FUNDAÇÃO DE ENSINO “EURÍPEDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPEDES DE MARÍLIA” – UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

MARCIO HENRIQUE CASTILHO CARDIM

RTRASSOC51 - MÓDULO DE PIPELINE PARA UM PROCESSADOR COM ARQUITETURA HARVARD SUPERESCALAR EMBARCADO (PAHSE)

MARÍLIA
2005

MARCIO HENRIQUE CASTILHO CARDIM

RTRASSOC51: MÓDULO DE PIPELINE PARA UM PROCESSADOR COM ARQUITETURA HARVARD SUPERESCALAR EMBARCADO (PAHSE)

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípedes de Marília, mantido pela Fundação de Ensino Eurípedes Soares Da Rocha, para obtenção do Título de Mestre em Ciência da Computação. (Área de Concentração: Arquitetura de Sistemas Computacionais).

Orientador:
Prof. Dr. Jorge Luiz e Silva

MARÍLIA
2005

Para minha querida esposa
Nívea
e minha amada filha
Ana Beatriz.

AGRADECIMENTOS

Obrigado Senhor, pela força e onipresença.

Agradecimento à Nívea, pelo companheirismo e pela resignação e aceitação desta fase.

Volto a agradecer a Deus pela filha maravilhosa que nos enviou no inicio desta jornada.

Gratidão a toda minha família, mas de modo especial e individual ao meu pai, minha mãe, e minha irmã, responsáveis por minha formação moral.

Toda minha gratidão para o Roberto Correa da Silva, que sempre acreditou no meu crescimento pessoal e profissional.

Todo meu respeito ao Dr. Jorge Luiz e Silva, o sentimento de orgulho em ter sido seu orientando, somente motiva a continuar a caminhada neste instigante mundo da pesquisa.

Aos companheiros, Prof. Ms. Marcelo Storion e Prof. Ricardo Veronesi, parceiros em toda a caminhada.

Agradecimentos a todos que direta ou indiretamente estiveram presentes na minha vida durante estes 30 meses.

"As pessoas dividem-se
em duas categorias:
umas procuram e não encontram,
outras encontram mas não
ficam satisfeitas."
Eminescu

CARDIM, Marcio H. C. **RtrASSoc51: Módulo de Pipeline para um Processador com Arquitetura Harvard Superescalar Embarcado (PAHSE).** 2005. 71 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

RESUMO

Aplicações embarcadas são sistemas que se caracterizam como elementos essenciais em produtos que estão presentes em quase tudo em nossas vidas: aviões, automóveis, monitoração médica, TV´s digitais, celulares, videogames, impressoras, copiadoras, fax, telefones, etc, muitos desses contendo microprocessadores ou microcontroladores, que estão sendo interligado uns aos outros processando informações. O RtrASSoc51 é um Sistema em Chip, Adaptável, Superescalar, e Reconfigurável, em desenvolvimento, sendo implementado em FPGA Virtex da Xilinx e será utilizado em aplicações embarcadas para reconhecimento de objetos. Esta dissertação apresenta a implementação do PAHSE, Processador com Arquitetura Harvard Superescalar Embutido, um CORE que configura a CPU do RtrASSoc51, implementado com uma arquitetura Harvard, e três linhas de pipeline. As instruções do processador foram baseadas nas instruções do microcontrolador 8051. O PAHSE foi implementado em VHDL e resultados de simulação, bem como, propostas futuras são apresentadas no final desta dissertação.

Palavras-chave: FPGA, VHDL, System on Chip, RtrASSoc51, Arquitetura Harvard, Superescalar.

CARDIM, Marcio H. C. **RtrASSoc51: Pipeline Module for Processor with Architecture Harvard Embedded Superscalar(PAHSE).** 2005. 71 f. Dissertation (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

ABSTRACT

Embedded applications are systems that are characterized as essential elements in products that are present in almost everything in our lives: airplanes, automobiles, medical machines, digital TV´s, cellular, videogames, printers, copiers, fax, telephones, etc, many of those containing microprocessors or microcontrolers, that each other are being interconnected processing information. RtrASSoc51 is a System in Chip, Adaptable, Superscalar, and Reconfigurable, in development, being implemented in FPGA Virtex of Xilinx and it will be used in embedded applications for recognition of objects. This dissertation presents the implementation of PAHSE, Processor with Architecture Harvard Embedded Superescalar, one CORE that it configures CPU of RtrASSoc51, implemented with an architecture Harvard, and three lines of pipeline. The instructions of the processor were based on the instructions of the microcontrolador 8051. PAHSE was implemented in VHDL and simulation results, as well as, proposed future are presented in the end of this dissertation.

Keywords: FPGA, VHDL, System on Chip, RtrASSoc, Harvard Architecture, Superscalar.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

A/D	Analógico para Digital
ALU	Unidade Lógica Aritmética
AMBA	Advanced Microcontroller Bus Architecture
API	Application Program Interface
ASIC	Application Specific Integrated Circuit
CAN	Controller Area Network
CI	Circuito Integrado
CLB	Configurable Logig Block
CLP	Controlador Lógico Programável
CPLD	Complex Programmable Logic Device

CRC	Cyclic Redundancy Check
D/A	Digital para Analógico
DISC	Dynamic Instruction Set Computer
DPGA	Dinamically Programmable Gate Array
DSP	Digital Signal Processing
E/S	Entrada e Saída
EDA	Eletronic Design Automation
EEPROM	Electrically Erasable PROM
EPLD	Erasable PLD
EPROM	Erasable PROM
ESB	Embedded System Block
FIFO	First-In First-Out
FIP	Factory Instrumentation Protocol
FIPSoC	Field Programmable SoC
FPGA	Field Programmable Gate Array
FPIC	Field Programmable Interconnect Component
FSM	Finite State Machine
GPP	General Poupouse Processor
HDL	Hardware Description Language
I2C	Inter Integrated Circuit
IOB	Input/Output Block
IOE	Input/Output Element
IRL	Internet Reconfigurable Logic
ISO	International Organization for Standardization,
LUT	Look-Up Table

MCU	Memory Control Unit
MPGA	Mask Programmable Gate Array
OSI	Open Systems Interconnection
PAL	Programmable Array of Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PROFIBUS	Process Fieldbus, um protocolo de barramento industrial
PROM	Programmable Read-Only Memory
rI2C	Reconfigurable Inter Integrated Circuit
RPU	Reconfigurable Processing Unit
RTL	Register Transfer Level
RTR	Run-Time Reconfiguration
SCL	Clock Signal
SDA	Data Signal
SoC	System-on-Chip
SoPC	System-on-a-Programable Chip
SRAM	Static Random Access Memory
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
ULA	Unidade Lógica e Aritmética
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

SUMÁRIO

1. INTRODUÇÃO

Muitas aplicações emergentes em telecomunicações e multimídia necessitam que suas funcionalidades permaneçam flexíveis mesmo depois do sistema ter sido manufaturado. Tal flexibilidade é fundamental, uma vez que requisitos dos usuários, características dos sistemas, padrões e protocolos podem mudar durante a vida do produto. Essa maleabilidade também pode prover novas abordagens de implementação voltadas para ganhos de desempenho, redução dos custos do sistema ou redução do consumo geral de energia (MESQUITA,2002).

Existe uma série de abordagens para se implementar um circuito digital. As abordagens utilizadas no design tendem desde a utilização de circuitos integrados com portas lógicas primárias (AND, OR etc) e até mesmo a total customização de um chip. Este último seria a produção de um chip especialmente montado para a solução de um problema específico.

Torna-se claro que quanto maior a customização maior a eficiência, no entanto os custos se elevam consideravelmente. Neste ponto entra a visão da Engenharia de se encontrar a solução mais apropriada para um determinado projeto. Uma das alternativas entre o genérico e o totalmente customizado é a abordagem por FPGA.

O FPGA evita os altos custos e o demorado tempo de fabricação dos sistemas customizados em troca de se obter uma menor densidade de portas. Estas portas se caracterizam por componentes cuja conectividade pode ser estabelecida através do simples carregamento de uma configuração de dados apropriada para a memória interna do dispositivo.

Muito recurso tem sido aplicado para definir sistemas reconfiguráveis baseados em FPGAs (ARNOLD, 1993; ARNOLD, 1992; ATHANAS,1992; QUENOT, 1994; GOKHALE, 1991; RATHA, 1995, WIRTHLIN,1995), tornando computação reconfigurável uma realidade. Da mesma forma os pesquisadores vêm despendendo muito esforço e tempo em

implementar esses sistemas, em especial sistemas reconfiguráveis gerados diretamente a partir de um programa escrito em linguagem C ou assembler.

O campo da computação reconfigurável avançou amplamente na década passada, utilizando FPGAs como a base para sistemas reprogramáveis de alto desempenho. Muitos desses sistemas alcançaram altos níveis de desempenho e demostraram sua aplicabilidade à resolução de uma grande variedade de problemas. Contudo, apesar dos autores desses sistemas os classificarem como reconfiguráveis, eles são tipicamente configurados uma vez antes de iniciarem a execução da aplicação.

Uma linguagem de descrição de hardware descreve o que um sistema faz e como. Esta descrição é um modelo do sistema hardware, que será executado em um software chamado simulador. Um sistema descrito em linguagem de hardware pode ser implementado em um dispositivo programável (FPGA), permitindo assim o uso em campo do seu sistema, tendo a grande vantagem da alteração do código a qualquer momento. O VHDL é uma linguagem de descrição de *hardware* que representa entradas e saídas, comportamento e funcionalidade de circuitos.

O RtrASSoc51 é um sistema em chip programável (SOPC) sendo implementado em VHDL para FPGAs Virtex da Xilinx tendo como base a família de microcontroladores 8051, acrescido de uma estrutura pipeline em três níveis a ser utilizado em aplicações embarcadas que necessitem de maior capacidade, melhor desempenho, e reconfiguração dinâmica podendo ser interconectado em rede (ARÓSTEGUE,2004; CARRILLO,2001; DOLPHIN,2003; MASSIMO,2002). A plataforma RtrASSoc51 vem sendo desenvolvida à três anos, inicialmente através da simulação e validação de alguns conceitos fundamentais como medida de desempenho do sistema para aplicações em processamento de imagens, e mais recentemente através da implementação de partes da plataforma e do mecanismo de

interconexão I2C (SILVA,2003; LOPES,2004; SILVA,2004; COSTA,2004; ZANGUETTIN,2004; FORNARI,2004).

A aplicação sendo desenvolvida consiste no reconhecimento de objetos através de redes neurais booleanas no modelo N-tuple (BONATO,2004). Uma câmera modelo CMOS será conectada à plataforma RtrASSoc51 que irá capturar imagens a serem comparadas a imagens pré-definidos de alguns objetos. Inicialmente o teste será realizado em uma única plataforma RtrASSoc51 e em seguida em um conjunto de plataformas RtrASSoc51 interconectados em rede em uma aplicação industrial (LOPES,2004).

Esta dissertação apresenta a implementação de um processador desenvolvido com arquitetura Harvard e uma estrutura superescalar com três níveis de pipeline embarcado. Este processador denominado PAHSE (Processador com Arquitetura Harvard Superescalar Embutido), foi desenvolvido na linguagem VHDL para um FPGA Virtex da Xilinx (CARDIM, 2005).

No capítulo 2 são apresentadas as arquiteturas dos microcontroladores 8051 e PIC, que foram utilizados como base da implementação do Processador com Arquitetura Harvard Superescalar Embutido (PAHSE).

O capítulo 3 apresenta a arquitetura dos processadores superescalares, seu funcionamento, seus problemas de execução e as evoluções para a máxima eficiência do processador.

O capítulo 4 descreve toda a tecnologia e as aplicações dos FPGAs e a família VirtexII que foram utilizadas no desenvolvimento do projeto.

O capítulo 5 apresenta a linguagem VHDL, utilizada na descrição do hardware implementado para o PAHSE

No capítulo 6 é apresentado o RtrASSoc, com toda sua estrutura detalhada e o estado da arte deste SOPC, e a implementação do Processador Harvard Superescalar Embutido (PHSE) com todos os seus detalhes.

Finalmente no capítulo 7 são apresentados resultados e as conclusões sobre o desenvolvimento do Processador Harvard Superescalar Embutido seguido de trabalhos futuros e referências bibliográficas.

2. MICROCONTROLADORES 8051 E PIC

A arquitetura de um sistema digital define quem são e como as partes que compõem o sistema estão interligadas. As duas arquiteturas mais comuns para sistemas computacionais digitais são as seguintes:

Arquitetura de Von Neuman: a Unidade Central de Processamento é interligada à memória por um único barramento. O sistema é composto por uma única memória onde são armazenados dados e instruções.

Arquitetura de Harvard: a Unidade Central de Processamento é interligada à memória de dados e à memória de programa por barramento específico.

2.1. Microcontrolador 8051

O 8051, da Intel, é, sem dúvida, um dos microcontroladores mais populares. O dispositivo em si é um microcontrolador de 8 bits relativamente simples, mas com ampla aplicação. Porém, o mais importante é que não existe somente o CI 8051, mas sim uma família de microcontroladores baseada no mesmo (8051 TUTORIAL, 2002). Entende-se família como sendo um conjunto de dispositivos que compartilha os mesmos elementos básicos, tendo também um mesmo conjunto básico de instruções.

Sistemas microcontrolados, como o 8051, são aqueles que têm por elemento central um microprocessador. O microprocessador funciona como um sistema seqüencial síncrono, onde a cada pulso, ou grupos de pulsos de clock, uma instrução é executada. Entre os microprocessadores mais conhecidos podemos citar o 8080 e 8085, Z-80, 8088, 8086, 80286, 68000, 80386 e superiores.

Embora já existam microprocessadores que trabalhem a centenas de milhares de MHz, o 8051 utiliza tipicamente um clock de 12 MHz, podendo chegar a 40 MHz, com tempos de execução de cada instrução variando entre 1ms e 4ms, podendo chegar a 25 vezes

mais rápido nos clocks de 40 MHz (8051 TUTORIAL, 2004).

A Figura 1 mostra um diagrama em blocos de um sistema genérico utilizando um microprocessador.

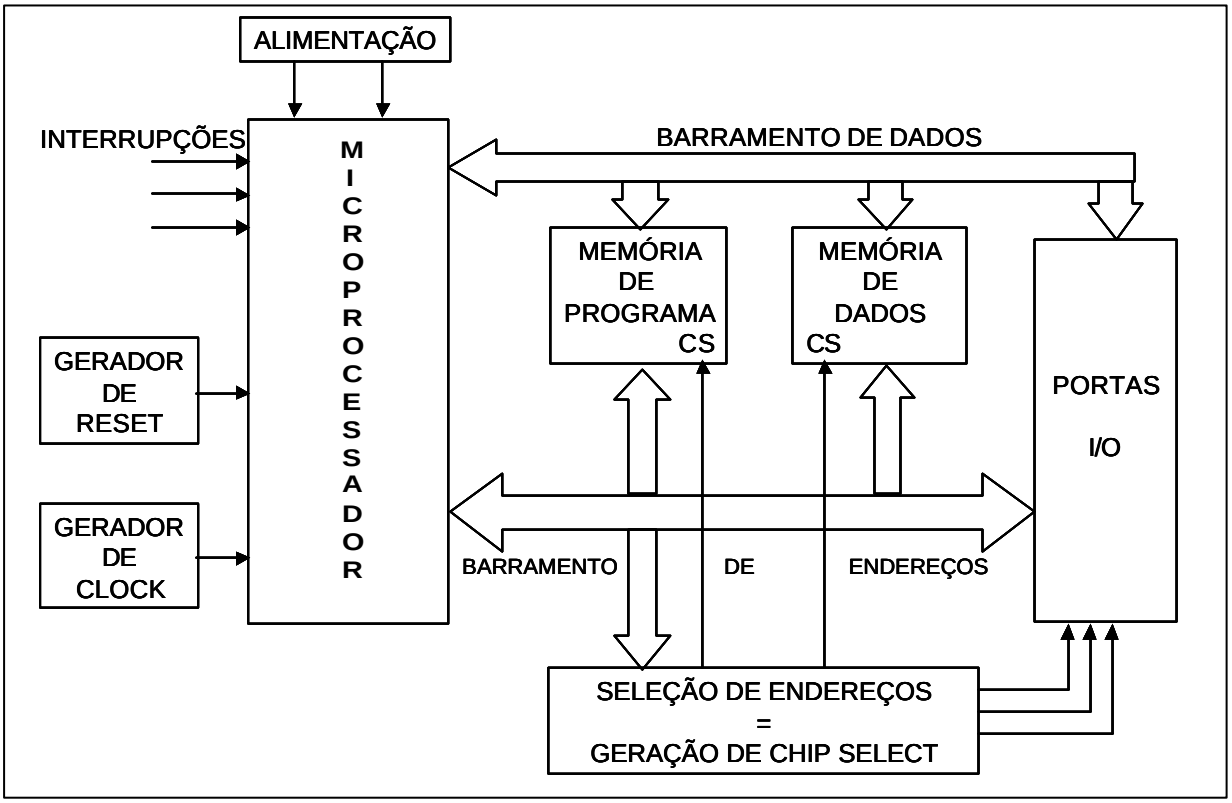

Figura 1: Diagrama de Blocos de um sistema genérico com microprocessador.

As Interrupções são entradas a partir de um sinal externo que fazem com que o processamento seja interrompido e seja iniciada uma sub-rotina específica. (Obs.: o 8051 tem interrupções com estrutura nesting, onde uma interrupção pode interromper outra que está sendo atendida, desde que tenha maior prioridade).

O Gerador de Reset é o responsável por inicializar o sistema ao ligar ou quando acionado. Quanto ao Gerador de Clock, é ele que gera os pulsos necessários ao sincronismo do sistema.

A Memória de Programa é onde o microprocessador vai procurar as instruções a executar. Em sistemas dedicados costuma-se utilizar memórias ROM, embora, em alguns casos, memórias RAM também sejam utilizadas.

É na Memória de Dados onde o microprocessador lê e escreve dados durante a operação normal. Geralmente é do tipo volátil, embora memórias não-voláteis possam ser utilizadas.

A Seleção de Endereços constitui a lógica para escolher qual memória ou periférico o microprocessador vai utilizar.

A função das Portas de I/O é realizar comunicação com o mundo externo. Através delas dispositivos como teclados, impressoras, displays, entre outros, comunicam-se com o sistema.

Um microcontrolador geralmente engloba todos esses elementos em um único chip, podendo utilizar elementos externos como expansão.

Como já foi citado, o 8051 é um microcontrolador de ampla utilização. O mesmo tem dois modos básicos de funcionamento.

O modo mínimo, onde somente recursos internos são utilizados pela CPU. Neste modo, estão disponíveis 4 KB de ROM para memória de programa e 128 bytes de RAM para memória de dados. O modo mínimo possui a vantagem (além da economia de componentes e espaço físico) de poder utilizar as quatro portas de 8 bits cada para controle (I/O).

Já no modo expandido, a memória de programa (ROM), a memória de dados (RAM), ou ambas podem ser expandidas para 64 KB, através do uso de CIs externos. No entanto, apresenta a desvantagem de "perder" duas das quatro portas para comunicação com as memórias externas.

A Figura 2 mostra, respectivamente, a organização da memória de programa e da memória de dados do 8051.

É importante salientar que, diferentemente de outros sistemas baseados em microprocessador, onde cada endereço de memória identifica uma única posição física, no 8051 o mesmo endereço hexadecimal pode identificar três posições físicas diferentes (e até

quatro, no caso do 8052, que tem 256 bytes de RAM interna). Por exemplo, temos o endereço 23H na RAM interna, o endereço 23H na RAM externa e o endereço 23H na ROM externa. Mesmo com esses endereços "iguais", não há conflito, pois as instruções e o modo de endereçamento são diferentes.

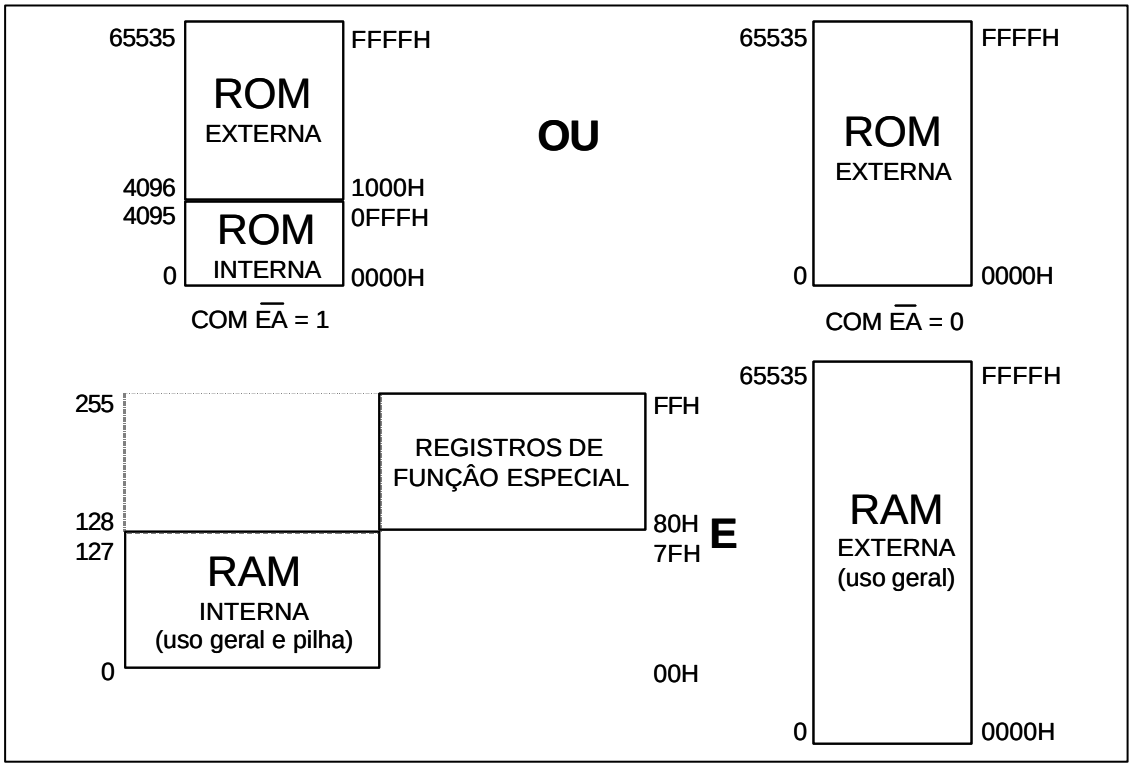

Figura 2: Organização da memória de programa e de dados do 8051.

A Figura 3 mostra em detalhes a divisão da RAM interna do 8051. Note a presença dos quatro bancos de registradores auxiliares, cada um contendo 8 registradores (R0, R1, R2, R3, R4, R5, R6, R7). Estes registradores são utilizados para endereçamento indireto. Apesar de existirem quatro bancos de registradores, somente um está ativo por vez para uso como índice. A seleção do banco ativo é feita no registro de função especial PSW, que será visto em breve.

Como pode ser visto pela Figura 4, os bytes 20 a 2F da RAM interna têm bits endereçáveis individualmente. Podemos, com os mesmos, executar várias instruções de bits.

Por exemplo, a instrução SETB 3CH coloca em nível alto o bit 3C, ou seja, o bit 4 da posição de memória 27 H da RAM interna.

O 8051 possui Registros de Função Especial (SFRs - Special Function Registers) que são responsáveis pela maior parte do controle do microcontrolador. Os mesmos são mostrados na Figura 4, sendo que alguns deles possuem bits endereçáveis. Note que alguns dos bits endereçáveis possuem inclusive um nome mnemônico, para maior facilidade de desenvolvimento de software em compiladores.

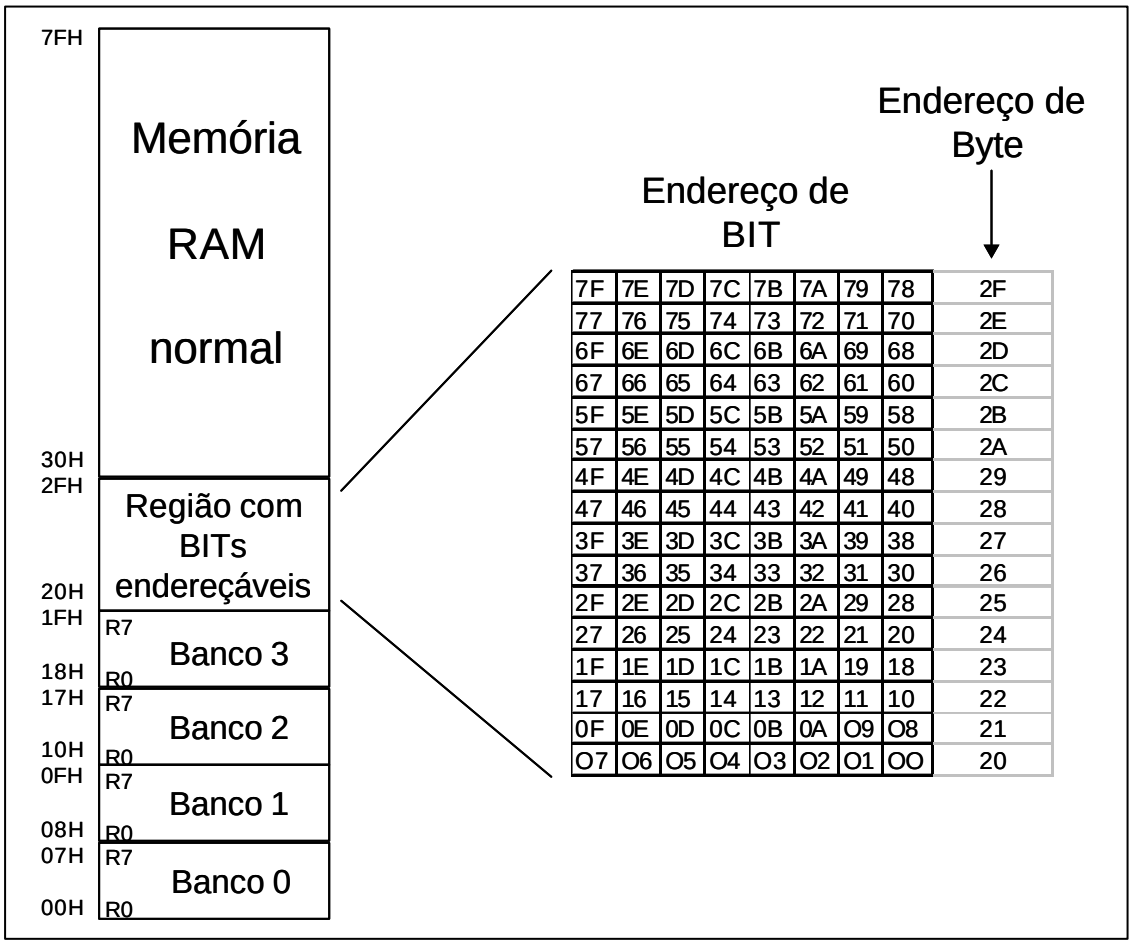

Figura 3: Detalhes a divisão da RAM interna do 8051.

O registro PSW é a palavra de status de programa, onde estão vários bits de status que refletem o estado corrente da CPU. O registro PSW é acessado pelo endereço D0h e também é bit endereçável.

A Tabela 1 apresenta a estrutura do registro PSW, que reside no espaço de endereçamento de registros de função especial.

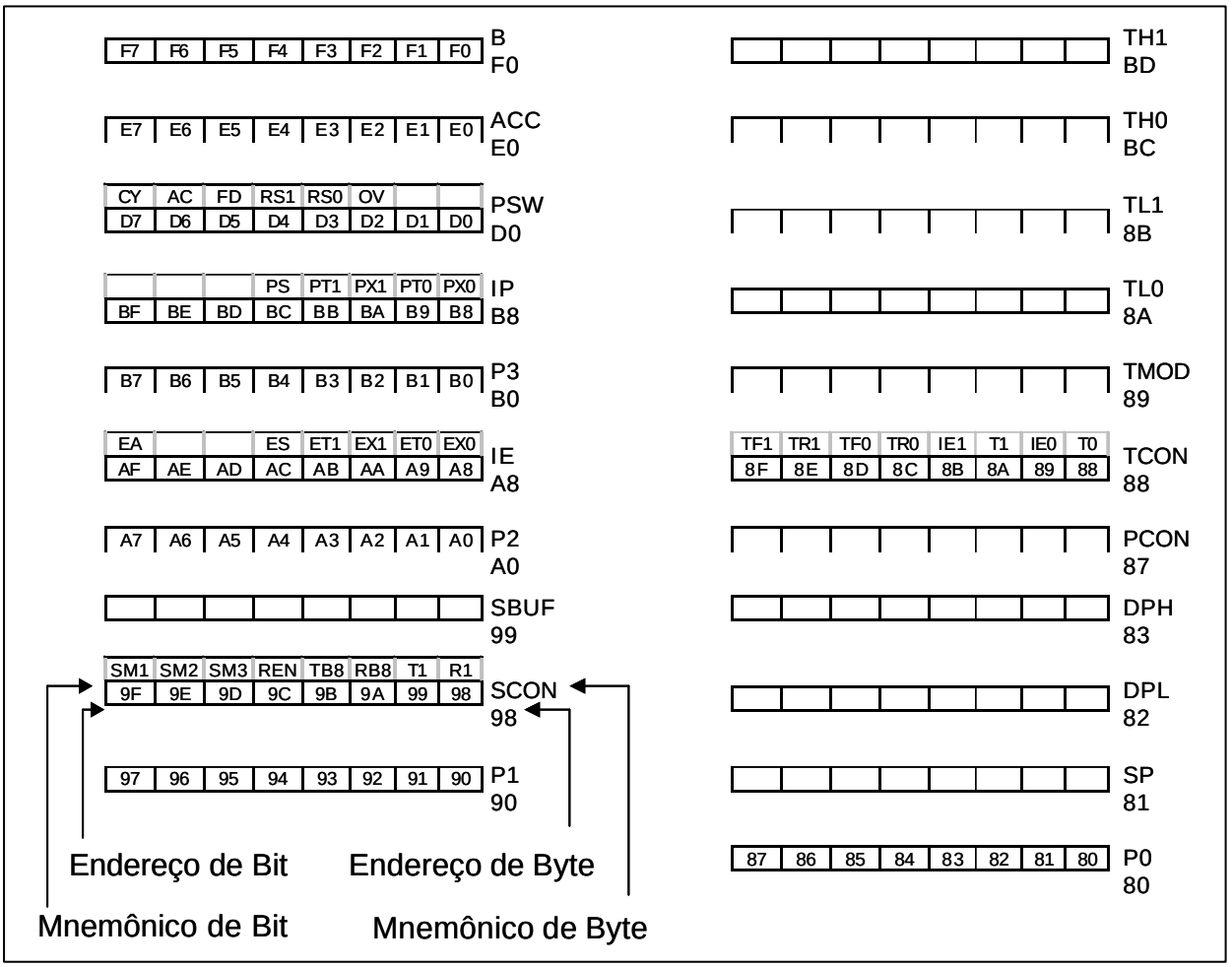

Figura 4: Registros de Função Especial.

Tabela 1: Estrutura do Registro PSW

D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H
CY	AC	F0	RS1	RS0	OV	-	P
PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0

PSW.7 - Flag de carry para bit 7 da ALU;

PSW.6 - Flag de carry auxiliar para bit 3 da ALU (para operações BCD);

PSW.5 - Flag 0 - Flag de status de propósito geral (definido pelo usuário);

PSW.4 - Bit 1 de seleção de banco;

PSW.3 - Bit 0 de seleção de banco;

PSW.2 - Flag de overflow para operações aritméticas;

PSW.1 -

O PSW.0 é o flag de paridade do acumulador (1=ímpar, 0=par). Este registro contém o bit de carry, carry auxiliar, para operações BCD, dois bits de seleção de banco de registros, flag de overflow, bit de paridade e dois flags de status definidos pelo usuário.

O bit de carry é usado em operações aritméticas e, também, serve como "acumulador" para operações booleanas.

O registro apontador de dados é formado por um byte alto (DPH) acessado pelo endereço 83 H e por um byte baixo (DPL), acessado pelo endereço 82 H. Este registro pode ser manipulado como um registro de 16 bits (DPTR), ou como dois registros independentes de 8 bits. A sua função é de manter um endereço de 16 bits, usado para acessar memórias externas.

Como o conjunto de instruções da família 8051 foi otimizado para aplicação de controle em 8 bits. Contém uma variedade de modos de endereçamento para acessar a RAM interna, facilitando operações de byte em pequenas estruturas de dados.

No modo de endereçamento direto, o operando é especificado na instrução por um campo de endereço de 8 bits. Somente a RAM de dados interna e os registros de função especial (primeiras 256 posições de memória) é que poderão ser endereçados diretamente, como mostra a Figura 5.

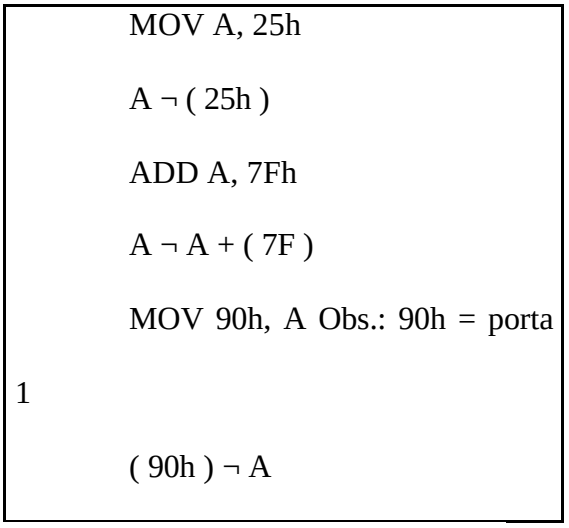

Figura 5: Endereçamento direto.

No modo de endereçamento de registro ou modo registrador, os bancos de registros, contendo de R0 a R7, serão acessados por certas instruções onde a especificação do registro será feita por três bits do próprio opcode. As instruções de acesso aos registros são eficientes, visto que nenhum endereço será necessário, como está demonstrado na Figura 6.

Quando a instrução for executada, um dos oito registros do banco selecionado será acessado. Um dos quatro bancos de registro será selecionado pelos bits de seleção de bancos do registro PSW (bits RS1 e RS0).

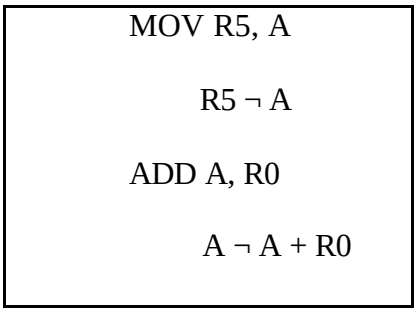

Figura 6: Endereçamento de registro.

Já no modo de endereçamento indireto a instrução especifica que o registro contém o endereço do operando. Tanto a memória interna, quanto a externa, poderão ser endereçadas indiretamente. Na Figura 7 fica ilustrado o endereçamento indireto do registrador R1. O registro de endereçamento usado para endereços de 8 bits deverá ser o registro R0 ou R1 do banco selecionado e para endereços de 16 bits será somente o registro apontador de dados (DPTR).

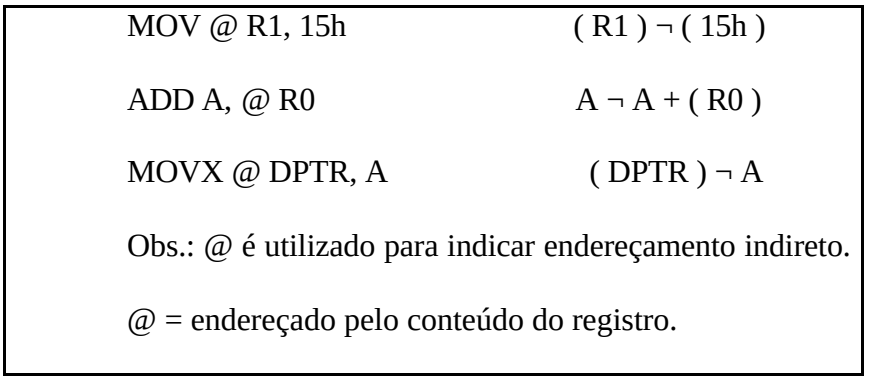

Figura 7: Endereçamento indireto.

No modo de endereçamento de registros específicos ou específicos a registro, algumas instruções referem-se a certos registros. Por exemplo, algumas instruções operam o acumulador, o registro DPTR, etc., assim nenhum byte de endereço será necessário, o opcode (instrução) já define qual o registro que será afetado. A Figura 8 mostra exemplos de endereçamento de registros específicos.

DA A	Faz o ajuste decimal do acumulador
CLR A	A ¬ 00h (zera o acumulador)
INC DPTR	DPTR ¬ DPTR + 1

Figura 8: Endereçamento de registros específicos.

Para o modo de endereçamento imediato ou constante imediata, o opcode (instrução) é seguido de um valor de uma constante que será operada. Na linguagem assembly, este modo é indicado através do símbolo #. Exemplos de endereçamento de constante está demonstrado na Figura 9.

```
MOV B, #252
        B ¬ FCh (252 em Hexa)
MOV A, #100
        A ¬ 64h (100 em Hexa)
MOV DPTR, #05FEh
        DPTR ¬ 05FEh
Obs.: # indica valor constante;
```

Figura 9: Endereçamento de Constante Imediata.

Quando após a constante aparecer um "h", o valor da constante é hexadecimal, quando tiver um "B" é binário, e quando a letra for omitida ou aparecer um "D" o valor será decimal.

Quando utilizamos o modo de endereçamento indexado, somente a memória de programa (ROM) poderá ser acessada. O endereço efetivo é a soma do acumulador e um registro de 16 bits (DPTR ou PC).

Este modo é usado para leituras de tabelas colocadas na memória de programa (ROM). Por exemplo, tabelas de conversão, ou de mensagens.

Um registro base de 16 bits, tal como o registro DPTR, ou contador de programa (PC), veja Figura 10, aponta para a base da tabela e o acumulador recebe o deslocamento dentro da tabela. Assim o endereço de entrada da tabela será formado com a soma do conteúdo do acumulador e o registro base.

MOVC A, @ A + DPTR	A ¬ (A+DPTR) da ROM
MOVC A, @ A + PC	A ¬ (A + PC) da ROM

Figura 10: Endereçamento Indexado na ROM.

Outro tipo de endereçamento indexado é usado nas instruções de "case jump". Neste caso o endereço destino do salto (jump) é calculado com a soma do conteúdo do acumulador e do conteúdo do apontador base, como mostra a Figura 11.

Assim o valor base do endereço do salto será carregado no apontador base (DPTR), e o valor de indexação do salto que realiza a condição (case) será carregado no acumulador.

JMP @ A + DPTR	PC ¬ A + DPTR
Faz um salto para o endereço dado por A + DPTR	

Figura 11: Endereçamento Indexado (case jump).

O 8051 pode ser interrompido de 5 maneiras:

- pela interrupção externa 0 (INT0\ - pino P3.2);

- pela interrupção externa 1 (INT1\ - pino P3.3);

- pelo contador/temporizador 0;

- pelo contador/temporizador 1;

- pelo canal de comunicação serial.

As interrupções do 8051 ocorrem através de vetores, ou seja, têm um endereço de início da rotina de tratamento da interrupção fixo.

O 8051 possui internamente dois Contadores/Temporizadores denominados como TEMPORIZADOR 0 E TEMPORIZADOR 1. Ambos podem ser configurados para operar como temporizador ou contador de eventos, individualmente. Podem ter a operação habilitada por software ou hardware.

Na função de temporizador, um registro será incrementado a cada ciclo de máquina. Considerando que cada ciclo de máquina consiste em 12 períodos do clock, a taxa de contagem será de 1/12 da freqüência do clock.

Na função de contador, um registro será incrementado em resposta a uma transição de "1" para "0" (borda de descida) de seu correspondente pino de entrada externa, T0 e T1.

Nesta função, os pinos externos (T0 e T1) são amostrados a cada ciclo de máquina. Quando uma amostragem indicar um nível alto em um ciclo de máquina e um nível baixo no próximo ciclo, o contador será incrementado.

A máxima taxa de contagem será de 1/24 da freqüência do clock, visto que são necessários dois ciclos de máquina para o reconhecimento de uma transição de "1" para "0".

A operação dos Contadores/Temporizadores terá quatro modos possíveis.

A seleção de Temporizador ou Contador é realizada através do registro de função especial TMOD (Modo do Temporizador).

O registro TMOD é dividido em duas partes iguais que controlam o TEMPORIZADOR 1 e TEMPORIZADOR 0. O registro TMOD é acessado pelo endereço 89h e não é bit endereçável.

O TMOD é o registro de controle de modo Temporizador/Contador, é neste registro que é feita a seleção de função Temporizador ou Contador e a seleção do modo de operação (modo 0, 1, 2 ou 3).

Finalizando, o microcontrolador 8051 segue algumas Instruções do 8051.

***PUSH* Direto** – Coloca na pilha o conteúdo da posição de memória. Incrementa o SP (Stack Pointer) e escreve na pilha. (2 bytes – 24 pulsos);

***POP* Direto** – Retira da pilha o Dado e coloca na posição de memória. (2 bytes – 24 pulsos);

CLR C – Zera o Carry. (1 byte – 12 pulsos) (CY=0);

CLR Bit – Zera o bit Endereçado. (2 bytes – 12 pulsos);

SETB C – Seta o Carry. (1 byte – 12 pulsos) (CY=1);

SETB Bit – Seta o bit endereçado. (2 bytes – 12 pulsos);

MOV C,Bit – Move o bit endereçado para o Carry. (2 bytes – 12 pulsos) (CY=?)

MOV Bit,C – Move o Carry para o bit endereçado. (2 bytes – 24 pulsos);

JC rel – Salta se o Carry for "1". O jump é relativo. (2 bytes – 24 pulsos);

JNC rel – Salta se o Carry for "0". O jump é relativo. (2 bytes – 24 pulsos);

JB Bit, rel – Salta se o bit endereçado estiver em "1". (3 bytes – 24 pulsos);

JNB Bit, rel – Salta se o bit endereçado estiver em "0". (3 bytes – 24 pulsos);

JBC Bit, rel – Salta se o bit endereçado estiver em "1" depois zera o bit. (3 bytes – 24 pulsos);

ADD A, Rn – Soma o conteúdo do Registro *n* ao Acumulador. (1 byte – 12 pulsos) (CY=?, AC=?, OV=?);

ACALL End 11 – Chama sub-rotina numa faixa de 2 Kbytes da atual posição. (2 bytes – 24 pulsos);

LCALL End 16 – Chama sub-rotina em qualquer posição da memória de programa (ROM). (3 bytes – 24 pulsos);

RET – Retorno de sub-rotina. (1 byte – 24 pulsos);

2.2. Microcontrolador PIC

Este componente de 18 pinos possui interrupções e não dispõe de periféricos como PWM, conversores A/D ou portas de comunicação serial (MICROCHIP, 2004).

O microntrolador PIC possui as características descritas na Tabela 2.

Tabela 2: Características do PIC 16F84

• Memória de Programa Flash (1024 words de 14bits)
• Memória de dados 68 bytes
• Memória EEPROM 64 bytes
• 13 Entradas / Saídas
• PORT A: RA0... RA4 (5 PINOS)
• PORT B: RB0... RB7 (8 PINOS)
• Cap. de corrente: 25mA (por pino)
• 4 tipos diferentes de Interrupção

A memória de programação é onde as instruções do programa são armazenadas. No caso do 16F84 esta memória é de 1024 palavras (words) de 14 bits cada uma. Alguns destes 14 bits informam o opcode (código da instrução) e o restante traz consigo o argumento da instrução correspondente. Na família PIC existem três tipos de memória de programa: EPROM, EEPROM e FLASH. Existem duas posições da memória do programa que recebem nomes especiais: vetor de reset e vetor de interrupção.

O vetor de reset é para onde o programa vai quando ele é inicializado, enquanto que o vetor de interrupção é a posição da memória de programa para onde o processamento é desviado quando ocorre uma interrupção.

A memória de dados é uma memória volátil do tipo R.A.M. (random access memory). O mapa de memória é divido em duas partes: registradores especiais (special function register - S.F.R.) e registradores de uso geral (general purpose register - G.P.R.). Como o ponteiro da memória de programa tem capacidade de endereçar somente 128 posições de memória de cada vez (7 bits), a memória de programa é dividida em bancos (banco 0 e banco 1 no 16F84).

Esta divisão implica, em termos, posições de memória que somente poderão ser acessadas caso o banco ao qual pertença seja previamente selecionado, através de um bit específico do S.F.R. STATUS.

A unidade lógica aritmética é onde todas as operações lógicas (funções lógicas booleanas: e/ou, exclusivo e complemento) e aritméticas (soma e subtração) são efetuadas. O registrador W sempre estará envolvido de alguma forma em toda operação lógica ou aritmética. Existem dois destinos possíveis para estas operações: o W (work) ou um registrador (posição da memória de dados) definido no argumento da instrução.

Como já visto, anteriormente, um microcontrolador pode ser entendido como sendo uma máquina que executa operações em ciclos. Todos os sinais necessários para a busca ou

execução de uma determinada instrução devem ser gerados em um período de tempo denominado Ciclo de Máquina. Nos PIC com memória de programa de 12 e 14 bits um Ciclo de Máquina corresponde a quatro períodos de clock (1:4) denominados Q1, Q2, Q3 e Q4, conforme pode ser verificado na Figura 12.

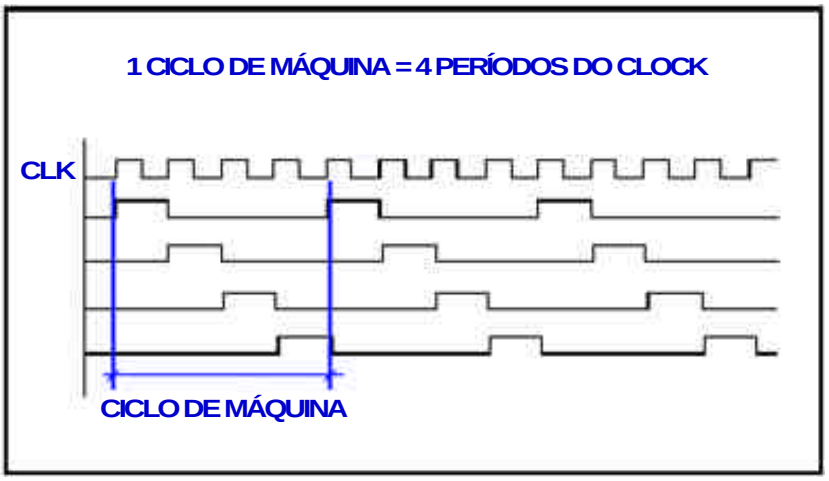

Figura 12: Ciclo de Máquina no PIC.

Se para efeito de análise dividirmos o processamento interno do PIC em ciclos de busca e execução, podemos afirmar que para cada instrução executada foi necessária a execução prévia de um ciclo de busca. Imagine um sistema que implemente um ciclo de busca e ao mesmo tempo processe um ciclo de execução. Desta forma, no início de cada Ciclo de Máquina haverá uma instrução pronta para ser executada, funcionando como pipeline.

No entanto, algumas instruções fazem com que este sistema seja desarticulado: são as chamadas instruções de desvio. As instruções de desvio são aquelas que alteram o valor do Program Counter (contador/ponteiro de programa).

Quando ocorre um desvio, a instrução que já foi previamente buscada pelo sistema de Pipeline não é válida, pois estava na posição de memória de programa apontada pelo PC antes de ele ter seu valor alterado para o destino especificado. Conseqüentemente torna-se necessária a execução de um novo ciclo de busca, que obviamente demandará mais um ciclo de máquina, resultando em um tempo de total de processamento igual a dois ciclos de

máquinas.

Toda instrução do PIC demanda um Ciclo de Máquina para ser executada, exceto aqueles que provocam desvio no programa os quais demandam dois Ciclos de Máquina.

O PIC possui internamente um recurso de hardware denominado Timer0. Trata-se de um contador de 8bits incrementado internamente pelo ciclo de máquina, ou por um sinal externo (borda de subida ou descida), sendo esta opção feita por software durante a programação (SFR). Como o contador possui 8 bits ele pode assumir 256 valores distintos (0 até 255). Caso o ciclo de máquina seja de 1us, cada incremento do Timer corresponderá a um intervalo de 1us. Caso sejam necessários intervalos de tempos maiores para o mesmo Ciclo de Máquina, utilizamos o recurso de PRESCALE.

O PréScale é um divisor de freqüência programável do sinal que incrementa o Timer0. Quando temos um PréScale de 1:1, cada ciclo de máquina corresponde a um incremento do Timer0 (unidade de Timer0). Ao alterarmos o préscale para, por exemplo, 1:4 (os valores possíveis são as potências de dois até 256), o Timer0 será incrementado uma vez a cada quatro ciclos de máquina.

O watchdog é um recurso disponível no PIC o qual parte do princípio de que todo sistema é passível de falha. Se todo sistema pode falhar, cabe ao mesmo ter recursos para que, na ocorrência de uma falha, algo seja feito de modo a tornar o sistema novamente operacional.

Dentro do PIC existe um contador incrementado por um sinal de relógio (clock) independente. Toda vez que este contador extrapola o seu valor máximo retornando a zero, é provocada a reinicialização do sistema (reset).

Se o sistema estiver funcionando da maneira correta, de tempos em tempos uma instrução denominada **clear watchdog timer** (CLRWDT) zera o valor deste contador, impedindo que o mesmo chegue ao valor máximo. Desta maneira, o watchdog somente irá "estourar" quando algo de errado ocorrer.

O período normal de estouro do Watchdog Timer é de aproximadamente 18 ms. No entanto, algumas vezes este tempo é insuficiente para que o programa seja normalmente executado. A saída, neste caso, é alocar o recurso de **PréScale** de modo a aumentar este período.

Se sem o Préscale o período é de 18ms, quando se atribui ao Watchdog Timer um PRÉ SCALE de 1:2 (um para dois) dobra-se este período de modo que o processamento possa ser executado sem que seja feita uma reinicialização.

As Interrupções são causadas através de eventos assíncronos (podem ocorrer a qualquer momento) que causam um desvio no processamento. Este desvio tem como destino o vetor de interrupção.

O PIC 16F84 possui quatro interrupções, conforme descrito na Tabela 3.

Tabela 3: Interrupções do PIC 16F84

• Interrupção externa (RB0)
• Interrupção por mudança de estado (RB4..RB7)
• Interrupção por tempo (TMR0)
• Interrupção de final de escrita na EEPROM

A habilitação das interrupções nos PIC segue a seguinte filosofia: existe uma chave geral (general interrupt enable) e chaves específicas para cada uma das interrupções. Deste modo, para habilitar a interrupção de tempo (TMR0) deve-se “setar” o bit da chave geral e também o bit da chave específica (T0IE), ambos presentes no registrador especial (S.F.R.) INTCON.

O estado **POWER ON RESET** é um sistema que faz com que durante a energização o pino de Master Clear (/MCLR) permaneça durante algum tempo em zero, garantindo a inicialização.

O **POWER UP TIMER** é o temporizador que faz com que o PIC, durante a energização (power up), aguarde alguns ciclos de máquina para garantir que todo o sistema

periférico (display, teclado, memórias, etc.) estejam operantes quando o processamento estiver sendo executado.

O **BROWN OUT** monitora a diferença de tensão entre VDD e VSS, provocando a reinicialização do PIC (reset) quando esta cai para um valor inferior ao mínimo definido em manual.

O **SLEEP** é o modo de operação Sleep e foi incluído na família PIC para atender um mercado, cada vez maior, de produtos que devem funcionar com pilhas ou baterias. Estes equipamentos devem ter um consumo mínimo para que a autonomia seja a máxima.

Quando o PIC é colocado em modo Sleep (dormir), através da instrução SLEEP, o consumo passa da ordem de grandeza de mA (mili àmperes) para uA (micro àmperes). Existem três maneiras de "acordar o PIC": por interrupção externa/estado, estouro de Watchdog ou reinicialização (/MCRL).

A linguagem de programação Assembler do PIC 16F84 é composta por 35 instruções. As instruções são expressas na forma de mnemônicos. O mnemônico é composto por termos e operações. As operações básicas do PIC estão descritas na Tabela 4.

As instruções são divididas em quatro grupos:

• instruções orientadas a byte (registradores);

• instruções orientadas a bit;

• instruções com constantes (literais);

• instruções de controle.

Tabela 4: Operações Básicas de Programação do PIC

ADD	Somar	MOV	Mover
AND	Lógica “E”	RL	Rodar para esquerda
CLR	Limpar	RR	Rodar para direita
COM	Complementar	SUB	Subtrair
DEC	Decrementar	SWAP	Comutar
INC	Incrementar	XOR	Lógica “OU EXCUSIVO”
IOR	Lógica “OU”		

3. PROCESSADORES SUPERESCALARES

3.1. Introdução

Os processadores possuem uma arquitetura formada por vários componentes que atuam juntamente para a realização do processamento das informações nos computadores. Para se iniciar o entendimento sobre o funcionamento dos processadores, as técnicas de pipeline e os processadores superescalares, é necessário considerar que uma CPU executa as ações descritas na Figura 13 (STALLINGS,2002).

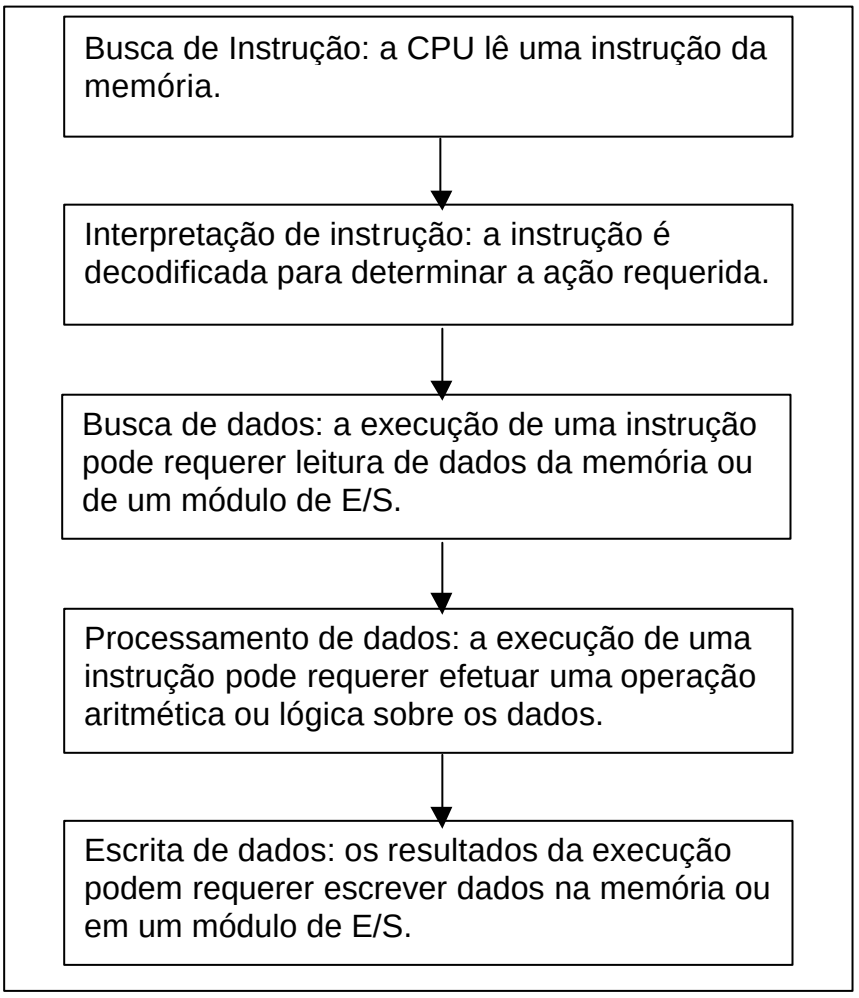

Figura 13: Ações executadas na CPU.

Para executar estas ações a CPU utiliza um grande número de componentes nos processadores, como registradores, contadores, unidades de controle (UC), unidades lógico-aritméticas (ULA), entre outros.

Todos estes componentes fazem parte da estrutura interna da CPU, como está demonstrado na Figura 14 (STALLINGS,2002).

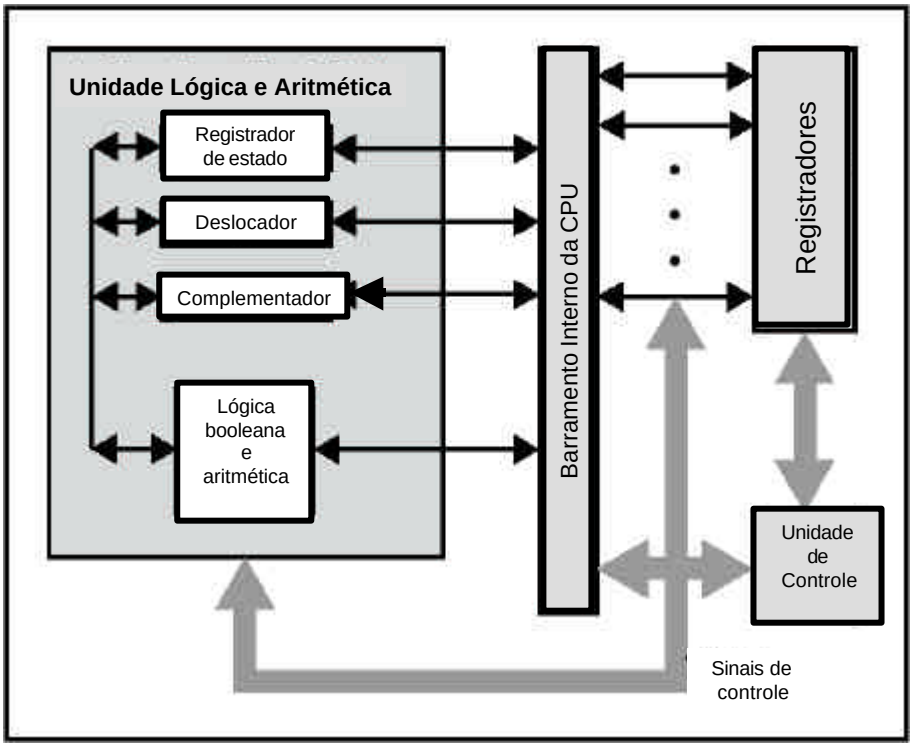

Figura 14: Estrutura Interna da CPU.

Basicamente, a Unidade de Controle (UC) executa as instruções, que se encontram nos registradores específicos, aplicando os valores, armazenados em outros registradores, na Unidade Lógica e Aritmética (ULA) através dos barramentos internos da CPU e dos sinais de controle.

Inicialmente, o fim do último estágio determina o início do primeiro estágio novamente. Toda esta relação determina o que é conhecido como ciclo de instrução.

3.2. Técnicas de Pipeline

As duas técnicas mais importantes, usadas e estudadas para aumentar a performance dos processadores na última década foram os pipelines e as memórias cache (JOUPPI, 1991).

Um pipeline de instruções é semelhante a uma linha de montagem de uma indústria (STALLINGS, 2002). O pipeline de instruções é a colocação de cada um dos estágios em uma linha de produção, o qual determina, com o fim de um estágio que iniciará outro, que este terminou e está apto para começar com o resultado do estágio anterior. O processamento de um pipeline de instruções é o processamento de cada estágio em cascata com a conexão linear entre o fim de um estágio e o início de outro (HWANG, 1993).

HENNESSY (1994) ilustra o pipeline usando o exemplo de uma lavanderia, que executa 4 fases na lavagem das roupas. A Figura 15 mostra que, se a lavanderia esperar acabar as quatro fases para iniciar o processo novamente, o tempo será muito maior, em relação ao uso das técnicas de pipeline onde o fim de uma fase já habilita esta fase a iniciar o processo novamente.

No início da década de 80, as técnicas de pipeline eram utilizadas exclusivamente em mainframes e supercomputadores. No fim da década o pipeline passou a ser usado na maioria das novas máquinas (JOUPPI, 1991).

A primeira questão para se iniciar o estudo das técnicas de pipeline é a determinação do número de estágios em que o ciclo de execução do processador será dividido.

A princípio, pode parecer que, quanto maior o número de estágios, mais eficiente será o pipeline. Já na determinação do número de estágios, começam a aparecer os primeiros problemas. Cada estágio possui um tempo de execução, isto é, existirá uma espera entre o fim de um estágio mais rápido e o início de um estágio mais lento.

Considere um ciclo de instruções dividido nos seguintes estágios (STALLINGS,2002):

? Busca de instrução (BI): lê a próxima instrução esperada e a armazena em uma área de armazenamento temporário.

? Decodificação da instrução (DI): determina o código de operação da instrução e as referências a operandos.

? Cálculo de operandos (CO): determina o endereço efetivo de cada operando fonte. Isso pode envolver endereçamento por deslocamento, endereçamento indireto, via registrador, endereçamento direto, assim como outras formas de calculo de endereço.

? Busca de operandos (BO): busca cada operando localizado na memória. Os operandos localizados em registradores não precisam ser buscados.

? Execução de instrução (EI): efetua a operação indicada e armazena o resultado, se houver, na localização do operando de destino especificado.

? Escrita de operando (EO): armazena o resultado na memória.

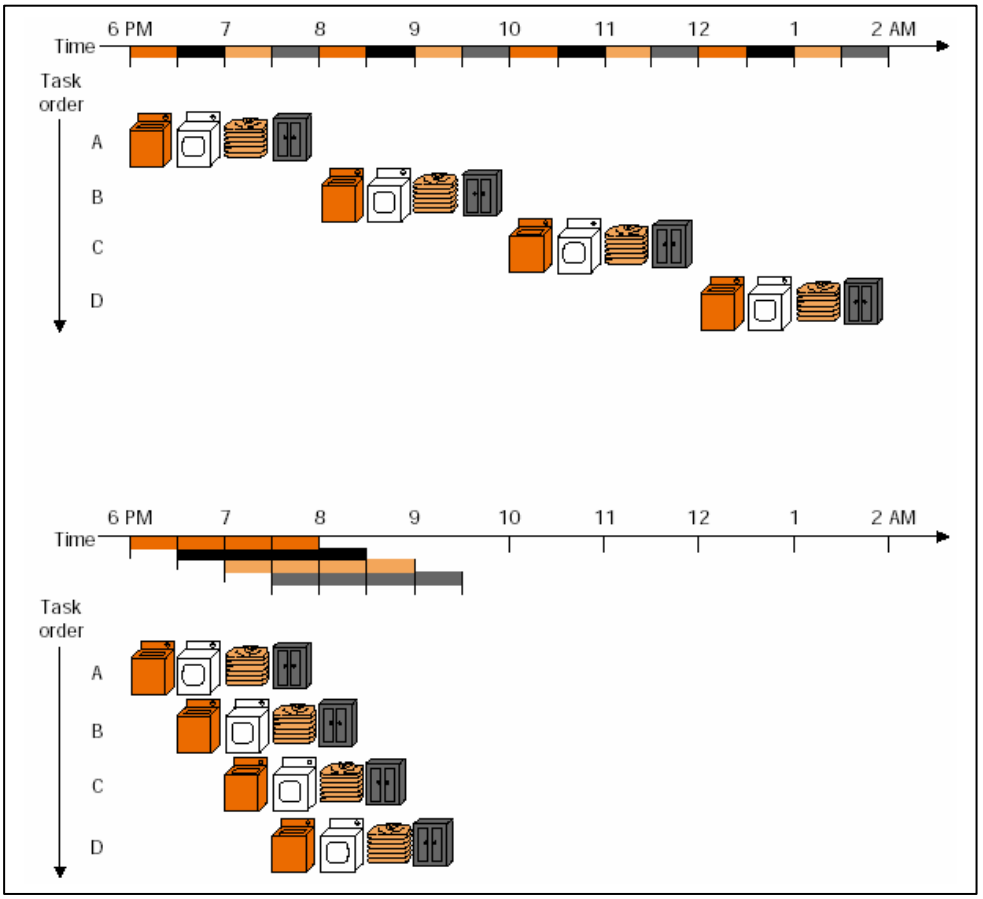

Figura 15: Ilustração das técnicas de pipeline em uma lavanderia.(HENNESSY. 1994).

Se os seis estágios não têm duração igual, existe certa espera envolvida em vários estágios do pipeline (STALLINGS, 2002).

Outro fator, que pode complicar a performance do pipeline, são os desvios condicionais. Neste caso, o desvio para uma outra linha de execução pode invalidar várias buscas realizadas.

A dependência do resultado de uma instrução para a execução da próxima é outro problema que afeta a execução do pipeline.

Na Figura 16, STALLINGS (2002) mostra a execução ótima, sem nenhum problema apresentado no pipeline de 6 estágios proposta em seu livro. Em seguida, na Figura 17, é apresentado o mesmo pipeline de 6 estágios com a ocorrência de um desvio condicional que ocorre na instrução 3 para a instrução 15, fazendo com que as instruções seguintes a 3 fiquem comprometidas na sua execução.

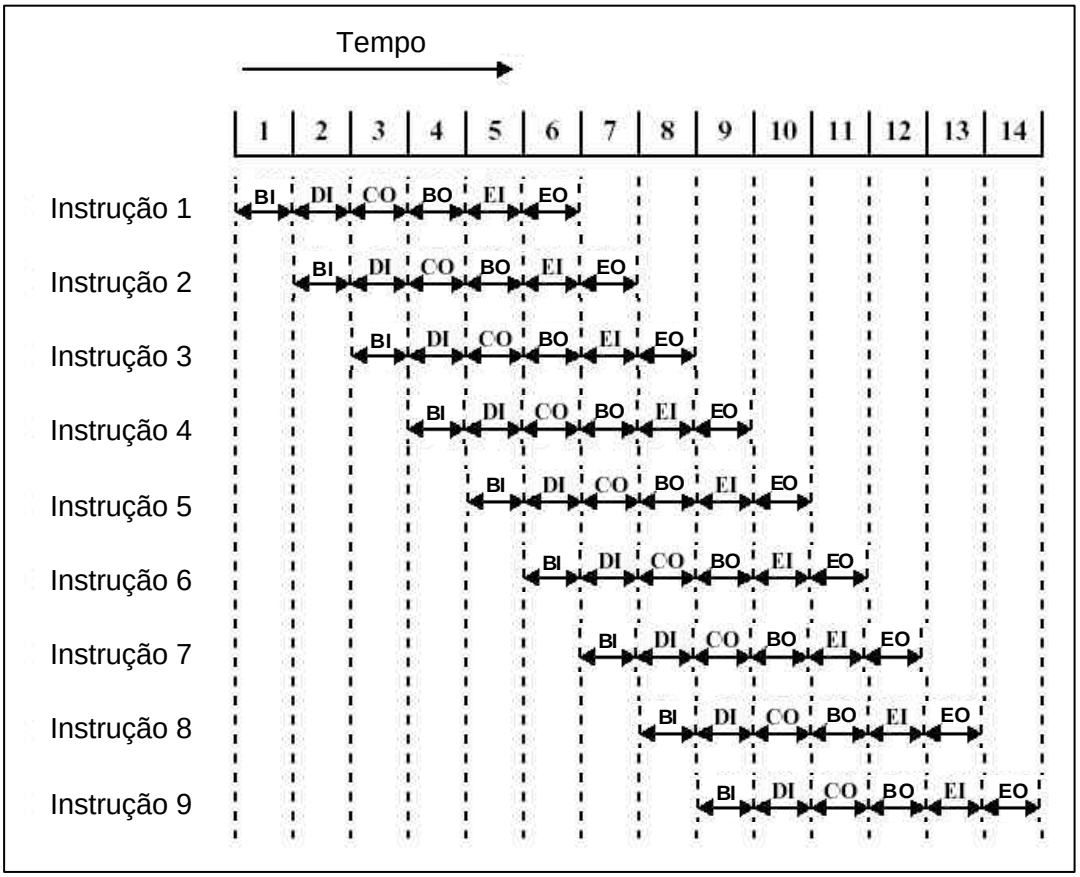

Figura 16: Diagrama de tempo para operação de pipeline de instruções. (STALLINGS,2002).

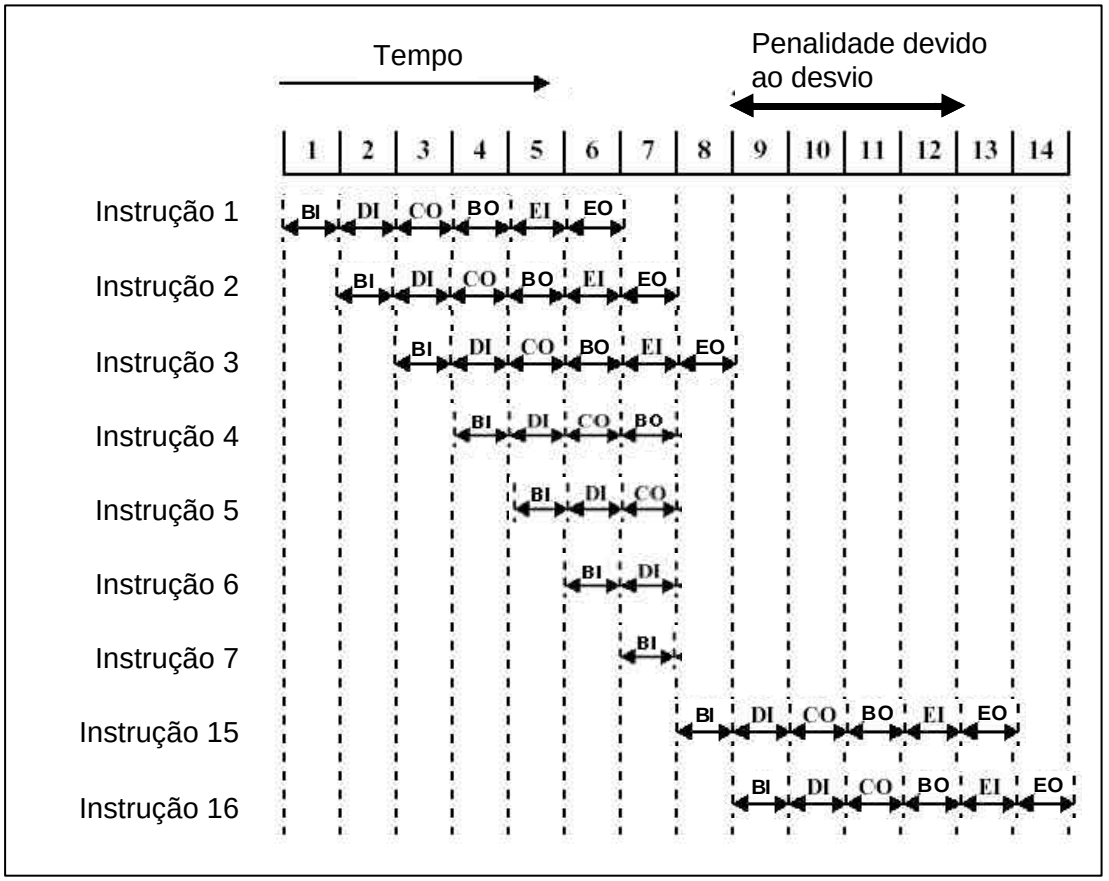

Figura 17: O efeito de um desvio condicional na operação de um pipeline de instruções (STALLINGS,2002).

O uso do pipeline de instruções é uma técnica poderosa para aumentar o desempenho, mas requer um projeto cuidadoso para que possa alcançar resultados ótimos com uma complexidade razoável (STALLINGS, 2002).

3.3. Desempenho do pipeline

Existem algumas maneiras de medir o desempenho das pipelines na execução das instruções. O speedup, a eficiência e o throughput são medidas discutidas por HWANG (1993).

Considerando um pipeline de *k* estágios, que pode processar *n* tarefas em $k + (n - 1)$ ciclos (clocks) da máquina em que está sendo executado o pipeline. Desta maneira temos que

o tempo total necessário para a execução será (HWANG, 1993):

$$T_k = [k + (n-1)]r$$

Onde r é o tempo do ciclo de execução da máquina, ou seja, o clock da máquina.

O fator de speedup baseado no número de estágios do pipeline será a relação entre o tempo inicial e o tempo total de execução das tarefas, definido por:

$$S_k = \frac{T_i}{T_k} = \frac{nkr}{Kr + (n-1)r} = \frac{nk}{k + (n-1)}$$

Na prática, HWANG (1993), afirma que o número de estágios ideal varia de *2 <= k <= 15*. A escolha do número de estágios do pipeline é o primeiro passo no projeto de arquitetura do processador, para se determinar a maior eficiência do pipeline. A Figura 18 mostra o gráfico dos speedup comparando um pipeline de 12, 9 e 6 estágios para o projeto de um processador.

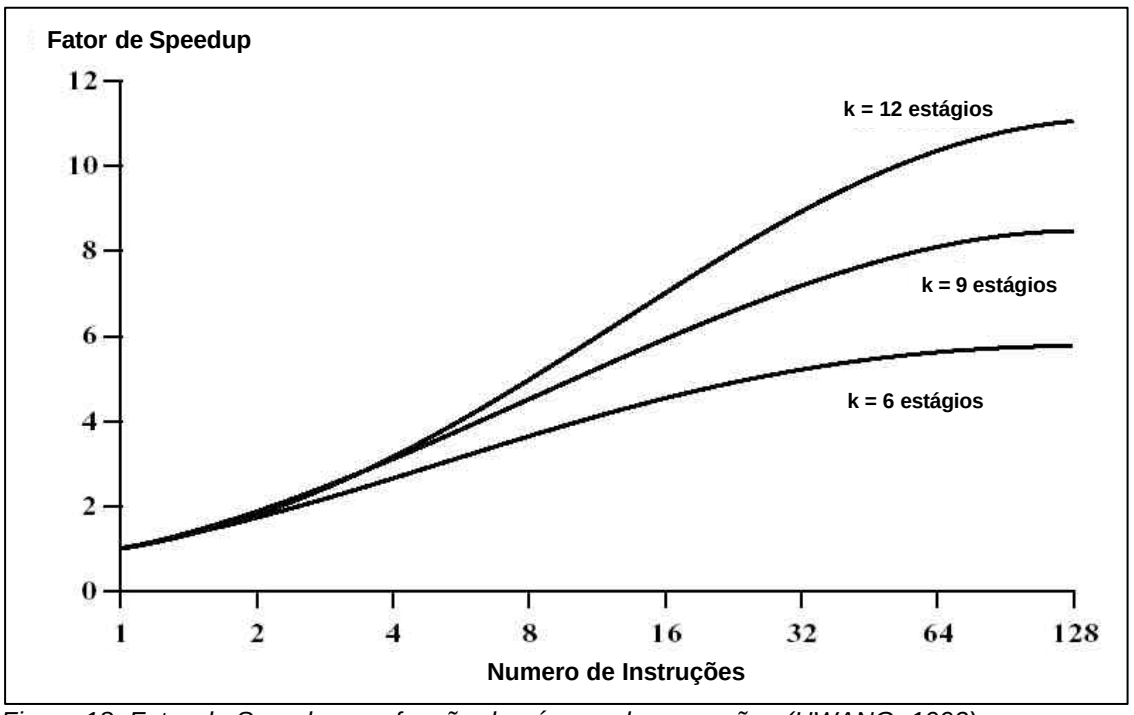

Figura 18: Fator de Speedup em função do número de operações (HWANG, 1993)

Uma outra definição para os problemas na utilização do pipeline é feita por HENNESSY (1994):

> Os perigos, ou conflitos, do pipeline são chamados de hazards, sendo, chamado de hazard todas as situações em que a próxima instrução não pode ser executada na seqüência do ciclo de instruções do clock.

Esses hazards ainda são temas de muito estudo para a evolução dos processadores, pois na prática são eles que sempre impedem que o pipeline obtenha total performance.

Existem três grandes causas para a degradação de desempenho em *pipelines* de instruções (RICARTE, 1999):

1. Conflitos na utilização de recursos (Structural Hazards);

2. Dependências de controle em procedimentos (Control Hazards);

3. Dependências de dados entre instruções (Data Hazards).

Os Conflitos de recursos (Structural Hazards), ou, hazards de estrutura são aqueles que geram conflito no hardware do processador. Exemplos típicos de conflitos incluem:

Acesso à memória: busca de instrução, busca de operandos e armazenamento de resultados estão sujeitos a conflitos na utilização da memória e do barramento do sistema;

Unidades de execução: incremento do registrador PC e execução de operações aritméticas podem ambos requerer o uso simultâneo da ULA.

A solução para este tipo de conflito usualmente envolve a duplicação de recursos, maior uso de registradores (restringindo acessos à memória principal a um único estágio, *fetch* de instrução) e a utilização de memória *cache* sob a forma de arquitetura Harvard, com separação entre *caches* e barramentos internos de dados e de instruções.

Nas **Dependências de controle (Control Hazards)**, os problemas estão associados principalmente a instruções de desvio em programas, principalmente a desvios condicionais. Quando a instrução é um desvio, não há como o primeiro estágio do pipeline conhecer o endereço da próxima instrução, uma vez que este valor só estará disponível após a execução

da operação em algum estágio posterior. Quando o endereço alvo de um desvio condicional é computado no pipeline, várias outras instruções podem estar presentes nos estágios anteriores. Se nenhum mecanismo especial for adotado, estas instruções deverão ser descartadas sempre que o endereço alvo do desvio for diferente da posição seguinte de memória.

O período durante o qual a saída do pipeline permanece ociosa devido ao descarte das operações pós-desvio é denominado uma bolha do pipeline. Como estatísticas mostram que desvios condicionais correspondem 10% a 20% das instruções de programas típicos, a degradação decorrente desse problema não é desprezível.

Uma técnica simples para se lidar com este problema é o **interlock**. Quando se detecta que a instrução é um desvio, a entrada de novas instruções no pipeline é bloqueada até a definição do endereço alvo da instrução de desvio. Obviamente, interlock é uma estratégia que impõe severa degradação no desempenho do pipeline, não sendo, portanto, uma solução para o problema.

O **Buffers de instruções (Prefetch)** é outra estratégia que envolve a utilização de buffers de instruções entre o estágio de BUSCA e os demais estágios. Para lidar com o problema de desvios condicionais, hardware especial é incorporado para detectar instruções de desvio e computar os endereços alternativos neste estágio. Como mostra a Figura 19, está incorporado um hardware especial denominado BrU (Branch Unit, ou Unidade de Branch) que tem a função de acelerar o procedimento, e que enquanto o endereço alvo do desvio não for computado por este “BrU”, os dois fluxos alternativos de instruções são transferidos para os buffers (RICARTE, 1999):

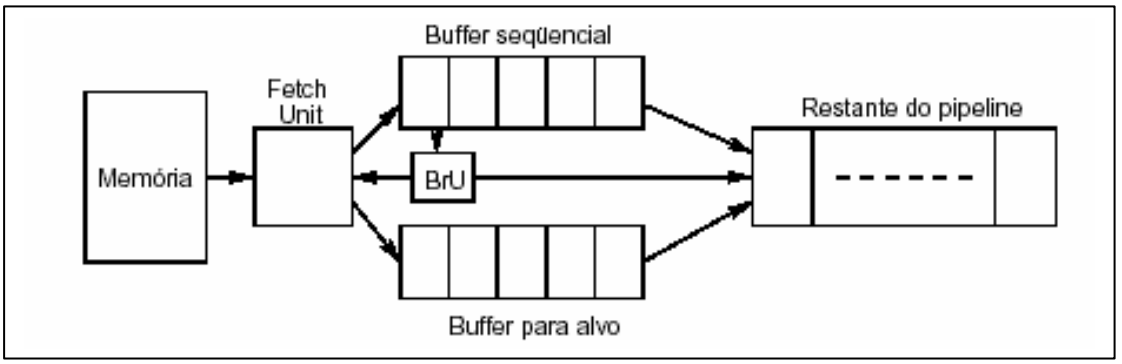

Figura 19: Buffer de Instruções (PREFETCH) (RICARTE, 1999).

Além de exigir a duplicação de *hardware*, esta estratégia não resolve o problema de desvios encadeados, ou seja, quando a instrução transferida da memória para o pipeline antes da resolução do desvio também for um desvio. Para tratar n instruções de desvio simultaneamente, 2n *buffers* seriam necessários.

No **Branch folding** um *buffer* de instruções é inserido entre o estágio de BUSCA/DECODIFICAÇÃO e o estágio de EXECUÇÃO. Neste *buffer*, para cada instrução decodificada está também o endereço da próxima instrução. Quando uma instrução de desvio é computada, o endereço alvo "cobre" o campo de endereço da instrução anterior no *buffer*. Para desvios condicionais, dois campos de endereços são mantidos. Um dos endereços é selecionado pela unidade de execução como endereço alvo tentativo, sendo utilizado para buscar as próximas instruções; o outro endereço é mantido até que o endereço alvo efetivo possa ser computado. Se a escolha foi correta, o processamento continua normalmente. Caso contrário, as instruções no *pipeline* são descartadas.

O **Branch target buffer** consiste na técnica baseada na existência de uma tabela de *lookup* de alta velocidade de acesso que mantém endereços alvos associados a instruções de desvio. Quando a instrução de desvio é encontrada pela primeira vez, uma predição inicial é feita e inserida na tabela.

Se a predição foi correta, não há penalidade ao desempenho; caso contrário, as instruções posteriores no *pipeline* são descartadas e a entrada associada ao endereço alvo daquela instrução na tabela de *lookup* é atualizada. Da próxima vez que essa instrução de desvio for encontrada, o endereço alvo da tabela é utilizado como predição. Esta estratégia é adequada para instruções de desvio associadas a iterações.

No **Delayed branch** o programador ou compilador posiciona L instruções após a instrução de desvio para serem executadas independentemente do resultado do teste. Em outros termos, buscam-se instruções anteriores à instrução de desvio (no corpo de uma iteração, por exemplo) que não afetem a condição do desvio, sendo estas instruções colocadas após a instrução de desvio. A Figura 20 apresenta um exemplo de delayed branch para um L=1.

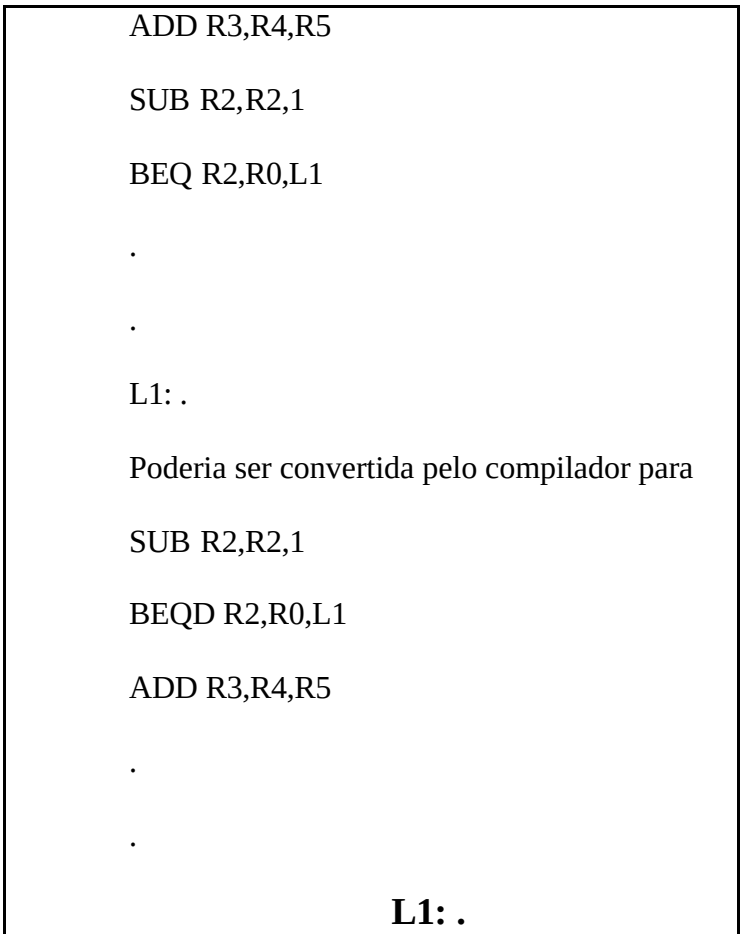

Figura 20: Exemplo de Delayed Branch.

Observando as duas rotinas da Figura 20, observamos que BEQD é uma versão *delayed* da instrução BEQ. Alguns processadores apresentam apenas a instrução de desvio com *delay*, sendo que nestes casos instruções NOP *(no operation)* são introduzidas após o desvio quando não é possível transferir outras instruções para aquela posição. Idealmente, em um *pipeline* de k estágios, L deve ser igual a k - 1 para evitar a criação de bolhas.

A **Predição estática** baseia-se no conhecimento do programador ou compilador sobre o comportamento do programa. Por exemplo, em instruções de desvio associadas a iterações é alta a probabilidade do desvio ocorrer. Por outro lado, em instruções de desvio associadas a condições de erro esta probabilidade é menor. Assim, na codificação da instrução de desvio haveria um *bit* adicional indicando a "preferência" do desvio.

As **dependências de dados (Data Hazards)** entre as instruções ocorrem quando operandos de uma instrução dependem de operandos de outra instrução. Há três tipos de dependências que podem ocorrer: a dependência de fluxo de dados, a antidependência e a dependência de saída.

A **dependência de fluxo (RAW)** (Read After Write, ou Leitura depois de uma Escrita) é a forma de dependência onde o operando de uma instrução depende diretamente do resultado de uma instrução anterior, como em:

```
ADD R3,R2,R1 ; R3 = R2 + R1
SUB R4,R3,1 ; R4 = R3 - 1
```

Neste caso, o valor de R3 só será conhecido após a conclusão da instrução ADD. Este tipo de dependência é também denominado de *hazard leitura-após-escrita* (RAW) ou de dependência real. A ocorrência deste tipo de dependência pode ocasionar a criação de bolhas no *pipeline*, a não ser que o programador ou compilador consiga inserir instruções independentes entre as duas instruções com dependência de fluxo de dados.

A **antidependência (WAR),** também conhecida como *hazard escrita-após-leitura* (Write After Read), consiste na forma de dependência que ocorre quando uma instrução posterior atualiza uma variável (registrador) que é lida pela instrução anterior, como acontece com R2 em

ADD R3, R2, R1; R3 = R2 + R1

SUB R2, R4, 1; R2 = R4 - 1

O cuidado que se deve tomar é não permitir que o conteúdo de R2 seja alterado pela instrução SUB antes de ser utilizado pela instrução ADD. O impacto deste tipo de dependência em um *pipeline* depende muito de sua estrutura, não sendo um problema na maior parte dos casos. Outra situação onde a antidependência pode ocasionar problemas é quando a ordem seqüencial de execução de instruções não é garantida.

A **dependência de saída (WAW)**, também conhecida como *hazard escrita-após-escrita* (Write After Write), ocorre quando duas instruções atualizam uma mesma variável, como em

ADD R3,R2,R1 ; R3 = R2 + R1

SUB R2,R3,1 ; R2 = R3 - 1

ADD R3,R2,R5 ; R3 = R2 + R5

Neste exemplo, além da dependência de fluxo e da antidependência entre as instruções consecutivas, há uma dependência de saída entre a primeira e a terceira instrução. Este tipo de dependência não causa problemas quando se garante que a ordem das instruções é preservada durante a execução. É também uma forma de conflito de recursos, que pode ser eliminado utilizando um outro registrador para armazenar o resultado da terceira instrução.

A técnica de ***interlock*** também pode ser utilizada para lidar com *hazards* em *pipelines*. Nesta técnica, um bit é associado a cada registrador de operandos para indicar (por exemplo, quando 1) que o conteúdo é válido. Quando uma instrução que foi buscada da

memória vai escrever em um registrador, ela verifica o valor do bit de validade. Se estiver setado, então o bit é resetado para indicar que o valor será alterado; caso contrário, a instrução deve aguardar o bit ser alterado antes de prosseguir. Esta condição de espera é necessária para evitar *hazards* de escrita após escrita. Quando o resultado da instrução é gerado o *bit* é setado, liberando o acesso do registrador para leitura por outras instruções.

A ***Data forwarding*** é uma técnica que permite acelerar o desempenho de *pipelines* oferecendo alternativas para acelerar a transferência de dados entre instruções, evitando acessos desnecessários à memória. Algumas destas técnicas podem ser aplicadas pelo compilador na fase de otimização.

O **Store-Load** ocorre quando uma instrução STORE é seguida por outra instrução LOAD referente à mesma posição de memória, a segunda instrução pode ser substituída por MOVE entre registradores. Veja na Figura 21 o exemplo a seguir:

STORE M, R1

LOAD R2, M

Ficaria:

STORE M, R1

MOVE R2, R1

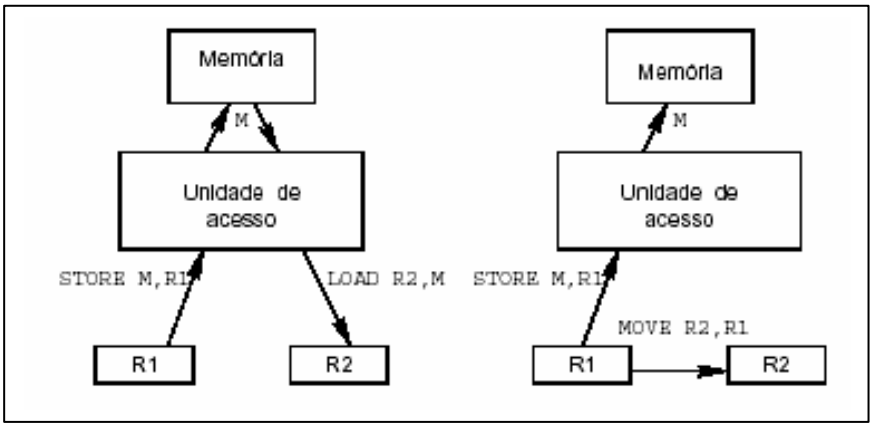

Figura 21: Store-Load (RICARTE, 1999).

Ocorre um **Load-Load** quando duas instruções LOAD referem-se a uma mesma posição de memória, a segunda instrução é substituída por uma transferência entre registradores, como mostra a Figura 22.

Por exemplo:

LOAD R1, M
LOAD R2, M

Ficaria:

LOAD R1, M
MOVE R2, R1

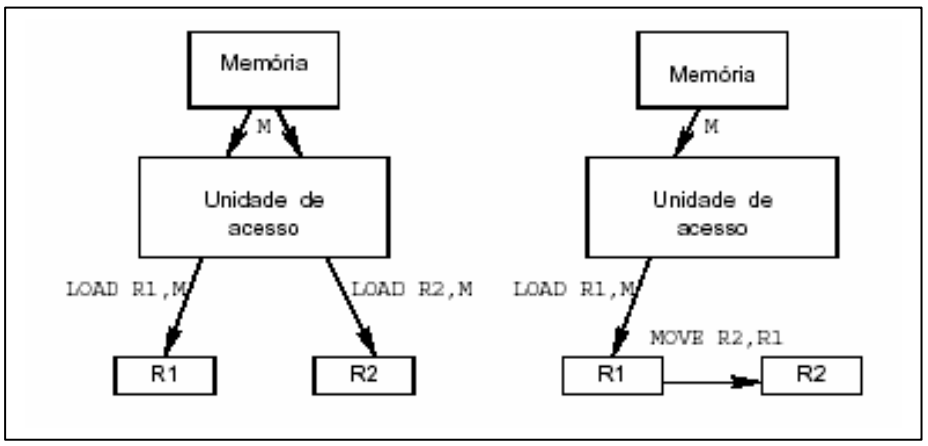

Figura 22: Load-Load (RICARTE, 1999).

Obviamente o **Store-Store** ocorre quando duas instruções STORE consecutivas armazenam valores na mesma posição de memória, a primeira operação pode ser eliminada, pois seu resultado será superposto pelo resultado da segunda instrução. Na Figura 23, está ilustrado o exemplo:

STORE M, R1

STORE M, R2

Ficaria simplesmente:

STORE M, R2

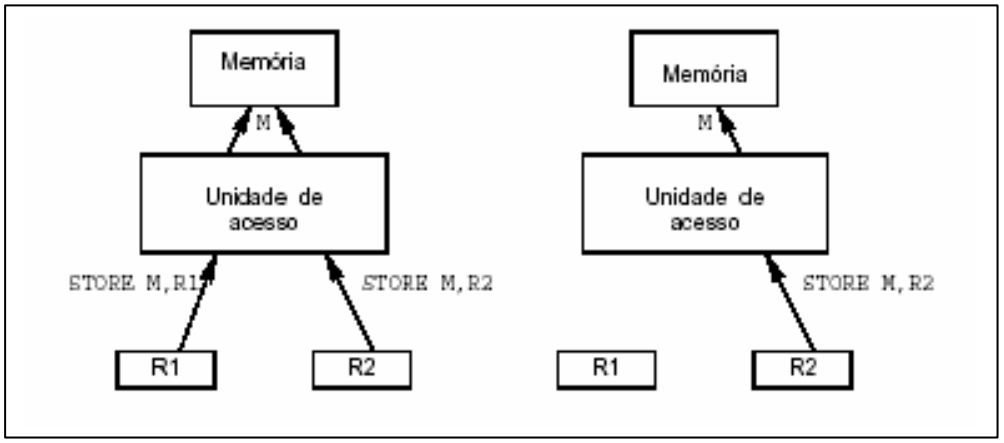

Figura 23: Store-Store (RICARTE, 1999).

O conceito de *forwarding* pode ser estendido para transferência de dados entre estágios de *pipelines*, reduzindo o impacto de *hazards* no desempenho. Neste caso, é preciso adicionar lógica e barramentos internos para detectar e explorar as transferências entre estágios do *pipeline*.

Considere a execução do seguinte trecho de código em um *pipeline* com estágios FETCHOPER, EXECUTE e STORE:

ADD R3, R2, R0

SUB R4, R3, 8

A referência a R3 na segunda instrução, sem o uso de *internal forwarding*, implica em um atraso de dois ciclos no *pipeline*, pois apenas após a instrução ADD completar STORE é que a instrução SUB poderá prosseguir com FETCHOPER. *Internal forwarding* oferece uma alternativa para essa situação onde um caminho é estabelecido entre a saída do estágio STORE para a entrada do estágio EXECUTE; a utilização deste caminho e a detecção de quando isto poderia acontecer ocorre por *hardware*. Deve-se observar que esta solução não elimina a bolha no *pipeline*, porém reduz o atraso de dois para um ciclo, neste caso.

3.4. Aplicações Práticas

Existem três tipos de pipeline. São eles os pipelines de instrução, os aritméticos e os superescalares.

Um pipeline de instrução típico possui as fases de busca, decodificação, busca de dados e *"write back"* (escrita dos resultados). Esses blocos são simples e geralmente são executados em apenas um ciclo de clock, com exceção do bloco de execução que por ser mais complexo pode demorar vários ciclos de clock para ser executado. Isso pode causar um desequilíbrio já que as fases possuem tempos de execução diferentes.

Para resolver esse problema, duas técnicas podem ser utilizadas. Uma delas consiste em subdividir a fase de execução em vários estágios e a outra consiste em alongar as fases mais curtas.

Existem também pipelines utilizados para aumentar a velocidade das operações aritméticas. Esses pipelines aritméticos são projetados para executar funções fixas.

Eles efetuam, separadamente, as operações em ponto fixo e ponto flutuante. Esse tipo de pipeline pode ter vários blocos dependendo da aplicação implementada. Todas as operações aritméticas (*add, subtract, multiply, division, squaring, rooting, logarithm*, etc.) podem ser implementadas através de um adicionador básico e de operações de deslocamento (TORRES, 1999).

Os pipelines estáticos são monofuncionais, pois executam apenas funções fixas, e os dinâmicos são multifuncionais, pois podem realizar mais de uma função. A diferença entre os dois é que o estático executa uma função de cada vez, e diferentes funções podem ser efetuadas em instantes diferentes, já o dinâmico pode realizar várias funções simultaneamente.

Os pipelines superescalares são basicamente um conjunto de pipelines funcionando em paralelo. Consiste em aumentar o número de pipelines, ao invés de um, têm-se dois ou três

pipelines em paralelo. As vantagens desse tipo de pipeline se dá pelo paralelismo real, com duas ou mais instruções sendo processadas em paralelo, com melhora significativa do desempenho, já as desvantagens consistem na necessidade do código ser preparado, aumento de complexidade e problemas de dependências e desvios.

Da mesma forma que o pipeline, a arquitetura superescalar é uma forma de paralelismo no nível das instruções, pois tem como objetivo aumentar o desempenho de um processador executando mais de uma instrução ao mesmo tempo. Esta arquitetura é composta basicamente de múltiplas unidades funcionais dentro de um único processador. De forma que, comparando a um suposto processador com apenas uma unidade funcional, esta tecnologia aumenta teoricamente n vezes a velocidade do processamento, devido a n unidades funcionais adicionadas à arquitetura para trabalhar em paralelo.

Os processadores RISC (Reduced Instruction Set Computer) são os campeões de utilização das pipelines e dos superescalares. Na arquitetura CISC (Complex Instruction Set Computer) também são implementados os pipelines e os superescalares em seus núcleos microprogramados. Nas próximas seções seguem-se alguns exemplos destes processadores.

O microprocessador MIPS R4400 é um microprocessador de 64 bits. Dispõe de uma unidade central de processamento de inteiros de 64 bits para realização de operações com tipo de dados inteiro de 64 bits, uma unidade de ponto flutuante de 64 bits para operações com algarismos com ponto flutuante de 64 bits, registros de 64 bits e um espaço de endereçamento virtual de 64 bits. A unidade de gerenciamento de memória (MMU – Memory Management Unit) do microprocessador MIPS R4400 conta para mapeamento com um TLB de 48 entradas de pares de páginas pares/impares, num total de 96 páginas, possui uma estrutura superpipeline e o espaço de endereçamento físico máximo é de 64 GB. Este microprocessador tem a capacidade de fazer leituras /escritas em rajada (burst) de 32 ou 64 bytes.

Este microprocessador conta com caches separadas para dados e instruções, de 16KB, com escrita posterior. A Figura 24 mostra um esquema dos caches separados no MIPS R4400 (SILICON GRAPHICS, 2003).

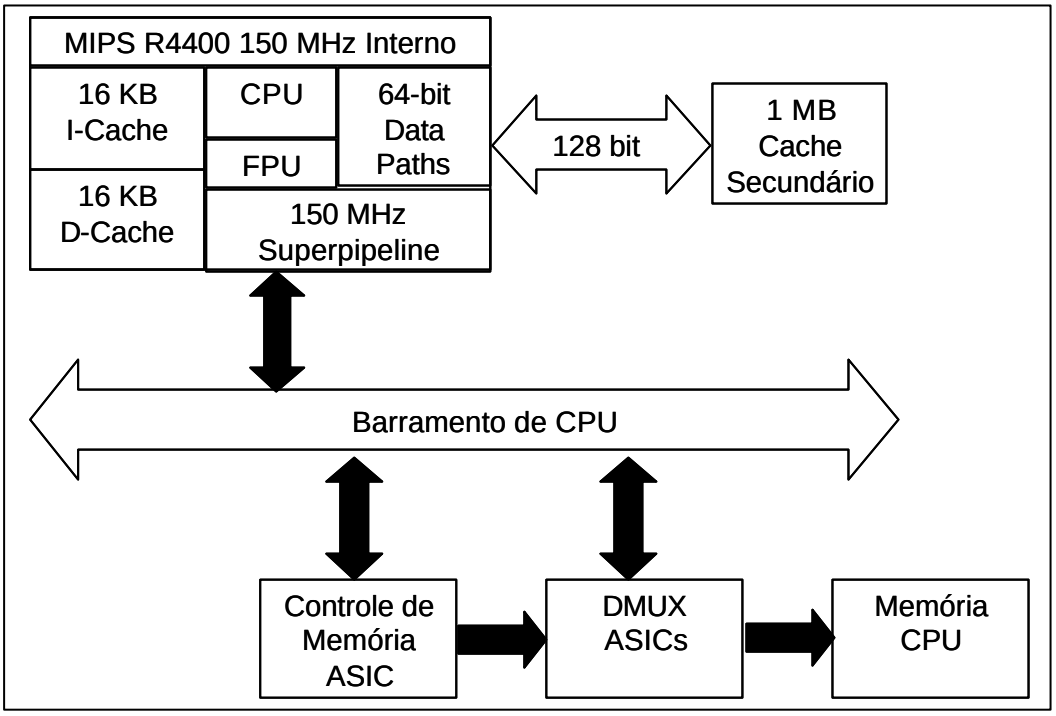

Figura 24: Esquema de Caches separados no MIPS R4400 (SILICON GRAPHICS, 2003).

No que diz respeito à performance, este microprocessador pode funcionar a freqüências internas entre os 100 e os 150 MHz (freqüências duplas das freqüências de relógio E/S).

O Processador Pentium incorpora uma arquitetura superescalar, unidade de cálculo em ponto flutuante melhorada, caches de instrução e dados separados incorporadas dentro do processador e com capacidades de escrita em write-back, barramento de dados externo de 64 bits e outras optimizações de performance relativamente a elementos anteriores da família Intel x86. Na Figura 25 está ilustrada a arquitetura do Pentium III.

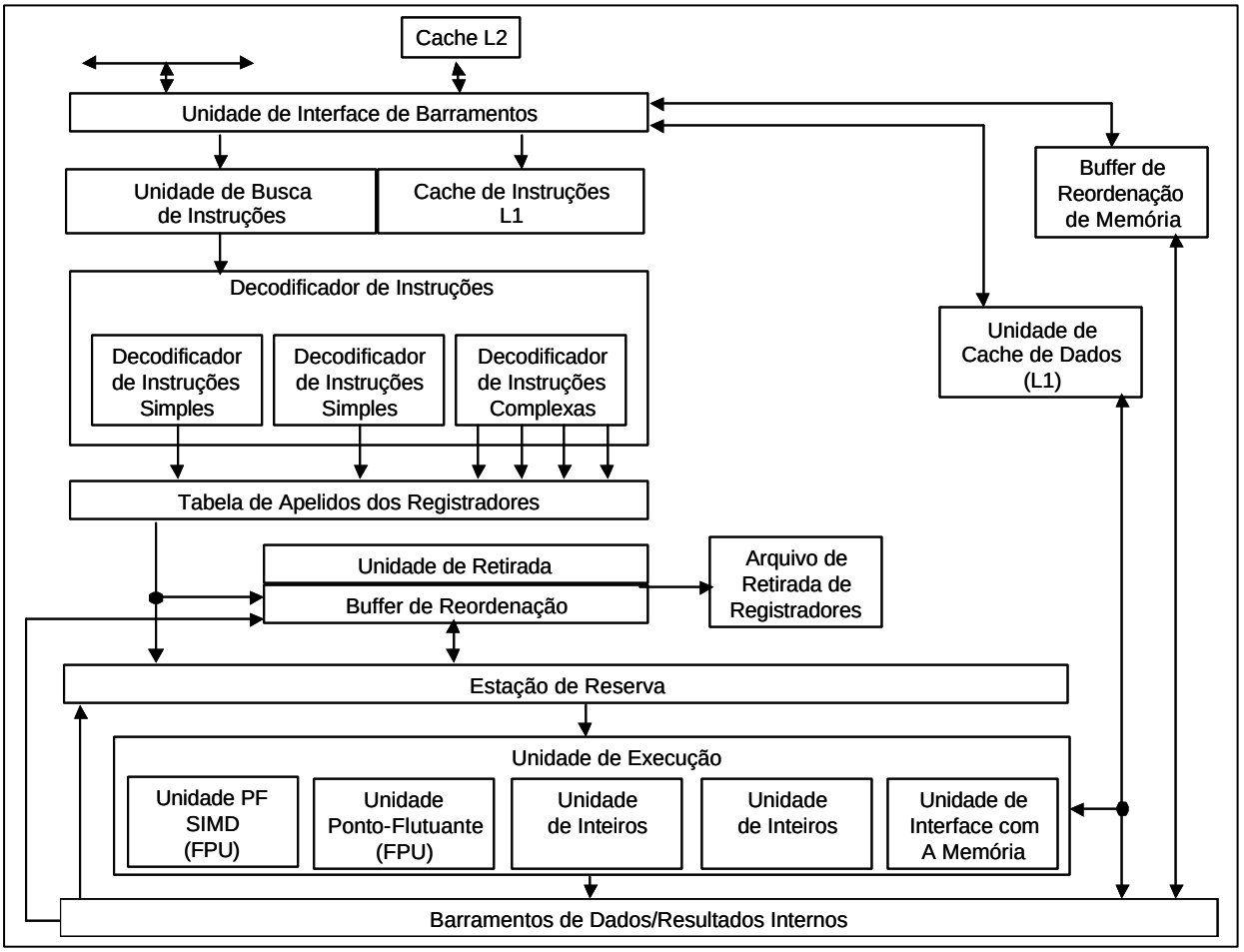

Figura 25: Arquitetura do Pentium III (IPCA, 2003).

Visando uma maior integração aliada à diminuição do tamanho de cada transistor, o processador Pentium consegue também diminuir a distância entre os componentes através do uso de várias camadas de metal (componentes CMOS) interligadas, conseguindo assim uma maior velocidade de comunicação entre os componentes. A tecnologia utilizada no Pentium utiliza três camadas. Estes melhoramentos correspondem ao que de melhor se fazia na altura em termos de tecnologia de design de microprocessadores, conseguindo atingir uma performance semelhante à de algumas implementações RISC, mantendo compatibilidade com a família x86 e a sua base instalada de utilizadores (IPCA, 2003).

O coração do processador Pentium é o seu design superescalar, construído à volta de dois pipelines de instruções, cada uma capaz de execução independente. Isto permite ao

processador executar duas instruções inteiras num único ciclo de relógio, duplicando a performance relativamente a um 80486 à mesma freqüência. Cada um destes pipelines é semelhante à pipeline do 486, mas foram optimizadas para maior performance. As pipelines estão compostas por 5 unidades: *Busca*, *Decodificação*, *Endereçamento*, *Execução* e *Armazenamento*. Assim que uma instrução passa da unidade *Busca* para a *Decodificação*, o pipeline pode iniciar o trabalho na próxima instrução.

No Pentium, cada linha da cache contém 32 bytes, simplificando a pesquisa. A cache de dados tem duas interfaces, uma para cada uma das pipelines, permitindo que dados de duas instruções diferentes possam ser lidos num único ciclo de relógio. Só quando a informação é removida da cache de dados é que a informação é atualizada em memória, técnica conhecida por write-back Esta técnica melhora a performance relativamente à estratégia write-through em que a informação é escrita na memória e na cache ao mesmo tempo.

O uso de caches separadas torna-se importante para melhorar a performance, evitando conflitos entre o *prefetch* de uma instrução e acesso a dados de outra, permitindo que ambas as operações ocorram ao mesmo tempo.

A performance também é melhorada pelo uso de uma cache pequena que permite predição dinâmica do destino de saltos condicionais. Denomina-se BTB (Branch Target Buffer) e armazena o endereço de todas as instruções de salto condicional e o endereço de destino tomado. Se esta instrução de salto for executada outra vez, é imediatamente utilizado o mesmo endereço de destino, poupando o tempo de espera para saber qual a verdadeira opção que deveria ser tomada. Se, posteriormente, for determinado que a predição foi verdadeira, o ciclo é executado sem demora, senão o trabalho feito tem de ser anulado. A grande vantagem desta técnica ocorre na execução de ciclos, que têm de avaliar a mesma condição várias vezes e repetir o salto. Claro que eventualmente o salto não vai ocorrer e há trabalho desperdiçado, mas prova-se que o uso desta técnica compensa em termos de

performance.

Internamente o Pentium usa um barramento de dados de 32 bits, como o do i486, mas externamente é possível aceder à memória com um barramento de 64 bits, duplicando a quantidade de informação que é possível transferir num único ciclo de relógio. O processador suporta um modo de acesso à memória denominado *burst* o qual permite carregar elementos de 256 bits para a cache, num único ciclo de relógio.

O Pentium II é um processador superpipeline de 14 níveis possuindo extensiva predição de ramos e execução especulativa através da alteração dos registros originais das instruções por registros extras.

Possui três decodificadores, sendo um utilizado para instruções complexas e dois para as instruções simples, que decompõe uma instrução 80x86 em micro-operações, resultando em 3 a 6 micro operações por ciclo. Um máximo de 5 instruções podem ser executadas em paralelo e fora de ordem nas 6 unidades de execução (uma para FP, duas para operações inteiras, duas para endereçamento e uma de load/store). As micro-instruções executadas são retidas numa área temporária da qual são retiradas completas e por ordem, para evitar criar estados inconsistentes (meia instrução executada a meio de uma interrupção, por exemplo).

Como uma instrução 80x86 pode gerar várias micro-instruções, a taxa de execução é tipicamente de 3 instruções por ciclo de relógio, mas as instruções de 16 bits são tratadas de forma especial originando que sejam executadas mais lentamente do que num processador Pentium.

3.5. Processadores Superescalares versus Superpipeline

Como já vimos anteriormente, as técnicas superescalares exploram o paralelismo em nível de instruções (STALLINGS, 2002).

O termo superescalar, usado originalmente em 1987 (STALLINGS, 2002 *apud* AGERWALA e COCKE, 1989), refere-se a máquinas projetadas para melhorar o desempenho da execução de instruções escalares. Esse nome evidencia o contraste entre o objetivo de projetos desse tipo e o de projetos de processadores vetoriais.

Os processadores superescalares constituem mais de um pipeline rodando ao mesmo tempo. Isto significa buscar, decodificar, executar e armazenar mais de uma instrução no mesmo ciclo de execução da máquina.

Os superescalares já são uma evolução dos superpipelines. Os superpipelines defendem a subdivisão de cada estágio em sub-estágios controlados por um clock interno, mais rápido que o clock da máquina.

Na Figura 26, JOUPPI e HENNESSY (1991), demonstram um pipeline simples, um superpipeline e um superescalar.

Ainda é proposto por HWANG (1993) um processador superpipeline superescalar, que propõe mais de um pipeline executada dentro do clock interno, dividindo cada estágio em sub-estágios, como mostra a Figura 27.

A tecnologia de superpipeline e superescalar já está muito avançada e já é realidade de mercado em muitos processadores.

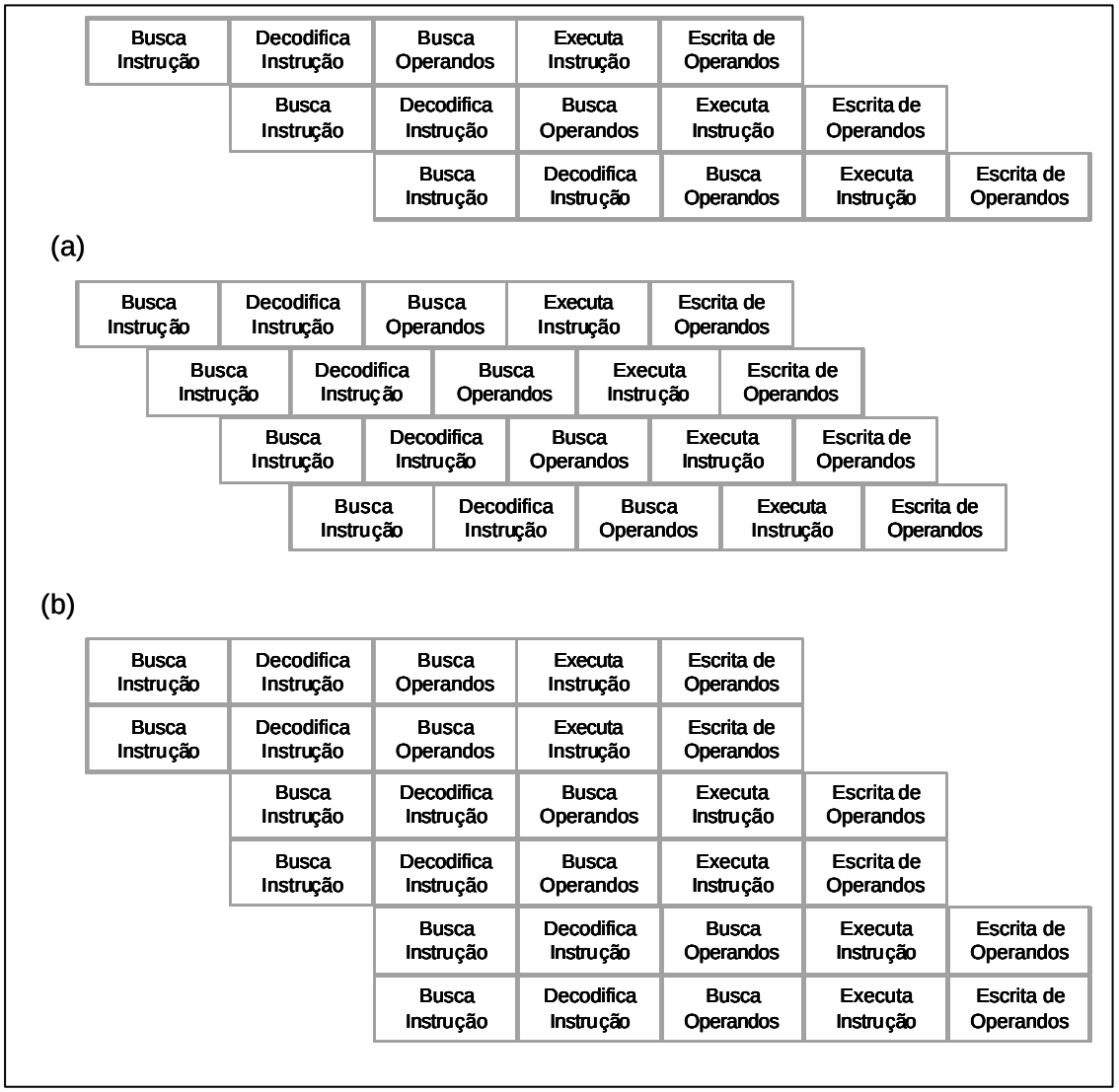

Figura 26: Pipeline simples. (a) Superpipeline (b) Superescalar (JOUPPI, 1991).

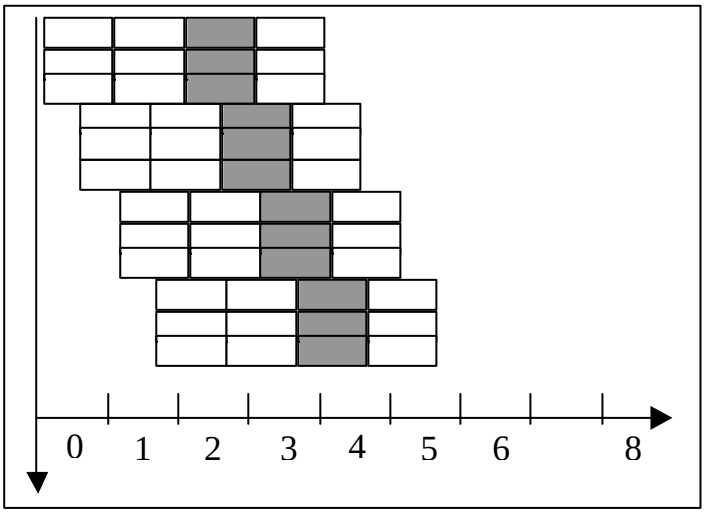

Figura 27: Superpipeline Superescalar (HWANG, 1993).

4- FPGA – PROGRAMMABLE FIELD GATE ARRAY

4.1- Introdução

A tecnologia de circuitos digitais vem se desenvolvendo rapidamente nas últimas décadas, ocasionando uma ampla transformação em todo o processo de desenvolvimento de hardware. Os elementos empregados em projetos de sistemas evoluíram de transistores a circuitos de larga escala de integração (*VLSI*). O emprego de poderosas ferramentas que auxiliam no desenvolvimento de sistemas, tais como o EDA (*Eletronic Design Automation*) e linguagens de descrição de hardware (HDLs), já estão consolidadas no meio acadêmico e industrial (TEIXEIRA, 2002).

Os primeiros chips programáveis pelo usuário foram as memórias PROM (*Programmable Read-Only Memory*), capazes de implementar circuitos lógicos. No início dos anos 70, foram introduzidos os PLAs (*Programmable Lógic Arrays*) os quais foram criados especialmente para a implementação de circuitos lógicos. Este dispositivo consiste em dois níveis de portas lógicas, um plano de portas AND e outro de portas OR, ambos programáveis. A Figura 28 ilustra o esquema simplificado de um PLA.

Com base na Figura 28, nota-se que o dispositivo é adequado para implementações de funções lógicas na forma de soma de produtos. Esta tecnologia possui um alto custo de fabricação e baixo desempenho em relação à velocidade, associados à presença de dois níveis de lógica configurável, difíceis de serem fabricados e que introduzem atrasos significativos de propagação dos sinais elétricos.

Seguindo a mesma linha de desenvolvimento, porém, com proposta de melhorar o desempenho dos dispositivos PLA, foram propostos os dispositivos PAL (*Programmable Array Logic*). A Figura 29 apresenta a estrutura básica do dispositivo.

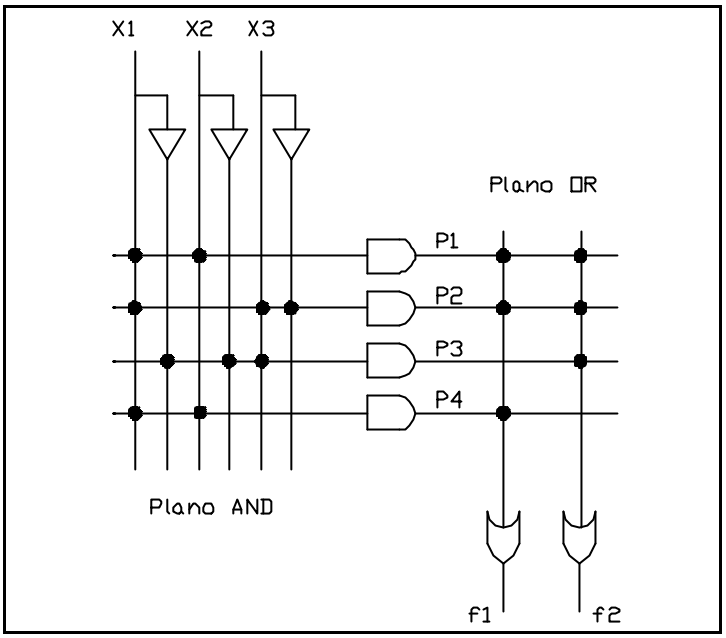

Figura 28: Esquema de um PLA.

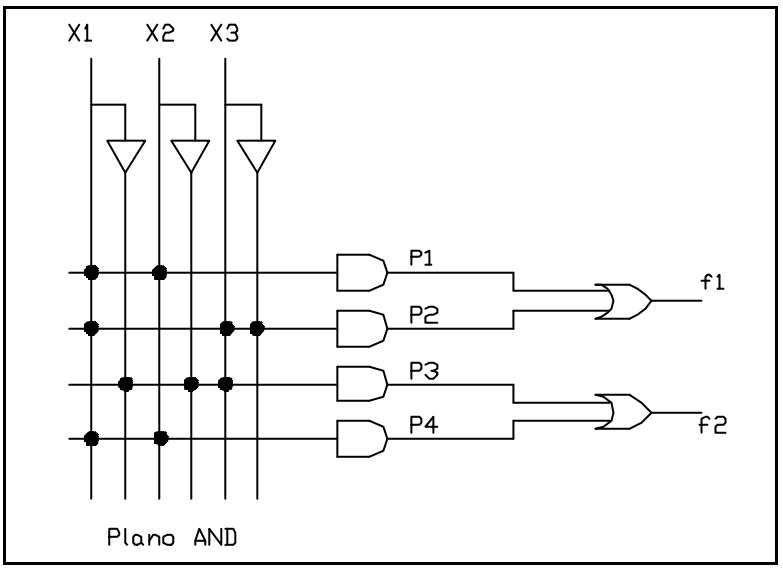

Figura 29: Estrutura básica PAL.

Os dispositivos PAL têm somente um nível de programação, assim possuindo um menor custo e melhor desempenho. Os dispositivos PLAs, PAL e outros similares, são classificados com SPLD (*Simple PLD*). Estes dispositivos possuem um custo relativamente baixo e um bom desempenho.

A necessidade de circuitos capazes de suportar aplicações mais complexas levou a integração de múltiplos SPLDs em uma única estrutura, tendo interconexões programáveis conectando os vários PLDs. Estes circuitos são chamados de CPLDs (*Complex PLDs*). A Figura 30 apresenta a estrutura de um CPLD.

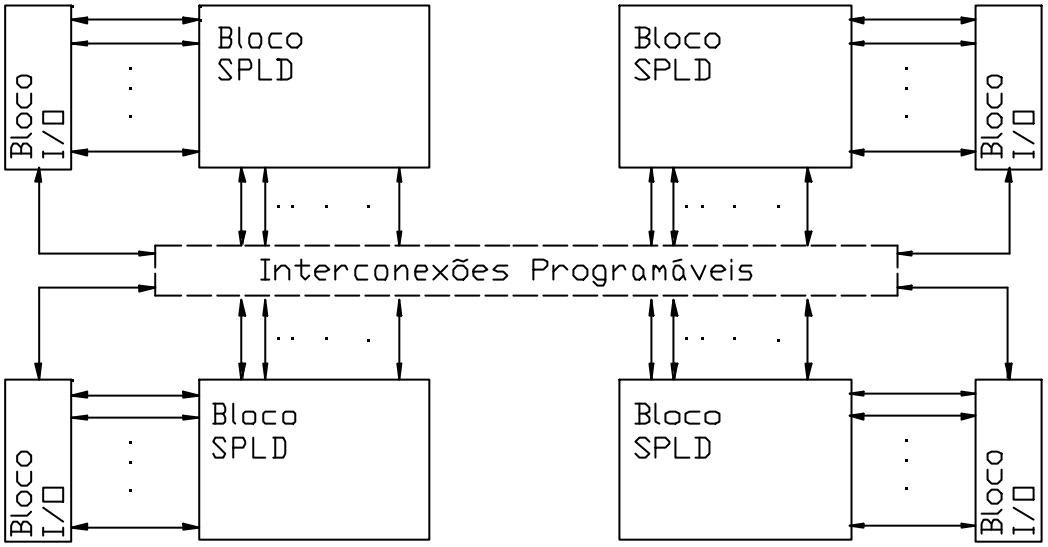

Figura 30: Estrutura de um CPLD.

Os dispositivos SPLDs e CPLDs podem ser utilizados em uma grande variedade de aplicações, porém eles apresentam uma capacidade lógica limitada a aplicações mais simples.

No ano de 1985 foi introduzido pela Xilinx Inc. o FPGA (*Field Programmable Gate Array*) que é um dispositivo lógico programável o qual suporta implementação de circuitos lógicos mais complexos. Os FPGAs são dispositivos reprogramáveis em campo e que podem ter sua configuração alterada sem a necessidade de removê-los do circuito global de um determinado sistema. Atualmente a capacidade destes circuitos chega a dezenas de milhares de portas lógicas. A Figura 31 apresenta a estrutura interna de um FPGA.

A estrutura interna dos FPGAs é formada por uma matriz de blocos lógicos reconfiguráveis chamados de CLBs (Configurable Logic Blocks), por uma rede de

interconexão programável e por blocos de entradas e saídas, os IOBs.

Os CLBs possuem uma arquitetura própria que varia de família para família de FPGAs e de fabricante para fabricante. A arquitetura básica é composta por pontos de entrada que se conectam a blocos com funções combinacionais, multiplexadores responsáveis pelo controle interno do fluxo de sinais e flip-flops ligados a saídas e com possibilidades de realimentação para as entradas.

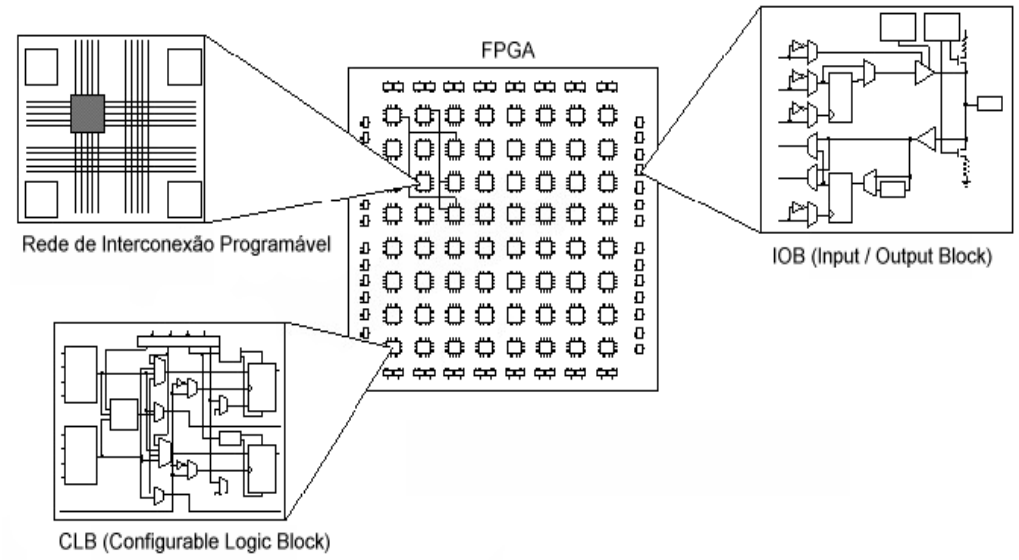

Figura 31: Estrutura interna de um FPGA.

A rede de interconexão programável é composta por diferentes segmentos de conexões capazes de interligar a maioria das entradas e saídas dos CLBs entre si e aos IOBs, permitindo a implementação de circuitos complexos. O modelo geral é composto por duas estruturas básicas. A primeira é o bloco de conexão que permite a conectividade das entradas e saídas de um bloco lógico com os segmentos de trilhas nos canais. A segunda estrutura é o bloco de comutação que permite conexão entre os segmentos de trilhas horizontais e verticais. É importante ressaltar que nem todas as arquiteturas seguem este modelo, porém o conceito geral é semelhante. A Figura 32 apresenta um modelo geral de arquitetura de roteamento de um FPGA.

Os circuitos FPGAs inicialmente não têm nenhuma funcionalidade especificada, há a necessidade de uma programação inicial para que determinadas funções possam ser executadas pelo circuito. Isto é realizado carregando um arquivo binário denominado *bit stream* em um arranjo de memórias estáticas (SRAM), cujo conteúdo (0 ou 1) define o estado do elemento reconfigurável do FPGA ou parte dele (COSTA, 2002).

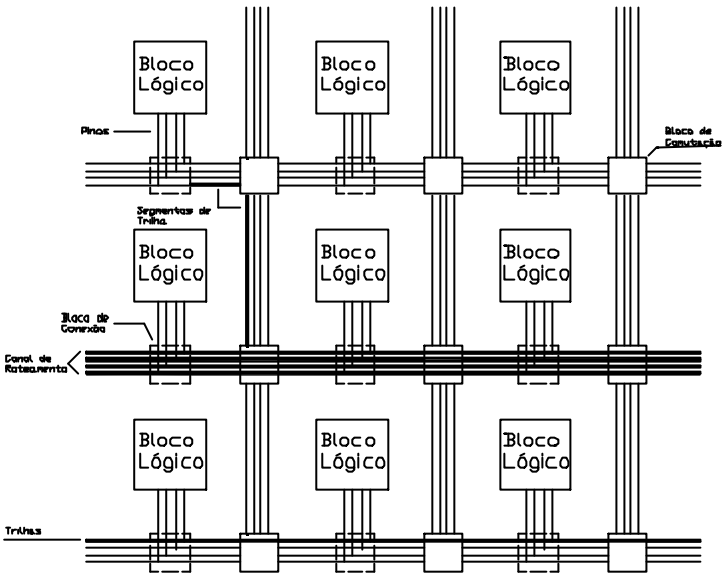

Figura 32: Arquitetura geral de roteamento de um FPGA.

4.2- Computação Reconfigurável

Conforme descrito no item 4.1, os FPGAs são dispositivos lógicos que podem ser reprogramados em campo, de forma repetidamente, sem necessidade da remoção do dispositivo do circuito global do sistema.

O processo de reconfiguração das primeiras FPGAs consumiam vários segundos ou mais. Recentemente, novos modelos de FPGAs podem ser reconfigurados com altas velocidades de forma que o hardware se adapta a mudanças no sinal de entrada ou em resposta a sinais externos, em tempo de execução. Este procedimento de reconfiguração em tempo de execução é chamado de *Run Time Reconfiguration – RTR.* Estes FPGAs são

também chamados de dispositivos dinamicamente reconfiguráveis e podem se reconfigurar total ou parcialmente. Esta característica de reconfigurabilidade dos FPGAs tem levado à criação de uma nova classe na organização dos computadores chamada de computação reconfigurável (TEIXEIRA, 2002).

A computação reconfigurável compartilha as características de um computador de uso geral, porém organiza a computação de maneira bastante diferente. Ao contrário dos computadores de uso geral que computam uma função seqüencialmente através de um conjunto de instruções no tempo, as arquiteturas reconfiguráveis computam as funções através de unidades configuradas no espaço, por meio dos diferentes blocos lógicos dentro dos FPGAs, tendo um perfil totalmente espacial paralelo.

A computação reconfigurável adere a uma abordagem mesclada entre os extremos dos computadores ASICs (*Circuitos Integrados de Aplicação Específica*) e processadores de uso geral, ou seja, um sistema reconfigurável típico tem uma aplicação mais abrangente do que um ASIC e possui um desempenho melhor do que o processador de uso geral em aplicações específicas (HENNESSY, 1996).

Os sistemas reconfiguráveis são plataformas de hardware onde sua arquitetura pode ser modificada via software, em tempo de execução ou não, para melhor se adequar a uma determinada aplicação em um instante particular. Os circuitos são configurados de forma que a seqüência de bits responsáveis pela sua configuração é buscada na memória e usada diretamente na configuração do hardware, não sendo necessária uma fase de interpretação (HAUCK, 2000).

Este tipo de configuração realizada por um projetista distingue os FPGAs dos processadores convencionais, pois estes últimos são configurados no processo de fabricação. Nos processadores de uso geral, as operações executadas são compostas e ordenadas usando registradores ou memórias para armazenar os resultados intermediários. Nos FPGAs a

reconfiguração é implementada pela configuração dos CLBs e pela interligação dos elementos que formam a matriz de interconexão.

Nesta linha de raciocínio, uma determinada aplicação que envolva códigos complexos e extensos, para serem carregados simultaneamente no hardware, poderia ser implementada de forma que o hardware assumisse configurações diferentes resolvendo partes da aplicação, até o resultado final.

A Figura 33 ilustra o diagrama esquemático da técnica *Run Time Reconfiguration*.

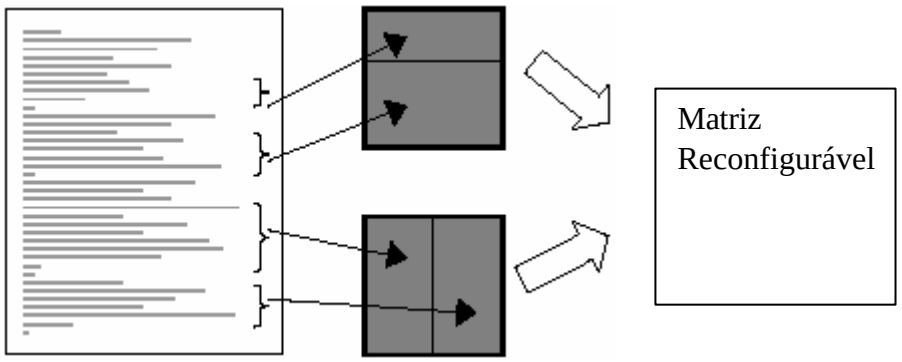

Figura 33: Diagrama esquemático da reconfiguração de hardware.

A técnica RTR é baseada no conceito de hardware virtual, semelhante à idéia de memória virtual. Nesta situação o hardware disponível é menor que o requerido, porém é realizada uma troca de configuração de acordo com as possíveis etapas de uma determinada aplicação.

Para os FPGAs atuais, existem três modelos de reconfiguração: o contexto simples, o contexto múltiplo e a reconfiguração parcial (HAUCK, 2000). No contexto simples, o FPGA é reconfigurado totalmente quando algum tipo de alteração necessita ser implementado. No contexto múltiplo, várias configurações pré-realizadas podem ser chaveadas a qualquer momento constituindo vários contextos simples multiplexados, no qual todo o FPGA é reconfigurado.

Em alguns casos a configuração não ocupa todo o hardware, de forma que é utilizada a reconfiguração parcial. Esta técnica realiza troca de configurações de forma semelhante à troca de informações entre um processador e uma memória RAM. A reconfiguração parcial permite seleção de dados a serem alterados, de forma que uma parte dos dados originais não seja alterada, funcionando normalmente.

Quando uma determinada configuração não requer a utilização de todo o FPGA, um certo número de configurações pode ser carregado para uma área disponível do FPGA que não esteja sendo utilizada (COSTA, 2002).

4.3-Vantagens da Utilização dos FPGAs

Os FPGAs podem ser utilizados em uma gama de aplicações. Uma delas é o desenvolvimento de protótipos de circuitos.

O desenvolvimento de protótipos está relacionado com a possibilidade de reprogramação do circuito reduzindo significantemente o tempo de desenvolvimento, pois os processos de simulação, teste, depuração e alteração do projeto são bastante ágeis.

Neste tipo de desenvolvimento, o comportamento do FPGA pode ser definido através da programação em *schematic*, utilizando uma biblioteca para gerar o diagrama elétrico do circuito ou uma linguagem de descrição de hardware, geralmente o VHDL. Desta forma, é produzido um arquivo de configuração que, quando carregado no FPGA, faz com que sua estrutura se comporte como o circuito projetado.

A implementação de algoritmos em hardware é particularmente eficiente para aplicações orientadas a bit, ou seja, que trabalham com manipulação direta de bits. Como exemplo de tais aplicações, pode-se considerar a compressão de dados, reconhecimento de padrões, criptografia, tratamento de imagens, processamento de sinais e outros.

4.4- A Família Virtex II

A família Virtex II da Xilinx é uma plataforma de FPGAs composta desde unidades de baixa densidade até unidades de alta densidade, relacionado ao número de estruturas lógicas internas. Os membros pertencentes a esta família de FPGAs podem ser aplicados em soluções voltadas às telecomunicações, redes wireless, comunicação de dados, vídeo e aplicações em processamento digital de sinais.

Sua estrutura de fabricação está relacionada à tecnologia CMOS otimizada para utilização em alta velocidade com taxa de clock na ordem de 420 MHz e com baixo consumo de energia. A Tabela 5 apresenta os FPGAs pertencentes à família Virtex II e suas principais características.

Tabela 5: FPGAs da família Virtex II

Modelo	Portas	CLB 1 CLB = 4 Blocos = Max 128 bits			Blocos Multipli-cadores	Blocos SelectRAM		DCMs	Max. E/S Pinos
		Linhas x Colunas	Blocos	Max. RAM Distribuida Kbits		Blocos 18 Kbits	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

Os FPGAs da família Virtex II são estruturas programáveis pelo usuário compostos por vários elementos internos. São compostos por blocos de entradas e saídas (*IOBs*) e blocos lógicos configuráveis (*CLBs*). Os blocos de entrada e saída fornecem interfaces entre os pinos do chip e a lógica configurável interna.

A configuração lógica interna inclui quatro elementos principais organizados em uma estrutura regular: os blocos configuráveis (*CLBs*), o bloco de memória RAM, os blocos multiplicadores e o gerenciador de clock digital (*DCM*). A Figura 34 apresenta a arquitetura interna dos componentes da família Virtex II.

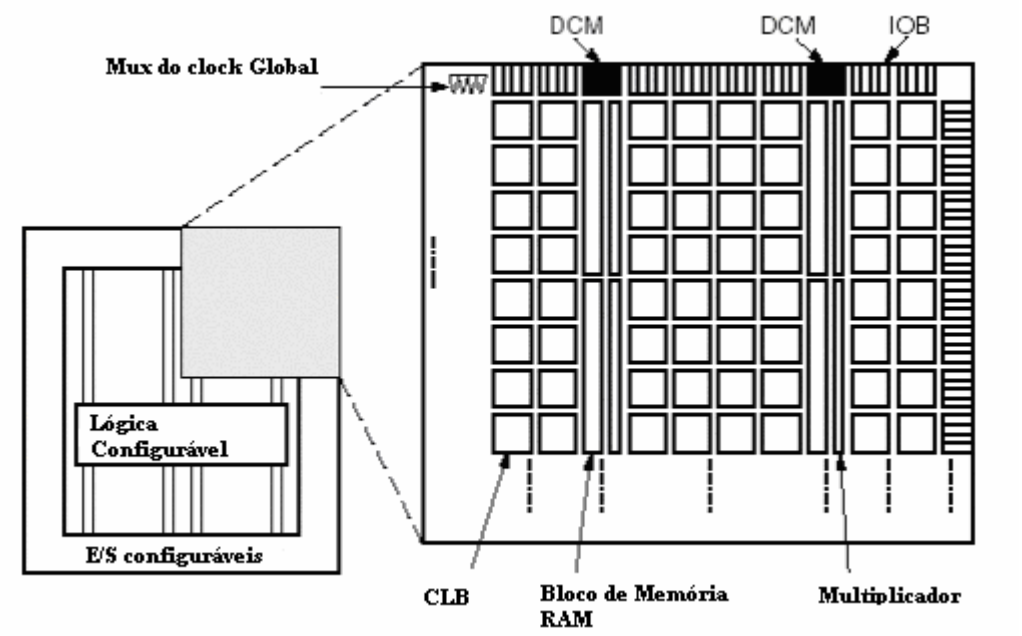

Figura 34: Arquitetura interna da família Virtex II.

Os blocos lógicos configuráveis (*CLBs*) fornecem elementos funcionais para a realização da lógica combinacional e seqüencial, incluindo elementos básicos de armazenamento de dados. São formados por quatro blocos iguais e dois *buffers* de três estados. Os quatro blocos são compostos por dois geradores de função, dois elementos de armazenamento de dados, portas lógicas aritméticas, multiplexadores e portas OR.

Os geradores de função podem ser configurados como LUTs de quatro entradas, registradores de deslocamento de 16 bits ou como memória RAM distribuída de 16 bits. Os elementos de armazenagem de dados são formados por flip flops tipo D.

Cada CLB tem um sistema de interconexão rápido conectado a uma matriz de chaveamento para acessar os recursos gerais de roteamento interno. A Figura 35 apresenta o diagrama geral do CLB e a Figura 36, as possibilidades de configuração interna do CLB.

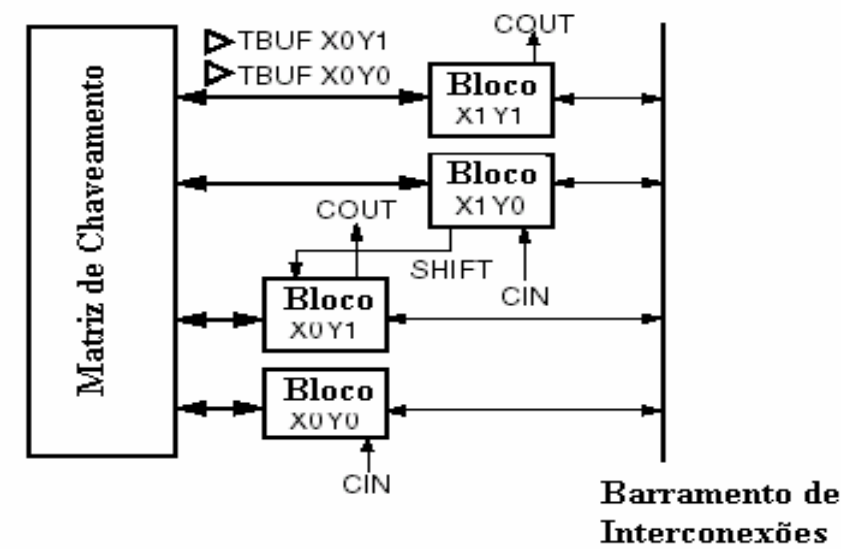

Figura 35: Diagrama geral do CLB.

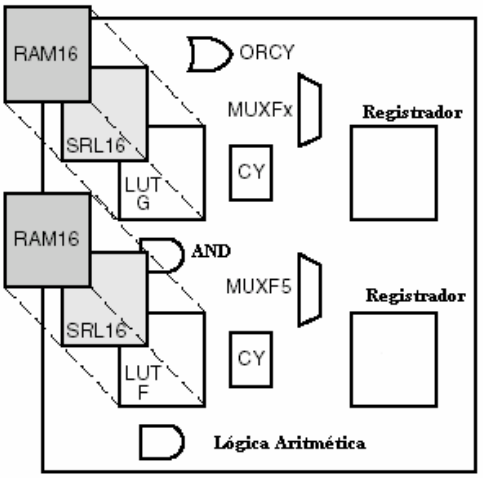

Figura 36: Possibilidades de configuração de cada bloco interno do CLB.

De acordo com a Figura 36, pode-se observar algumas possibilidades de configuração utilizadas nos blocos internos dos CLBs, tais como LUTs (*Look-up tables*), registradores, memória RAM distribuída, multiplexadores, somadores, etc. A Figura 37 apresenta detalhes dos blocos que compõem o CLB.

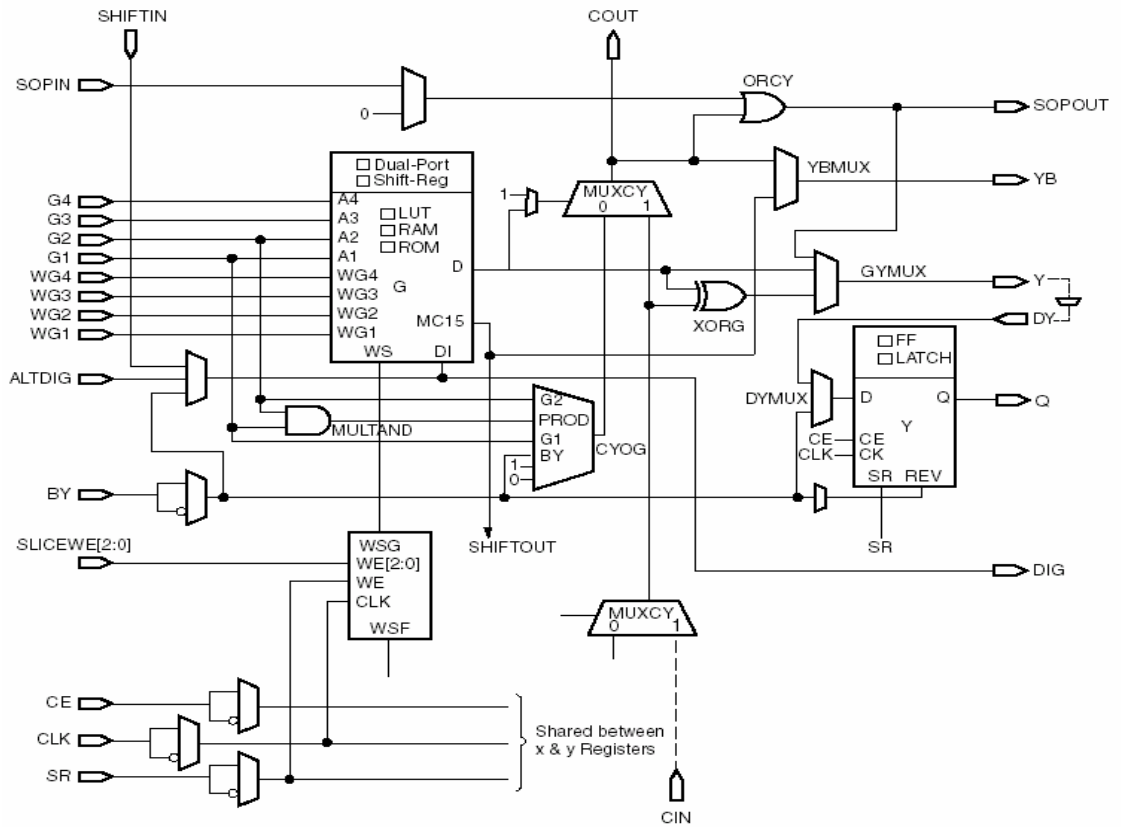

Figura 37: Detalhes dos blocos lógicos que compõe o CLB.

Dentre os elementos que compõem o CLB é importante destacar a capacidade de memória presente em cada bloco. Cada LUT pode implementar um recurso de memória RAM de 16 x 1 bit denominado elemento de memória RAM distribuída (Xilinx, 2002). Os elementos de memória RAM dentro de um CLB podem ser configurados para operar da seguinte forma:

- Uma porta com 16 x 8 bits
- Uma porta com 32 x 4 bits
- Uma porta com 64 x 2 bits

- Uma porta com 128 x 1 bit
- Duas portas 16 x 4 bits
- Duas portas 32 x 2 bits
- Duas portas 64 x 1 bit

Para as configurações de uma porta, a memória RAM distribuída tem somente um endereço para escrita e leitura. Para configurações com duas portas, a memória RAM distribuída tem uma porta para escrita síncrona e leitura assíncrona, e uma outra porta para leitura assíncrona. A LUT tem entradas de endereços de leitura A1, A2, A3 e A4 separadas das entradas de endereços de escrita WG1/WF1, WG2/WF2, WG3/WF3 e WG4/WF4.

No modo de operação com uma porta, os endereços de leitura e escrita compartilham o mesmo barramento. No modo de operação com duas portas, um gerador de função (R/W) é conectado com os endereços de leitura e escrita. O segundo gerador de função tem A entradas de leitura conectadas à segunda porta de endereços somente de leitura e as W entradas para escrita são compartilhadas com a primeira porta de endereços para leitura e escrita.

5. VHDL

A linguagem VHDL foi desenvolvida como um padrão de linguagem de programação que descreve a estrutura e o funcionamento de circuitos integrados digitais.

Na tabela 6 é dado o significado da sigla VHDL.

Tabela 6 - Significado da sigla VHDL

Very **H**ight **S**peed **I**ntegrated **C**ircuit (**VHSIC**)
Hardware
Description
Language

VHDL é uma linguagem de descrição de *hardware* que representa entradas e saídas, comportamento e funcionalidade de circuitos. Atualmente faz parte da maioria das ferramentas de projetos eletrônicos (Marques, 2000). VHDL é uma forma de se descrever, através de um programa, o comportamento de um circuito ou componente digital (Marchi, 2002).

A utilização do VHDL possibilita o desenvolvimento metodológico de sistemas complexos, e permite a descrição do sistema em partes, ou seja, a decomposição de um grande sistema em subsistemas indicando como estes subsistemas estão conectados. Ele permite a utilização de formas padrões de programação no desenvolvimento de um sistema digital e, como conseqüência, também permite a simulação do sistema digital antes de sua implementação (Parma, 2003). Além disto, por ser uma linguagem de programação de circuitos digitais, o código VHDL é desenvolvido independentemente do CI a ser utilizado permitindo, desta forma, uma grande flexibilidade na hora da implementação do sistema.

A estrutura de um programa VHDL é baseada em blocos, com a finalidade de expor os recursos que a linguagem oferece para a realização de um projeto qualquer, em suas diversas etapas, demonstrando como são utilizados. Os principais blocos são: a declaração de

entidades (*entity*), a arquitetura (*architecture*), os sub-programas, a declaração de pacotes (*package*) e o corpo do pacote (*package body*).

Uma linguagem de descrição de hardware descreve o que um sistema faz e como. Esta descrição é um modelo do sistema hardware, que será executado em um software chamado simulador. Um sistema descrito em linguagem de hardware pode ser implementado em um dispositivo programável (FPGA), permitindo assim o uso em campo do seu sistema, tendo a grande vantagem da alteração do código a qualquer momento.

A idéia da criação de uma linguagem de descrição de *hardware* partiu do Departamento de Defesa dos Estados Unidos da América. As forças armadas americanas compravam grande quantidade de placas de circuitos impressos, sendo que muitas delas compostas de circuitos integrados de aplicação específica (ASIC).

Como era comum, empresas da área de eletrônica, mudarem de área ou trocar de ramo, necessitava-se garantir a reposição das peças durante a vida útil das placas, com isso o Departamento de Defesa Americano iniciou o desenvolvimento de uma linguagem padrão de descrição de *hardware.*

A linha cronológica da história do surgimento do VHDL é apresentada na tabela 7.

Tabela 7 - Cronologia do surgimento da Linguagem VHDL

Ano	**Ocorrência**
1968	Foram desenvolvidas as primeiras linguagens de descrição de *hardware*.
1970	Tinham-se inúmeras linguagens com sintaxe e semântica incompatíveis
1973	Surge o primeiro esforço de padronização da linguagem, comandado pelo projeto CONLAN (*CONsensus LANguage*), cujo o objetivo principal era: definir formalmente uma linguagem de multi-nível com sintaxe única e semântica igual. Paralelamente, iniciou-se outro projeto financiado pelo Departamento de Defesa Americano cujo o objetivo era criar uma linguagem de programação.
1983	Em janeiro de 1983, foi publicado o relatório final do projeto CONLAN. Neste mesmo ano,

	também foi publicado o relatório final do projeto Departamento de Defesa Americano, que deu origem a linguagem ADA. Em março de 83, o Departamento de Defesa Americano começou o programa VHSIC, afim de melhorar a tecnologia de concepção, engenharia e construção nos EUA. Participaram deste projeto a IBM, Intermetrics e Texa s Instruments.
1986	A Intermetrics desenvolveu um compilador e um simulador. Além disso, foi criado um grupo de padronização da IEEE para VHDL.
1988	Primeiros *softwares* são comercializados.
1991	Recomeçou um novo processo de padronização, cujo objetivo era a coleta e análise de requisitos, definição dos objetivos e a especificação das modificações à linguagem.
1993	Um novo padrão é publicado, chamado VHDL-93, padronizado IEEE Std 1164-1993.
1997	Em dezembro de 97 foi publicado o manual de referência da linguagem VHDL.

A descrição de um sistema em VHDL apresenta inúmeras vantagens, tais como o Intercâmbio de projetos entre grupos de pesquisa sem a necessidade de alteração; permite ao projetista considerar no seu projeto os *delay´s* comuns aos circuitos digitais; a linguagem independe da tecnologia atual, ou seja, você pode desenvolver um sistema hoje e implementá-lo depois; os projetos são fáceis de serem modificados; o custo de produção de um circuito dedicado é elevado, enquanto que usando VHDL e Dispositivos Programáveis, isto passa a ser muito menor; reduz consideravelmente o tempo de projeto e implementação.

6. RTRASSOC51 : Módulo Pipeline para um Processador com Arquitetura Harvard Superescalar Embutido (PAHSE)

O RtrASSoc51 é um sistema em chip adaptável, superescalar e reconfigurável conforme descrito na figura 38. O RtrASSoc51 será utilizado em sistemas embarcados que necessitam de alta capacidade, desempenho e baixo custo, baseado em sistemas em chip programáveis (PSOC), onde parte do sistema será um processador Harvard superescalar embutido (PAHSE) com três linhas de pipeline, uma outra parte será composta por um sistema operacional embutido (SOE) e finalmente a parte reconfigurável, composta por rotinas de reconfiguração (RR) obtidas a partir do programa da aplicação. O processador Harvard superescalar embutido (PAHSE) é parte integrante do PSOC, baseada na arquitetura Harvard com barramento de dados e instruções independentes de 8 bits, conjunto de instruções e pipeline baseado na estrutura utilizada no microcontrolador PIC16F84, com objetivo de atingir o desempenho necessário ao processamento digital de imagens, voltado ao reconhecimento de padrões relacionado às formas básicas de objetos.

Figura 38: Estrutura do RtrASSoc51.

A arquitetura completa proposta para o PAHSE é composta por uma estrutura superescalar com três linhas de pipeline e com arquitetura Harvard, utilizando os princípios de um pipeline simples utilizado no microcontrolador PIC16F84, com o conjunto de instruções totalmente baseado na família do microcontrolador 8051. A estrutura completa do PAHSE é apresentada na Figura 39.

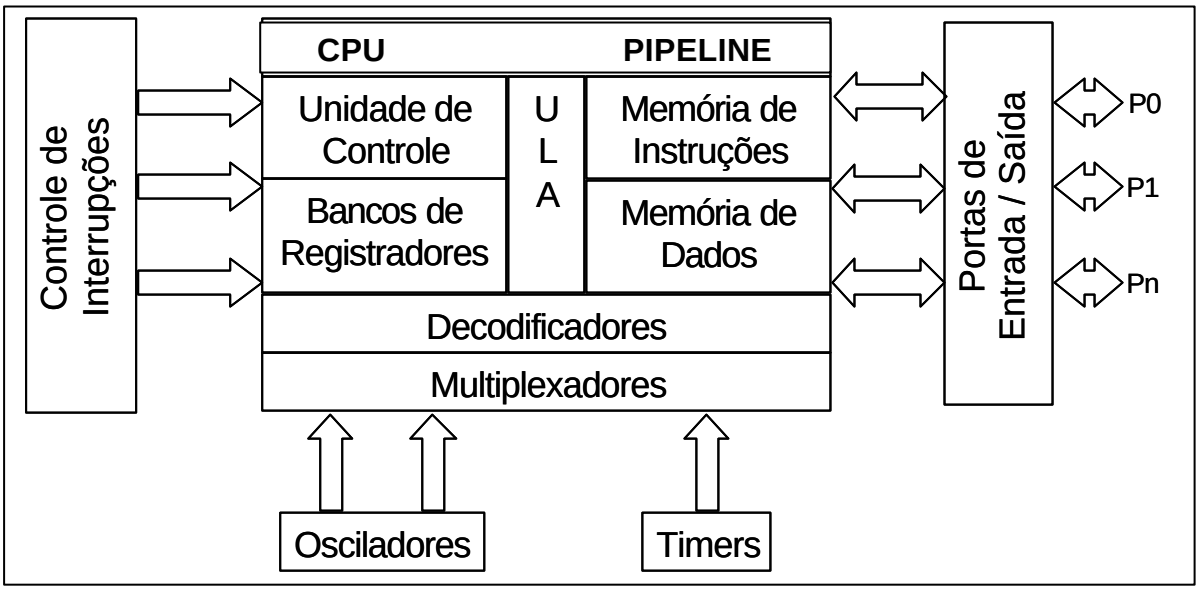

Figura 39: Estrutura completa do PHSE.

A estrutura interna da CPU do PAHSE está descrita na Figura 40, que mostra todas as unidades de execução desenvolvidas e testadas através de simulação na ferramenta da XILINX.

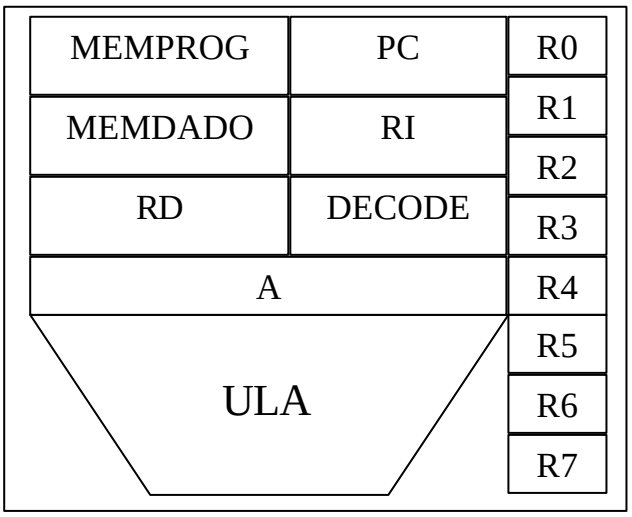

Figura 40: Unidades de Execução da CPU do PHSE.

Obedecendo às premissas da arquitetura Harvard, da mesma maneira que no microcontrolador PIC, o PAHSE possui:

a) uma memória de programa (MEMPROG);

b) uma memória de dados (MEMDADO) separada e com barramentos distintos da memória de programa;

c) um registrador de instrução (RI);

d) um registrador de dados (RD);

e) um contador de programas (PC) que realiza os gatilhos na memória de programa (MEMPROG);

f) uma unidade DECODE que realiza a decodificação das instruções recebidas da memória de programa (MEMPROG). Para o PAHSE desenvolvido, foi implementada a codificação das instruções do microcontrolador 8051;

g) os registradores A, R0, R1, R2, R3, R4, R5, R6 e R7 também incorporaram a estrutura do PAHSE, para entrar em conformidade com as instruções do microcontrolador 8051 que estão baseadas nestes registradores, as quais foram utilizadas para esta implementação do PAHSE;

h) uma unidade lógico-aritmética (ULA) está implementada na estrutura básica da CPU do processador para as operações realizadas na execução das instruções submetidas ao RtrASSoc51.

Estas unidades estão implementadas em VHDL e foram executadas na ferramenta Foundation Series 3.1i da XILINX.

6.1 Unidade de Controle do PAHSE

Para esta implementação foi determinado, como base, apenas uma linha de pipeline e algumas instruções da família do microcontrolador 8051.

Assumindo um pipeline com quatro estágios (Busca, Decodificação, Execução e Armazenamento) e um modelo de controle baseado no conceito de microprogramação foi implementado um registrador de controle (REGCONTROLE) com 16 bits, onde cada bit está associado ao início e fim da execução de uma das unidades de execução da CPU, como mostra a Figura 41.

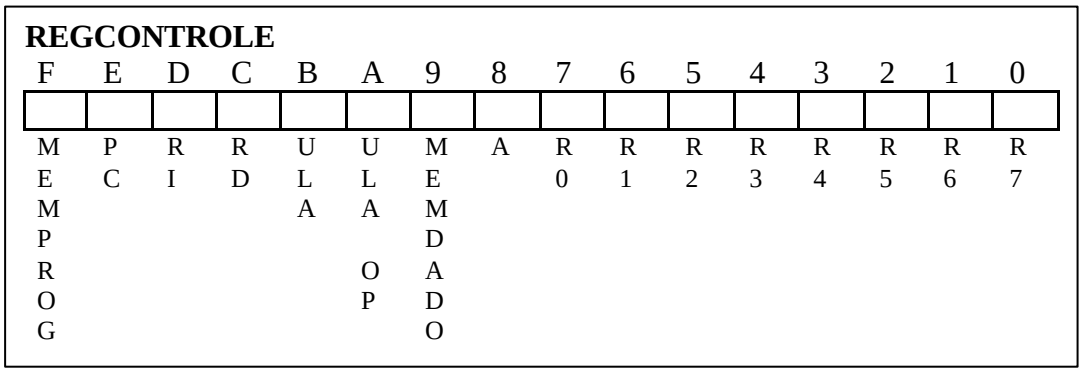

Figura 41: Registrador de controle da CPU do PHSE.

Cada bit do REGCONTROLE foi associado a uma unidade de execução, e é com a mudança de 0 (zero) para 1 (um) e vice versa neste bit que o microprograma controla a ativação e desativação de cada uma das unidades de execução respectivamente. Por exemplo, quando o pipeline estiver liberado para buscar uma nova instrução, o bit do REGCONTROLE que se encontra na posição 16, ou F em Hexadecimal, é posicionado com o valor 1, que "acorda" a memória de programa para realizar a busca de uma instrução, logo em seguida, o microprograma posiciona a mesma posição do REGCONTROLE com o valor 0, deixando a memória de programa em estado de espera, até que seja ativada novamente para uma nova busca. Desta maneira cada um dos 16 bits do REGCONTROLE tem a função de realizar a sincronia entre as unidades de execução do PAHSE.

Para demonstrar a sincronização do PAHSE, a Figura 42 apresenta um diagrama com os níveis de transferência entre os registradores, denominado RTL (Register Transfer Level) do PAHSE.

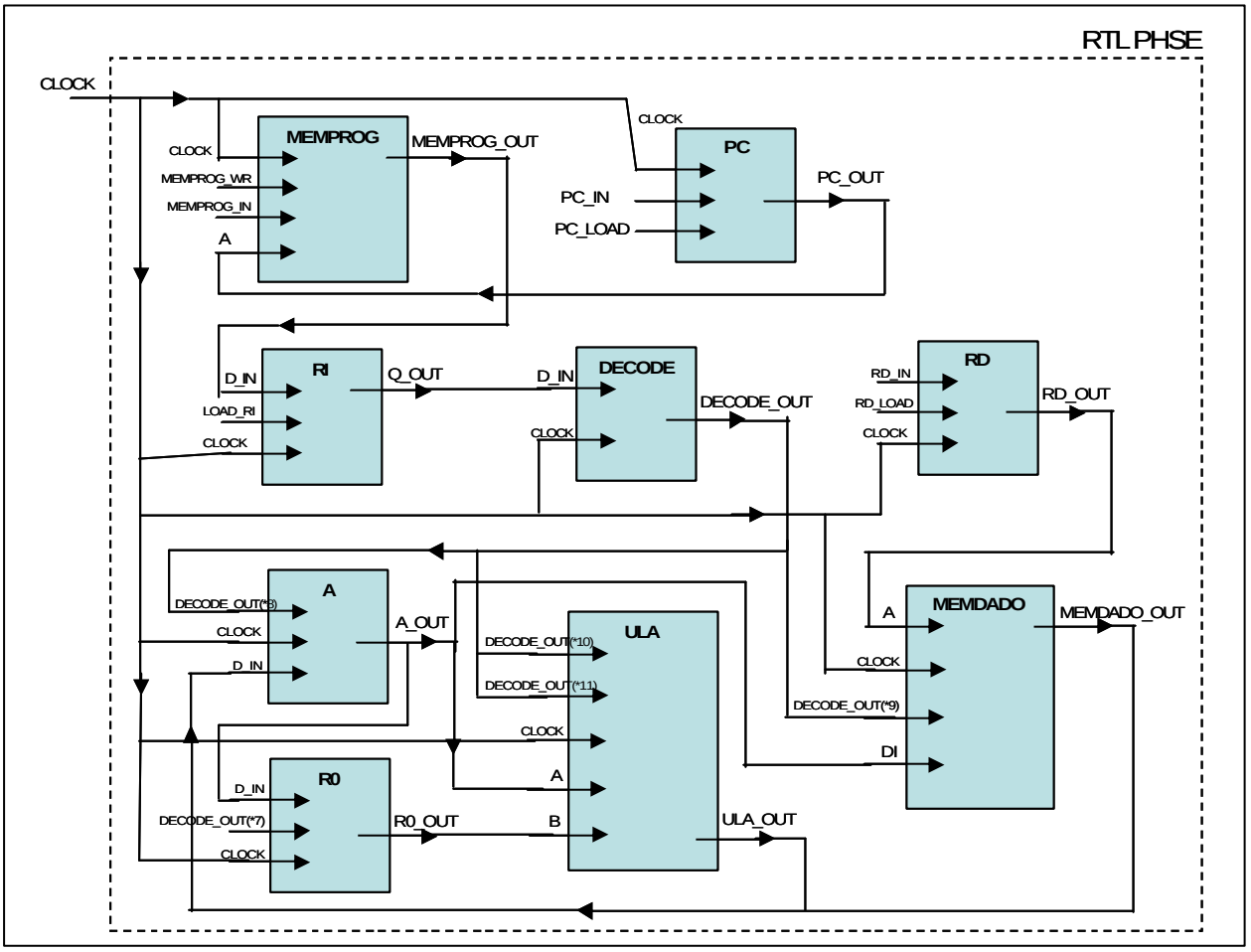

Figura 42: RTL do PHSE.

O diagrama de transferência entre os registradores mostra as entradas e saídas existentes nas unidades de execução.

O exemplo de um diagrama de sincronismo para o RTL descrito acima é mostrado na Figura 43.

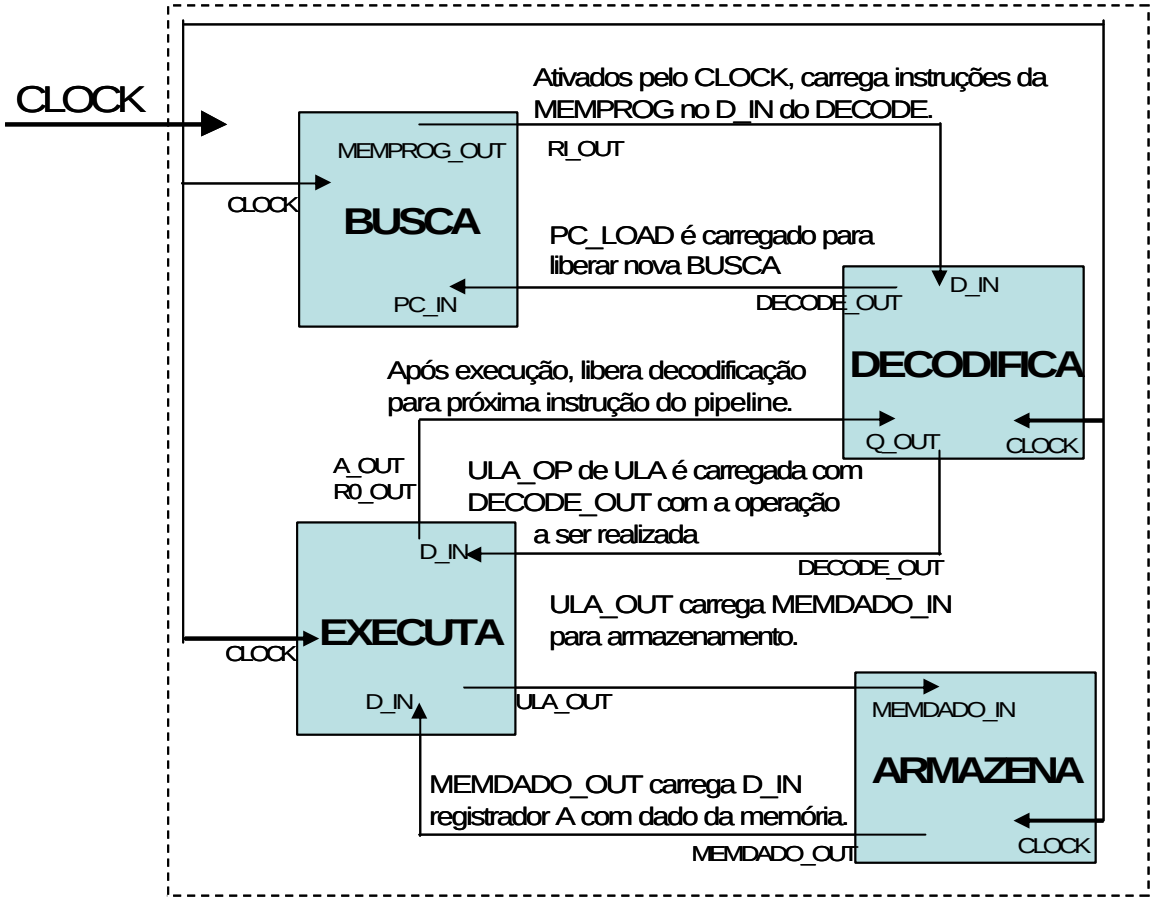

Figura 43: Exemplo de diagrama de estado do PHSE.

6.2 Simulações do PAHSE

As principais simulações do PAHSE, estão demonstradas com os seu códigos VHDL desenvolvidos na ferramenta da Xilinx, porém, todos os códigos VHDL criados para este projeto encontram-se anexos no final da dissertação.

Na simulação verifica-se na Figura 44 que o sinal A (Address) recebe o endereço, e o sinal WR_EN (Write/Read Enable) pode alternar entre caixa baixa (0) para leitura, e caixa alta (1) para a gravação na memória. Em particular com o WR_EN setado para leitura e o sinal A sendo alimentado a cada clock por endereços em um arquivo texto fornecido ao simulador, o sinal DO (Date Out) é carregado a clock com o dado correspondente ao endereço na memória de programa do PAHSE.

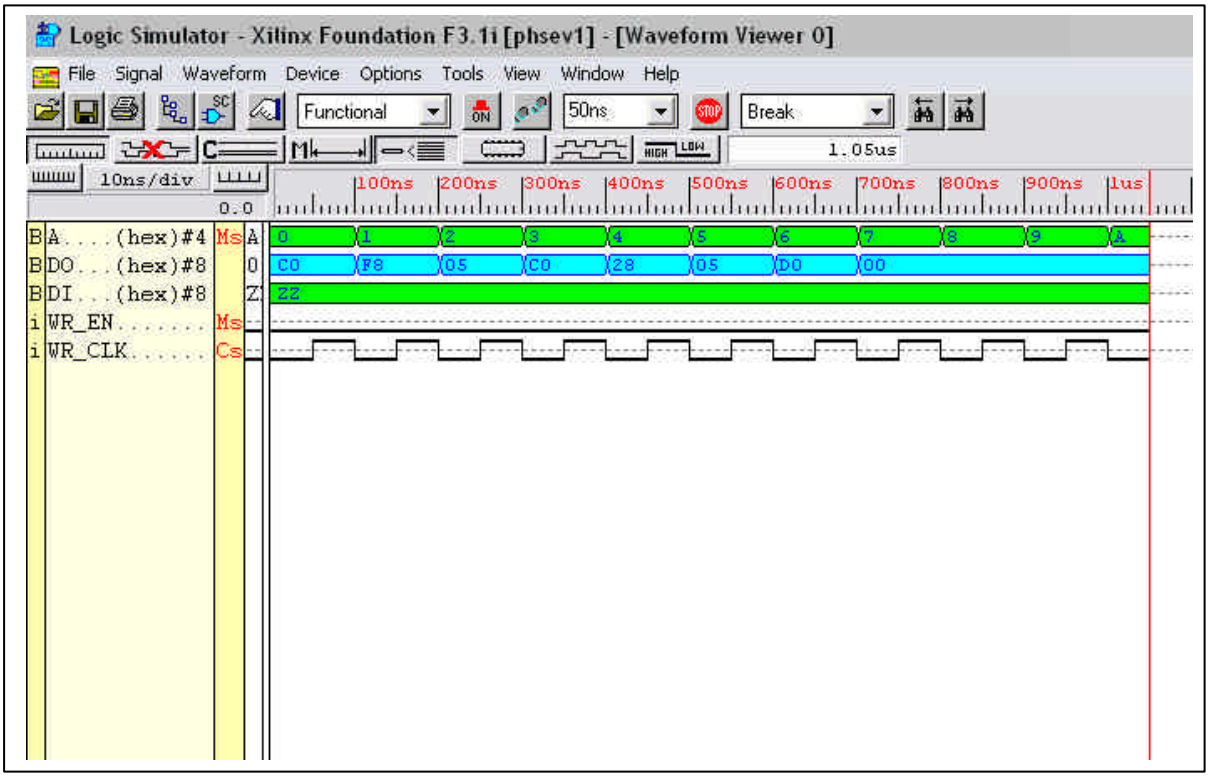

Figura 44: Simulação da memória de programa.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity teste_busca is
    port (
           CLOCK: in STD_LOGIC;
           PC_IN: in STD_LOGIC_VECTOR(3 DOWNTO
0);
           PC_LOAD: in STD_LOGIC;
           MEMPROG_WR: in STD_LOGIC;
           PC_OUT: out STD_LOGIC_VECTOR(3 DOWNTO
0);
           MEMPROG_OUT: out STD_LOGIC_VECTOR(7
DOWNTO 0)
    );
end teste_busca;

architecture teste_busca_arch of teste_busca is

component MEMPROG is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO 0);
           DI: IN std_logic_vector(7 DOWNTO 0);
           WR_EN: IN std_logic;
           WR_CLK: IN std_logic
    );
end component;

component PC is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO 0);
           LOAD: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3 DOWNTO
0)
         );
end component;
```

```
signal MEMPROG_IN : std_logic_vector(7
DOWNTO 0);
signal PC_OUT_SINAL : std_logic_vector(3
DOWNTO 0);


begin

  I02 : PC      port map
       ( D_IN => PC_IN,
         LOAD => PC_LOAD,
         CLOCK => CLOCK,
         Q_OUT => PC_OUT_SINAL
       );

  PC_OUT <= PC_OUT_SINAL;

  I01 : MEMPROG port map
       ( A => PC_OUT_SINAL,
         DO => MEMPROG_OUT,
         DI => MEMPROG_IN,
         WR_EN => MEMPROG_WR,
         WR_CLK => CLOCK
       );

end teste_busca_arch;
```

Figura 45: Código VHDL para a Memória de Programas.

O código VHDL utilizado para a simulação da Figura 44 é apresentado na Figura 45. Em toda a programação VHDL foi utilizado a instanciação de componentes configurados na ferramenta LOGIBlox, que integra o Foundation Series da Xilinx.

Após todas as unidades de execução estarem validadas, individualmente, através de simulações, foram feitas instanciações em conjunto para validar os estágios do pipeline, busca, decodificação, execução e armazenamento. Para estas simulações, foi utilizado um código assembler com as instruções do microcontrolador 8051. Este código foi colocado na memória de programa, e alguns dados colocados na memória de dados. A Figura 46 mostra a instanciação das unidades necessárias para a simulação da busca e da decodificação de instruções carregadas da memória de programa do PAHSE, e na Figura 47, o código VHDL desta simulação.

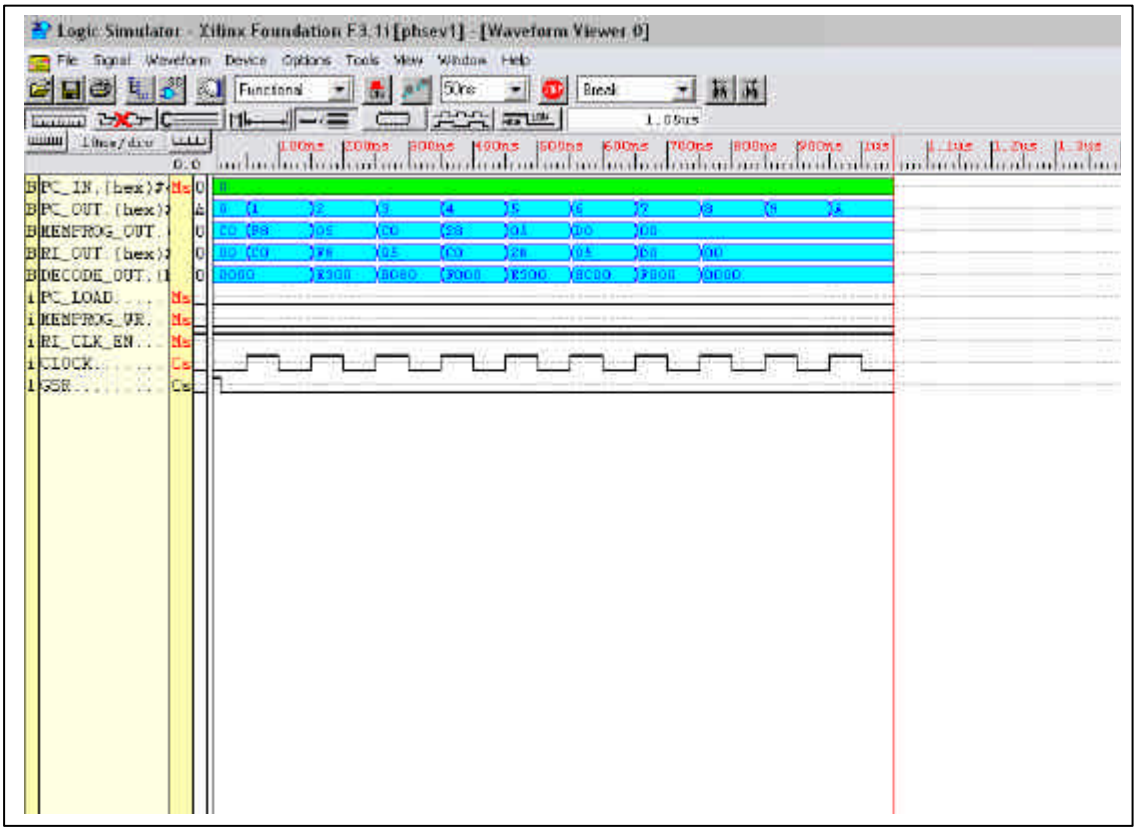

Figura 46: Simulação da busca e decodificação do pipeline.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tBI_DE is
    port (
           CLOCK: in STD_LOGIC;
           PC_IN: in STD_LOGIC_VECTOR(3 DOWNTO
0);
           PC_LOAD: in STD_LOGIC;
           MEMPROG_WR: in STD_LOGIC;
           RI_CLK_EN: in STD_LOGIC;
           PC_OUT: out STD_LOGIC_VECTOR(3
DOWNTO 0);
           MEMPROG_OUT: out STD_LOGIC_VECTOR(7
DOWNTO 0);
           RI_OUT: out STD_LOGIC_VECTOR(7
DOWNTO 0);
           DECODE_OUT:  out
STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
end tBI_DE;

architecture tBI_DE_arch of tBI_DE is

component MEMPROG is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO
0);
           DI: IN std_logic_vector(7 DOWNTO
0);
           WR_EN: IN std_logic;
          WR_CLK: IN std_logic
    );
end component;

component PC is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO
0);
           LOAD: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3
DOWNTO 0)
         );
end component;

component RI is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO
0);
          CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7
DOWNTO 0)
    );
end component;
```

```
component decode is

    port (
        D_IN: in STD_LOGIC_VECTOR (7 downto
0);
        CLOCK: in STD_LOGIC;
        D_OUT: out STD_LOGIC_VECTOR (15 downto
0)
    );
end component;

signal MEMPROG_IN : std_logic_vector(7 DOWNTO
0);
signal PC_OUT_SINAL : std_logic_vector(3
DOWNTO 0);
signal MEMPROG_OUT_SINAL : std_logic_vector(7
DOWNTO 0);
signal RI_OUT_SINAL : std_logic_vector(7
DOWNTO 0);


begin

  I01 : PC      port map
       ( D_IN => PC_IN,
         LOAD => PC_LOAD,
         CLOCK => CLOCK,
         Q_OUT => PC_OUT_SINAL
       );

  PC_OUT <= PC_OUT_SINAL;

  I02 : MEMPROG port map
       ( A => PC_OUT_SINAL,
         DO => MEMPROG_OUT_SINAL,
         DI => MEMPROG_IN,
         WR_EN => MEMPROG_WR,
         WR_CLK => CLOCK
       );

  MEMPROG_OUT <= MEMPROG_OUT_SINAL;

  I03 : RI      port map
       ( D_IN => MEMPROG_OUT_SINAL,
         CLK_EN => RI_CLK_EN,
         CLOCK => CLOCK,
         Q_OUT => RI_OUT_SINAL
       );

  RI_OUT <= RI_OUT_SINAL;

  I04 : DECODE  port map
       ( D_IN => RI_OUT_SINAL,
         CLOCK => CLOCK,
         D_OUT => DECODE_OUT
       );


end tBI_DE_arch;
```

Figura 47: Código VHDL para a simulação da busca e decodificação do pipeline.

Na instanciação das unidades para a simulação da busca e da decodificação, o contador de programas (PC) efetua os gatilhos na memória de programa através do sinal PC_OUT, com o endereço a memória, por sua vez, carrega o registrador de instrução (RI) através do sinal MEMPROG_OUT. O registrador de instrução, no clock seguinte, alimenta o decodificador (DECODE), utilizando o sinal RI_OUT. O decodificador reconhece a instrução e carrega o sinal de saída DECODE_OUT, que contém os gatilhos necessários para o registrador de controle (REGCONTROLE) acordar as unidades que iniciarão o próximo estágio do pipeline que é a execução.

Finalmente, foram instanciadas todas as unidades de execução para a simulação do ciclo completo de instrução, utilizando todos os estágios do pipeline. Para esta simulação a Figura 46, mostra o código assembler utilizado e como foram inicializadas as memórias de programa e de dados.

A memória de programa foi carregada com o código hexadecimal de cada instrução do código assembler, obedecendo à mesma convenção da tabela de instruções do microcontrolador 8051. A simulação ocorreu tendo como objetivo efetuar a soma de dois valores carregados da memória de dados e o armazenamento deste resultado, também na memória de dados.

CÓDIGO ASSEMBLER	MEMÓRIA DE PROGRAMA		MEMÓRIA DE DADOS	
PUSH RD	0	C0	0	6
MOV R0,A	1	F8	1	9
INC RD	2	05	2	0
PUSH RD	3	C0	3	0
ADD A,R0	4	28	4	0
INC RD	5	05	5	0
POP RD	6	D0	6	0

Figura 48: Código Assembler, Memória de Programa e Memória de Dados inicializadas para a simulação do ciclo de execução no PAHSE.

Na Figura 49, apresentamos a primeira simulação executada com a instanciação de

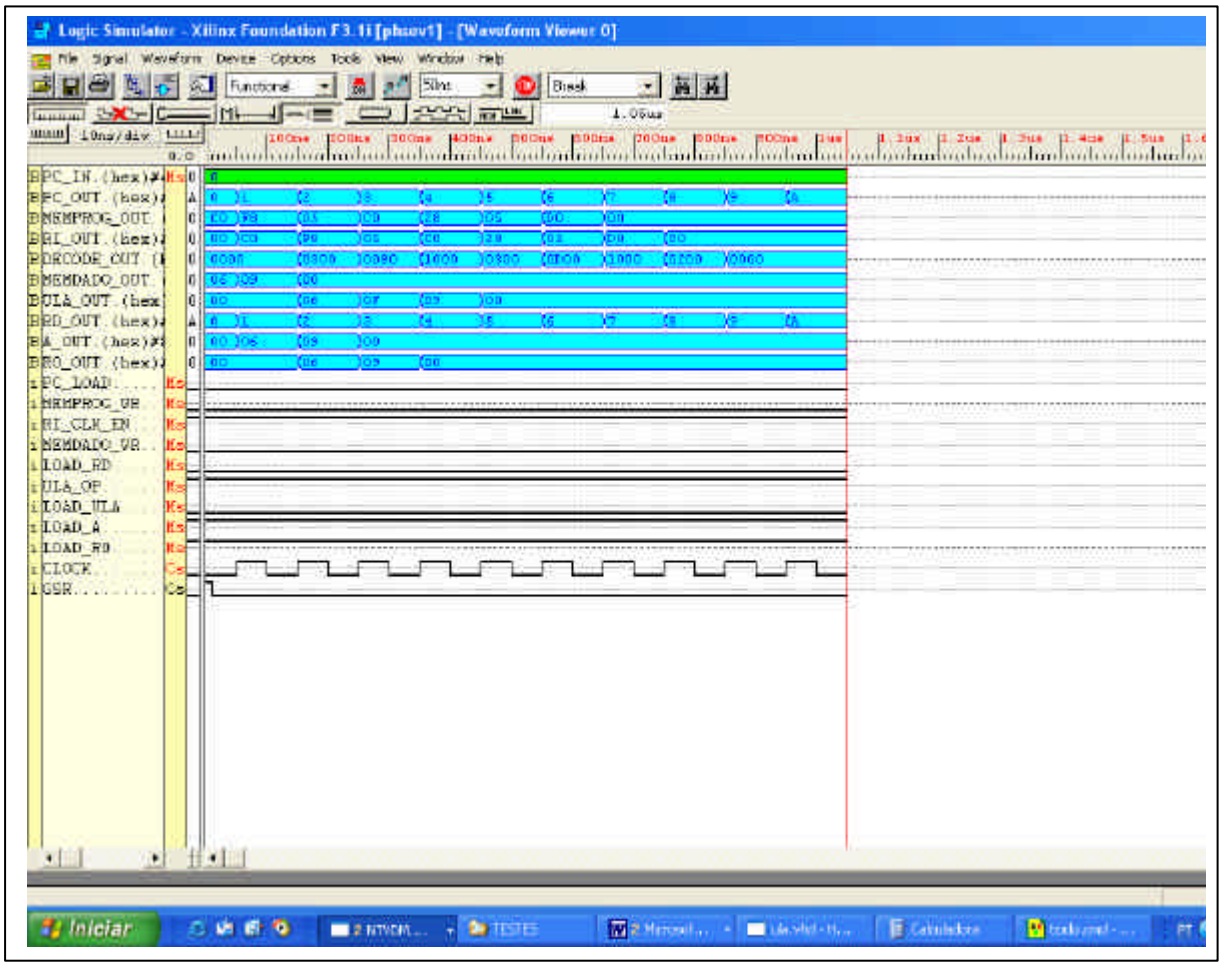

Figura 49: Primeira simulação do PAHSE.

Pode-se observar que no sinal ULA_OUT aos 100ns é posicionado o dado 08 da memória de dados de um dos lados da ULA. Em seguida aos 200ns, o dado 09 é posicionado e aos 500ns a soma é o resultado da saída da ULA.

A simulação demonstrou que todas as unidades de execução processaram suas entradas e forneceram todas as saídas, com o sinal de saída de uma unidade sendo o sinal de entrada de outra. O processador passou por todos os estágios do pipeline. Porém não houve um sincronismo e os sinais de cada unidade eram extraídos sem que seu sucessor tivesse processado o sinal anterior.

Com o resultado obtido nesta simulação do processador, foi realizada a introdução de um contador de clock que teria a função de atrasar a saída de uma unidade de execução até

que a próxima esteja liberada da instrução anterior, como mostra a Figura 50. O código VHDL completo que valida o pipeline do PAHSE está no Anexo desta dissertação.

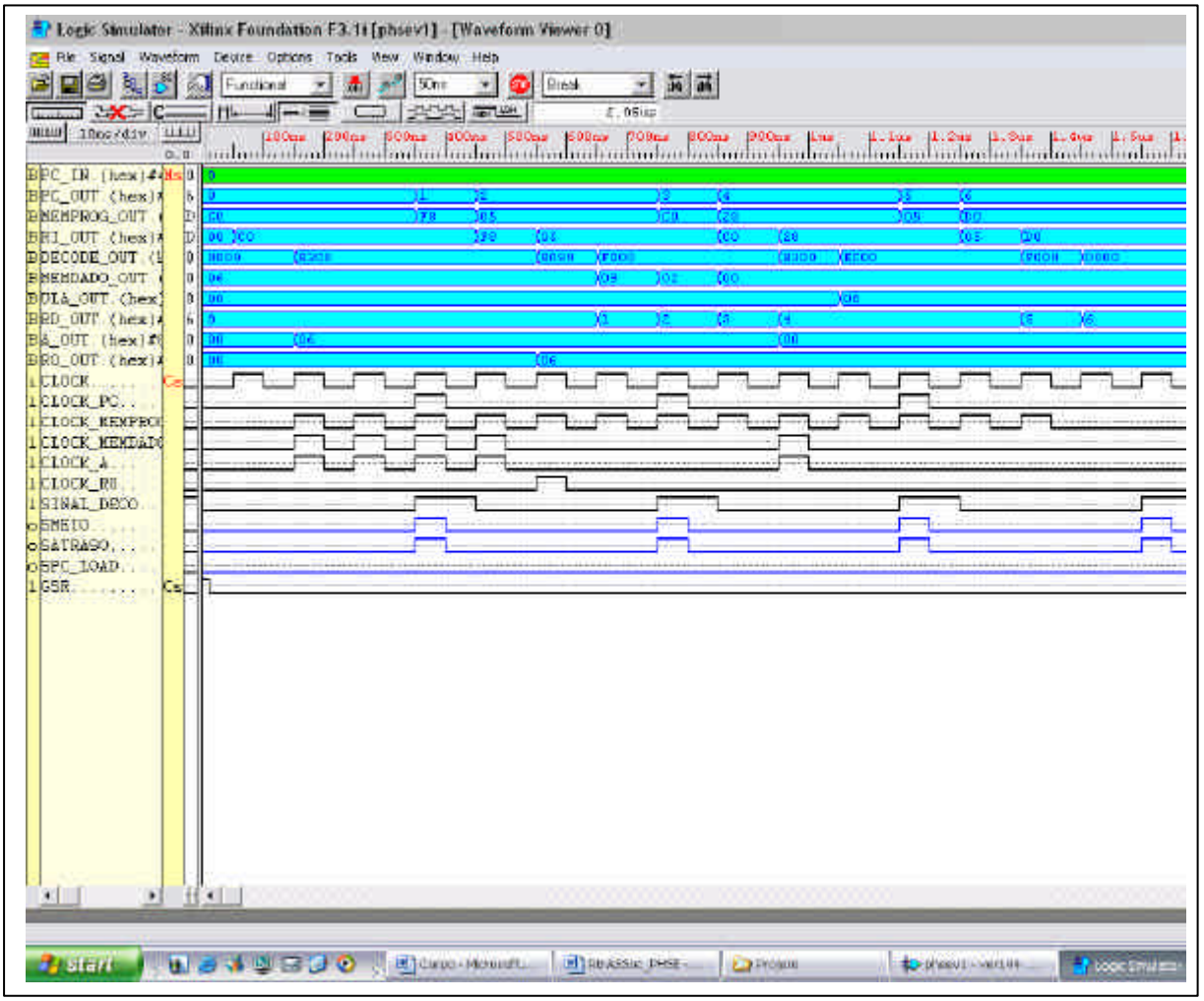

Figura 50: Validação parcial do PAHSE.

Agora, é possível observar que o carregamento do primeiro dado 06 no sinal ULA_OUT aconteceu em 1000ns, demonstrando que cada unidade aguarda a liberação da anterior.

Com o atraso de 4 clocks na execução de cada unidade de execução, verificou-se a validação parcial dos 4 estágios do pipeline proposto para o processador, busca, decodificação, execução e armazenamento.

A validação parcial do PAHSE mostrou uma instanciação completa das unidades de execução proposta como estrutura básica da CPU do PAHSE, e a simulação de um ciclo

completo de instrução com os quatro estágios do pipeline, através de um código em assembler alocado na memória de programa interna da CPU.

7. CONCLUSÃO

Conforme proposta inicial foi apresentada uma série de refêrencias sobre a validação das aplicações no RtrASSoc51, em especial o uso da estrutura do RtrASSoc51, baseado em microcontroladres da família 8051 para reconhecimento de objetos.

Também foram apresentadas considerações sobre o desenvolvimento e aplicações dos FPGAs em especial as FPGAs Virtex da Xilinx, cuja característica principal é a sua capacidade e a reconfiguração dinâmica.

Finalmente apresentou-se, o Processador com Arquitetura Harvard Superescalar Embutido (PAHSE), em particular, os módulos implementados para a realização do pipeline, consistindo de ciclos de busca, decodificação, execução e armazenamento. Nessa dissertação foram apresentados os módulos que compõem o PAHSE e os resultados da simulação, validando o conceito da execução do processador com um pipeline de 4 estágios.

Para o desenvolvimento dos módulos de execução, foi utilizado todo o suporte que a linguagem VHDL oferece para que o código fosse elaborado com a utilização de loops e chamadas de função. Os módulos escritos com esta estrutura não obtiveram sucesso e não foi possível obter simulação e tão pouco implementação.

Apesar da linguagem VHDL oferecer suporte todos os recursos de programação alto nível, a implementação do PAHSE com esta estrutura demonstrou que com a execução concorrente dos FPGAs, fica inviabilizada a utilização destes recursos da linguagem VHDL.

A partir deste fracasso, todos os códigos VHDL foram escritos como componentes com a ajuda da ferramenta LOGIBlox, parte integrante do Foundation Series da Xilinx. Os componentes foram instanciados, e com a execução concorrente do FPGA ocorreu a validação do PAHSE para a execução do pipeline de instruções com 4 estágios.

No relatório de implementação do PAHSE, utilizando as ferramentas da Xilinx em um FPGA XC4000EX, foram observados que a taxa de ocupação para IOBs foi de 47%, e de

CLBs foi de 17%. Constatou-se um tempo de CPU de 8s, e uma média de espera de conexão de 12,918ns.

Com estes valores de implementação, é possível que algum trabalho futuro, conclua o RtrASSoc51, composto pelo PAHSE (CARDIM,2005), o SOE (COSTA,2004), as Rotinas Reconfiguráveis (FORNARI,2004), e com isso, a validação completa dos tempos de reconhecimento de padrões (ZANGUETTIN,2004).

Também, como trabalhos futuros, podemos destacar a implementação completa dos três níveis de pipeline, e o gerenciamento dos problemas relacionados ao processo de execução de sistemas baseados em pipeline.

REFERÊNCIAS

8051 TUTORIAL, 2004 - http://www.8052.com/tut8051.phtml. Acessado em 20/01/2004.

ALTERA – *"System On Programmable Chip"* – San Jose, California, 2004

AGERWALA,T. COCKE,J. *High perfomance for operating systems and multiprogramming*. Boston. Kluwer Academic Publishers, 1989.

ANDERSON, D. SPARACIO, F. TOMASULO, F. *The IBM system / 360 model 91: machine philosophy and instruction handling*. IBM Journal of Reserch and Development. Janeiro, 1967.

ARNOLD, J.– *"The SPLASH 2 Software Evironment"*, IEEE Workshop on FPGAs for Custom Computing Machines, pp 88-93, April, 1993

ARNOLD, J. and D. Buell and E. Davis– *"SPLASH 2"*, Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, pp 316-324, 1992.

ARÓSTEGUE, J.M.M – *"FIPSoc – Sistema Chip Programable en Campo"* - Universitat Politècnica de Catalunya, Espanha, 2004.

ATHANAS, P. M. - *"An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*", Ph.D. thesis, Brown Univerity, USA, 1992

ATMEL – *"8 BIT Microcontroller with 32K bytes QuickFlash"* – San Jose, 2004.

BANNATYNE, Ross and Greg Viot – *"Introduction to Microcontrollers – Part 1 and 2"*, Embedded Systems Conference, San Francisco, 1998.

BITNER, R. and P. Athanas - *"WormHole Run-Time Reconfiguration"*, ACM/SIGDA International Symposium on Field Programmable Gate Array - FPGA'97, October 1997.

BONATO, V. – *"Projeto de um módulo de aquisição e pré-processamento de imagem colorida baseado em computação reconfigurável e aplicado a robôs móveis"* – Defesa e Dissertação Apresentada ao Programa de Pós-Graduação em Ciência da Computação do ICMC-USP para obtenção do título de Mestre, São Carlos, S.P.,2004.

BONDALAPATI, Kiran and Viktor K. Prasanna – *"Dynamic Precision Management for Loop Computations on Reconfigurable Architectures"*, IEEE Symposium on FPGAs for Custom Computing Machines, 1999.

CARDIM, M. *"RtrASSoc51-Adaptavél, Superescalar, Reconfigurável Programável Sistema em Chip- PHSE. Um Processador Superescalar baseado na arquitetura Harvard"*. Submetido ao WSCAD2005, Rio de Janeiro, 24 a 27 de outubro de 2005.

CARRILLO, Jorge E. and Paul Chow – *"The Effect of Reconfigurable Units in Superscalar Processors"*, ACM/SIGDA International Symposium on Field Programmable Gate Array – ,FPGA´2001, pp. 141-150, February, 2001.

CARRIÓN. D. *FPGA Field-Programmable Gate Array.* Universidade Federal do Rio de Janeiro. 2001.

CNSEC, - *"Embedded, Everywhere - A Research Agenda for Networked Systems of Embedded Computers"*- Committee on Networked Systems of Embedded Computers; Computer Science and Telecommunications Board; Division on Engineering and Physical Sciences; National Research Council; NATIONAL ACADEMY PRESS Washington, DC, 2001

COSTA, K. A. P., - *" RtrASSoc - Adaptável Superescalar Reconfigurável Sistema em Chip - O Sistema Operacional Embutido"* – Defesa de Dissertação Apresentada do Programa de Pós-Graduação em Ciência da Computação do UNIVEM para obtenção do título de Mestre, Marília, S.P., 2004

DANDALIS, Andreas and Viktor K. Prasana – *"Configuration Compression for FPGA-based Embedded Systems"*, ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2001, pp 173-182, February, 2001.

DEHON, A. - *"Reconfigurable Architecture for General-Purpose Computing"*, Ph.D. thesis, Massachusetts Institute of Technology, 1996.

DEHON, André; CASPI, Eylon; CHU, Michael; HUANG, Randy; YEH, Joseph; MARKOVSKY, Yury; WAWRZYNEK, John. *Stream computations organized for reconfigurable execution (SCORE): introduction and tutorial*. FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, 2000. Anais. . . 2000.

DOLPHIN – *"Flip 8051 – CORE 8051"* – France, 2003.

FORNARI, A. – *" RtrASSoc - Adaptável Superescalar Reconfigurável Sistemas em Chip Rotinas Reconfiguráveis no Modelo de Compressão Relocação e Defragmentação"* – Defesa de Dissertação Apresentada do Programa de Pós-Graduação em Ciência da Computação do UNIVEM para obtenção do título de Mestre, Marília, S.P., 2004

GONZALES, R.C., RICHARD E. W. - *Digital Image Processing*, 1992

GOKHALE, M and W. Holmes and A. Kopser and S. Lucas and R. Minnich and D. Sweely and D. Lopresti – *"Building and Using a Highly Parallel Programmable Logic Array"* – IEEE Computer, pp 81-89, Jan. 1991.

HADLEY, John D.; HUTCHINGS, Brad L. *Design methodologies for partially reconfigured systems.* IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1995, California, Estados Unidos. Anais. . . 1995. p.78–84.

HAUCK, S. - *"Reconfigurable Computing: A Survey of Systems and Software"***,** submitted *to* ACM Computing Surveys, *2000*.

HAUCK, S. - *"The Roles of FPGAs in Reprogrammable Systems"* **,** Proceedings of the IEEE, Vol. 86, No. 4, pp.615-638, April, 1998.

HAUCK, S., T. W. Fry, M. M. Hosler, J. P. Kao, *"The Chimaera Reconfigurable Functional Unit"* , IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87-96, 1997.

HAUSER, J. R. and J. Wawrzynek - *"Garp: A MIPS Processor with a Reconfigurable Coprocessor"*, Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM '97, USA, April 1997.

HENESSY, J.L. and D.A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publichers, Inc., San Francisco, CA, second edition, 1996.

HENNESSY, J. PATTERSON, D. *Computer Organization and design: the hardware/ software interface*. São Francisco. Morgan Kaufmann Publishers, 1994.

HUY Nuguyen and LIZY Kurian Johyn – "*Exploiting SIMD Parallelism in DSP and Multimídia Algorithms Using the Altivec Technology*", Rhodes, Greece, 1999.

HWANG, KAI – "Advanced Computer Architecture- Parallelism, Scalability and Programmability" – McGraw-Hill, 771pg, 1993

IEEE Computer Society – A survey of " *Configurable Computing: Technology and Applications"* – Computer, April, 2000.

IPCA. *Arquiteturas de Computadores.* Acessado em 01/12/2003. http://www.ipca.pt/prof/anabais/arq/slides/RISC.pdf. 2003.

ISELI, Christian – "*Spyder - A Reconfigurable Processor Development System"*, Thèse de Docteur, École Polytechnique Fédéraçe de Lausanne, Lausanne, 1996.

JAMES O. Hamblem AND MICHAEL D. Furman – *Rapid Prototyping of Digital Systems*, 2001

JOUPPI, N. HENNESSY, J. *Computer Technology and Architecture: An Evolving Interaction*. IEEE Computer Graphics and applications. Setembro, 1991.

LATHI, B.P.- Sistemas de Comunicação, 1968

LEE, S., et al –"*Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model"*- ACM/SIGDA – 10th International Workshop on Hardware/Software Codesign, PP 199-204, 2002.

LOPES, J. J. – "*RtrASSoc –Adaptável Superescalar Sistema em Chip – hw/sw codesign"* – Relatório FAPESP, Proc: 03/06913-6, 2004.

MASSIMO Baleani., et al – "*HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform"*. ACM/SIGDA - 10th International Workshop on Hardware/Software Codesign, pp 151-156, 2002.

MESQUITA. D. *Contribuições para reconfiguração parcial, remota e dinâmica de FPGAs*. Pontifícia Universidade Católica do Rio Grande do Sul. Porto Alegre. 2002.

MICROCHIP Microcontrolador PIC Disponível em http://www.microchip.com. Acessado em: 25/01/2004.

MOHAMED Shalan & Vincent J. Mooney – *" Hardware Support for Real-Time Embedded Multiprocessor System-on-Chip Memory Management"*. ACM/SIGDA - 10th International Workshop on Hardware/Software Codesign, pp 79-84, 2002.

OGÊ Marques Filho, HUGO Vieira Neto - Processamento Digital de Imagens, 1999

ORDONEZ, E. D. M. & Silva, J.L. – *" Reconfigurable Computing –Experiences and Perspectives"* , Fundação de Ensino Eurípides Soares da Rocha (FEESR), Marília, 294pg., Agosto, 2000.

PIACENTINO, Michael R., Gooitzen S. Van der Wal, Michael W. Hansen – *"Reconfigurable Elements for a Video Pipeline Processor"*, IEEE Symposium on FPGAs for Custom Computing Machines, pp 1-10, 1999.

PHILLIPS, S. & HAUCK, S. –*"Automatic Layout of Dpmain-Specific Reconfigurable Subsystem for System-on-Chip"* - 10th ACM International Symposium on Field Programmable Gate Array – FPGA'2002, pp 135-173,2002.

QUENOT, G., I. Kraljic, J. Serot and B. Zavidovique – *"A Reconfigurable Computing Engine for Real-Time Vision Automata Prototyping"*, IEEE Workshop on FPGAs ofr Custom Computing Machine, April, 1994.

RAJAMANI, S., P. Viswanath – *"A Quantitative Analysis of Processor – Programmable Logic Interface"*, IEEE Symposium on FPGA for Custom Computing Machines, 1996.

RATHA, N. and A. Jain and D. Rover, *"Convolution on SPLASH 2"*, IEEE Workshop on FPGA for Custom Computing Machine, pp 204-213, April, 1995

RAVI Bhagava, LIZY K.John, BRIAN L. Evans and RAMESH Radhakrishinan – *"Evaluating MMX Technology Using DSP and Multimidia Applications"*, Dallas, Texas, 1998.

RICARTE, I. *Organização de Computadores*. Universidade Estadual de Campinas - FEEC/UNICAMP, 1999.

COSTA, R.M. – *"Microcontroladores em computação reconfigurável (Run-Time Reconfiguration - RTR) - uma aplicação"*. Universidade Federal de São Carlos, SP, 2002.

SANCHEZ, Eduardo; SIPPER, Moshe; HAENNI, Jacques-Olivier; BEUCHAT, Jean-Luc; STAUFFER, André; PEREZ-URIBE, Andrés. *Static and dynamic configurable systems.* Nova Iorque, Estados Unidos: [s.n.], 1999. p.556–564.

SILVA, J.L., CASTRO, R. M., JORGE, G. H. R. – *"RtrASSoc – An Adaptable Superscalar Reconfigurable System On Chip – The Simulator"* In: The 3rd IEEE International Workshop on System-on-Chip for Real Time Applications, IWSOC' 2003, Calgary, Alberta, 2003.

SILVA, J.L., COSTA, K.A.P. – *"RtrASSoc – An Adaptable Superscalar Reconfigurable System On Chip – The Operational System Inlaid Storage and Vectorial Allocation."* In: The 4th IEEE International Workshop on System-on-Chip for Real Time Applications, IWSOC' 2003, Bawff, Alberta, 2004.

SHALAN, M. & Mooney, V.J. –*"Hardware Support for Real Time Embedded Multiprocessor System-on-Chip Memory Management"*. - ACM/SIGDA – 10th International Workshop on Hardware/SoftwaREcODESIGN, PP 79-84, 2002.
SKLAR, Bernard – *Digital Communications, 1988*

SILICON GRAPHICS. *MIPS Group*. Acessado em 01/12/2003. http://www.terravista.pt/PortoSanto/1926/MIPS.htm. 2003.

STALLINGS, William. *Arquitetura e Organização de Computadores: Projeto para o Desempenho*. São Paulo. Prentice Hall, 2002.

TANENBAUM, A. – Organização Estruturada de Computadores, 1990

TEIXEIRA. M.A. – *"Técnicas de Reconfigurabilidade dos FPGAs a família APEX 20K"* – ALTERA, USP – São Carlos, 2002

TEXAS INSTRUMENT– *TMS320C203 Digital Signal Processor*, 1998

TORRES, Gabriel. *Hardware Curso Completo.* 3a.ed. Rio de janeiro: Axcels Books, 1999.

VILLASENOR, J. and W. H. M. Smith - *"Configurable Computing"*, Scientific American, USA, June 1997.

WIRTHLIN, M. J. and B. L. Hutchings – *"A Dynamic Instruction Set Computer"*, proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1995.

XILINX, Inc. – *Virtex II Plataform FPGAs*, 2002

XILINX - *"Embedded Operating System"*- San Jose, California, 2004

YE, Zhi Alex, Nagaraj Shenoy, Prithviraj Banerjee – *"A C Compiler for a Processor with a Reconfigurable Functional Unit"*, ACM/SIGDA International Symposium on Field Progammable Gate Array – FPGA´2000, pp 95-100, 2000.

ZANGUETTIN, O. F. – *"RtrASSoc - Sistema em Chip Adaptável Superescalar Reconfigurável - Processador Superescalar Embutido"* – Defesa de Dissertação Apresentada ao Programa de Pós-Graduação em Ciência da Computação do UNIVEM para obtenção do título de Mestre, Marília, S.P., 2004

ANEXO A – Memória de Programa (MEMPROG)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MEMPROG is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO 0);
           DI: IN std_logic_vector(7 DOWNTO 0);
           WR_EN: IN std_logic;
           WR_CLK: IN std_logic
    );
end MEMPROG;

architecture MEMPROG_arch of MEMPROG is

-----------------------------------------------------
-- COMPONENTE DE MEMORIA DO LOGIBLOX
-----------------------------------------------------
component LBRAMPR
  PORT(
    A: IN std_logic_vector(3 DOWNTO 0);
    DO: OUT std_logic_vector(7 DOWNTO 0);
    DI: IN std_logic_vector(7 DOWNTO 0);
    WR_EN: IN std_logic;
    WR_CLK: IN std_logic);
end component;

begin

  instance_name : LBRAMPR port map
    (A => A,
     DO => DO,
     DI => DI,
     WR_EN => WR_EN,
     WR_CLK => WR_CLK);

end MEMPROG_arch;
```

ANEXO B – Memória de Dados (MEMDADO)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MEMDADO is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO 0);
           DI: IN std_logic_vector(7 DOWNTO 0);
           WR_EN: IN std_logic;
           WR_CLK: IN std_logic
    );
end MEMDADO;

architecture MEMDADO_arch of MEMDADO is

-----------------------------------------------------
-- COMPONENTE DE MEMORIA DO LOGIBLOX
-----------------------------------------------------
component LBRAMDA
  PORT(
    A: IN std_logic_vector(3 DOWNTO 0);
    DO: OUT std_logic_vector(7 DOWNTO 0);
    DI: IN std_logic_vector(7 DOWNTO 0);
    WR_EN: IN std_logic;
    WR_CLK: IN std_logic);
end component;

begin

  instance_name : LBRAMDA port map
    (A => A,
     DO => DO,
     DI => DI,
     WR_EN => WR_EN,
     WR_CLK => WR_CLK);

end MEMDADO_arch;
```

ANEXO C – Decodificador (DECODE)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity decode is
    port (
        D_IN: in STD_LOGIC_VECTOR (7 downto 0);
        CLOCK: in STD_LOGIC;
        D_OUT: out STD_LOGIC_VECTOR (15 downto 0)
    );
end decode;

architecture decode_arch of decode is
begin

 process (D_IN,CLOCK)
 begin
  if CLOCK'EVENT and CLOCK = '1' then
      case D_IN is
        when "11000000"  => D_OUT <= "1110001100000000";    -- PUSH address C0
        when "11111000"  => D_OUT <= "1110000010000000";    -- MOV R0,A    F8
       when "00000101"  => D_OUT <= "1111000000000000";     -- INC ADDRESS 05

       when "00101000"  => D_OUT <= "1110110000000000";     -- ADD A,R0    28

        when "11010000"  => D_OUT <= "0000000000000000";    -- POP  address D0



       when "00000000"  => D_OUT <= "0000000000000000";     -- NOP         00
       when "00000001"  => D_OUT <= "0000000000000000";     -- AJMP        01
       when "00000010"  => D_OUT <= "0000000000000000";     -- LJMP        02
       when "00000011"  => D_OUT <= "0000000000000000";     -- RR          03
       when "00000100"  => D_OUT <= "0000000000000000";     -- INC         04

       when "00000110"  => D_OUT <= "0000000000000000";     -- INC         06
       when "00000111"  => D_OUT <= "0000000000000000";     -- INC         07
       when "00001000"  => D_OUT <= "0000000000000000";     -- INC         08
       when "00001001"  => D_OUT <= "0000000000000000";     -- INC         09
       when "00001010"  => D_OUT <= "0000000000000000";     -- INC         0A
       when "00001011"  => D_OUT <= "0000000000000000";     -- INC         0B
       when "00001100"  => D_OUT <= "0000000000000000";     -- INC         0C
       when "00001101"  => D_OUT <= "0000000000000000";     -- INC         0D
       when "00001110"  => D_OUT <= "0000000000000000";     -- INC         0E
       when "00001111"  => D_OUT <= "0000000000000000";     -- INC         0F
       when "00010000"  => D_OUT <= "0000000000000000";     -- JBC         10
       when "00010001"  => D_OUT <= "0000000000000000";     -- ACALL       11
       when "00010010"  => D_OUT <= "0000000000000000";     -- LCALL       12
       when "00010011"  => D_OUT <= "0000000000000000";     -- RRC         13
       when "00010100"  => D_OUT <= "0000000000000000";     -- DEC         14
       when "00010101"  => D_OUT <= "0000000000000000";     -- DEC         15
       when "00010110"  => D_OUT <= "0000000000000000";     -- DEC         16
       when "00010111"  => D_OUT <= "0000000000000000";     -- DEC         17
       when "00011000"  => D_OUT <= "0000000000000000";     -- DEC         18
       when "00011001"  => D_OUT <= "0000000000000000";     -- DEC         19
       when "00011010"  => D_OUT <= "0000000000000000";     -- DEC         1A
       when "00011011"  => D_OUT <= "0000000000000000";     -- DEC         1B
       when "00011100"  => D_OUT <= "0000000000000000";     -- DEC         1C
       when "00011101"  => D_OUT <= "0000000000000000";     -- DEC         1D
       when "00011110"  => D_OUT <= "0000000000000000";     -- DEC         1E
       when "00011111"  => D_OUT <= "0000000000000000";     -- DEC         1F
       when "00100000"  => D_OUT <= "0000000000000000";     -- JB          20
       when "00100001"  => D_OUT <= "0000000000000000";     -- AJMP        21
       when "00100010"  => D_OUT <= "0000000000000000";     -- RET         22
       when "00100011"  => D_OUT <= "0000000000000000";     -- RL          23
       when "00100100"  => D_OUT <= "0000000000000000";     -- ADD         24
       when "00100101"  => D_OUT <= "0000000000000000";     -- ADD         25
       when "00100110"  => D_OUT <= "0000000000000000";     -- ADD         26
       when "00100111"  => D_OUT <= "0000000000000000";     -- ADD         27

       when "00101001"  => D_OUT <= "0000000000000000";     -- ADD         29
       when "00101010"  => D_OUT <= "0000000000000000";     -- ADD         2A
       when "00101011"  => D_OUT <= "0000000000000000";     -- ADD         2B
       when "00101100"  => D_OUT <= "0000000000000000";     -- ADD         2C
```

```
when "00101101"  => D_OUT <= "0000000000000000";     -- ADD         2D
when "00101110"  => D_OUT <= "0000000000000000";     -- ADD         2E
when "00101111"  => D_OUT <= "0000000000000000";     -- ADD         2F
when "00110000"  => D_OUT <= "0000000000000000";     -- JNB         30
when "00110001"  => D_OUT <= "0000000000000000";     -- ACALL       31
when "00110010"  => D_OUT <= "0000000000000000";     -- RETI        32
when "00110011"  => D_OUT <= "0000000000000000";     -- RLC         33
when "00110100"  => D_OUT <= "0000000000000000";     -- ADDC        34
when "00110101"  => D_OUT <= "0000000000000000";     -- ADDC        35
when "00110110"  => D_OUT <= "0000000000000000";     -- ADDC        36
when "00110111"  => D_OUT <= "0000000000000000";     -- ADDC        37
when "00111000"  => D_OUT <= "0000000000000000";     -- ADDC        38
when "00111001"  => D_OUT <= "0000000000000000";     -- ADDC        39
when "00111010"  => D_OUT <= "0000000000000000";     -- ADDC        3A
when "00111011"  => D_OUT <= "0000000000000000";     -- ADDC        3B
when "00111100"  => D_OUT <= "0000000000000000";     -- ADDC        3C
when "00111101"  => D_OUT <= "0000000000000000";     -- ADDC        3D
when "00111110"  => D_OUT <= "0000000000000000";     -- ADDC        3E
when "00111111"  => D_OUT <= "0000000000000000";     -- ADDC        3F
when "01000000"  => D_OUT <= "0000000000000000";     -- JC          40
when "01000001"  => D_OUT <= "0000000000000000";     -- AJMP        41
when "01000010"  => D_OUT <= "0000000000000000";     -- ORL         42
when "01000011"  => D_OUT <= "0000000000000000";     -- ORL         43
when "01000100"  => D_OUT <= "0000000000000000";     -- ORL         44
when "01000101"  => D_OUT <= "0000000000000000";     -- ORL         45
when "01000110"  => D_OUT <= "0000000000000000";     -- ORL         46
when "01000111"  => D_OUT <= "0000000000000000";     -- ORL         47
when "01001000"  => D_OUT <= "0000000000000000";     -- ORL         48
when "01001001"  => D_OUT <= "0000000000000000";     -- ORL         49
when "01001010"  => D_OUT <= "0000000000000000";          -- ORL         4A
when "01001011"  => D_OUT <= "0000000000000000";     -- ORL         4B
when "01001100"  => D_OUT <= "0000000000000000";          -- ORL         4C
when "01001101"  => D_OUT <= "0000000000000000";          -- ORL         4D
when "01001110"  => D_OUT <= "0000000000000000";          -- ORL         4E
when "01001111"  => D_OUT <= "0000000000000000";          -- ORL         4F
when "01010000"  => D_OUT <= "0000000000000000";          -- JNC         50
when "01010001"  => D_OUT <= "0000000000000000";          -- ACALL       51
when "01010010"  => D_OUT <= "0000000000000000";          -- ANL         52
when "01010011"  => D_OUT <= "0000000000000000";          -- ANL         53
when "01010100"  => D_OUT <= "0000000000000000";          -- ANL         54
when "01010101"  => D_OUT <= "0000000000000000";          -- ANL         55
when "01010110"  => D_OUT <= "0000000000000000";          -- ANL         56
when "01010111"  => D_OUT <= "0000000000000000";          -- ANL         57
when "01011000"  => D_OUT <= "0000000000000000";          -- ANL         58
when "01011001"  => D_OUT <= "0000000000000000";          -- ANL         59
when "01011010"  => D_OUT <= "0000000000000000";          -- ANL         5A
when "01011011"  => D_OUT <= "0000000000000000";          -- ANL         5B
when "01011100"  => D_OUT <= "0000000000000000";          -- ANL         5C
when "01011101"  => D_OUT <= "0000000000000000";          -- ANL         5D
when "01011110"  => D_OUT <= "0000000000000000";          -- ANL         5E
when "01011111"  => D_OUT <= "0000000000000000";          -- ANL         5F
when "01100000"  => D_OUT <= "0000000000000000";          -- JZ          60
when "01100001"  => D_OUT <= "0000000000000000";          -- AJMP        61
when "01100010"  => D_OUT <= "0000000000000000";          -- XRL         62
when "01100011"  => D_OUT <= "0000000000000000";          -- XRL         63
when "01100100"  => D_OUT <= "0000000000000000";          -- XRL         64
when "01100101"  => D_OUT <= "0000000000000000";          -- XRL         65
when "01100110"  => D_OUT <= "0000000000000000";     -- XRL         66
when "01100111"  => D_OUT <= "0000000000000000";     -- XRL         67
when "01101000"  => D_OUT <= "0000000000000000";     -- XRL         68
when "01101001"  => D_OUT <= "0000000000000000";     -- XRL         69
when "01101010"  => D_OUT <= "0000000000000000";     -- XRL         6A
when "01101011"  => D_OUT <= "0000000000000000";     -- XRL         6B
when "01101100"  => D_OUT <= "0000000000000000";     -- XRL         6C
when "01101101"  => D_OUT <= "0000000000000000";     -- XRL         6D
when "01101110"  => D_OUT <= "0000000000000000";     -- XRL         6E
when "01101111"  => D_OUT <= "0000000000000000";     -- XRL         6F
when "01110000"  => D_OUT <= "0000000000000000";     -- JNZ         70
when "01110001"  => D_OUT <= "0000000000000000";     -- ACALL       71
when "01110010"  => D_OUT <= "0000000000000000";     -- ORL         72
when "01110011"  => D_OUT <= "0000000000000000";     -- JMP         73
when "01110100"  => D_OUT <= "0000000000000000";     -- MOV         74
when "01110101"  => D_OUT <= "0000000000000000";     -- MOV         75
when "01110110"  => D_OUT <= "0000000000000000";     -- MOV         76
when "01110111"  => D_OUT <= "0000000000000000";     -- MOV         77
when "01111000"  => D_OUT <= "0000000000000000";     -- MOV         78
when "01111001"  => D_OUT <= "0000000000000000";     -- MOV         79
```

```
when "01111010"  => D_OUT <= "0000000000000000";    -- MOV         7A
when "01111011"  => D_OUT <= "0000000000000000";    -- MOV         7B
when "01111100"  => D_OUT <= "0000000000000000";    -- MOV         7C
when "01111101"  => D_OUT <= "0000000000000000";    -- MOV         7D
when "01111110"  => D_OUT <= "0000000000000000";    -- MOV         7E
when "01111111"  => D_OUT <= "0000000000000000";    -- MOV         7F
when "10000000"  => D_OUT <= "0000000000000000";    -- SJMP        80
when "10000001"  => D_OUT <= "0000000000000000";    -- AJMP        81
when "10000010"  => D_OUT <= "0000000000000000";    -- ANL         82
when "10000011"  => D_OUT <= "0000000000000000";    -- MOVC        83
when "10000100"  => D_OUT <= "0000000000000000";    -- DIV         84
when "10000101"  => D_OUT <= "0000000000000000";    -- MOV         85
when "10000110"  => D_OUT <= "0000000000000000";    -- MOV         86
when "10000111"  => D_OUT <= "0000000000000000";    -- MOV         87
when "10001000"  => D_OUT <= "0000000000000000";    -- MOV         88
when "10001001"  => D_OUT <= "0000000000000000";    -- MOV         89
when "10001010"  => D_OUT <= "0000000000000000";    -- MOV         8A
when "10001011"  => D_OUT <= "0000000000000000";    -- MOV         8B
when "10001100"  => D_OUT <= "0000000000000000";    -- MOV         8C
when "10001101"  => D_OUT <= "0000000000000000";    -- MOV         8D
when "10001110"  => D_OUT <= "0000000000000000";    -- MOV         8E
when "10001111"  => D_OUT <= "0000000000000000";    -- MOV         8F
when "10010000"  => D_OUT <= "0000000000000000";    -- MOV         90
when "10010001"  => D_OUT <= "0000000000000000";    -- ACALL       91
when "10010010"  => D_OUT <= "0000000000000000";    -- MOV         92
when "10010011"  => D_OUT <= "0000000000000000";    -- MOVC        93
when "10010100"  => D_OUT <= "0000000000000000";    -- SUBB        94
when "10010101"  => D_OUT <= "0000000000000000";    -- SUBB        95
when "10010110"  => D_OUT <= "0000000000000000";    -- SUBB        96
when "10010111"  => D_OUT <= "0000000000000000";    -- SUBB        97
when "10011000"  => D_OUT <= "0000000000000000";    -- SUBB        98
when "10011001"  => D_OUT <= "0000000000000000";    -- SUBB        99
when "10011010"  => D_OUT <= "0000000000000000";    -- SUBB        9A
when "10011011"  => D_OUT <= "0000000000000000";    -- SUBB        9B
when "10011100"  => D_OUT <= "0000000000000000";    -- SUBB        9C
when "10011101"  => D_OUT <= "0000000000000000";    -- SUBB        9D
when "10011110"  => D_OUT <= "0000000000000000";    -- SUBB        9E
when "10011111"  => D_OUT <= "0000000000000000";    -- SUBB        9F
when "10100000"  => D_OUT <= "0000000000000000";    -- ORL         A0
when "10100001"  => D_OUT <= "0000000000000000";    -- AJMP        A1
when "10100010"  => D_OUT <= "0000000000000000";    -- MOV         A2
when "10100011"  => D_OUT <= "0000000000000000";    -- INC         A3
when "10100100"  => D_OUT <= "0000000000000000";    -- MUL         A4
when "10100101"  => D_OUT <= "0000000000000000";    -- reserved    A5
when "10100110"  => D_OUT <= "0000000000000000";    -- MOV         A6
when "10100111"  => D_OUT <= "0000000000000000";    -- MOV         A7
when "10101000"  => D_OUT <= "0000000000000000";    -- MOV         A8
when "10101001"  => D_OUT <= "0000000000000000";    -- MOV         A9
when "10101010"  => D_OUT <= "0000000000000000";    -- MOV         AA
when "10101011"  => D_OUT <= "0000000000000000";    -- MOV         AB
when "10101100"  => D_OUT <= "0000000000000000";    -- MOV         AC
when "10101101"  => D_OUT <= "0000000000000000";    -- MOV         AD
when "10101110"  => D_OUT <= "0000000000000000";    -- MOV         AE
when "10101111"  => D_OUT <= "0000000000000000";    -- MOV         AF
when "10110000"  => D_OUT <= "0000000000000000";    -- ANL         B0
when "10110001"  => D_OUT <= "0000000000000000";    -- ACALL       B1
when "10110010"  => D_OUT <= "0000000000000000";    -- CPL         B2
when "10110011"  => D_OUT <= "0000000000000000";    -- CPL         B3
when "10110100"  => D_OUT <= "0000000000000000";    -- CJNE        B4
when "10110101"  => D_OUT <= "0000000000000000";    -- CJNE        B5
when "10110110"  => D_OUT <= "0000000000000000";    -- CJNE        B6
when "10110111"  => D_OUT <= "0000000000000000";    -- CJNE        B7
when "10111000"  => D_OUT <= "0000000000000000";    -- CJNE        B8
when "10111001"  => D_OUT <= "0000000000000000";    -- CJNE        B9
when "10111010"  => D_OUT <= "0000000000000000";    -- CJNE        BA
when "10111011"  => D_OUT <= "0000000000000000";    -- CJNE        BB
when "10111100"  => D_OUT <= "0000000000000000";    -- CJNE        BC
when "10111101"  => D_OUT <= "0000000000000000";    -- CJNE        BD
when "10111110"  => D_OUT <= "0000000000000000";    -- CJNE        BE
when "10111111"  => D_OUT <= "0000000000000000";    -- CJNE        BF

when "11000001"  => D_OUT <= "0000000000000000";    -- AJMP        C1
when "11000010"  => D_OUT <= "0000000000000000";    -- CLR         C2
when "11000011"  => D_OUT <= "0000000000000000";    -- CLR         C3
when "11000100"  => D_OUT <= "0000000000000000";    -- SWAP        C4
when "11000101"  => D_OUT <= "0000000000000000";    -- XCH         C5
when "11000110"  => D_OUT <= "0000000000000000";    -- XCH         C6
```

```
        when "11000111"  => D_OUT <= "0000000000000000";    -- XCH         C7
        when "11001000"  => D_OUT <= "0000000000000000";    -- XCH         C8
        when "11001001"  => D_OUT <= "0000000000000000";    -- XCH         C9
        when "11001010"  => D_OUT <= "0000000000000000";    -- XCH         CA
        when "11001011"  => D_OUT <= "0000000000000000";    -- XCH         CB
        when "11001100"  => D_OUT <= "0000000000000000";    -- XCH         CC
        when "11001101"  => D_OUT <= "0000000000000000";    -- XCH         CD
        when "11001110"  => D_OUT <= "0000000000000000";    -- XCH         CE
        when "11001111"  => D_OUT <= "0000000000000000";    -- XCH         CF

        when "11010001"  => D_OUT <= "0000000000000000";    -- ACALL       D1
        when "11010010"  => D_OUT <= "0000000000000000";    -- SETB        D2
        when "11010011"  => D_OUT <= "0000000000000000";    -- SETB        D3
        when "11010100"  => D_OUT <= "0000000000000000";    -- DA          D4
        when "11010101"  => D_OUT <= "0000000000000000";    -- DJNZ        D5
        when "11010110"  => D_OUT <= "0000000000000000";    -- XCHD        D6
        when "11010111"  => D_OUT <= "0000000000000000";    -- XCHD        D7
        when "11011000"  => D_OUT <= "0000000000000000";    -- DJNZ        D8
        when "11011001"  => D_OUT <= "0000000000000000";    -- DJNZ        D9
        when "11011010"  => D_OUT <= "0000000000000000";    -- DJNZ        DA
        when "11011011"  => D_OUT <= "0000000000000000";    -- DJNZ        DB
        when "11011100"  => D_OUT <= "0000000000000000";    -- DJNZ        DC
        when "11011101"  => D_OUT <= "0000000000000000";    -- DJNZ        DD
        when "11011110"  => D_OUT <= "0000000000000000";    -- DJNZ        DE
        when "11011111"  => D_OUT <= "0000000000000000";    -- DJNZ        DF
        when "11100000"  => D_OUT <= "0000000000000000";    -- MOVX        E0
        when "11100001"  => D_OUT <= "0000000000000000";    -- AJMP        E1
        when "11100010"  => D_OUT <= "0000000000000000";    -- MOVX        E2
        when "11100011"  => D_OUT <= "0000000000000000";    -- MOVX        E3
        when "11100100"  => D_OUT <= "0000000000000000";    -- CLR         E4
        when "11100101"  => D_OUT <= "0000000000000000";    -- MOV         E5
        when "11100110"  => D_OUT <= "0000000000000000";    -- MOV         E6
        when "11100111"  => D_OUT <= "0000000000000000";    -- MOV         E7
        when "11101000"  => D_OUT <= "0000000000000000";    -- MOV         E8
        when "11101001"  => D_OUT <= "0000000000000000";    -- MOV         E9
        when "11101010"  => D_OUT <= "0000000000000000";    -- MOV         EA
        when "11101011"  => D_OUT <= "0000000000000000";    -- MOV         EB
        when "11101100"  => D_OUT <= "0000000000000000";    -- MOV         EC
        when "11101101"  => D_OUT <= "0000000000000000";    -- MOV         ED
        when "11101110"  => D_OUT <= "0000000000000000";    -- MOV         EE
        when "11101111"  => D_OUT <= "0000000000000000";    -- MOV         EF
        when "11110000"  => D_OUT <= "0000000000000000";    -- MOVX        F0
        when "11110001"  => D_OUT <= "0000000000000000";    -- ACALL       F1
        when "11110010"  => D_OUT <= "0000000000000000";    -- MOVX        F2
        when "11110011"  => D_OUT <= "0000000000000000";    -- MOVX        F3
        when "11110100"  => D_OUT <= "0000000000000000";    -- CPL         F4
        when "11110101"  => D_OUT <= "0000000000000000";    -- MOV         F5
        when "11110110"  => D_OUT <= "0000000000000000";    -- MOV         F6
        when "11110111"  => D_OUT <= "0000000000000000";    -- MOV         F7

        when "11111001"  => D_OUT <= "0000000000000000";    -- MOV         F9
        when "11111010"  => D_OUT <= "0000000000000000";    -- MOV         FA
        when "11111011"  => D_OUT <= "0000000000000000";    -- MOV         FB
        when "11111100"  => D_OUT <= "0000000000000000";    -- MOV         FC
        when "11111101"  => D_OUT <= "0000000000000000";    -- MOV         FD
        when "11111110"  => D_OUT <= "0000000000000000";    -- MOV         FE
        when "11111111"  => D_OUT <= "0000000000000000";    -- MOV         FF
       when others =>      D_OUT <= "ZZZZZZZZZZZZZZZZ";
      end case;
  end if;
 end process;

end decode_arch;
```

ANEXO D – Unidade Lógico Aritmética (ULA)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ULA is
    port (
           ADD_SUB: IN std_logic;
           A: IN std_logic_vector(7 DOWNTO 0);
           B: IN std_logic_vector(7 DOWNTO 0);
           LOAD: IN std_logic;                              -- 0 -> A | 1 -> A +/- B
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end ULA;

architecture ULA_arch of ULA is

component lbregula
  PORT(
    ADD_SUB: IN std_logic;
    A: IN std_logic_vector(7 DOWNTO 0);
    B: IN std_logic_vector(7 DOWNTO 0);
    LOAD: IN std_logic;
    CLOCK: IN std_logic;
    Q_OUT: OUT std_logic_vector(7 DOWNTO 0));
end component;

begin

  instance_name : lbregula port map
    (ADD_SUB => ADD_SUB,
     A => A,
     B => B,
     LOAD => LOAD,
     CLOCK => CLOCK,
     Q_OUT => Q_OUT);

end ULA_arch;
```

ANEXO E – Contador de Programas (PC)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity PC is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO 0);
           LOAD: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3 DOWNTO 0)
         );
end PC;

architecture PC_arch of PC is

---------------------------------------------------
-- Component Declaration
---------------------------------------------------
component LBCON4
  PORT(
    D_IN: IN std_logic_vector(3 DOWNTO 0);
    LOAD: IN std_logic;
    CLOCK: IN std_logic;
    Q_OUT: OUT std_logic_vector(3 DOWNTO 0));
end component;

begin
  instance_name : LBCON4 port map
    (D_IN => D_IN,
     LOAD => LOAD,
     CLOCK => CLOCK,
     Q_OUT => Q_OUT);
end PC_arch;
```

ANEXO F – Registrador de Instruções (RI)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RI is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end RI;

architecture RI_arch of RI is

component LBREN8
  PORT(
    D_IN: IN std_logic_vector(7 DOWNTO 0);
    CLK_EN: IN std_logic;
    CLOCK: IN std_logic;
    Q_OUT: OUT std_logic_vector(7 DOWNTO 0));
end component;

begin

  instance_name : LBREN8 port map
    (D_IN => D_IN,
     CLK_EN => CLK_EN,
     CLOCK => CLOCK,
     Q_OUT => Q_OUT);

end RI_arch;
```

ANEXO G – Registrador de Dados (RD)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RD is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO 0);
           LOAD: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3 DOWNTO 0)
    );
end RD;

architecture RD_arch of RD is

component LBCON4
  PORT(
    D_IN: IN std_logic_vector(3 DOWNTO 0);
    LOAD: IN std_logic;
    CLOCK: IN std_logic;
    Q_OUT: OUT std_logic_vector(3 DOWNTO 0));
end component;

begin
  instance_name : LBCON4 port map
    (D_IN => D_IN,
     LOAD => LOAD,
     CLOCK => CLOCK,
     Q_OUT => Q_OUT);

end RD_arch;
```

ANEXO H – Registrador A

```
library IEEE;
use IEEE.std_logic_1164.all;

entity A is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end A;

architecture A_arch of A is

component LBREN8
  PORT(
    D_IN: IN std_logic_vector(7 DOWNTO 0);
    CLK_EN: IN std_logic;
    CLOCK: IN std_logic;
    Q_OUT: OUT std_logic_vector(7 DOWNTO 0));
end component;

begin

  instance_name : LBREN8 port map
    (D_IN => D_IN,
     CLK_EN => CLK_EN,
     CLOCK => CLOCK,
     Q_OUT => Q_OUT);

end A_arch;
```

ANEXO I – Registrador R0

```
library IEEE;
use IEEE.std_logic_1164.all;

entity R0 is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end R0;

architecture R0_arch of R0 is

component LBREN8
  PORT(
    D_IN: IN std_logic_vector(7 DOWNTO 0);
    CLK_EN: IN std_logic;
    CLOCK: IN std_logic;
    Q_OUT: OUT std_logic_vector(7 DOWNTO 0));
end component;

begin

  instance_name : LBREN8 port map
    (D_IN => D_IN,
     CLK_EN => CLK_EN,
     CLOCK => CLOCK,
     Q_OUT => Q_OUT);

end R0_arch;
```

ANEXO J – Teste de Busca de Instruções

```
library IEEE;
use IEEE.std_logic_1164.all;

entity teste_busca is
    port (
            CLOCK: in STD_LOGIC;
            PC_IN: in STD_LOGIC_VECTOR(3 DOWNTO 0);
            PC_LOAD: in STD_LOGIC;
            MEMPROG_WR: in STD_LOGIC;
            PC_OUT: out STD_LOGIC_VECTOR(3 DOWNTO 0);
            MEMPROG_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
end teste_busca;

architecture teste_busca_arch of teste_busca is

component MEMPROG is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
            DO: OUT std_logic_vector(7 DOWNTO 0);
            DI: IN std_logic_vector(7 DOWNTO 0);
            WR_EN: IN std_logic;
            WR_CLK: IN std_logic
    );
end component;

component PC is
    port (
            D_IN: IN std_logic_vector(3 DOWNTO 0);
            LOAD: IN std_logic;
            CLOCK: IN std_logic;
            Q_OUT: OUT std_logic_vector(3 DOWNTO 0)
          );
end component;

signal MEMPROG_IN : std_logic_vector(7 DOWNTO 0);
signal PC_OUT_SINAL : std_logic_vector(3 DOWNTO 0);


begin

  I02 : PC      port map
        ( D_IN => PC_IN,
          LOAD => PC_LOAD,
          CLOCK => CLOCK,
          Q_OUT => PC_OUT_SINAL
        );

  PC_OUT <= PC_OUT_SINAL;

  I01 : MEMPROG port map
        ( A => PC_OUT_SINAL,
          DO => MEMPROG_OUT,
          DI => MEMPROG_IN,
          WR_EN => MEMPROG_WR,
          WR_CLK => CLOCK
        );

end teste_busca_arch;
```

ANEXO L – Teste de Busca e Decodificação de Instrução

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tBI_DE is
    port (
           CLOCK: in STD_LOGIC;
           PC_IN: in STD_LOGIC_VECTOR(3 DOWNTO 0);
           PC_LOAD: in STD_LOGIC;
           MEMPROG_WR: in STD_LOGIC;
           RI_CLK_EN: in STD_LOGIC;
           PC_OUT: out STD_LOGIC_VECTOR(3 DOWNTO 0);
           MEMPROG_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           RI_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           DECODE_OUT:  out STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
end tBI_DE;

architecture tBI_DE_arch of tBI_DE is

component MEMPROG is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO 0);
           DI: IN std_logic_vector(7 DOWNTO 0);
           WR_EN: IN std_logic;
           WR_CLK: IN std_logic
    );
end component;

component PC is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO 0);
           LOAD: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3 DOWNTO 0)
         );
end component;

component RI is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end component;

component decode is
    port (
        D_IN: in STD_LOGIC_VECTOR (7 downto 0);
        CLOCK: in STD_LOGIC;
        D_OUT: out STD_LOGIC_VECTOR (15 downto 0)
    );
end component;

signal MEMPROG_IN : std_logic_vector(7 DOWNTO 0);
signal PC_OUT_SINAL : std_logic_vector(3 DOWNTO 0);
signal MEMPROG_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal RI_OUT_SINAL : std_logic_vector(7 DOWNTO 0);


begin

  I01 : PC      port map
       ( D_IN => PC_IN,
         LOAD => PC_LOAD,
         CLOCK => CLOCK,
         Q_OUT => PC_OUT_SINAL
       );

  PC_OUT <= PC_OUT_SINAL;

  I02 : MEMPROG port map
```

```
        ( A => PC_OUT_SINAL,
          DO => MEMPROG_OUT_SINAL,
          DI => MEMPROG_IN,
          WR_EN => MEMPROG_WR,
          WR_CLK => CLOCK
        );

  MEMPROG_OUT <= MEMPROG_OUT_SINAL;

  I03 : RI      port map
        ( D_IN => MEMPROG_OUT_SINAL,
          CLK_EN => RI_CLK_EN,
          CLOCK => CLOCK,
          Q_OUT => RI_OUT_SINAL
        );

  RI_OUT <= RI_OUT_SINAL;

  I04 : DECODE  port map
        ( D_IN => RI_OUT_SINAL,
          CLOCK => CLOCK,
          D_OUT => DECODE_OUT
        );


end tBI_DE_arch;
```

ANEXO M – Simulação do PAHSE – Busca, Decodificação, Execução e Armazenamento.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tciclo4 is
    port (
           CLOCK: in STD_LOGIC;
           PC_IN: in STD_LOGIC_VECTOR(3 DOWNTO 0);

           PC_OUT: out STD_LOGIC_VECTOR(3 DOWNTO 0);
           MEMPROG_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           RI_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           DECODE_OUT:  out STD_LOGIC_VECTOR(15 DOWNTO 0);
           MEMDADO_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           ULA_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           RD_OUT: out STD_LOGIC_VECTOR(3 DOWNTO 0);
           A_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           R0_OUT: out STD_LOGIC_VECTOR(7 DOWNTO 0);
           SATRASO: out STD_LOGIC;
           SMEIO: out STD_LOGIC;
           SPC_LOAD: out STD_LOGIC
    );
end tciclo4;

architecture tciclo4_arch of tciclo4 is

component MEMPROG is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO 0);
           DI: IN std_logic_vector(7 DOWNTO 0);
           WR_EN: IN std_logic;
           WR_CLK: IN std_logic
    );
end component;

component PC is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO 0);
           LOAD: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3 DOWNTO 0)
         );
end component;

component RI is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end component;

component decode is
    port (
           D_IN: in STD_LOGIC_VECTOR (7 downto 0);
           CLOCK: in STD_LOGIC;
           D_OUT: out STD_LOGIC_VECTOR (15 downto 0)
         );
end component;

component MEMDADO is
    port ( A: IN std_logic_vector(3 DOWNTO 0);
           DO: OUT std_logic_vector(7 DOWNTO 0);
           DI: IN std_logic_vector(7 DOWNTO 0);
           WR_EN: IN std_logic;                      -- 0 -> Leitura | 1 -> Grava
           WR_CLK: IN std_logic
    );
end component;

component ULA is
    port (
           ADD_SUB: IN std_logic;
```

```
           A: IN std_logic_vector(7 DOWNTO 0);
           B: IN std_logic_vector(7 DOWNTO 0);
           LOAD: IN std_logic;                                -- 0 -> A | 1 -> A +/- B
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end component;

component RD is
    port (
           D_IN: IN std_logic_vector(3 DOWNTO 0);
           LOAD: IN std_logic;                                -- 0 -> Contador | 1 -> D_IN
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(3 DOWNTO 0)
    );
end component;

component A is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end component;

component R0 is
    port (
           D_IN: IN std_logic_vector(7 DOWNTO 0);
           CLK_EN: IN std_logic;
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0)
    );
end component;

component CONTA_CLOCK is
    port (
           CLOCK: IN std_logic;
           Q_OUT: OUT std_logic_vector(7 DOWNTO 0);
           START_DECO: OUT std_logic
     );
end component;

component MEIO_CLOCK is
port (
        CLOCK: in STD_LOGIC;
        SINAL_DECO: in STD_LOGIC;
        MEIOCLOCK: out STD_LOGIC
    );
end component;

signal MEMPROG_IN : std_logic_vector(7 DOWNTO 0);
signal PC_OUT_SINAL : std_logic_vector(3 DOWNTO 0);
signal MEMPROG_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal RI_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal DECODE_OUT_SINAL : std_logic_vector(15 DOWNTO 0);
signal MEMDADO_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal ULA_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal RD_OUT_SINAL : std_logic_vector(3 DOWNTO 0);
signal RD_IN : std_logic_vector(3 DOWNTO 0);
signal A_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal R0_OUT_SINAL : std_logic_vector(7 DOWNTO 0);
signal CLOCK_MEMPROG : std_logic;
signal CLOCK_PC : std_logic;
signal CLOCK_RI : std_logic;
signal CLOCK_RD : std_logic;
signal CLOCK_MEMDADO : std_logic;
signal CLOCK_ULA : std_logic;
signal CLOCK_A : std_logic;
signal CLOCK_R0 : std_logic;
signal CLOCK_DECODE : std_logic;
signal PC_LOAD : std_logic;
signal MEMPROG_WR : std_logic;
signal MEMDADO_WR : std_logic;
signal RD_LOAD : std_logic;
signal ULA_LOAD : std_logic;
signal LOAD_RI  : std_logic;
```

```
signal CONTA_OUT : std_logic_vector(7 DOWNTO 0);
signal SINAL_DECO  : std_logic;
signal SINAL_MEIOCLOCK  : std_logic;
signal SINAL_ATRASO  : std_logic;

begin

  CLOCK_PC <= CLOCK AND DECODE_OUT_SINAL(14) AND SINAL_ATRASO;
  CLOCK_DECODE <= CLOCK;
  CLOCK_MEMPROG <= CLOCK AND DECODE_OUT_SINAL(15);
  CLOCK_RI <= CLOCK AND DECODE_OUT_SINAL(13);
  CLOCK_MEMDADO <= CLOCK AND DECODE_OUT_SINAL(9);
  CLOCK_ULA <= CLOCK AND DECODE_OUT_SINAL(11);
  CLOCK_RD <= CLOCK AND DECODE_OUT_SINAL(12);
  CLOCK_A <= CLOCK AND DECODE_OUT_SINAL(8);
  CLOCK_R0 <= CLOCK AND DECODE_OUT_SINAL(7);

  PC_LOAD <= '0';
  MEMPROG_WR <= NOT DECODE_OUT_SINAL(15);
  MEMDADO_WR <= NOT DECODE_OUT_SINAL(9);
  RD_LOAD <='0';
  ULA_LOAD <= NOT DECODE_OUT_SINAL(11);

  LOAD_RI <= '1';

  SINAL_ATRASO <= SINAL_MEIOCLOCK;
  SATRASO <= SINAL_ATRASO;
  SMEIO <= SINAL_MEIOCLOCK;
  SPC_LOAD <= PC_LOAD;

  I001 : MEIO_CLOCK port map
         ( CLOCK => CLOCK,
           SINAL_DECO => SINAL_DECO,
           MEIOCLOCK => SINAL_MEIOCLOCK
      );

  I00 : CONTA_CLOCK   port map
     ( CLOCK => CLOCK,
       Q_OUT => CONTA_OUT,
       START_DECO => SINAL_DECO
     );


  I01 : PC      port map
     ( D_IN => PC_IN,
       LOAD => PC_LOAD,
       CLOCK => CLOCK_PC,
       Q_OUT => PC_OUT_SINAL
     );

  PC_OUT <= PC_OUT_SINAL;

  I02 : MEMPROG port map
        ( A => PC_OUT_SINAL,
          DO => MEMPROG_OUT_SINAL,
          DI => MEMPROG_IN,
          WR_EN => MEMPROG_WR,
          WR_CLK => CLOCK_MEMPROG
        );

  MEMPROG_OUT <= MEMPROG_OUT_SINAL;

  I03 : RI      port map
        ( D_IN => MEMPROG_OUT_SINAL,
          CLK_EN => LOAD_RI,
          CLOCK => CLOCK,
          Q_OUT => RI_OUT_SINAL
        );

  RI_OUT <= RI_OUT_SINAL;

  I04 : DECODE  port map
        ( D_IN => RI_OUT_SINAL,
          CLOCK => CLOCK_DECODE,
          D_OUT => DECODE_OUT_SINAL
        );
```

```
  DECODE_OUT <= DECODE_OUT_SINAL;

  I05 : MEMDADO port map
       ( A => RD_OUT_SINAL,
         DO => MEMDADO_OUT_SINAL,
         DI => A_OUT_SINAL,
         WR_EN => MEMDADO_WR,
         WR_CLK => CLOCK_MEMDADO
       );

  MEMDADO_OUT <= MEMDADO_OUT_SINAL;

  I06 : ULA     port map
       ( ADD_SUB => DECODE_OUT_SINAL(10),
         A => A_OUT_SINAL,
         B => R0_OUT_SINAL,
         LOAD => ULA_LOAD,
         CLOCK => CLOCK_ULA,
         Q_OUT => ULA_OUT_SINAL
       );

  ULA_OUT <= ULA_OUT_SINAL;

  I07 : RD      port map
       ( D_IN => RD_IN,
         LOAD => RD_LOAD,
         CLOCK => CLOCK_RD,
         Q_OUT => RD_OUT_SINAL
       );

  RD_OUT <= RD_OUT_SINAL;

  I08 : A       port map
       ( D_IN => MEMDADO_OUT_SINAL,
         CLK_EN => DECODE_OUT_SINAL(8),
         CLOCK => CLOCK_A,
         Q_OUT => A_OUT_SINAL
       );

  A_OUT <= A_OUT_SINAL;

  I09 : R0      port map
       ( D_IN => A_OUT_SINAL,
         CLK_EN => DECODE_OUT_SINAL(7),
         CLOCK => CLOCK_R0,
         Q_OUT => R0_OUT
       );

  R0_OUT <= R0_OUT_SINAL;

end tciclo4_arch;
```