RM 1(0),2;// LE PROXIMO CARACTER
BNE 19(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
JUMP 2;
JUMP A;
MOVL 6(0);
COPY 1(0),0(1);
RM 1(0),3;// LE PROXIMO CARACTER
BNE 1A(1),1(0),0(1);// VERIFICA SE IGUAL
ADD 16(0),1(1),16(0);// INCREMENTA OS ACERTOS
NOP ;
COPY 17(0),6(1);// VEFICA SE O NUMERO DE ACERTOS FOI ALCANÇADO
RETL 0(0);
NOP ;
MOVL 1(0);
ADD 17(0),1(0),17(0);//INCREMENTA PARA O PROXIMO USUARIOS
JUMP FFDD;
RETL 1(0); // RETORNA ERRO
```

E.12 Função Exclui Usúario

```
FUNCTION EX_USER
NOP ; // FUNCAO EXCLUI USUARIO
MOVL 17(0);
COPY 1(0),0(1);// CARREGA VALOR DO PULO
BL 17(0),13(0),0(1); // VERIFICA SE O NUMERO DE USUARIOS FOI ALCANÇADO
MOVL 1(1);
ADD 17(0),1(1),17(0);// SOMO MAIS UM
MOVL 8(0);//
MULT 17(0),1(0),1F(1);// COMPONHO O ENDEREÇO DE LEITURA
RM 17(1),0;
RM 18(1),1;
RM 19(1),2;
RM 1A(1),3;
RM 1B(1),4;
RM 1C(1),5;
RM 1D(1),6;
RM 1E(1),7;
SUB 1F(1),1(0),1F(1);// DECREMENTO 8 DO ENDEREÇO
WM 17(1),0;
WM 18(1),1;
WM 19(1),2;
WM 1A(1),3;
WM 1B(1),4;
WM 1C(1),5;
WM 1D(1),6;
WM 1E(1),7;
JUMP FFEA;
SUB 13(0),1(1),13(0);// SUBTRAIO UM USUARIO
RET ;
```

E.13 Função Mostra Usuários Salvos

```
FUNCTION USR_SALV
CALL LIMPA_LCD;
NOP ;
COPY 0(0),9(1);
COPY 0(0),1F(1);
MOVL 2D(0);
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),0;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),1;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),2;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),3;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 1(0);
COPY 1(0),4(0);
MOVL 20(0);
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 7F(0);
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 7E(0);
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 1(0);
COPY 1(0),4(0);
NOP ;
CALL TCL;
NOP ;
MOVL 28(0);// INICIALIZACAO
DA FUNCAO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO
QUANTOS ms A FUNCAO VAI
CONTAR
CALL fMS; // CHAMA FUNCAO
ms
MOVL 4(0);
COPY 1(0),0(1);// CARREGA VA-
LOR DO PULO
MOVL 7602(0);// CARREGA VA-
LOR DO ESC
BNE 5(1),1(0),0(1);// VERIFICA
SE É O VALOR DO ESC
MOVL 1(0);// CARREGA VALOR
DO ESC
COPY 1(0),15(1);// INDICA O
APERTO DO ESC
RET ; // SAI PARA O FIM
NOP ;
MOVL 2(0);
COPY 1(0),0(1);// CARREGA
PULO
MOVL 5A03(0);
BNE 5(1),1(0),0(1);// VERIFICA
SE É ENTER
RET ;
MOVL 1(1);
MOVL 5(0);
COPY 1(0),0(1);// CARREGA
PULO
MOVL 6051(0);// ESQUERDA
BNE 5(1),1(0),0(1);// VERIFICA
SE É PARA DIREITA
NOP ;
ADD 9(1),1(1),9(1);// ADICIONA
MAIS UM
CALL LIMPA_LCD;
NOP ;// CHAMA MENU OPÇÕES
MOVL 8(0);
COPY 1(0),0(1);// CARREGA
PULO
MOVL 602D(0);
BNE 5(1),1(0),0(1);//VERIFICA
SE É SETA A ESQUERDADA
MOVL 4(0);
COPY 1(0),0(1);// CARREGA
PULO
BE 9(1),0(0),0(1);// VERIFICA SE
A OPÇÃO É 5
SUB 9(1),1(1),9(1);// ADICIONA
MAIS UM
CALL LIMPA_LCD;
JUMP 2;// CHAMA MENU OP-
ÇÕES
JUMP FFD8;
COPY 0(0),1B(0);
COPY 0(0),1C(0);
CALL COLUNA;
MOVL 8(0);
MULT 9(1),1(0),1F(1);
MOVL 2D(0);
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),0;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),1;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
```

```
CALL ESCREVE;
RM 1(0),2;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
RM 1(0),3;
SHR 1(0),8,1(1);
SHL 1(1),8,1(1);
SUB 1(0),1(1),6(0);
SHR 1(1),8,5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
COPY 6(0),5(1);
CALL CONV_ESP;
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 1(0);
COPY 1(0),4(0);
MOVL 20(0);
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 7F(0);
COPY 1(0),1A(0);
CALL ESCREVE;
MOVL 7E(0);
COPY 1(0),1A(0);
CALL ESCREVE;
JUMP FF94;
```

E.14 Função que Define a Coluna a ser Escrita no LCD

```
FUNCTION COLUNA
NOP ;
MOVL 3(0); // DEFINE TAMANHO DO PULO
COPY 1(0),0(1); // ATRIBUI O TAMANHO DO PULO
MOVL 2(0); // DEFIN O VALOR PARA COMPARACAO
BE 1C(0),1(0),0(1); // VERIFICA SE A OPECAO É ESCRITA NA SEGUNDA COLUNA
COPY 0(0),1D(0); // CASO NAO ATRIBUI ZERO PARA 1D
JUMP 3; // PULA DUA INSTRUCOES
MOVL 40(0);// CASO SEJA A OPCAO DE ESCREVER NA SEGUNDSA LINHA ATRIBUI HX40
COPY 1(0),1D(0); //DEFINE 1D CIOM VALOR 40
ADD 1D(0),1B(0),1D(0);// SOMA O INDICE COMO O VALOR DE 1d
CALL ADDR; // CHAMA FUNÇÃO DE ENDEREÇO
NOP ; // RETORNO DA FUNÇÃO DE ENDEREÇO DO DISPLAY
RET ; // RETORNA PARA FUNÇÃO CHAMADAL
```

E.15 Função que define a linha a ser escrito no LCD

```
FUNCTION ADDR
NOP ;
MOVL 0(0);
COPY 1(0),2(0);// LIMPA DADOS DO DISPLAY
MOVL 3(0); // DETERMINA O TAMANHO DO PULO DAS CONDICOES
COPY 1(0),0(1); // SET VALOR DO PULO
BCLEAR 2(0),9;//DESABILITA ESCRITA NO DISPLAY
BCLEAR 2(0),10;//ENVIO DE COMANDO
BSET 2(0),7;//DEFINI ENTRADA DE ENDEREÇO
BBCLEAR 1D(0),0,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),0;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),0; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1D(0),1,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),1;//SET BIT DB1 DO DISPLAY
JUMP 2;
BCLEAR 2(0),1; // CLEAR BIT BD1 DISPLAY
BBCLEAR 1D(0),2,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),2;//SET BIT DB2 DO DISPLAY
JUMP 2;
BCLEAR 2(0),2; // CLEAR BIT
```

```
BD2 DISPLAY
BBCLEAR 1D(0),3,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),3;//SET BIT DB3 DO DISPLAY
JUMP 2;
BCLEAR 2(0),3; // CLEAR BIT BD3 DISPLAY
BBCLEAR 1D(0),4,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),4;//SET BIT DB4 DO DISPLAY
JUMP 2;
BCLEAR 2(0),4; // CLEAR BIT BD4 DISPLAY
BBCLEAR 1D(0),5,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),5;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),5; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1D(0),6,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),6;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),6; // CLEAR BIT BD0 DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNCAO ms
RET ;
```

E.16 Limpa Caracteres do LCD

```
FUNCTION LIMPA_LCD
NOP ;
COPY 0(0),2(0); // LIMPA PORTA DE SAIDA
BSET 2(0),0;// COMANDO PARA LIMPAR DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ;
RET ;
```

E.17 Função que Lê Carácter do Teclado

```
FUNCTION TCL
NOP ;
MOVL 0(0);
COPY 1(0),0(0);
COPY 0(0),5(1);
MOVL 3(0);
COPY 0(0),7(1);
COPY 1(0),9(0);
NOP ;
MOVL E4(0);
COPY 1(0),8(0);
NOP ;
MOVL 7(0);
COPY 1(0),17(0);
MOVL 1(1);
```

```
MOVL A(0);
COPY 1(0),0(1);
MOVL FFFF(0);
COPY 1(0),7(0);
BBCLEAR 3(0),15,0(1);
BE 7(0),0(0),9(0);
SUB 7(0),1(1),7(0);
JUMP FFFD;
BE 8(0),0(0),9(0);
SUB 8(0),1(1),8(0);
JUMP FFF8;
MOVL 76(0)
COPY 1(0),5(1);
RET ; // COLOCAR RET AQUI
NOP ;
BBSET 3(0),14,0(1);
NOP ;
JUMP FFFE;
NOP ;
BBCLEAR 3(0),14,0(1);
NOP ;
JUMP FFFE;
NOP ;
NOP ;
NOP ;
NOP ;
NOP ;
NOP ;
NOP ;
NOP ;
NOP ;
BBSET 3(0),15,0(1);
SHR 5(1),1,5(1);
JUMP 3;
BSET 5(1),7;
SHR 5(1),1,5(1);
MOVL 3(0);
COPY 1(0),0(1);
BE 17(0),0(0),0(1);
SUB 17(0),1(1),17(0);
JUMP FFE7;
MOVL 1(1);
MOVL 4(0);
COPY 1(0),0(1)
BE 4(0),0(0),0(1);
SUB 4(0),1(1),4(0);
COPY 5(1),7(1);
JUMP FFCF;
SHL 7(1),8,7(1);
ADD 5(1),7(1),5(1);
RET ;
```

E.18 Converte Carácter do Teclado para ASCII

```
FUNCTION CONV NOP ; // FUNCAO CONVERTE CARACTER
COPY 5(1),17(0);
MOVL 2(0);
COPY 1(0),0(1);
MOVL 1C(0);
BNE 17(0),1(0),0(1);
RETL 41(0);
MOVL 32(0);
BNE 17(0),1(0),0(1);
RETL 42(0);
MOVL 21(0);
BNE 17(0),1(0),0(1);
RETL 43(0);
MOVL 23(0);
BNE 17(0),1(0),0(1);
RETL 44(0);
MOVL 24(0);
BNE 17(0),1(0),0(1);
RETL 45(0);
MOVL 2B(0);
BNE 17(0),1(0),0(1);
RETL 46(0);
MOVL 34(0);
BNE 17(0),1(0),0(1);
RETL 47(0);
MOVL 33(0);
BNE 17(0),1(0),0(1);
RETL 48(0);
MOVL 43(0);
BNE 17(0),1(0),0(1);
RETL 49(0);
MOVL 3B(0);
BNE 17(0),1(0),0(1);
RETL 4A(0);
MOVL 42(0);
BNE 17(0),1(0),0(1);
RETL 4B(0);
MOVL 4B(0);
BNE 17(0),1(0),0(1);
RETL 4C(0);
MOVL 3A(0);
BNE 17(0),1(0),0(1);
RETL 4D(0);
MOVL 31(0);
BNE 17(0),1(0),0(1);
RETL 4E(0);
MOVL 44(0);
BNE 17(0),1(0),0(1);
RETL 4F(0);
MOVL 4D(0);
BNE 17(0),1(0),0(1);
RETL 50(0);
MOVL 15(0);
BNE 17(0),1(0),0(1);
RETL 51(0);
MOVL 2D(0);
BNE 17(0),1(0),0(1);
RETL 52(0);
MOVL 1B(0);
BNE 17(0),1(0),0(1);
RETL 53(0);
MOVL 2C(0);
BNE 17(0),1(0),0(1);
RETL 54(0);
MOVL 3C(0);
BNE 17(0),1(0),0(1);
RETL 55(0);
MOVL 2A(0);
BNE 17(0),1(0),0(1);
RETL 56(0);
MOVL 1D(0);
BNE 17(0),1(0),0(1);
RETL 57(0);
MOVL 22(0);
BNE 17(0),1(0),0(1);
RETL 58(0);
MOVL 35(0);
BNE 17(0),1(0),0(1);
RETL 59(0);
MOVL 1A(0);
BNE 17(0),1(0),0(1);
RETL 5A(0);
MOVL 45(0);
BNE 17(0),1(0),0(1);
RETL 30(0);
MOVL 16(0);
BNE 17(0),1(0),0(1);
RETL 31(0);
MOVL 1E(0);
BNE 17(0),1(0),0(1);
RETL 32(0);
MOVL 26(0);
BNE 17(0),1(0),0(1);
RETL 33(0);
MOVL 25(0);
BNE 17(0),1(0),0(1);
RETL 34(0);
MOVL 2E(0);
BNE 17(0),1(0),0(1);
RETL 35(0);
MOVL 36(0);
```

```
BNE 17(0),1(0),0(1);
RETL 36(0);
MOVL 3D(0);
BNE 17(0),1(0),0(1);
RETL 37(0);
MOVL 3E(0);
BNE 17(0),1(0),0(1);
RETL 38(0);
MOVL 46(0);
BNE 17(0),1(0),0(1);
RETL 39(0);
RETL 1(0);
```

E.19 Apaga Carácter do LCD

```
FUNCTION APAGA
NOP ;
CALL COLUNA; // ENDERECO A SER APAGADO
NOP ;
MOVL 20(0); // DETERMINA VALOR DO ESPAÇO EM BRANCO
COPY 1(0),1A(0);//SALVA CARACTER NO REGISTRADOR DE CARACTER
CALL ESCREVE;// CHAMA FUNAÇÃO LETRA
NOP ;
CALL COLUNA;// REDEFINE ESSE COMO O ENDERECO A SER ESCRITO
NOP ;
RET ;
```

E.20 Escreve Carácter no LCD

```
FUNCTION ESCREVE
NOP ;
MOVL 0(0);
COPY 1(0),2(0);
MOVL 3(0); // DETERMINA O TAMANHO DO PULO DAS CONDICOES
COPY 1(0),0(1); // SET VALOR DO PULO
BCLEAR 2(0),9;//DESABILITA ESCRITA NO DISPLAY
BSET 2(0),10;//ENVIO DE COMANDO
BBCLEAR 1A(0),0,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),0;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),0; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),1,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),1;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),1; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),2,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),2;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),2; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),3,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),3;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),3; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),4,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),4;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),4; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),5,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),5;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),5; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),6,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),6;//SET BIT DB0 DO DISPLAY
JUMP 2;
BCLEAR 2(0),6; // CLEAR BIT BD0 DISPLAY
BBCLEAR 1A(0),7,0(1);//TESTE SE O BIT DO ENDEREÇO É 0 E PULA DUAS CASA SE SIM
BSET 2(0),7;//SET BIT DB0 DO DISPLAY
JUMP 2;
```

BCLEAR 2(0),7; // CLEAR BIT BD0 DISPLAY
BSET 2(0),9;//ABILITA ESCRITA NO DIPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 1 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNO ms
BCLEAR 2(0),9; //DESABILITA ESCRITA NO DISPLAY
MOVL 0(0);// INICIALIZAO DA FUNO ms PARA 15 ms
COPY 1(0),1E(0);// DETERNO QUANTOS ms A FUNCAO VAI CONTAR
CALL fMS; // CHAMA FUNO ms
NOP ; // RETORNO DA FUNCAO ms
RET ; // RETORNO DA FUNCAO

E.21 Função Delay de Mili-Segundos

FUNCTION fMS

NOP ;

MOVL 0003(0); // CARREGA VALRO DO PULO

COPY 1(0),0(1); // CARREGA VALOR DO PULO PARA REGISTRADOR TEMPORARIO

MOVL 014C(0); // MOVE VALOR DO NUMERO DE REPITIÇÕES

MOVL 0001(1); // MOVE VALOR A SER DECESCIDO

BE 1(0),0(0),0(1);// SE O VALOR DO REGISTRADOR DE NUMERO DE PULOS FOR IGUAL A ZERO SAI DO PULO

SUB 1(0),1(1),1(0);// SUBTRAI O NUMERO DE LoopS

JUMP FFFE; // VOLTA PARA DUAS LINHAS ANTES

BE 1E(0),0(0),0(1);// VERIFICA SE O NUMERO DE ms FOI ALCANÇADO

SUB 1E(0),1(1),1E(0);// SUBTRAI O NUMERO DE 1ms

JUMP FFF9; // VOLTA 7 LINHAS PARA REINICIAR A CONTAGEM

RET ; // RETORNBA PARA ONDE FOI CHAMADO