LEONARDO AUGUSTO CASILLO

Projeto e implementação em FPGA de um processador com conjunto de instrução reconfigurável utilizando VHDL

Natal-RN

Maio de 2005

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

LEONARDO AUGUSTO CASILLO

Projeto e implementação em FPGA de um processador com conjunto de instrução reconfigurável utilizando VHDL

Dissertação de mestrado apresentada à banca examinadora do Programa de Pós-Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte, como requisito para a obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Dr. Ivan Saraiva Silva

Natal-RN

Maio de 2005

LEONARDO AUGUSTO CASILLO

Projeto e implementação em FPGA de um processador com conjunto de instrução reconfigurável utilizando VHDL

Dissertação de mestrado apresentada à banca examinadora do Programa de Pós-Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte, como requisito para a obtenção do título de Mestre em Sistemas e Computação.

BANCA EXAMINADORA

Prof. Dr. Ivan Saraiva Silva
DIMAp - UFRN

Prof. Dr. Marcelo Alves Barros
UFCG - PB

Prof. Dr. David Paul Boris Déharbe
DIMAp - UFRN

Prof. Dr. Eduardo Bráulio Wanderley Netto
CEFET - RN

Agradecimentos

Agradeço primeiramente a Deus por ter dado saúde e força a mim e a minha família para enfrentar as dificuldades impostas durante esta caminhada.

Aos meus pais Francisco e Cristina por serem os principais responsáveis por ter chegado aonde cheguei. Pelo amor e carinho, cuidado, paciência, força, e paitrocínio.

A minha namorada, amiga, companheira, entre outros adjetivos, Danielle, por todo amor e ajuda. Obrigado por ter tido paciência durante todas as crises de stress obtidas durante este período de mestrado.

Ao meu orientador Ivan Saraiva, por todas as ótimas conversas sobre futebol, política, religião, entre outros assuntos relevantes e irrelevantes, e pela orientação durante este mestrado.

Ao professor Galileu Batista de Souza, por me ajudar em várias questões essenciais para o desenvolvimento deste projeto.

Aos amigos e colegas de sofrimento do Programa de Pós Graduação em Sistemas e Computação da UFRN: Adriana (Takahashi), Adriano (Fú), Augusto (Mancha), Angélica (Keka), Camilinha, Glaucia (Legauzinha), Joseane (Jou), Macilon (Makz), Michael (Schuenk), Stiff e Thais (Tatá). O apoio que demos um ao outro foi importante para que todos nós juntos atingíssemos nossas metas.

Ao grupo de pesquisa em Arquitetura e Sistemas Integrados: Linária, Rodrigo, Daniel (Dr. Amaral), Mônica, Sílvio, Diego (Tomate), Tássia e Gustavo Girão.

Ao Programa Nacional de Microeletrônica (PNM), por financiar meus estudos.

As professoras e amigas Nádja R. Santiago e Karla Darlene, que prestaram um auxílio fundamental para que conseguisse terminar este projeto.

E a todos que ajudaram diretamente ou indiretamente para a realização deste objetivo...

MUITO OBRIGADO!

"Que as palavras que eu falo não sejam ouvidas como prece nem repetidas com fervor, Apenas respeitadas como a única coisa que resta a um homem inundado de sentimento. Porque metade de mim é o que eu ouço, mas a outra metade é o que calo."

Oswaldo Montenegro

Resumo

A Computação Reconfigurável é uma solução intermediária na resolução de problemas complexos, possibilitando combinar a velocidade do *hardware* com a flexibilidade do *software*. Uma arquitetura reconfigurável possui várias metas, entre estas o aumento de desempenho.

Dentre os vários segmentos em relação às arquiteturas reconfiguráveis, destacam-se os Processadores Reconfiguráveis. Estes processadores combinam as funções de um microprocessador com uma lógica reconfigurável e podem ser adaptados depois do processo de desenvolvimento. Processadores com Conjunto de Instruções Reconfiguráveis (RISP - *Reconfigurable Instruction Set Processors*) são um subconjunto dos processadores reconfiguráveis, que visa como meta a reconfiguração do conjunto de instruções do processador, envolvendo características referentes aos padrões de instruções como formatos, operandos, e operações elementares.

Este trabalho possui como objetivo principal o desenvolvimento de um processador RISP, combinando as técnicas de configuração do conjunto de instruções do processador executadas em tempo de desenvolvimento, e de reconfiguração do mesmo em tempo de execução. O projeto e implementação em VHDL deste processador RISP tem como intuito provar a aplicabilidade e a eficiência de dois conceitos: utilizar mais de um conjunto de instrução fixo, com apenas um ativo em determinado momento, e a possibilidade de criar e combinar novas instruções, de modo que o processador passe a reconhecê-las e utilizá-las em tempo real como se estas existissem no conjunto de instrução fixo.

A criação e combinação de instruções é realizada mediante uma unidade de reconfiguração incorporada ao processador. Esta unidade permite que o usuário possa enviar instruções customizadas ao processador para que depois possa utilizá-las como se fossem instruções fixas do processador.

Neste trabalho também encontram-se simulações de aplicações envolvendo instruções fixas e customizadas e resultados das comparações entre estas aplicações em relação ao consumo de potência e ao tempo de execução que confirmam a obtenção das metas para as quais o processador foi desenvolvido.

Palavras-Chave: processadores reconfiguráveis, RISP, VHDL, conjunto de instrução.

Abstract

The Reconfigurable Computing is an intermediate solution at the resolution of complex problems, making possible to combine the speed of the hardware with the flexibility of the software. An reconfigurable architecture possess some goals, among these the increase of performance. The use of reconfigurable architectures to increase the performance of systems is a well known technology, specially because of the possibility of implementing certain slow algorithms in the current processors directly in hardware.

Amongst the various segments that use reconfigurable architectures the reconfigurable processors deserve a special mention. These processors combine the functions of a microprocessor with a reconfigurable logic and can be adapted after the development process. Reconfigurable Instruction Set Processors (RISP) are a subgroup of the reconfigurable processors, that have as goal the reconfiguration of the instruction set of the processor, involving issues such formats, operands and operations of the instructions.

This work possess as main objective the development of a RISP processor, combining the techniques of configuration of the set of executed instructions of the processor during the development, and reconfiguration of itself in execution time. The project and implementation in VHDL of this RISP processor has as intention to prove the applicability and the efficiency of two concepts: to use more than one set of fixed instructions, with only one set active in a given time, and the possibility to create and combine new instructions, in a way that the processor pass to recognize and use them in real time as if these existed in the fixed set of instruction.

The creation and combination of instructions is made through a reconfiguration unit, incorporated to the processor. This unit allows the user to send custom instructions to the processor, so that later he can use them as if they were fixed instructions of the processor.

In this work can also be found simulations of applications involving fixed and custom instructions and results of the comparisons between these applications in relation to the consumption of power and the time of execution, which confirm the attainment of the goals for which the processor was developed.

Keywords: Reconfigurable Processor, RISP, VHDL, instruction set.

Lista de Figuras

1 Acoplamento através do barramento de E/S . . . . . p. 10

2 Acoplamento através de um coprocessador . . . . . p. 10

3 Acoplamento através de uma RFU . . . . . p. 10

4 Codificação de instruções por meio do CodOp . . . . . p. 11

5 Codificação de instruções mediante um identificador . . . . . p. 12

6 Codificação *hardwired* . . . . . p. 13

7 Codificação fixa . . . . . p. 13

8 Codificação flexível . . . . . p. 14

9 Caminho de dados do PRISC . . . . . p. 16

10 Formato de instrução do PRISC . . . . . p. 17

11 Arquitetura do Chimaera . . . . . p. 19

12 Bloco Lógico do Chimaera . . . . . p. 20

13 Arquitetura do GARP . . . . . p. 20

14 Arquitetura da RFU do OneChip . . . . . p. 21

15 Processador CRISP . . . . . p. 23

16 Processador XiRISC . . . . . p. 23

17 Formato de instrução do VISC . . . . . p. 24

18 Sinais OCP . . . . . p. 25

19 Temporização OCP envolvendo operações de escrita e leitura . . . . . p. 27

20 Temporização OCP com controle de fluxo . . . . . p. 29

21 Temporização OCP com latência de leitura . . . . . p. 30

22 Unidades constituintes do processador . . . . . p. 33

23 Unidade Operativa do processador . . . . p. 34

24 Representação das unidades adaptadas ao padrão OCP . . . . p. 36

25 Conjunto de estados do CodOp 05H . . . . p. 38

26 Conjunto de estados da instrução LODD . . . . p. 39

27 Diagrama de estados do processo de execução de instruções fixas . . . . p. 42

28 Implementação da instrução LOCO . . . . p. 43

29 Unidade de reconfiguração . . . . p. 44

30 Diagrama de estados do processo de reconfiguração . . . . p. 45

31 Exemplo de funcionamento da *Lpm_ram* de verificação. (a) nova instrução. (b) continuação da instrução. (c) término da instrução. (d) erro de reconfiguração. . . . . p. 47

32 Exemplo de funcionamento da tabela de endereços para um novo CodOp. . . p. 49

33 Exemplo de funcionamento da tabela de endereços para um mesmo CodOp. . p. 50

34 Processo de reconfiguração de um novo CodOp . . . . p. 51

35 Continuação do processo de reconfiguração de um CodOp . . . . p. 52

36 Diagrama de estados do processo de execução de instruções reconfiguradas . p. 53

37 Processo de reconfiguração de um novo CodOp . . . . p. 54

38 Continuação do processo de reconfiguração de um CodOp . . . . p. 55

39 Disposição dos bits da palavra de reconfiguração . . . . p. 55

40 Visão geral da simulação de alternância de conjuntos de instruções . . . . . . p. 62

41 Simulação da instrução muda_modo . . . . p. 62

42 Simulação da instrução LOCO 1AH . . . . p. 63

43 Simulação da instrução IR = 0D084200H do conjunto 1 . . . . p. 64

44 Simulação da alteração do conjunto de instrução 1 para 2 . . . . p. 64

45 Simulação da instrução IR = 0D084200H do conjunto 2 . . . . p. 65

46 Simulação da alteração do conjunto de instrução 2 para 3 . . . . p. 65

47 Simulação da instrução IR = 0D084200H do conjunto 3 (a) . . . . p. 66

48 Simulação da instrução IR = 0D084200H do conjunto 3 (b) . . . . p. 66

49 Diagrama de estados do CodOp . . . . p. 67

50 Simulação do processo de reconfiguração (b) . . . . p. 68

51 Simulação do processo de reconfiguração (a) . . . . p. 68

52 Simulação do processo de obtenção de valores . . . . p. 69

53 Simulação do processo de execução da instrução reconfigurada . . . . p. 69

54 Simulação do processo de execução das instruções MAC e LDR . . . . p. 70

55 Visão geral da simulação do Fatorial utilizando instruções fixas . . . . p. 73

56 Execução das operações durante a simulação do Fatorial . . . . p. 73

57 Término da execução do Fatorial . . . . p. 74

58 Visão geral da simulação do Fatorial utilizando instruções INC R e REC R . . p. 75

59 Execução das instruções INC R e REC R . . . . p. 76

60 Repetição do processo de fatorial sem reconfiguração de INC e DEC . . . . . p. 76

61 Repetição do processo de fatorial sem reconfiguração de FAT . . . . p. 78

Lista de Tabelas

1 Codificação dos comandos MCmd . . . . p. 26

2 Codificação dos comandos SResp . . . . p. 26

3 Conjuntos de instruções do processador . . . . p. 41

4 Composição da palavra de configuração . . . . p. 56

5 Valores obtidos na compilação do processador . . . . p. 60

6 Valores obtidos para cada conjunto de instrução . . . . p. 61

7 Execução do fatorial utilizando apenas instruções fixas . . . . p. 72

8 Execução do fatorial utilizando instruções reconfiguradas . . . . p. 75

9 Informações adicionais das instruções reconfiguradas . . . . p. 75

10 Execução do fatorial utilizando instruções reconfiguradas com mais de uma operação . . . . p. 77

11 Informações adicionais das instruções reconfiguradas . . . . p. 78

Lista de Siglas e Abreviaturas

FPGA (*Field Programmable Gate Array*), p. 2

RTR (*Run-Time Reconfiguration*), p. 2

SRAM (*Static Random Access Memory*), p. 3

RISP (*Reconfigurable Instruction Set Processors*), p. 4

OCP (*Open Core Protocol*), p. 6

ASICs (*Aplication Specific Integrated Circuits*), p. 7

ASIPs (*Specific Instruction Set Processors*), p. 7

RPU (*Reconfigurable Processing Unit*), p. 9

PCI (*Peripheral Component Interconnect*), p. 9

RFU (*Reconfigurable Functional Unit*), p. 10

CodOp (Código da Operação), p. 11

LUT (*Lookup Tables*), p. 14

ULA (Unidade Lógica Aritmética), p. 15

NOC - *Network On Chip*), p. 15

PRISM (*Processor Reconfigurable through Instruction-Set Metamorphosis*), p. 16

PRISC (*PRogrammable Instruction Set Computer*), p. 16

RISC (*Reduced Instruction Set Computer*), p. 16

PFUs (*Programmable Functional Units*), p. 16

SoC (*System-on-a-Chip*), p. 17

DISC (*Dynamic Instruction Set Computer*), p. 17

RCP (*Reconfigurable Co-Processor*), p. 18

(PE - *Processing Elements*), p. 19

CLBs (*Configurable Logic Blocks*), p. 20

RBT (*Reconfigurable Bits Table*, p. 21

VLIW (*Very Long Instruction Word*), p. 21

FFUs (*Fixed Functional Units*), p. 21

XiRISC (*eXented Instruction Set RISC*), p. 22

DSP (*Digital Signal Processing*), p. 22

VISC (*Variable Instruction Set Communications Architecture*), p. 24

FSM (*Finite State Machine*), p. 33

Ac (Acumulador), p. 35

SP (*Stack Pointer* - Apontador de Pilha), p. 35

PC (*Counter Program* - Contador de Programa), p. 45

VHDL (VHSIC *Hardware Description Language*), p. 59

Sumário

1 INTRODUÇÃO p. 1

1.1 Contextualização . . . . p. 1

1.2 Definições . . . . p. 3

1.3 Objetivos . . . . p. 4

1.4 Dificuldades . . . . p. 5

1.5 Organização . . . . p. 5

2 REVISÃO BIBLIOGRÁFICA p. 6

2.1 A integração entre *Hardware* e *Software* . . . . p. 6

2.2 ASIC, ASIP e RISP . . . . p. 7

2.3 Desenvolvimento de um RISP . . . . p. 9

2.3.1 Acoplamento ou Integração . . . . p. 9

2.3.2 Codificação de instruções . . . . p. 11

2.3.3 Operandos . . . . p. 12

2.3.3.1 *Hardwired* . . . . p. 12

2.3.3.2 Fixo . . . . p. 12

2.3.3.3 Flexível . . . . p. 13

2.4 Desenvolvimento da lógica reconfigurável . . . . p. 13

2.4.1 Granularidade . . . . p. 14

2.4.2 Interconexão . . . . p. 15

2.4.3 Reconfigurabilidade . . . . p. 15

2.5 Trabalhos Relacionados . . . . p. 16

2.6 O Padrão OCP . . . . p. 24
2.6.1 Sinais Básicos do OCP . . . . p. 25
2.6.2 Temporização . . . . p. 27

3 ARQUITETURA PROPOSTA p. 32
3.1 Visão Geral . . . . p. 32
3.2 Unidade Operativa . . . . p. 34
3.3 Modos de funcionamento . . . . p. 35
3.4 Utilização do padrão OCP . . . . p. 36
3.5 Unidade de Controle . . . . p. 37
3.6 Instruções . . . . p. 40
3.7 Processo de Execução de Instruções Fixas . . . . p. 42
3.8 Unidade de Reconfiguração . . . . p. 44
3.8.1 Processo de reconfiguração de uma instrução . . . . p. 45
3.8.2 Processo de Execução de Instruções Reconfiguradas . . . . p. 52
3.9 Reconfiguração de uma instrução . . . . p. 55

4 SIMULAÇÕES E RESULTADOS p. 59
4.1 Resultados da Compilação . . . . p. 59
4.1.1 Análise de custo dos conjuntos de instruções . . . . p. 60
4.2 Simulações . . . . p. 61
4.2.1 Simulação da alternância entre os conjuntos de instruções . . . . p. 61
4.2.2 Simulação da criação de uma instrução: MAC D, A, B, C . . . . p. 67
4.2.3 Simulação de uma aplicação: Fatorial . . . . p. 70

5 Considerações finais p. 79

Referências p. 82

Apêndice A – Esquema completo do processador p. 85

Apêndice B – Conjunto de registradores p. 88

Apêndice C – Fluxograma da máquina de estados do processador p. 89

Apêndice D – Conjuntos de Instruções p. 94

D.1 Conjunto 1 . . . . p. 94

D.2 Conjunto 2 . . . . p. 96

D.3 Conjunto 3 . . . . p. 98

Apêndice E – Microinstruções dos conjuntos de Instruções Fixos p. 100

E.1 Conjunto de instrução 1 . . . . p. 100

E.2 Conjunto de instrução 2 . . . . p. 103

E.3 Conjunto de instrução 3 . . . . p. 105

Apêndice F – Codificação dos formatos de operações p. 108

1 INTRODUÇÃO

Este Capítulo define o escopo do trabalho e o contextualiza, bem como introduz algumas definições sobre computação reconfigurável, com ênfase para os processadores reconfiguráveis. Estes conceitos são necessários para um melhor entendimento acerca desta dissertação.

1.1 Contextualização

Um dos principais usos da computação consiste na resolução de problemas complexos existentes nas áreas de engenharia e ciências exatas, geralmente relacionados a fatores como desempenho, tempo de resposta, eficiência, disponibilidade, tolerância à falhas, entre outros. Existem diversas pesquisas para que tais soluções consumam cada vez menos recursos computacionais necessários para armazenamento, recuperação, transmissão e processamento de informações e para que estas sejam realizadas em um menor período de tempo possível.

[Martins et al., 2003] afirma que as soluções para estes problemas podem ser classificadas basicamente em dois paradigmas, que são as soluções implementadas em ***Hardware*** **Fixo** (HF) ou *Hardwired* e as soluções implementadas em ***Hardware*** **Programável através de** ***Software*** (HP+SW). As principais deficiências do paradigma de *hardware* fixo são a falta de flexibilidade e sua sub-utilização quando da execução de aplicações diversas. Além disso, os componentes de *hardware* utilizados neste paradigma não podem ser alterados após a fabricação, não sendo assim possível realizar adaptações durante a execução de tarefas computacionais.

Já no paradigma de *Hardware* Programável através de *Software*, a principal deficiência está relacionada com o desempenho durante a execução de aplicações, que é muitas vezes insatisfatório. Apesar da grande flexibilidade para a execução de qualquer tipo de trabalho que envolva processamento computacional, a característica genérica destes sistemas faz com que as soluções implementadas apresentem desempenho inferiores ao ótimo.

As desvantagens relacionadas com desempenho, flexibilidade, custo, entre outros, associados a cada um desses tipos de abordagens de implementação de *hardware* são os principais

problemas que motivaram o desenvolvimento de novos modelos e tipos de implementação de soluções computacionais.

A **Computação Reconfigurável** é uma solução intermediária entre as soluções em dispositivos dedicados e de propósito geral. De acordo com [Martins et al., 2003], entre os motivos para o seu estudo, constam a demanda computacional de algumas importantes aplicações atuais e futuras, a inadequação de alguns modelos e estilos de computação atual em termos de desempenho e flexibilidade, o interesse das principais empresas e universidades do mundo, e também a evolução dos dispositivos computacionais em termos de arquitetura e tecnologia.

Em termos básicos, a Computação Reconfigurável combina a velocidade do *hardware* com a flexibilidade do *software*. [Adário, 1997] cita que a tecnologia da computação reconfigurável consiste na habilidade de se modificar o *hardware* da arquitetura do sistema para se adequar à aplicação; e isto pode ou não ser realizado com o sistema em funcionamento.

A Computação Reconfigurável, segundo [Dehon, 1996], ainda é uma área em desenvolvimento, diferente de arquiteturas de computadores onde os conceitos e mecanismos já são alvo de estudos a muito tempo. Apesar dos conceitos básicos de computação reconfigurável terem sido propostos na década de 60, as primeiras demonstrações só vieram ocorrer em meados da década de 90. A partir desta década, de acordo com [Mesquita, 2002], houveram avanços significativos no campo da computação reconfigurável utilizando-se *FPGAs* (*Field Programmable Gate Arrays*) [1] como base para sistemas reprogramáveis de alto desempenho. Muitos desses sistemas alcançaram altos níveis de desempenho e resultados satisfatórios na aplicação de determinadas tarefas. No entanto, os FPGAs são tipicamente configurados antes de iniciarem a execução da aplicação, ou seja, apesar de serem classificados como dispositivos reconfiguráveis, estes são reconfigurados apenas em tempo de desenvolvimento. Isto fez com que os estudos neste campo continuassem sendo amplamente explorados em busca de novas soluções, como por exemplo a **Reconfiguração Dinâmica**.

A reconfiguração dinâmica, também chamada de *Run-Time Reconfiguration*(RTR), *on-the-fly reconfiguration* ou *in-circuit reconfiguration*, consiste no processo de reconfiguração de um dispositivo em em tempo de execução, em que não há necessidade de reiniciar o circuito ou remover elementos reconfiguráveis para realizar a programação do dispositivo.

Uma arquitetura reconfigurável possui como principais metas: acréscimo de velocidade e capacidade de prototipagem, representadas pelo número de portas lógicas equivalentes, proporcionando aumento de desempenho; reconfiguração dinâmica e/ou parcial; baixo consumo na

[1] FPGAs são dispositivos que consistem de uma matriz de blocos lógicos, formada por blocos de entrada e saída, e conectada por fios de interconexão, onde todos esses ítens são configuráveis

reconfiguração; dispositivos livres de falhas; granularidade adequada ao problema, entre outros. O uso de arquiteturas reconfiguráveis para incrementar o desempenho de sistemas é uma tecnologia muito utilizada, principalmente pela possibilidade de implementação de certos algoritmos lentos nos atuais processadores diretamente no *hardware*.

Esta dissertação se insere neste contexto de busca pelo desenvolvimento de arquiteturas reconfiguráveis mais adequadas ao uso de sistemas computacionais, particularmente visando um acréscimo de desempenho na solução de tarefas computacionais.

1.2 Definições

O termo **Sistemas de Computação Reconfigurável** geralmente se refere a habilidade de reconfigurar um FPGA baseado na tecnologia *SRAM* (*Static Random Access Memory*).De acordo com [Goncalves et al., 2003], esta tecnologia propicia *chips* extremamente eficientes e totalmente reaproveitáveis, pois estes podem ser reprogramados e reutilizados quantas vezes forem necessárias.

Os Sistemas de Computação Reconfigurável são plataformas de *hardware* cujas arquiteturas podem ser modificadas por *software*, em tempo de execução, para melhor adequar-se à aplicação que está sendo executada naquele instante. Deste modo, o processador reconfigurável passa a trabalhar com uma arquitetura desenvolvida exclusivamente para aquele determinado tipo de aplicação, permitindo uma eficiência muito maior do que a normalmente encontrada em processadores de uso geral. Isto ocorre porque o *hardware* é otimizado para executar os algoritmos necessários para uma aplicação específica.

Apesar do nome, as arquiteturas reconfiguráveis podem ser híbridas, além de puramente reconfiguráveis. Uma arquitetura reconfigurável híbrida consiste em dispositivos de *hardware* fixo e *hardware* programável em conjunto com um *hardware* reconfigurável. Também é possível a reconfiguração através de *software*, onde os objetos definidos como *hardware* podem ser redefinidos como blocos ou módulos construtivos, como mostra [Barat and Lauwereins, 2000].

O *hardware* reconfigurável apresenta-se como uma alternativa tecnológica que pode adaptar-se a aplicação com a facilidade de um processador de propósito geral, enquanto mantém as vantagens de desempenho do *hardware* dedicado. De acordo com [Aragão et al., 2000], a computação reconfigurável tem um grande potencial a ser explorado especialmente em aplicações que necessitam de alto desempenho como, por exemplo, processamento de imagens e aplicações de tempo real. Segundo [Silva, 2001], as arquiteturas reconfiguráveis são atualmente utilizadas em diversas áreas de aplicação, propiciando resultados satisfatórios em aplicações multimídia,

Processamento Digital de Sinais, Telecomunicações, Computação Móvel, Criptografia entre outros.

[Barat and Lauwereins, 2000] mostra que, dentre os vários segmentos em relação ao *hardware* reconfigurável, destacam-se os **Processadores Reconfiguráveis**. Estes processadores combinam as funções de um microprocessador com uma lógica reconfigurável e podem ser adaptados depois do processo de desenvolvimento, do mesmo modo que os processadores programáveis podem se adaptar a mudanças na aplicação.

[Lodi et al., 2003] define os **Processadores com Conjunto de Instruções Reconfiguráveis** (*RISP- Reconfigurable Instruction Set Processors*) como um subconjunto dos processadores reconfiguráveis. Os processadores RISP consistem em um núcleo do microprocessador que se estende à lógica reconfigurável. A partir da década de 90, as pesquisas acerca dos processadores RISP se intensificaram, propiciando o surgimento de diversos processadores desenvolvidos para o aprimoramento dos conceitos envolvendo sua utilização, e o surgimento de novas tendências, como a utilização dos RISPs destinados a aplicações em geral.

1.3 Objetivos

O presente trabalho tem como intuito estudar e propor uma arquitetura de um microprocessador que possibilite a criação e reconfiguração de novos padrões de instruções. Nesta dissertação foi desenvolvido um processador RISP, combinando as técnicas de configuração do conjunto de instruções do processador executadas em tempo de desenvolvimento, e de reconfiguração do mesmo em tempo de execução, de uma forma simples e eficiente.

Para tanto, este processador apresenta a incorporação de uma unidade reconfigurável destinada a reconfiguração de novas instruções. Com o objetivo de possibilitar a avaliação do impacto da unidade reconfigurável, o processador desenvolvido inclui uma unidade de controle onde três conjuntos de instruções distintos foram implementados. A escolha da utilização de cada um destes conjuntos pode ser realizada através de instruções privilegiadas, como será visto posteriormente.

O processador proposto neste trabalho permite, então, explorar a relação entre a complexidade da reconfiguração dinâmica de instruções e o desempenho no uso de um conjunto de instrução especifico *hardwired*.

Também é objetivo desta dissertação contribuir com a linha de pesquisa em Sistemas Integrados e Distribuídos, acumulando conhecimentos sobre o projeto e implementação de micro-

processadores.

Este processador é apresentado nesta dissertação como uma **Prova de conceito**, ou seja, pretende-se provar, através de evidências como simulações e testes, que o modelo de arquitetura proposto pode ser realmente utilizado na prática. Como nem sempre é possível determinar se projeções teóricas do conceito de uma arquitetura podem ser aplicadas na prática, torna-se essencial desenvolver um modelo de prova de conceito que possa executar a função desejada em um ambiente real. Aplicando este tipo de desenvolvimento, pode-se portanto verificar com antecedência se o conceito original possui falhas ou funcionará como esperado.

1.4 Dificuldades

O modelo de arquitetura e a prova de conceito seriam melhor explorados com a disponibilidade de um compilador. Um compilador para a linguagem TIGER está sendo desenvolvido por outro pesquisador do grupo de pesquisa em Arquiteturas e Sistemas Integrados da UFRN/CCET/DIMAp. Entretanto, até o momento em que esta dissertação foi finalizada, este compilador ainda não se encontrava concluído, portanto não houve possibilidade de se realizarem testes de maior porte, sendo estes testes uma das prioridades do grupo de pesquisa para trabalhos futuros.

1.5 Organização

Este trabalho está estruturado da seguinte forma: o capítulo 2 introduz alguns conceitos teóricos fundamentais com relação aos processadores reconfiguráveis, dando ênfase aos processadores RISP e aos trabalhos anteriormente publicados. O capítulo 3 apresenta o processador proposto nesta dissertação, no que diz respeito à sua organização, estrutura e características principais. O capítulo 4 mostra a implementação do processador, onde são apresentados resultados de simulações obtidas das unidades funcionais implementadas. Por último, o capítulo 5 apresenta as considerações finais acerca deste trabalho.

2 *REVISÃO BIBLIOGRÁFICA*

Este Capítulo aborda alguns aspectos do estado-da-arte da computação reconfigurável, em particular os processadores RISP. Serão extraídas informações quanto a concepção e implementação destes processadores. O capítulo também inclui uma breve análise de algumas arquiteturas da literatura que serviram como base para a criação do processador proposto nesta dissertação. Este Capítulo traz também uma explicação sobre o padrão de comunicação *OCP* (*Open Core Protocol*) [Partnerchip, 2001], utilizado para a interface entre o processador e o mundo externo.

2.1 A integração entre *Hardware* e *Software*

Os Sistemas Integrados hoje são compostos de muitos componentes de *hardware* e *software* que interagem entre si, de modo que o equilíbrio entre estes componentes determina o sucesso do sistema. [Lodi et al., 2003] diz que, devido à natureza do *software*, seus componentes são mais fáceis de serem modificados do que os de *hardware*.

Os componentes de *software*, em razão de sua flexibilidade, oferecem uma maneira mais fácil de corrigir erros, mudar a aplicação, reutilizar componentes e reduzir o período entre o desenvolvimento do produto e sua chegada ao mercado (*time-to-market*). Entretanto, quando comparados às soluções em *hardware* puro, os componentes de *software* são mais lentos e consomem mais energia. Os componentes de *hardware* são utilizados quando a velocidade e/ou o consumo de energia são críticos. Por outro lado, os componentes de *hardware* requerem um processo de desenvolvimento longo e caro.

De acordo com [Martins et al., 2003], uma arquitetura reconfigurável pode compartilhar todas as características de uma arquitetura tradicional. No entanto, organiza e implementa a computação de maneira diferente. Ao invés de processar uma função por meio de um conjunto de instruções executadas seqüencialmente ao longo do tempo, assim como em um processador, as arquiteturas reconfiguráveis geralmente processam a função por meio de unidades lógicas mapeadas em diferentes blocos construtivos básicos disponíveis dentro dos dispositivos recon-

figuráveis, como os FPGAs. Assim, as arquiteturas reconfiguráveis substituem a computação temporal pela computação espacial, o que possibilita ganho de desempenho e menor consumo de energia. [Compton and Hauck, 2000] cita que as arquiteturas reconfiguráveis permitem a reconfiguração dos blocos construtivos, tanto na sua lógica como em sua funcionalidade interna, assim como a reconfiguração dos blocos de interconexão, responsáveis pela interligação destes blocos construtivos. Os blocos construtivos normalmente implementam ou são as próprias unidades de processamento, armazenamento, comunicação e entrada/saída de dados.

Do exposto acima, conclui-se pela vantagem teórica da utilização das arquiteturas reconfiguráveis. Esta dissertação explora este espaço de projeto de sistemas computacionais em *hardware*, propondo o desenvolvimento de um processador com conjunto de instrução reconfigurável.

2.2 ASIC, ASIP e RISP

[Smith, 1997] define os *ASICs* (*Aplication Specific Integrated Circuits*), também chamados de *hardware* totalmente customizados, como processadores projetados para executarem uma determinada operação específica, o que os tornam rápidos e eficientes quando estão executando a operação para a qual eles foram desenvolvidos. Contudo, após fabricado, o ASIC não pode ser alterado. Caso seja necessário modificá-lo, o ASIC deve ser desenvolvido e fabricado novamente.

Os microprocessadores, Por outro lado, são dispositivos flexíveis que podem ser programados para desempenhar qualquer tarefa computacional. Entretanto, estes possuem menor desempenho quando comparados com um ASIC por causa do tempo gasto na busca e decodificação de instruções e da sua não-especialização.

Os Processadores com Conjunto de Instruções para Aplicações Específicas (*ASIPs - Specific Instruction Set Processors*) são processadores cujo conjunto de instruções inclui instruções dedicadas à aplicação, ou grupo de aplicações, para as quais foi desenvolvido. Na literatura é possível encontrar ferramentas que propõem a geração de uma descrição sintetizável de um ASIP, como em [It and Carro, 2000]. Estas descrições podem servir como entrada para um processo de síntese, realizado ou não pela própria ferramenta, dando origem a um arquivo de configuração de um FPGA. Apesar da flexibilidade aparente deste processo e, não obstante ao aumento de desempenho obtido quando comparado com processadores de propósito geral, os ASIPs assim obtidos apresentam algumas restrições quanto a desempenho e flexibilidade. Em [Ito et al., 2001] o conjunto de instruções é um simples subconjunto dos "*byte codes*"java, por-

tanto não necessariamente melhor adaptados à aplicação alvo. Quanto a flexibilidade do ASIP, de um modo geral, apresentam melhor adequação apenas para a aplicação, ou grupo de aplicações, para as quais foi desenvolvido. Quando a aplicação muda, a adequação desaparece e o desempenho sofre uma degradação.

Em termos de desempenho, os processadores ASICs e ASIPs apresentam vantagens em relação aos outros sistemas computacionais, como os microprocessadores e microcontroladores, em relação à execução das aplicações para as quais foram originalmente desenvolvidas. Entretanto, esta vantagem se torna deficiente quando da execução de outras aplicações. Algumas vezes a utilização destes processadores para outras aplicações torna-se impraticável. Além disso, [Silva, 2001] afirma que estas arquiteturas não são adaptáveis do ponto de vista da capacidade de reorganização estrutural e criação de novas funções em *hardware*.

Os processadores reconfiguráveis, como por exemplo os RISPs, em detrimento das demais arquiteturas tais quais ASICs e ASIPs, não possuem uma estrutura adaptada a uma finalidade específica. Em outras palavras, os recursos internos e a estrutura dos dispositivos reconfiguráveis encontram-se incompletos até que sejam mapeados em *hardware* mediante uma configuração. Uma possibilidade seria o mapeamento em *hardware* do conjunto de instruções do processador, que pode ser modificado de acordo com as necessidades da aplicação vigente.

[Barat et al., 2002b] mostra que a diferença de um processador RISP para um ASIC, em relação à execução de instruções, está no fato de que existe um conjunto de instruções fixo implementado por meio de um *hardware* fixo e um conjunto de instruções reconfigurável implementado sob uma lógica reconfigurável, que pode sofrer alterações durante o tempo de execução. As unidades funcionais reconfiguráveis oferecem a adaptação do processador à aplicação, enquanto o núcleo do processador oferece a programabilidade do *software*. O mesmo autor também afirma que o custo do projeto do núcleo de um microprocessador é reduzido mediante a reutilização do mesmo para várias aplicações diferentes. Nos ASIPs, o custo para cada nova geração vem da reformulação do projeto das unidades funcionais especializadas, o que pode ser relativamente alto. Técnicas de prototipagem rápida são essenciais para reduzir o custo e o tempo necessário para esta reformulação do projeto. Um RISP pode ser usado como um protótipo para uma família de processadores ASIP que compartilham o mesmo núcleo ou conjunto de instruções fixas.

2.3 Desenvolvimento de um RISP

De acordo com [Barat and Lauwereins, 2000], o desenvolvimento de um processador reconfigurável, em particular um RISP, deve ser realizado a partir de duas características principais. A primeira é a interface entre o microprocessador e a lógica reconfigurável, que envolve os aspectos relacionados sobre de que modo os dados são transferidos do processador para a lógica reconfigurável e vice-versa, assim como a sincronização entre estes. A segunda é o desenvolvimento da lógica reconfigurável, onde estão envolvidas questões referentes a granularidade, reconfigurabilidade, interconexão, entre outros.

Nas sub-seções seguintes serão vistas as diferentes possibilidades de implementação existentes para cada uma das características citadas acima. Tais informações foram retiradas da literatura. Na sessão 2.5, alguns trabalhos relacionados encontrados na literatura serão analisados.

2.3.1 Acoplamento ou Integração

Um processador reconfigurável consiste de um microprocessador acoplado a uma lógica reconfigurável. A posição da lógica reconfigurável, denominada Unidade de Processamento Reconfigurável (*RPU - Reconfigurable Processing Unit*), em relação ao microprocessador afeta o desempenho do sistema e o tipo de aplicação que utilizará este *hardware* reconfigurável. [Carrillo and Chow, 2001] diz que o benefício obtido da execução do código na lógica reconfigurável depende de dois aspectos, que são o tempo de comunicação e o tempo de execução. O tempo necessário para executar uma operação é a soma do tempo necessário para transferir os dados a serem processados e o tempo necessário para processá-lo. Se este tempo total for menor do que o tempo obtido pelo processador sem a lógica reconfigurável, então confirma-se o benefício.

O *hardware* reconfigurável pode ser acoplado ao microprocessador por três maneiras distintas: a primeira destas consiste em anexar a lógica reconfigurável em algum barramento de Entrada e Saída, por exemplo em um barramento *PCI* (Peripheral Component Interconnect), como ilustrado na Figura 1.

A segunda maneira é por meio de um coprocessador, em que a lógica reconfigurável é inserida próxima ao microprocessador. Deste modo, a comunicação entre as unidades é realizada utilizando-se protocolos de comunicação similares aos utilizados em coprocessadores de ponto flutuante. A Figura 2 mostra este modo de acoplamento.

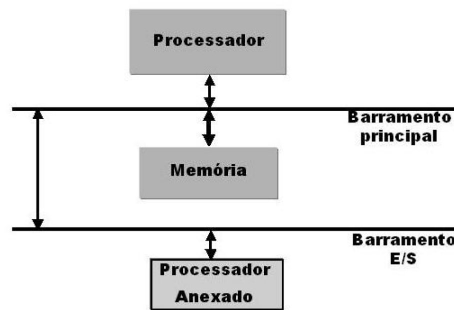


Figura 1: Acoplamento através do barramento de E/S

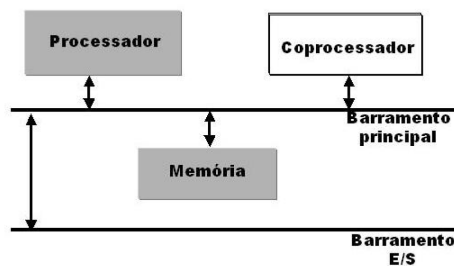


Figura 2: Acoplamento através de um coprocessador

A terceira maneira de acoplamento consiste na utilização de uma ou mais Unidades Funcionais Reconfiguráveis $(RFU - Reconfigurable\ Functional\ Unit)$, em que a lógica reconfigurável é inserida no microprocessador, como exemplificado na Figura 3. Os processadores RISP seguem este tipo de acoplamento.

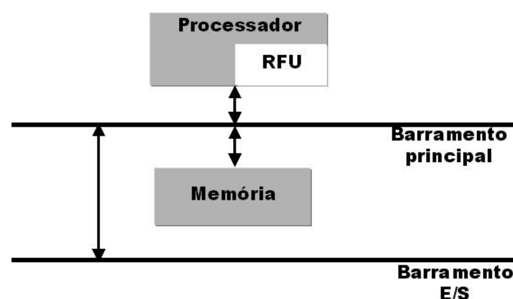


Figura 3: Acoplamento através de uma RFU

Os dois primeiros modos de acoplamento são conhecidos como fracamente acoplados. A característica principal destes modos de acoplamento se deve ao fato de que pode existir um aumento de desempenho em virtude da utilização da lógica reconfigurável, mas este aumento é penalizado devido ao alto custo adicional durante a transferência de dados. [Barat et al., 2002a]

cita como vantagens dessa forma de acoplamento a facilidade no projeto de sistemas, devido a utilização de componentes-padrão como FPGAs; o fato de poder utilizar uma maior porção do *hardware* reconfigurável, pelo motivo do *hardware* não estar limitado ao espaço disponível no processador; e a possibilidade do microprocessador e lógica reconfigurável poderem realizar diferentes tarefas ao mesmo tempo em virtude de serem dois dispositivos independentes.

Já a integração de uma unidade funcional recebe a denominação de fortemente acoplado. Utilizando-se este tipo de acoplamento, o custo de comunicação entre o microprocessador e a lógica reconfigurável é praticamente nulo pelo fato de se tornarem um único dispositivo, conseqüentemente acarretando um aumento na velocidade, tornando possível a aceleração de diversas aplicações.

Como desvantagens, podem-se citar um maior custo de desenvolvimento, por não ser possível utilizar componentes-padrão, e o fato do tamanho do *hardware* reconfigurável ser limitado ao tamanho do processador.

2.3.2 Codificação de instruções

As instruções reconfiguráveis geralmente são identificadas por um *CodOp* (Código da Operação) especial, na forma de um campo extra contido na instrução, o qual pode ser implementado de duas formas:

A primeira forma consiste em especificar no CodOp um endereço de memória com a palavra de reconfiguração para uma instrução. Esta forma de codificação causa um aumento no número de bits do tamanho da palavra de instrução. A Figura 4 ilustra este caso.

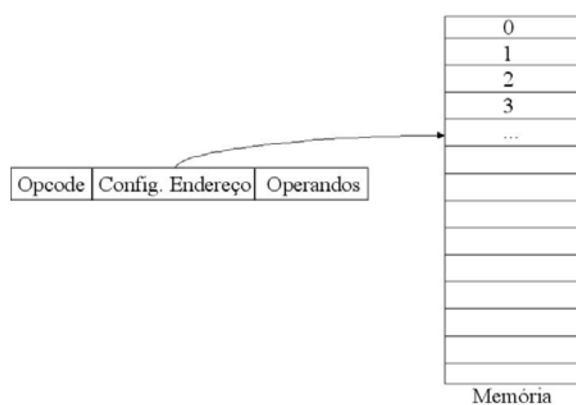


Figura 4: Codificação de instruções por meio do CodOp

A segunda forma é a utilização de um identificador, responsável por indexar uma tabela de configuração onde são armazenadas informações, como por exemplo a palavra de reconfigura-

ção. Esta configuração propicia um menor número de bits no tamanho da palavra de instrução, mas o número de instruções distintas passa a ser limitado pelo tamanho da tabela de configuração, como pode ser visto na Figura 5.

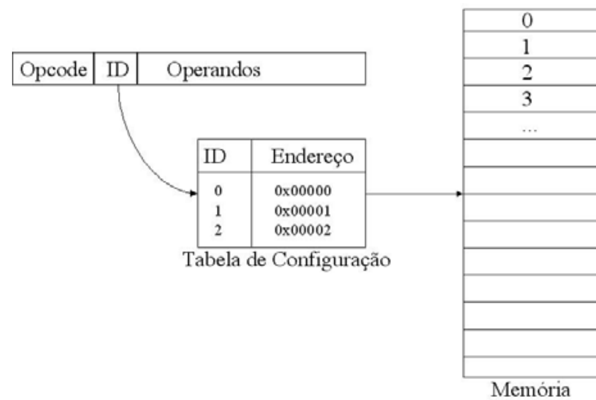


Figura 5: Codificação de instruções mediante um identificador

2.3.3 Operandos

A palavra de instrução especifica os operandos (valores imediatos, endereços, identificadores, etc) que são enviados para a RFU, codificados em um dos formatos vistos a seguir:

2.3.3.1 *Hardwired*

O conteúdo de todos os registradores é enviado à RFU, sendo que os registradores utilizados são definidos de acordo com a instrução configurada na RFU. O resultado obtido é roteado para o registrador indicado na palavra de instrução, fazendo com que a RFU acesse uma maior quantidade de registradores. A desvantagem deste formato deve-se a dificuldade na geração do código necessário para efetuar tal roteamento e para definição de quais registradores serão utilizados. A Figura 6 ilustra este esquema.

2.3.3.2 Fixo

Dois operandos, selecionados por um banco de registradores, estão em posições fixas na palavra de instrução e possuem tipos fixos. Este formato necessita de mais bits de instrução, mas permite maior flexibilidade para especificar operandos. A Figura 7 mostra o esquema de codificação de operandos fixos.

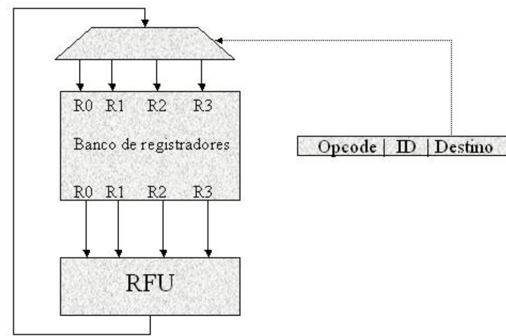


Figura 6: Codificação *hardwired*

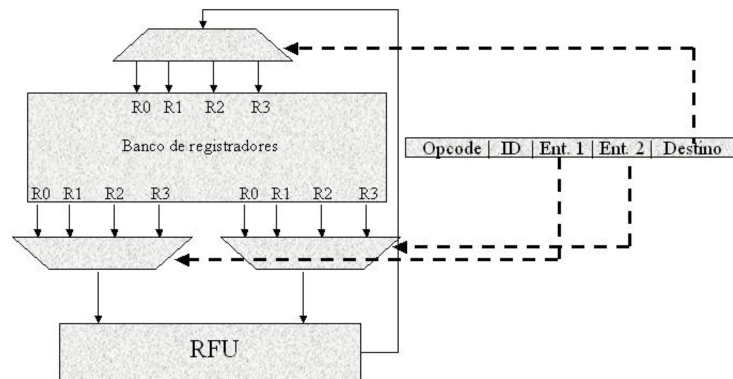


Figura 7: Codificação fixa

2.3.3.3 Flexível

A posição dos operandos é configurável. Se uma tabela de configuração for utilizada, esta pode especificar a decodificação dos operandos. Neste caso, um operando é selecionado a partir do banco de registradores e o outro selecionado ou do banco de registradores ou tratado como uma constante incluída na palavra de instrução, dependendo da tabela de configuração. A Figura 8 ilustra este esquema.

2.4 Desenvolvimento da lógica reconfigurável

As Arquiteturas reconfiguráveis podem ser classificadas segundo muitos critérios. Entre os mais utilizados, são considerados a granularidade de suas unidades reconfiguráveis, a interconexão entre segmentos e o momento da reconfiguração.

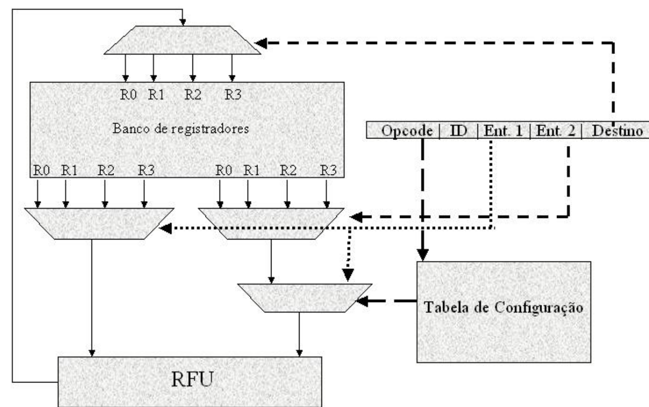


Figura 8: Codificação flexível

2.4.1 Granularidade

A granularidade é um termo originalmente usado por [Hartenstein, 2001] para designar a complexidade e a largura das palavras das unidades de reconfiguração. Quanto à granularidade, as arquiteturas reconfiguráveis podem ser classificadas como:

- Arquiteturas Reconfiguráveis de Granularidade Fina: apresentam menor quantidade de lógica ou poder computacional nas unidades de reconfiguração. [Hartenstein, 2001] afirma que estas arquiteturas geralmente oferecem maior flexibilidade de implementação de algoritmos em *hardware*, impondo, entretanto um custo de roteamento bem mais elevado. Essas características possibilitam uma maior aproximação da arquitetura pós-configuração das características do algoritmo implementado, geralmente resultando em um melhor desempenho. Além do roteamento, esse tipo de granularidade apresenta outras desvantagens, dentre elas, alto consumo de energia durante a reconfiguração e maiores atrasos de propagação dos sinais, como afirma [Dehon, 1996]. A granularidade fina utiliza portas como blocos básicos de construção, implementados utilizando blocos funcionais compostos por tabelas $(Lookup\ Tables - LUT)$, flip-flops e multiplexadores. As aplicações em que este tipo de granularidade são recomendadas são as que envolvem operações de manipulação de bits.

- Arquiteturas Reconfiguráveis de Granularidade Grossa: apresentam maior quantidade de lógica ou poder computacional nas unidades de reconfiguração. Estas arquiteturas são utilizadas para o mapeamento e execução de aplicações que necessitam de caminho de dados com largura de uma palavra. Suas unidades de reconfiguração são otimizadas para maior poder computacional, apresentando melhor desempenho em algoritmos com tais

características. Na granularidade grossa, os blocos de construção são maiores, tipicamente *ULAs*, multiplicadores, deslocadores, entre outros. Geralmente são utilizados em aplicações que envolvem paralelismo.

2.4.2 Interconexão

A interconexão é utilizada para conectar elementos de processamento a fim de obter a funcionalidade desejada. Algumas arquiteturas reconfiguráveis agrupam os seus blocos construtivos básicos em segmentos. Segmentos são unidades mínimas de *hardware* que podem ser configuradas em uma operação de reconfiguração. As arquiteturas reconfiguráveis possuem unidades dedicadas que possibilitam a interconexão intra-segmentos e intersegmentos. O modo de interconexão intra-segmentos conecta elementos situados dentro de um segmento, enquanto que o modo de interconexão intersegmentos conecta elementos de diferentes segmentos.

Em dispositivos de granularidade fina, são encontrados vários níveis de interconexão intra-segmentos, enquanto que em dispositivos de granularidade grossa, a interconexão é feita por barramentos ou redes em chip (*NOC - Network On Chip*). Nesta última forma de interconexão, pacotes podem ser roteados entre as unidades de processamento reconfiguráveis, assim como pode ser visto em [Azevedo et al., 2003] [Soares et al., 2004]. A interconexão intersegmentos é utilizada em RFUs para suportar instruções com múltiplos segmentos, com a finalidade de transmitir dados entre os diferentes segmentos.

2.4.3 Reconfigurabilidade

Se uma RFU pode ser configurada apenas em tempo de desenvolvimento, então considera-se que a unidade é apenas configurável. Se a RFU pode ser configurada após sua inicialização, em tempo de execução, a unidade é dita reconfigurável. Somente alguns dispositivos programáveis, como alguns tipos de FPGAs, possuem a capacidade de serem reprogramados, o que não as tornam suficientes para que sejam considerados dispositivos reconfiguráveis.

Quanto ao momento de reconfiguração as arquiteturas reconfiguráveis podem ser classificadas como estáticas ou dinâmicas. [Adário, 1997] diz que a reconfiguração dinâmica possibilita a reconfiguração durante a execução enquanto que a reconfiguração estática requer que a execução seja interrompida.

2.5 Trabalhos Relacionados

As arquiteturas reconfiguráveis foram sendo desenvolvidas e aprimoradas de acordo com a disponibilidade de tecnologia e visando a resolução de novos problemas e desafios. De acordo com [Mesquita, 2002], a partir do amadurecimento da tecnologia de FPGAs, as primeiras arquiteturas reconfiguráveis foram criadas com o intuito principal de aumentar o desempenho de algoritmos que até então eram executados em *software*. O primeiro processador de propósito geral acoplado a um FPGA foi denominado *PRISM* (*Processor Reconfigurable through Instruction-Set Metamorphosis*), em 1993 [Athanas and Silverman, 1993]. Devido a limitação da tecnologia dos FPGAs da época, somente era possível desenvolver um sistema fracamente acoplado. Um FPGA comunicava-se com um processador por meio de um barramento.

O primeiro processador dito reconfigurável foi denominado *PRISC* (*PRogrammable Instruction Set Computer*), em 1994 por [Razdan and Smith, 1994]. PRISC é o primeiro a explorar na pratica a existência de uma RFU contida em um caminho de dados de um processador MIPS.

O PRISC contém um conjunto de instruções *RISC* (*Reduced Instruction Set Computer*) convencional, juntamente com um conjunto de instrução de aplicação específica, que são implementadas em *hardware* por meio de unidades funcionais programáveis (*PFUs* - *Programmable Functional Units*). Estas PFUs são adicionadas à arquitetura mantendo os benefícios das técnicas RISC de alto desempenho e diminuindo o impacto do tempo de ciclo do processador. A Figura 9 mostra os recursos de *hardware* programável anexados diretamente no caminho de dados da CPU na forma de uma PFU. A PFU é adicionada em paralelo com as unidades funcionais existentes de forma que aumente, sem que ocorra uma substituição, a funcionalidade do caminho de dados existente. Cada PFU possui duas portas de entrada para os operandos e uma porta de saída contendo os resultados.

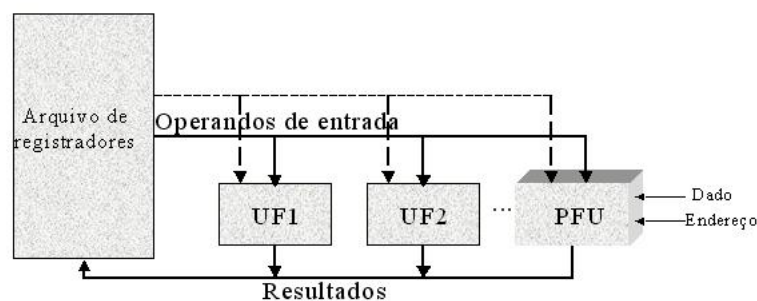


Figura 9: Caminho de dados do PRISC

Para programar a PFU, foi criada uma instrução denominada *expfu* (Execute PFU). A Fi-

gura 10 mostra um exemplo deste formato de instrução no MIPS. A instrução *expfu* atua como uma função booleana de duas entradas e uma saída. O sistema de compilação/síntese nomeia um número lógico da PFU para cada função booleana da aplicação. O campo de *LPnum* na instrução de *expfu* especifica a função particular para ser executada. Os 11 bits do campo *LPnum* permitem 2048 diferentes configurações da programação de PFU por aplicação. A informação da programação de cada função lógica da PFU é parte do segmento de dados do arquivo de objeto da aplicação. O número lógico da PFU é utilizado para indexação dentro do segmento de dados e para encontrar a informação de programação apropriada. Para cada PFU é associado um registrador de 11 bits denominado *Pnum*, contendo a função lógica da PFU atualmente programada. Se o valor de *LPnum* contida na instrução for o mesmo do registrador *Pnum*, a instrução *expfu* executa normalmente. Caso contrário, é gerada uma exceção.

expfu	Entrada 1	Entrada 2	Saída	LPnum

Figura 10: Formato de instrução do PRISC

Assim, PRISM e PRISC são representantes da primeira geração de processadores reconfiguráveis, onde verificou-se a eficiência da utilização de FPGAs para aplicações específicas, aumentando o desempenho com relação a soluções implementadas em *software*. O problema principal desta geração, segundo [Mesquita, 2002], é o gargalo de comunicação entre o processador e os FPGAs e o alto tempo de reconfiguração. O fato da reconfiguração ser estática impunha a interrupção da execução para que o sistema possa ser reconfigurado.

Com a evolução da tecnologia de integração, surgiu a segunda geração de processadores reconfiguráveis, tornando possível o desenvolvimento de um sistema composto pelo processador, FPGAs e memória em um único *chip*, resultando no embrião do que hoje se identifica por *SoC* (*System-on-a-Chip*). Também foi possível o desenvolvimento de processadores utilizando-se reconfiguração dinâmica.

[Wirthlin, 1995] desenvolveu o processador *DISC* (*Dynamic Instruction Set Computer*), implementado utilizando FPGAs parcialmente reconfiguráveis. A reconfiguração parcial permite a configuração de sub-seções de um FPGA. Cada instrução é tratada como um módulo de circuito independente, e estas instruções ocupam espaço neste FPGA somente quando necessário, de modo que estes recursos podem ser reusados para implementar instruções específicas a uma aplicação aumentando o seu desempenho.

Em 1997 surgiu o processador reconfigurável Chimaera, criado por [S. Hauck, 1997] e

[de Beeck et al., 2001]. O processador Chimaera é um *hardware* que consiste em um microprocessador com um coprocessador reconfigurável (*RCP - Reconfigurable Co-Processor*) integrado.

O Chimaera trata a lógica reconfigurável não como um recurso fixo, mas como um *cache* para instruções da RFU. As instruções que têm sido executadas recentemente, ou as que de alguma maneira existe uma predição de que serão necessárias, são mantidas na lógica reconfigurável. Se uma outra instrução for requerida, esta é enviada para a RFU, sobrescrevendo uma ou mais instruções previamente carregadas. Desta maneira, o sistema usa técnicas de reconfiguração em tempo de execução parciais para controlar a lógica reconfigurável. Isto requer que a lógica reconfigurável seja um tanto simétrica, de modo que uma instrução dada possa ser colocada no RFU onde quer que exista uma lógica disponível.

A fim usar instruções na RFU, o código da aplicação inclui chamadas à RFU, e os mapeamentos correspondentes na RFU são contidos no segmento da instrução dessa aplicação. As chamadas à RFU são feitas por instruções especiais que dizem ao processador para executar uma instrução RFU. Uma instrução ID é especificada determinando que instrução específica deve ser executada. Se essa instrução estiver presente na RFU, o resultado dessa instrução é escrito no registrador de destino. Desta maneira, as chamadas à RFU atuam como qualquer outra instrução. Se a instrução solicitada não estiver armazenada atualmente na RFU, o processador é parado enquanto a RFU busca a instrução da memória e se reconfigura.

Uma configuração própria de RFU determina de quais registradores serão lidos os operandos. Uma única instrução de RFU pode ler de todos os registradores conectados à RFU, permitindo que uma única instrução de RFU possa usar até nove operandos diferentes. Assim, a chamada de RFU consiste somente no CodOp denominado *RFUOP*, que indica que chamada à instrução RFU. Neste CodOp, um operando ID que especifica qual instrução será chamada e o operando do registrador de destino. Todas as instruções carregadas na RFU especulam sua execução durante cada ciclo do processador, embora seus resultados somente sejam escritos nos registradores quando sua chamada correspondente à RFU sejam realmente realizadas. Uma instrução RFU pode ser executada em múltiplos ciclos sem a necessidade de interromper o processador.

A arquitetura de Chimaera é mostrada na Figura 11. O componente principal do sistema é o arranjo reconfigurável, projetada para suportar execuções de alto desempenho. É neste arranjo que todas as instruções de RFU são executadas realmente. Este arranjo recebe suas entradas diretamente do processador ou de um banco de registradores espelho que duplica um subconjunto dos valores presentes no banco de registradores do processador. Próximo a este arranjo existe

um conjunto de "*Content Addressable Memory*"(CAM), uma por linha no arranjo reconfigurável. Estas CAMs verificam a instrução seguinte e determina se a instrução é do tipo *RFUOP*. Se o valor da CAM combinar com a *RFUOP* ID, o valor contido nesta linha no arranjo reconfigurável será escrito no barramento de resultado e enviado ao banco de registradores. Se a instrução que corresponde ao *RFUOP* ID não estiver presente, o controle da lógica de busca interrompe o processador e carrega a instrução RFU a partir da memória presente no arranjo reconfigurável. A lógica de busca também determina que partes do arranjo reconfigurável irão sobrescrever a instrução que está sendo carregada, e tenta reter as instruções RFU que provavelmente poderão ser utilizadas brevemente.

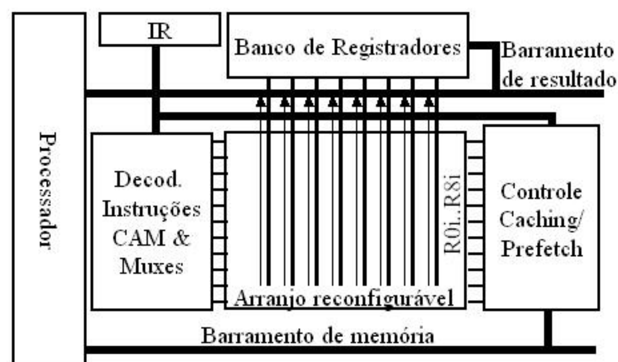


Figura 11: Arquitetura do Chimaera

O Chimaera é um processador criado visando alto desempenho, baseado em uma unidade funcional de granularidade fina conectada a um processador MIPS R4000. A lógica reconfigurável é organizada em 32 linhas com 32 blocos lógicos cada, com um total de 1024 Elementos de Processamento (*PE - Processing Elements*). A Figura 12 representa um bloco lógico do Chimaera. O bloco lógico pode ser configurado como uma 4-LUT, duas 3-LUTs, ou uma 3-LUT e uma execução de *carry*. Utilizando a configuração de *carry*, operações aritméticas e lógicas tais como a adição, subtração, comparação, paridade e outras podem ser suportadas eficientemente.

O GARP é uma arquitetura híbrida desenvolvida por [Hauser and Wawrzynek, 1997] em 1997, na qual o FPGA é apresentado como uma unidade funcional escrava localizada no mesmo espaço do processador. Esta arquitetura possui um coprocessador integrado a um núcleo padrão MIPS, com a característica que o coprocessador possui acesso próprio à memória. Ao contrário do PRISC e Chimaera, o GARP suporta operações multiciclo. O processador principal possui instruções especiais para controle, destinadas para carregar configurações, para fazer cópia de dados entre o arranjo reconfigurável e os registradores do processador, para manipulação do contador de relógio, entre outros. A Figura 13 mostra um esquema simplificado do GARP. O

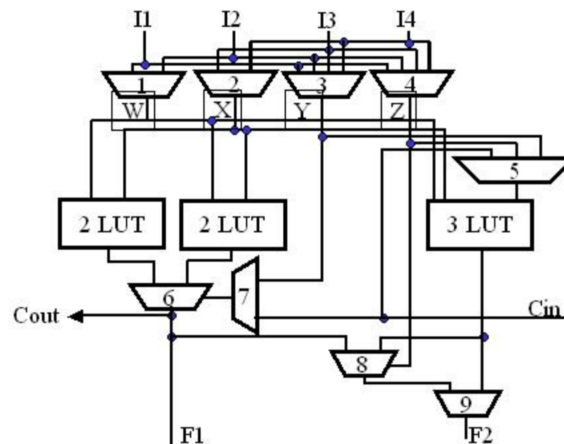


Figura 12: Bloco Lógico do Chimaera

arranjo reconfigurável do GARP é constituído de elementos denominados blocos (*blocks*). Um bloco em cada linha é denominado como bloco de controle (*Control Block*), enquanto que os restantes são denominados Blocos Lógicos Configuráveis (*CLBs* - *Configurable Logic Blocks*).

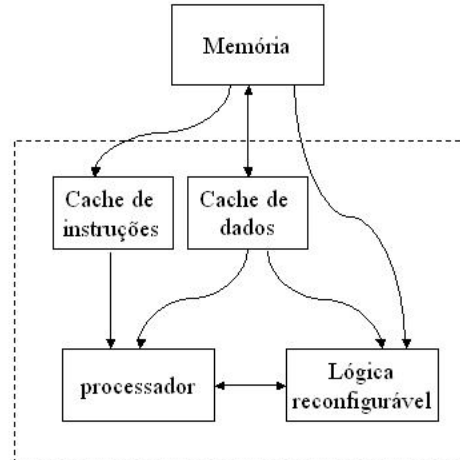


Figura 13: Arquitetura do GARP

Em 2001 foi criada por [Carrillo and Chow, 2001] a arquitetura OneChip. Esta arquitetura integra uma unidade reconfigurável dentro de um processador superescalar RISC *pipeline*. A OneChip permite a execução de múltiplas instruções simultaneamente e execução de instruções fora de ordem. Isto acarreta um melhor desempenho devido ao fato do processador e da lógica reconfigurável poderem executar diversas instruções em paralelo.

As características principais do processador OneChip são: reconfiguração dinâmica, lógica

programável no *pipeline* do processador, *pipeline* superescalar, que permite múltiplas instruções enviadas por ciclo e compressão da configuração, que reduz o tamanho da configuração.

[Carrillo and Chow, 2001] afirma que a Unidade Funcional Reconfigurável (RFU) do OneChip contém um ou mais FPGAs e um controlador FPGA, como visto na Figura 14. Estes FPGAs possuem múltiplos contextos e são capazes de suportar mais de uma programação para a lógica reconfigurável. Estas configurações são armazenadas em uma memória de contexto, que faz com que o FPGA seja capaz de alternar rapidamente entre as configurações. Cada contexto de um FPGA é configurado independentemente dos outros e atua como um *cache* de configurações. Somente um contexto pode ser ativado em um determinado momento. O Controlador FPGA é responsável pela programação dos FPGAs, pela alternância entre contextos e pela seleção de configurações para respostas se necessário. A *RBT* (*Reconfigurable Bits Table* - Tabela de Reconfiguração de Bits) atua como um gerente de configuração que mantém sob observação onde as configurações dos FPGAs estão alocadas. As informações contidas na tabela são o endereço de cada configuração e sinais que mantém sob observação as configurações carregadas e ativas.

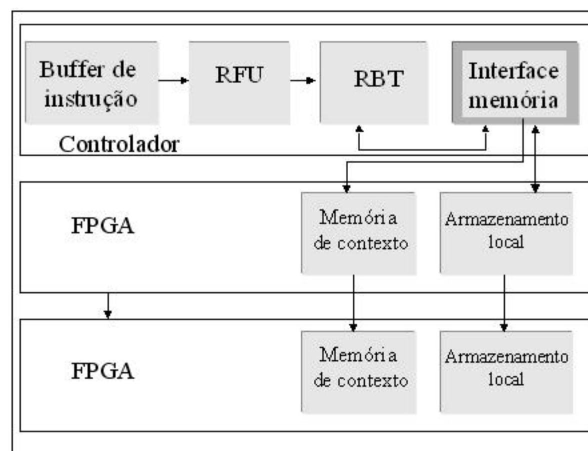


Figura 14: Arquitetura da RFU do OneChip

O processador CRISP (*Configurable and Reconfigurable Instruction Set Processor*), foi proposto originalmente por [Barat et al., 2002a] em 2002 depois redefinido como (*Coarse-grained Reconfigurable Instruction Set Processor*) em 2003 [Barat et al., 2003]. O CRISP é um processador que possui a característica de ser configurável em tempo de desenvolvimento e reconfigurável em tempo de execução. Consiste em um processador *VLIW* (*Very Long Instruction Word*), que executa instruções compostas por operações paralelas, executadas em unidades funcionais fixas (*FFUs* - *Fixed Functional Units*) e uma unidade funcional reconfigurável.

Segundo [de Beeck et al., 2001], a sua capacidade de reconfiguração em tempo de execução é similar as existentes no Chimaera e OneChip. As unidades funcionais fixas são unidades utilizadas em processadores VLIW típicos, como multiplexadores e unidades de E/S.

A unidade reconfigurável do CRISP recebe um CodOp e um ou mais operandos do banco de registradores, produzindo um ou mais resultados que são armazenados no banco de registradores. Esta arquitetura possui um decodificador reconfigurável, ao invés de um decodificador *hardwired*. Este decodificador consiste em uma memória de configuração endereçada pelo CodOp, de modo que o CodOp determina qual configuração será utilizada na lógica reconfigurável. Modificando os índices da memória de configuração, é possível alterar o comportamento da unidade reconfigurável, modificando o conjunto de instruções do processador.

Internamente, a unidade reconfigurável é composta por vários arranjos reconfiguráveis compostos por Elementos de Processamento de granularidade grossa, em que os elementos de processamento são cópias das unidades fixas, conectadas por meio de uma rede de conexão *crossbar*. Este *crossbar* pode conectar a saída de qualquer Elemento de Processamento à entrada de qualquer outro Elemento de Processamento. Também é possível conectar cada Elemento de Processamento ao banco de registradores mediante as portas de E/S da RFU. Cada Elemento de Processamento possui um registrador em sua saída, que pode ser *bypassed*. Ao combinar este registrador opcional com a rede *crossbar*, é possível realizar computação espacial.

Somente as Unidades Fixas são necessárias para executar programas. O arranjo reconfigurável é utilizado para aumentar o desempenho e diminuir o consumo de potência, comparado aos arranjos tradicionais, como as tabelas LUTs (*Look-Up Tables*). A Figura 15 mostra um esquema simplificado do processador CRISP.

O *XiRISC* (*eXented Instruction Set RISC*), descrito por [Lodi et al., 2003] em 2003, é um processador VLIW baseado em um RISC clássico com um *pipeline* de 5 estágios. Esta arquitetura apresenta unidades funcionais *hardwired* para cálculos *DSP* e um caminho de dados configurável adicional com *pipeline*, denominado ρ*Gate Array*, ou PiCoGA, que atua como um repositório de unidades funcionais de aplicações específicas virtual.

O XiRISC apresenta uma arquitetura *Load/Store*, onde os dados lidos da memória são armazenados em um banco de registradores antes de serem enviados para as unidades funcionais. A unidade funcional reconfigurável fornece a capacidade de dinamicamente estender o conjunto de instruções do processador com instruções multiciclo de uma aplicação específica, e então concluir a configuração em tempo de execução. O processador busca duas instruções de 32 bits em cada ciclo de relógio, que são executadas paralelamente nas unidades funcionais disponíveis, determinando dois fluxos de operações separados de modo simétrico, denominados canais

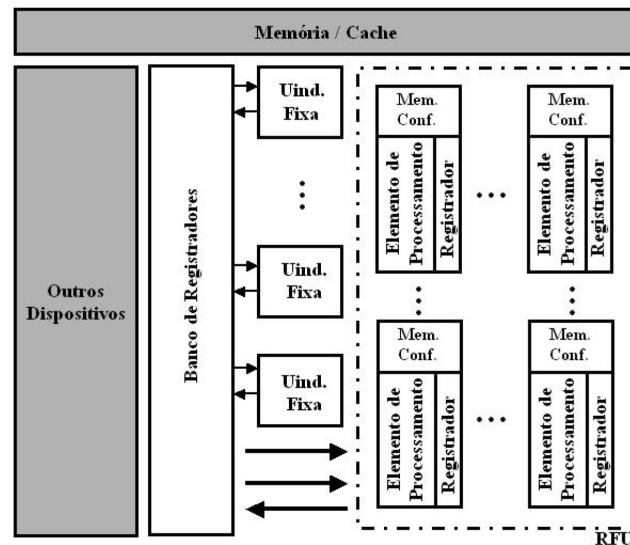


Figura 15: Processador CRISP

de dados (*data channels*). Os processadores de propósito gerais executam operações DSP. Já a unidade reconfigurável provê a capacidade de extender dinamicamente o conjunto de instruções do processador incluindo instruções multiciclo de aplicação específica. A fim de melhor explorar o paralelismo no nível de instrução, o ρ*GA* suporta até quatro entradas e dois registradores de destino para cada instrução *assembly* emitida. O conjunto de instruções do XiRISC possui duas instruções adicionais, que são: ρ*GA*-*load*, que carrega uma configuração dentro do PiCoGA, e ρ*GA*-*op*, que inicia a execução de uma função de aplicação específica previamente armazenada. A Figura 16 apresenta um esquema simplificado desta arquitetura.

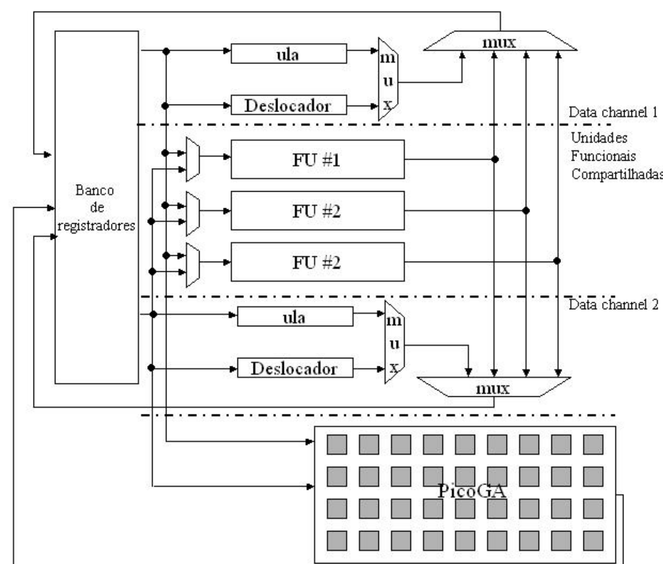


Figura 16: Processador XiRISC

Em 2003 surgiu a arquitetura *VISC* (*Variable Instruction Set Communications Architecture*), apresentado por [Liu et al., 2003]. A característica que diferencia o conjunto de instrução deste processador dos demais é a utilização de um dicionário que define o tipo de cada instrução e permite ao compilador configurar o melhor conjunto de instruções para a execução de um determinado programa. Durante a sua execução, o campo de CodOp contido na instrução é usado como índice do dicionário para obter uma palavra de controle, que configura o caminho de dados e as unidades de execução contidas na unidade reconfigurável. Para cada novo conjunto de instrução é redefinido um novo conteúdo do dicionário. Uma instrução VISC contém um CodOp com até quatro operandos, como visto na Figura 17.

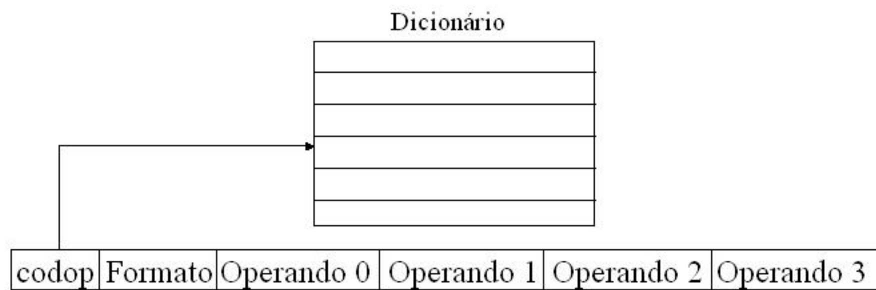


Figura 17: Formato de instrução do VISC

Atualmente os processadores reconfiguráveis estão voltados para aplicações multimídia, que não são facilmente implementadas em FPGAs convencionais. O gargalo da comunicação entre os dispositivos foi reduzido devido ao forte acoplamento entre o processador e a lógica reconfigurável. Estudos ainda continuam para que esse gargalo possa ser reduzido ainda mais. Nota-se nos processadores atuais uma tendência em obter diferentes formas quanto a utilização da lógica reconfigurável, em particular ao conjunto de instruções, projetando diferentes modelos de RISP.

2.6 O Padrão OCP

O padrão OCP (*Open Core Protocol*) [Partnerchip, 2001] é um conjunto de definições de sinais e protocolos de comunicação desenvolvido de modo a prover a interconexão de módulos de circuitos de uma forma padronizada, de forma a tornar compatível a transferência de informações entre eles, independente da forma de processamento de cada módulo. A utilização do padrão OCP possibilita a redução do tempo de projeto e do custo de manutenção de sistemas integráveis.

O padrão define uma interface ponto a ponto usando duas entidades de comunicação. Uma das entidades é definida como o mestre (*master*) e a outra definida como o escravo (*slave*). O mestre é a entidade de controle da comunicação e cabe a ele iniciar e gerar os sinais de controle. A entidade escravo responde aos comandos do mestre, tomando as ações correspondentes específicas de cada comando, recebendo os dados do mestre e/ou entregando dados ao mestre.

Para que um sistema seja considerado padrão OCP compatível, é necessário que cada interface OCP do sistema contenha no mínimo os sinais básicos OCP, cumpra um protocolo mínimo definido sobre os sinais básicos e obedeça a temporização dos sinais definida pelo padrão. Além disso, o sistema e suas interfaces devem ser descritas usando a sintaxe definida pelo padrão OCP e as especificações de desempenho devem ser definidas utilizando-se as regras estabelecidas. A Figura 18 ilustra os sinais OCP entre as entidades mestre e escravo.

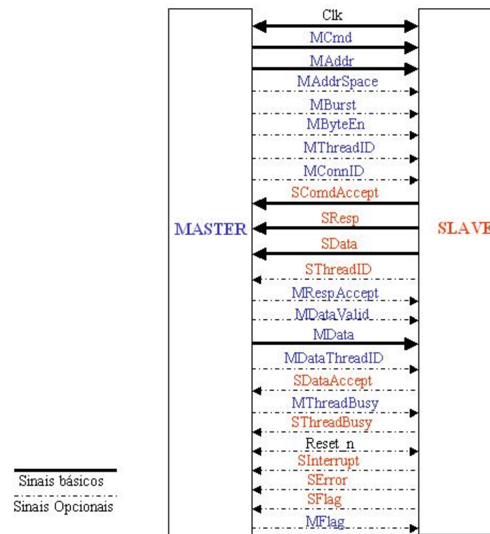


Figura 18: Sinais OCP

2.6.1 Sinais Básicos do OCP

Os sinais básicos do OCP são:

- Clk (1 bit): Todos os sinais são sincronizados com a borda positiva do relógio;
- MAddr (1-32 bits): Endereço de transferência, especifica o endereço de escrita do dado;
- MCmd (3 bits): Comando de transferência, indica o tipo de transferência de dados, codificado segundo a Tabela 1:

Tabela 1: Codificação dos comandos MCmd

MCmd[2:0]	**Tipo**	**Mnemônico**
0 0 0	Idle	IDLE
0 0 1	Write	WR
0 1 0	Read	RD
0 1 1	ReadEx	RDEX
1 0 0	Reservado	
1 0 1	Reservado	
1 1 0	Reservado	
1 1 1	BroadCast	BCST

- MData (8/16/32/64/128 bits): Dado de escrita, barramento de dados unidirecional do mestre para o escravo;

- SCmdAccept (1 bit): Indica que o escravo aceita o pedido de transferência do mestre;

- SData (8/16/32/64/128 bits): Dado de leitura, barramento de dados unidirecioanl do escravo para o mestre;

- SResp (2 bits): Resposta do escravo ao pedido de transferência do mestre, codificado segundo a Tabela 2:

Tabela 2: Codificação dos comandos SResp

SResp[1:0]	**Resposta**	**Mnemônico**
0 0	Sem resposta	NULL
0 1	dado válido/aceito	DVA
1 0	Reservado	RD
1 1	Erro	ERR

Além dos sinais básicos o padrão especifica outras categorias de sinais:

- *Simple Extensions* - Sinais que adicionam ao padrão maiores capacidades de endereçamento, de dados, e de suporte a transferência de dados em rajadas, e maior controle do fluxo de dados (*MAddrSpace*, *MBurst*, *MByteEn*, *MDataValid*, *MRespAccept*, *SDataAccept*) ;

- *Complex Extensions* - Suporte a threads e conexões (*MConnID*, *MDataThreadID*, *MThreadBusy*, *MThreadID*, *SThreadBusy*, *SThhreadID*);

- *Sideband Signals* - Sinais que não fazem parte do controle de fluxo de dados, sinais de flags, erro, interrupções, etc (*MFlag*, *Reset_n*, *SError*, *SFlag*, *SInterrupt*, *Control*, *ControlBusy*, *ControlWr*, *Status*, *StatusBusy*, *StatusRd*);

- *Test Signals* - Sinais que dão suporte a *scan*, controle de relógio, utilizados para teste do circuito (*Scanctrl*, *Scanin*, *Scanout*, *ClkByp*, *Testclk*, TCK, TDI, TDO, TMS, TRST_N)

2.6.2 Temporização

A Figura 19 mostra um exemplo do esquema de temporização entre as entidades mestre e escravo utilizando a interface OCP em operações de escrita e leitura utilizando apenas os sinais básicos.

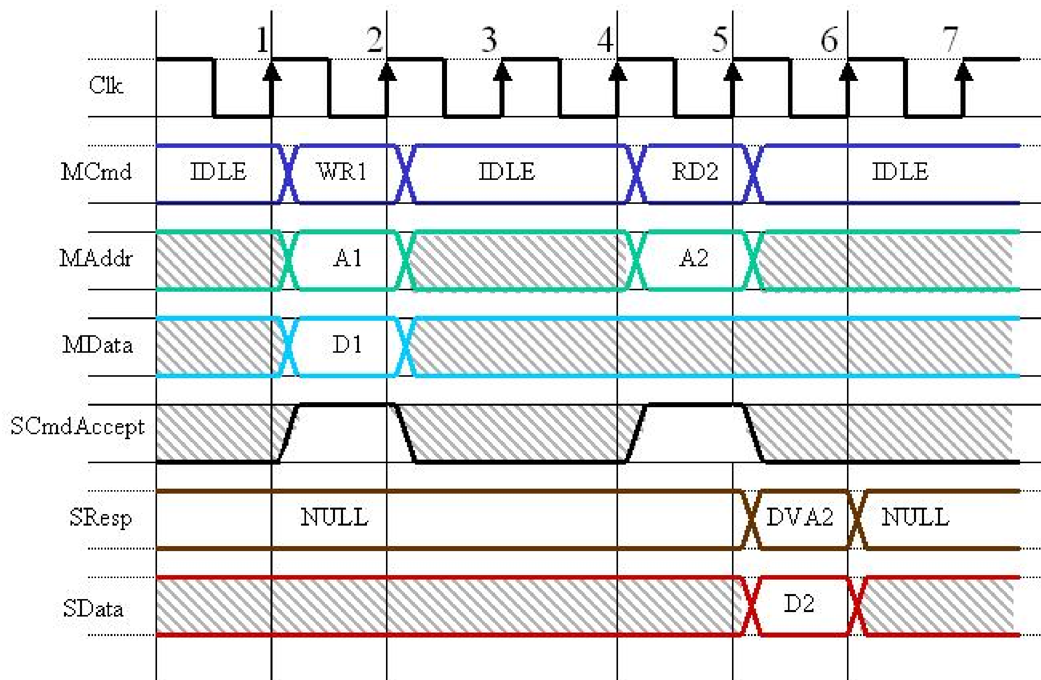


Figura 19: Temporização OCP envolvendo operações de escrita e leitura

O sinal de *MCmd* com valor *IDLE* até a primeira subida do relógio, indetificada na figura pelo número 1, indica que o sistema está ocioso (*IDLE*), sem execução de escrita ou leitura. A partir desse ponto, é efetuada uma solicitação de escrita da entidade mestre para a entidade escravo. Para que isto ocorra, é necessário que o barramento de comando da entidade mestre contenha um pedido de escrita (*MCmd* = WR1), o barramento de endereço da entidade mestre contenha o endereço de escrita (*MAddr* = A1) e o barramento de dados da entidade mestre contenha o dado a ser armazenado (*MData* = D1).

Durante o pedido em que a entidade mestre está em modo de escrita, o *MCmd* deve permanecer no estado WR1 até que a entidade escravo envie uma confirmação de que está pronta para armazenar o valor requisitado, ou seja, até que *SCmndAccept* = 1. Ao receber esta confirmação,

o barramento de comando retorna ao estado ocioso a partir da próxima subida de relógio, à espera de uma nova solicitação de escrita ou leitura.

O modo de funcionamento da unidade escravo durante uma operação de escrita ocorre da seguinte forma: ao ser confirmada a operação de escrita, o endereço e o dado enviados pela entidade mestre ficam disponíveis no barramento a espera de que a unidade escrava possa receber estes valores. Quando a unidade escravo estiver disponível, esta recebe os valores a fim de efetuar a escrita, ao mesmo tempo em que envia para a unidade mestre o sinal de confirmação (*SCmndAccept* = 1).

A quarta subida de relógio do exemplo ilustra uma solicitação de leitura. Neste caso, ao mesmo tempo que o barramento de comando recebe o valor correspondente a esta solicitação (*MCmd* = RD2), o valor do endereço de leitura é colocado no barramento de endereço da entidade mestre (*MAddr* = A2). Novamente, para que a operação seja efetuada, é necessário que a entidade escravo envie um sinal de confirmação indicando que está pronta para realizar o pedido (*SCmndAccept* = 1).

No período de relógio seguinte à confirmação (*clk* = 5), a entidade escravo, após enviar a confirmação que está apta a realizar a leitura, recebe o endereço A2 enviado e o utiliza para realizar a operação de leitura interna. O processo de resposta é realizado por meio dos barramentos *SResp* e *SData*. Supondo no exemplo que se trata de um dado válido, o barramento de resposta envia o valor (*SResp* = DVA2), ao mesmo tempo em que a entidade escravo envia para a entidade mestre o dado lido através do barramento de dados da entidade escravo (*SData* = D2).

A entidade mestre, ao receber o sinal de resposta indicando que o valor lido é um dado válido, recebe o valor contido em *SData*, finalizando a operação de leitura. Ao término desta etapa, o barramento de controle volta ao estado ocioso até que seja realizada uma nova operação.

Neste exemplo, durante a operação de escrita a entidade escravo envia o sinal de confirmação no mesmo período de relógio em que a unidade mestre solicitou a operação de escrita. Este processo não é obrigatório, pois não é necessário que a entidade escravo, como por exemplo uma memória, responda imediatamente ao pedido da entidade mestre, no caso um processador.

O próximo exemplo ilustra um esquema de temporização no qual ocorrem várias solicitações de escrita com fluxo de dados, como visto na Figura 20. Este exemplo contém três processos de escrita, com diferentes latências.

O exemplo inicia-se com um pedido de escrita feito pela unidade mestre durante a subida de relógio indicada pelo valor 1 (*clk* = 1). Este procedimento altera o valor do barramento

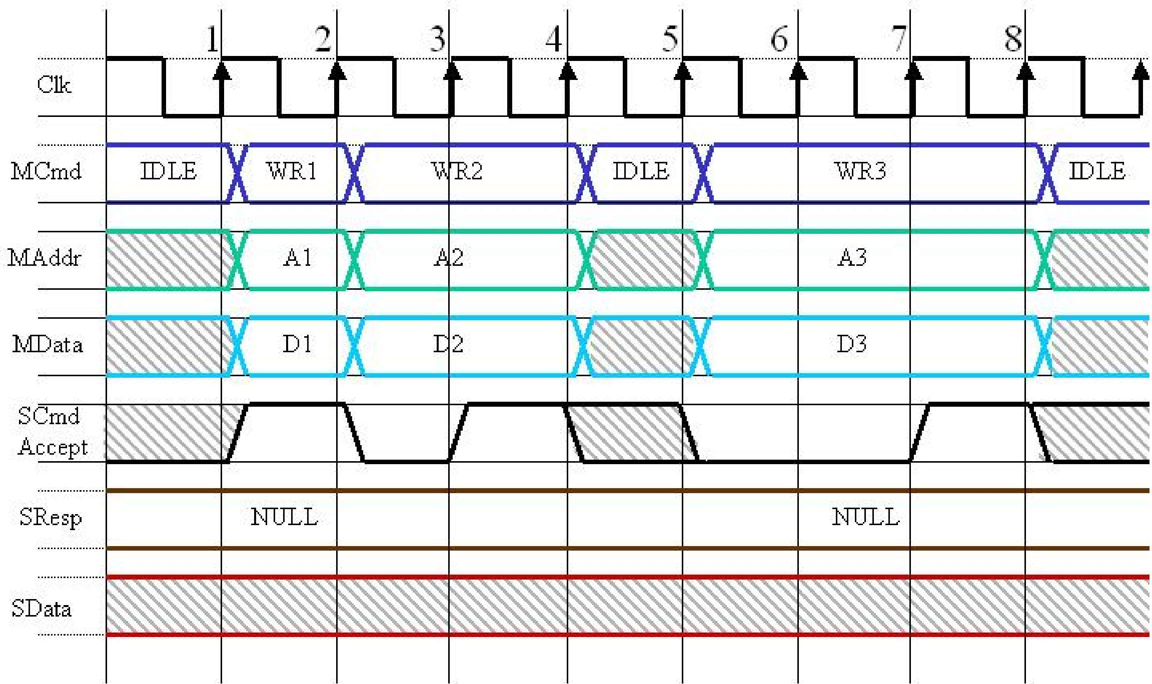


Figura 20: Temporização OCP com controle de fluxo

de controle (*MCmd*) de *IDLE* para WR1. Ao mesmo tempo o valor do endereço de escrita é colocado no barramento de endereço da entidade mestre (*MAddr* = A1) e o dado a ser transferido no barramento de dados da entidade mestre (*MData* = D1). A entidade escravo indica que aceita a solicitação do mestre durante o mesmo período de relógio ativando o sinal ScmdAccept = 1.

Na seqüência, ao mesmo tempo em que a entidade mestre inicia um novo pedido de escrita (*MCmd* = WR2), a entidade escravo recebe o endereço (A1) e o dado (D1) usando-os para realizar a operação de escrita interna. Como a entidade escravo ainda está trabalhando com os valores recebidos anteriormente, esta desativa o sinal de *ScmdAccept*, indicando que não está pronta para aceitar uma nova operação. Neste caso, enquanto o sinal *ScmdAccept* estiver desativado (*ScmdAccept* = 0), a entidade mestre mantém as informações dos barramentos (*MCmd*, *MAddr* e *MData)*, ou seja, permanece no mesmo estado e não poderá enviar novos valores.

Na subida de relógio 3, a entidade escravo ativa o sinal *ScmdAccept*, indicando estar pronta para realizar o pedido de escrita pendente. Portanto, na subida de relógio seguinte, o escravo captura o novo endereço (A2) e novo o dado (D2) usando-os para realizar a segunda operação de escrita interna.

Após um ciclo ocioso *(IDLE)*, a entidade mestre inicia um terceiro ciclo de escrita. Como o sinal *ScmdAccept* está desativado, as informações dos barramentos devem permanecer até que a entidade escravo envie o sinal de confirmação, não importa quantos ciclos de relógio

demorem. Neste exemplo, na subida de relógio 7 é enviado o sinal de confirmação, enquanto que no oitavo ciclo a entidade escravo captura as informações dos barramentos de escrita e de dados do mestre, mantidos nos dois ciclos anteriores, usando-os para realizar a operação de escrita interna pendente.

O último exemplo mostra um esquema de temporização com latência no processo de leitura, visto na Figura 21. A solicitação de leitura possui uma latência = 2, correspondente ao período em que o sinal *ScmdAccept* está desativado, enquanto que o processo de resposta possui uma latência = 3, correspondente ao período entre o final da solicitação da entidade mestre e a resposta e entrega dos valores pela entidade escravo.

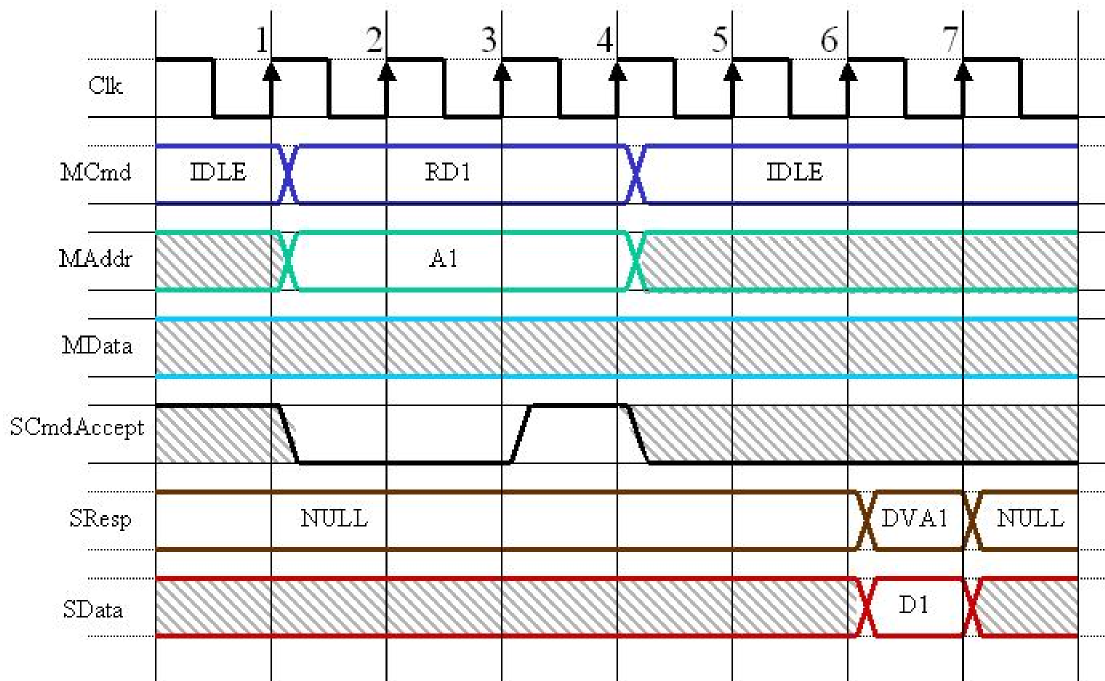


Figura 21: Temporização OCP com latência de leitura

O período de relógio 1 do exemplo ilustra uma solicitação de leitura pela entidade mestre, alterando o sinal do barramento de *IDLE* para RD1, ao mesmo tempo em que um valor do endereço é inserido no barramento de endereço. Por ser uma operação de leitura, nenhum valor é colocado no barramento de dados. Neste momento, a entidade escravo ainda não pode atender ao pedido, de modo que o sinal *ScmdAccept* está desativado.

No período de relógio 2, a entidade mestre recebe a indicação de que a entidade escravo ainda não está pronta para atender ao pedido, então esta mantém os valores contidos nos barramentos e continua esperando a confirmação do escravo. Somente no período de relógio 3 o sinal *ScmdAccept* é ativado, indicando o início do processo de leitura pela entidade escravo. A enti-

dade mestre ainda deve manter as informações nos barramentos e não poderá realizar nenhum outro processo enquanto não receber a resposta sobre o último pedido de leitura realizado.

Durante o período 4, a entidade escravo captura as informações enviadas pelo mestre. O processo de leitura somente é concluído quando o escravo emitir um sinal de resposta. Caso a entidade escravo necessite esperar algum dado de algum dispositivo externo, o sinal de resposta deverá permanecer como *NULL*.

Quando o escravo estiver pronto, este envia um sinal de resposta indicando que o dado é valido (*SResp* = DVA1), bem como a informação requisitada no barramento de leitura (*SData* = D1). Após o mestre receber o sinal de resposta, o período de relógio seguinte é dedicado ao armazenamento da informação solicitada. Após este último passo, torna-se possível novas requisições de leitura ou escrita.

Os exemplos de temporização apresentados mostram que as operações de escrita gastam no mínimo dois ciclos de relógio para serem efetuadas, enquanto que as operações de leitura gastam pelo menos três ciclos de relógio.

O padrão OCP oferece outros modos de temporização, bem como diversas opções de utilização, que não serão vistas por não serem utilizadas no escopo deste trabalho.

3 ARQUITETURA PROPOSTA

Este capítulo disserta sobre o modelo de arquitetura proposto nesta dissertação, no que tange a concepção das unidades constituintes do processador - unidades operativa, de controle e de reconfiguração.

A tendência dos RISP atuais é a busca de novas técnicas para a reconfiguração do conjunto de instruções em tempo real. Geralmente, tais processadores realizam modificações no caminho de dados para adequar-se à lógica reconfigurável.

Esta característica faz-se uma das diferenças entre os processadores RISP mencionados anteriormente e o modelo de arquitetura proposto nesta dissertação. Parte da lógica reconfigurável encontra-se inserida dentro da unidade de controle do processador, enquanto que foram realizadas apenas as modificações necessárias no caminho de dados do processador para que o processador pudesse realizar a reconfiguração do conjunto de instruções. Também foi criada uma unidade de reconfiguração com a finalidade de criação, modificação e combinação de novas instruções, em tempo de execução, enviadas pelo usuário utilizando-se o padrão OCP.

Outra característica importante do processador é a possibilidade de poder trabalhar com diversos conjuntos de instruções contidos dentro da unidade de controle, denominados conjuntos de instruções fixos e de alternar entre estes conjuntos de acordo com o desejo do usuário, podendo escolher o conjunto de instrução que melhor se adapte à aplicação atual, proporcionando um melhor desempenho na sua execução, como visto em [Casillo et al., 2005].

As sessões seguintes contém maiores detalhes sobre como são realizadas a criação e modificação de novas instruções através da unidade reconfigurável e como são implementados os conjuntos de instruções fixos.

3.1 Visão Geral

A arquitetura proposta foi projetada originalmente a partir do microprocessador MIC-1 descrito por [Tanenbaum, 1992]. O MIC possui um caminho de dados composto por 16 regis-

tradores idênticos de 16 bits, formando uma memória de rascunho acessível apenas ao nível de microprogramação. Cada registrador é ligado a dois barramentos de saída, A e B, que alimentam uma ULA de 16 bits. A saída da ULA é ligada a um deslocador, que pode deslocar o resultado da operação em 1 bit para a esquerda ou direita, e o resultado final pode ser armazenado na memória de rascunho ou ser enviado para a memória externa.

Já em [Ramos et al., 2004], foi apresentada uma expansão desta arquitetura, mantendo-se a estrutura original do caminho de dados de [Tanenbaum, 1992], com o acréscimo de instruções de entrada/saída e de tratamento de interrupção, e também foram realizadas alterações de tal forma a transformar o MIC-1 em uma entidade mestre OCP. Para tanto, projetou-se um *wrapper* [1], que recebe sinais do MIC-1 e converte-os para sinais no padrão OCP, assim como recebe sinais no padrão OCP e transfere-os para o MIC-1.

Utilizando estas arquiteturas como base, foi projetada a arquitetura apresentada nesta dissertação. Internamente, este modelo de arquitetura é um processador multiciclo com vias de dados de 32 bits, composto por uma unidade operativa, na qual o caminho de dados é baseada na estrutura do MIC-1, mas com diversas modificações para adaptar-se às novas funcionalidades; uma unidade de controle contendo uma Máquina de Estados Finitos (*Finite State Machine* - *FSM*) que controla o funcionamento da arquitetura e engloba os conjuntos de instruções fixos; e uma unidade de reconfiguração, projetada para reconfigurar e combinar instruções. A Figura 22 apresenta uma visão geral dos blocos funcionais presentes no processador e a ligação entre estes. Cada bloco funcional será descrito posteriormente ao longo deste capítulo. O esquema completo do processador pode ser visto no Apêndice A.

Figura 22: Unidades constituintes do processador

[1] wrapper trata-se de um conector que integra dispositivos OCP a outros dispositivos não adaptados a este padrão

3.2 Unidade Operativa

A unidade operativa é composta por *latches*, multiplexadores, dois registradores denominados MAR (*Memory Address Register*) e MBR (*Memory Buffer Register*), uma ULA e um deslocador de 32 bits e um conjunto de 32 registradores de trabalho. A Figura 23 ilustra o caminho de dados do processador.

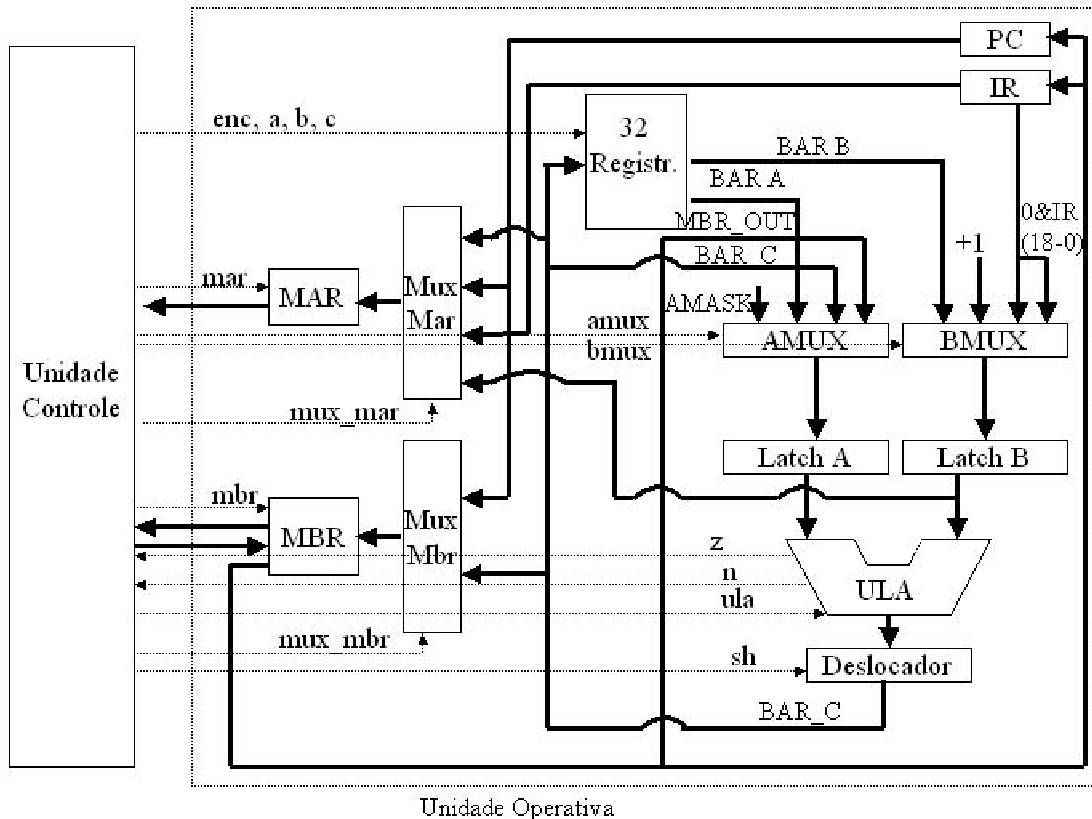


Figura 23: Unidade Operativa do processador

A arquitetura de [Tanenbaum, 1992] pode realizar apenas quatro operações: soma de dois operandos, *and* lógico, transparência do primeiro operando e negação do primeiro operando. A operação de subtração entre dois operandos é realizada utilizando-se a forma de complemento de 2 ($x - y = x + 1 + not(y)$), de modo que são gastos 3 ciclos de relógio para sua execução.

Para eliminar estes ciclos extras e aumentar o desempenho do processador na execução de aplicações, a ULA do processador proposto foi projetada para realizar oito operações distintas com operandos de 32 bits: transparência do primeiro operando, *and* e *or* lógicos, negação do primeiro e do segundo operando e operações aritméticas de adição, subtração e multiplicação de dois operandos, sendo todas estas realizadas em um ciclo de relógio.

Um ciclo básico da ULA consiste da seqüência dos seguintes eventos: carga dos *latches* A e B, execução da operação programada pela ULA e/ou pelo deslocador e armazenamento dos resultados nos registradores. Assim como na arquitetura de [Tanenbaum, 1992], para obter o seqüenciamento correto de eventos, foi utilizado um relógio com quatro subciclos. Os eventos chave durante cada um dos subciclos são:

- Receber os sinais de controle responsáveis pela execução do estado atual a partir da unidade de controle;
- Colocar os conteúdos dos registradores contidos na memória de rascunho nos barramentos de saída A e B e armazená-los nos latches A e B;
- Produzir uma saída estável a partir da ULA e do deslocador, e carregar o MAR se requisitado;
- Armazenar o valor existente no barramento C na memória de rascunho e carregar o MBR ou MAR, se for necessário.

O conjunto de registradores contém 32 registradores de 32 bits, sendo os quatro primeiros destinados a funções específicas, como Acumulador (*AC*) e apontador de pilha (*SP* - *Stack Pointer*), e 28 restantes para uso geral. O relação completa do banco de registradores do processador e sua codificação encontra-se no Apêndice B.

3.3 Modos de funcionamento

O processador pode atuar sob dois modos de funcionamento: modo de reconfiguração e modo de execução. No modo de execução, podem ser tratadas as instruções fixas (instruções contidas nos conjuntos de instruções fixos) ou instruções reconfiguráveis (instruções criadas ou combinadas a partir da unidade de reconfiguração). A cada início de execução de uma instrução, verifica-se se esta instrução atual é fixa ou reconfigurável. Caso seja uma instrução fixa, a unidade de controle envia os comandos correspondentes aos estados que compõem a instrução. Caso contrário, a unidade de controle busca os estados correspondentes a instrução reconfigurada armazenados na unidade de reconfiguração para em seguida enviá-los à parte operativa a fim de serem executados.

O funcionamento do modo de reconfiguração exige que o usuário envie um sinal ao processador informando que deseja reconfigurar uma instrução, juntamente com os valores do CodOp reconfigurável, de 8 bits, e do valor da palavra de reconfiguração, de 28 bits, que representa a

operação atual da instrução reconfigurada. Caso uma instrução possua mais de uma operação, o valor do CodOp deve ser mantido, e para cada nova operação é enviada uma nova palavra de reconfiguração até que seja indicado o término da instrução. Estes modos de funcionamento serão detalhados nas sessões referentes às unidades de reconfiguração e unidade de controle.

3.4 Utilização do padrão OCP

O processador proposto foi projetado adotando as normas do padrão OCP. Por já encontrar-se adaptado ao padrão, não é necessário a utilização de *wrappers* para a comunicação do processador com dispositivos externos, contanto que tais dispositivos também estejam adaptados ao padrão OCP. Foram implementados apenas os sinais básicos do OCP para realizar operações simples de escrita e leitura.

O processador se comunica com um dispositivo de E/S e com a memória. A comunicação entre o processador e a memória é realizada de modo que o processador foi definido como a entidade mestre (*master*) e a memória definida como entidade escravo (*slave*). Neste esquema, o processador é considerado a entidade de controle da comunicação e possui como tarefa gerar os sinais de controle e requisições de escrita ou leitura na memória. A memória, por sua vez, responde aos comandos do processador tomando as ações correspondentes específicas para cada solicitação, recebendo os dados do processador e/ou entregando dados a este.

Já a comunicação entre o dispositivo de E/S e o processador é realizada de modo inverso, sendo o dispositivo de E/S atuando como entidade mestre e o processador como entidade escravo. A tarefa deste dispositivo de E/S é de gerar apenas requisições de escrita no processador, enviando as informações referentes as instruções que serão reconfiguradas. Os detalhes desta implementação serão vistos adiante. A Figura 24 mostra uma representação da comunicação entre estes dispositivos.

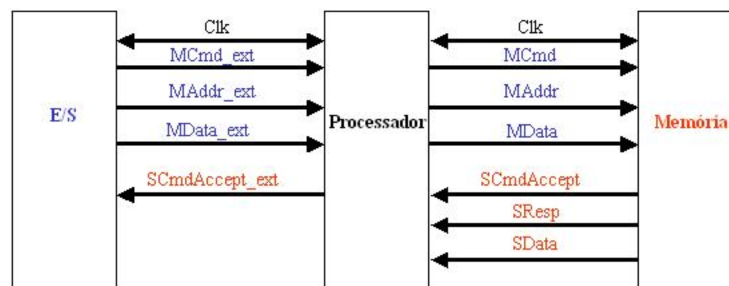


Figura 24: Representação das unidades adaptadas ao padrão OCP

3.5 Unidade de Controle

A unidade de controle é implementada sob a forma de uma máquina de estados finitos com 118 estados. A ilustração da máquina de estados completa encontra-se no Apêndice C.

Com o intuito de validar a proposta do processador conter mais de um conjunto de instrução e poder alternar entre estes conjuntos durante o seu funcionamento, foram implementados três conjuntos de instruções simples para possibilitar a realização de operações envolvendo a alternância entre tais conjuntos.

Os três conjuntos de instruções implementados são compostos, cada um, por 32 instruções, sendo 30 de propósito geral específicas de cada conjunto, e 2 instruções comuns a todos os conjuntos, vistas adiante. O conjunto de instrução 1 é basicamente o conjunto apresentado em [Tanenbaum, 1992] para o processador MIC-1 e aprimorado em [Ramos et al., 2004] com o acréscimo de instruções de tratamento de interrupção e Entrada/Saída (E/S). Neste conjunto apenas o acumulador e o apontador de pilha são acessíveis ao usuário. Os demais conjuntos utilizam o conjunto 1 como base e acrescentam novas instruções. O segundo conjunto possui instruções em que os 32 registradores são acessíveis ao usuário e o terceiro conjunto é baseado em instruções que utilizam operações envolvendo a pilha. Maiores detalhes acerca destes conjuntos de instruções podem ser vistos no Apêndice D.

Cada CodOp corresponde a uma série de estados, os quais podem estar relacionados a uma ou mais instruções distintas, de acordo com o conjunto de instrução utilizado. A Figura 25 apresenta um exemplo da implementação dos estados dos conjuntos de instruções durante a execução de um mesmo CodOp. Como exemplo, será utilizado o CodOp ”00000101” (Op_05H).

Neste exemplo, cada conjunto executa uma instrução diferente a partir do mesmo CodOp recebido. O conjunto de instrução 1 inicia a instrução ADDL X, o conjunto de instrução 2 inicia a instrução LODR Ra, Rx, Ry, e o conjunto de instrução 3 inicia a instrução LODI. Cada instrução pode ser executada de maneira independente por cada conjunto, contento a quantidade de operações necessárias para sua execução. Este exemplo demonstra a possibilidade de que, para um mesmo CodOp em particular, os conjuntos de instruções podem implementar instruções de maior ou menor quantidade de estados. As microinstruções presentes em cada um dos conjuntos de instruções são:

Estado 05_1:

- ADDL - MAR := SP + IR; RD; GOTO OP13_2
- LODR - MAR := Rx + Ry; RD

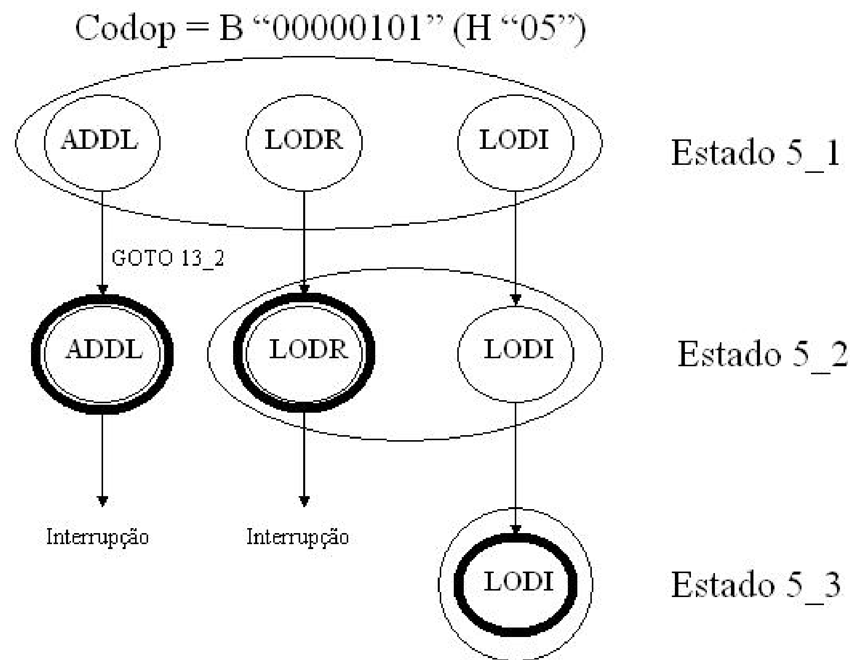


Figura 25: Conjunto de estados do CodOp 05H

- LODI - MAR := SP; RD

Estado 05_2:

- ADDL - AC := MBR + AC; GOTO INT;
- LODR - Ra := MBR; GOTO INT;
- LODI - MAR := MBR; RD

Estado 05_3:

- ADDL - instrução concluída
- LODR - instrução concluída
- LODI - MAR := SP; WR; GOTO INT

Qualquer instrução, não importa em qual conjunto de instrução esteja, termina apenas quando é executado o comando GOTO INT, o que dá início ao processo de interrupção. A rotina de interrupção consiste em uma seqüência de 11 estados com a finalidade de armazenar o conteúdo do acumulador, apontador de pilha e contador de programa.

Para que possa ocorrer uma interrupção é preciso que aconteçam dois eventos. O primeiro é a execução da instrução SETI, presente em todos os conjuntos de instruções implementados. O segundo evento é a ativação da entrada *INTR*, que indica o pedido de interrupção. Somente quando estes dois eventos acontecerem, o processador atenderá ao pedido de interrupção. Para recuperar o contexto depois de uma interrupção, foi implementada a instrução RETI, que também está presente em todos os conjuntos de instruções. Ao ser executada uma interrupção, um sinal indicando que o pedido de interrupção foi aceito é ativado (*INTA*). Após o término do processo, a máquina de estados retorna ao estado inicial.

A Figura 26 ilustra mais um exemplo da utilização dos conjuntos de instruções fixos, utilizando a instrução LODD. O estado OP0_1 é utilizada pelos três conjuntos implementados. No segundo estado, OP0_2, os conjuntos 1 e 2 finalizam a sua execução, embora o conjunto 3 contenha ainda mais estados necessários para ser concluído. Isto demonstra a possibilidade de compartilhamento de estados por mais de um conjunto de instruções. Este fato leva a redução da lógica utilizada para a implementação da máquina de estados.

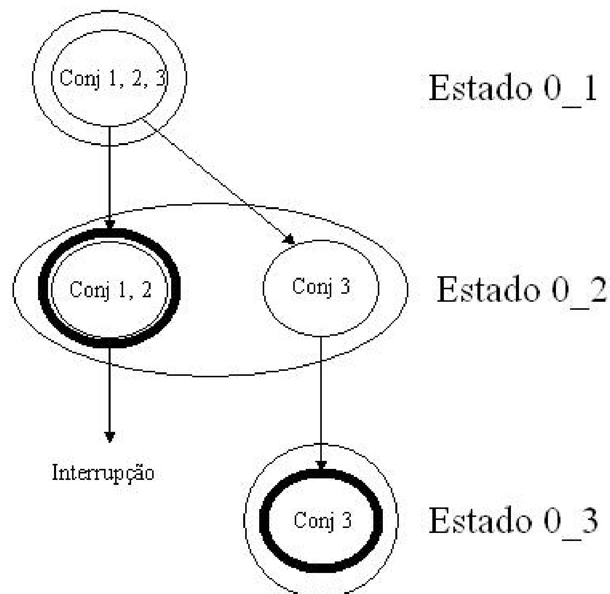


Figura 26: Conjunto de estados da instrução LODD

Para que o usuário possa alternar entre os conjuntos de instruções implementados, foram criadas duas instruções, denominadas *muda_conj* e *muda_modo*. A função da instrução muda_conj consiste em alterar o conjunto de instrução atualmente utilizado. Foi utilizado um registrador de 8 bits denominado *registrador de estado* dentro da unidade de controle. Este

registrador é responsável por armazenar os bits que informam qual conjunto de instrução está selecionado atualmente, codificados de forma que os dois último bits, quando possuem valor ”00” representam o conjunto 1, o valor ”01” representa o conjunto 2 e o valor ”10” representa o conjunto 3. Ao executar a instrução muda_conj, os 2 últimos bits do CodOp são enviados ao registrador de estado, indicando qual conjunto deve ser selecionado. O registrador de estado é alterado somente mediante esta instrução, portanto não é possível alternar entre os conjuntos durante a execução de uma instrução. A cada reinício do processador, o conjunto de instrução 1 é selecionado.

A instrução muda_modo altera o modo de funcionamento do controle fixo. Ao executar esta instrução, os bits 7 e 6 do CodOp são enviados ao registrador de estados. Estes bits representam o modo de funcionamento do processador, codificados da seguinte forma: o valor ”00” representa o modo usuário e o valor ”01” representa o modo supervisor. Somente é possível utilizar a instrução muda_conj se o processador estiver em modo supervisor. Outros modos de funcionamento não foram implementados.

3.6 Instruções

Todas as instruções implementadas estão classificadas em 5 formatos distintos. O primeiro formato é do tipo (8, 24), sendo oito bits de CodOp e 24 bits utilizados para endereço ou dados. Este formato é utilizado em instruções de leitura/armazenamento, tais como LODD X e STOD X.

O segundo formato, do tipo (8, 5, 19), é formado por oito bits de CodOp, cinco bits para o registrador destino da operação e 19 bits para dados ou endereço. As instruções que usam esse formato são as instruções que realizam operações dado-registrador, tal como a instrução LODL Rx,Y, que significa carregar um valor de Y de 19 bits no registrador Rx.

O terceiro formato possui o tipo (8, 5, 5, 14), formado por oito bits de CodOp, cinco bits para o registrador destino da operação, 5 bits para um registrador de entrada e 14 bits para dados ou endereço. Não existem instruções presentes nos conjuntos fixos que utilizam este formato, sendo portanto utilizado apenas em instruções reconfiguradas. No Capítulo 4 será visto um exemplo de implementação deste formato.

O quarto formato é do tipo (8, 5, 5, 5, 9), com oito bits de CodOp, cinco bits para o registrador destino e cinco bits para os registradores que contêm o primeiro e segundo operandos, respectivamente. São instruções típicas desse formato as operações aritméticas de três endereços, como ADDR Ra, Rx, Ry.

O quinto e último formato consiste no tipo (8, 5, 5, 5, 5, 4), com oito bits de CodOp, cinco bits para o registrador destino e cinco bits para os registradores que contêm o primeiro, segundo e terceiros operandos, respectivamente. Este formato somente é utilizado em instruções reconfiguradas. O Capítulo 4 também contém um exemplo da implementação de uma instrução utilizando este formato.

A Tabela 3 a seguir contém a relação de todas as instruções contidas nos três conjuntos de instruções. No Apêndice E encontram-se informações detalhadas sobre os conjuntos implementados.

Tabela 3: Conjuntos de instruções do processador

CodOp	**Conjunto 1**	**Conjunto 2**	**Conjunto 3**
00000000	LODD X	LODD X	LODDp X
00000001	STOD X	STOD X	STODp X
00000010	LOCO X	LOCO X	LOCO X
00000011	LODL X	LODL Rx, Y	LODLp X
00000100	STOL X	STOL Y, Rx	STOLp X
00000101	ADDL X	LODR Ra, Rx, Ry	LODIp X
00000110	SUBL X	STOR Ra, Rx, Ry	STOIp X
00000111	CALL X	CALL X	CALL X
00001000	JUMP X	JUMP X	JUMP X
00001001	JNEG X	JNEG X	JNEG X
00001010	JPOS X	JPOS X	JPOS X
00001011	JNZE X	JNZE X	JNZE X
00001100	JZER X	JZER X	JZER X
00001101	ADDD X	ADDR Ra, Rx, Ry	ADDp X
00001110	SUBD X	SUBR Ra, Rx, Ry	SUBp X
00001111	MULL X	MULL Ra, Rx, Ry	MULp X
11110000	RETN	RETN	RETN
11110001	PSHI	SHL	SHLp
11110010	POPI	SHR	SHRp
11110011	PUSH	PUSH	PUSH
11110100	POP	POP	POP
11110101	SWAP	SWAP	SWAP
11110110	SETI	SETI	SETI
11110111	RETI	RETI	RETI
11111000	INSP	INSP	INSP
11111001	DESP	DESP	DESP
11111010	LDIO	LDIO	LDIO
11111011	STIO	STIO	STIO
11111100	-	-	-
11111101	-	-	-
11111110	muda_conj	muda_conj	muda_conj
11111111	muda_modo	muda_modo	muda_modo

3.7 Processo de Execução de Instruções Fixas

A Figura27 apresenta o diagrama de estados contendo apenas as etapas necessárias para este processo.

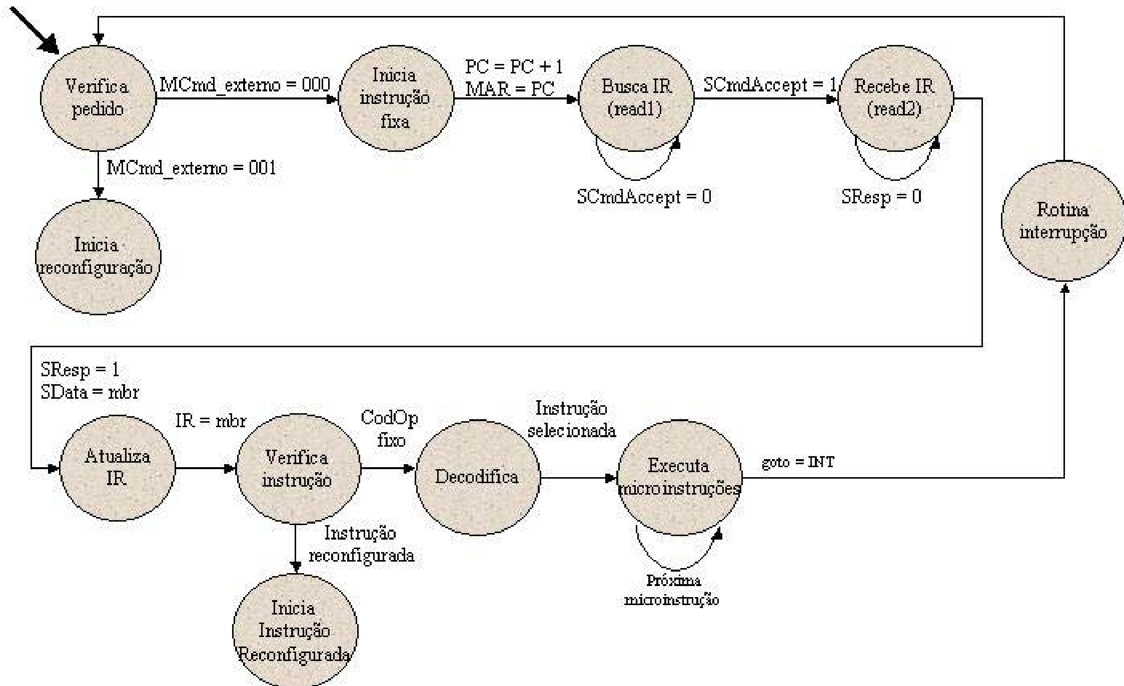


Figura 27: Diagrama de estados do processo de execução de instruções fixas

O estado *Verifica_pedido* é responsável por verificar se houve um pedido de reconfiguração de instrução através do sinal de *MCommand* externo. Caso este sinal contenha um pedido de reconfiguração ($MCmd_externo = 001$), será iniciada a seqüência de reconfiguração de instruções, que será vista na subseção 3.8.1. Caso contrário ($MCmd_externo = 000$), iniciará a execução de uma instrução fixa ou reconfigurada. A seqüência de funcionamento de instruções reconfiguradas será vista conforme a subseção 3.8.2.

Ao ser iniciado o processo de execução de instruções fixas, o primeiro evento consiste na busca de uma instrução. Para que este evento ocorra, é necessário realizar uma busca na memória ($MCmd = 010$), enviando o valor de PC através do padrão OCP ($MAddr = PC$), fato este que ocorre no estado "*Read1*". A máquina de estados deve permanecer neste estado enquanto não receber um sinal de confirmação da memória ($SCmmdAccept = 1$). A partir desta confirmação, é executado o estado "*Read2*", em que o processador recebe da memória um sinal que indica se o dado é valido (*SResp = 001*), juntamente com o valor solicitado da instrução (*MBR = SData*).

No estado seguinte (*Atualiza_IR*), o valor do registrador de instrução (IR) é atualizado com o valor de MBR.

O estado *Verifica_instrução* é responsável por verificar se o CodOp (IR) recebido é do tipo fixo ou reconfigurável. Se os quatro primeiros bits do CodOp forem ”0000” ou ”1111”, significa que a instrução pertence a algum dos conjuntos fixos, caso contrário a instrução a ser executada é do tipo reconfigurável. A sessão 3.8 contém maiores detalhes sobre CodOps fixos e reconfiguráveis.

Em se tratando de uma instrução pertencente aos conjuntos de instruções fixos, o estado seguinte será o estado de decodificação da instrução. Após a decodificação, a instrução será executada de acordo com o conjunto de instrução selecionado.

O exemplo a seguir mostra a execução da instrução de CodOp "00000010"(02H), que representa a instrução LOCO X (*Load Constant X*). Esta instrução possui apenas uma microinstrução:

- Acc := BAND(AMASK, IR); GOTO INT;

Para executar esta microinstrução, a Unidade Operativa recebe da Unidade de Controle os sinais que controlam seus blocos funcionais, como visto na Figura 28.

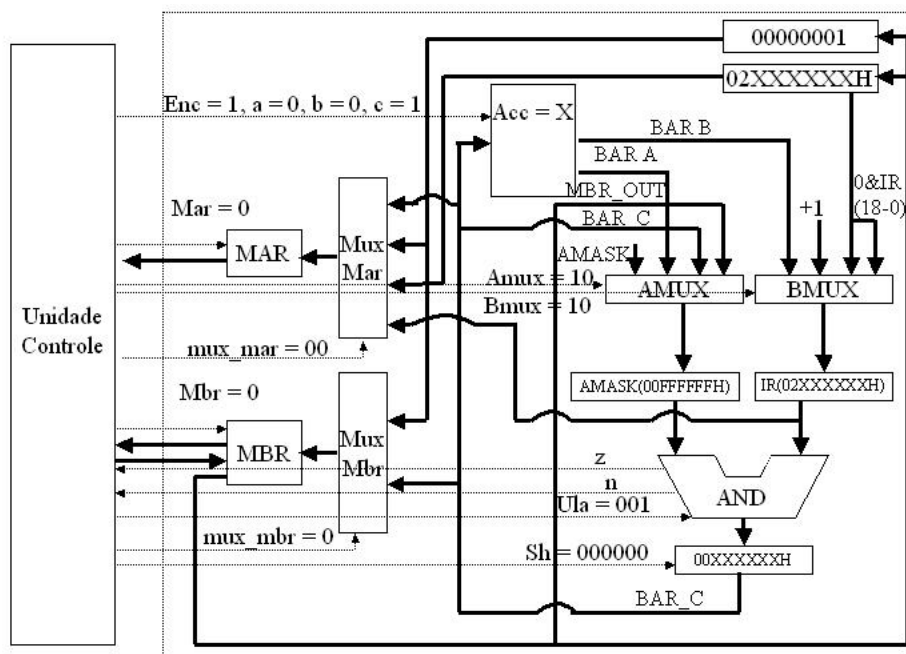


Figura 28: Implementação da instrução LOCO

O significado desta instrução consiste no acumulador (registrador "00001") receber o resultado de uma operação *AND* entre o valor de IR e a máscara AMASK. Esta operação faz com que os 8 bits de CodOp presentes no registrador IR se tornem iguais a zero. Assim, o acumulador receberá o valor ”00000000*B*” juntamente com os 24 bits restantes de IR. Em outras palavras, o acumulador recebe um valor constante de 24 bits.

Todas as demais instruções, presentes nos três conjuntos de instruções fixos, são executadas de maneira análoga a anteriormente exemplificada. A presença da microinstrução *GOTO INT* representa o final da execução da instrução.

3.8 Unidade de Reconfiguração

A unidade de reconfiguração foi projetada para o tratamento de instruções reconfiguráveis. A Figura 29 ilustra, de um modo geral, os blocos funcionais constituintes desta unidade. O funcionamento desta unidade, bem como a descrição e utilização de cada bloco funcional que compõe a unidade, serão vistos nas subseções a seguir.

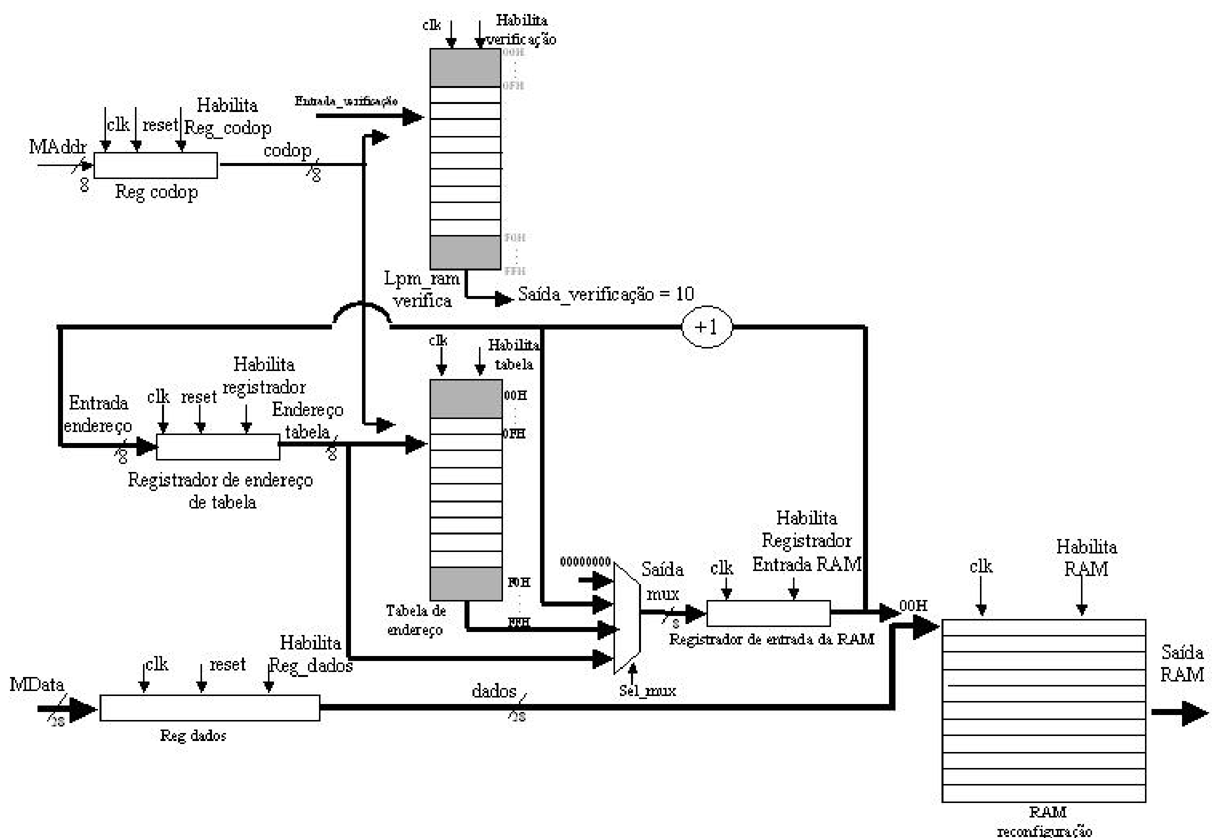


Figura 29: Unidade de reconfiguração

Esta unidade possibilita criar uma instrução que não se encontra nos conjuntos de instruções fixos e combinar duas ou mais instruções reconfiguradas. A seção 3.9 contém exemplos da implementação de instruções reconfiguradas sob cada uma destas técnicas.

3.8.1 Processo de reconfiguração de uma instrução

O Digrama de estados da Figura 30 representa os estados correspondentes ao processo de reconfiguração de uma instrução. Estes estados fazem parte da máquina de estados (FSM) do processador, que encontra-se no Apêndice C.

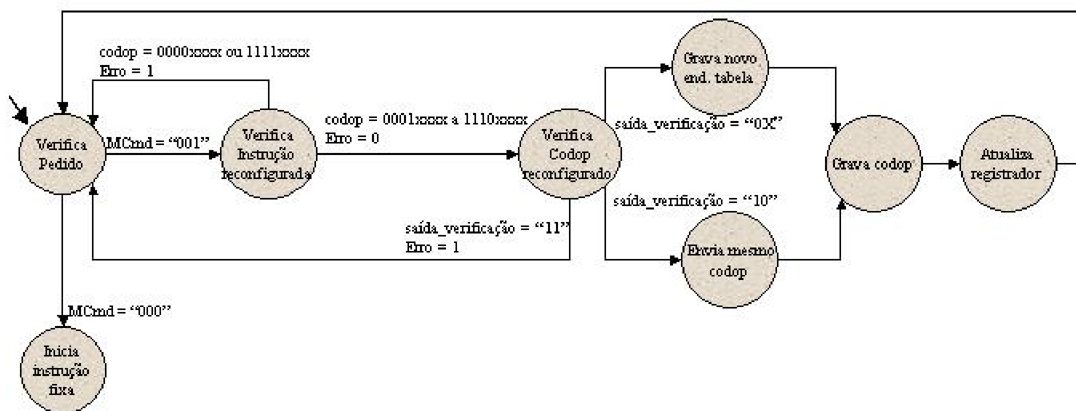


Figura 30: Diagrama de estados do processo de reconfiguração

A reconfiguração de uma instrução ocorre quando o processador recebe um sinal externo, via OCP, indicando um pedido de configuração. Em outras palavras, o processador deve receber do meio externo uma requisição de escrita indicada por $MCmd_ext =$ ”001”. Este processo ocorre no estado *verifica_pedido*. Ao entrar no processo de reconfiguração, o valor do Contador de Programa (PC) permanece com o valor atual até que a reconfiguração seja finalizada.

Ao receber o valor $MCmd_ext =$ ”001”, é confirmado o pedido de reconfiguração. No estado *Verifica Instrução Reconfigurada*, o processador recebe o valor do CodOp da instrução reconfigurada por meio do sinal (*MAddr_externo*), bem como o valor da palavra de reconfiguração mediante o sinal (*MData_externo*) contendo o primeiro estado da instrução. O processador envia o sinal $SCmmdAccept =' 1'$ indicando que aceitou o pedido de reconfiguração e armazena os valores do CodOp e da palavra de reconfiguração nos registradores *reg_codop* e *reg_dados*. A codificação da palavra de reconfiguração será vista na seção 3.9.

Neste mesmo estado é realizada uma verificação no valor do CodOp recebido afim de verificar se este valor corresponde a um CodOp reconfigurado válido. Os quatro primeiros bits

do CodOp são comparados com os valores ”0000” e ”1111”. Caso o resultado dessa comparação seja positiva, a instrução não pode ser configurada, porque todos os CodOps iniciados por ”0000” e ”1111” são reservados para as instruções presentes nos conjuntos de instruções fixos. Como resultado, é gerado um sinal de erro indicando que não é possível realizar esta operação, e a máquina de estados retorna ao estado anterior para receber um novo valor ou continuar a execução de instruções dos conjuntos fixos. Caso o resultado da comparação seja negativo, o CodOp recebido encontra-se na faixa de valores reservados para instruções reconfiguráveis, continuando a seqüência de execução.

O estado *verifica_codop_reconifigurado* é responsável por verificar se a palavra de reconfiguração atual faz parte de uma nova instrução ou de uma instrução previamente armazenada. Para realizar esta verificação, foi utilizado um componente de memória do tipo *lpm_ram*, denominado lmpram_verifica, com 256 posições e 2 bits de palavra. Cada endereço desta *lpm_ram* corresponde a um CodOp reconfigurado. O primeiro bit informa se existe ou não uma instrução reconfigurada utilizando este endereço (CodOp), enquanto que o segundo bit informa se a instrução está parcialmente reconfigurada, ou seja, se ainda devem ser incluídas novas operações, ou se a instrução já foi totalmente reconfigurada.

A Figura 31 mostra um exemplo de utilização desta *lpm_ram*, em que deseja-se reconfigurar uma instrução de CodOp 1AH contendo 3 operações. A cada nova palavra de reconfiguração (operação que compõe uma instrução), é realizada uma leitura nesta memória.

A Figura 31.a ilustra a primeira verificação nesta memória. A *Lpm_ram* recebe o valor do atual CodOp (1AH), utilizado como endereço da memória, e o valor do último bit da atual palavra de reconfiguração (exemplo: A839940H). Este bit, denominado *bit_end*, indica o término da instrução.

Nesta consulta, o valor existente na *lpm_ram* possui valor ”00”. Esta codificação indica que não existe nenhuma instrução utilizando este CodOp. Deste modo, é realizada uma operação de escrita nesta memória. O valor do primeiro bit é alterado para ’1’, indicando que a partir deste momento existe uma instrução para este CodOp, e o valor do segundo bit será o valor do *bit_end* da atual palavra de reconfiguração, que neste exemplo possui valor ’0’. Identificado que se trata de uma nova instrução, deverá ser iniciado o processo de armazenamento de um novo CodOp.

A Figura 31.b mostra uma segunda consulta a memória. Os valores recebidos pela tabela são o CodOp = 1AH e o *bit_end* da palavra de reconfiguração atual, que neste exemplo contém o valor 4C20040H. Ao buscar o valor da memória endereçada por 1AH, verifica-se que o valor recebido é igual a "10", que foi o valor armazenado durante a execução anterior.

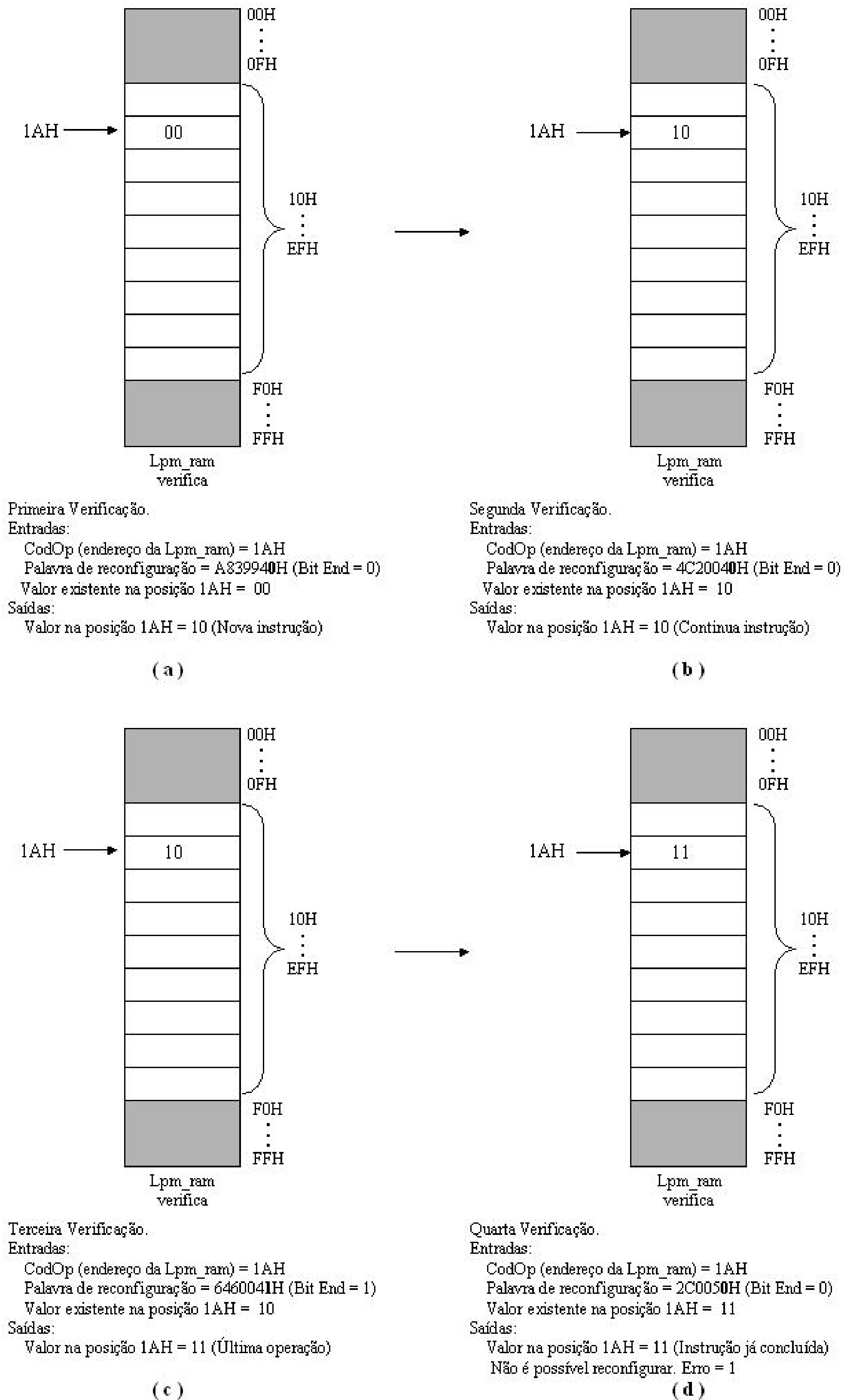


Figura 31: Exemplo de funcionamento da *Lpm_ram* de verificação. (a) nova instrução. (b) continuação da instrução. (c) término da instrução. (d) erro de reconfiguração.

Esta codificação indica que existe uma instrução utilizando este CodOp, mas ainda não se encontra concluída. Em outras palavras, indica que ainda existem mais palavras de reconfiguração (operações) a serem armazenadas. O Código armazenado na memória será novamente ”10”, em que ’1’ indica a presença de uma instrução para este CodOp e ’0’ indica que ainda deverão ser enviadas outras operações, pelo fato de que a palavra de reconfiguração atual também possui seu último bit com valor ’0’.

Se o valor do CodOp desta segunda consulta fosse diferente do utilizado na primeira consulta, o processador interpretará como se este CodOp seja uma nova instrução ao invés da segunda operação de uma mesma instrução. Isto ocasionará um erro na execução da instrução anteriormente armazenada. Após a primeira consulta, o valor armazenado na *lpm_ram* na posição 1AH recebeu o valor ”10”, ou seja, a instrução que utiliza o CodOp 1AH ainda não foi concluída. Se a unidade de reconfiguração receber outro CodOp (por exemplo 1BH) antes que o CodOp 1AH esteja concluído, não é possível continuar a seqüência de armazenamento do CodOp 1AH. Por este motivo, um novo CodOp só pode ser reconfigurado após todas as operações do CodOp anterior serem armazenadas.

A terceira consulta à memória é representado pela Figura 31.c. Neste exemplo, o endereço da memória continua sendo o valor do CodOp 1AH, pelo motivo de que anteriormente foi informado que a instrução ainda não estava concluída. A diferença deste exemplo para o anterior está no fato de que a nova palavra de reconfiguração (6460041H) possui o valor de *bit_end* igual a ’1’. O armazenamento na memória também é análogo ao realizado no exemplo (b), com a diferença de que o segundo bit armazenado conterá o valor ’1’, indicando que trata-se da última operação desta instrução.

Após o término de uma instrução, não é possível reconfigurar novamente o mesmo CodOp até que o processador seja reinicializado. Este procedimento é necessário para evitar o risco das novas operações sobrescreverem informações armazenadas anteriormente por outro CodOp e comprometer a execução de outras instruções reconfiguradas, fato que sera provado ainda nesta sessão.

O último exemplo, na Figura 31.d, mostra uma tentativa de reconfigurar o mesmo CodOp após a indicação de que a instrução está concluída. Neste caso, não importa o valor do *bit_end* contido na palavra de reconfiguração. Ao buscar a informação na memória e ser constatado que o segundo bit armazenado possui valor ’1’, um sinal de erro é enviado à entidade externa indicando que não foi possível reconfigurar a instrução utilizando este CodOp e a máquina de estados retorna ao estado de verificação de pedido de reconfiguração.

Assim, conclui-se que, ao ser inicializada a reconfiguração de um determinado CodOp,

obrigatoriamente todas as operações que compõem este CodOp devem ser armazenadas seqüencialmente até que seja indicado o sinal de término da instrução, e que cada CodOp pode ser reconfigurado apenas uma vez.

Após a verificação do CodOp, em se tratando de uma nova instrução, será executado o estado *Grava_novo_end_tabela*, responsável por armazenar a posição inicial da instrução na memória RAM de reconfiguração. Para realizar este processo, são utilizados um registrador de 8 bits, denominado *registrador de endereço da tabela* e outro componente do tipo "*lpm_ram*", denominado tabela de endereços, com 256 posições e 8 bits de palavra. A função deste registrador é a de conter o endereço inicial do próximo CodOp a ser reconfigurado, ou seja, o registrador atua como um ponteiro. Já a função desta *lpm_ram* é a de armazenar o endereço inicial da memória de reconfiguração de todos os CodOps já reconfigurados.

Por definição, ao se inicializar o computador, o registrador de endereço de tabela recebe o valor 00H. A partir deste ponto, a cada vez que o estado *Grava_novo_end_tabela* é executado, um sinal de escrita é enviado para a tabela de endereços, que recebe como endereço o valor do CodOp atual e como dados o valor de saída do registrador. A Figura 32 ilustra um exemplo desta operação.

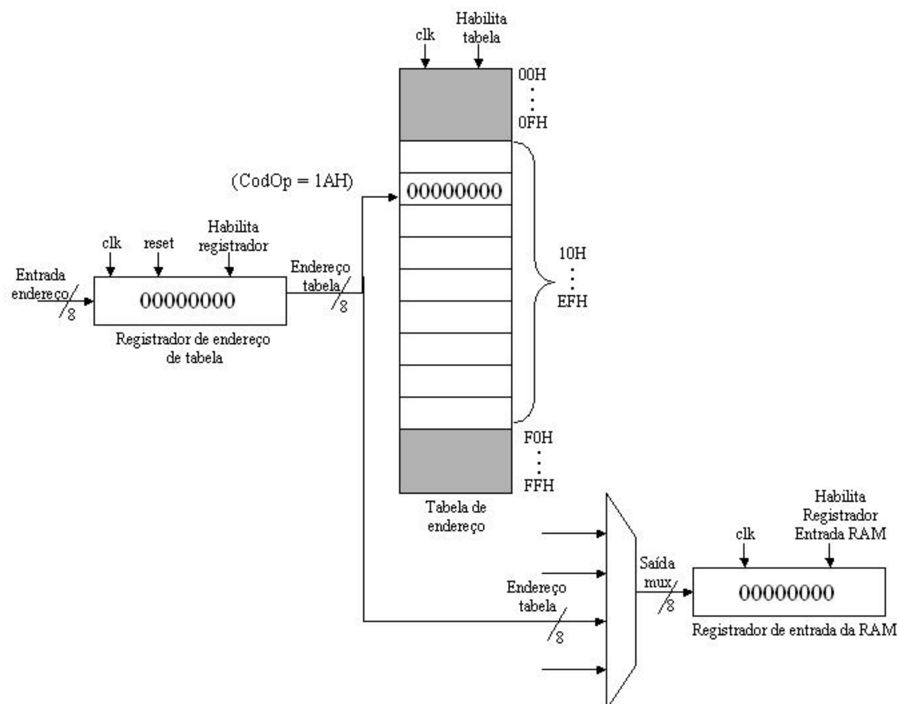


Figura 32: Exemplo de funcionamento da tabela de endereços para um novo CodOp.

Ainda no estado *Grava_novo_end_tabela*, o valor armazenado no registrador de endereço de tabela é enviado a um registrador de 8 bits denominado *registrador de entrada da RAM*.

A função deste registrador é a de armazenar o endereço a ser enviado para o barramento de endereços da RAM de reconfiguração.

Se o estado *verifica_codop_reconifigurado* informou que o CodOp atualmente reconfigurado já existe, então o próximo estado a ser executado é o estado nomeado *Envia_mesmo_CodOp*, por tratar-se de uma continuação do processo de reconfiguração de uma instrução. Este estado, ao contrário do anteriormente descrito, não executa operação de escrita na tabela, devido ao fato de que já encontra-se armazenado na tabela o valor do endereço inicial desta instrução, processo este realizado na execução do estado *Grava_novo_end_tabela*.

O processo de armazenamento do registrador de entrada da RAM é realizado da mesma forma realizada pelo estado *Grava_novo_end_tabela*, ou seja, a unidade de controle envia um sinal de ativação de escrita no registrador, ao mesmo tempo em que este recebe o conteúdo do registrador de endereço da tabela, como pode ser visto na Figura 33.

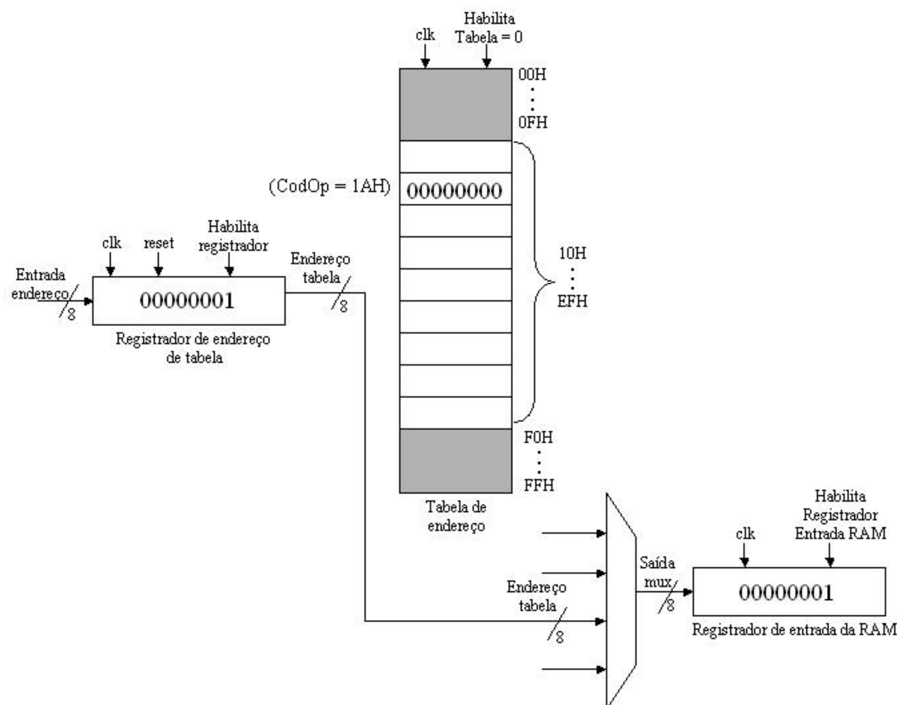


Figura 33: Exemplo de funcionamento da tabela de endereços para um mesmo CodOp.

Ambos os estados *Grava_novo_end_tabela* e *Envia_mesmo_CodOp* terão como seguinte estado *Grava_CodOp*. Este estado possui como única função habilitar o processo de escrita na memória RAM de reconfiguração. Esta RAM foi implementada também utilizando um componente "*lpm_ram*", com 8 bits de endereço e 28 bits de palavra. A RAM recebe no seu barramento de endereços o valor contido no registrador de entrada da RAM e no barramento de dados o valor da palavra de reconfiguração atual, que estava armazenada em *reg_dados*.

O estado final do processo de reconfiguração é denominado *Atualiza_registrador*. Neste estado, o registrador de endereço de tabela assume o valor de uma posição seguinte à que encontra-se armazenada no registrador de entrada da RAM. Como visto nos estados anteriores, o valor contido neste registrador pode ser utilizado para indicar o início de uma nova instrução ou para continuar a seqüência de armazenamento de operações de um mesmo CodOp na RAM de reconfiguração.

As Figuras a seguir apresentam um exemplo completo do processo de reconfiguração de uma instrução contendo duas operações. A Figura 34 ilustra a seqüência de armazenamento de um novo CodOp 1AH contendo a palavra de reconfiguração 1A00000H.

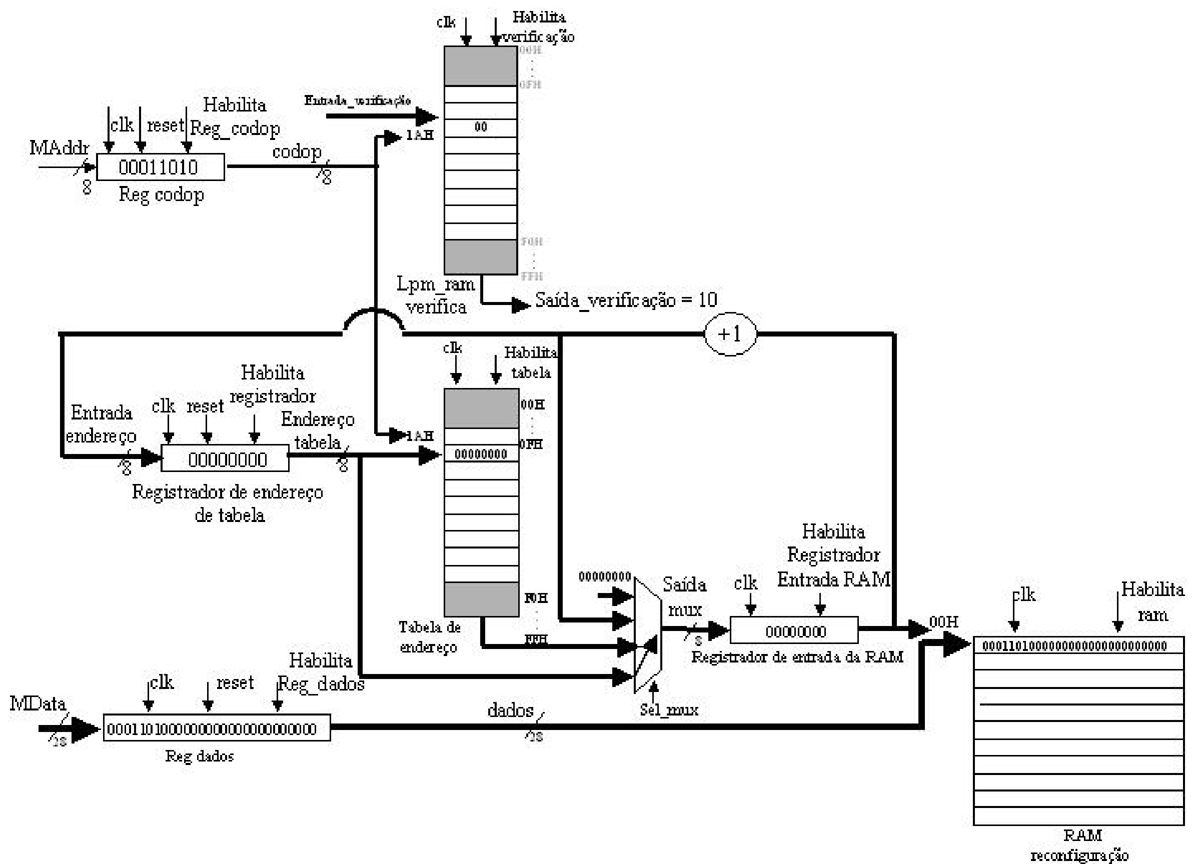


Figura 34: Processo de reconfiguração de um novo CodOp

A Figura 35 ilustra a seqüência de armazenamento da segunda operação da instrução de CodOp 1AH contendo a palavra de reconfiguração 3A802C1H.

Como citado anteriormente, um mesmo CodOp só poderá ser novamente reconfigurado após o reinício do processador. Supondo que um CodOp X foi reconfigurado contendo quatro operações, de modo que a instrução se inicia no endereço ”00*H*” e termina no endereço ”03*H*”.

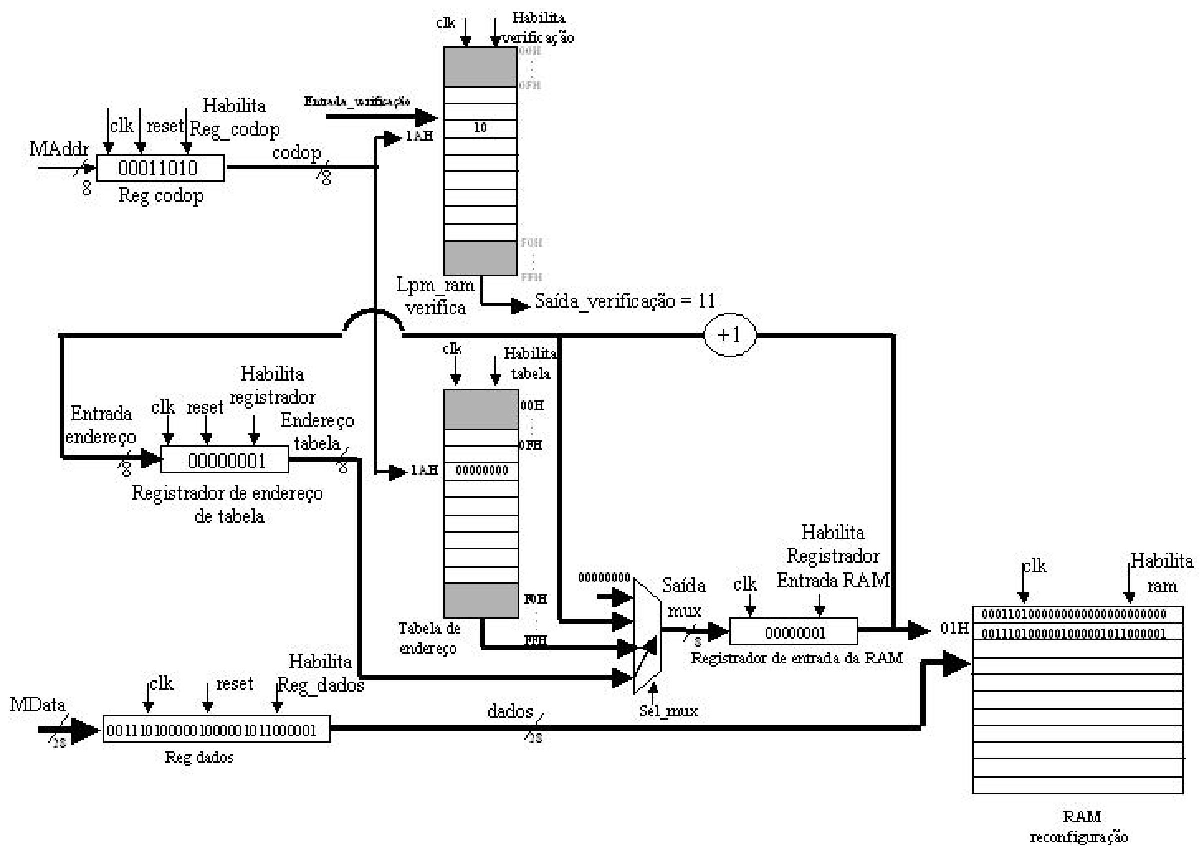


Figura 35: Continuação do processo de reconfiguração de um CodOp

Após o término deste processo, foi reconfigurado um CodOp Y contendo duas operações. Este CodOp Y terá como endereço inicial o valor imediatamente seguinte ao utilizado para armazenar a última operação do CodOp X, ou seja, o endereço inicial de CodOp Y será o valor ”04*H*”. Se posteriormente fosse desejado reconfigurar novamente o CodOp X, contendo agora cinco operações, a quinta operação armazenada ultrapassaria o limite de posições de memória utilizado anteriormente para o CodOp X, ocupando assim o espaço utilizado atualmente pelo CodOp Y. Este processo tornaria a execução do CodOp Y completamente diferente da que foi originalmente descrita.

3.8.2 Processo de Execução de Instruções Reconfiguradas

O processo de execução de instruções reconfiguradas é realizado de maneira análoga ao processo de execução de instruções fixas, visto na seção 3.7. Desse modo, serão detalhadas apenas as etapas que representam este processo. A Figura 36 mostra o diagrama de estados contendo as etapas deste processo.

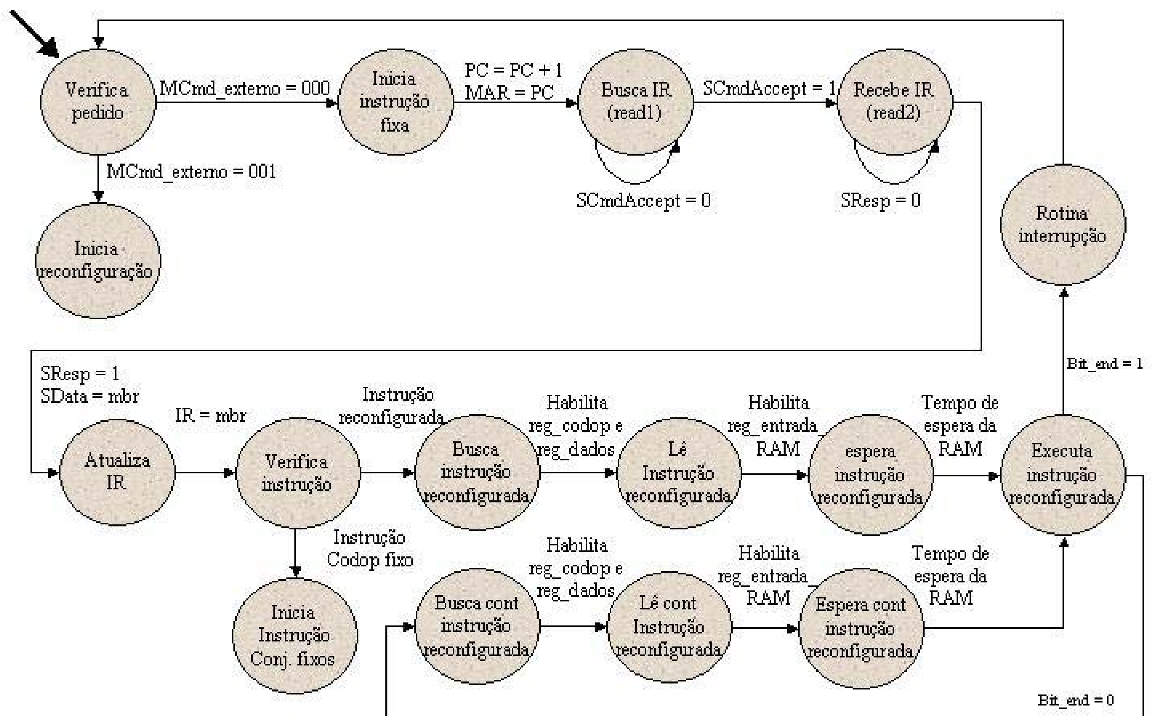


Figura 36: Diagrama de estados do processo de execução de instruções reconfiguradas

A partir do estado *Verifica_instrução*, caso os quatro primeiros bits de CodOp estejam entre ”0001” e ”1110”, será iniciado o processo de execução de uma instrução reconfigurada. O estado seguinte é denominado *Busca instrução reconfigurada*. Neste estado, o registrador *reg_codop* recebe o valor do CodOp solicitado e fará a busca do endereço inicial da instrução na tabela de endereços. No estado *Lê instrução reconfigurada* o registrador de entrada da RAM recebe o valor contido na tabela de endereço, o qual será o barramento de endereço da memória RAM de reconfiguração. O estado *Espera instrução reconfigurada* consiste em um estado de espera enquanto é realizada a busca nesta memória. Este estado é necessário devido ao fato de que a memória de reconfiguração necessita de 3 ciclos de relógio para disponibilizar o resultado. Como cada estado possui 4 ciclos de relógio, somente é necessário adicionar um ciclo de espera.

O próximo estado é denominado *Executa instrução reconfigurada*. Neste estado, o valor da palavra de reconfiguração armazenado na memória de reconfiguração é recebido pela Unidade de Controle, que posteriormente a envia para a Unidade Operativa, de maneira similar a realizada na execução de instruções fixas. A codificação da palavra de reconfiguração será vista na seção seguinte.

Ainda neste estado, o *bit_end* da palavra de reconfiguração é verificado. Caso este seja igual

a '1', o estado seguinte será o estado INT, indicando o término da instrução. Caso contrário, serão executados os estados nomeados *Busca cont instrução reconfigurável*, *Lê cont instrução reconfigurável* e *Espera cont instrução reconfigurável*. A funcionalidade destas instruções é praticamente idêntica aos estados *Busca instrução reconfigurável*, *Lê instrução reconfigurável* e *Espera instrução reconfigurável*, com exceção de que o registrador de entrada da RAM não recebe o valor contido na tabela, e sim o valor de saída do incrementador, fazendo com que a memória acesse uma posição seguinte à executada anteriormente. As figuras seguintes mostram um exemplo de execução da instrução reconfigurada 1AH. A Figura 37 ilustra a execução da primeira operação de uma instrução reconfigurável.

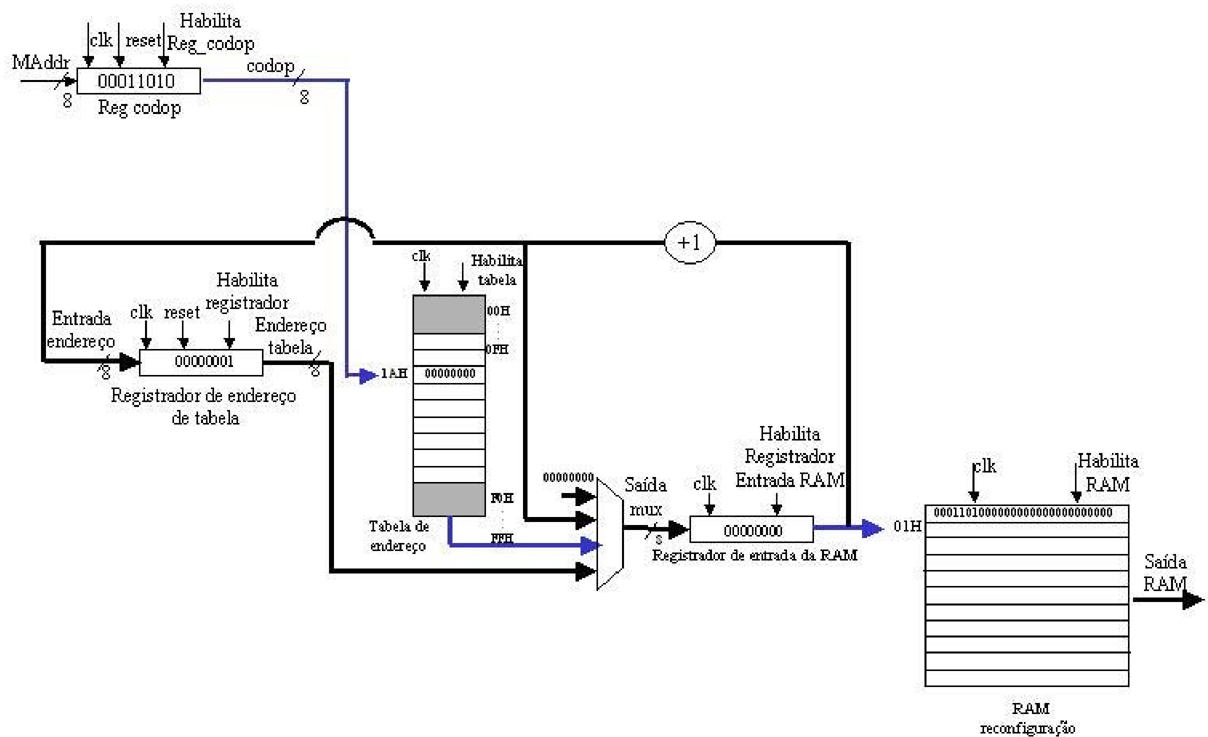


Figura 37: Processo de reconfiguração de um novo CodOp

A Figura 38 mostra a execução das demais operações da instrução reconfigurável até que o valor do *bit_end* seja igual a '1'.

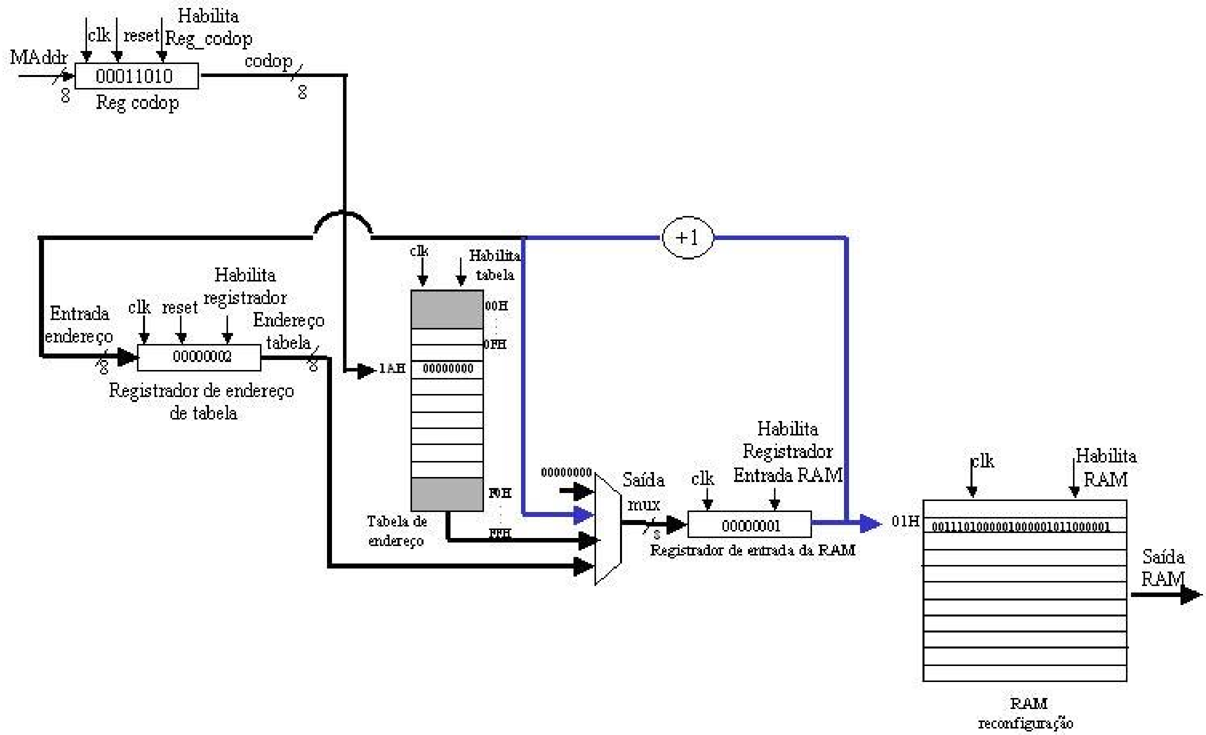


Figura 38: Continuação do processo de reconfiguração de um CodOp

3.9 Reconfiguração de uma instrução

Como visto anteriormente, para reconfigurar uma instrução, é necessário informar o valor do endereço (CodOp) e dos dados (palavra de reconfiguração). A palavra de configuração contém 28 sinais, dispostos conforme a Figura 39, responsáveis por manipular os blocos funcionais da unidade operativa. Este processo é similar ao realizado para a execução de instruções presentes nos conjuntos de instruções fixos, mas com algumas modificações.

6	2	2	3	6	1	1	1	2	1	1	1	1
Formato	AMUX	BMUX	ULA	SH	MBR	MAR	ENC	MUX_MAR	MUX_MBR	WR	RD	Bit_end
28												0

Figura 39: Disposição dos bits da palavra de reconfiguração

A configuração da palavra de configuração é apresentada na Tabela 4. A manipulação destes bits permite a criação e/ou combinação de qualquer instrução que seja possível para o

processador realizá-la.

Tabela 4: Composição da palavra de configuração

Nome	**Função**	**Valores**
Formato	Define formato da instrução	Vide Apêndice F
AMUX	Controla o operando esquerdo da ULA	00 = Bar A 01 = Bar C 10 = AMASK 11 = MBR
BMUX	Controla o operando direito da ULA	00 = Bar B 01 = +1 10 = IR 11 = 00H & IR(18-0)
ULA	Seleciona operação da ULA	000 = A 001 = A AND B 010 = NOT(A) 011 = NOT(B) 100 = A + B 101 = A - B 110 = A * B 111 = A OR B
SH	Deslocador	Bit 1: Direção (esq/dir) Bits 2 a 6: deslocamento
MBR	Carrega MBR	0 = não carrega 1 = carrega
MAR	Carrega MAR	0 = não carrega 1 = carrega
ENC	Armazena resultado no conjunto de registradores	0 = não armazena 1 = armazena
MUX_MAR	Entrada de MAR	00 = PC 01 = IR 10 = *Latch* B 11 = Bar C
MUX_MBR	Entrada de MBR	0 = PC 1 = Bar C
WR	Estado de escrita na RAM	0 = não ocorre escrita 1 = ocorre escrita
RD	Estado de leitura na RAM	0 = não ocorre leitura 1 = ocorre leitura
BIT_END	Final da instrução	0 = não concluída 1 = concluída

As operações podem conter de 1 a 4 registradores, ou seja podem assumir os formatos de ”OPA” a ”OPC, A, B, D”. Dependendo do formato utilizado e da quantidade de operações que compõe a instrução, um registrador utilizado como destino na operação ’1’ pode ser utilizado como entrada na operação ’2’. O Apêndice F apresenta todos os formatos possíveis de serem

utilizados na palavra de reconfiguração.

Analisando o conjunto de instrução 2, constata-se que para realizar uma operação de incremento de um valor X, é necessário primeiramente que um dos 32 registradores de trabalho receba o valor '1', para posteriormente ser adicionado ao valor de X. E para que um registrador receba o valor '1', primeiramente o acumulador irá receber este valor para que depois o conteúdo do acumulador seja movido para o registrador desejado. A partir destas informações citadas, conclui-se que para executar o incremento de um registrador são necessárias 3 instruções.

Através do processo de reconfiguração de uma instrução, é possível criar a instrução **INC R**, em que deseja-se realizar a operação R = R + 1. Para criar esta instrução, é necessário definir o CodOp da instrução e as operações necessárias, ou seja, configurar as palavras de reconfiguração da instrução.

A primeira ação a ser tomada na definição da palavra de reconfiguração é quanto ao formato da operação. Para que ocorra R = R + 1, o valor de "*R*" deve estar presente no barramento de Entrada A e também no barramento de resultado C. Verificando a tabela de codificação (Apêndice F), é possível utilizar qualquer formato que esteja na forma $A = A\ op\ X$, visto que não é necessário informar o segundo operando. Para este exemplo, será utilizado o formato "000000"(A = A op A).

Analisando a unidade operativa do processador, para que a ULA receba como entrada o barramento A, o valor de *AMux* deverá ser "00". Pelo fato de uma das entradas de *BMux* estar definida como o valor constante '1', não é necessário que seja enviado um valor no barramento B, pois somente é necessário que *BMux* assuma o valor "01", selecionando a constante '1' como a segunda entrada da ULA.

Como a operação desejada é uma soma, os bits referentes ao controle da ULA deverão assumir os valores "100". Como não é necessário efetuar nenhum deslocamento no resultado, os bits referentes a SH deverão ser iguais a '0'.

A operação desejada não necessita de nenhum acesso a memória, por isto todos os bits referentes a MAR, MBR, WR e RD assumem valor '0'. Para que o resultado da operação seja armazenado no registrador "R", é necessário que o bit ENC seja igual a '1'.

Finalizando, como apenas a operação R = R + 1 é necessária para efetuar a instrução INC R, o valor de bit_end deve ser ativado. Assim, a palavra de reconfiguração deverá conter estes valores na seqüência de campos tal como foi visto na figura 39: "000000000110000000000001000001" (060041H). O Capítulo 4 contém um exemplo de simulação da criação e utilização desta ins-

trução reconfigurada.

O segundo exemplo ilustra a reconfiguração de uma instrução do tipo MAC (Multiplica e Acumula). Será criada a instrução **MAC D, A, B, C,** em que $D = A + (B * C)$. Como visto na seção 3.3, esta instrução encontra-se no formato 4 (OP 5, 5, 5, 5, 4). A instrução desejada pode ser decomposta em duas operações: $D = B * C$ e $D = D + A$, de modo que cada uma destas operações será armazenada na unidade de reconfiguração, utilizando-se o mesmo CodOp, que neste caso será definido como 11H.

Para definir a primeira palavra de reconfiguração, primeiramente deve-se definir o formato da operação atual. Para a operação $D = B * C$, o primeiro operando deverá ser enviado ao barramento C, o terceiro operando ao barramento A e o quarto operando ao barramento B. Isto resulta no formato de operação *Bar C = Bar B op Aux*, o que corresponde a codificação "101010". O formato final desta palavra de reconfiguração é "1010100000110000000001000000", ou A830040H.

A segunda operação ($D = D + A$) apresenta o primeiro operando como registrador destino (Bar C) e como primeiro registrador de entrada (Bar A), enquanto que o segundo operando representa o segundo registrador de entrada (Bar B). A codificação do formato BAR C = BAR C op BAR A é igual ao valor "010011". Ao final da análise dos bits restantes da segunda e última palavra de reconfiguração, esta assume valor "0100110000100000000001000001", ou 4C20041H.

Após o término do processo de reconfiguração, o processador deve continuar o processo de execução de instruções normalmente. Se em determinado momento o processador receber uma instrução na qual IR seja igual a "11390A60H", a consulta ao valor 11H (00010001B) fará com que o processador inicie o processo de execução de instrução reconfigurável. Analisando o valor de IR de acordo com o formato (8-5-5-5-5- 4), tem-se "00010001 00111 00100 00101 00110 0000". Isto representa a instrução MAC RD, RA, RB, RC.

4 SIMULAÇÕES E RESULTADOS

Este capítulo apresenta alguns resultados quanto à implementação das unidades funcionais constituintes do processador e simulações do comportamento e funcionalidade dos mesmos. O processador foi descrito utilizando-se a linguagem de descrição *VHDL*(VHSIC *Hardware Description Language*[IEEE, 1993], através da ferramenta de desenvolvimento Quartus II, da Altera[Altera, 2005].

4.1 Resultados da Compilação

O processador foi compilado utilizando o dispositivo EP1S10F484C5, da família de dispositivos Stratix, da Altera. Este dispositivo foi escolhido em virtude da possibilidade de se estimar o consumo de potência da aplicação.

O resultado da compilação mostra que foram utilizados 3.798 elementos lógicos deste dispositivo, o que corresponde a 35% da área disponível do dispositivo (10.570 elementos lógicos). Foram utilizados 154 pinos dos 336 disponíveis (45% do total) e 9.728 bits de memória para a implementação dos componentes *lpm_ram*, o que corresponde a apenas 1% da capacidade do processador (920.448). Foram obtidas as seguintes freqüências para cada subciclo de relógio:

- CLK_0: 90.20 MHz (período = 11.086 ns)
- CLK_1: 61.07 MHz (período = 16.374 ns)
- CLK_2: 290.87 MHz (período = 3.438 ns)
- CLK_3: 136.54 MHz (período = 7.324 ns)

A freqüência de mais alto valor (CLK_1) é devido ao tempo gasto para os *latches* A e B da unidade operativa receberem os valores armazenados na memória de rascunho do processador. Já a freqüência de mais alto valor (CLK_2) é conseqüência da utilização de mega funções disponíveis no Quartus para efetuar as operações aritméticas de adição, subtração e multiplicação

em apenas um ciclo. Em virtude dos valores obtidos, conclui-se que o processador possui uma freqüência máxima de operação de 61MHz.

A Tabela 5 mostra a quantidade de elementos lógicos (EL), de registradores e de bits de memória (BM) utilizados em cada componente do processador. O componente que mais consome elementos lógicos é a memória de rascunho, pelo motivo da sua implementação ter sido realizada utilizando um conjunto de 32 registradores. A área ocupada pela memória de rascunho poderia ser reduzida caso este componente fosse implementado utilizando *lpm_rams*, como foi realizado com a memória de reconfiguração. Este processo aumentaria a freqüência maxima de operação e diminuiria a área do processador. No entanto, acarretaria alterações na temporização do processador e, conseqüentemente, uma reformulação em toda a máquina de estados.

Tabela 5: Valores obtidos na compilação do processador

Unidade	**Componente**	**E.L**	**Registradores**	**B.M**
Unidade de controle		1504	498	0
Unidade Operativa		2234	1120	0
	AMUX	32	0	0
	BMUX	19	0	0
	Latch A	32	32	0
	Latch B	32	32	0
	Mar	32	32	0
	Mbr	32	32	0
	Mux_mar	26	0	0
	Memória de rascunho	1353	992	0
	Deslocador	406	0	0
	Ula	270	0	0
Unidade de reconfiguração		60	52	9728
	registrador de CodOp	8	8	0
	registrador de dados	28	28	0
	registrador de endereço RAM	8	8	0
	registrador de endereço tabela	8	8	0
	multiplexador 4_1	8	0	0
	Lpm_Ram verificação	0	0	512
	Lpm_Ram tabela	0	0	2048
	Lpm_Ram dados (Memória Rec)	0	0	7168
Total		3798	1670	9728

4.1.1 Análise de custo dos conjuntos de instruções

Para verificar o impacto do acréscimo de um conjunto de instrução, foram realizadas compilações utilizando cada conjunto separadamente e combinações entre estes. Em todas as compilações dos conjuntos, foi utilizado o dispositivo EP1S20F780C5 da família de dispositivos Stratix da Altera. Foram analisados o total de elementos lógicos, a quantidade de estados necessários

apenas para a execução de instruções fixas, excluindo os estados iniciais e de interrupção, e o total de estados do conjunto. A Tabela 6 apresenta os resultados.

Tabela 6: Valores obtidos para cada conjunto de instrução

Conjunto de instrução	**E. L**	**Estados de execução**	**Total de estados**	**% área do dispositivo**
Conjunto 1	822	59	90	3%
Conjunto 2	828	53	84	3%
Conjunto 3	909	87	118	4%
Conjuntos 1 e 2	1023	64	95	5%
Conjuntos 1 e 3	1081	87	118	6%
Conjuntos 2 e 3	1175	87	118	7%
Conjuntos 1, 2 e 3	1380	87	118	7%

Com base nestes resultados, conclui-se que a diferença entre o menor conjunto de instrução e o conjunto de instrução completo é de 558 elementos lógicos.

4.2 Simulações

Nesta seção serão vistos algumas simulações do comportamento do processador na execução de instruções fixas e reconfiguráveis e durante o processo de reconfiguração de uma ou mais instruções.

4.2.1 Simulação da alternância entre os conjuntos de instruções

Esta simulação mostra a execução da instrução de CodOp 0DH por cada um dos conjuntos implementados. A Figura 40 apresenta uma visão geral do processo de simulação desde a inicialização do processador até a execução da última instrução desejada, com um tempo de execução de aproximadamente $7,36\mu s$. As Figuras 41 a 48 mostram maiores detalhes deste processo.

A Figura 41 mostra a execução da instrução *muda_modo*. Sem a execução desta instrução, o processador não executará nenhum comando de alteração de um conjunto de instrução para outro. Esta figura também apresenta o comportamento do processador durante a seqüência de estados iniciais para a execução de uma instrução fixa, como visto na seção 3.7. As setas indicativas da figura mostram os principais eventos que ocorrem durante os estados iniciais de uma instrução. Todas as instruções fixas possuem o mesmo padrão de comportamento, alterando-se apenas os valores de PC, IR e SData.

Durante esta simulação o acumulador recebe alguns valores aleatórios através de instruções LOCO para realizar as operações. Na Figura 42, o acumulador recebe o valor "1AH", mediante

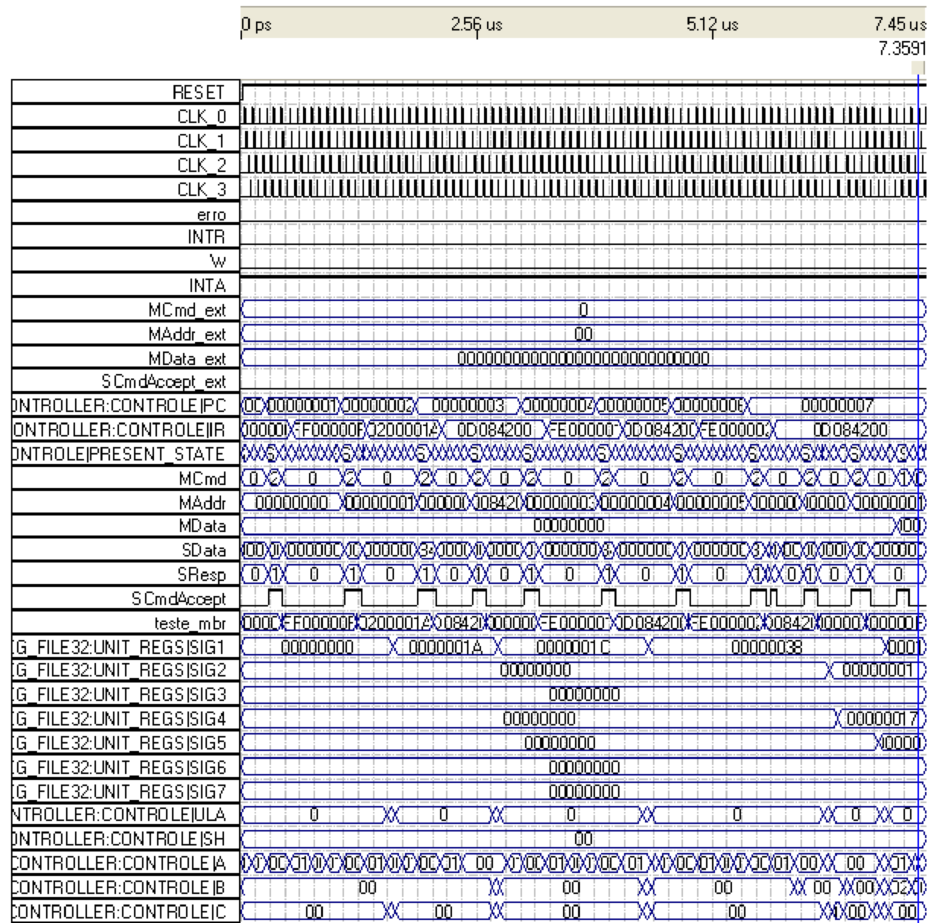


Figura 40: Visão geral da simulação de alternância de conjuntos de instruções

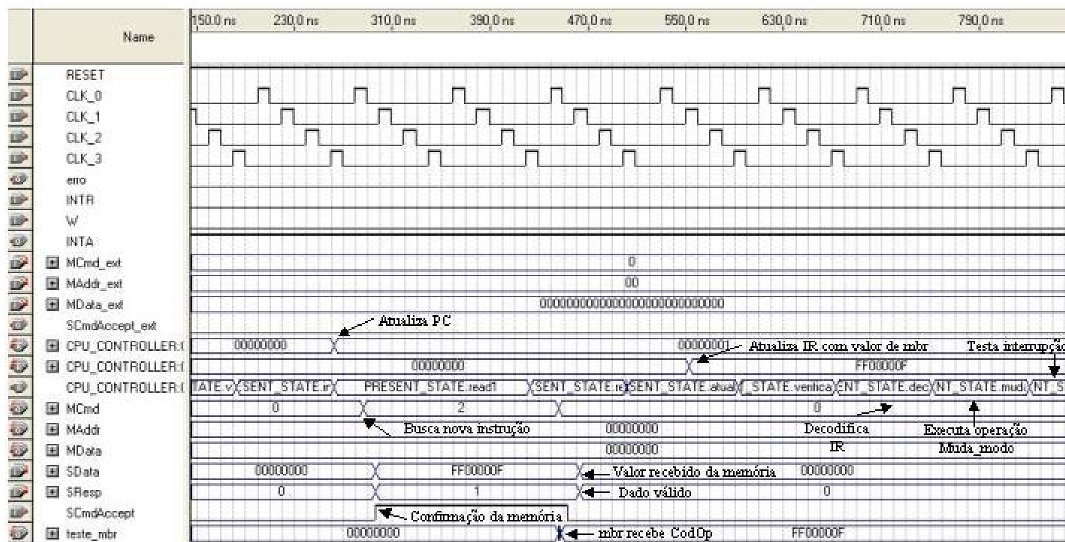


Figura 41: Simulação da instrução muda_modo

a instrução LOCO 1AH.

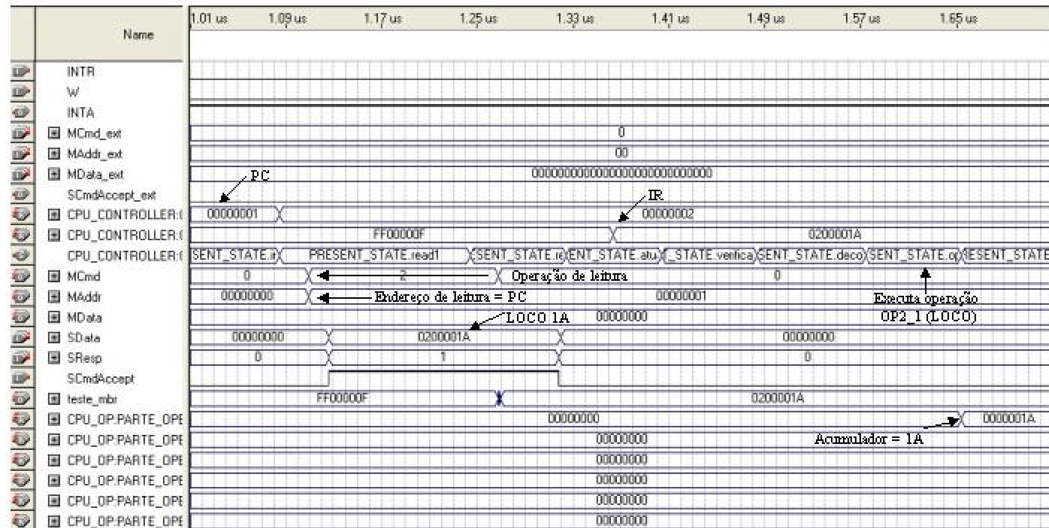


Figura 42: Simulação da instrução LOCO 1AH

Após a execução de LOCO 1AH, a instrução seguinte a ser executada é a instrução IR = 0D084200H. Como até o presente momento não houve nenhuma instrução para selecionar um conjunto de instrução, então será utilizado o conjunto de instrução ’1’. O CodOp 0DH do conjunto 1 refere-se a instrução **ADD X**, em que o acumulador recebe o conteúdo de memória endereçado por X. Esta instrução pertence ao formato de instrução ’1’, em que os 8 primeiros bits (00001101B - 0DH) representam o CodOp da instrução e os 24 bits restantes (000010000100001000000000B - 084200H) representam o valor do endereço de memória a ser consultado.

A Figura 43 mostra apenas os estados de execução desta instrução. O estado OP13_1 indica uma operação de leitura na memória, em que o valor a ser recebido será o conteúdo da célula de memória da posição 00084200H. Após a leitura deste valor, o estado OP13_2 é responsável por realizar uma adição entre o valor do acumulador e o valor de MBR.

Em seguida ocorre a execução da instrução IR = FE000001H. Esta instrução seleciona qual conjunto de instrução será utilizado a partir da próxima instrução. Os dois últimos bits de IR ("01"B) indicam que será selecionado o conjunto de instrução ’2’. A execução desta instrução é apresentada na Figura 44.

Na Figura 45, novamente o processador executa a instrução IR = 0D084200H. A diferença está no fato de que agora o CodOp 0DH resultará na execução da instrução **ADDR Ra, Rx, Ry**. Esta instrução pertence ao formato de instrução ’2’, em que os 8 primeiros bits (00001101B - 0DH) representam o CodOp da instrução e os 24 bits restantes (00001 00001

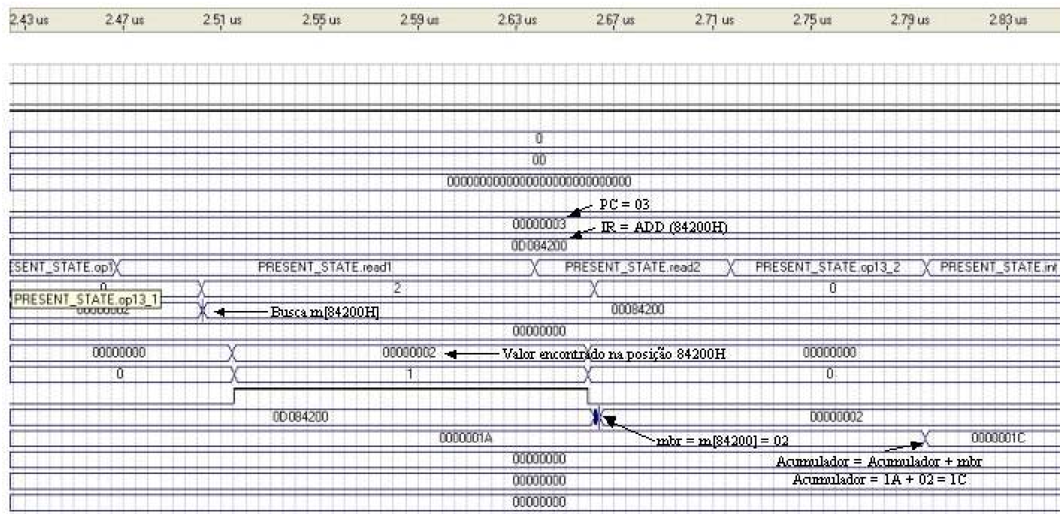


Figura 43: Simulação da instrução IR = 0D084200H do conjunto 1

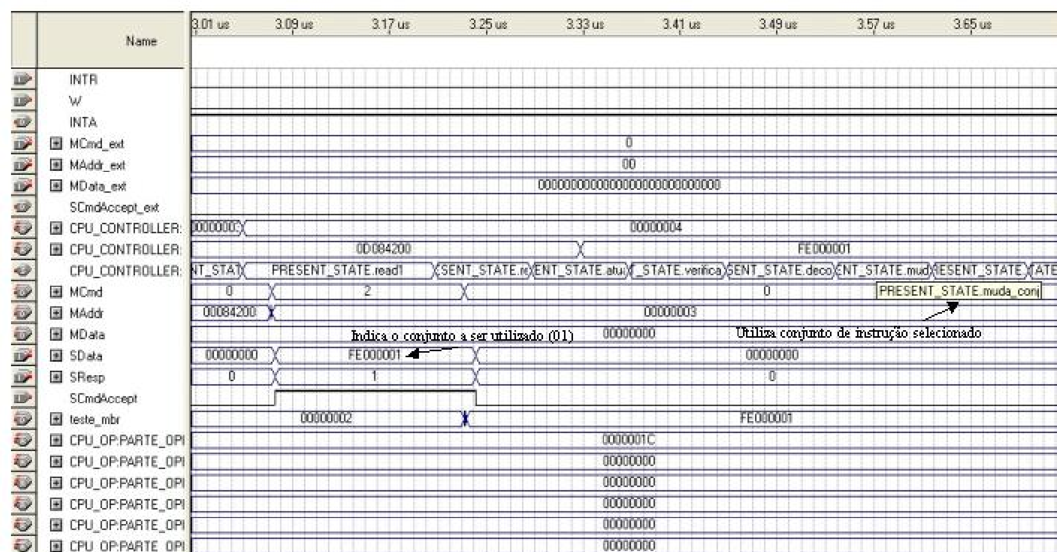


Figura 44: Simulação da alteração do conjunto de instrução 1 para 2

00001 000000000B - 084200H) são interpretados na forma 5-5-5-9, que representam, respectivamente, a codificação dos registradores Ra, Rx e Ry.

Ao contrário da execução apresentada na Figura 43, esta instrução contém apenas uma operação (OP13_1), em que o Ra (Acumulador) recebe o resultado da soma entre Rx (Acumulador) e Ry (Acumulador).

Na seqüência, a instrução a ser executada é IR = FE000002H. O processo de execução desta instrução (muda_conjunto) é semelhante ao anteriormente descrito, com a diferença de que esta instrução seleciona o conjunto de instrução ’3’, como pode ser visto na Figura 46.

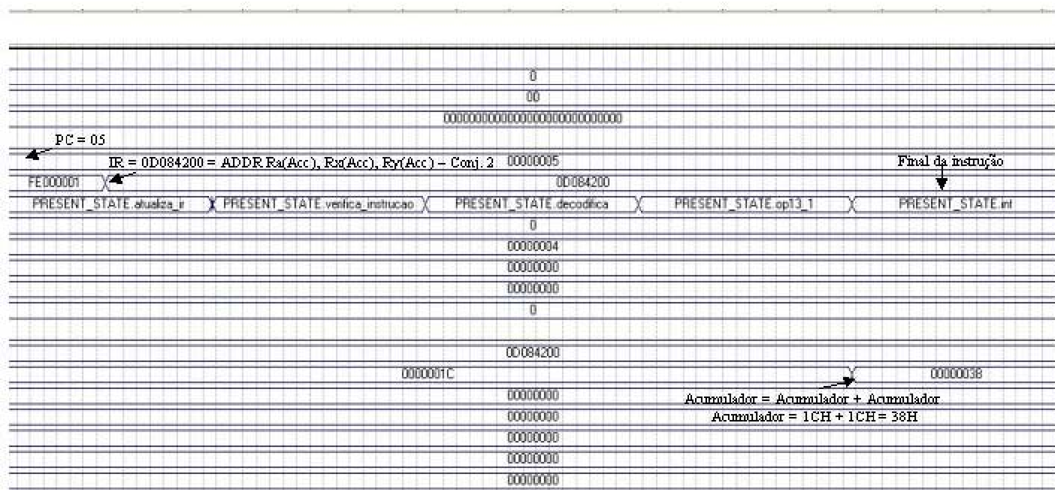


Figura 45: Simulação da instrução IR = 0D084200H do conjunto 2

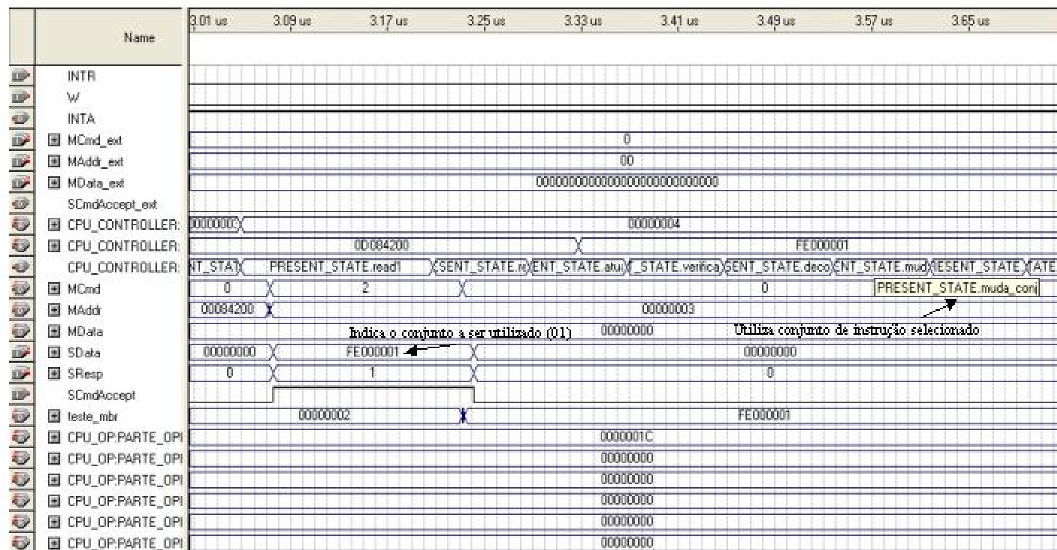


Figura 46: Simulação da alteração do conjunto de instrução 2 para 3

A última parte desta simulação apresenta novamente a execução de IR = 0D084200H. Este CodOp dará início a execução da instrução **ADDp**, em que o processador recebe dois valores armazenados na pilha, efetua a adição entre estes valores e empilha o resultado final. Esta instrução também encontra-se no formato ’1’, com a diferença de que os 24 bits restantes não representam nenhum valor utilizado durante esta execução.

A Figura 47 apresenta os primeiros estados desta instrução. As setas indicativas mostram os eventos ocorridos no processo de busca de um valor na pilha. Neste exemplo, foi solicitada uma busca na pilha na posição "00H", obtendo-se o valor "17H", sendo este valor armazenado no Registrador A.

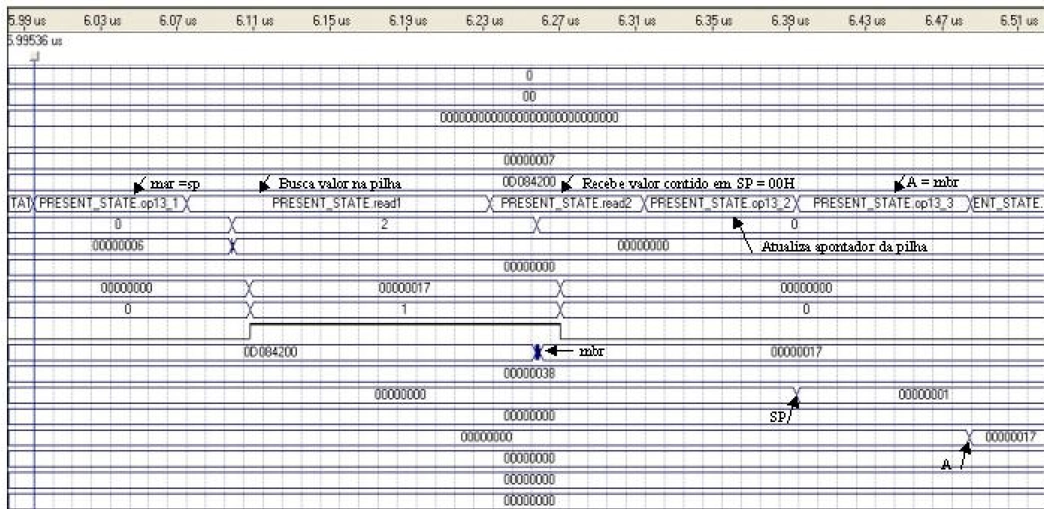


Figura 47: Simulação da instrução IR = 0D084200H do conjunto 3 (a)

Já a Figura 48 mostra um processo semelhante, em que é realizada a busca na pilha na posição "01H"e recebendo o valor "F1H", posteriormente armazenado no Registrador B. As últimas operações representam a adição dos valores contidos em A e B, e o armazenamento do resultado na pilha, por meio de uma operação de escrita.

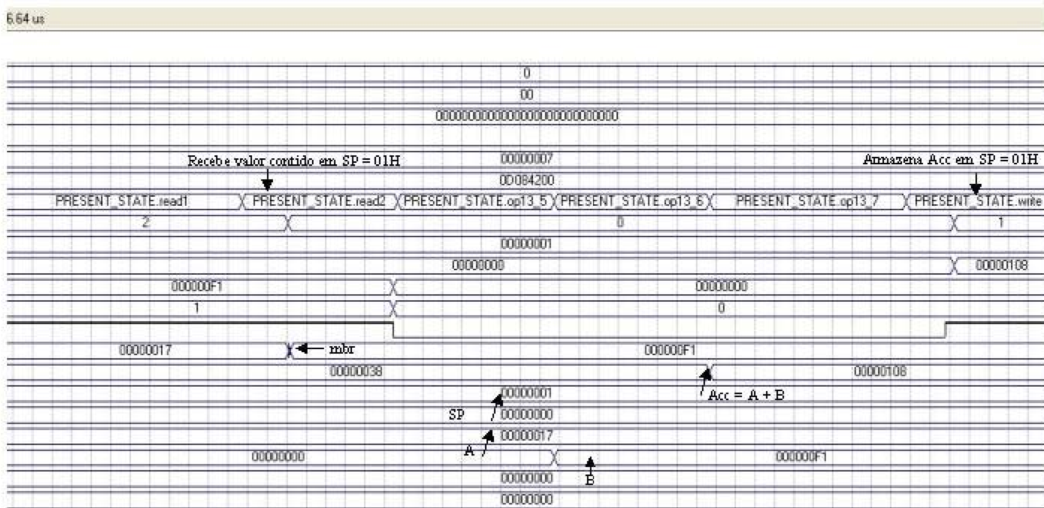


Figura 48: Simulação da instrução IR = 0D084200H do conjunto 3 (b)

A Figura 49 mostra o diagrama de estados que confirma o resultado das simulações realizadas, em que o mesmo CodOp pode ser destinado a três operações distintas.

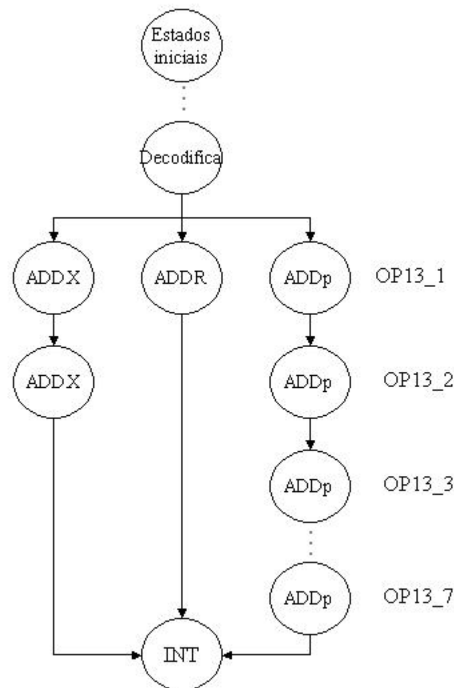


Figura 49: Diagrama de estados do CodOp 0D084200H

4.2.2 Simulação da criação de uma instrução: MAC D, A, B, C

Esta simulação permite verificar o comportamento do processador durante o processo de reconfiguração de uma instrução contendo mais de uma operação e sua execução. Como exemplo, será criada a instrução MAC D, A, B, C. O processo de criação desta instrução encontra-se na página 58 da sessão 3.9. A instrução MAC é composta por duas operações (A830040H e 4C20041H). Cada operação de uma instrução reconfigurada necessita de 6 estados para ser efetuada.

Esta primeira tela de simulação (Figura 50) apresenta o processo de reconfiguração desta instrução. O período utilizado é de $20ns$ para cada subciclo de relógio, o que resulta em um total de $80ns$ para cada instrução.

A Figura 51 mostra o funcionamento da unidade de reconfiguração durante a criação de MAC. Os locais indicados na figura mostram os principais eventos deste processo. O tempo total gasto para reconfigurar as duas operações foi de $1.06\mu s$.

A Figura 52 mostra o processo de inicialização das variáveis A, B e C. Para definir o valor de cada variável, é necessário executar a instrução LOCO informando o valor desejado, para posteriormente mover o valor do acumulador para o registrador. Como não existe a instrução MOV, é utilizada a instrução ADDR R, Acc, 0, em que o registrador alvo recebe o conteúdo do

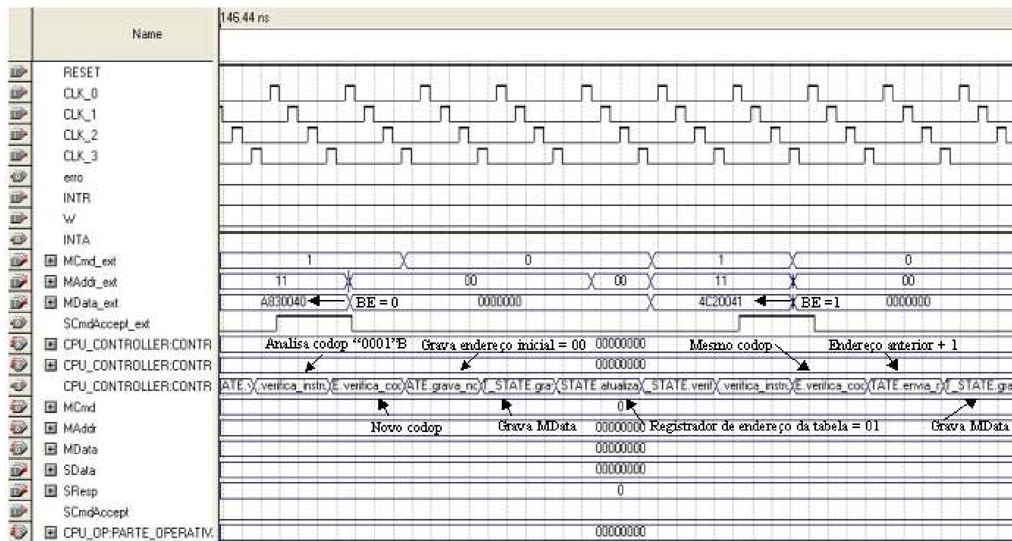


Figura 50: Simulação do processo de reconfiguração (b)

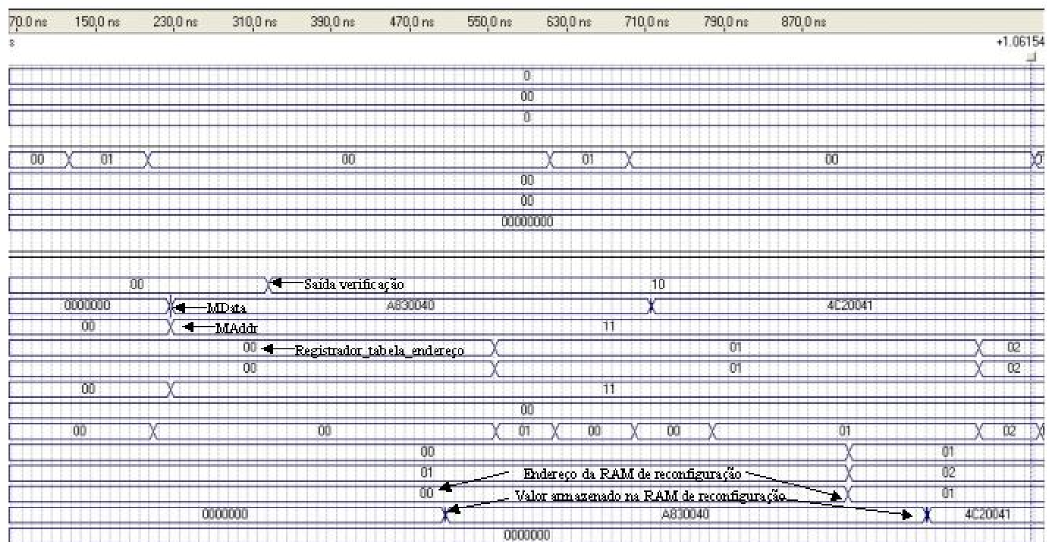


Figura 51: Simulação do processo de reconfiguração (a)

registrador acumulador somado com "00000000"H.

A Figura 53 ilustra a execução da instrução MAC. O valor IR = 11390A60H (0001 0001 00111 00100 00101 00110 0000B) informa quais são os 4 registradores utilizados nesta operação. Esta operação necessita de 15 estados para ser efetuada, sendo 10 estados para a primeira operação (incluindo os 6 estados iniciais de busca e verificação) e 5 estados para a segunda operação (incluindo o estado de verificação de interrupção).

A aplicação foi executada em um total de 99 estados, resultando em tempo total de $9,32\mu s$.

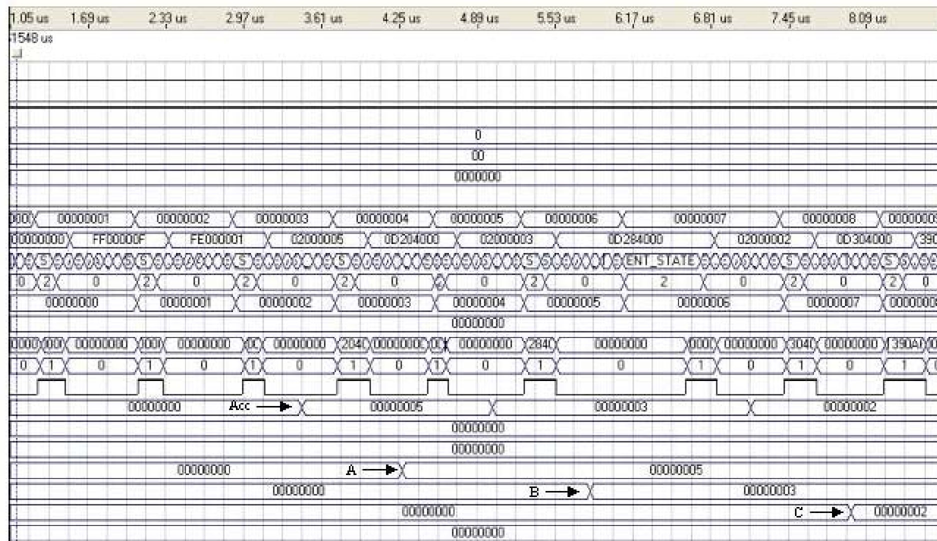


Figura 52: Simulação do processo de obtenção de valores

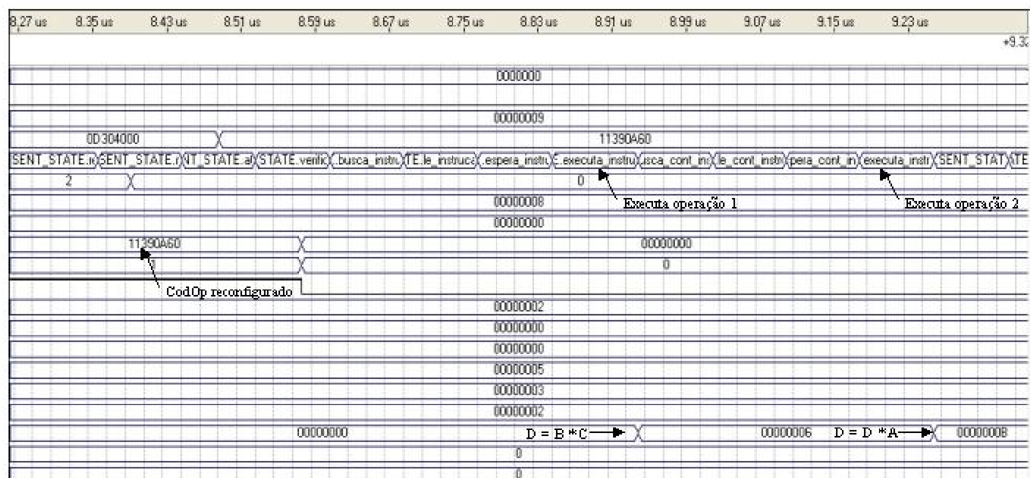


Figura 53: Simulação do processo de execução da instrução reconfigurada

Excluindo-se o tempo gasto para reconfigurar a instrução, o tempo necessário apenas para a execução da aplicação é de $8,26\mu s$.

É possível reduzir este tempo de execução criando a instrução **LDR X**. Esta instrução permite que qualquer registrador possa receber diretamente um valor constante de 14 bits, ou seja, elimina-se a obrigatoriedade do valor desejado primeiramente ser armazenado no acumulador. Esta instrução contém somente uma microinstrução: Ra = "00000000"H + "0000000000000"B & IR(18-0).

Para executar esta microinstrução, é necessário que o valor de IR seja (8, 5, 5, 14). Os 8 primeiros bits correspondam ao CodOp, assim como ocorre em todas as instruções. Os 5 bits do

primeiro operando correspondem ao registrador destino da operação. Os 5 bits do primeiro operando devem ser "00000", o que corresponde ao registrador '0'. Os 5 bits do segundo operando podem assumir qualquer valor, pois não serão utilizados. Isto se deve ao fato de que os bits de *Bmux* da palavra de reconfiguração devem ser "11", selecionando a entrada "0000000000000"B & IR(18-0). Esta é a única configuração do processador que permite a realização da instrução LDR X em uma única operação.

A última tela desta simulação, correspondente a Figura 54, apresenta a execução de uma aplicação MAC utilizando a instrução LDR. O processo de execução desta aplicação necessita de 54 estados, sendo 6 estados para reconfigurar a instrução LDR, 11 estados para executá-la e 15 estados para executar a instrução MAC. O tempo de execução desta aplicação, excluindo-se o tempo necessário para reconfigurar as instruções, é de $4,17\mu s$, ou seja, houve uma redução de 50% na execução desta aplicação.

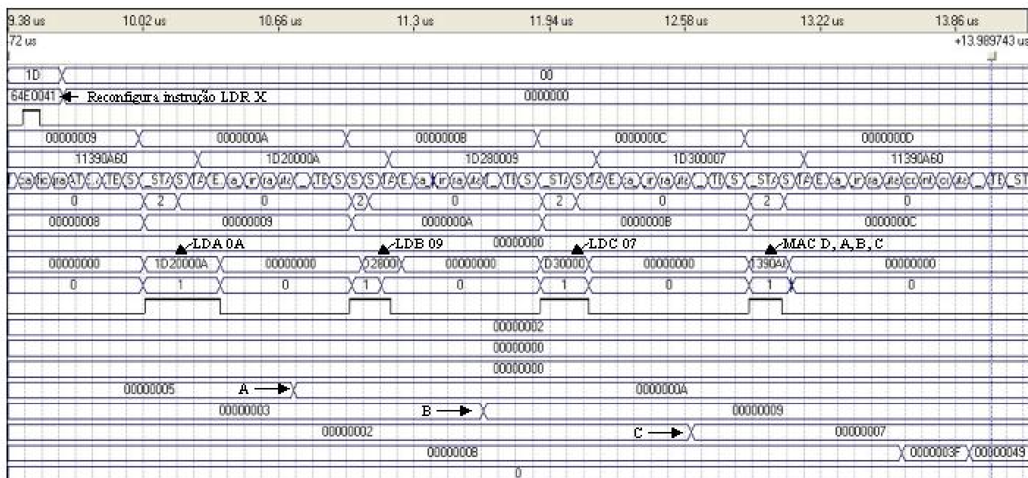


Figura 54: Simulação do processo de execução das instruções MAC e LDR

4.2.3 Simulação de uma aplicação: Fatorial

Esta simulação envolve a execução de uma aplicação completa para a obtenção do fatorial de um número. A execução desta aplicação foi realizada com base neste algoritmo:

```
Algoritmo Fatorial Acc

Início: Leia Acc;              (01)
        A <= 1;                (02)
        B <= 1;                (03)
```

```
        C <= 1;                 (04)
        Acc <= Acc - C;         (05)
        Se Acc = 0 Então        (06)
            vá para Fim;
        Fim Se;
Loop:   B <= B + C;             (07)
        A <= A * B;             (08)
        Acc <= Acc - C;         (09)
        Se Acc != 0 Então       (10)
            vá para Loop;
        Fim Se;
Fim:    Acc <= A;               (11)
```

Utilizando como exemplo Acc = "05"D, a aplicação deste algoritmo consiste nos seguintes passos:

```
Fatorial 05

Início: Acc = 5;                (01)
        A = 1;                  (02)
        B = 1;                  (03)
        C = 1;                  (04)
        Acc = 5 - 1; (Acc = 4)  (05)
        Acc != 0                (06)
Loop:   B = 1 + 1;   (B = 2)    (07)
        A = 1 * 2;   (A = 2)    (08)
        Acc = 4 - 1; (Acc = 3)  (09)
        Acc != 0                (10)
Loop:   B = 2 + 1;   (B = 3)    (07)
        A = 2 * 3;   (A = 6)    (08)
        Acc = 3 - 1; (Acc = 2)  (09)
        Acc != 0                (10)
Loop:   B = 3 + 1;   (B = 4)    (07)
        A = 6 * 4;   (A = 24)   (08)
        Acc = 2 - 1; (Acc = 1)  (09)
        Acc != 0                (10)
```

```
Loop:   B = 4 + 1;   (B = 5)     (07)
        A = 24 * 5;  (A = 120)   (08)
        Acc = 1 - 1; (Acc = 0)   (09)
        Acc = 0                  (10)
Fim:    Acc <= 120;              (11)
```

Inicialmente, esta aplicação foi implementada utilizando apenas as instruções do conjunto de instrução 2. A Tabela 7 mostra a seqüência de execução da aplicação. Cada linha da tabela representa a execução de uma operação distinta.

Tabela 7: Execução do fatorial utilizando apenas instruções fixas

PC	**IR**	**Mnemônico**	**Efeito**	**Ciclos**
00H	FF00000FH	Muda_modo	modo supervisor	9
01H	FE00001FH	Muda_conjunto	conjunto 2	9
02H	02000001H	LOCO 01	Acc = 1	9
03H	0D204000H	ADDR A, Acc, 0	A = Acc + 0	9
04H	0D284000H	ADDR B, Acc, 0	B = Acc + 0	9
05H	0D304000H	ADDR C, Acc, 0	C = Acc + 0	9
06H	02000005H	LOCO 05	Acc = 5	9
07H	0E084C00H	SUBR Acc, Acc, C	Acc = Acc - C	9
08H	0C00000DH	JZER 0D	IF Acc = 0 PC = 0DH	9
09H	0D294C00H	ADDR B, B, C	B = B + C	9
0AH	0F210A00H	MULR A, A, B	A = A * B	9
0BH	0E84C00H	SUBR Acc, Acc, C	Acc = Acc - C	9
0CH	0B000009H	JNZE 09	IF $Acc \neq 0$ PC = 09H	10
0DH	0D090000H	ADDR Acc, A, 0	Acc = A + 0	9

A Figura 55 apresenta uma tela contendo uma visão geral da execução da aplicação. A Figura mostra que nesta aplicação não houve nenhuma alteração nos bits MCmd_ext, MData_ext e MAddr_ext. Isto significa que em nenhum momento durante a execução desta instrução houve um pedido de reconfiguração de uma instrução. Conseqüentemente, não foi realizada nenhuma operação a partir de uma instrução reconfigurada.

A figura também ilustra o conteúdo dos registradores 1 a 7, em que SIG1 representa o acumulador, SIG4 representa o registrador A e assim sucessivamente. Com base nos valores, representados em hexadecimal, assumidos pelos registradores, comprova-se a correta execução da aplicação.

A Figura 56 apresenta uma visão um pouco mais detalhada do processo de cálculo do fatorial no período de simulação aonde ocorrem as operações de adição, multiplicação e subtração envolvendo os registradores selecionados.

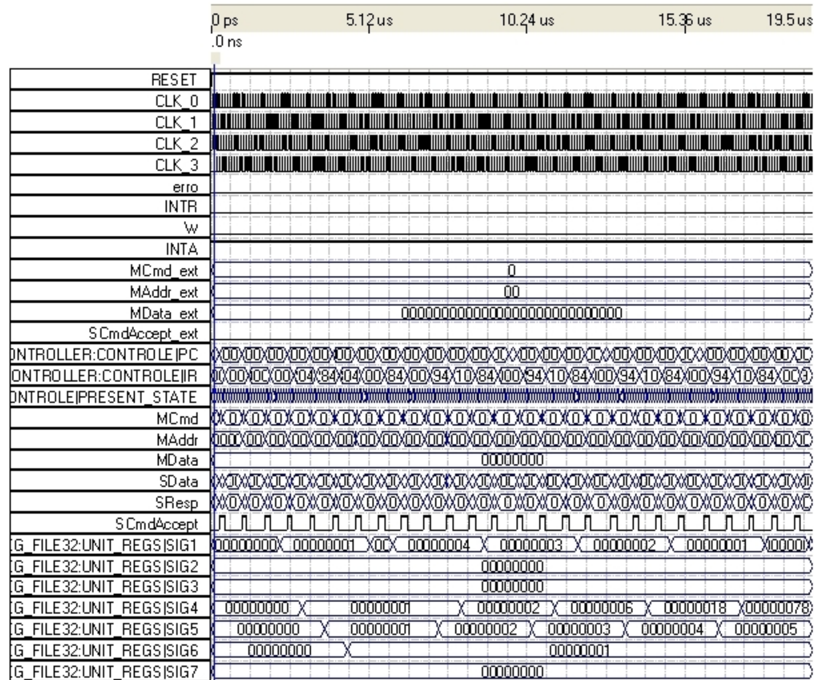


Figura 55: Visão geral da simulação do Fatorial utilizando instruções fixas

Figura 56: Execução das operações durante a simulação do Fatorial

O processo a partir do estado *Verifica_Pedido* até o estado de decodificação possui 8 estados distintos. Todas as instruções fixas executam estes estados, além dos estados específicos destinados as operações da instrução. Desse modo, conclui-se que a execução de uma instrução fixa demora no mínimo 9 estados para ser finalizada. Para se executar o algoritmo acima, foram necessários 237 estados. Como cada estado possui 4 ciclos de relógio, e cada ciclo de relógio possui um período de 20*ns*, o tempo mínimo de execução desta aplicação é de 18.960*ns*, embora o tempo real de execução da aplicação dependa do tempo de resposta da memória RAM

para cada solicitação de leitura.

A Figura 57, indica o momento em que a aplicação é finalizada ($19,46\mu s$). Sabendo-se que a aplicação foi iniciada com $120ns$, o tempo total gasto para a execução desta aplicação foi de $19,34\mu s$. A potência total consumida durante este período foi de $131,35mW$, o que resulta em um gasto de energia de $2,540\mu J$.

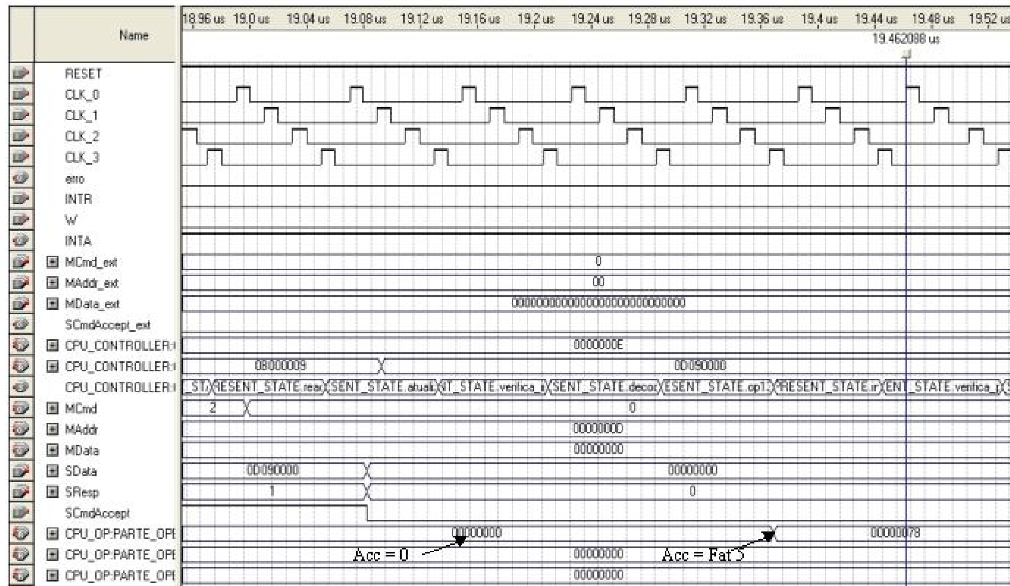


Figura 57: Término da execução do Fatorial

Um segundo exemplo de um fatorial foi implementado, utilizando agora instruções reconfiguradas. Para auxiliar a execução do fatorial foram criadas as instruções **INC R** e **DEC R**, vistas na seção 3.9. A Figura 58 mostra a seqüência completa da execução desta aplicação utilizando as instruções reconfiguradas.

A Tabela 8 mostra o novo algoritmo para o cálculo do fatorial utilizando as instruções criadas. A Tabela 9 contém algumas informações adicionais sobre as instruções reconfiguradas utilizadas neste exemplo.

A Figura 59 mostra uma tela de simulação aonde ocorre o processo de execução das instruções reconfiguradas. Como visto anteriormente, todas as instruções reconfiguradas que possuem apenas uma operação necessitam de 11 estados para serem executadas.

Por este motivo, não se deve criar uma instrução reconfigurada contendo uma operação para substituir uma instrução fixa que realize esta mesma operação, pois a instrução reconfigurável gasta 2 ciclos a mais do que a instrução fixa que contenha também uma operação. Esta diferença é existente devido aos ciclos extras que o componente lpm_ram utilizado necessita para

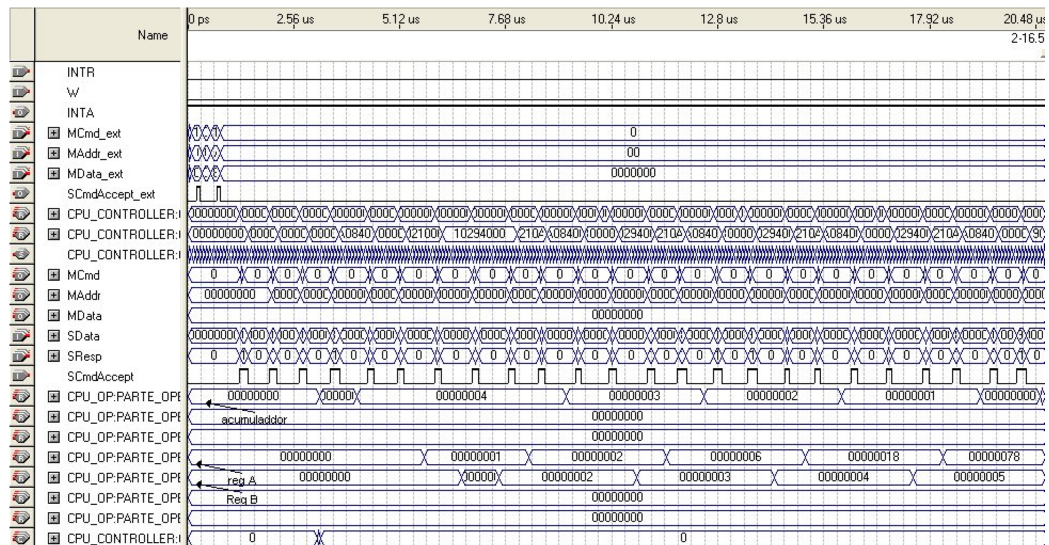


Figura 58: Visão geral da simulação do Fatorial utilizando instruções INC R e REC R

Tabela 8: Execução do fatorial utilizando instruções reconfiguradas

PC	IR	Mnemônico	Efeito	Ciclos
00H	-	Criar INC R	R = R + 1	6
00H	-	Criar DEC R	R = R - 1	6
00H	FF00000FH	Muda_modo	modo supervisor	9
01H	FE00001FH	Muda_conjunto	conjunto 2	9
02H	02000005H	LOCO 05	Acc = 5	9
03H	1A084000H	DEC Acc	Acc = Acc - 1	11
04H	0C00000BH	JZER 0B	IF Acc = 0 PC = 0BH	9
05H	10210000H	INC A	A = A + 1	11
06H	10294000H	INC B	B = B + 1	11
07H	10294000H	INC B	B = B + 1	11
08H	0F210A00H	MULR A, A, B	A = A * B	9
08H	1A084000H	DEC Acc	Acc = Acc - 1	11
0CH	0B000007H	JNZE 07	IF $Acc \neq 0$ PC = 07H	9/10
0DH	0D090000H	ADDR Acc, A, 0	Acc = A + 0	9

Tabela 9: Informações adicionais das instruções reconfiguradas

Instrução	Operação	MData	MAddr	Endereço RAM reconf.
INC R	INC R	6460041H	1AH	00
DEC R	DEC R	6468041H	10H	01

disponibilizar o dado. Caso haja outro componente que realize esta operação em menos ciclos, esta diferença se tornará inexistente, proporcionando um aumento de desempenho ainda maior.

Esta aplicação foi executada em um total de 253 estados. Multiplicando este valor pelo período de relógio total (80ns), o tempo mínimo de execução desta aplicação é de 20.240ns. O tempo real desta simulação, considerando a variação de tempo durante as operações de leitura, é de $20,765\mu s$. Excluindo-se o período gasto para iniciar o processador, o tempo de execução

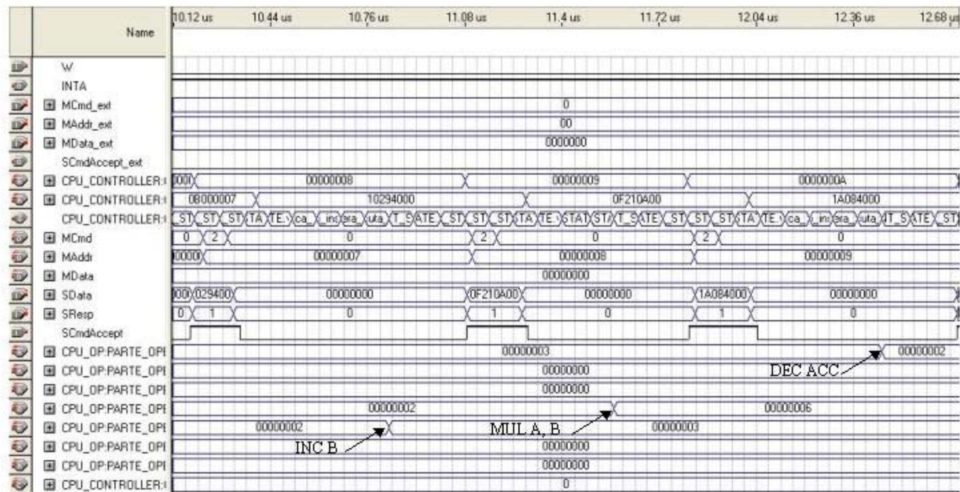


Figura 59: Execução das instruções INC R e REC R

desta aplicação foi de $20,645\mu s$. A potência total consumida foi de $135.84mW$, proporcionando um gasto de energia de $2,80\mu J$.

A partir da segunda vez que esta aplicação é executada, não se faz necessário repetir o processo de reconfiguração das instruções, como visto na Figura 60. Desse modo, o total de estados necessários para executar o fatorial é reduzido para 241 estados, resultando em um tempo mínimo de simulação de $19.517ns$. A potência neste intervalo é de $121,30mW$, consumindo uma energia de $2,36\mu J$.

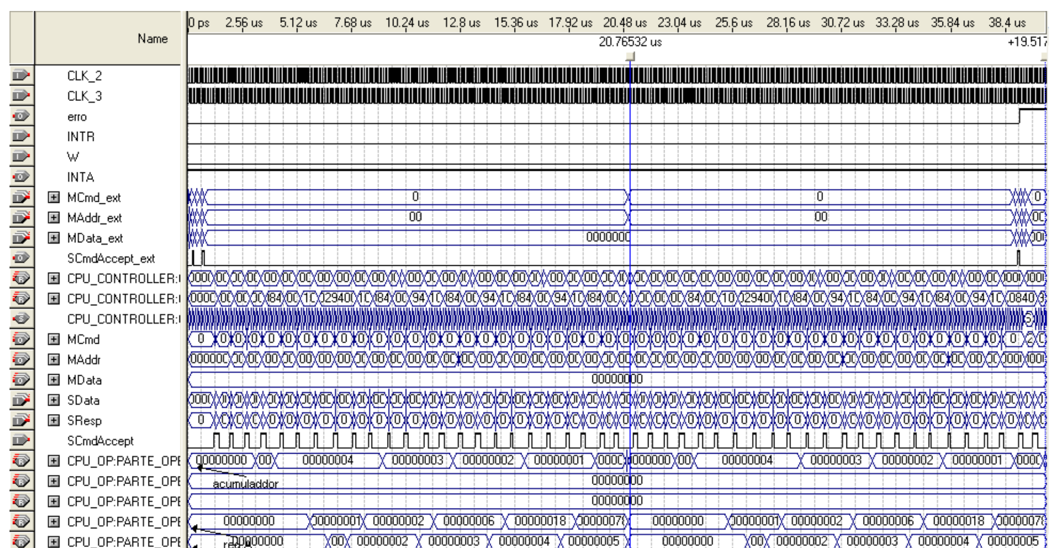


Figura 60: Repetição do processo de fatorial sem reconfiguração de INC e DEC

O último exemplo consiste na criação da instrução **FAT Acc, A, B**, por meio da combinação de instruções. Esta instrução FAT contém 3 operações, que são INC B, MULR A, A, B e DEC

Acc. Como somente um valor de IR será recebido, e para cada operação os registradores de destino e entrada são diferentes, cada uma das operações deverá seguir um formato distinto.

A primeira operação (INC B) deve fazer com que o registrador B receba o valor do próprio registrador B adicionado de uma unidade. Como B é o terceiro operando (BAR B) da instrução FAT, esta operação deve seguir algum formato em que **BAR B = BAR B + X**. Mediante a tabela de codificação de formatos (apêndice F), os bits reservados para o campo *formato* da palavra de reconfiguração terão valor "001010". Após a definição dos valores dos demais componentes, o valor final da primeira palavra de reconfiguração será 2860040H.

Devem ser seguidos os mesmos procedimentos para todas as operações seguintes. A segunda operação deve realizar $A = A * B$, ou seja, **BAR A = BAR A * BAR B**. O valor final desta palavra de reconfiguração será 0430040H. Para realizar a operação DEC Acc, o formato da operação deve ser **BAR C = BAR C - X**, resultando na palavra de reconfiguração igual a 4C68041H. Como o último bit desta palavra contém o valor '1', significa o término da reconfiguração desta instrução.

A instrução FAT demora 19 estados para ser executada, sendo 10 estados para a primeira operação, 4 para a segunda e 5 para a última. A Tabela 10 mostra o novo algoritmo para o cálculo do fatorial utilizando a instrução FAT. A Tabela 11 contém algumas informações adicionais sobre as instruções reconfiguradas utilizadas neste exemplo.

Tabela 10: Execução do fatorial utilizando instruções reconfiguradas com mais de uma operação

PC	**IR**	**Mnemônico**	**Efeito**	**Ciclos**
00H	-	Criar FAT C, A, B	INC R	6
00H	-	Continua FAT C, A, B	MULR A, A, B	6
00H	-	Continua FAT C, A, B	Dec R	6
00H	FF00000FH	Muda_modo	modo supervisor	9
01H	FE00001FH	Muda_conjunto	conjunto 2	9
02H	02000005H	LOCO 05	Acc = 5	9
02H	-	Criar DEC R	R = R - 1	6
03H	15084001H	DEC Acc	Acc = Acc - 1	11
04H	0C000009H	JZER 09	IF Acc = 0 PC = 09H	9
04H	-	Criar INC R	R = R + 1	6
05H	18210001H	INC A	A = A + 1	11
06H	18294001H	INC B	B = B + 1	11
07H	10090A00H	FAT Acc, A, B	Acc!	19
08H	0B000007H	JNZE 07	IF $Acc \neq 0$ PC = 07H	9/10
09H	0D090000H	ADDR Acc, A, 0	Acc = A + 0	9

A Figura 61 apresenta o processo de simulação do fatorial realizada duas vezes. A primeira execução, considerando o tempo gasto para reconfigurar a instrução FAT, dura $18,498 \mu s$, em um total de 223 estados. A potência total deste processo é de $135,45 mW$. A segunda execução,

Tabela 11: Informações adicionais das instruções reconfiguradas

Instrução	Operação	MData	MAddr	Endereço RAM Reconf.
FAT C, A, B	INC R	2860040H	10H	00
FAT C, A, B	MULR A, A, B	0430040H	10H	01
FAT C, A, B	DEC R	4C68041H	10H	02
DEC R	DEC R	6468041H	15H	03
INC R	INC R	6460041H	18H	04

por não necessitar que o processo de reconfiguração seja realizado novamente, é realizada em $16,145\mu s$, consumindo $121,95mW$ de potência. A energia consumida neste último período é de $1,968\mu J$. Em relação a primeira execução do fatorial, em que não foram utilizadas instruções reconfiguradas, houve um ganho de 22,50% em relação ao consumo de energia.

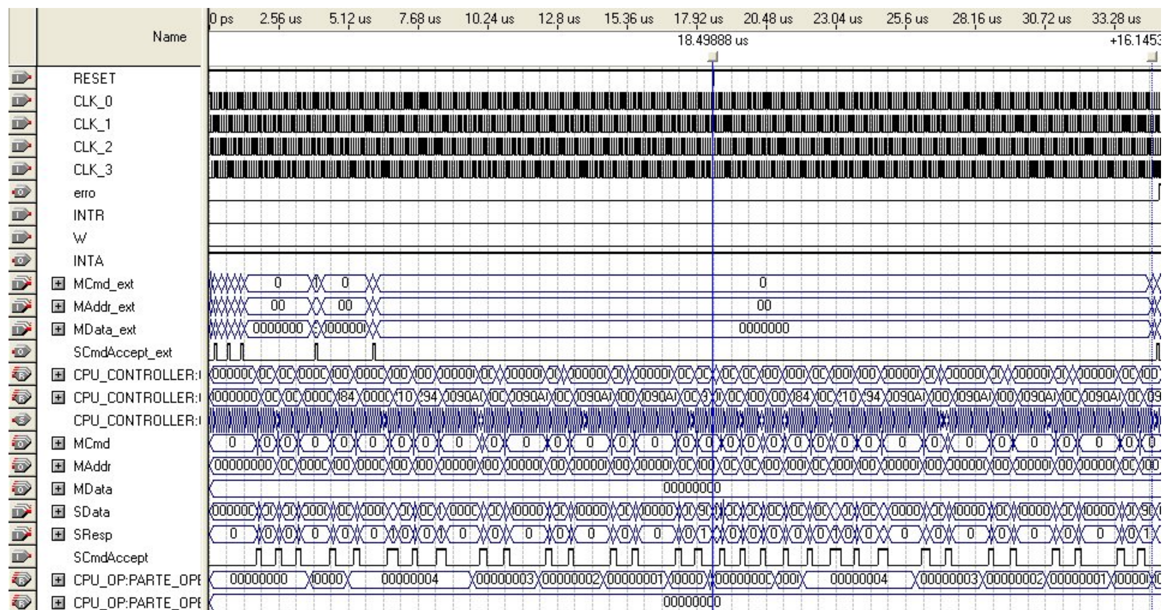


Figura 61: Repetição do processo de fatorial sem reconfiguração de FAT

5 Considerações finais

O desenvolvimento deste trabalho teve como objetivo central o projeto e implementação de um processador com conjunto de instrução reconfigurável. Com o projeto e implementação do processador apresentado nesta dissertação pretende-se estudar a aplicabilidade e a eficiência de dois conceitos. O primeiro destes conceitos diz respeito à utilização no processador de mais de um conjunto de instrução. Nesta dissertação estes conjuntos de instrução foram chamados de conjuntos de instrução fixos, em oposição à possibilidade de criar novas instruções em tempo de execução. O segundo conceito diz respeito justamente à possibilidade de, em tempo de execução, definir novos padrões de instruções, definindo seu formato, operandos, bem como as operações elementares executadas durante o ciclo de vida desta instrução. Dos conjuntos de instrução fixos apenas um deve estar ativo em um determinado momento. A criação de instruções possibilita a combinação de padrões de instruções pré-existentes, possibilitando uma melhor adaptação entre o conjunto de instruções e a aplicação. Uma vez criadas novas instruções, o processador passa a reconhecê-las e utilizá-las como uma de suas instruções nativas. Para tornar possível a utilização de mais de um conjunto de instrução durante a execução de uma aplicação, foram criadas instruções privilegiadas que definem o conjunto em uso.

Na arquitetura de processador cuja proposta e implementação são objeto desta dissertação, três conjuntos de instruções foram criados. Obedecendo aos objetivos de projeto e implementação de uma plataforma de hardware que possibilite o estudo de viabilidade, aplicabilidade e eficiência dos conceitos já mencionados, os conjuntos criados foram consideravelmente simples. Cada conjunto possui características e instruções próprias, entretanto algumas instruções básicas são compartilhadas por mais de um conjunto ou pelos três. A implementação da unidade de controle do processador (uma máquina de estados finitos)leva em consideração a possibilidade de compartilhamento de características e recursos e, visando a minimização da área ocupada, compartilha estados sempre que possível.

Nesta dissertação foram realizados experimentos no sentido de analisar o custo de cada um dos conjuntos de instrução. Com base nestes estudos verificou-se que 822 células lógicas do dispositivo EP1S20F780C5 da família Sratrix foram necessárias para a implementação de um

conjunto. Para a integração de um segundo um conjunto o número de células lógicas cresceu para 1023, o que representa um aumento de 201 células lógicas, o que corresponde a um aumento de até 2% da área utilizada pelo dispositivo. A implementação final com três conjuntos requisitou um total de 1380 células lógicas, representando um aumento de 4% com relação a uma unidade de controle com apenas um conjunto de instrução. Na dissertação o leitor encontra simulações e tabelas com resultados de síntese que comprovam a eficiência da implementação da máquina de estados com três conjuntos de instrução.

Devido a existência de diferentes conjuntos de instruções na mesma arquitetura, espera-se que o compilador possa explorar o espaço de soluções de compilação possível, escolhendo para cada trecho de código, rotina ou laço, qual o conjunto de instrução que melhor se adequado segundo uma métrica de desempenho que resta a ser definida. Esta métrica de desempenho pode estar baseada no tempo de execução ou na potência consumida, por exemplo. Nesta dissertação de mestrado não foi possível realizar experimentos de compilação, entretanto no grupo de pesquisa em arquiteturas e sistemas integrados, trabalhos desta natureza, utilizando o processador aqui proposto, já foram iniciados.

A criação de novas instruções apresenta-se como uma proposta agressiva no que diz respeito a definição, em tempo de execução ou compilação, de um conjunto de instrução mais adequado a aplicação. Mais uma vez a adequação pode ser vista segundo diferentes métricas. Para permitir a criação de instruções, foi projetada uma unidade de reconfiguração junto a unidade de controle. Nesta dissertação o leitor encontra simulações e resultados de área e potência consumida que comprovam a eficiência da unidade de reconfiguração. Quanto aos resultados de potência consumida observa-se que 131,35 mW são necessários para a execução do algoritmos de cálculo do fatorial sem o uso da unidade de reconfiguração, ou seja, sem o uso de instruções customizadas. O mesmo algoritmo, quando utilizando ao máximo a possibilidade de criação de novas instruções, consome uma potência de 121,95 mW. Nestes resultados destaca-se que para o cálculo do algoritmo original não foi considerado o consumo de potência associado aos acessos à memória para busca de instruções e dados, entretanto é bem conhecido que o acesso à memória representa a principal parcela de consumo de potência na execução de um código qualquer. Por outro lado, na execução da versão otimizada do algoritmo, as instruções customizadas são buscadas em memórias internas ao dispositivo utilizado para síntese e simulação. Por encontrarem-se internas ao dispositivo, o consumo de potência destes acessos à memória foram considerados. No corpo da dissertação encontram-se resultados de potência consumida durante a execução dos algoritmos, durante o processo de reconfiguração e durante a execução pós-reconfiguração. A técnica de criação de novas instruções implica em consumo de potência para reconfiguração, entretanto, se o trecho de código que utiliza instruções reconfiguradas é

um trecho cuja execução se repete e é executada em menos tempo que o algoritmo original, o resultado é um consumo de energia menor que a do algoritmo original.

Em relação a quantidade de estados utilizados na FSM do processador, verifica-se que para reconfigurar uma instrução são necessários 6 estados para cada operação existente. Para executar uma operação fixa, são necessários no mínimo 9 estados, considerando uma instrução com apenas uma operação. Cada operação extra consome 1 estado a mais durante a execução. Já para a execução de uma instrução customizada, são necessários 11 estados para executar a primeira operação, e as demais operações necessitam de apenas 4 ou 5 estados. Estes valores também comprovam a eficiência na resolução de problemas utilizando a técnica de reconfiguração.

A comparação da execução do algoritmo original e de sua versão otimizada mostrou que a utilização de instruções reconfiguradas, principalmente quando é realizada uma combinação entre duas ou mais operações, proporciona aumento de desempenho do processador, atingindo assim os objetivos esperados.

Com base nas informações encontradas, conclui-se que os resultados obtidos frente à compilação do processador foram considerados satisfatórios. Foram utilizados 3.798 células lógicas do dispositivo EP1S10F484C5 da família Stratix, resultando em uma freqüência máxima de operação de 61MHz. Estes resultados são equivalentes a outras implementações em FPGA que não utilizam pipeline. Desse modo, é possível afirmar que o processador reconfigurável atende as finalidades para as quais foi projetado.

Referências

[Adário, 1997] Adário, A. M. S. (1997). Arquiteturas reconfiguráveis. Porto Alegre, RS, Brasil.

[Altera, 2005] Altera (2005). *Quartus II 4.1 version Altera*.

[Aragão et al., 2000] Aragão, A., Romero, R., and Marques, E. (2000). Computação reconfigurável aplicada à robótica. Marília, SP, Brasil. Workshop de Computação Reconfigurável - CORE 2000.

[Athanas and Silverman, 1993] Athanas, P. M. and Silverman, H. F. (1993). Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18.

[Azevedo et al., 2003] Azevedo, A., Soares, R., and Silva, I. S. (2003). A new hybrid parallell/reconfigurable architecture: The x4cp32. In *Proceedings of 16th Symposium on Integrated Circuits and Systems Design*, pages 225–230, São Paulo, SP, Brasil. IEEE Circuits and Systems Society.

[Barat et al., 2003] Barat, F., Jayapala, M., Aa, T. V., Lauwereins, R., Deconinck, G., and Corporaal, H. (2003). Low power coarse-grained reconfigurable instruction set processor. In *3th International Conference on Field Programmable Logic and Applications*, 1st - 3rd Sept. 2003, in Lisbon, Portugal.

[Barat et al., 2002a] Barat, F., Jayapala, M., de Beeck, P. O., Deconinck, G., and Leuven, K. U. (2002a). Software pipelining for coarse-grained reconfigurable instruction set processors. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 338, Washington, DC, USA. IEEE Computer Society.

[Barat et al., 2002b] Barat, F., Lauwereins, R., and Deconinck, G. (2002b). Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Trans. Softw. Eng.*, 28(9):847–862.

[Barat and Lauwereins, 2000] Barat, F. and Lauwereins, R. (2000). Reconfigurable instruction set processors: A survey. In *RSP '00: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, page 168, Washington, DC, USA. IEEE Computer Society.

[Carrillo and Chow, 2001] Carrillo, J. E. and Chow, P. (2001). The effect of reconfigurable units in superscalar processors. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 141–150, New York, NY, USA. ACM Press.

[Casillo et al., 2005] Casillo, L. A., Lima, L. M. P., de Souza, G. B., and Silva, I. S. (2005). Projeto e implementação em um fpga de um processador com conjunto de instrução reconfigurável utilizando vhdl. In *In proceedings of XI Workshop Iberchip*, Salvador, BA, Brasil.

[Compton and Hauck, 2000] Compton, K. and Hauck, S. (2000). An introduction to reconfigurable computing. IEEE Computer.

[Dehon, 1996] Dehon, A. M. (1996). *Reconfigurable architectures for general-purpose computing*. PhD thesis. Supervisor-Thomas Knight, Jr.

[de Beeck et al., 2001] de Beeck, P. O., Barat, F., Jayapala, M., and Lauwereins, R. (2001). Crisp: A template for reconfigurable instruction set processors. In *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 296–305, London, UK. Springer-Verlag.

[Goncalves et al., 2003] Goncalves, R. A., Moraes, P. A., Cardoso, J. M. P., Wolf, D. F., Fernandes, M. M., Romero, R. A. F., and Marques, E. (2003). Architect-r: A system for reconfigurable robots. In *In proceedings of ACM Anual symposium on Applied Computing - SAC*, Melbourne - Florida.

[Hartenstein, 2001] Hartenstein, R. (2001). A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA. IEEE Press.

[Hauser and Wawrzynek, 1997] Hauser, J. R. and Wawrzynek, J. (1997). Garp: a mips processor with a reconfigurable coprocessor. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, page 12, Washington, DC, USA. IEEE Computer Society.

[IEEE, 1993] IEEE (1993). *IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual*.

[Ito et al., 2001] Ito, S. A., Carro, L., and Jacob, R. (2001). Making java work for microcontroller applications. *IEEE Design and Test of Computers*, 18(5):100–110.

[It and Carro, 2000] It, S. A. and Carro, L. (2000). A comparison of microcontrollers targeted to fpga-based embedded applications. In *13th Symposium on Integrated Circuits and Systems Design*, pages 399–402, Manaus, AM, Brasil. Brasilian Computer Society.

[Liu et al., 2003] Liu, J., Chow, F., Kong, T., and Roy, R. (2003). Variable instruction set architecture and its compiler support. *IEEE Trans. Comput.*, 52(7):881–895.

[Lodi et al., 2003] Lodi, A., Mario Toma, F. C., Cappelli, A., and Guerrieri, R. (2003). A vliw processor uit reconfigurable instruction set for embedded applications. *IEEE Journal of solid-state circuits*, 538(11).

[Martins et al., 2003] Martins, C. A. P. S., Ordonez, E. D. M., Corrêa, J. B. T., and Carvalho, M. B. (2003). Computação reconfigurável: conceitos, tendências e aplicações. In *Jornada de Atualização em Informática (JAI 2003)*, pages 339–388, Campinas, SP, Brasil. Sociedade Brasileira de Computação.

[Mesquita, 2002] Mesquita, D. G. (2002). Contribuições para reconfiguração parcial, remota e dinâmica de fpgas. Master's thesis, Pontificia Universidade Catolica do Rio Grande do Sul, Porto Alegre, RS, Brasil.

[Partnerchip, 2001] Partnerchip, O. I. (2001). *Open Core Protcol Specification*.

[Ramos et al., 2004] Ramos, K. D. N., Casillo, L. A., Santiago, N. R. A. C., Oliveira, J. A. N., Augusto, A., and Silva, I. S. (2004). Projeto baseado em reuso: Implementação de um ip de processador didático em fpga com interface ocp. In *In proceedings of X Iberchip*, Cartagena del Indias, Colômbia.

[Razdan and Smith, 1994] Razdan, R. and Smith, M. D. (1994). A high-performance microarchitecture with hardware-programmable functional units. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 172–180, New York, NY, USA. ACM Press.

[Silva, 2001] Silva, I. S. (2001). Arquiteturas reconfiguráveis: Teoria e prática. In *III Escola de Microeletrônica da SBC-Sul*, pages 161–182, Santa Maria, RS, Brasil. EMICRO2001.

[Smith, 1997] Smith, M. J. S. (1997). *Aplication-Specific Integrated Circuits*. Addison-Wesley, Berkeley, California, 3 edition.

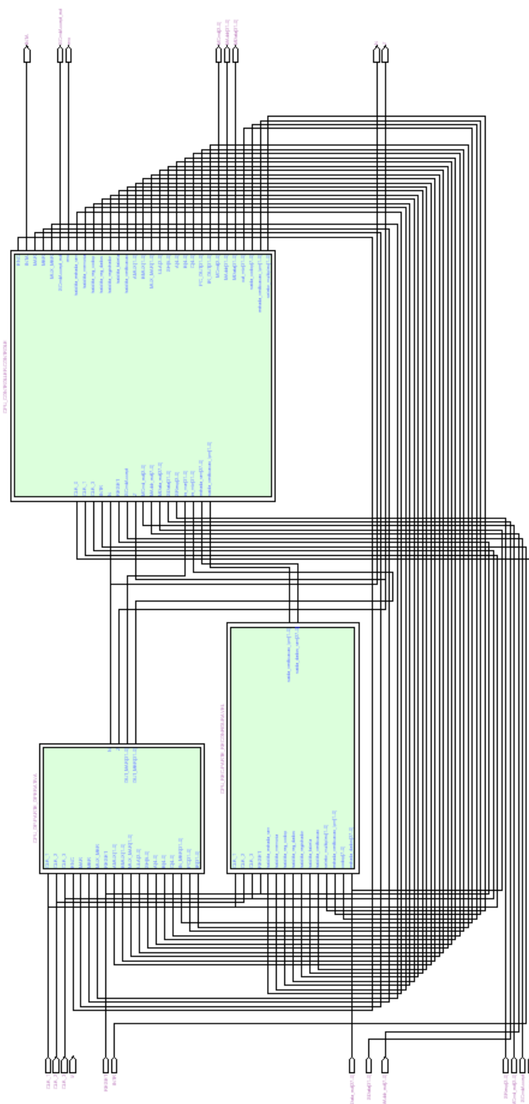
[Soares et al., 2004] Soares, R., Azevedo, A., and Silva, I. S. (2004). When reconfigurable architecture meets network-on-chip. In *Proceedings of 17th Symposium on Integrated Circuits and Systems Design*, pages 216–221, Porto de Galinhas, PE, Brasil. IEEE Circuits and Systems Society.

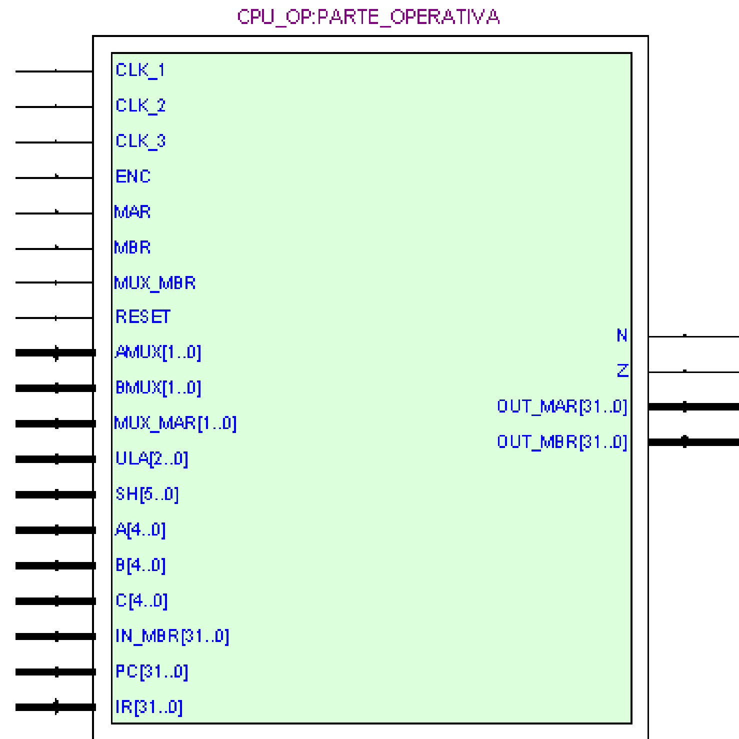
[S. Hauck, 1997] S. Hauck, T. W. Fry, M. M. H. J. P. K. (1997). The chimaera reconfigurable functional unit. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, page 87, Washington, DC, USA. IEEE Computer Society.

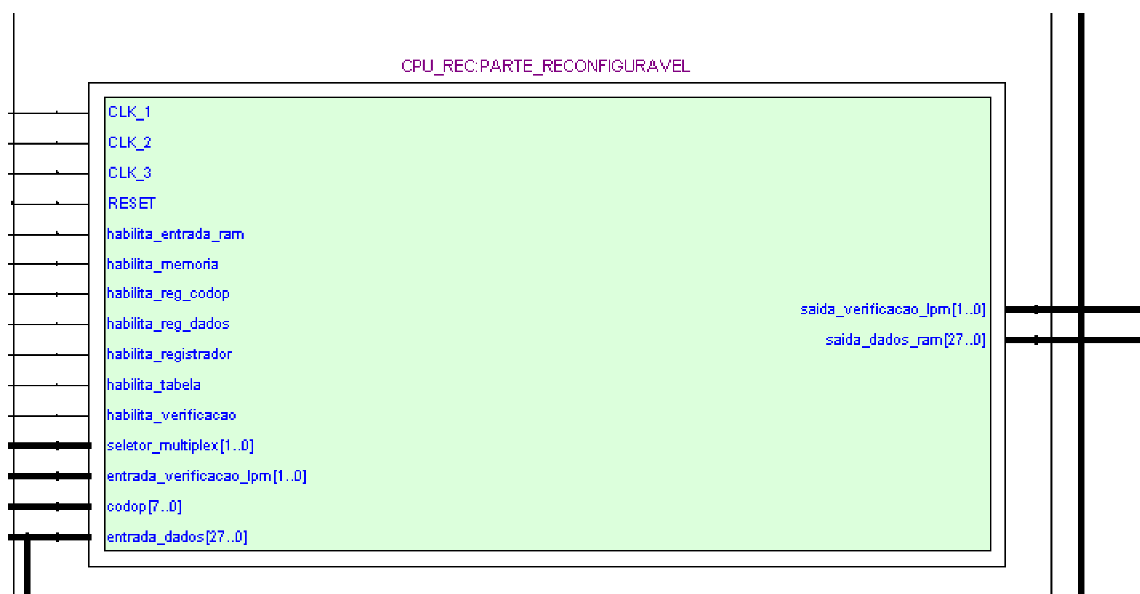
[Tanenbaum, 1992] Tanenbaum, A. S. (1992). *Organização Estruturada de Computadores*. Prentice/Hall do Brasil, Rio de Janeiro, RJ, Brasil, 3 edition.

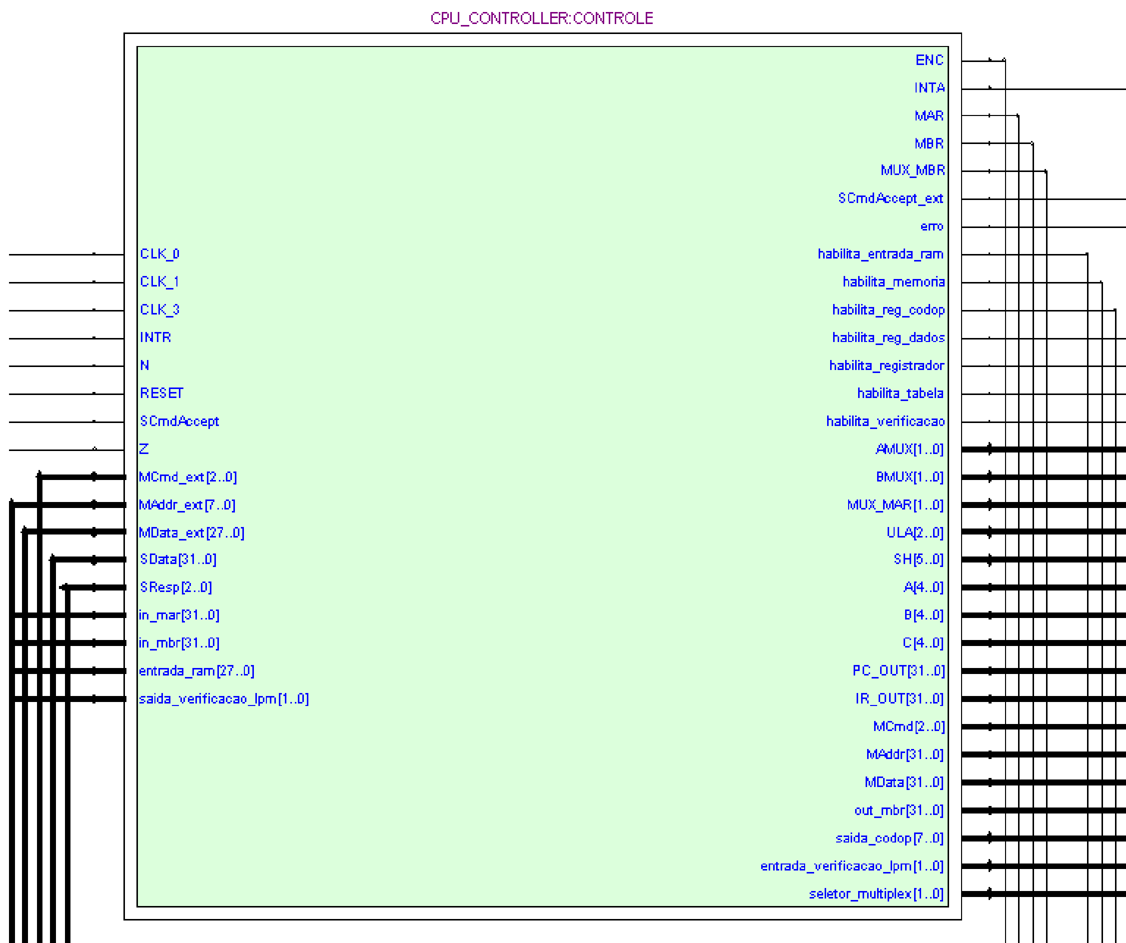
[Wirthlin, 1995] Wirthlin, M. J. (1995). A dynamic instruction set computer. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 99, Washington, DC, USA. IEEE Computer Society.

APÊNDICE A – Esquema completo do processador




CPU_OP:PARTE_OPERATIVA
CLK_1
CLK_2
CLK_3
ENC
MAR
MBR
MUX_MBR
RESET
AMUX[1..0]
BMUX[1..0]
MUX_MAR[1..0]
ULA[2..0]
SH[5..0]
A[4..0]
B[4..0]
C[4..0]
IN_MBR[31..0]
PC[31..0]
IR[31..0]
N
Z
OUT_MAR[31..0]
OUT_MBR[31..0]

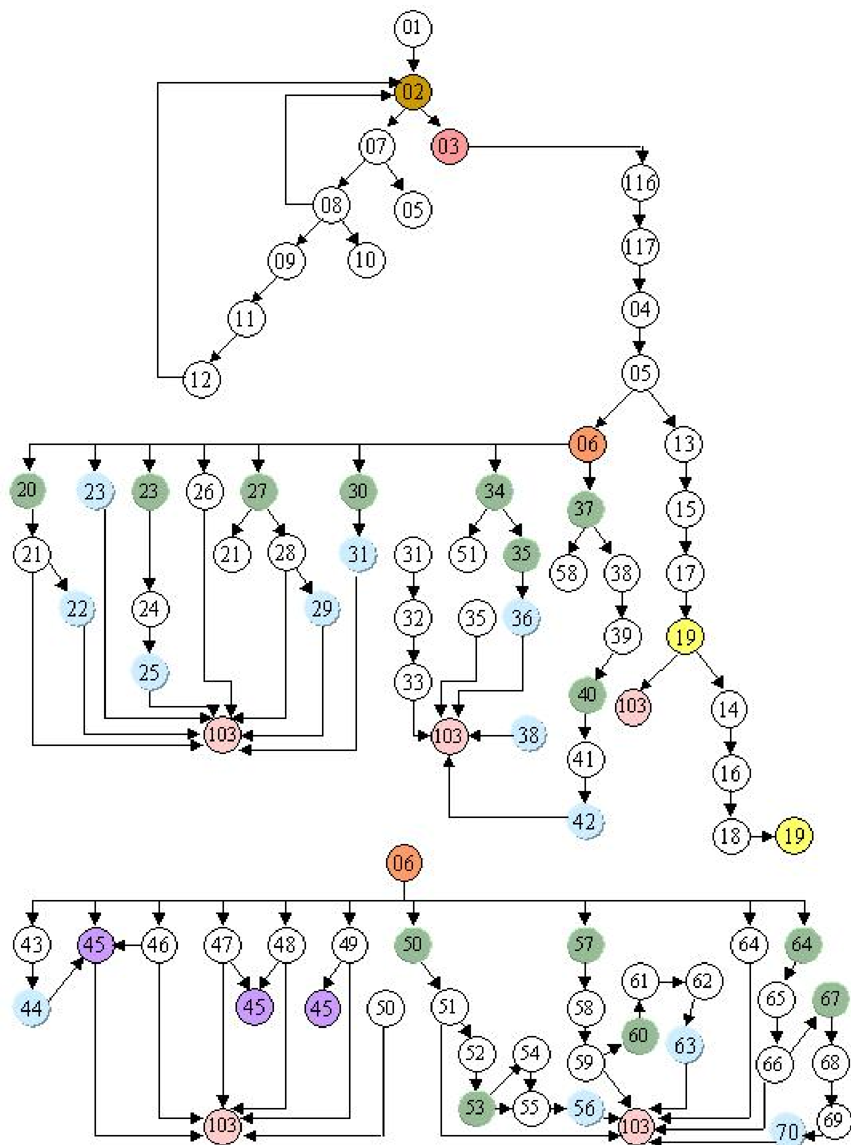

CPU_REC:PARTE_RECONFIGURAVEL
CLK_1
CLK_2
CLK_3
RESET
habilita_entrada_ram
habilita_memoria
habilita_reg_codop
habilita_reg_dados
habilita_registrador
habilita_tabela
habilita_verificacao
seletor_multiplex[1..0]
entrada_verificacao_lpm[1..0]
codop[7..0]
entrada_dados[27..0]
saida_verificacao_lpm[1..0]
saida_dados_ram[27..0]

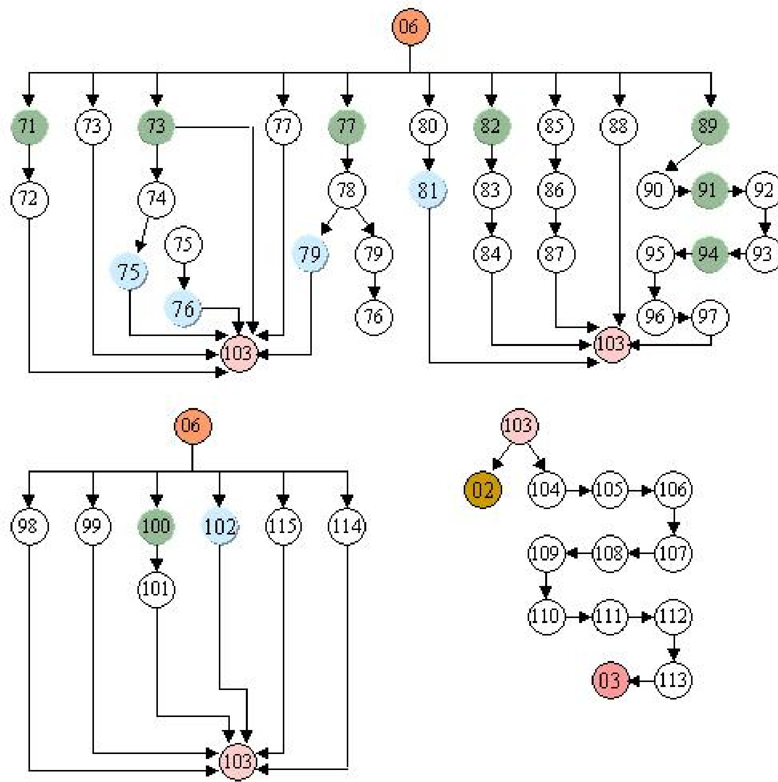

CPU_CONTROLLER:CONTROLE
ENC
INTA
MAR
MBR
MUX_MBR
SCmdAccept_ext
erro
habilita_entrada_ram
habilita_memoria
habilita_reg_codop
habilita_reg_dados
habilita_registrador
habilita_tabela
habilita_verificacao
AMUX[1..0]
BMUX[1..0]
MUX_MAR[1..0]
ULA[2..0]
SH[5..0]
A[4..0]
B[4..0]
C[4..0]
PC_OUT[31..0]
IR_OUT[31..0]
MCmd[2..0]
MAddr[31..0]
MData[31..0]
out_mbr[31..0]
saida_codop[7..0]
entrada_verificacao_lpm[1..0]
seletor_multiplex[1..0]
CLK_0
CLK_1
CLK_3
INTR
N
RESET
SCmdAccept
Z
MCmd_ext[2..0]
MAddr_ext[7..0]
MData_ext[27..0]
SData[31..0]
SResp[2..0]
in_mar[31..0]
in_mbr[31..0]
entrada_ram[27..0]
saida_verificacao_lpm[1..0]

APÊNDICE B – Conjunto de registradores

Código	Registrador
00000	’0’
00001	Acc
00010	FP
00011	SP
00100	A
00101	B
00110	C
00111	D
01000	E
01001	F
01010	G
01011	H
01100	I
01101	J
01110	K
01111	L
10000	M
10001	N
10010	O
10011	P
10100	Q
10101	R
10110	S
10111	T
11000	U
11001	V
11010	W
11011	Y
11100	X
11101	Z
11110	α
11111	β

APÊNDICE C – Fluxograma da máquina de estados do processador




06
71
73
73
77
77
80
82
85
88
89
72
74
75
75
76
78
79
79
76
81
83
84
86
87
90
91
92
95
94
93
96
97
103
103
06
98
99
100
102
115
114
101
103
103
02
104
105
106
109
108
107
110
111
112
03
113

(01) → reseta
(02) → verifica pedido
(03) → inicial
(04) → atualiza IR
(05) → verifica instrução
(06) → decodifica
(07) → verifica instrução reconfigurada
(08) → verifica CodOp reconfigurado
(09) → grava novo end. tabela
(10) → envia mesmo CodOp
(11) → grava CodOp
(12) → atualiza registrador
(13) → busca instrução reconfigurada
(14) → busca cont instrução reconfigurada
(15) → lê instrução reconfigurada
(16) → lê cont, instrução reconfigurada
(17) → espera instrução reconfigurada
(18) → espera cont. instrução reconfigurada
(19) → executa instrução reconfigurada
(20) → OP0_1
(21) → OP0_2
(22) → OP0_3
(23) → OP1_1
(24) → OP1_2
(25) → OP1_3
(26) → OP2_1
(27) → OP3_1
(28) → OP3_2
(29) → OP3_3
(30) → OP4_1
(31) → OP4_2
(32) → OP4_3
(33) → OP4_4
(34) → OP5_1
(35) → OP5_2
(36) → OP5_3
(37) → OP6_1
(38) → OP6_2
(39) → OP6_3
(40) → OP6_4
(41) → OP6_5
(42) → OP6_6

(43) → OP7_1	(64) → OP15_1
(44) → OP7_2	(65) → OP15_2
(45) → OP8_1	(66) → OP15_3
(46) → OP9_1	(67) → OP15_4
(47) → OP10_1	(68) → OP15_5
(48) → OP11_1	(69) → OP15_6
(49) → OP12_1	(70) → OP15_7
(50) → OP13_1	(71) → OP16_1
(51) → OP13_2	(72) → OP16_2
(52) → OP13_3	(73) → OP17_1
(53) → OP13_4	(74) → OP17_2
(54) → OP13_5	(75) → OP17_3
(55) → OP13_6	(76) → OP17_4
(56) → OP13_7	(77) → OP18_1
(57) → OP14_1	(78) → OP18_2
(58) → OP14_2	(79) → OP18_3
(59) → OP14_3	(80) → OP19_1
(60) → OP14_4	(81) → OP19_2
(61) → OP14_5	(82) → OP20_1
(62) → OP14_6	(83) → OP20_2
(63) → OP14_7	(84) → OP20_3

(85) → OP21_1	(106) → INT_4
(86) → OP21_2	(107) → INT_5
(87) → OP21_3	(108) → INT_6
(88) → OP22_1	(109) → INT_7
(89) → OP23_1	(110) → INT_8
(90) → OP23_2	(111) → INT_9
(91) → OP23_3	(112) → INT_10
(92) → OP23_4	(113) → INT_11
(93) → OP23_5	(114) → muda modo
(94) → OP23_6	(115) → muda conj.
(95) → OP23_7	(116) → read 1
(96) → OP23_8	(117) → read 2
(97) → OP23_9	(118) → write
(98) → OP24_1	(Nº) → Operação "read1" e "read2"
(99) → OP25_1	(Nº) → Operação "write"
(100) → OP26_1	
(101) → OP26_2	
(102) → OP27_1	
(103) → INT	
(104) → INT_2	
(105) → INT_3	

APÊNDICE D – Conjuntos de Instruções

D.1 Conjunto 1

Binário	**Hexadecimal**	**Inst. Inicial**	**Mnemônico**
00000000xxxxxxxxxxxxxxxxxxxxxxxx	00XXXXXX	OP0_1	LODD X
00000001xxxxxxxxxxxxxxxxxxxxxxxx	01XXXXXX	OP1_1	STOD X
00000010xxxxxxxxxxxxxxxxxxxxxxxx	02XXXXXX	OP2_1	LOCO X
00000011xxxxxxxxxxxxxxxxxxxxxxxx	03XXXXXX	OP3_1	LODL X
00000100xxxxxxxxxxxxxxxxxxxxxxxx	04XXXXXX	OP4_1	STOL X
00000101xxxxxxxxxxxxxxxxxxxxxxxx	05XXXXXX	OP5_1	ADDL X
00000110xxxxxxxxxxxxxxxxxxxxxxxx	06XXXXXX	OP6_1	SUBL X
00000111xxxxxxxxxxxxxxxxxxxxxxxx	07XXXXXX	OP7_1	CALL X
00001000xxxxxxxxxxxxxxxxxxxxxxxx	08XXXXXX	OP8_1	JUMP X
00001001xxxxxxxxxxxxxxxxxxxxxxxx	09XXXXXX	OP9_1	JNEG X
00001010xxxxxxxxxxxxxxxxxxxxxxxx	0AXXXXXX	OP10_1	JPOS X
00001011xxxxxxxxxxxxxxxxxxxxxxxx	0BXXXXXX	OP11_1	JNZE X
00001100xxxxxxxxxxxxxxxxxxxxxxxx	0CXXXXXX	OP12_1	JZER X
00001101xxxxxxxxxxxxxxxxxxxxxxxx	0DXXXXXX	OP13_1	ADDD X
00001110xxxxxxxxxxxxxxxxxxxxxxxx	0EXXXXXX	OP14_1	SUBD X
00001111xxxxxxxxxxxxxxxxxxxxxxxx	0FXXXXXX	OP15_1	MUL X
11110000000000000000000000000000	F7000000	OP16_1	RETN
11110001000000000000000000000000	F0000000	OP17_1	PSHI
11110010000000000000000000000000	F1000000	OP18_1	POPI
11110011000000000000000000000000	F2000000	OP19_1	PUSH
11110100000000000000000000000000	F3000000	OP20_1	POP
11110101000000000000000000000000	F4000000	OP21_1	SWAP
11110110000000000000000000000000	F5000000	OP22_1	SETI
11110111000000000000000000000000	F6000000	OP23_1	RETI
11111000xxxxxxxxxxxxxxxxxxxxxxxx	F8XXXXXX	OP24_1	INSP X
11111001xxxxxxxxxxxxxxxxxxxxxxxx	F9XXXXXX	OP25_1	DESP X
11111010000000000000000000000000	FAXXXXXX	OP26_1	LDIO
11111011000000000000000000000000	FBXXXXXX	OP27_1	STIO
11111110000000000000000000000000	-	-	
11111101000000000000000000000000	-	-	
111111100000000000000000yyyyyyyy	FE00000YY	OP30_1	MUDA_MODO
111111110000000000000000yyyyyyyy	FF00000YY	OP31_1	MUDA_CONJ

Mnemônico	Instrução	Significado
LODD X	Carrega direto	ac := m[x]
STOD X	Armazena direto	m[x] := ac
LOCO X	Carrega constante	ac := $x(0 \ll x \ll 16777216)$
LODL X	Carrega local	ac := m[sp + x]
STOL X	Armazena local	m[x + sp] := ac
ADDL X	Adiciona local	ac := ac + m[sp + x]
SUBL X	Subtrai local	ac := ac - m[sp + x]
CALL X	Chama procedimento	sp := sp - 1; m[sp] := pc; pc := x
JUMP X	Desvia	pc := x
JNEG X	Desvia se negativo	If $ac < 0$ then pc := x
JPOS X	Desvia se positivo	If $ac > 0$ then pc := x
JNZE X	Desvia se não zero	If $ac \neq 0$ then pc := x
JZER X	Desvia se zero	If $ac = 0$ then pc := x
ADDD X	Adiciona direto	ac := ac + m[x]
SUBD X	Subtrai direto	ac := ac - m[x]
MUL X	Mulriplica direto	ac := ac * m[x]
RETN	Retorno de subrotina	pc := m[sp]; sp := sp + 1
PSHI	Empilha direto	sp := sp - 1; m[sp] := m[ac]
POPI	Desempilha direto	m[ac] := m[sp]; sp := sp + 1
PUSH	Coloca na pilha	sp := sp - 1; m[sp] := ac
POP	Retira da pilha	ac := m[sp]; sp := sp + 1
SWAP	Troca ac, sp	tmp:= ac; ac := sp; sp := tmp
SETI	Inverte estado de IE	ie := inv(ie); armazena PC, AC, SP
RETI	Retorno da interrupção	ie := 0; restaura PC, AC, SP
INSP X	Incrementa sp	sp := sp + X
DESP X	Decrementa sp	sp := sp - X
LDIO	Carrega de IO	ac := io[y]
STIO	Armazena em IO	io[y] := ac
MUDA_MODO	Altera modo de funcionamento	
MUDA_CONJ	Altera Conjunto de Instrução	

D.2 Conjunto 2

Binário	Hexadecimal	Inst. Inicial	Mnemônico
00000000xxxxxxxxxxxxxxxxxxxxxxxx	00XXXXXX	OP0_1	LODD X
00000001xxxxxxxxxxxxxxxxxxxxxxxx	01XXXXXX	OP1_1	STOD X
00000010xxxxxxxxxxxxxxxxxxxxxxxx	02XXXXXX	OP2_1	LOCO X
00000011xxxxxyyyyyyyyyyyyyyyyyyy	03XXYYYY	OP3_1	LODL Rx, Y
00000100xxxxxyyyyyyyyyyyyyyyyyyy	04XXYYYY	OP4_1	STOL Y, Rx
00000101aaaaaxxxxxyyyyyzzzzzzzzz	05AAXXYY	OP5_1	LODR Ra, Rx, Ry
00000110aaaaaxxxxxyyyyyzzzzzzzzz	06AAXXYY	OP6_1	STOR Rx, Ry, Ra
00000111xxxxxxxxxxxxxxxxxxxxxxxx	07XXXXXX	OP7_1	CALL X
00001000xxxxxxxxxxxxxxxxxxxxxxxx	08XXXXXX	OP8_1	JUMP X
00001001xxxxxxxxxxxxxxxxxxxxxxxx	09XXXXXX	OP9_1	JNEG X
00001010xxxxxxxxxxxxxxxxxxxxxxxx	0AXXXXXX	OP10_1	JPOS X
00001011xxxxxxxxxxxxxxxxxxxxxxxx	0BXXXXXX	OP11_1	JNZE X
00001100xxxxxxxxxxxxxxxxxxxxxxxx	0CXXXXXX	OP12_1	JZER X
00001101aaaaaxxxxxyyyyyzzzzzzzzz	0DXXXXXX	OP13_1	ADDR Ra, Rx, Ry
00001110aaaaaxxxxxyyyyyzzzzzzzzz	0EXXXXXX	OP14_1	SUBR Ra, Rx, Ry
00001111aaaaaxxxxxyyyyyzzzzzzzzz	0FXXXXXX	OP15_1	MULR Ra, Rx, Ry
11110000000000000000000000000000	F7000000	OP16_1	RETN
11110001xxxxxzzzzzzzzzzzzzzzzzzz	F0000000	OP17_1	SHL Rx, z
11110010xxxxxzzzzzzzzzzzzzzzzzzz	F1000000	OP18_1	SHR Rx, z
11110011000000000000000000000000	F2000000	OP19_1	PUSH
11110100000000000000000000000000	F3000000	OP20_1	POP
11110101000000000000000000000000	F4000000	OP21_1	SWAP
11110110000000000000000000000000	F5000000	OP22_1	SETI
11110111000000000000000000000000	F6000000	OP23_1	RETI
11111000xxxxxxxxxxxxxxxxxxxxxxxx	F8XXXXXX	OP24_1	INSP X
11111001xxxxxxxxxxxxxxxxxxxxxxxx	F9XXXXXX	OP25_1	DESP X
11111010000000000000000000000000	FAXXXXXX	OP26_1	LDIO
11111011000000000000000000000000	FBXXXXXX	OP27_1	STIO
11111100000000000000000000000000	-	-	
11111101000000000000000000000000	-	-	
111111100000000000000000yyyyyyyy	FE00000YY	OP30_1	MUDA_MODO
111111110000000000000000yyyyyyyy	FF00000YY	OP31_1	MUDA_CONJ

Mnemônico	**Instrução**	**Significado**
LODD X	Carrega direto	ac := m[x]
STOD X	Armazena direto	m[x] := ac
LOCO X	Carrega constante	ac := $x(0 \ll x \ll 16777216)$
LODL Rx, Y	Carrega local	Rx := m[fp + y]; $x(0 \ll y \ll 524288)$
STOL Y, Rx	Armazena local	m[fp + y] := Rx
LODR Ra, Rx, Ry	Carrega registrador	Ra := m[Rx + Ry]
STOR Ra, Rx, Ry	Armazena registrador	m[Rx + Ry] := Ra
CALL X	Chama procedimento	sp := sp - 1; m[sp] := pc; pc := x
JUMP X	Desvia	pc := x
JNEG X	Desvia se negativo	If $ac < 0$ then pc := x
JPOS X	Desvia se positivo	If $ac > 0$ then pc := x
JNZE X	Desvia se não zero	If $ac \neq 0$ then pc := x
JZER X	Desvia se zero	If $ac = 0$ then pc := x
ADDR Ra, Rx, Ry	Adiciona registradores	Ra := Rx + Ry
SUBR Ra, Rx, Ry	Subtrai registradores	Ra := Rx - Ry
MULR Ra, Rx, Ry	Mulriplica registradores	Ra := Rx * Ry
RETN	Retorno de subrotina	pc := m[sp]; sp := sp + 1
PSHI	Desloca à esquerda	ac := Rx « z
POPI	Desloca à direita	ac := Rx » z
PUSH	Coloca na pilha	sp := sp - 1; m[sp] := ac
POP	Retira da pilha	ac := m[sp]; sp := sp + 1
SWAP	Troca ac, sp	tmp:= ac; ac := sp; sp := tmp
SETI	Inverte estado de IE	ie := inv(ie); armazena PC, AC, SP
RETI	Retorno da interrupção	ie := 0; restaura PC, AC, SP
INSP X	Incrementa sp	sp := sp + X
DESP X	Decrementa sp	sp := sp - X
LDIO	Carrega de IO	ac := io[y]
STIO	Armazena em IO	io[y] := ac
MUDA_MODO	Altera modo de funcionamento	
MUDA_CONJ	Altera Conjunto de Instrução	

D.3 Conjunto 3

Binário	Hexadecimal	Inst. Inicial	Mnemônico
00000000xxxxxxxxxxxxxxxxxxxxxxxx	00XXXXXX	OP0_1	LODDp X
00000001xxxxxxxxxxxxxxxxxxxxxxxx	01XXXXXX	OP1_1	STODp X
00000010xxxxxxxxxxxxxxxxxxxxxxxx	02XXXXXX	OP2_1	LOCO X
00000011xxxxxxxxxxxxxxxxxxxxxxxx	03XXXXXX	OP3_1	LODLp X
00000100xxxxxxxxxxxxxxxxxxxxxxxx	04XXXXXX	OP4_1	STOLp X
00000101xxxxxxxxxxxxxxxxxxxxxxxx	05XXXXXX	OP5_1	LODI X
00000110xxxxxxxxxxxxxxxxxxxxxxxx	06XXXXXX	OP6_1	STOI X
00000111xxxxxxxxxxxxxxxxxxxxxxxx	07XXXXXX	OP7_1	CALL X
00001000xxxxxxxxxxxxxxxxxxxxxxxx	08XXXXXX	OP8_1	JUMP X
00001001xxxxxxxxxxxxxxxxxxxxxxxx	09XXXXXX	OP9_1	JNEG X
00001010xxxxxxxxxxxxxxxxxxxxxxxx	0AXXXXXX	OP10_1	JPOS X
00001011xxxxxxxxxxxxxxxxxxxxxxxx	0BXXXXXX	OP11_1	JNZE X
00001100xxxxxxxxxxxxxxxxxxxxxxxx	0CXXXXXX	OP12_1	JZER X
00001101000000000000000000000000	0D000000	OP13_1	ADDp
00001110000000000000000000000000	0E000000	OP14_1	SUBp
00001111000000000000000000000000	0F000000	OP15_1	MULp
11110000000000000000000000000000	F7000000	OP16_1	RETN
11110001000000000000000000000000	F0000000	OP17_1	SHLp
11110010000000000000000000000000	F1000000	OP18_1	SHRp
11110011000000000000000000000000	F2000000	OP19_1	PUSH
11110100000000000000000000000000	F3000000	OP20_1	POP
11110101000000000000000000000000	F4000000	OP21_1	SWAP
11110110000000000000000000000000	F5000000	OP22_1	SETI
11110111000000000000000000000000	F6000000	OP23_1	RETI
11111000xxxxxxxxxxxxxxxxxxxxxxxx	F8XXXXXX	OP24_1	INSP X
11111001xxxxxxxxxxxxxxxxxxxxxxxx	F9XXXXXX	OP25_1	DESP X
11111010000000000000000000000000	FAXXXXXX	OP26_1	LDIO
11111011000000000000000000000000	FBXXXXXX	OP27_1	STIO
11111100000000000000000000000000	-	-	
11111101000000000000000000000000	-	-	
111111100000000000000000yyyyyyyy	FE00000YY	OP30_1	MUDA_MODO
111111110000000000000000yyyyyyyy	FF00000YY	OP31_1	MUDA_CONJ

Mnemônico	Instrução	Significado
LODDp X	Carrega direto usando pilha	push m[x]
STODp X	Armazena direto usando pilha	pop m[x]
LOCO X	Carrega constante	ac := $x(0 \ll x \ll 16777216)$
LODLp X	Carrega local usando pilha	push m[fp+y]]
STOLp X	Armazena local usando pilha	pop m[fp+y]
LODI X	Carrega indireto	pop x; push m[x]
STOI X	Armazena indireto	pop dado; pop x; m[x] = dado
CALL X	Chama procedimento	sp := sp - 1; m[sp] := pc; pc := x
JUMP X	Desvia	pc := x
JNEG X	Desvia se negativo	If $ac < 0$ then pc := x
JPOS X	Desvia se positivo	If $ac > 0$ then pc := x
JNZE X	Desvia se não zero	If $ac \neq 0$ then pc := x
JZER X	Desvia se zero	If $ac = 0$ then pc := x
ADDp	Adiciona registradores	a = m[sp]; sp = sp + 1;
		b = m[sp];
		m[sp] = m[b + a]
SUBp	Subtrai registradores	a = m[sp]; sp = sp + 1;
		b = m[sp];
		m[sp] = m[b - a]
MULp	Mulriplica registradores	a = m[sp]; sp = sp + 1;
		b = m[sp];
		m[sp] = m[b * a]
SHLp	Desloca à esquerda	a = m[sp]; ; a « Z; m[sp] = m[a]
SHRp	Desloca à esquerda	a = m[sp]; ; a » Z; m[sp] = m[a]
PUSH	Coloca na pilha	sp := sp - 1; m[sp] := ac
POP	Retira da pilha	ac := m[sp]; sp := sp + 1
SWAP	Troca ac, sp	tmp:= ac; ac := sp; sp := tmp
SETI	Inverte estado de IE	ie := inv(ie); armazena PC, AC, SP
RETI	Retorno da interrupção	ie := 0; restaura PC, AC, SP
RETN	Retorno de subrotina	pc := m[sp]; sp := sp + 1
INSP X	Incrementa sp	sp := sp + X
DESP X	Decrementa sp	sp := sp - X
LDIO	Carrega de IO	ac := io[y]
STIO	Armazena em IO	io[y] := ac
MUDA_MODO	Altera modo de funcionamento	
MUDA_CONJ	Altera Conjunto de Instrução	

APÊNDICE E – Microinstruções dos conjuntos de Instruções Fixos

Os Códigos referentes aos estados estão baseados nos códigos do fluxograma da Máquina de Estados Global

E.1 Conjunto de instrução 1

Estado	**Código**	**Microinstrução**
OP0_1	20	MAR := IR; RD;
OP0_2	21	Acc := MBR; GOTO INT (TESTE INTERRUPCAO);
OP0_3	22	MAR := SP; WR; GOTO INT
OP1_1	23	MAR := IR; MBR := Acc; WR; GOTO INT;
OP2_1	26	MAR := SP + IR; RD; GOTO OP0_2
OP3_1	27	MAR := SP + IR; RD; GOTO OP0_2
OP4_1	30	A := SP + IR;
OP4_2	31	MBR := AC; MAR := A; WR; GOTO INT
OP5_1	34	MAR := SP + IR; RD; GOTO CONT ADD
OP6_1	37	MAR := SP + IR; RD; GOTO CONT SUB;
OP7_1	43	SP := SP - 1;
OP7_2	44	MAR := SP; MBR := PC; WR; GOTO JUMP
OP8_1	45	PC := IR(XXXXXXXXXXXX); GOTO INT;
OP9_1	46	ALU := Acc; IF N THEN GOTO JUMP ELSE GOTO INT;
OP10_1	47	ALU := Acc; IF N THEN GOTO INT ELSE GOTO JUMP;
OP11_1	48	ALU := Acc; IF Z THEN GOTO INT ELSE GOTO JUMP;
OP12_1	49	ALU := Acc; IF Z THEN GOTO JUMP ELE GOTO INT;

```
OP13_1    50    MAR := IR; RD;
OP13_2    51    AC := MBR + AC; GOTO INT;
OP14_1    57    MAR := IR; RD;
OP14_2    58    A := MBR;
OP14_3    59    Acc := Acc - A; GOTO INT
OP15_1    64    MAR := IR; RD;
OP15_2    65    A := MBR;
OP15_3    66    Acc := Acc * A; GOTO INT
OP16_1    71    MAR := SP; PC := MBR; RD
OP16_2    72    SP := SP + 1; GOTO INT
OP17_1    73    MAR := AC; RD;
OP17_2    74    SP := SP - 1
OP17_3    75    MAR := SP; WR; GOTO INT
OP18_1    77    MAR := SP; RD;
OP18_2    78    SP := SP + 1;
OP18_3    79    MAR := AC; WR; GOTO INT
OP19_1    80    SP := SP - 1
OP19_2    81    MAR := SP; MBR := AC; WR; GOTO INT
OP20_1    82    MAR := SP; RD;
OP20_2    83    SP := SP + 1
OP20_3    84    Acc := MBR; GOTO INT
OP21_1    85    A := AC;
OP21_2    86    AC := SP;
OP21_3    87    SP := A; GOTO INT
OP22_1    88    IE := INV(IE); GOTO INT
OP23_1    89    MAR := SP; RD;
OP23_2    90    SP := SP + 1; PC := MBR;
OP23_3    91    MAR := SP; RD;
OP23_4    92    SP := SP + 1;
OP23_5    93    Acc := MBR;
OP23_6    94    MAR := SP; RD;
OP23_7    95    SP := SP + 1;
OP23_8    96    SP := MBR
OP23_9    97    IE := 1; GOTO INT;
```

OP24_1	98	SP := SP + BAND(SMASK, IR); GOTO INT
OP25_1	99	SP := SP - BAND(SMASK, IR); GOTO INT
OP26_1	100	MAR := IR; RD_IO
OP26_2	101	Acc := MBR; GOTO INT;
OP27_1	102	MAR := IR; MBR := Acc; WR_IO; GOTO INT
Muda conj.	50	muda_conj
Muda modo	51	muda_modo
Read1	116	Busca valor na memória
Read2	117	Recebe valor da memória
Write	118	Escreve valor na memória

Total: 59 estados

E.2 Conjunto de instrução 2

Estado	Código	Microinstrução
OP0_1	20	MAR := IR; RD;
OP0_2	21	Acc := MBR; GOTO INT (TESTE INTERRUPCAO);
OP1_1	23	MAR := IR; MBR := Acc; WR; GOTO INT;
OP2_1	26	MAR := SP + IR; RD; GOTO OP0_2
OP3_1	27	MAR := BAND(SMASK, IR) + FP; RD;
OP3_2	28	Rx := MBR; GOTO INT;
OP4_1	30	Acc := BAND(SMASK, IR) + FP;
OP4_2	31	MBR := Rx; MAR := Acc; WR; GOTO INT
OP5_1	34	MAR := Rx + Ry; RD;
OP5_2	35	Ra := MBR; GOTO INT;
OP6_1	37	Acc := Rx + Ry;
OP6_2	38	MBR := Ra; MAR := Acc; WR; GOTO INT
OP7_1	43	SP := SP - 1;
OP7_2	44	MAR := SP; MBR := PC; WR; GOTO JUMP
OP8_1	45	PC := IR(XXXXXXXXXXXX); GOTO INT;
OP9_1	46	ALU := Acc; IF N THEN GOTO JUMP ELSE GOTO INT;
OP10_1	47	ALU := Acc; IF N THEN GOTO INT ELSE GOTO JUMP;
OP11_1	48	ALU := Acc; IF Z THEN GOTO INT ELSE GOTO JUMP;
OP12_1	49	ALU := Acc; IF Z THEN GOTO JUMP ELE GOTO INT;
OP13_1	50	Ra := Rx + Ry; GOTO INT;
OP14_1	57	Ra := Rx - Ry; GOTO INT;
OP15_1	64	Ra := Rx * Ry; GOTO INT;
OP16_1	71	MAR := SP; PC := MBR; RD
OP16_2	72	SP := SP + 1; GOTO INT
OP17_1	73	Ra := Rx « Z; GOTO INT
OP18_1	77	Ra := Rx » Z; GOTO INT
OP19_1	80	SP := SP - 1
OP19_2	81	MAR := SP; MBR := AC; WR; GOTO INT

OP20_1	82	MAR := SP; RD;
OP20_2	83	SP := SP + 1
OP20_3	84	Acc := MBR; GOTO INT
OP21_1	85	A := AC;
OP21_2	86	AC := SP;
OP21_3	87	SP := A; GOTO INT
OP22_1	88	IE := INV(IE); GOTO INT
OP23_1	89	MAR := SP; RD;
OP23_2	90	SP := SP + 1; PC := MBR;
OP23_3	91	MAR := SP; RD;
OP23_4	92	SP := SP + 1;
OP23_5	93	Acc := MBR;
OP23_6	94	MAR := SP; RD;
OP23_7	95	SP := SP + 1;
OP23_8	96	SP := MBR
OP23_9	97	IE := 1; GOTO INT;
OP24_1	98	SP := SP + BAND(SMASK, IR); GOTO INT
OP25_1	99	SP := SP - BAND(SMASK, IR); GOTO INT
OP26_1	100	MAR := IR; RD_IO
OP26_2	101	Acc := MBR; GOTO INT;
OP27_1	102	MAR := IR; MBR := Acc; WR_IO; GOTO INT
Muda conj.	50	muda_conj
Muda modo	51	muda_modo
Read1	116	Busca valor na memória
Read2	117	Recebe valor da memória
Write	118	Escreve valor na memória

Totsl: 53 estados

E.3 Conjunto de instrução 3

Estado	Código	Microinstrução
OP0_1	20	MAR := IR; RD;
OP0_2	21	SP := SP - 1;
OP0_3	22	MAR := SP; WR; GOTO INT
OP1_1	23	MAR := SP; RD
OP1_2	24	SP := SP + 1;
OP1_3	25	MAR := IR; WR; GOTO INT
OP2_1	26	ACC := BAND(AMASK, IR); GOTO INT;
OP3_1	27	MAR := BAND(SMASK, IR) + FP; RD;
OP3_2	28	SP := SP - 1
OP3_3	29	MAR := SP; WR; GOTO INT
OP4_1	30	MAR := SP; RD;
OP4_2	31	SP := SP + 1
OP4_3	32	A := MBR
OP4_4	33	FP := A - BAND(SMASK, IR); GOTO INT;
OP5_1	34	MAR := SP; RD
OP5_2	35	MAR := MBR; RD
OP5_3	36	MAR := SP; WR; GOTO INT
OP6_1	37	POP (A) (MAR := SP; RD);
OP6_2	38	CONT POP(A) (SP := SP + 1)
OP6_3	39	CONT POP(A) (A := MBR)
OP6_4	40	POP (A) (MAR := SP; RD);
OP6_5	41	CONT POP(ACC) (ACC := MBR)
OP6_6	42	MAR := A; MBR := ACC; WR; GOTO INT
OP7_1	43	SP := SP - 1;
OP7_2	44	MAR := SP; MBR := PC; WR; GOTO JUMP
OP8_1	45	PC := IR(XXXXXXXXXXXX); GOTO INT;
OP9_1	46	ALU := Acc; IF N THEN GOTO JUMP ELSE GOTO INT;
OP10_1	47	ALU := Acc; IF N THEN GOTO INT ELSE GOTO JUMP;
OP11_1	48	ALU := Acc; IF Z THEN GOTO INT ELSE GOTO JUMP;
OP12_1	49	ALU := Acc; IF Z THEN GOTO JUMP ELE GOTO INT;

```
OP13_1   50   MAR := SP; RD
OP13_2   51   SP := SP + 1;
OP13_3   52   A := MBR;
OP13_4   53   MAR := SP; RD
OP13_5   54   B := MBR;
OP13_6   55   ACC := A + B;
OP13_7   56   MAR := SP; MBR := ACC; WR; GOTO INT
OP14_1   57   MAR := SP; RD
OP14_2   58   SP := SP + 1;
OP14_3   59   A := MBR;
OP14_4   60   MAR := SP; RD
OP14_5   61   B := MBR;
OP14_6   62   Acc := A - B;
OP14_7   63   MAR := SP; MBR := ACC; WR; GOTO INT
OP15_1   64   MAR := SP; RD
OP15_2   65   SP := SP + 1;
OP15_3   66   A := MBR;
OP15_4   67   MAR := SP; RD
OP15_5   68   B := MBR;
OP15_6   69   Acc := A * B;
OP15_7   70   MAR := SP; MBR := ACC; WR; GOTO INT
OP16_1   71   MAR := SP; PC := MBR; RD
OP16_2   72   SP := SP + 1; GOTO INT
OP17_1   73   MAR := SP; RD
OP17_2   74   A := MBR;
OP17_3   75   Acc := A « Z; GOTO 0P17_4
OP17_4   76   MAR := SP; MBR := ACC; WR; GOTO INT
OP18_1   77   MAR := SP; RD;
OP18_2   78   A := MBR;
OP18_3   79   Acc := A » Z; GOTO 17_4
OP19_1   80   SP := SP - 1
OP19_2   81   MAR := SP; MBR := AC; WR; GOTO INT
```

OP20_1	82	MAR := SP; RD;
OP20_2	83	SP := SP + 1
OP20_3	84	Acc := MBR; GOTO INT
OP21_1	85	A := AC;
OP21_2	86	AC := SP;
OP21_3	87	SP := A; GOTO INT
OP22_1	88	IE := INV(IE); GOTO INT
OP23_1	89	MAR := SP; RD;
OP23_2	90	SP := SP + 1; PC := MBR;
OP23_3	91	MAR := SP; RD;
OP23_4	92	SP := SP + 1;
OP23_5	93	Acc := MBR;
OP23_6	94	MAR := SP; RD;
OP23_7	95	SP := SP + 1;
OP23_8	96	SP := MBR
OP23_9	97	IE := 1; GOTO INT;
OP24_1	98	SP := SP + BAND(SMASK, IR); GOTO INT
OP25_1	99	SP := SP - BAND(SMASK, IR); GOTO INT
OP26_1	100	MAR := IR; RD_IO
OP26_2	101	Acc := MBR; GOTO INT;
OP27_1	102	MAR := IR; MBR := Acc; WR_IO; GOTO INT
Muda conj.	50	muda_conj
Muda modo	51	muda_modo
Read1	116	Busca valor na memória
Read2	117	Recebe valor da memória
Write	118	Escreve valor na memória

Total: 87 estados

APÊNDICE F – Codificação dos formatos de operações

Esta codificação foi descrita para operações do tipo **Op C, A, B, D**. Os bits da palavra de reconfiguração seguem a seqüência Bar C (5 bits), Bar A (5 bits), Bar B (5 bits), Aux (5 bits).

Formato (Bin)	Formato (Dec)	BAR C	BAR A	BAR B	Efeito
000000	0	A	A	A	A = A op A
000001	1	A	A	B	A = A op A
000010	2	A	B	A	A = A op B
000011	3	A	B	B	A = B op B
000100	4	A	A	C	A = A op C
000101	5	A	C	A	A = C op A
000110	6	A	C	C	A = C op C
000111	7	A	C	B	A = C op B
001000	8	A	B	C	A = B op C
001001	9	B	B	B	B = B op B
001010	10	B	B	A	B = B op A
001011	11	B	A	B	B = A op B
001100	12	B	A	A	B = A op A
001101	13	B	B	C	B = B op C
001110	14	B	C	B	B = C op B
001111	15	B	C	C	B = C op C
010000	16	B	A	C	B = A op C
010001	17	B	C	A	B = C op A

Formato (Bin)	Formato (Dec)	BAR C	BAR A	BAR B	Efeito
010010	18	C	C	C	C = C op C
010011	19	C	C	A	C = C op A
010100	20	C	A	C	C = A op C
010101	21	C	A	A	C = A op A
010110	22	C	B	C	C = B op C
010111	23	C	C	B	C = C op B
011000	24	C	B	B	C = B op B
011001	25	C	A	B	C = A op B
011010	26	C	B	A	C = B op A
011011	27	A	A	D	A = A op D
011000	28	A	B	D	A = B op D
011101	29	A	C	D	A = C op D
011110	30	A	D	A	A = D op A
011111	31	A	D	B	A = D op B
100000	32	A	D	C	A = D op C
100001	33	A	D	D	A = D op D
100010	34	B	A	D	B = A op D
100011	35	B	B	D	B = B op D
100100	36	B	C	D	B = C op D
100101	37	B	D	A	B = D op A
100110	38	B	D	B	B = D op B
100111	39	B	D	C	B = D op C
101000	40	B	D	D	B = D op D

Formato (Bin)	**Formato (Dec)**	**BAR C**	**BAR A**	**BAR B**	**Efeito**
101001	41	C	A	D	C = A op D
101010	42	C	B	D	C = B op D
101011	43	C	C	D	C = C op D
101100	44	C	D	A	C = D op A
101101	45	C	D	B	C = D op B
101110	46	C	D	C	C = D op C
101111	47	C	D	D	C = D op D
110000	48	D	A	D	D = A op D
110001	49	D	D	A	D = D op A
110010	50	D	B	D	D = B op D
110011	51	D	D	B	D = D op A
110100	52	D	C	D	D = C op D
110101	53	D	D	C	D = X op C
110110	54	D	D	D	D = D op D
110111	55	D	A	B	D = A op B
111000	56	D	B	A	D = B op A
111001	57	D	A	C	D = A op C
111010	58	D	C	A	D = C op A
111011	59	D	B	C	D = B op C
111100	60	D	C	B	D = C op B
111101	61	D	A	A	D = A op A
111110	62	D	B	B	D = B op B
111111	63	D	C	C	D = C op C

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo