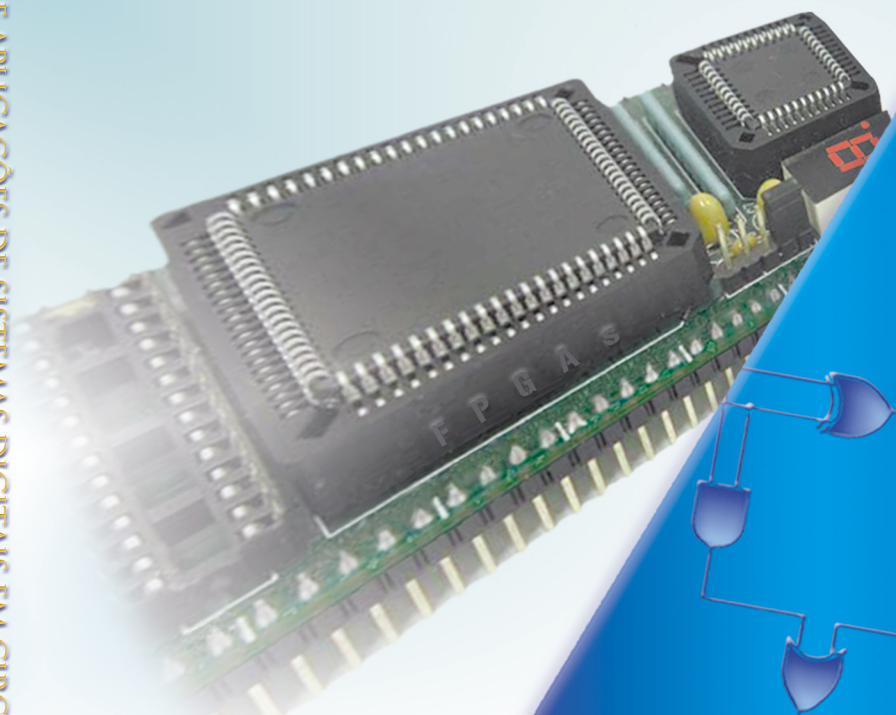
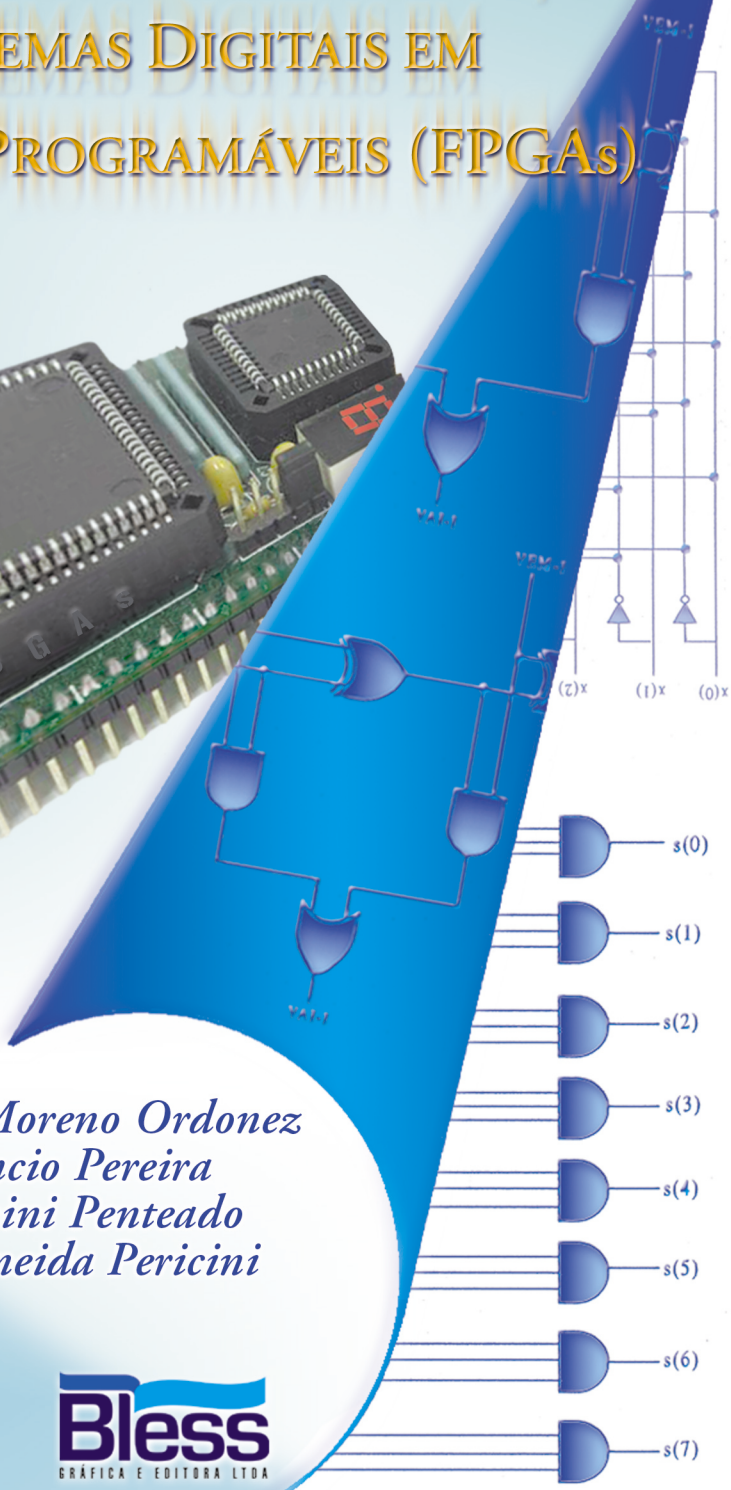
PROJETO, DESEMPENHO E APLICAÇÕES DE SISTEMAS DIGITAIS EM CIRCUITOS PROGRAMÁVEIS (FPGAs)

PROJETO, DESEMPENHO E APLICAÇÕES DE SISTEMAS DIGITAIS EM CIRCUITOS PROGRAMÁVEIS (FPGAs)






```
Library ieee;
use ieee.std_logic_1164.all;

entity ucp is
   port (clk  : in std_logic;
         rw   : out std
         adr  : out st
end ucp;

architecture ucp_arch of ucp is
begin
process (clk)
  if rst='0' then
     state <= reset;
  elsif clk'event and clk='1' then
        case state is
          when reset =>
          when busca =>
          when executa =>
          when others => null;
       end case;
  end if;
```

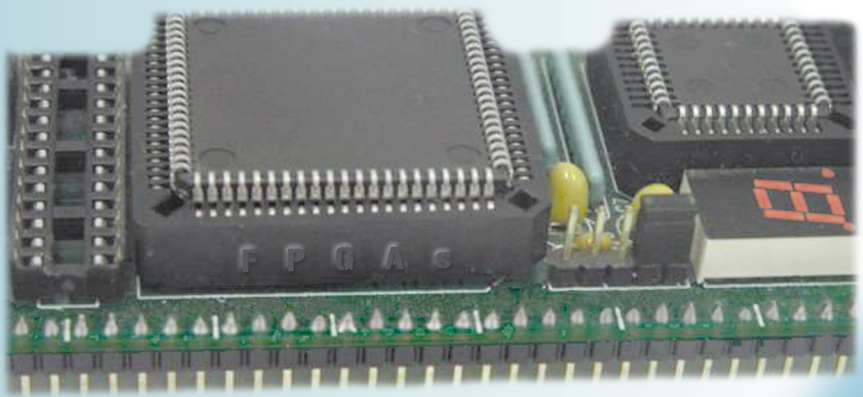
Edward David Moreno Ordonez
Fábio Dacêncio Pereira
Cesar Giacomini Penteado
Rodrigo de Almeida Pericini

Bless
GRÁFICA E EDITORA LTDA

Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)

Os temas abordados neste livro possibilitam convergências tecnológicas, uma vez que focalizam assuntos importantes na formação de pessoas na área de arquitetura de computadores: interligando a linguagem VHDL, circuitos programáveis FPGAs, conceitos de lógica digital, avaliação de desempenho de circuitos e arquitetura de computadores (em especial arquitetura de processadores e microcontroladores).

Assim, recomenda-se este livro aos leitores da área de ciência da computação, sistemas de informação, engenharia elétrica e eletrônica, que pretendam iniciar estudos e adquirir conhecimento nestas áreas.

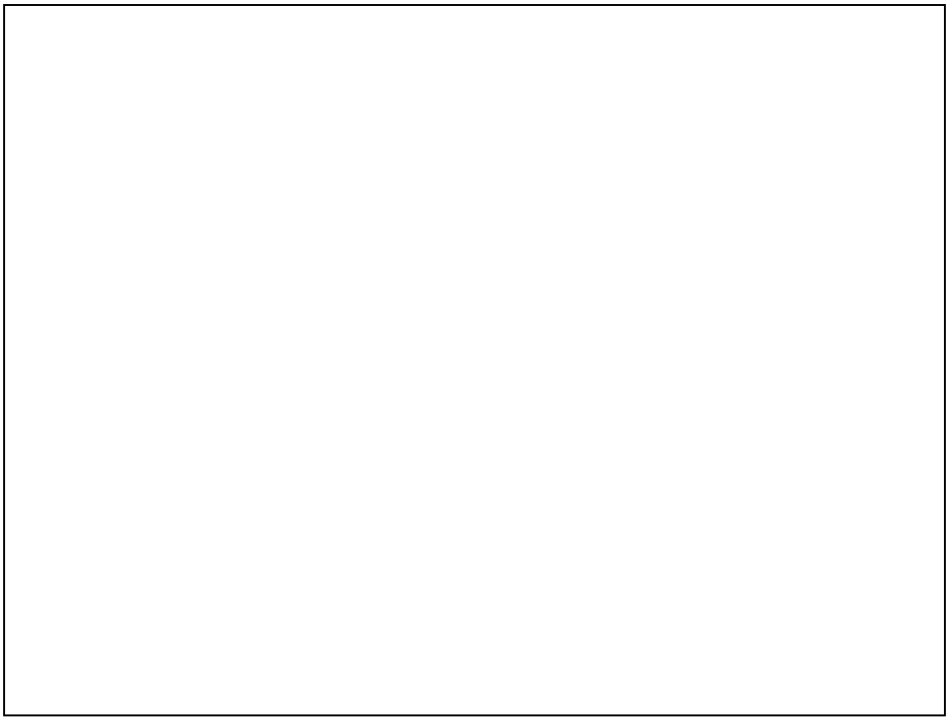


Códigos VHDL adicionais podem ser encontrados na página:

http://www.pdafpga.kit.net.

ISBN 85-87244-13-2

Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)



Edward David Moreno Ordonez
Fábio Dacêncio Pereira
Cesar Giacomini Penteado
Rodrigo de Almeida Pericini

BLESS Gráfica e Editora Ltda.

Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)

Autores

Edward David Moreno Ordonez

Professor Doutor
Faculdade Informática
Fundação de Ensino Eurípides Soares da Rocha - FEESR

Fábio Dacêncio Pereira

Bacharel em Ciência da Computação
Faculdade Informática
Fundação de Ensino Eurípides Soares da Rocha - FEESR

Cesar Giacomini Penteado

Bacharel em Ciência da Computação
Faculdade Informática
Fundação de Ensino Eurípides Soares da Rocha - FEESR

Rodrigo de Almeida Pericini

Bacharel em Ciência da Computação
Faculdade Informática
Fundação de Ensino Eurípides Soares da Rocha - FEESR

Apoio:

Faculdade de Informática, FEESR

Fundação de Amparo à Pesquisa do Estado de São Paulo

BLESS Gráfica e Editora Ltda.

Pompéia, S.P., Janeiro de 2003

Este livro foi realizado com apoio da Faculdade de Informática da Fundação de Ensino Euripides Soares da Rocha (FEESR) e FAPESP (Fundação de Amparo à Pesquisa no Estado de São Paulo).

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

P964

Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs).

Edward David Moreno Ordonez ... [et Al.]. – Pompéia : Bless, 2003.
300p. ; 23 cm.

ISBN: 85-87244-13-2

1. Eletrônica Digital I. Ordonez, Edward David Moreno.

CDD 621.381

ÍNDICE

CAPÍTULO III – CIRCUITOS COMBINACIONAIS

CAPÍTULO IV – CIRCUITOS SEQÜENCIAIS

CAPÍTULO V – GLT: UM SISTEMA DIGITAL PARA AUXILIAR NOS SERVIÇOS TELEFÔNICOS

CAPÍTULO VI – CISIET - CONTROLE INTELIGENTE PARA SISTEMAS DE INJEÇÃO ELETRÔNICA DE TURBINAS.

CAPÍTULO VII – PROJETO E DESEMPENHO DE PROCESSADORES EM FPGAS

CAPÍTULO VIII – PROJETO E DESEMPENHO DE MICROCOTROLADOR EM FPGAS

CAPÍTULO IX – REFERÊNCIAS BIBLIOGRÁFICAS

Prefácio

Os FPGAs estão se consolidando como um dos principais dispositivos configuráveis que permitem uma rápida prototipagem, reconfigurabilidade e baixo custo de desenvolvimento. Projetar circuitos e sistemas digitais usando-se dessa tecnologia é possível através de várias maneiras, sendo uma delas a programação com a linguagem VHDL. Dessa forma, o livro apresenta conceitos básicos e avançados de como usar FPGAs e VHDL em projetos de sistemas digitais básicos como, portas lógicas, circuitos combinacionais e seqüenciais, evoluindo até sistemas avançados tais como, processadores e microcontroladores.

O livro possui a característica de apresentar algumas definições importantes de FPGAs e VHDL e enfatiza no projeto de circuitos e sistemas digitais com exemplos. Os respectivos códigos são devidamente explicados, enfatizando o desempenho dos circuitos projetados em FPGAs. Essas diferenciações são descritas nos seguintes pontos**:**

- Mostra a linguagem VHDL e sua utilização em circuitos projetados em FPGAs;
- Descreve bibliotecas lógicas em VHDL, úteis em projetos de circuitos e sistemas digitais;
- Testa e valida os códigos VHDL gerados relacionados com as bibliotecas selecionadas;
- Gera dados estatísticos da utilização dessas bibliotecas em circuitos reais da XILINX;
- Facilita o aprendizado de projeto de circuitos e sistemas digitais construídos usando-se de VHDL e FPGAs e
- Mostra exemplos de diferentes aplicações que usam sistemas digitais, os quais podem ser implementados em circuitos programáveis e enfatiza sua performance em FPGAs.

Interessante perceber que os temas abordados possibilitam convergências tecnológicas, uma vez que focalizam assuntos importantes na formação de pessoas na área de arquitetura de computadores: interligando linguagem VHDL, circuitos programáveis (FPGAs), conceitos de lógica digital e arquitetura de computadores (em especial arquitetura de processadores e microcontroladores) e avaliação de

desempenho. Além disso, exemplifica algumas aplicações práticas onde o projeto de um sistema digital devidamente realizado em FPGAs auxilia em atividades industriais. Dessa maneira, o livro traz à tona a possibilidade de se transmitir (educação) conceitos avançados relacionados com os assuntos em questão e como realizar otimizações e verificar melhoras com novas propostas de projeto (ciência).

Assim, recomenda-se este livro aos leitores da área de ciência da computação, sistemas de informação, engenharia elétrica e eletrônica, que pretendam iniciar estudos e adquirir conhecimento na área de lógica digital, arquitetura de processadores, arquitetura de computadores e projeto de sistemas digitais.

Estrutura e Organizacao do livro.

O livro está organizado em três partes, contendo um total de nove capítulos, a saber:

PARTE I: CONCEITOS BÁSICOS

Capítulo I: Introdução.
Capítulo II: Conceitos de Linguagem VHDL e FPGAs
Capítulo III: Circuitos Combinacionais
Capítulo IV: Circuitos Seqüenciais

PARTE II: APLICAÇÕES DE SISTEMAS DIGITAIS MODESTOS

Capítulo V: GLT: Um Sistema Digital para Auxiliar nos Serviços Telefônicos
Capítulo VI: CISIET - Um Controlador Inteligente para Injeção Eletrônica de Turbinas

PARTE III: APLICAÇÕES DE SISTEMAS DIGITAIS COMPLEXOS

Capítulo VII: Projeto e Desempenho de Processadores em FPGAs
Capítulo VIII: Projeto e Desempenho de Microcontroladores em FPGAs
Capítulo IX: Referências Bibliográficas

Códigos VHDL que não estão no livro e outros podem ser encontrados na página http://www.pdafpga.kit.net. Assim como, informações sobre os projetos e autores.

CAPÍTULO I - INTRODUÇÃO

Neste capítulo introdutório são discutidos conceitos básicos importantes, apresentando uma visão geral dos assuntos a serem tratados nos próximos capítulos. Entre estes conceitos encontram-se a importância da geração de bibliotecas ou funções lógicas e como essa metodologia de projeto de circuitos digitais pode influenciar no produto final.

São tratados conceitos básicos de circuitos programáveis (FPGA) e da linguagem de descrição de hardware VHDL, discutindo como deve ser o perfil de um projetista de hardware com a chegada desta tecnologia.

A implementação de circuitos digitais complexos era uma ciência dominada apenas por grandes empresas ou universidades de renome internacional. Com o avanço da tecnologia surgiram os FPGAs. Esta tecnologia inovadora está viabilizando a construção e prototipação de circuitos digitais complexos sem a necessidade de muitos recursos computacionais e financeiros. A possibilidade de implementar um circuito digital em um ambiente simplificado e de baixo custo está popularizando cada vez mais esta tecnologia. Atualmente pode-se descrever um circuito digital para FPGA utilizando a linguagem VHDL.

VHDL é uma linguagem de descrição de hardware de alta performance e flexível utilizada na industria e para fins acadêmicos. Esta linguagem é um padrão para descrição de hardware adotado pela tecnologia FPGA.

Este livro integra os conceitos de lógica digital e construção de circuitos digitais utilizando a linguagem VHDL e a tecnologia FPGA. Partindo de circuitos digitais básicos e evoluindo até circuitos complexos como processadores e microcontroladores. Dois escopos principais são discutidos: uma teoria básica da tecnologia FPGA e da linguagem VHDL e aplicações práticas que comprovam o potencial desta tecnologia.

O livro apresenta e discute a descrição em VHDL e a implementação em FPGAs de bibliotecas lógicas de circuitos combinacionais e seqüenciais, que facilitam a implementação de circuitos digitais mais complexos. Entre os circuitos combinacionais que mais se destacam estão:

- Portas lógicas básicas
- Demultiplexadores
- Decodificadores
- Somadores
- Multiplicadores
- Multiplexadores
- Codificadores
- Comparadores
- Subtratores
- Divisores

Entre os circuitos seqüenciais que mais se destacam, estão:

- Diferentes tipos de flip-flops e latches.
- Registradores de deslocamento.
- Máquinas de estado finita (FSM)
- Registradores.
- Contadores.

Estes circuitos combinacionais e seqüenciais são apresentados, respectivamente nos capítulos III e IV. Ambos descritos utilizando a linguagem VHDL

e implementados em FPGA. Por este motivo, o capítulo II apresenta os conceitos básicos de VHDL e FPGAs.

Os capítulos V e VI dedicam-se à implementação de duas aplicações que utilizam alguns dos circuitos estudados nos capítulos III e IV. Estas aplicações são respectivamente, a implementação de um sistema digital que comporta funcionalidades úteis nos serviços telefônicos atuais e um sistema digital que monitora e controla parâmetros físicos importantes para o correto funcionamento de uma turbina de pequeno porte.

No capítulo VII tem-se a implementação e análise de um sistema digital relativamente complexo. Neste caso, uma UCP (Unidade Central de Processamento) de 8 bits. A UCP é implementada utilizando duas diferentes metodologias de descrição VHDL, a descrição microprogramada e com máquina de estados finita (FSM). Também é realizado um estudo comparativo entre uma UCP de 8 e 16 bits.

No final do capítulo VII, tem-se a descrição em VHDL da UCP do microcontrolador HC05 da Motorola, assim como detalhes do seu funcionamento e estrutura.

Similarmente ao capítulo VII de processadores, o capítulo VIII mostrará detalhes de projeto e desempenho de microcontroladores descritos em VHDL e implementados em FPGA. Em especial, será realizado um estudo de uma versão simplificada do microcontrolador M68HC11 da Motorola.

1.1 - RESUMO DA EVOLUÇÃO DA ELETRÔNICA ATÉ OS CIRCUITOS PROGRAMÁVEIS

Na segunda metade do século XX a eletrônica começa a se destacar e influenciar na área industrial, na comunicação, no entretenimento, na medicina, na tecnologia espacial, entre outras áreas. Surgem aparelhos de comunicação, de controle, utilitários domésticos e etc. As grandes evoluções na eletrônica que possibilitaram esses avanços tecnológicos, podem ser destacadas como:

- A invenção da válvula no início da década de 40. A válvula é um componente que trabalha com tensões elétricas relativamente altas. Era comum alguma válvula queimar com pouco tempo de funcionamento.
- O transistor criado em 1947, veio para substituir as válvulas, um componente baseado na tecnologia de semicondutores, elementos com propriedades físicas especiais, tal como o germânio e o silício. O transistor é considerado um

"componente de estado sólido" e possui a grande vantagem de não se aquecer como as válvulas, além de ser fisicamente menor. Gradativamente as válvulas passaram a ser substituídas por transistores, fazendo com que a maioria dos equipamentos eletrônicos passassem a ocupar um menor espaço físico, esquentando menos e consumindo menos energia elétrica.

- Em 1961 surge o primeiro circuito integrado disponível comercialmente, com a junção de vários transistores em um só componente, colocando um circuito relativamente grande dentro de uma só pastilha de silício.

Fatores como mercado de eletro-eletrônicos em contínua expansão, consumidor mais exigente, empresas que buscam tecnologias capazes de aumentar a produção e a qualidade, diminuindo o tempo e custo final do produto, globalização e muitos outros, estão mudando o cenário dos ambientes de projetos de sistemas digitais e o perfil dos profissionais que trabalham nesta área.

Originalmente existiam projetistas de *hardware* ou de *software* separadamente. Os modernos projetistas de sistemas computacionais dedicados, devem possuir conhecimentos multidisciplinares, de arquitetura de computadores a desempenho de algoritmos de processamento digital de sinais.

A computação reconfigurável introduziu novos paradigmas aos modelos computacionais atuais, tanto em nível de software quanto de hardware. Em muitos sistemas digitais como, ambientes de tempo real, os processadores de propósito geral não têm um desempenho satisfatório. Alguns recursos alternativos como: processador digital de sinal (DSP) e processadores de aplicação especifica, melhoram a performance e desempenho destes sistemas.

Uma tecnologia relativamente nova é a implementação de circuitos de aplicações especificas em FPGAs (*Field Programmable Gate Array*). O rápido desenvolvimento da tecnologia ligada a dispositivos de lógica programável em nível de velocidade e capacidade, permite aos projetistas implementar circuitos e arquiteturas cada vez mais complexas, sem a necessidade do uso de grandes recursos de fundição em silício.

Sabendo que a maioria desses circuitos são reprogramáveis, sua primeira aplicação seria em projetos de prototipagem, economizando consideravelmente tempo e custo, devido à agilidade e facilidade em todo o processo de desenvolvimento, simulação, teste, depuração e alteração do projeto. Dentre as vantagens da rápida

realização de protótipos de sistemas computacionais, para aplicações dedicadas, usando tecnologia de circuitos programáveis, destacam-se:

- Maior velocidade de chegada do produto ao mercado consumidor, pela detecção antecipada de problemas quanto ao *hardware* do sistema;
- Maior confiabilidade do sistema, item chave para desenvolvimento de sistemas de tempo real ou biomédico;
- Possibilidade de desenvolvimento conjunto de *hardware* e *software*, sem interdependências, de modo a aumentar a velocidade com que o produto final chega à linha de produção.

Estas vantagens serão demonstradas nos próximos capítulos, através de estatísticas de desempenho dos circuitos implementados utilizando esta tecnologia.

1.2 - FUNÇÕES OU BIBLIOTECAS LÓGICAS

O desenvolvimento de funções ou bibliotecas projetadas de maneira otimizada, testadas e validadas para futuramente integrar toda uma lógica padronizada, possibilita a concentração de esforços no desenvolvimento de hardware programável diferenciado. Para o desenvolvimento destas bibliotecas é necessário uma grande pesquisa em busca do melhor algoritmo e métodos para testes e validação, gerando assim bibliotecas otimizadas e confiáveis. Esta metodologia de projeto possui um grande impacto na produtividade, pois as funções padronizadas são, por definição, reutilizáveis e podem atender a uma grande variedade de aplicações trazendo, em pouco tempo, o retorno do investimento inicial.

As bibliotecas lógicas são descritas em VHDL, uma linguagem de descrição de hardware, podendo ser utilizadas em circuitos programáveis FPGAs. O software utilizado para descrever os circuitos em VHDL é a ferramenta da Xilinx, Foundation Series, e os FPGAs utilizados pertencem às famílias XC4000XL e Virtex, especificamente, o XC4010XLpc84 e o XCV100pq240, ambos do fabricante Xilinx.

1.3 - FPGAS

FPGAs são circuitos programáveis compostos por um conjunto de células lógicas ou blocos lógicos alocados em forma de uma matriz. Em algumas arquiteturas, os blocos lógicos possuem recursos seqüenciais tais como *flip-flops* e/ou registradores.

Cada fabricante nomeia seu bloco lógico, podendo haver mais de um nome para um mesmo fabricante, como mostrado na tabela 1.1.

Fabricante	Nome do Bloco Lógico
Xilinx	CLB (Configurable Logic Block).
Actel	LM (Logic Modules).
Altera	LE (Logic Element) para as séries 8000 e 10000. Macrocell para as séries 5000, 7000 e 9000.

Tabela 1.1 - Lista de fabricantes e seus blocos lógicos.

Em geral, a funcionalidade destes blocos assim como seu roteamento são configuráveis via *software*. Os FPGAs além de proporcionar um ambiente de trabalho simplificado e de baixo custo, possibilita operar com um número ilimitado de circuitos através da configuração do próprio dispositivo.

1.4 – ELEMENTOS BÁSICOS DE UM FPGA

A estrutura básica de um FPGA pode variar de fabricante para fabricante, de família para família ou até em uma mesma família pode existir variações, mas alguns elementos fundamentais são mantidos. Pode-se destacar três elementos fundamentais em um FPGA:

- CLB (*Configurable Logic Block*): bloco lógico configurável, unidade lógica de um FPGA.
- IOB (*In*/*Out Block*): bloco de entrada e saída, localizado na periferia dos FPGAs, são responsáveis pela interface com o ambiente.
- SB (*Switch Box*): caixa de conexão, responsável pela interconexão entre os CLBs, através dos canais de roteamento.

Na figura 1.1, tem-se a representação dos elementos básicos de um FPGA e sua disposição interna, destacando-se os CLBs, IOBs e SBs.

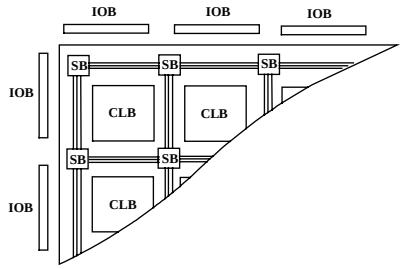


Figura 1.1 - Representação dos elementos básicos de um FPGA.

Nos últimos anos a quantidade de portas lógicas disponíveis num FPGA tem crescido num ritmo muitíssimo acelerado, possibilitando a implementação de arquiteturas cada vez mais complexas.

1.5 - ÁREAS DE APLICAÇÕES COM FPGAS

Tratando-se de FPGAs, um circuito flexível e poderoso, é difícil e injusto definir uma lista das áreas de aplicações, pois é uma tecnologia relativamente recente, onde a cada dia novas aplicações são implementadas. Porém, pode-se definir as áreas mais comuns de aplicações, citadas a seguir:

- *Previsão do Tempo:* HDTV e CATV.
- *Consumo:* Decodificador de áudio digital, arcade games, vídeo games e sistemas de karaokê.
- *Transportes:* Sistemas de estradas de ferro.
- *Industrial:* Equipamentos de teste e medidas, equipamentos médicos, controle remoto, robótica, emulador ASIC e sistemas de visão.
- *Comunicação de Dados:* Multiplexadores, roteadores, vídeo conferência, criptografia, modems, compressão de dados, LANs, HUBs, FDDI e Wireless LANs.
- *Telecomunicação:* Interfaces SONET, interfaces de fibras ópticas, ATM, interfaces ISDN, controlador de voice-mail, multiplexadores T1 e compressão de dados.
- *Militar:* Sistemas de computadores, comunicação e controle de fogo.
- *Computadores:* Interfaces de memória, controladores DMA, controladores de cache, co-processadores SSP, multimídia e gráficos.

- *Periféricos:* Controladores de disco, controladores de vídeo, FAX, máquinas de caixa, modems, sistemas POS, cartões de aquisição de dados, terminais, impressoras, scanners e copiadoras.

Uma área de aplicação que vem se destacando é a de processadores embarcados, onde um processador é integrado a um sistema maior com objetivo de auxiliar no controle e execução de tarefas.

1.6 - DESEMPENHO DE UM CIRCUITO

Há vários parâmetros para se medir o desempenho de circuitos digitais em FPGAs, os principais parâmetros são: (i) a *ocupação espacial*, que determina quantos componentes são necessários para implementar o circuito. (ii) O *desempenho temporal*, que determina o tempo de atraso do sinal (informação) através do circuito.

Ao implementar um circuito digital deseja-se que o espaço utilizado e o tempo de execução sejam os menores possíveis. Isto é, o circuito deve ser veloz e ocupar pouco espaço no FPGA. Infelizmente, satisfazer estes dois critérios nem sempre é possível. Isto é verdade, seja qual for a tecnologia de projeto de circuitos digitais utilizada.

Existem vários algoritmos de otimização de circuitos específicos, como por exemplo, algoritmos de circuitos aritméticos, somadores, subtratores, divisores, multiplicadores, etc. Alguns *softwares* de síntese de circuitos digitais possuem mecanismos de otimização de circuitos genéricos, onde normalmente o projetista define o grau de otimização temporal ou espacial e o *software* através de algoritmos genéricos tenta atender a configuração desejada. No caso da otimização espacial, o *software* busca a eliminação de trechos redundantes do circuito. Já na otimização temporal busca-se diminuir o tempo que o circuito utiliza para gerar a partir das entradas a informação desejada, mesmo que o espaço necessário para implementação torne-se maior.

Todos os circuitos digitais descritos em VHDL neste livro, possuem estatísticas de desempenho espacial e temporal geradas a partir da implementação em FPGA. Estas estatísticas são analisadas a fim de discutir as formas mais otimizadas de descrição dos circuitos propostos.

1.7 - FERRAMENTAS CAE E DE SÍNTESE

As ferramentas CAE juntamente com as ferramentas de síntese evoluíram consideravelmente possibilitando ao projetista desenvolver circuitos cada vez mais confiáveis e otimizados. Uma das funções da síntese é otimizar projetos desenvolvidos nas ferramentas CAE. Essa otimização pode ser controlada pelo projetista através de restrições impostas antes do processo de síntese. Essas restrições devem ser definidas cuidadosamente, pois poderão afetar o circuito final. Para ganhar espaço e/ou velocidade é exigido dos projetistas conhecimentos específicos do projeto e técnicas de otimização. Algumas características das ferramentas CAE, são:

- Especificação do comportamento do FPGA através de diagramas esquemáticos, linguagens de descrição de Hardware (HDL - *Hardware Description Language*) e/ou diagrama de fluxo (máquina de estados).
- Sintetização do circuito obedecendo às restrições impostas pelo projetista; não havendo restrições, a ferramenta de síntese busca a configuração padrão para a síntese do circuito.
- Verificação do funcionamento do circuito através de simulação funcional e temporal, já considerando os tempos de atraso gerados pela lógica resultante do processo de síntese.
- Capacidade de gerar relatórios estatísticos, com dados de comportamento e desempenho do circuito desenvolvido.
- Possibilita a implementação do projeto em nível físico, fazendo com que o circuito programável assuma o comportamento descrito no projeto.

A seguir exemplos de algumas ferramentas CAE para tecnologias baseadas em circuitos programáveis, FPGA:

Fabricante	Nome da Ferramenta CAE
Xilinx	Xilinx Foundation Series Xilinx Foundation ISE
Altera	MAX-PLUS MAX-PLUS II Quartus Quartus II

Exemplo de algumas ferramentas de síntese de circuito mais utilizadas:

- LeonardoSpectrum
  http://www.mentor.com/leonardospectrum/
- Synopsys FPGA Express
  http://www.synopsys.com/

As ferramentas de síntese na maioria das vezes não são desenvolvidas pelos fabricantes de ferramentas CAE e sim por empresas especializadas nessa tecnologia específica, sendo atribuídas às ferramentas CAE posteriormente. Uma ferramenta CAE pode conter uma ou mais ferramentas de síntese. Dependendo do projeto, o desempenho de uma ferramenta de síntese pode ser melhor do que de outra, ficando com o projetista a responsabilidade de analisar e definir a melhor ferramenta para seu projeto. Daí, a importância de conhecer as vantagens e desvantagens de cada uma, para selecionar a ferramenta adequada.

Os circuitos digitais descritos neste livro foram sintetizados pela ferramenta Synopsys FPGA Express, específica para FPGAs da Xilinx.

1.8 - LINGUAGENS DE DESCRIÇÃO DE HARDWARE (HDL)

Uma linguagem de descrição de *hardware* (HDL) é própria para modelar a estrutura e/ou o comportamento de um *hardware*. Existem dois aspectos importantes para a descrição de um *hardware* que uma HDL pode facilitar: o verdadeiro comportamento abstrato e a estrutura do hardware.

Comportamento abstrato: Uma linguagem de descrição de hardware é estruturada de maneira a facilitar a descrição abstrata do comportamento do hardware para propósitos de especificação. O comportamento pode ser modelado e representado em vários níveis de abstração durante o projeto.

Estrutura de Hardware: É possível o modelamento de uma estrutura de hardware em uma linguagem de descrição independente do comportamento do circuito.

Exemplos de linguagens HDL: VHDL, VERILOG, AHDL (linguagem desenvolvida pela ALTERA), Handel-C, SDL, ISP, ABEL e outras mais.

Todos os circuitos digitais estudados e apresentados neste livro foram descritos utilizando a linguagem VHDL, discutida na seção seguinte.

1.9 - LINGUAGEM VHDL

VHDL : **V** ery High Speed Integrated Circuit

H ardware
D escription
L anguage

VHDL é uma linguagem padronizada para descrever componentes digitais, permitindo a transferência de componentes ou projetos para qualquer tecnologia em construção de hardware existente ou que ainda será desenvolvida. Sua estrutura tem forte influência da linguagem ADA, linguagem encomendada para ser padrão no desenvolvimento de *software*.

Diferente da linguagem ADA, que não teve sucesso devido a sua complexidade, a linguagem VHDL firmou-se como um padrão internacional. Toda ferramenta comercial de síntese de circuitos aceita ao menos um subconjunto do VHDL.

A linguagem VHDL oferece uma rica variedade de construções que permitem modelar o *hardware* em um elevado nível de abstração. A importância da utilização de linguagens de descrição de hardware manifesta-se em diversos aspectos do projeto:

- Documentação do sistema: a própria descrição do sistema já é uma forma de documentação para os projetistas em VHDL.
- Simulação em diversos níveis: desde a sua especificação funcional e temporal o circuito pode ser simulado para verificar seu correto funcionamento. Simulações mistas podem ser feitas com blocos estruturais e comportamentais.
- Simplifica a migração tecnológica: o sistema pode ser facilmente re-sintetizado em outras tecnologias, desde que se disponha das ferramentas de baixo nível correspondentes.
- Reutilização de recursos: a construção de bibliotecas de módulos na linguagem permite reutilizar parte de projetos já realizados. Um centro de projetos pode desenvolver bibliotecas específicas para sua própria área de aplicação.

Apesar de amplamente utilizada na descrição e síntese de sistemas digitais, a linguagem VHDL foi primariamente definida com objetivos de simulação. Estes objetivos são muitas vezes, conflitantes, e esta linguagem vem sofrendo contínuas alterações visando uma melhor adaptação aos propósitos gerais em que vem sendo utilizada.

Alguns autores tentam utilizar VHDL como a única ferramenta para descrição do sistema e de seu ambiente, aproveitando-se da riqueza da linguagem e extrapolando suas limitações quanto à síntese de alto nível.

1.9.1 - RESUMO HISTÓRICO DA LINGUAGEM VHDL

Em 1980 o Departamento de Defesa dos Estados Unidos (DoD) pretendia desenvolver um circuito que adotava uma metodologia comum e que pudesse ser reutilizável em novas tecnologias. Ficou claro que havia a necessidade de uma linguagem de programação padronizada para descrever a função e a estrutura de circuitos digitais para o projeto de circuitos integrados (CI).

O DoD criou então o *VHSIC Hardware Description Language* ou VHDL como é mais conhecido atualmente. A *IBM*, *Texas Instruments* e *Intermetrics* agruparam-se para desenvolvimento da linguagem VHDL, com a união de experiências no desenvolvimento de linguagem de alto nível e técnicas de projeto *top-down*, juntamente com ferramentas de simulação.

O ano de 1987 foi marcado por dois acontecimentos importantes: O DoD designou que todos os circuitos eletrônicos digitais fossem descritos em VHDL e o *Institute of Electrical and Electronics Engineers* (IEEE), ratificou o VHDL como IEEE Padrão 1076, assegurando o sucesso da linguagem.

O F22, aeronave de tática de combate avançada, foi um dos primeiros projetos a ter todos os subsistemas eletrônicos descritos em VHDL. O sucesso do projeto ajudou a estabelecer esta linguagem.

Assim, o VHDL torna-se um padrão industrial, mas a falta inicial de ferramentas significou que a linguagem estava demorando em ser adotada comercialmente. Em 1993, a linguagem VHDL foi revisada, definindo o padrão IEEE 1076'93.

Em 1996, as ferramentas de simulação e de síntese foram incorporadas pelo padrão IEEE 1076'93. Isso permitiu a utilização desta versão padronizada em metodologias de projetos *top-down*. O pacote de ferramentas de síntese para linguagem VHDL torna-se parte do padrão IEEE 1076, especificamente o pacote 1076.3. Isto melhorou consideravelmente a portabilidade dos projetos entre diferentes ferramentas de síntese. No padrão IEEE 1076.4 (VITAL) foram agregados modelos ASIC e bibliotecas para FPGA em VHDL. Em dezembro de 1997 foi publicado o manual de referência da linguagem VHDL.

1.9.2 - VANTAGENS EM UTILIZAR A LINGUAGEM VHDL

O surgimento desta linguagem se fez necessário devido ao rápido avanço tecnológico alcançado pelas indústrias de circuito integrado, tendo como ápice da tecnologia de alta velocidade VHSIC (*Very High Speed Integrated Circuits*) o que permitia uma maior integração e consequentemente uma maior complexidade de circuitos contido em uma mesma pastilha. Algumas vantagens em utilizar VHDL, são:

- Reduz tempo/custo de desenvolvimento.
- Maior nível de abstração.
- Projetos independentes da tecnologia.
- Facilidade de atualização dos projetos.
- Grande número de usuários (internacional).

No capítulo seguinte, apresenta-se uma descrição mais detalhada de circuitos programáveis (FPGAs) e da linguagem VHDL.

CAPÍTULO II - FPGAS E LINGUAGEM VHDL

Neste capítulo, apresentam-se conceitos básicos de circuitos programáveis FPGAs e da linguagem de descrição de hardware VHDL. Inicialmente é apresentada a estrutura interna e roteamento dos circuitos FPGAs, discutindo conceitos de reconfiguração. Posteriormente tem-se a definição das formas de descrever um circuito utilizando VHDL, exemplificando cada metodologia de descrição de hardware, onde a sintaxe da linguagem é baseada no padrão VHDL'93. Além de descrever os principais elementos que compõem a estrutura da linguagem como: operadores, expressões, atributos, tipos da linguagem, declaração de entidades e arquiteturas, expressões concorrentes, expressões seqüenciais, processos, comando seqüenciais e outros.

Como ponto de partida para este capítulo, faz-se necessário apresentar a estrutura interna de circuitos programáveis FPGAs, demostrando algumas arquiteturas desta tecnologia, bem como discutir os conceitos de reconfiguração e roteamento desses circuitos. Não é objetivo deste livro explorar em profundidade detalhes da tecnologia FPGA, mas apenas alguns conceitos básicos.

2.1 - ESTRUTURA INTERNA

FPGAs são circuitos programáveis compostos de *CLBs*, *switch boxes*, *IOBs* e canais de roteamento, discutidos no capítulo I. A figura 2.1 apresenta quatro principais arquiteturas internas utilizadas em circuitos programáveis FPGA: matriz simétrica, *sea-of-gates*, *row-based* e PLD hierárquico.

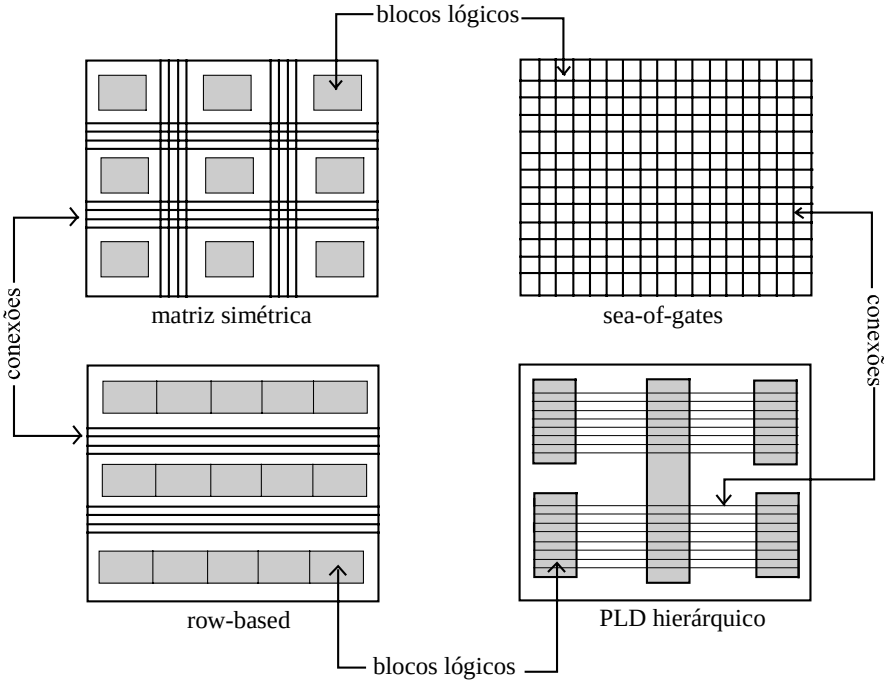


Figura 2.1 – Arquiteturas interna de circuitos programáveis FPGA.

A arquitetura *sea-of-gates* é um circuito composto por transistores ou blocos lógicos de baixa complexidade. A vantagem dessa arquitetura é a grande disponibilidade de portas lógicas por área. Porém, como não há uma área dedicada para

o roteamento, é necessário que o mesmo seja feito sobre as células, muitas vezes inutilizando áreas disponíveis para implementação de uma determinada lógica.

Nos circuitos de arquitetura *row-based* os blocos lógicos estão dispostos horizontalmente. Existe uma área dedicada de roteamento localizada entre as linhas de blocos lógicos. As arquiteturas *row-based* e *sea-of-gates* originaram-se das metodologias de projeto de ASICs, *standard-cells* e *gate-array*.

A arquitetura tipo PLD hierárquico é constituído por uma matriz de blocos lógicos, denominados *logic arrays blocks*, sendo interligados através do recurso de roteamento conhecido como matriz programável de interconexão (PIA). Esse tipo de dispositivo é dito hierárquico, porque os blocos lógicos podem ser agrupados entre si.

A arquitetura tipo matriz simétrica é flexível no roteamento, pois possui canais verticais e horizontais.

2.2 – ROTEAMENTO E RECONFIGURAÇÃO

Nesta seção são discutidos conceitos básicos de roteamento e reconfiguração de circuitos programáveis e reconfiguráveis FPGAs.

ROTEAMENTO

O roteamento é a interconexão entre blocos lógicos através de uma rede de camadas de metal. As conexões físicas entre os blocos lógicos são feitas com transistores controlados por bits de memória (*PIP*) ou por chaves de interconexão (*switch matrix*).

Eis alguns elementos básicos utilizados na malha de roteamento da família XC4000 da Xilinx:

- **Conexões globais:** estas formam uma rede de interconexão em linhas e colunas ligadas através de chaves de interconexão. Esta rede circunda os blocos lógicos (CLBs) e os blocos de E/S (IOBs).
- **Matriz de conexão (Switch Matrix):** são chaves de interconexão que permitem o roteamento entre os blocos lógicos através das conexões globais.
- **Conexões diretas:** interligam CLBs vizinhos e permitem conectar blocos com menor atraso, pois não utilizam recursos globais de roteamento.
- **Linhas longas:** são conexões que atravessam todo o circuito sem passar pelas matrizes de conexão e são utilizadas para conectar sinais longos.

RECONFIGURAÇÃO

A exigência de alto poder de processamento e o surgimento de novas aplicações, promovem uma constante busca por alternativas e arquiteturas que visem melhorar a performance dos computadores, especialmente em aplicações de tempo real.

Acoplando um dispositivo programável FPGA a um processador de propósito geral (GPP), torna-se possível a exploração eficiente do potencial das chamadas arquiteturas reconfiguráveis.

Arquiteturas reconfiguráveis permitem ao projetista a criação de novas funções, e possibilita a execução de operações com um número consideravelmente menor de ciclos do que necessário em GPPs. Em uma arquitetura reconfigurável, são desnecessárias muitas das unidades funcionais complexas usualmente encontradas em processadores de propósito geral.

Os métodos de reconfiguração de um dispositivo programável e reconfigurável podem ser classificados como:

- **Reconfiguração total:** é a forma de configuração, onde o dispositivo reconfigurável é inteiramente alterado. Também tratada apenas como configuração.
- **Reconfiguração parcial:** é a forma de configuração que permite que somente uma parte do dispositivo seja reconfigurada. A reconfiguração parcial pode ser: não-disruptiva, onde as porções do sistema que não estão sendo reconfiguradas permanecem completamente funcionais durante o ciclo de reconfiguração; ou disruptiva, onde a reconfiguração parcial afeta outras partes do sistema, necessitando de uma parada no funcionamento do mesmo.
- **Reconfiguração dinâmica:** também chamada de *run-time reconfiguration* (RTR), *on-the-fly reconfi-guration* ou *in-circuit reconfiguration*. Todas essas expressões podem ser traduzidas também como reconfiguração em tempo de execução. Nesse tipo de reconfiguração não há necessidade de reiniciar o circuito ou remover elementos reconfiguráveis para programação.
- **Reconfiguração extrínseca:** reconfigura parcialmente o sistema, mas somente considerando cada FPGA que o compõe como unidade atômica de reconfiguração.
- **Reconfiguração intrínseca:** reconfigura parcialmente cada FPGA que compõe o sistema.

A partir desta seção apresentam-se conceitos básicos da linguagem VHDL utilizados nas implementações dos circuitos digitais propostos nos capítulos posteriores.

2.3 - DESCRIÇÃO ESTRUTURAL E COMPORTAMENTAL

Em VHDL existem duas formas para a descrição de circuitos digitais: a *estrutural* e a *comportamental*. A forma *estrutural* indica os diferentes componentes que constituem o circuito e suas respectivas interconexões. Desta maneira pode-se especificar um circuito e saber como é seu funcionamento.

A forma *comportamental* consiste em descrever o circuito pensando no seu comportamento e funcionamento e não na sua estrutura. Essa metodologia facilita a descrição de circuitos onde a estrutura interna não está disponível, mas o seu funcionamento e comportamento podem ser interpretados. Assim, esse tipo de descrição vem se desenvolvendo a cada dia. A descrição comportamental pode-se dividir em duas metodologias, dependendo do nível de abstração: a descrição *algorítmica* e de *fluxo de dados*.

Na descrição de um circuito utilizando a linguagem VHDL, é comum ter-se trechos implementados de maneira comportamental e estrutural, sendo de responsabilidade do projetista a utilização correta dos métodos de implementação de circuitos em VHDL.

2.4 - EXEMPLO DOS ESTILOS DE DESCRIÇÕES EM VHDL

A figura 2.1 representa um comparador de 1 bit simplificado. Este será descrito nos dois estilos de descrição de circuitos digitais em VHDL, estrutural e comportamental. Onde U1, U2 são os componentes do circuito, L1 é uma linha de conexão entre os componentes, e E1, E2 e S são as portas de entrada e saída do circuito.

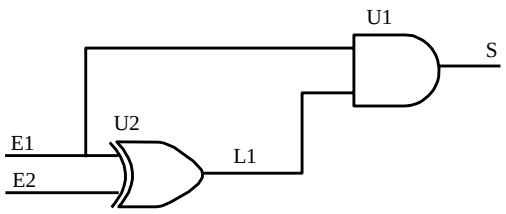


Figura 2.1 - Representação de um comparador de 1 bit

O funcionamento do comparador de 1 bit é simplificado, onde os valores lógicos da portas de entrada E1 e E2 serão comparados. Se E2 for menor que E1 a saída S recebe o valor lógico '1', caso contrário, '0'. Este exemplo do comparador de 1 bit será utilizado para demostrar as formas de descrição de circuito digitais em VHDL nas seções 2.4.1, 2.4.2 e 2.4.3.

2.4.1 - DESCRIÇÃO ALGORÍTMICA

A descrição algorítmica é um conjunto de passos que descreve de forma comportamental o circuito digital projetado. Em primeiro lugar, deve-se descrever a entidade do circuito, onde são definidas as portas de entrada e saída. A entidade (*entity*) independente do método de descrição de circuitos digitais sempre se mantém a mesma. Já a arquitetura (*architecture*) é responsável pela descrição do circuito de maneira estrutural ou comportamental. Neste caso, o comparador de 1 bit (figura 2.1) é descrito na seqüência, de maneira comportamental, aplicando o princípio algorítmico.

Comparador de 1 Bit (Descrição Algorítmica)

```
-- definição da entidade: portas de E/S
entity comp is
port (e1,e2: in bit;
          s: out bit);
end comp;

-- definição da arquitetura: descrição da lógica interna do circuito
architecture comp_alg of comp is
begin
      process (e1,e2)
      begin
            if e2 < e1 then
               s<='1';
            else
               s<= '0';
            end if;
       end process;
end comp_alg;
```

Nota-se que a comparação foi descrita de forma comportamental utilizando o comando seqüencial IF-THEN-ELSE, desprezando a lógica composta pelas portas lógicas XOR e AND do comparador de 1 bit representado na figura 2.1.

2.4.2 - DESCRIÇÃO DE FLUXO DE DADOS

A descrição de fluxo de dados pode ser visualizada como a transferência entre registradores possibilitando o paralelismo de instruções. Quando se tem várias instruções, estas tornam-se concorrentes entre si.

Comparador de 1 Bit (Descrição de Fluxo de Dados)

```
architecture comp_fluxo_dados of comp is
begin
        s<= e1 when  e2> e1
        else e2;
end comp_fluxo_dados;
```

É interessante perceber que em ambas as descrições comportamentais não foi necessário utilizar os componentes do circuito descrito na figura 2.1, descrevendo apenas o comportamento em um alto nível de abstração.

2.4.3 - DESCRIÇÃO ESTRUTURAL

Para descrever um circuito na forma estrutural, deve-se conhecer seus componentes e interconexões. Em sistemas onde não é possível visualizar a estrutura interna de maneira detalhada, torna-se difícil a descrição utilizando a metodologia estrutural.

Comparador de 1 bit (Descrição Estrutural)

```
architecture comp_fluxo_dados of comp is
signal l1: bit;
begin
        u1: entity xor2 port map (e1,e2,l1);
        u2: entity and2 port map (e1,l1,s);
end comp_fluxo_dados;
```

Interessante observar que sem a correta visualização dos componentes e as respectivas interconexões do comparador de 1 bit, conforme a descrição da figura 2.1, pode dificultar sua descrição estrutural em VHDL.

2.5 - ELEMENTOS SINTÁTICOS DO VHDL

Toda linguagem possui elementos sintáticos, tipos de dados e estruturas. A linguagem VHDL não é diferente, porém é uma linguagem de descrição de hardware, portanto deve oferecer suporte para a descrição de trechos concorrentes. A seguir os elementos sintáticos mais comuns:

Comentários: Qualquer linha precedida de -- não será compilada.

Símbolos especiais: Contendo apenas um caracter: + - / * ( ) . , : ; & ‘ “ < > = | # Contendo dois caracteres: ** => := /= >= =< <> --

Identificadores: Identificadores são usados para nomear objetos da linguagem como variáveis, sinais, rotinas, etc. Composto por letras, números e o símbolo _ . Nunca pode conter símbolos especiais ou coincidir com palavras reservadas. Maiúsculas e minúsculas são consideradas iguais, assim VHDL, vhdl e VhDl são possibilidades que representam o mesmo objeto.

Números: Qualquer número que se encontra é considerado na base 10. Admite-se a notação científica convencional para números com ponto flutuante. É possível alterar a base de um número usando o simbolo # . Exemplo: 2#00001111, 16#0F ambos representam o número 15, respectivamente na base 2 e 16.

Caracteres: Qualquer caracter deve estar entre aspas simples: ‘1’ ‘A’.

Cadeia de bits: Cadeia ou vetor de bits é uma seqüência de bits em uma determinada base Ex.: B “00001111”, X”0F”, onde B é binário e X Hexadecimal.

Palavras reservadas: São palavras que possuem um significado especial, são instruções e elementos que permitem definir sentenças. As palavras reservadas do padrão VHDL’93 são:

ABS	**CONFIGURATION**	**INOUT**	**OR**	**THEN**
ACCESS	**CONSTANT**	**IS**	**OTHERS**	**TO**
AFTER	**DISCONNECT**	**LABEL**	**OUT**	**TRANSPORT**
ALIAS	**DOWNTO**	**LIBRARY**	**PACKAGE**	**TYPE**
ALL	**ELSE**	**LINKAGE**	**PORT**	**UNITS**
AND	**ELSIF**	**LOOP**	**PROCEDURE**	**UNTIL**

ARCHITECTURE	**END**	**MAP**	**PROCESS**	**USE**
ARRAY	**ENTITY**	**MOD**	**RANGE**	**VARIABLE**
ASSERT	**EXIT**	**NAND**	**RECORD**	**WAIT**
ATTRIBUTE	**FILE**	**NEW**	**REGISTER**	**WHEN**
BEGIN	**FOR**	**NEXT**	**REM**	**WHILE**
BLOCK	**FUNCTION**	**NOR**	**REPORT**	**WITH**
BODY	**GENERATE**	**NOT**	**RETURN**	**XOR**
BUFFER	**GENERIC**	**NULL**	**SELECT**	**XNOR**
BUS	**GUARDED**	**OF**	**SEVERITY**	**SLL**
CASE	**IF**	**ON**	**SIGNAL**	**SLA**
COMPONENT	**IN**	**OPEN**	**SUBTYPE**	**ROL**
GROUP	**POSTPONED**	**ROR**	**SRA**	**REJECT**
IMPURE	**PURE**	**SHARED**	**SRL**	**INERTIAL**
LITERAL	**UNAFFECTED**			

2.6 - OPERADORES

Operador de concatenação:

& concatena bits e cadeias de bits, gerando um novo vetor de bits.

Ex.: S<= “000”&”111”; onde S é um vetor de 6 bits;

Operadores aritméticos

****** serve para elevar um número a uma potência: 2**3 é 2^3. O operador pode ser inteiro ou real.

ABS devolve o valor absoluto.

***** serve para multiplicar qualquer dado do tipo numérico (bit e bit_vector não são numéricos).

/ serve para dividir qualquer dado numérico;

MOD calcula o módulo de dois números e os operandos só podem ser inteiros.

REM calcula o resto da divisão inteira.

+ quando está entre dois operandos calcula a soma, quando está na frente de um operando, indica que este é positivo.

- quando está entre dois operandos calcula a subtração. Quando esta na frente de um operando, indica que este é negativo.

Operadores relacionais

= ou /= o primeiro indica *true*, se os operandos forem iguais e *false* se forem diferentes; o segundo funciona justamente ao contrário.

< , <= , > ou >= indicam respectivamente menor, menor igual, maior e maior igual.

Operadores lógicos

NOT , AND , NAND , OR , NOR , XOR ou **XNOR** atuam sobre os tipos bit, bit_vector e boolean. No caso de operações com vetores, a operação é realizada bit a bit.

2.7 - TIPOS DE DADOS

Há basicamente dois tipos de dados: escalares e compostos. A seguir, uma melhor descrição de cada um deles.

2.7.1 - TIPOS ESCALARES

Inteiros: São dados cujo conteúdo é um valor numérico inteiro. Pode definir um intervalo usando a palavra reservada **RANGE**, os limites do intervalo são do tipo numérico inteiro.
Ex.: **TYPE** integer **IS RANGE** 0 **TO** 255, neste caso tem-se um tipo inteiro que pode variar dentro do intervalo de 0 a 255.

Reais: Conhecidos como numeração com ponto flutuante que define um número real. Pode-se utilizar intervalos do tipo numérico real.
Ex.: **TYPE** real **IS RANGE** 0.0 **TO** 9.0;

Físicos: Como o próprio nome já diz, são dados que expressão medidas físicas.

Ex.: **TYPE** altitude **IS RANGE** 0 **TO** 1.0e6;

UNITS

um;

mm=1000um;

M=1000mm

Inch=25.4 mm;

END UNITS;

Enumerados: São dados que podem assumir qualquer valor especificado em um conjunto finito. Este conjunto é indicado através de uma lista entre parênteses onde os elementos estão separados por vírgula.

Ex.: **TYPE** estados **IS** (reset, busca, executa);

2.7.2 - TIPOS COMPOSTOS

Matrizes: É uma coleção de elementos do mesmo tipo acessados por um índice. Pode ser unidimensional e multidimencional.

Ex.: **TYPE** positivo **IS ARRAY** (byte **RANGE** 0 **TO** 127) **OF** integer;

Registro: É equivalente ao tipo registro (record) de outra linguagens.

Ex.: **TYPE** agenda **IS**

RECORD

Nome: string;

Rg: integer;

END RECORD;

Subtipos de dados : É um nome de um conjunto de tipos já definidos.

Ex.: **SUBTYPE** digitos **IS** integer **RANGE** 0 **TO** 9, neste caso o subtipo digitos representa os números inteiros entre 0 e 9.

2.8 - ATRIBUTOS

Os elementos como variáveis, sinais e outros podem ser acompanhados por *atributos*. Os atributos são utilizados para explorar as opções de cada elemento da linguagem. Os atributos estão associados a elementos da linguagem através da aspa simples. O tipo do elemento define os atributos herdados. Os atributos definidos neste tópico são os mais utilizados.

Se o elemento *t* é um tipo enumerado, inteiro, real, físico, herdará os seguintes atributos:

t'left. limite esquerdo do tipo t
t'right limite direito do tipo t.
t'low limite inferior do tipo t.
t'high limite superior do tipo t.

Supondo um tipo *t* como o anterior, um membro *x* desse tipo, e um *n* como valor inteiro, pode-se utilizar os seguintes atributos.

t'pos (x) posição de *x* dentro do tipo *t*.
t'val (n) elemento *n* do tipo *t*.
t'leftof (x) elemento que está à esquerda de *x* em *t*.
t'highof (x) elemento que está à direita de *x* em *t*.
t'pred (x) elemento que está adiante de *x* em *t*
t'succ (x) elemento que está atrás de *x* em *t*

Supondo que *a* é uma matriz e *n* é um inteiro desde 1 até o número de dimensão da matriz, então pode-se utilizar os seguintes atributos.

a'left (n) limite esquerdo do intervalo de dimensão *n* de *a*.
a'right (n) limite direito do intervalo de dimensão *n* de *a*.
a'low (n) limite inferior do intervalo de dimensão *n* de *a*.
a'high (n) limite superior do intervalo de dimensão *n* de *a*.
a'range (n) intervalo do índice de dimensão *n* de *a*.
a'length (n) comprimento do índice de dimensão *n* de *a*.

Supondo que *s* é um sinal, pode-se utilizar os seguintes atributos:

s'event Devolve *true*, se acontecer uma troca de valores do sinal *s*.

s'stable (temp) Devolve *true*, se o sinal for estável durante o último período de *temp*.

O atributo *event* é um dos atributos mais utilizados. Através dele é possível detectar a borda de subida de determinado sinal. Ex.: **IF** s'event **AND** s='1' **THEN**

2.9 - CONSTANTES, VARIÁVEIS E SINAIS.

Um elemento em VHDL contém um valor de um tipo específico. Há três tipos de elemento em VHDL: constantes, variáveis e sinais. As variáveis e as constantes possuem um conceito semelhante ao de outras linguagem, enquanto que o conceito de sinais é diferente.

Constantes

Uma constante é um elemento inicializado com um determinado valor que não pode ser modificado depois de ter sido atribuído.

Ex.: **CONSTANT** zera: integer : = 0;

CONSTANT max: bit_vector (3 downto 0) : = “1010”;

Variáveis

Uma variável em VHDL é similar ao conceito de variável de outras linguagens. Toda variável deve ser declarada em um processo (**PROCESS**) ou num subprograma. Seu escopo é válido apenas no processo ou subprograma em que foi declarada. As variáveis são elementos abstratos da linguagem e possuem uma concordância física real imediata, isto é, quando é atribuído algum valor para uma variável, a atualização é instantânea.

Ex.: **VARIABLE** flag: bit;

VARIABLE registrador: bit_vector (2 downto 0);

Sinais

Os sinais são declarados da mesma maneira que as constantes e variáveis com a diferença que os sinais podem ser, *normal*, *register* ou *bus*. Se na declaração não se especifica nada, o sinal é do tipo *normal*. Para utilizar os tipos *register* e *bus*, deve-se declarar explicitamente com as palavras reservadas **REGISTER** e **BUS.**

Os sinais devem ser declarados unicamente nas arquiteturas, pacotes ou nos blocos concorrentes. O conceito de sinais possui um significado físico. Representam conexões reais de um circuito quando um sinal sofre alguma atualização. Isto na verdade não ocorre instantaneamente e sim quando o processo acaba ou encontra uma sentença **WAIT**.

Ex.: **SIGNAL** teste: bit;

SIGNAL byte: bit_vector (7 **DOWNTO** 0) **BUS** : = “00001111”;

2.10 - ENTIDADE E ARQUITETURA

Entidade (ENTITY)

A entidade é uma estrutura onde se define as entradas e saídas de um determinado circuito. Na declaração desta estrutura pode-se incluir outros elementos. A forma geral para declaração de uma entidade é:

ENTITY nome **IS**
[**GENERIC** (lista de parâmetros);]
[**PORT** (lista de portas);]
[declarações]
[**BEGIN** sentenças]
END [**ENTITY**] [nome];

Na declaração da entidade, observam-se os elementos como **PORT** e **GENERIC**. Após a instrução **PORT**, é declarada uma lista de portas de entrada e saída do circuito. Já a instrução **GENERIC** define uma lista de parâmetros que são instanciados com a instrução **GENERIC MAP**, assim como o **PORT MAP** instancia as portas de entradas e saídas do circuito, o **GENERIC MAP** instancia os parâmetros pré-definidos na entidade. A seguir um exemplo de utilização:

Ex.: **ENTITY** registrador **IS**
GENERIC (tamanho: integer);
PORT (Clk: **IN** bit;
Enable: **IN** bit;
Din: **IN** bit_vector (tamanho-1 **DOWNTO** 0);
Dout: **OUT** bit_vector (tamanho-1 **DOWNTO** 0));
END registrador;

Arquitetura (ARCHITECTURE)

Na arquitetura implementa-se o funcionamento do módulo definido na entidade. Toda arquitetura faz referência a uma entidade especifica, mas uma entidade pode pertencer a diferentes arquiteturas. A forma geral para declaração de uma arquitetura é:

```
ARCHITECTURE nome OF nome_entidade IS
[declarações]
BEGIN
    [sentenças concorrentes]
END [ARCHITECTURE] [nome];
```

2.11 - COMPONENTES

O componente é uma estrutura que referencia diretamente uma entidade e possibilita a instanciação e replicação da mesma sem a necessidade de descrevê-la novamente. A forma geral para **declarar** um componente é:

COMPONENT nome [**IS**]
[**GENERIC** (lista de parâmetros);]
[**PORT** (lista de portas);]
END COMPONENT [nome];

Ex.: **COMPONENT** inv
PORT (e: **IN** bit; s: **OUT** bit);
END COMPONENT;

A forma geral para **instanciar** um componente é:

ref_id: [**COMPONENT**] nome_do_componente | **ENTITY** nome_da_entidade [(nome_da_arquitetura)] | **CONFIGURATION** nome_da_configuração [**GENERIC MAP** (parametros)] [**PORT MAP** (lista de portas)];

Onde, ref_id é uma referência ao componente que está sendo instanciado, possibilitando a réplica do mesmo componente, bastando mudar o identificador de referência.

Ex.: u1:inv **PORT MAP** (e=>, s => s1);
u2:inv **PORT MAP** (e=>, s => s1);

2.12 - PACOTES (PACKAGE)

O pacote ou PACKAGE é uma coleção de tipos, constantes, subprogramas, etc. Esta é a maneira de agrupar elementos relacionados. Os pacotes estão divididos em duas partes, a declaração e corpo, onde o corpo contém definições de procedimentos e funções que podem ser omitidos, se não há nenhum desses elementos para declarar. A forma geral para a declaração de um pacote é:

PACKAGE nome **IS**
declarações
END [**PACKAGE**] [nome];

```
PACKAGE BODY nome IS
declarações , subprogramas, etc.
END [PACKAGE BODY] [nome];
```

Uma vez declarado o pacote, os elementos podem ser referenciados através do nome do pacote. Pode-se referenciar apenas um elemento ou todos, bastando declarar no programa principal quais os elementos do pacote estarão disponíveis. A seguir, um exemplo utilizando **PACKAGE**.

```
PACKAGE cpu IS
        SUBTYPE byte IS bit_vector (7 DOWNTO 0);
        FUNCTION inc (valor: interger) RETURN interger;
END cpu;

PACKAGE BODY cpu IS
    FUNCTION inc (valor: interger) RETURN interger IS
    VARIABLE result: integer;
    BEGIN
            Result <= valor +1;
    RETURN result;
   END inc;
END cpu;
```

Ex.: Utilização do pacote declarado.

```
   VARIABLE reg: work.cpu.byte;
    pc <= work.cpu.inc(pc);
```

Ou ainda: quando é declarado o uso do pacote no início do programa, não é necessário apontar o caminho de referência.

```
Ex.:  USE work.cpu.ALL;
       ...
      VARIABLE reg: byte;
       pc <= inc(pc);
```

2.13- CONFIGURAÇÃO (CONFIGURATION)

A configuração é um bloco especial do VHDL que permite especificar os mínimos enlaces “componente-entidade”, através da parte declarativa de uma arquitetura. A forma geral para a declaração de uma configuração é:

CONFIGURATION nome **OF** nome_entidade **IS**
{ sentenças | atributos }
END [**CONFIGURATION**] [nome];

Outros elementos podem ser declarados na configuração, mas não são discutidos neste livro.

2.14 - PROCEDIMENTOS E FUNÇÕES.

Procedimentos e funções podem ser declarados nas partes declarativas de arquiteturas, blocos, pacote, etc. A forma geral para a declaração de procedimentos e funções é:

PROCEDURE nome [(parâmetros)] **IS**
[declarações]
BEGIN
[sentenças seqüenciais]
END [**PROCEDURE**] [nome];

FUNCTION nome [(parâmetros)] **RETURN** tipo **IS**
[declarações]
BEGIN
[sentenças seqüenciais]
END [**FUNCTION**] [nome];

Com relação aos parâmetros nas funções, estes devem ser apenas do tipo **IN** (entrada) e podem ser **CONSTANT** e **SIGNAL**. Nos procedimentos, os parâmetros podem ser do tipo **CONSTANT, IN**, **OUT** e **INOUT,** sendo que **CONSTANT** é apenas do tipo **IN.**

Ex.: **PROCEDURE** inc_pc **IS**
BEGIN
next_PC := PC + 1;
END;

2.15 - EXECUÇÃO CONCORRENTE

A execução de instruções correntes é feita através de atribuições entre sinais, utilizando o operador <= . Para facilitar algumas atribuições complexas, o VHDL

possui alguns elementos de alto nível que são instruções condicionais, de seleção, e outras que auxiliam a implementação.

Atribuição condicional concorrente: WHEN...ELSE...

Toda expressão condicional descreve o hardware de forma concorrente. Deve incluir todas as possibilidades de variação de um sinal. A forma geral para declaração de uma atribuição condicional concorrente é:

```
sinal <= {forma_de_onda WHEN condição
         ELSE} forma_de_onda [ WHEN condição];

Ex.: s <= '0'  WHEN  a>b  ELSE
          '1'  WHEN  a<b  ELSE
          'X';
```

Atribuição com seleção concorrente: WITH...SELECT...WHEN

Este tipo de atribuição é semelhante a construções do *case* e do *switch* em Pascal e C. A atribuição é executada com base no valor da expressão em teste. A expressão deve sempre retornar um tipo discreto enumerado, inteiro ou um vetor de uma dimensão. A forma geral para declaração de uma atribuição com seleção concorrente é:

```
WITH expressão SELECT
sinal <= forma_de_onda WHEN  caso
         {, forma_de_onda WHEN  caso};

Ex.: WITH sel SELECT
       s <=  '1' WHEN  "00",
             '0' WHEN OTHERS;
```

Estes são os principais e mais utilizados métodos para atribuições concorrentes. As atribuições em blocos concorrentes (**BLOCK**) não são discutidas neste livro.

2.16 - EXECUÇÃO SEQÜENCIAL

Processo (PROCESS)

Os processos são, por definição, concorrentes, mas o conteúdo de cada processo é executado de forma seqüencial. A forma geral para declaração de um processo é:

```
[id_proc:] [POSTPONED] PROCESS [(lista sensível)] [IS]
   Declarações
   BEGIN
         Instruções seqüenciais
END [POSTPONED] PROCESS [id_proc];
```

Onde id_proc é um rótulo ou uma etiqueta opcional, lista sensível é uma lista de sinais separados por vírgula que estão entre parênteses. Quando ocorre um evento de mudança em qualquer sinal da lista sensitiva, o processo é executado. Processos podem não possuir lista de sensibilidade, mas devem ser controlados pela sentença **WAIT**.

Na parte de *declarações*, podem ser criadas variáveis, tipos, subprogramas, atributos, etc. Não é possível declarar sinais. Declarar um processo e utilizar o elemento **POSTPONED** significa que este processo só será executado após a execução dos outros processos existentes na descrição de um determinado circuito digital.

```
Ex.: PROCESS (a,b);
     BEGIN
          a<= a AND b;
     END PROCESS;
```

Comando condicional seqüencial: IF...THEN...ELSE

A partir do resultado de uma expressão booleana, decide-se qual sentença será executada. A forma geral para descrever um comando condicional seqüencial é:

```
[id_if:] IF condição THEN
            sentença
         {ELSIF condição THEN      sentença}
         {ELSE     sentença}
          END IF [id_if];
```

Note que há a possibilidade de utilizar *IFs* aninhados, utilizando a palavra reservada ELSIF, sendo que ELSIF não pede um END IF como finalização de comando.

```
Ex.: IF a>b THEN
        c <= a;
     ELSIF a<b THEN
```

```
    c <= b;
  ELSE
    c <= a AND b;
  END IF;
```

Comando de seleção seqüencial : CASE

Esta estrutura permite executar uma ou outra instrução, dependendo do resultado de uma expressão, esta deve ser do tipo discreto ou uma matriz de caracteres de uma dimensão. A forma geral para descrever um comando de seleção seqüencial é:

```
[id_case:] CASE expressão IS
                WHEN caso => sentença;
                {WHEN caso => sentenças;}
                [WHEN OTHERS => sentenças]
          END CASE; [id_case]
```

O comando case exige que todas as possibilidades de casos sejam esgotadas. A palavra reservada **OTHERS** satisfaz todos os casos não previstos anteriormente.

```
Ex.: CASE semaforo IS
       WHEN "00" => sinal <= "verde";
       WHEN "01" => sinal <= "amarelo";
       WHEN "10" => sinal <= "vermelho";
       WHEN OTHERS => sinal <= "Erro";
  END CASE;
```

Comandos de laço: FOR e WHILE

A definição dos laços **FOR** e **WHILE** é a mesma de outras linguagens, onde determinado trecho seqüencial é executado o número de vezes determinado em um intervalo. No caso do comando **FOR** ou **WHILE**, a sentença que irá repetir deve estar entre as palavras reservadas **LOOP**...**END**. A forma geral para descrever os comandos de laço **FOR** e **WHILE** é:

Comando **FOR**

```
[id_for:] FOR identificador IN intervalo LOOP
            sentença
          END LOOP [id_for];
```

Comando **WHILE**

```
[id_while:] WHILE condição LOOP
              sentença
            END LOOP [id_while];
```

Os comandos de laço possuem métodos auxiliares para a interrupção usando-se os comandos: **NEXT** e **EXIT**. O comando **NEXT** detém a execução do laço atual e passa para a próxima execução, enquanto que o comando **EXIT** finaliza o laço que está executando.

Ex.: **FOR** I **IN** 0 **TO** 9 **LOOP**
A <= i+1;
EXIT WHEN a = 6;
END LOOP;

CAPÍTULO III - CIRCUITOS COMBINACIONAIS

Neste capítulo, apresentam-se conceitos e a correspondente descrição em VHDL de circuitos combinacionais como: portas lógicas básicas do VHDL'93, multiplexadores, demultiplexadores, somadores, subtratores, multiplicadores, divisores, decodificadores, codificadores e comparadores. Para cada circuito implementado, são gerados dados estatísticos de sua implementação em FPGA. As estatísticas geradas são: *espaciais*, como número de flip-flops, de LUTs de 4 e 3 entradas e de CLBs utilizadas; e *temporais*, relacionada aos atrasos para gerar a saída desejada de um determinado circuito. Este tempo é o resultado da soma do tempo de lógica mais o de roteamento do circuito digital. As estatísticas geradas são comparadas e analisadas.

3.1 - CIRCUITOS COMBINACIONAIS

Um circuito combinacional é constituído por um conjunto de portas lógicas que determinam os valores das saídas diretamente, a partir dos valores atuais das entradas. Pode-se dizer que um circuito combinacional realiza uma operação de processamento de informação, a qual pode ser especificada por meio de um conjunto de equações booleanas. No caso, cada combinação de valores de entrada pode ser vista como uma informação diferente, e cada conjunto de valores de saída representa o resultado da operação. Na figura 3.1, observa-se a representação geral de um circuito combinacional com *n* entradas que passam por uma lógica combinacional até gerar *m* saídas.

Figura 3.1 - Diagrama genérico de um circuito combinacional.

O objetivo da análise de um circuito combinacional é determinar seu comportamento. Então, dado o diagrama de um circuito, deseja-se encontrar as equações que descrevem suas saídas, a partir de um conjunto de entradas. Uma vez encontradas tais equações, pode-se obter a tabela verdade, caso esta seja necessária.

É importante certificar-se de que o circuito é combinacional e não seqüencial. Um modo prático é verificar se existe algum caminho (ou ligação) entre a saída e a entrada do circuito. Caso não exista, o circuito é combinacional.

Os circuitos combinacionais são responsáveis por operações lógicas e aritméticas dentro de um sistema digital. Além das operações lógicas e aritméticas como adição, subtração e complementação, existem ainda outras funções necessárias para a realização de conexões entre os diversos operadores. Dentre estas funções, estão

a multiplexação e a decodificação. Os elementos que realizam estas duas últimas operações são denominados multiplexadores e decodificadores.

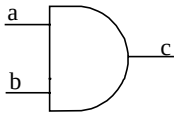
A maioria dos circuitos e sistemas digitais são implementados usando circuitos básicos, conhecidos como portas lógicas. Estes circuitos simples e básicos são estudados na seção 3.2.

3.2 - PORTAS LÓGICAS BÁSICAS

As portas lógicas básicas estão presentes em todos os circuitos lógicos, possuem estrutura e funcionamento simplificados e podem ser classificadas como portas: AND, OR, NOT, NAND, NOR e XOR e XNOR, adotando o padrão VHDL'93.

3.2.1 - PORTA AND

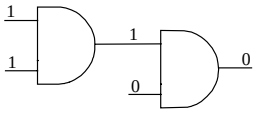
A função lógica da porta AND é combinar dois ou mais sinais de entrada de forma equivalente a um circuito em série, para produzir um único sinal de saída. A porta AND produz uma saída **1**, se todos os sinais de entrada forem **1.** Caso qualquer um dos sinais de entrada for **0**, a porta AND produzirá um sinal de saída igual a **0**. Na figura 3.2, visualiza-se a representação de uma porta AND de 2 entradas, a tabela verdade e um exemplo de funcionamento.



(a)

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

(b)



(c)

Figura 3.2 – (a) representação, (b) tabela verdade e (c) exemplo de funcionamento de uma porta AND de duas entradas.

O exemplo de funcionamento da figura 3.2 (c), demostra duas portas AND, onde a saída de uma alimenta a entrada da outra. A resposta de cada operação AND está relacionada com a tabela verdade, figura 3.2 (b).

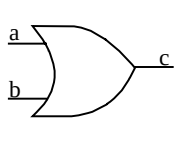
A linguagem VHDL tem como uma de suas primitivas a porta AND, assim como todas as outras portas lógicas básicas, facilitando a implementação de circuitos digitais. O código VHDL de uma porta AND de duas entradas, *a* e *b* e uma saída *c* é:

Porta AND de duas entradas de 1 bit

```
library ieee;
use ieee.std_logic_1164.all;

-- declaração da entidade
entity and2 is
   port (a,b: in std_logic;
         c: out std_logic);
end and2;

-- declaração da arquitetura
architecture arch_and2 of and2 is
begin
    c <= a and b;
end arch_and2;
```

3.2.2 - PORTA OR

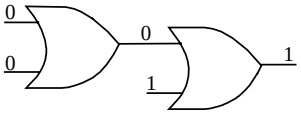
A função lógica de uma porta OR consiste em combinar dois ou mais sinais de entrada, de forma equivalente a um circuito em paralelo, para produzir um único sinal de saída. A porta OR produz uma saída **1**, se qualquer um dos sinais de entrada for igual a **1,** ou produzirá um sinal de saída igual a **0** se todos os sinais de entrada forem **0**. A figura 3.3, mostra a representação de uma porta OR de 2 entradas, a tabela verdade e um exemplo de funcionamento.



(a)

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

(b)



(c)

Figura 3.3 – (a) representação, (b) tabela verdade e (c) exemplo de funcionamento de uma porta OR de duas entradas.

O código VHDL de uma porta OR de duas entradas *a* e *b* e uma saída *c* utilizando as primitivas que o VHDL'93 oferece é:

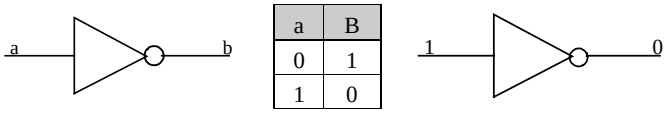
Porta OR de duas entradas de 1 bit

```
library ieee;
use ieee.std_logic_1164.all;
entity or2 is
   port (a,b: in std_logic;
         c: out std_logic);
end or2;
architecture arch_or2 of or2 is
```

```
begin
    c <= a or b;
end arch_or2;
```

3.2.3 - PORTA NOT

A função lógica implementada por uma porta NOT é inverter o sinal de entrada, ou seja, se o sinal de entrada for **0** ela produz uma saída **1**, se a entrada for **1** produz uma saída **0**. A representação de uma porta NOT, a tabela verdade e um exemplo de funcionamento é melhor visualizada na figura 3.4.



a	B
0	1
1	0

(a) **(b)** **(c)**

Figura 3.4 – (a) representação, (b) tabela verdade e (c) exemplo de funcionamento de uma porta NOT (c).

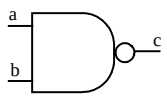
O código VHDL de uma porta NOT utilizando as primitivas que o VHDL'93 oferece é:

Porta NOT de 1 bit

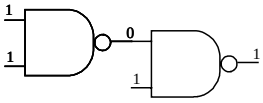
```
library ieee;
use ieee.std_logic_1164.all;
entity inv is
   port (a: in std_logic;
         c: out std_logic);
end inv;
architecture arch_inv of inv is
begin
    c <= not a;
end arch_inv;
```

3.2.4 - PORTA NAND

A função lógica implementada por uma porta NAND é eqüivalente a uma porta AND seguida de uma porta NOT, isto é, ela produz uma saída que é o inverso da saída produzida pela porta AND. Na figura 3.5 tem-se a representação de uma porta NAND de duas entradas *a* e *b* e uma saída *c*, a tabela verdade e um exemplo de funcionamento.



a	b	c
0	0	1
0	1	1
1	0	1
1	1	0



(a) **(b)** **(c)**

Figura 3.5 – (a) Representação, (b) tabela verdade e (c) exemplo de funcionamento de uma porta NAND de duas entradas.

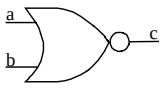
O código VHDL de uma porta NAND de duas entradas *a* e *b* e uma saída *c* utilizando as primitivas que o VHDL'93 oferece é:

Porta NAND de duas entradas de 1 bit

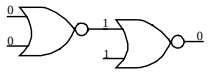
```
library ieee;
use ieee.std_logic_1164.all;
entity nand2 is
   port (a,b: in std_logic;
         c: out std_logic);
end nand2;
architecture arch_nand2 of nand2 is
begin
   c <= a nand b;
end arch_nand2;
```

3.2.5 - PORTA NOR

A função lógica implementada por uma porta NOR é eqüivalente a uma porta OR seguida por uma porta NOT. Isto é, ela produz uma saída que é o inverso da saída produzida pela porta OR. Na figura 3.6, visualiza-se a representação de uma porta NOR de duas entradas *a* e *b* e uma saída *c*, a tabela verdade e um exemplo de funcionamento.



a	**b**	**c**
0	0	1
0	1	0
1	0	0
1	1	0



(a) **(b)** **(c)**

Figura 3.6 – (a) representação, (b) tabela verdade e (c) exemplo de funcionamento respectivamente de uma porta NOR de duas entradas.

O código VHDL de uma porta NOR de duas entradas *a* e *b* e uma saída *c* utilizando as primitivas que o VHDL'93 oferece é:

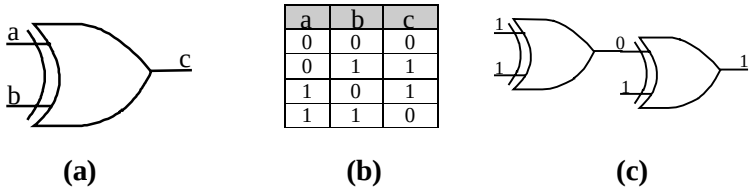
Porta NOR de duas entradas de 1 bit

```
library ieee;
```

```
use ieee.std_logic_1164.all;
entity nor2 is
   port (a,b: in std_logic;
         c: out std_logic);
end nor2;
architecture arch_nor2 of nor2 is
begin
    c <= a nor b;
end arch_nor2;
```

3.2.6 - PORTA XOR

A função lógica implementada por uma porta XOR ou também conhecida como "ou exclusivo" é comparar os bits e produzir saída **0** quando todos os bits de entrada são iguais, e saída **1** quando pelo menos um dos bits de entrada é diferente dos demais. Na figura 3.7, tem a representação de uma porta XOR de duas entradas *a* e *b* e uma saída *c*, a tabela verdade e um exemplo de funcionamento.



a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Figura 3.7 – (a) representação, (b) tabela verdade e (c) exemplo de funcionamento respectivamente de uma porta XOR de duas entradas.

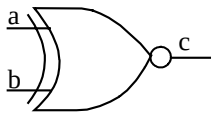
O código VHDL de uma porta XOR de duas entradas *a* e *b* e uma saída *c* utilizando as primitivas que o VHDL'93 oferece é:

Porta XOR de duas entradas de 1 bit

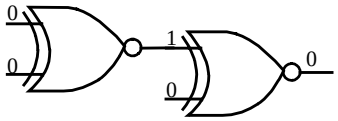
```
library ieee;
use ieee.std_logic_1164.all;

entity xor2 is
   port (a,b: in std_logic;
         c: out std_logic);
end xor2;

architecture arch_xor2 of xor2 is
begin
    c <= a xor b;
end arch_xor2;
```

3.2.7 - PORTA XNOR

A função lógica da porta XNOR é eqüivalente a uma porta XOR seguida por uma porta NOT. Isto é, ela produz uma saída que é o inverso da saída produzida pela porta XOR. A representação de uma porta XNOR de duas entradas, a tabela verdade e um exemplo de funcionamento, pode ser visualizado na figura 3.8.



a	b	c
0	0	1
0	1	0
1	0	0
1	1	1



(a) **(b)** **(c)**

Figura 3.8 – (a) representação, (b) tabela verdade e (c) exemplo de funcionamento respectivamente de uma porta XNOR de duas entradas.

O código VHDL de uma porta XNOR de duas entradas *a* e *b* e uma saída *c* utilizando as primitivas que o VHDL'93 oferece é:

Porta XNOR de duas entradas de 1 bit

```
library ieee;
use ieee.std_logic_1164.all;
entity xnor2 is
   port (a,b: in std_logic;
         c: out std_logic);
end xnor2;
architecture arch_xnor2 of xnor2 is
begin
    c <= a xnor b;
end arch_xnor2;
```

3.2.8 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA - PORTA LÓGICAS BÁSICAS.

Observa-se nos resultados da tabela 3.1 que as portas lógicas são implementadas basicamente com um único elemento, uma única LUT de 4 entradas, utilizando assim parte de uma CLB, com exceção da porta lógica NOT, onde não foi necessário a utilização de nenhum recurso listado na tabela 3.1.

Isto acontece, porque a lógica de inversão da porta NOT pode ser implementada por inversor que compõe uma das IOBs, não tendo a necessidade de utilizar nenhum outro recurso do FPGA. Isto justifica o atraso total de execução do circuito ser menor que das outras portas lógica (tabela 3.2).

Porta Lógica	FF	LUTs 4	LUTs 3	CLBs

AND	0	1	0	1
OR	0	1	0	1
NOT	0	0	0	0
NAND	0	1	0	1
NOR	0	1	0	1
XOR	0	1	0	1
XNOR	0	1	0	1

Tabela 3.1 – Recursos utilizados do FPGAs – Portas lógicas básicas

3.2.9 - TEMPORIZAÇÃO DAS PORTAS LÓGICAS BÁSICAS

Os tempos gerados representam o tempo levado para executar cada operação lógica listada na tabela 3.2. Esse tempo pode ser um dos elementos para medir o desempenho de um circuito. Neste caso, está-se medindo o tempo de execução de operações lógicas básicas. Esse tempo é dado em: Tempo de Lógica (TL), o tempo que leva efetivamente para implementar a lógica do circuito; Tempo de Roteamento (TR), o tempo gasto no roteamento implementado pelo software. Sendo que a soma do TL com TR resulta no Atraso Total (AT).

Porta Lógica	TL	TR	AT
AND	6.899ns	2.451ns	9.350ns
OR	6.899ns	2.472ns	9.371ns
NOT	5.709ns	0.194ns	5.903ns
NAND	6.899ns	2.472ns	9.371ns
NOR	6.899ns	2.472ns	9.371ns
XOR	6.899ns	2.472ns	9.371ns
XNOR	6.899ns	2.472ns	9.371ns

Tabela 3.2 – Temporização das portas lógicas básicas.

É interessante perceber que quase todas a funções lógicas possuem os mesmos tempos. Isso acontece, porque são circuitos básicos e simples, o tempo de roteamento deste circuitos é quase mínimo.

3.3 – MULTIPLEXADORES E DEMULTIPLEXDORES

A figura 3.9 (a) representa um circuito multiplexador 2x1 que possui algumas características como: a presença de dois sinais lógicos de entrada *e1* e *e2*, um de saída *s* e um de seleção *sel*. A função lógica do multiplexador é simplesmente selecionar um destes sinais de entrada *e1* e *e2* através do sinal de seleção *sel*, atribuindo à saída *s* o sinal de entrada desejado. Portanto, podemos definir um multiplexador como um circuito que possui 2^n entradas, *n* sinais de controle e uma saída.

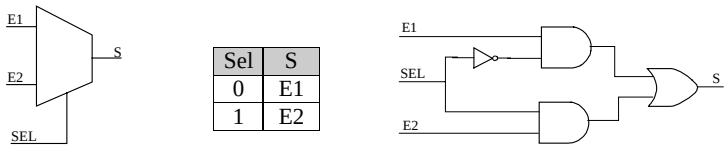


Figura 3.9 – (a) representação, (b) tabela verdade e (c) descrição com portas lógicas.

Tem-se também os demultiplexadores, que implementam a lógica inversa dos multiplexadores, circuitos com um único sinal de entrada, 2^n saídas e n sinais de controle, que tem a função de selecionar qual deve ser a saída desejada.

O VHDL permite descrever um mesmo circuito em inúmeras formas, isto é, um circuito *X* pode ser descrito de várias maneiras, mantendo a sua função lógica original.

Seguindo e usando essas possibilidades a maioria dos circuitos descritos neste livro apresentam mais de uma descrição. A seguir apresentam-se quatro diferentes descrições em VHDL de um mesmo circuito multiplexador incluindo também suas respectivas estatísticas de desempenho espacial e temporal em um FPGA.

As implementações do circuito multiplexador 2x1 utiliza quatro diferentes estruturas disponíveis na linguagem VHDL, que são:

- Execução concorrente com WITH...SELECT.
- Execução seqüencial com IF...THEN...ELSE.
- Execução seqüencial com CASE
- Implementação com portas lógicas.

Código VHDL de um multiplexador 2x1 usando o comando de execução concorrente WITH SELECT:

Multiplexador 2x1

```
library ieee;
use ieee.std_logic_1164.all;

entity mult2x1 is
   port ( e1,e2,sel: in std_logic;
                  s: out std_logic);
end mult2x1;
```

```
architecture arch_mult2x1 of mult2x1 is
begin
     with sel select
            s <= e1 when '0',
                 e2 when others;
end arch_mult2x1;
```

Código VHDL de multiplexador 2x1 usando o comando de execução seqüencial IF...THEN...ELSE:

Multiplexador 2x1

```
library ieee;
use ieee.std_logic_1164.all;
entity mult2x1 is
   port ( e1,e2,sel: in std_logic;
                    s: out std_logic);
end mult2x1;

architecture arch_mult2x1 of mult2x1 is
begin
      process (e1,e2,sel)
      begin
         if sel='0' then
            s <= e1;
         else
            s <= e2;
         end if;
      end process;
end arch_mult2x1;
```

Código VHDL de multiplexador 2x1 usando o comando de execução seqüencial CASE:

Multiplexador 2x1

```
library ieee;
use ieee.std_logic_1164.all;
entity mult2x1 is
   port ( e1,e2,sel: in std_logic;
                    s: out std_logic);
end mult2x1;
architecture arch_mult2x1 of mult2x1 is
begin
      process (e1,e2,sel)
      begin
            case sel is
             when '0' => s <= e1;
             when others => s <= e2;
            end case;
      end process;
end arch_mult2x1;
```

Código VHDL de multiplexador 2x1 usando portas lógicas básicas:

Multiplexador 2x1

```
library ieee;
use ieee.std_logic_1164.all;

entity mult2x1 is
   port ( e1,e2,sel: in std_logic;
                  s: out std_logic);
end mult2x1;

architecture arch_mult2x1 of mult2x1 is
begin
        s <= (e1 and not (sel)) or (e2 and sel);
end arch_mult2x1;
```

Observa-se que usando os comandos IF-THEN-ELSE e CASE precisa-se definir um processo, pois são comando seqüenciais e são executados a partir de um evento predefinido.

3.3.1 – ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA – MULTIPLEXADOR 2X1.

Multiplexador	FF	LUTs 4	LUTs 3	CLBs
With Select	0	1	0	1
If-Then-Else	0	1	0	1
Case	0	1	0	1
Porta Lógica	0	1	0	1

Tabela 3.3 – Recursos utilizados do FPGAs – Multiplexador 2x1

Observa-se que apesar dos códigos VHDL dos multiplexadores 2x1 serem diferentes, do ponto de vista das estatísticas de utilização de recursos do FPGA (tabela 3.3) e dos tempos de funcionamento (tabela 3.4), são equivalentes, pois nem a estatística de elementos utilizados para implementação do circuito e a temporização foram alterados.

3.3.2 – TEMPORIZAÇÃO DOS MULTIPLEXADORES.

Multiplexador	TL	TR	AT
With Select	6,899ns	3,336ns	10,235ns
If-Then-Else	6,899ns	3,336ns	10,235ns

Case	6,899ns	3,336ns	10,235ns
Porta Lógica	6,899ns	3,336ns	10,235ns

Tabela 3.4 – Temporização dos multiplexadores 2x1.

É interessante perceber que o tempo de lógica do multiplexador é igual à de algumas portas lógicas básicas (tabela 3.2). Isso mostra que do ponto de vista de implementação em FPGA, um multiplexador 2x1, seja qual for a sua implementação, também pode ser considerado como um circuito básico e simples.

Já quando se observa o tempo de roteamento, percebe-se que é maior que o tempo de uma porta lógica básica, pois um multiplexador possui uma maior complexidade quando comparado a uma simples porta lógica, mas não deixa de ser um circuito simples. Esta diferença no tempo de roteamento, faz com que o atraso total seja maior que das portas lógicas básicas.

3.4 - DECODIFICADORES

O tipo mais comum de decodificador é o chamado decodificador binário, possui *n* entradas e 2^n saídas, onde apenas uma estará ativa para cada combinação possível das suas entradas. Na tabela 3.5, observa-se a tabela verdade de um decodificador 3x8.

Entradas			**Saídas**							
X2	**X1**	**X0**	**S7**	**S6**	**S5**	**S4**	**S3**	**S2**	**S1**	**S0**
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Tabela 3.5 – Tabela verdade de um decodificador 3x8.

Na figura 3.10, visualiza-se a representação e a implementação com portas lógicas básicas de um decodificador 3x8. As entradas são indicadas com *x(0)*, *x(1)* e *x(2)* e as saídas com *s(0)* a *s(7)*.

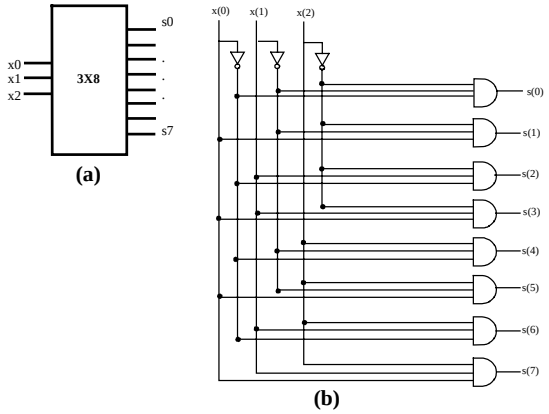


Figura 3.10 – (a) Representação e (b) implementação com portas lógicas básicas de um decodificador 3x8 (b).

A seguir, similar ao multiplexador, apresentam-se quatro diferentes descrições em VHDL de um circuito decodificador e suas estatísticas de desempenho espacial e temporal.

Código VHDL de decodificador 3x8 usando o comando de execução concorrente WITH...SELECT:

Decodificador 3x8

```
library ieee;
use ieee.std_logic_1164.all;
entity dec3x8 is
   port ( x: in std_logic_vector (2 downto 0);
          s:  out std_logic_vector (7 downto 0));
end dec3x8;
architecture arch_dec3x8 of dec3x8 is
begin
    with x select
    s <=  "00000001" when "000",
          "00000010" when "001",
          "00000100" when "010",
          "00001000" when "011",
          "00010000" when "100",
          "00100000" when "101",
          "01000000" when "110",
          "10000000" when "111",
          "ZZZZZZZZ" when others;
end arch_dec3x8;
```

Código VHDL de decodificador 3x8 usando o comando de execução seqüencial IF...THEN...ELSE:

Decodificador 3x8

```
library ieee;
use ieee.std_logic_1164.all;
entity dec3x8 is
   port (s: out std_logic_vector (7 downto 0);
           x: in std_logic_vector (2 downto 0));
end dec3x8;
architecture arch_dec3x8 of dec3x8 is
begin
process (x)
   begin
            if      (x = "000") then s <= "00000001";
            elsif (x = "001") then s <= "00000010";
            elsif (x = "010") then s <= "00000100";
            elsif (x = "011") then s <= "00001000";
            elsif (x = "100") then s <= "00010000";
            elsif (x = "101") then s <= "00100000";
            elsif (x = "110") then s <= "01000000";
            elsif (x = "111") then s <= "10000000";
            else s <= "ZZZZZZZZ";
       end if;
   end process;
end arch_dec3x8 ;
```

Código VHDL de decodificador 3x8 usando o comando de execução seqüencial CASE:

Decodificador 3x8

```
library ieee;
use ieee.std_logic_1164.all;
entity dec3x8 is
   port (     x: in std_logic_vector (2 downto 0);
              s: out std_logic_vector (7 downto 0));
end dec3x8;
architecture arch_dec3x8 of dec3x8 is
begin
    process (x)
    begin
        case x is
            when "000" => s <= "00000001";
            when "001" => s <= "00000010";
            when "010" => s <= "00000100";
            when "011" => s <= "00001000";
            when "100" => s <= "00010000";
            when "101" => s <= "00100000";
            when "110" => s <= "01000000";
            when "111" => s <= "10000000";
            when others => s <= "ZZZZZZZZ";
        end case;
    end process;
end arch_dec3x8;
```

Código VHDL de decodificador 3x8 usando o comando de execução concorrente portas lógicas básicas:

Decodificador 3x8

```
   library ieee;
use ieee.std_logic_1164.all;
entity  dec3x8 is
   port (    x: in std_logic_vector (2 downto 0);
             s: out std_logic_vector (7 downto 0));
end dec3x8;
architecture arch_dec3x8 of dec3x8 is
begin
        s(0) <= not (x(2))  and not(x(1)) and  not(x(0));
        s(1) <= not(x(2))   and not(x(1)) and       x(0);
        s(2) <= not(x(2))   and      x(1)  and  not(x(0));
        s(3) <= not(x(2))   and      x(1)  and       x(0);
        s(4) <= x(2)        and not(x(1)) and  not(x(0));
        s(5) <= x(2)        and not(x(1)) and       x(0);
        s(6) <= x(2)        and      x(1)  and  not(x(0));
        s(7) <= x(2)        and      x(1)  and       x(0);
end arch_dec3x8;
```

3.4.1 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA – DECODIFICADOR 3X8.

Decodificador	FF	LUTs 4	LUTs 3	CLBs
If-Then-Else	0	8	0	4
With-Select	0	8	0	4
Case	0	8	0	4
Porta Lógica	0	8	0	4

Tabela 3.6 – Recursos utilizados do FPGAs – Decodificador 3x8

Nota-se que um decodificador 3x8 é um circuito mais complexo que os anteriores. Um decodificador utiliza muito mais recursos do FPGA do que circuitos mais simples como portas lógicas básicas e multiplexadores e, mesmo sendo mais complexo, o atraso total destes circuitos não são muito maiores que os tempos de circuitos básicos.

3.4.2 – TEMPORIZAÇÃO DOS DECODIFICADORES.

Decodificador	TL	TR	AT
If-Then-Else	6.899ns	5.079ns	11.978ns
With –Select	6.899ns	5.079ns	11.978ns
Case	6.899ns	5.079ns	11.978ns
Porta Lógica	6.899ns	7.872ns	14.771ns

Tabela 3.7 - Temporização dos decodificador 3x8 .

Neste caso, a implementação de um decodificador 3x8 com os comandos IF...THEN...ELSE, WITH...SELECT e CASE tiveram um melhor desempenho que a

implementação do mesmo circuito com portas lógicas básicas. Nota-se que o tempo de lógica é igual para todos os decodificadores implementados, mas isso não acontece com o tempo de roteamento, pois o circuito implementado com portas básicas teve um pior desempenho.

É interessante perceber que o decodificador implementado apenas com portas lógicas teve o seu atraso total prejudicado em aproximadamente 18.9% quando comparado ao atraso total dos outros decodificadores. Esses dados são importantes para demonstrar a influência do tipo de implementação no desempenho final de um circuito. Observa-se que nem sempre a implementação com portas lógicas é a mais otimizada. Neste caso, esta implementação não obteve as melhores estatísticas de desempenho.

3.5 – CODIFICADORES

Os codificadores implementam a função inversa dos decodificadores, isto é, um codificador binário possui 2^n entradas, codificadas em saídas de n bits. Na figura 3.11, nota-se a representação de um codificador 8x3.

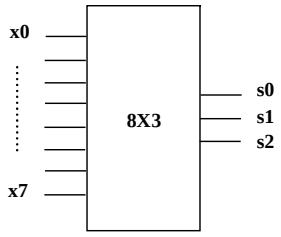


Figura 3.11 – Representação de um codificador 8x3.

Na tabela 3.8 observa-se a tabela verdade de um codificador com 8 entradas e 3 saídas. As entradas são indicadas como *x0* a *x7* e as saídas como *s0*, *s1* e *s2*.

Entradas								Saídas		
X7	**X6**	**X5**	**X4**	**X3**	**X2**	**X1**	**X0**	**S2**	**S1**	**S0**
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Tabela 3.8 - Tabela verdade de um codificador 8x3 .

A seguir, quatro implementações em VHDL do mesmo circuito codificador 8x3 e suas respectivas estatísticas de desempenho espacial e temporal.

Código VHDL do codificador 8x3 usando o comando de execução concorrente WITH...SELECT:

Codificador 8x3

```
library ieee;
use ieee.std_logic_1164.all;
entity cod8x3 is
   port (  x: in std_logic_vector (7 downto 0);
           s: out std_logic_vector (2 downto 0));
end cod8x3;
architecture arch_cod8x3 of cod8x3 is
begin
   with x select
    s <=   "000" when "00000001",
           "001" when "00000010",
           "010" when "00000100",
           "011" when "00001000",
           "100" when "00010000",
           "101" when "00100000",
           "110" when "01000000",
           "111" when "10000000",
           "ZZZ" when others;
end arch_cod8x3;
```

Código VHDL do codificador 8x3 usando o comando de execução seqüencial IF...THEN...ELSE:

Codificador 8x3

```
library ieee;
use ieee.std_logic_1164.all;
entity cod8x3 is
   port (  x: in std_logic_vector (7 downto 0);
           s: out std_logic_vector (2 downto 0));
end cod8x3;
architecture arch_cod8x3 of cod8x3 is
begin
    process (x)
    begin
            if     (x= "00000001") then s <= "000";
            elsif (x= "00000010") then s <= "001";
            elsif (x= "00000100") then s <= "010";
            elsif (x= "00001000") then s <= "011";
            elsif (x= "00010000") then s <= "100";
            elsif (x= "00100000") then s <= "101";
            elsif (x= "01000000") then s <= "110";
            elsif (x= "10000000") then s <= "111";
            else   s <= "ZZZ";
      end if;
```

```
    end process;
end arch_cod8x3;
```

Código VHDL do codificador 8x3 usando o comando de execução seqüencial CASE:

Codificador 8x3

```
library ieee;
use ieee.std_logic_1164.all;
entity cod8x3 is
   port (  x: in std_logic_vector (7 downto 0);
           s: out std_logic_vector (2 downto 0));
end cod8x3;
architecture arch_cod8x3 of cod8x3 is
begin
    process (x)
    begin
          case x is
             when "00000001" => s <= "000";
             when "00000010" => s <= "001";
             when "00000100" => s <= "010";
             when "00001000" => s <= "011";
             when "00010000" => s <= "100";
             when "00100000" => s <= "101";
             when "01000000" => s <= "110";
             when "10000000" => s <= "111";
             when others => s <= "ZZZ";
         end case;
end process;
end arch_cod8x3;
```

Código VHDL de codificador 8x3 usando portas lógicas básicas:

Codificador 8x3

```
library ieee;
use ieee.std_logic_1164.all;
entity cod8x3 is
   port (  x: in std_logic_vector (7 downto 0);
           s: out std_logic_vector (2 downto 0));
end cod8x3;
architecture arch_cod8x3 of cod8x3 is
begin
       s(2)<= x(4) or x(5) or x(6) or x(7) or x(0);
       s(1)<= x(2) or x(3) or x(6) or x(7) or x(0);
       s(0)<= x(1) or x(3) or x(5) or x(7) or x(0);
end arch_cod8x3;
```

3.5.1 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA E TEMPORIZAÇÃO DO CIRCUITO CODIFICADOR 8X3.

Codificador	FF	LUTs 4	LUTs 3	CLBs
If-Then-Else	0	3	0	2
With -Select	0	3	0	2
Case	0	3	0	2
Porta Lógica	0	3	0	2

Tabela 3.9 – Recursos utilizados do FPGAs – Multiplexador 2x1

Codificador	TL	TR	AT
If-Then-Else	6.899ns	4.222ns	11.121ns
With -Select	6.899ns	4.222ns	11.121ns
Case	6.899ns	4.222ns	11.121ns
Porta Lógica	6.899ns	4.222ns	11.121ns

Tabela 3.10 – Temporização dos codificadores 8x3

O codificador é um circuito mais simples que um decodificador percebe-se observando o número menor de componentes utilizados para implementa-lo e o menor tempo de atraso do seu funcionamento.

Neste caso a implementação do circuito codificador utilizando portas lógicas básicas não alterou o desempenho final do circuito implementado. Isto acontece pois o circuito codificador implementado não é complexo, utilizando apenas 2 CLBs. Já em um circuito que utilizam vários componentes e possui um roteamento mais complexo é possível visualizar esta diferença no desempenho final do circuito implementado.

3.6 - CIRCUITOS COMBINACIONAIS ARITMÉTICOS

Os circuitos combinacionais aritméticos são encontrados em todos os sistemas digitais que realizam operações lógicas e aritméticas. Existem circuitos aritméticos de baixa complexidade como, somadores, subtratores e outros. Existem também circuitos aritméticos que possuem maior complexidade, como operações com ponto flutuante.

Nesta seção é discutido a descrição dos seguintes circuitos: comparadores simples e com múltiplas comparações, um somador completo de 1 e 4 bits, um subtrator completo de 4 bits , um multiplicador de 4 bits e um divisor também de 4 bits. Ao final tem-se estatísticas de desempenho e temporização que são comparadas e analisadas.

3.6.1 - COMPARADORES

Um comparador pode ser classificado como, comparador simples ou com múltiplas comparações. Independente do tipo de comparador a comparação sempre é

realizada bit a bit desde o menos significativo até o mais significativo ou vice-versa. No padrão VHDL'93 é possível realizar as comparações listadas na tabela 3.11.

Símbolo	Significado	Símbolo	Significado
=	Igual	>=	Maior Igual
!=	Diferente	<	Menor
>	Maior	<=	Menor Igual

Tabela 3.11 – Tabela de comparações possíveis.

A seguir apresenta-se três descrições VHDL, estas são: um comparador simples, um comparador simples utilizando portas lógicas e um comparador usando múltiplas comparações.

Código VHDL do comparador simples usando o comando de execução seqüencial IF...THEN...ELSE:

Comparador simples

```
library ieee;
use ieee.std_logic_1164.all;
entity comp_simp is
   port (   x , y : in std_logic_vector (3 downto 0);
            s : out std_logic);
end comp_simp;
architecture arch_comp_simp of comp_simp is
begin
    process (x,y)
    begin
        if x<y then
           s <='0';
        else
           s <='1';
        end if;
    end process;
end arch_comp_simp;
```

Código VHDL do comparador simples usando portas lógicas básicas:

Comparador simples

```
library ieee;
use ieee.std_logic_1164.all;
entity comp_simp is
   port (     x, y: in std_logic_vector (3 downto 0);
              s: out std_logic);
end comp_simp;
architecture arch_comp_simp of comp_simp is
begin
    process (x,y)
    variable aux : std_logic_vector(3 downto 0);
```

```
        variable vaium : std_logic;
    begin
            vaium := '0';
            for i in 0 to 3 loop
                aux(i) := (x(i) xor y(i) xor vaium);
                vaium := (y(i) and aux(i)) or (y(i) and vaium) or
               (vaium and aux(i));
            end loop;
    s <= not vaium;
    end process;
end arch_comp_simp;
```

Código VHDL de comparador com múltiplas comparações usando o comando de execução seqüencial IF...THEN...ELSE:

Comparador com múltiplas comparações

```
library ieee;
use ieee.std_logic_1164.all;

entity comp_mult is
   port (    x, y, z, w: in std_logic_vector (3 downto 0);
             s: out std_logic);
end comp_mult;

architecture arch_comp_mult of comp_mult is
begin
    process (x,y,z,w)
    begin
        if (x<=y and (z<=w or x=z)) then
          s <= '0';
        else
          s <= '1';
        end if;
    end process;
end arch_comp_mult;
```

3.6.2 – SOMADORES E SUBTRATORES

Os somadores e subtratores são circuitos combinatórios dedicados, que executam, respectivamente, as operações de adição e subtração. Estes circuitos podem fazer parte de uma ULA (Unidade Lógica Aritmética), a qual está contida em calculadoras eletrônicas, microprocessadores e outros sistemas digitais.

Há várias formas de implementar somadores, uma delas, bem conhecida, é o circuito “meio-somador”, que pode perfeitamente efetuar operações de 1 bit, mas em

caso de palavras maiores, falha por não levar em conta o estouro da adição anterior. Para resolver este problema precisa-se de um somador completo ou *full adder* que considera o estouro da adição anterior. Para isso são implementados sinais de controle chamados de "vem-um" ou *carry in* e "vai-um" ou *carry out*.

O subtrator obedece ao mesmo raciocínio do circuito somador mudando apenas o tipo de opereção a ser executada. A seguir são apresentadas descrições em VHDL de um somador completo de 1 e 4 bits e um subtrator completo de 4 bits.

A representação de um somador completo de 1 bit com portas lógicas pode ser visualizada na figura 3.12.

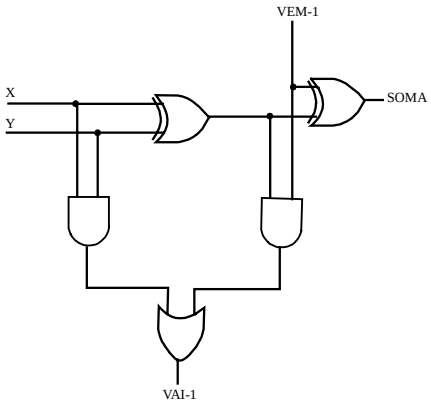


Figura 3.12 – Somador completo de 1 bit.

Código VHDL do somador completo de 1 bit usando portas lógicas básicas:

Somador completo de 1 bit

```
library ieee;
use ieee.std_logic_1164.all;
entity somador1bit is
   port ( cin, a, b: in std_logic;
          s, cout: out std_logic);
end somador1bit;

architecture arch_somador1bit of somador1bit is
begin
   s <= a xor b xor cin;
   cout <= (a and b) or (cin and a) or (cin and b);
end arch_somador1bit;
```

Código VHDL do somador completo de 4 bits usando portas lógicas básicas:

Somador completo de 4 bit

```
library ieee;
use ieee.std_logic_1164.all;
entity somador4bits is
   port ( cin: in std_logic;
          a,b: in std_logic_vector (3 downto 0);
          cout: out std_logic;
          s: out std_logic_vector (3 downto 0));
end somador4bits;
architecture arch_ somador4bits of somador4bits is
begin
       process(a, b, cin)
       variable soma: std_logic_vector(3 downto 0);
       variable       c: std_logic;
       begin
                c := cin;
                for i in 0 to 3 loop
                    soma(i) := a(i) xor b(i) xor c;
                       c := (a(i) and b(i)) or ((a(i) xor b(i)) and c);
                 end loop;
                 cout <= c;
                 s  <= soma;
       end process;
end arch_somador4bits;
```

Código VHDL de um subtrator completo de 4 bits usando portas lógicas básicas:

Subtrator completo de 4 bit

```
library ieee;
use ieee.std_logic_1164.all;
entity sub4bits is
   port (a,b: in std_logic_vector (3 downto 0);
            s: out std_logic_vector (3 downto 0));
end sub4bits;
architecture arch_ sub4bits of sub4bits is
begin
       process(a, b)
       variable result: std_logic_vector(3 downto 0);
       variable vaium: std_logic;
       begin
              vaium:='0';
              for i in 0 to 3 loop
                   result(i) := (a(i) xor b(i) xor vaium);
                   vaium := ( b(i) and result(i) ) or ( b(i) and vaium)
                            or (vaium and result(i) ) ;
              end loop ;
              s<=result;
       end process;
end arch_ sub4bits;
```

3.6.3 - MULTIPLICADORES

Um multiplicador é um circuito mais complexo em nível de lógica de funcionamento. Existem muitos algoritmos para a sua implementação. Neste caso, foi selecionado um algoritmo que talvez não seja o mais otimizado, mas é um algoritmo com passos simples e bem definidos, facilitando a compreensão do seu funcionamento. Este algoritmo pode ser implementado para multiplicadores de *n* bits.

Algoritmo do Multiplicador

```
Inicio:
  Contador de bits recebe o numero de bits a ser multiplicado
  Inicializa produto com zero
  Para i de 0 até o número de bits da multiplicação faça
        deslocar (produto)
        Vaium recebe bit mais significativo do multiplicador
         Se vaium igual a 1 (um) então
             Soma (produto,multiplicando)
         deslocar (multiplicador)
  Resultado recebe produto
Fim
```

A tabela 3.12 demonstra uma multiplicação usando o algoritmo proposto, onde o resultado final é obtido quando o contador de bits for 0. Neste caso está fazendo a multiplicação de:

0011b → 3 (decimal - multiplicando)
x 0010b → 2 (decimal - multiplicador)
0110b → 6 (decimal - resultado)

Vai-um	Multiplicador	Multiplicando	Resultado	Contador de bits
0	0010	0011	0000	04
0	0100	0011	0000	03
0	1000	0011	0000	02
1	0000	0011	0000 +0011 0011	01
0	0000	0011	**0110**	00

Tabela 3.12 – Multiplicação aplicando o algoritmo.

A seguir, mostra-se a descrição em VHDL de um multiplicador de 4 bits usando um somador e um deslocador de 4 bits.

Multiplicador de 4 bit

```
library ieee;
use ieee.std_logic_1164.all;
```

```
    entity mult4bits is
   port (   a, b: in std_logic_vector (3 downto 0);
              s: out std_logic_vector (3 downto 0));
end mult4bits;
architecture arch_mult4bits of mult4bits is

-- deslocamento de 1 bit para esquerda, zerando o bit menos significativo
function deslocador (x : std_logic_vector (3 downto 0))
return std_logic_vector is
variable y : std_logic_vector (3 downto 0);
begin
       for i in 3 downto 1 loop
            y(i) := x(i-1);
       end loop ;
       y(0) := '0';
    return y;
end;

-- somador de 4 bits
function somador4bits (a : std_logic_vector (3 downto 0);
                                b : std_logic_vector (3 downto 0))
return std_logic_vector is
variable vaium : std_logic;
variable soma  : std_logic_vector (3 downto 0);
begin
       vaium := '0';
       for i in 0 to 3 loop
           soma(i) := a(i) xor b(i) xor vaium;
           vaium := ( a(i) and b(i) ) or ( b(i) and vaium) or
                       (vaium and a(i) );
       end loop;
return soma;
end;


begin
       process(a,b)
       variable aux1  : std_logic_vector (3 downto 0);
       variable aux2  : std_logic_vector (3 downto 0);
       variable vaium : std_logic;
       begin

              -- inicializacoes
              aux1 := "0000";
              aux2 := a;
              vaium := '0';

              -- implementacao do algoritmo
              for i in 0 to 3 loop
                   aux1 := deslocador( aux1 );
                   vaium := aux2(3);
                   if vaium = '1' then
                      aux1 := somador4bits( aux1, b );
                   end if;
                   aux2 := deslocador( aux2 );
              end loop;
              s <= aux1;
   end process;
```

```
end arch_mult4bits;
```

3.6.4 - DIVISORES

Os divisores, assim como os multiplicadores, são circuitos relativamente complexos. Neste caso, o divisor descrito em VHDL consegue dividir números de 4 bits, apresentando ao final da operação o resultado e o resto. A seguir, o algoritmo principal da operação de divisão de *n* bits com ponto fixo.

Algoritmo do Divisor

```
Inicio:
Para i de 0 até o número de bits da divisão faça
    Vaium do quociente  recebe o bit mais significativo do quociente.
    Deslocar (quociente).
    Bit menos significativo do quociente recebe sinal de resto maior que divisor.
    Deslocar (resto).
    Bit menos significativo do resto recebe sinal de Vaium do quociente.
    Sinal de resto maior que o divisor recebe o resultado da comparação (divisor, resto).
    Se o sinal de resto maior for igual a um então
        Resto recebe o resto menos o divisor.
Deslocar (quociente)
Bit menos significativo do quociente recebe sinal de resto maior que divisor.
Fim
```

A seguir a descrição em VHDL'93 do algoritmo aplicado para um divisor de 4 bits de ponto fixo.

Divisor de 4 bit

```
library ieee;
use ieee.std_logic_1164.all;
entity div4bits is
    port(a, b : in std_logic_vector (3 downto 0);
            q, r : out std_logic_vector (3 downto 0));
end div4bits;
architecture arch_div4bits of div4bits is

-- deslocamento de 1 bit para esquerda, zerando o bit menos significativo
function deslocador (x : std_logic_vector (3 downto 0))
return std_logic_vector is
variable y : std_logic_vector (3 downto 0);
begin
        for i in 3 downto 1 loop
            y(i) := x(i-1);
        end loop ;
        y(0) := '0';
    return y;
end;
```

```
    -- compara dois numeros
function comparador (x : std_logic_vector; y : std_logic_vector)
return std_logic is
variable s : std_logic_vector (3 downto 0);
variable vaium : std_logic;
begin
     vaium := '0';
     for i in 0 to 3 loop
        s(i) := (x(i) xor y(i) xor vaium);
        vaium := ( y(i) and s(i) ) or ( y(i) and vaium) or (vaium and s(i) ) ;
     end loop;
     vaium := not vaium;
return vaium;
end comparador;

-- subtracao de dois numeros
function sub4bits (x : std_logic_vector; y : std_logic_vector)
return std_logic_vector is
variable s : std_logic_vector (3 downto 0);
variable vaium : std_logic;
begin
     vaium := '0';
     for i in 0 to 3 loop
       s(i) := (x(i) xor y(i) xor vaium);
       vaium := ( y(i) and s(i) ) or (y(i) and vaium) or (vaium and s(i) ) ;
    end loop;
return s;
end sub4bits;
begin
process (a,b)
variable vq, vr : std_logic_vector (3 downto 0);
variable quoc, resto : std_logic;
begin
       vq := a;
       quoc := '0';
       resto := '0';
       vr := "0000";
       for i in 0 to 3 loop
            quoc := vq(3);
            vq := deslocador(vq);
            vq(0) := resto;
            vr := deslocador(vr);
            vr(0) := quoc;
            resto := comparador(vr, b);
            if resto = '1' then
               vr := sub4bits (vr, b);
            end if;
      end loop;
      vq := deslocador (vq);
      vq(0) := resto;
      q <= vq;
      r <= vr;
end process;
end arch_div4bits;
```

3.6.5 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA E TEMPORIZAÇÃO DOS CIRCUITOS COMBINACIONAIS ARITIMÉTICOS.

	FF	LUTs 4	LUTs 3	CLBs
Comp. Simples	0	3	1	2
Comp. Porta lógica	0	3	1	2
Múltipla Comp.	0	10	2	6
Somador 1bit	0	2	0	1
Somador 4bit	0	9	1	5
Sub. 4bit	0	6	0	4
Mult. 4 bits	0	10	0	6
Div. 4 bits	0	45	5	25

Tabela 3.13 – Recursos utilizados do FPGA - Circuitos aritméticos.

O multiplicador e o divisor por apresentar um algoritmo complexo geram os maiores atrasos e utilização de recursos do FPGA (tabela 3.13 e 3.14). Normalmente algoritmos de implementação de multiplicadores e divisores são complexos. O tratamento da multiplicação e divisão com pontos flutuantes atingem níveis altos de complexidade.

	TL	TR	AT
Comp. Simples	8,858ns	5,814ns	14,672ns
Comp. Porta lógica	8,858ns	5,183ns	14,041ns
Múltipla Comp.	10,048ns	7,925ns	17,973ns
Somador 1bit	6,899ns	3,234ns	10,133ns
Somador 4bit	10,048ns	6,938ns	16,557ns
Sub 4bit	8,089ns	6,025ns	14,114ns
Mult. 4 bits	9,279ns	8,515ns	17,794ns
Div. 4 bits	18,378ns	30,664ns	49,042ns

Tabela 3.14 – Temporização dos circuitos aritméticos.

Neste capítulo, foram apresentadas descrições em VHDL de circuitos combinacionais, ressaltando suas principais características, enfatizando o funcionamento, metodologias de descrição e estatísticas de desempenho espacial e temporal.

CAPÍTULO IV - CIRCUITOS SEQÜENCIAIS

Neste capítulo, apresentam-se alguns circuitos seqüenciais, tais como: tipos variados de flip-flops e latches, registradores, registradores de deslocamento, contadores e máquina de estados finita.

Para cada circuito implementado, são gerados dados estatísticos de sua implementação em FPGA. As estatísticas geradas são: estatísticas espaciais, como número de flip-flops, de LUTs de 4 e 3 entradas e de CLBs utilizadas; e estatísticas temporais, como tempo de lógica, tempo de roteamento e o tempo total de execução do circuito. Além de geradas, as estatísticas são comparadas e analisadas.

4.1 - INTRODUÇÃO

Neste capítulo, apresentam-se descrições em VHDL de circuitos seqüenciais, tais como: Flip-Flops, Latches, Registradores, Registradores de Deslocamento, Contadores e Máquina de Estados Finita. Estatísticas de desempenho espacial e temporal são analisadas e comparadas.

Circuitos seqüenciais caracterizam-se por apresentar saídas que dependem não só dos valores atuais das entradas, mas também da seqüência com que os valores são aplicados nas entradas. São constituídos por uma lógica combinacional e também por células de memória que armazenam o estado atual do sistema. O estado atual do sistema define, em conjunto com as entradas, o comportamento futuro das saídas e dos próximos estados.

Na figura 4.1, denota-se a representação geral de um circuito seqüencial, com portas de entrada e saída, uma lógica combinacional e células de memória que podem definir o comportamento do circuito.

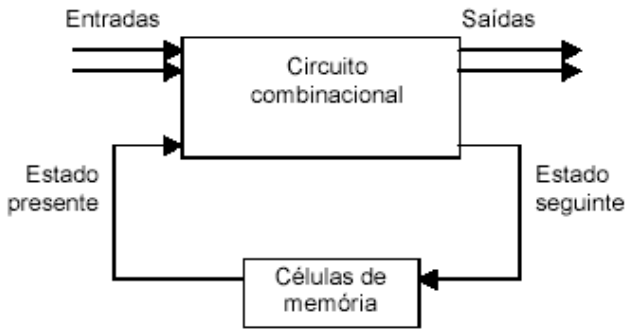


Figura 4.1 – Representação geral de um circuito seqüencial.

Na seção 4.2, pode ser visualizada descrição funcional e em VHDL de circuitos seqüenciais do tipo Flip-Flop.

4.2 - FLIP-FLOPS

Um flip-flop é um circuito digital básico que armazena um bit de informação. A saída de um flip-flop só muda de estado durante a transição do sinal do *clock*. Existem vários tipos de flip-flops, tais como: flip-flops D, flip-flop D com *reset* assíncrono, flip-flop D com *reset* síncrono, flip-flop D com *clock enable*, flip-flop T, flip-flop SR, flip-flop JK e outros.

4.2.1 - FLIP-FLOP D

A figura 4.2 mostra o flip-flop tipo D que possui uma lógica relativamente simplificada contendo apenas um *clock*, uma entrada *din* e uma saída *dout*. Este Flip-Flop funciona através de um pulso de *clock*. Quando se tem uma borda de subida do *clock*, a entrada do circuito *din* é atribuída à saída *dout*. Detectar uma borda de subida em VHDL é possível através do teste do evento do sinal de *clock*. Ex: **clk'event and clk ='1'**.

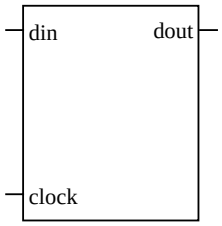


Figura 4.2 – Representação do flip-flop D

A seguir, o código VHDL de um circuito seqüencial do tipo flip-flop D usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop D

```
library ieee;
use ieee.std_logic_1164.all;

entity dff is
    port( d, clk : in std_logic;
                q : out std_logic);
end dff;

architecture arch_dff of dff is
begin
process (clk)
begin
       if (clk'event and clk ='1') then
          q <= d;
      end if;
end process;
end arch_dff;
```

4.2.2 - FLIP-FLOP D COM RESET ASSÍNCRONO

Este tipo de flip-flop D possui um sinal de *reset* assíncrono, isto é, que reage de imediato a qualquer alteração do sinal de *reset* independente do *clock* do circuito.

Interessante observar no código VHDL que primeiro testa-se o sinal de *reset* para depois testar o evento de borda de subida do *clock*. Neste caso se o sinal de *reset* for ativado, será atribuído à saída o valor **0**. Ocorrendo uma borda de subida do sinal de *clock* o valor da entrada *din* é atribuído à saída *dout*. Na figura 4.3, observa-se a representação de todas as entradas *din, clk* e *reset* e saída *dout* do flip-flop D com reset.

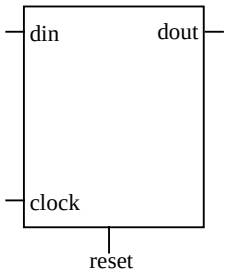


Figura 4.3 – Representação do flip-flop D com reset

Na seqüência o código VHDL de um circuito seqüencial do tipo flip-flop D com r*eset* assíncrono, usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop D com Reset Assíncrono

```
library ieee;
use ieee.std_logic_1164.all;
entity dffr is
    port(clk: in std_logic;
              reset: in std_logic;
              din: in std_logic;
              dout: out std_logic);
end dffr;
architecture arch_dffr of dffr is
begin
process (clk, reset)
begin
        if reset='1' then
           dout <= '0';
        elsif (clk'event and clk='1') then
           dout <= din;
        end if;
end process;
end arch_dffr;
```

4.2.3 - FLIP-FLOP D COM RESET SÍNCRONO

Este tipo de flip-flop D possui um sinal de *reset* síncrono, isto é, reage de forma sincronizada com um sinal de *clock*. Interessante observar que primeiro testa-se

o evento de borda de subida do *clock* para depois testar o sinal de *reset*. O valor da entrada *din* é atribuído à saída *dout* somente se ocorrer uma borda de subida do sinal de *clock* e o *reset* estiver desabilitado. A representação de todas as entradas e a saída do flip-flop D com *reset* síncrono pode ser melhor visualizada na figura 4.3.

A seguir, o código VHDL de um circuito seqüencial do tipo flip-flop D com *reset* síncrono, usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop D com Reset Síncrono

```
library ieee;
use ieee.std_logic_1164.all;

entity dffr2 is
    port(clk: in std_logic;
              reset: in std_logic;
              din: in std_logic;
              dout: out std_logic);
end dffr2;

architecture arch_dffr2 of dffr2 is
begin
process (clk)
begin
       if clk'event and clk='1' then
           if reset='1' then
              dout <= '0';
           else
               dout <= din;
           end if;
       end if;
end process;
end arch_dffr2;
```

4.2.4 - FLIP-FLOP D COM CLOCK ENABLE

Este tipo de flip-flop D possui um sinal de *clock enable*, isto é, o valor da entrada *din* só será atribuído à saída *dout*, quando ocorrer uma borda de subida do sinal de *clock* e o sinal de *clock enable* estiver em **1**.

Interessante observar que primeiro testa-se o evento da borda de subida do sinal de *clock*, para depois testar o sinal de *clock enable*, e se os dois testes confirmarem, o valor da entrada *din* é entregue à saída *dout*. Na figura 4.4 denota-se a representação de todas as entradas *din*, *enable* e *clock* e saída *dout* do flip-flip D com *clock enable*.

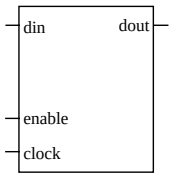


Figura 4.4 – Representação do flip-flop D com c*lock* e*nable*

Na seqüência, o código VHDL de um circuito seqüencial do tipo flip-flop D com *clock enable* usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop D com Clock Enable

```
library ieee;
use ieee.std_logic_1164.all;
entity dffe is
    port(clk: in std_logic;
           enable: in std_logic;
              din: in std_logic;
              dout: out std_logic);
end dffe;
architecture arch_dffe of dffe is
begin
process (clk)
begin
       if clk'event and clk='1' then
           if enable='1' then
              dout <= din;
           end if;
       end if;
end process;
end arch_dffe;
```

4.2.5 - FLIP-FLOP T

Este tipo de flip-flop possui uma lógica de inversão, onde a cada borda de subida do sinal de *clock,* o sinal de saída passa por uma operação de inversão de sinal. Interessante observar que o trecho do código em VHDL, em que a saída *dout* recebe o valor do sinal interno é totalmente concorrente. Na figura 4.5 tem-se a representação das portas de entrada *clock* e saída *dout* do flip-flop T.

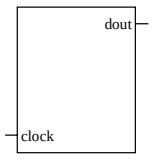


Figura 4.5 – Representação do flip-flop T

Na seqüência, o código VHDL de um circuito seqüencial do tipo flip-flop T usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop T

```
library ieee;
use ieee.std_logic_1164.all;
entity tff is
    port(clk: in std_logic;
              dout: out std_logic);
end tff;
architecture arch_tff of tff is
signal s : std_logic;
begin
dout <= s;
process (clk)
begin
       if clk'event and clk='1' then
          s <= not s;
       end if;
end process;
end arch_tff;
```

4.2.6 - FLIP-FLOP SR

Este tipo de flip-flop caracteriza-se por possuir entradas de *set* e *reset.* O comportamento do flip-flop é baseado nos valores destas entradas. Quando *reset* for igual a 1, a saída *q* recebe 0; caso *set seja* igual a 1, a saída *q* recebe 1. Na figura 4.6 mostra-se a representação das portas de entrada *s* e *r* e saída *q* do flip-flop SR.

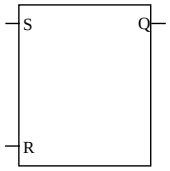


Figura 4.6 – Representação do Flip-Flop SR

A seguir, o código VHDL de um circuito seqüencial do tipo flip-flop SR usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop SR

```
library ieee;
use ieee.std_logic_1164.all;
entity srff is
    port(reset, set : in std_logic;
```

```
    q : out std_logic);
end srff;

architecture arch_srff of srff is
begin
process (set,reset)
begin
      if reset='1' then
         q<='0';
      end if;
      if set='1' then
         q<='1';
      end if;
end process;
end arch_srff;
```

4.2.7 - Flip-Flop JK

Este tipo de flip-flop possui uma entrada de *clock* e as entradas *j* e *k* que determinam o estado do flip-flop, semelhante ao flip-flop SR, porém com uma diferença: se a condição ***J=K=1*** não produz um estado ambíguo na saída. Para esta condição, o flip-flop JK sempre irá para o estado oposto em que se encontra. Com isto diz-se que o flip-flop está em *modo de comutação* ou *chaveamento*. Na figura 4.7 tem-se a representação do flip-flop JK.

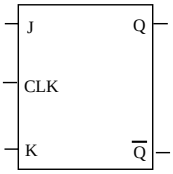


Figura 4.7 – Representação do Flip-Flop JK

Na seqüência, o código VHDL de um circuito seqüencial do tipo flip-flop JK usando o comando de execução seqüencial IF...THEN...ELSE:

Flip-Flop JK

```
library ieee;
use ieee.std_logic_1164.all;
entity jkff is
    port(j, k, clk : in std_logic;
                 q : buffer std_logic;
                nq : out std_logic);
end jkff;
architecture arch_jkff of jkff is
begin
```

```
process (clk)
begin
      if (clk'event and clk='1') then
         if j='0' then
            if k='1' then
               q <= '0';
            end if;
      else
        if k='0' then
          q <= '1';
        else
          q <= not(q);
        end if;
      end if;
    end if;
    nq <= not(q);
end process;
end arch_jkff;
```

4.2.8 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA E TEMPORIZAÇÃO - FLIP-FLOPS.

Observa-se na tabela 4.1 que flip-flops são circuitos relativamente simples, o tempo de lógica e de roteamento é reduzido devido à sua simplicidade. O flip-flop D e o flip-flop D com *reset* assíncrono possue o tempo de roteamento zero. Isso acontece, porque estes utilizam flip-flops já implementados no FPGA, não havendo necessidade de roteamento para implementar a lógica descrita em VHDL.

Estatísticas de recursos utilizados do FPGA				
Flip-Flop	**FF**	**LUTs 4**	**LUTs 3**	**CLBs**
D	1	0	0	0
D com Reset Assíncrono	1	0	0	0
D com Reset Síncrono	1	1	0	1
T	1	1	0	1
SR	1	1	0	1
JK	1	1	0	1
Temporização				
Flip-Flop	**TL**	**TR**		**TT**
D	8,150ns	0,000ns		8,150ns
D com Reset Assíncrono	8,150ns	0,000ns		8,150ns
D com Reset Síncrono	2,579ns	3,657ns		6,236ns
T	6,090ns	1,470ns		7,560ns
SR	2,279ns	6,423ns		8,702ns
JK	6,090ns	41,792ns		7,882ns

Tabela 4.1 - Estatísticas de desempenho de flip-flops

4.3 - LATCHES

O *latch* é um circuito lógico muito importante, que consiste basicamente no acoplamento de um par de inversores, de forma que a saída do primeiro inversor é ligada à entrada do segundo, e a saída do segundo é ligada à entrada do primeiro. Na figura 4.8, tem-se a representação de um *latch* estático.

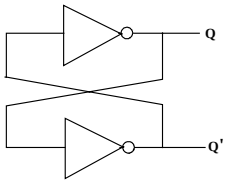


Figura 4.8 - Representação de um circuito *latch* estático.

Sem intervenção externa, o *latch* permanece indefinidamente em uma das duas situações possíveis, chamadas *estados*. Supondo Q = 0, a saída do inversor inferior da figura 4.8, seria Q'= 1. Por esta característica, o *latch* pode ser usado para estabelecer e manter um nível lógico sem qualquer intervenção externa. Esta independência das entradas externas do *latch*, possibilita a utilização para armazenar, ou lembrar um bit lógico. Nas próximas seções, alguns *latches* tais como: latch D, latch D com reset e latch SR.

4.3.1 - LATCH D

O *latch* tipo D possui uma lógica relativamente simplificada, contendo apenas um pino de controle *porta*, uma entrada *din* e uma saída *dout*. Este *latch* funciona através do sinal da porta de controle. Quando se tem uma borda de subida deste sinal, a entrada do circuito *din* é atribuída à saída *dout*. Observe que o evento de transição do sinal de controle não é testado e sim apenas o seu estado. (“**porta” deve ser igual a 1**). Na seqüência, o código VHDL de um circuito seqüencial do tipo latch D usando o comando de execução seqüencial IF...THEN...ELSE:

Latch D

```
library ieee;
use ieee.std_logic_1164.all;
entity dl is
    port(porta: in std_logic;
                 din: in std_logic;
                dout: out std_logic);
```

```
end dl;
architecture arch_dl of dl is
begin
process (porta, din)
begin
      if porta='1' then
          dout <= din;
       end if;
end process;
end arch_dl;
```

4.3.2 - LATCH D COM RESET

Este tipo de latch D possui um sinal de *reset*, isto é, quando o sinal de *reset* for ativado (**reset='1'**) atribuí-se à saída *dout* o valor 0. Observa-se que primeiro testa-se o sinal de *reset* para depois testar o de controle *porta*. Confirmando o sinal de controle *porta*, o valor da entrada *din* é atribuído à saída *dout*.

A seguir, o código VHDL de um circuito seqüencial do tipo latch D com *reset* usando o comando de execução seqüencial IF...THEN...ELSE:

Latch D com Reset

```
library ieee;
use ieee.std_logic_1164.all;
entity dlr is
    port(porta: in std_logic;
              reset: in std_logic;
            din: in std_logic;
              dout: out std_logic);
end dlr;
architecture arch_dlr of dlr is
begin
process (porta, din, reset)
begin
       if reset='1' then
          dout <= '0';
       elsif porta='1' then
          dout <= din;
       end if;
end process;
end arch_dlr;
```

4.3.3 - LATCH SR

O latch do tipo SR é constituído por duas portas NOR, possui uma lógica relativamente simplificada, contendo apenas um sinal de *set* , um *reset* e as saídas *q* e

q'. Observe que a saída de uma porta NOR serve de entrada para a outra, como visualizado na figura 4.9.

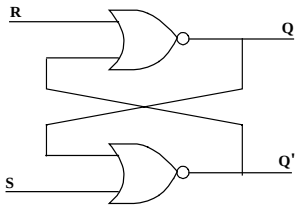


Figura 4.9- Representação lógica de um *latch* SR

Na seqüência, o código VHDL de um circuito seqüencial do tipo latch SR usando portas lógicas básicas, especificamente portas NOR:

Latch SR

```
library ieee;
use ieee.std_logic_1164.all;

entity srl2 is
    port(s, r  : in std_logic;
           q, qn : buffer std_logic);
end srl2;

architecture arch_srl2 of srl2 is
begin
        qn <= s nor q;
        q   <= r nor qn;
end arch_srl2;
```

4.3.4 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA E TEMPORIZAÇÃO - LATCHES

Observa-se na tabela 4.2 que os circuitos *latches* assim como os *flip-flops*, possuem uma lógica simplificada utilizando poucos recursos disponíveis no FPGA. Interessante observar que o latch do tipo SR tem o seu atraso total (AT) estimado em 11.879ns sendo que um latch D com *reset* é de 8.677ns. Esta diferença significativa é gerada pela utilização da porta lógica NOR na implementação do latch SR, enquanto que o latch D com *reset* utiliza um latch disponível no FPGA, gerando uma temporização otimizada.

Estatísticas de recursos utilizados do FPGA				
Latch	**Latch**	**LUTs 4**	**LUTs 3**	**CLBs**
D	1	0	0	0
D com Reset	1	1	0	1
SR	0	2	0	1

Temporização			
Latch	**TL**	**TR**	**AT**
D	8,150ns	0,000ns	8,150ns
D com Reset	2,279ns	6,398ns	8,677ns
SR	8,089ns	3,790ns	11,879ns

Tabela 4.2 - Estatísticas de desempenho de *latches*

A seguir, a descrição em VHDL funcional de circuitos seqüenciais, tais como: registadores, registradores de deslocamento e contadores.

4.4 – REGISTRADORES

Os registradores são circuitos compostos por um conjunto de *flip-flops* ou *latches.* A função principal dos registradores é armazenar e recuperar um dado.

A operação mais comum que se realiza sobre dados armazenados em um registrador é a operação de transferência, que envolve transferir dados de um registrador para outro. A transferência pode ser **síncrona** ou **assíncrona**: a síncrona é feita utilizando as entradas síncronas dos flip-flops e o sinal de *clock*; a assíncrona pode ser feita usando as entradas *preset* e *reset* do flip-flop.

A seguir, o código VHDL de um circuito seqüencial de um registrador de 8 bits usando o comando de execução seqüencial IF...THEN...ELSE. Seu funcionamento é simplificado, bastando identificar uma borda de subida do sinal de *clock* para que a entrada *din* seja atribuída à saída *dout*.

Registrador de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;
entity reg is
    port(din  : in  std_logic_vector (7 downto 0);
           clk  : in  std_logic;
           dout : out std_logic_vector (7 downto 0));
end reg;
architecture arch_reg of reg is
begin
process(clk)
begin
        if (clk'event and clk='1')
          then dout <= din;
```

```
        end if;
end process;
end arch_reg;
```

4.5 - REGISTRADOR DE DESLOCAMENTO

Registrador de deslocamento ou *shift register* é constituído por um arranjo de vários flip-flops, de modo que os dados binários armazenados em um determinado flip-flop é deslocado para o flip-flop adjacente em toda borda de subida do sinal de *clock*. Os registradores podem ser configurados para realizar o deslocamento de dados tanto para a direita quanto para a esquerda.

O registrador de deslocamento proposto é composto pelas entradas *din*, *clock*, *load* e pela saída *dout.* O funcionamento é baseado nos sinais de controle: *clock* e *load.*

Caso ocorrer um sinal de *clock* e o sinal de *load* estiver habilitado (load='1'), a entrada *din* é atribuída à saída *dout*. Porém, se o sinal de *load* estiver desabilitado (load='0'), é feito o deslocamento para esquerda do valor armazenado.

Na seqüência, o código VHDL de um circuito seqüencial do tipo registrador de deslocamento de 4 bits, usando o comando de execução seqüencial IF...THEN...ELSE:

Registrador de Deslocamento de 4 bits

```
library ieee;
use ieee.std_logic_1164.all;

entity shiftreg is
    port(din  : in  std_logic_vector (3 downto 0);
         clk  : in  std_logic;
         load : in std_logic;
         dout : out std_logic_vector (3 downto 0));
end shiftreg;

architecture arch_shiftreg of shiftreg is
begin
process (clk)
variable aux: std_logic_vector(3 downto 0);
begin
    if clk'event and clk='1' then
      if load='1' then
        dout<=din;
        aux:=din;
      else
        for i in 3 downto 1 loop
           aux(i) := aux(i-1);
        end loop ;
        aux(0) := '0';
```

```
        dout<=aux;
      end if;
    end if;
 end process;
end arch_shiftreg;
```

4.6 - CONTADOR

Um contador pode ser definido como um registrador capaz de incrementar ou decrementar o valor da sua saída a cada pulso do *clock*. Contadores podem ser classificados como: básicos contendo apenas um *clock* e uma saída ou ainda com as opções:

- *load*, que permite a contagem a partir de um valor carregado;
- *up/down*, sinal de controle para definir se o contador irá incrementar ou decrementar;
- *clock enable*, habilita/desabilita a contagem independente do *clock*; entre outras.

A seguir, o código VHDL de um circuito seqüencial do tipo contador de 4 bits usando o comando de execução seqüencial IF...THEN...ELSE:

Contador de 4 bits

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity cont4bits is
    port(clk  : in  std_logic;
         dout : out std_logic_vector (3 downto 0));
end cont4bits;

architecture arch_cont4bits of cont4bits is
signal i: std_logic_vector (3 downto 0);
begin
process (clk)
begin
      if clk'event and clk='1' then
          i <= i + '1';
      end if;
end process;
dout <= i;
end arch_cont4bits;
```

4.7 - ESTATÍSTICAS E RECURSOS UTILIZADOS DO FPGA E TEMPORIZAÇÃO DO REGISTRADOR, REGISTRADOR DE DESLOCAMANTO E CONTADOR

Na tabela 4.3, visualizam-se estatísticas de desempenho de registradores, registradores de deslocamento e contadores. São circuitos implementados a partir do mesmo elemento básico, *flip-flops* ou *latches*. É interessante observar que o registrador de deslocamento é um circuito que envolve uma lógica mais complexa, com controle de *load* e uma lógica de deslocamento de bits. Com isso, se justifica a diferença significativa na taxa de temporização deste circuito em relação aos demais.

Estatísticas de recursos utilizados do FPGA				
Registrador	**FF**	**LUTs 4**	**LUTs 3**	**CLBs**
8 bits	8	0	0	0
Deslocamento	4	6	1	5
Contador	4	4	0	2
Temporização				
Registrador	**TL**	**TR**	**AT**	
8 bits	8,150ns	0,000ns	8,150ns	
Deslocamento	9,458ns	5,013ns	14,471ns	
Contador	6,090ns	1,819ns	7,909ns	

Tabela 4.3 - Estatísticas de desempenho de registradores

Observa-se na tabela 4.3 que o registrador de 8 bits é formado por oito flip-flops, não tendo a necessidade de utilizar recursos como LUTs ou CLBs. Os oito flip-flops estão dispostos de forma paralela, por isso, um registrador de 8 bits formado por oito flip-flops tem a mesma temporização de um único flip-flop D (tabela 4.3 e 4.1)

4.8 – MÁQUINA FINITA DE ESTADO

A descrição em VHDL de uma máquina de estados finita (***F**inite **S**tate **M**achine*) síncrona pode ser implementada utilizando diferentes metodologias.

Uma forma bastante eficiente consiste em descrever a máquina de estados usando dois processos. O primeiro, para descrever os estados da máquina e suas transições, e o segundo, para gerar o sinal de *clock* e possibilitar a transição de estados.

Outra maneira é a descrição com apenas um processo, onde o pulso de transição de estado e os estados estão localizados no mesmo processo. Nestes casos, os sinais de *reset* das máquinas de estados descritas podem ser assíncronos, isto é, reagem independente do *clock* do sistema.

A representação dos estados que uma máquina de estados finita proposta que é capaz de percorrer, pode ser melhor visualizadas figura 4.10.

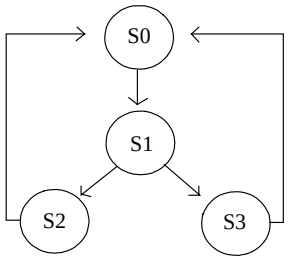


Figura 4.10 – Descrição dos Estados

Na seqüência, a descrição em VHDL de um circuito seqüencial do tipo máquina de estados finita (FSM) utilizando um único processo.

FSM com um único processo

```
library ieee;
use ieee.std_logic_1164.all;
entity fsm is
    port(reset  : in std_logic;
         codigo : in std_logic;
         clk    : in std_logic;
        dout   : out std_logic);
end fsm;
architecture arch_fsm of fsm is
type estados is (s0,s1,s2,s3);
signal est:estados;
begin
    process (clk,reset,codigo) is
    begin
       if reset='1' then
          est<=s0;
       elsif (clk'event and clk='1') then
          case est is
          when s0 => est <=s1;
          when s1 => if codigo='1' then
                        est<=s2;
                     else est<=s3;
                     end if;
          when s2 => dout<='0';
                     est<=s0;
          when s3 => dout<='1';
                     est<=s0;
          end case;
      end if;
end process;
end arch_fsm;
```

A seguir, a descrição em VHDL de um circuito seqüencial do tipo máquina finita de estados (FSM) utilizando dois processos.

FSM com dois processos

```
library ieee;
use ieee.std_logic_1164.all;

entity fsm is
    port(reset  : in std_logic;
         codigo : in std_logic;
         clk    : in std_logic;
         dout   : out std_logic);
end fsm;

architecture arch_fsm of fsm is
type estados is (s0,s1,s2,s3);
signal atual,prox : estados;
begin

process (clk, reset)
begin
       if reset='1' then
          atual<=s0;
       elsif (clk'event and clk='1') then
          atual<=prox;
       end if;
end process;
process (atual,prox,codigo) is
begin
   case atual is
     when s0 => prox <=s1;
     when s1 => if codigo='1' then
                        prox<=s2;
                         else prox<=s3;
                    end if;
      when s2 => dout<='0';
           prox<=s0;
      when s3 => dout<='1';
           prox<=s0;
   end case;
end process;
end arch_fsm;
```

4.9 - ESTATÍSTICAS DE RECURSOS UTILIZADOS DO FPGA E TEMPORIZAÇÃO - FSM

Dois modelos de implementação foram avaliados: FSM implementada com um processo e com dois processos. É importante ressaltar que o número de estados das duas implementações é o mesmo e todos os estados são iguais. Em princípio, a implementação como um único processo obteve melhores resultados tanto na estatística

de utilização de recursos do FPGA quanto na temporização obtida. A implementação utilizando um único processo obteve um ganho na temporização realmente significativo de aproximadamente 60% em relação à implementação com dois processos. Isso acontece, pois a implementação com dois processos gera a necessidade de uma lógica de roteamento maior, levando à uma temporização menos otimizada.

Na tabela 4.4 apresentam-se estatísticas de desempenho de circuitos seqüenciais do tipo FSM.

Estatísticas de recursos utilizados da FPGA				
FSM	**FF**	**LUTs 4**	**LUTs 3**	**CLBs**
1 Processo	5	4	0	2
2 Processos	5	5	0	3
Temporização				
FSM	**TL**	**TR**	**AT**	
1 Processo	5,230ns	2,666ns	7,896ns	
2 Processos	2,660ns	10,500ns	13,160ns	

Tabela 4.4 - Estatísticas de desempenho de FSM

Neste capítulo, foram apresentadas descrições em VHDL de circuitos seqüências, ressaltando suas principais características enfatizando o funcionamento, metodologias de descrição e estatísticas de desempenho espacial e temporal.

CAPÍTULO V – GLT: UM SISTEMA DIGITAL PARA AUXILIAR NOS SERVIÇOS TELEFÔNICOS

Este capítulo, apresenta uma proposta de um sistema digital para auxiliar nos serviços telefônicos, discutindo as fases de projeto, descrição, implementação, simulação e testes dos seus principais módulos descritos em VHDL e mapeados em um FPGA. São abordados temas como: a arquitetura do sistema, seus principais módulos funcionais, a descrição e implementação de cada um dos módulos. Finalmente, implementa-se o sistema integrado em FPGAs, enfatizando as estatísticas de desempenho espacial e temporal do sistema.

5.1 - INTRODUÇÃO

No mundo moderno, a comunicação e a informação tornaram-se essenciais para o mercado globalizado e surgiram empresas em todo o mundo especializadas na área de telecomunicações.

O mercado baseado em telecomunicações está apenas em sua fase inicial. Isto significa que ainda há muito o que fazer nessa área, tornando-se um incentivo para pesquisadores que se interessam e para os próprios consumidores que estão cada vez mais exigentes.

É de um grande interesse para os consumidores equipamentos ou ferramentas para gerenciar todo o fluxo de informações dos serviços telefônicos. Com isso, os consumidores poderiam monitorá-lo, ou seja, poderiam ter um controle melhor de sua conta telefônica.

Nota-se que no atual serviço telefônico, tanto residencial quanto comercial, há um problema que consiste na perda de controle ou as poucas informações que se tem da utilização destes serviços prestados por empresas da área. Por exemplo, pessoas que ligam para um determinado destino dificilmente mantém, em tempo real, estatísticas destas ligações.

É de grande interesse que exista um sistema para gerenciar alguns problemas como: ligação efetuada, data, horário, tempo total e do custo associado em uma determinada ligação, obtendo-se o controle sobre a utilização deste serviço em um determinado período de tempo. Desta forma, muitos consumidores teriam, à sua disposição, um mecanismo que auxilia no controle e na organização das informações.

5.2 – PROPÓSITOS DO SISTEMA

Ao analisar a situação atual e considerações anteriores, foi proposto um sistema digital, entitulado de Gerenciador de Linha Telefônica (GLT), onde o usuário pode ter um controle de sua linha telefônica, a qual pode ser monitorada constantemente através de um sistema implementado em circuitos FPGAs.

Desta maneira, o objetivo principal deste capítulo é: definir, projetar, implementar, simular e testar os principais módulos do *núcleo* do sistema. O GLT, inicialmente, terá funções básicas, tais como:

- verificar e registrar o número do destino das ligações telefônicas realizadas em um determinado lugar.
- medir o tempo de cada ligação. Além disso, calcular e armazenar estatísticas de utilização do serviço telefônico.
- calcular o tempo total de ligações e fazer uma estimativa de custo num determinado período de tempo.

O sistema proposto está dividido em três fases principais:

- **Primeira fase:** Definição da arquitetura básica proposta para o sistema GLT, apresentada neste capítulo.
- **Segunda fase:** consiste na descrição, simulação e implementação de cada um dos módulos principais da arquitetura do sistema. Além disto, são feitas análises das estatísticas de desempenho do circuito proposto, oferecidas pela ferramenta Xilinx Foundation Series, como: atraso total, freqüência de funcionamento, número de CLBs e LUTs.
- **Terceira fase:** união de todos os módulos definidos e projetados anteriormente num único sistema, o GLT. Finalmente, apresenta-se uma análise e discussão dos resultados obtidos.

5.3 - DESCRIÇÃO GERAL DO FUNCIONAMENTO DO GLT

O GLT foi proposto para funcionar da seguinte maneira. Quando realizada uma ligação de um ponto origem para um ponto destino (figura 5.1), será armazenado o número do telefone discado e será ativado também um contador de tempo total da ligação realizada. Estas operações são feitas, respectivamente, pelo receptor de dados e pelo contador de tempo, conforme visualiza-se na figura 5.2, onde tem-se as entradas *telin* (número do telefone) e *stel* (sinal de ativação) e a saída *telout* (telefone registrado) do módulo receptor de dados. Já o módulo contador de tempo possui uma entrada *stel* e a saída *TTL* (tempo total de ligação).

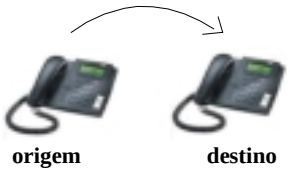


Figura 5.1 –Comunicação entre dois telefones

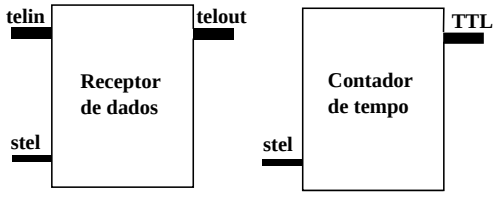


Figura 5.2 – Receptor de Dados e Contador de Tempo

Quando a ligação realizada for atendida, são ativados os módulos, relógio de tempo real e calendário que registram o horário e data em que a ligação foi realizada. O horário será expresso no formato *hora : minuto : segundo,* e a data será expresso no formato *dia : mês : ano.* Na figura 5.3 (a e b) são visualizados os módulos relógio e calendário.

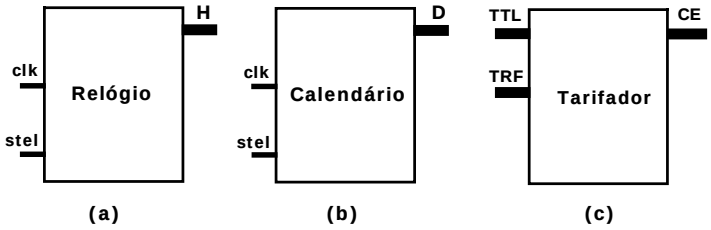


Figura 5.3 – (a) Módulos Relógio, (b) Calendário e (c) Tarifador

O módulo relógio é constituido pelas entradas *stel* e *clk* (clock geral do sistema) e saída *H* (horário da ligação). Já o módulo calendário é composto pelas entradas *stel* e *clk* e saída *D* (data da ligação).

Após o término da ligação é calculado o custo estimado, através do tempo total da ligação expresso em minutos e o valor estimado da tarifa. O cálculo do custo estimado é obtido pela fórmula:

*Custo = tempo total (em minutos) * tarifa.*

Esta operação é realizada pelo módulo Tarifador, ver figura 5.3 (c), que possui as portas de entradas *TTL* e *TRF* (valor da tarifa) e a saída *CE* (custo estimado).

Os dados: número do telefone, data, horário, tempo total e custo estimado estão relacionados com uma determinada ligação e são armazenados em uma memória no formato indicado na figura 5.4.

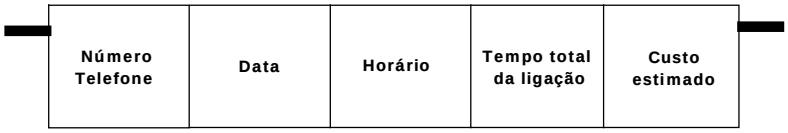


Figura 5.4 – Formato do armazenamento em memória de uma determinada ligação

5.4 - DESCRIÇÃO DO GLT - CARACTERÍSTICAS

O GLT tem a arquitetura mostrada na figura 5.5. Nesta figura percebe-se que o GLT é composto por seis grandes módulos: (i) Receptor de Dados, (ii) Contador de Tempo, (iii) Tarifador, (iv) Relógio, (v) Calendário e (vi) Memória.

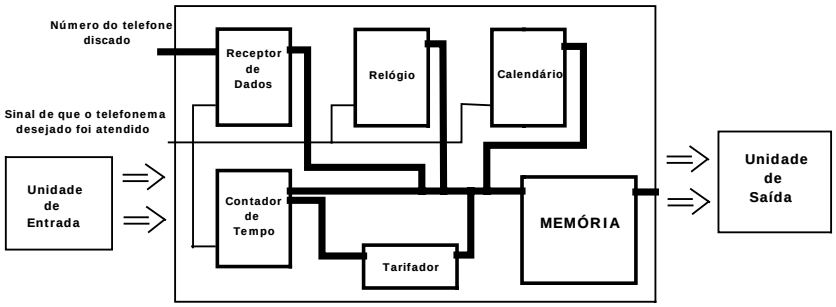


Figura 5.5 – Arquitetura do GLT

O GLT é composto também por uma unidade de entrada onde o usuário poderá atualizar alguns parâmetros, tais como: acertar o horário corrente, acertar dia, mês e ano, apagar antigos dados armazenados, etc.

5.4.1 - RECEPTOR DE DADOS

Este módulo tem como principal função armazenar o número do telefone e o código da cidade (DDD) de uma determinada ligação. Seu funcionamento é descrito a seguir: (i) recebe o número do telefone e o código da cidade; (ii) depois que a ligação for realizada, armazena o número na memória.

Este módulo consiste em um registrador que armazena os telefones destinos discados no telefone origem, ou seja, ele registra todos os telefonemas que foram realizados. Basicamente o módulo receptor de dados consiste em um único registrador que armazena o número do telefone e o código da cidade.

A seguir a descrição VHDL do módulo receptor de dados:

Receptor de dados

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity receptor is
port (peclk_sinc : in std_logic;
        peddd : in std_logic_vector(1 downto 0);
        pentelefone : in std_logic_vector(26 downto 0);
        psddd : out std_logic_vector(1 downto 0);
        psntelefone : out std_logic_vector(26 downto 0));
end receptor;

architecture arch_receptor of receptor is
   signal sddd : std_logic_vector(1 downto 0);
   signal sntelefone : std_logic_vector(26 downto 0);
begin
process(peclk_sinc)
begin
   if peclk_sinc'event and peclk_sinc='0' then
      sddd <= peddd;
      sntelefone <= pentelefone;
   end if;
end process;
psddd <= sddd;
psntelefone <= sntelefone;
end arch_receptor;
```

Na figura 5.6 visualiza-se a simulação do funcionamento do módulo receptor de dados. Esta simulação foi realizada utilizando a ferramenta da Xilinx, Foundation Series. Nesta figura, observa-se que o módulo é ativado depois que o telefonema for atendido pelo telefone destino (peClk_Sinc = 1), e após a ligação (peClk_Sinc = 0), os dados (código da cidade e número do telefone) são armazenados.

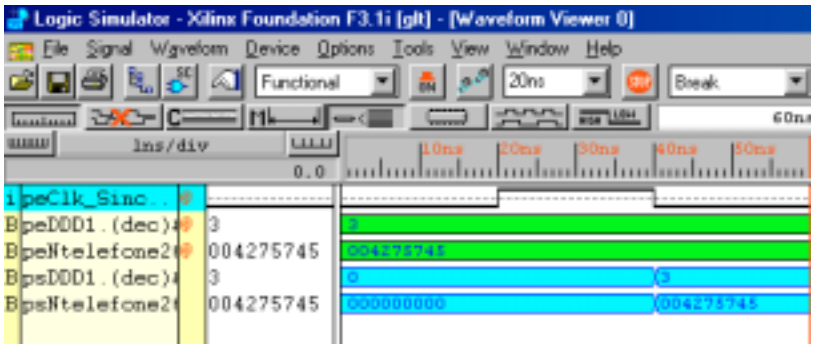


Figura 5.6 – Simulação do módulo receptor de dados

5.4.2 - CONTADOR DE TEMPO

Este módulo tem como principal função calcular o tempo total realizado em uma determinada ligação. Seu funcionamento tem como base o clock geral do sistema, que é de 12 Mhz (kit XS-40, da Xess). A partir deste clock, é implementado um divisor que gera um clock a cada 1 segundo. Seu funcionamento é ativado depois que a ligação realizada for atendida. Quando a ligação for finalizada, o tempo total calculado é armazenado na memória do sistema.

A seguir a descrição VHDL do módulo contador de tempo:

Contador de tempo

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity contador_tempo is
port ( peclk_geral : in std_logic;
       pesinal_tel : in std_logic;
       psregsegundo : out std_logic_vector(5 downto 0);
       psregminuto : out std_logic_vector(5 downto 0);
       psreghora : out std_logic_vector(4 downto 0);
       pstempo_minuto : out std_logic_vector(7 downto 0));
end contador_tempo;

architecture arch_contador_tempo of contador_tempo is
  signal sclk_1seg : std_logic;
  signal sregsegundo: std_logic_vector(5 downto 0);
  signal sregminuto: std_logic_vector(5 downto 0);
  signal sreghora: std_logic_vector(4 downto 0);
  signal stempo_minuto : std_logic_vector(7 downto 0);
begin
process (peclk_geral)
  variable vcont : integer;
begin
  if peclk_geral'event and peclk_geral='1' then
    vcont := vcont+1;
    if vcont <= 6000000 then
      sclk_1seg <='1';
    else
      sclk_1seg <='0';
    end if;
    if vcont=12000000 then
      vcont :=0;
    end if;
  end if;
end process;
process (pesinal_tel, sclk_1seg)
  variable vregsegundo,vregminuto: integer;
  variable vreghora : integer;
```

```
      variable vreg_out_min : std_logic_vector(7 downto 0);
  variable vflag : std_logic;
begin
  if sclk_1seg'event and sclk_1seg ='1' then
    if pesinal_tel = '1' then
      if vregsegundo <= 59 then
        vregsegundo := vregsegundo +1;
        sregsegundo <= sregsegundo +1;
      end if;
      if vregsegundo = 60 then
        vregminuto:= vregminuto +1;
        vreg_out_min := vreg_out_min+1;
        sregminuto <= sregminuto+1;
      end if;
      if vregminuto = 60 then
        vreghora := vreghora +1;
        sreghora<= sreghora+1;
      end if;
      if vregsegundo = 60 then
        vregsegundo :=0;
        sregsegundo <= (others =>'0');
      end if;
      if vregminuto = 60 then
        vregminuto := 0;
        sregminuto <= (othres =>'0');
      end if;
      if vreghora = 24 then
        vreghora := 0;
        sreghora <= (others =>'0');
      end if;
        vflag:='1';
      end if;
 if pesinal_tel = '0' and vflag='1' then
        psregsegundo <= sregsegundo;
        psregminuto <= sregminuto;
        psreghora <= sreghora;
        pstempo_minuto <= vreg_out_min;
        vflag:='0';
      elsif pesinal_tel = '0' and vflag='0' then
        vregsegundo := 0;
        vregminuto := 0;
        vreghora := 0;
        vreg_out_min := (others=>'0');
        sregsegundo <= (others =>'0');
        sregminuto  <= (others =>'0');
        sreghora    <= (others =>'0');
      end if;
    end if;
end process;
end arch_contador_tempo;
```

O módulo contador de tempo é composto pelos seguintes blocos:

- um contador para os segundos, um contador para os minutos e um contador para as horas;
- um registrador que armazena o tempo total da ligação no seguinte formato: hh:mm:ss, onde hh, mm e ss são, respectivamente, hora, minuto e segundo;
- um registrador que armazena o tempo total da ligação em minutos para o cálculo do custo da ligação, realizado pelo módulo tarifador.

Este módulo contador de tempo foi implementado com duas saídas, a primeira armazena o tempo total da ligação e, a segunda armazena o tempo total em minutos.Na figura 5.7 visualiza-se a simulação do módulo contador de tempo. Nesta simulação observa-se que, enquanto a ligação está em andamento (peSinalTel = 1), o tempo total está sendo calculado. Quando a ligação terminar (peSinalTel = 0), o tempo total será registrado e será armazenado posteriormente na memória.

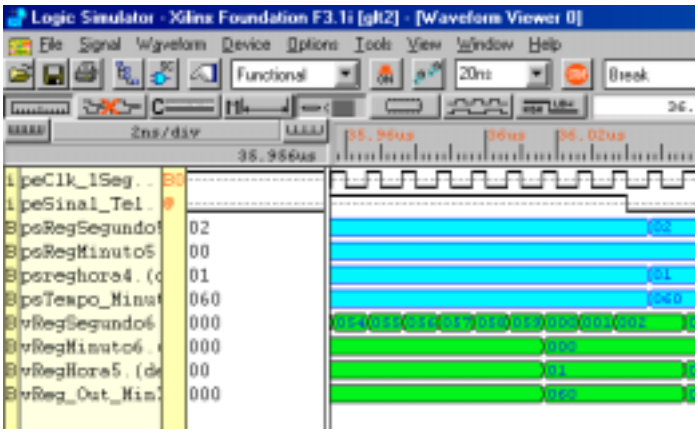


Figura 5.7 – Simulação do Módulo Contador de Tempo

5.4.3 - TARIFADOR

Este módulo tem como principal função calcular uma estimativa do custo da ligação. Este cálculo é realizado com base no tempo total da ligação (calculado no módulo contador de tempo) multiplicado por uma tarifa. Seu funcionamento se inicia depois que o tempo total da ligação for calculado. O valor da tarifa será uma constante. Desta forma este custo será expresso pela seguinte formula: tempo total (em minutos) * tarifa (constante).

Este custo poderá ser mais realístico usando-se de tarifas próximas às existentes. Neste caso, o GLT poderá receber, via mundo externo, as tarifas "reais" das ligações. Esta poderá ser outra versão do GLT, mais sofisticada, onde tarifa é uma variável em função da central telefônica usada. O tarifador é composto por:

- um registrador que armazena o tempo total em minutos;
- um registrador que armazena a tarifa;
- um multiplicador que calcula o custo estimado de cada ligação;
- um registrador que armazena o resultado do custo calculado.

Na figura 5.8 visualiza-se a simulação do módulo tarifador. Este módulo é ativado depois que o módulo contador de tempo finalizar seu funcionamento.

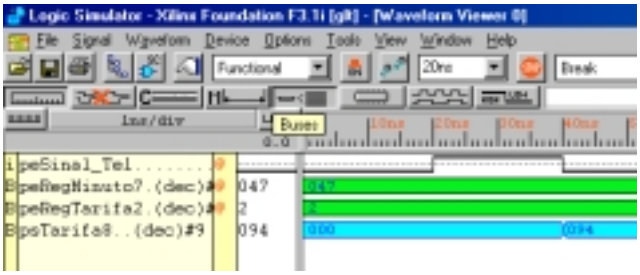


Figura 5.8 - Simulação do módulo tarifador

5.4.4 - RELÓGIO EM TEMPO REAL

Este módulo tem como principal função registrar o horário em que foi realizada a ligação. Depois que a ligação for atendida, o horário corrente é armazenado. O relógio em tempo real é composto por:

- um gerador de clock de 1 Hz;
- contadores para expressar os segundos, minutos e horas;
- um registrador que armazena o horário da ligação.

Figura 5.9 – Simulação do módulo relógio em tempo real

Na figura 5.9 demostra-se a simulação do módulo relógio em tempo real. Nota-se que o horário da ligação realizada é armazenado quando a ligação for atendida e o relógio continua com o seu funcionamento normalmente.

5.4.5 - CALENDÁRIO

Neste módulo implementa-se um calendário em tempo real, para manter a data em que foi realizada a ligação. Este calendário será expresso da seguinte maneira: dd/mm/aa, sendo respectivamente, dia, mês e ano.

Este módulo tem como principal função registrar a data em que foi realizada a ligação. A cada 24 horas é incrementado o contador de dia, de acordo com o mês, lembrando que o número de dias pode chegar a 30 ou 31, e se o mês for fevereiro o número de dias pode chegar a 28 ou 29, no caso de um ano bissexto. A cada 12 meses é incrementado o contador de anos. Depois que a ligação for atendida, a data corrente desta ligação é armazenada. O Calendário será composto pelos seguintes blocos:

- contadores de dia, mês e ano;
- um divisor de ano por 4, para detectar o ano bissexto;
- um registrador que armazena a data em que foi realizada a ligação.

Na figura 5.10 ilustra-se a simulação do módulo calendário em tempo real. Nesta simulação pode-se observar que o dia é incrementado a cada 24 horas (peSinal_Rel=1) e que a data é registrada toda vez que uma determinada ligação for atendida (peSinal_Tel=1).

Figura 5.10 – Simulação do módulo Calendário em Tempo real

5.4.6 - MEMÓRIA

Este módulo tem como principal função armazenar todos os dados obtidos em uma ligação. Seu modo de gravação inicializa depois que o cálculo da tarifa for realizada, pois este cálculo é a última tarefa do sistema.

Este módulo consiste em uma memória, onde todos os dados, dos módulos mencionados anteriormente, são armazenados.

A memória é composta pelos seguintes blocos:

- um registrador temporário, onde são armazenados todos os dados antes de ir para a memória;
- uma memória
- um contador de endereço para que não se sobrescreva os dados.

Na figura 5.11 visualiza-se a simulação do módulo memória.

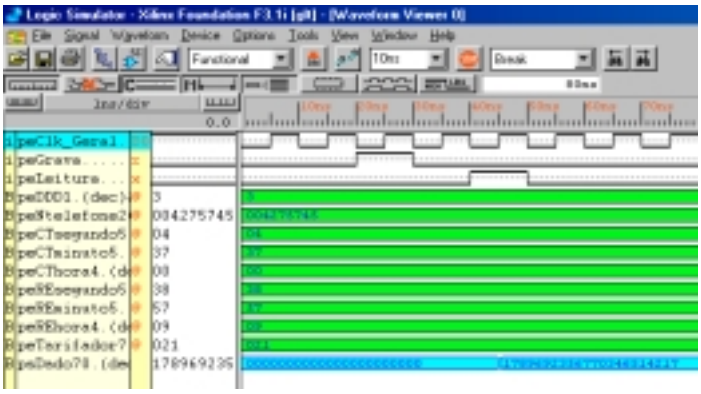


Figura 5.11 – Simulação do módulo memória

5.5 - DESCRIÇÃO E SIMULAÇÃO GERAL DO GLT

Sua principal função é armazenar todos os dados relevantes realizados em uma determinada ligação, seu funcionamento é inicializado depois que a ligação realizada for atendida. A partir desse momento, todos os dados relevantes para o sistema são armazenados, tais como: (i) o código da cidade para onde a ligação foi feita; (ii) o número do telefone; (iii) o tempo total da ligação; (iv) o custo estimado da ligação realizada; (v) o horário em que foi feita a ligação; (vi) a data em que foi realizada a ligação. Todos esses dados são armazenados na memória do sistema.

Com base no clock geral, o sistema é ativado a partir do momento em que a ligação foi atendida (peSinal_Tel = 1). Os dados são calculados e armazenados para que possam estar prontos à serem gravados na memória.

Quando a ligação for finalizada (peSinal_Tel= 0), os dados ficam à espera da instrução (peGrava = 1) para que sejam armazenados na memória. Para a visualização dos dados armazenados, basta ativar a instrução de leitura (peLeitura = 1), então todos os dados relevantes para o sistema podem ser vistos em forma de um vetor, como ilustra a figura 5.12.

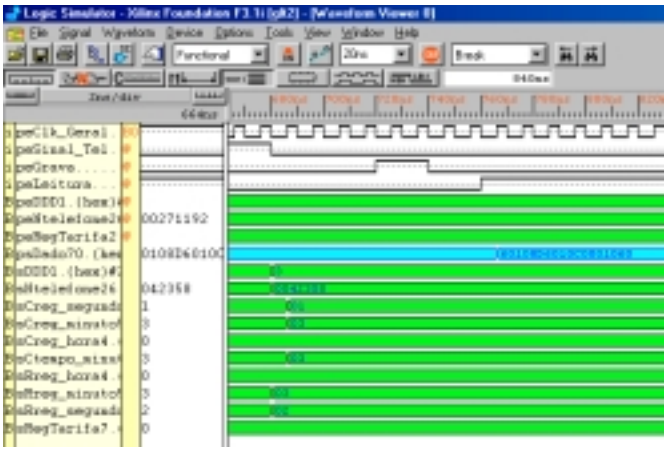


Figura 5.12 –Simulação do sistema GLT

5.6 - DESEMPENHO DO GLT - ESTATÍSTICAS DE RECURSOS UTILIZADOS NO FPGA

Na tabela 5.1 visualizam-se estatísticas de utilização no FPGA, dos principais módulos e do sistema como um todo. Estes dados mostram recursos citados a seguir: (i)

número de flip-flops; (ii) número de LUTs 3 entradas; (iii) número de LUTs 4 entradas; (iv) número de CLBs utilizadas.

A implementação foi realizada em um FPGA família XC4000, especificamente o FPGA XC4010 PC84, da Xilinx, que possui 800 Flip-Flops, 400 LUTS de 3 entradas, 800 LUTS de 4 entradas e, 400 CLBs.

Na tabela 5.1 percebe-se a quantidade que cada um dos módulos requer e indica-se também a porcentagem de utilização no FPGA como um todo.

Módulos	Número de Flip-Flops	Número de LUTs 3 entradas	Número de LUTs 4 entradas	Número de CLBs utilizadas
Receptor de Dados	**0** (0%)	**0** (0%)	**1** (0,25%)	**1** (1%)
Contador de Tempo	**71** (8,875%)	**16** (4%)	**141** (17,625%)	**77** (19,25%)
Tarifador	**0** (0%)	**3** (0,75%)	**37** (4,625%)	**22** (5,5%)
Calendário Tempo Real	**26** (3,25%)	**3** (0,73%)	**63** (7,875%)	**34** (8,5%)
Relógio Tempo Real	**62** (7,75%)	**13** (3,25%)	**151** (12,5%)	**85** (21,25%)
Memória	**75** (9,375%)	**41** (10,25%)	**85** (10,625%)	**43** (10,75%)
Sistema GLT	**134** (16,75%)	**19** (4,75%)	**242** (30,25%)	**151** (37,75%)

Tabela 5.1 – Recursos utilizados no FPGAs

Na tabela 5.1 observa-se também que o módulo que mais utilizou recurso no FPGA foi o relógio tempo real, obtendo 85 CLBs que representa 21,25% do total disponível no FPGA. O módulo memória utilizou 10,75% de CLBs e o módulo que utilizou menos recursos no FPGA foi o receptor de dados, utilizando menos de 1% do total de CLBs.

Interessante perceber, em relação aos dados da tabela 5.1, que somando os recursos utilizados no FPGA, de cada módulo separadamente, e comparando-os com os recursos utilizados no FPGA, pelo sistema como um todo, devidamente interconectado, tem-se uma grande otimização.

Observa-se que na somatória dos recursos dos módulos implementados, o número de CLBs utilizadas chegou a 228, o que representa 57% do total. Já no sistema completo, o número de CLBs é de 151, o que representa 37,75%. Observando estes dados nota-se uma otimização de 20%.

5.7 - DESEMPENHO DO GLT - ESTATÍSTICAS DE TEMPORIZAÇÃO

Os tempos gerados representam o tempo levado para executar todas as operações dos módulos implementados. Este tempo pode ser um dos elementos para

medir o desempenho de um sistema computacional. Assim, na tabela 5.2 demonstram-se os principais tempos dos módulos implementados separadamente e do sistema como um todo.

Módulos	Tempo de Lógica	Tempo de Roteamento	Atraso Total	Freqüência de Operação
Receptor de Dados	2,279 ns	5,695 ns	7,974 ns	122,699 Mhz
Contador de Tempo	12,247 ns	15,584 ns	27,831 ns	30,677 Mhz
Tarifador	2,279 ns	6,618 ns	8,897 ns	47,649 Mhz
Calendário Tempo Real	5,915 ns	8,454 ns	14,369 ns	69,594 Mhz
Relógio Tempo Real	11,629 ns	20,418 ns	32,047 ns	30,677 Mhz
Memória	3,530 ns	10,303 ns	13,833 ns	47,026 Mhz
Sistema GLT	12,729 ns	26,718 ns	39,447 ns	25,35 Mhz

Tabela 5.2 – Temporização dos principais módulos implementados

Ao fazer uma comparação entre a temporização do sistema como um todo e dos principais módulos implementados separadamente, percebe-se que o sistema em geral é bem mais rápido. Isto mostra que o sistema como um todo, devidamente interconectado, oferece uma grande otimização.

Observa-se que quando feita uma somatória das temporizações dos módulos implementados, o tempo de lógica é de 37,876 ns. Já no sistema como um todo, o tempo de lógica é de 12,729ns, o que corresponde a uma otimização de aproximadamente 68%. Similarmente acontece com o tempo de atraso total; se este fosse linear a todos os módulos, seria um total de 104,951ns, mas o sistema geral apresenta um atraso de 39,447ns (25,35 Mhz), o que representa uma otimização de 65%.

CAPÍTULO VI – CISIET - CONTROLE INTELIGENTE PARA SISTEMAS DE INJEÇÃO ELETRÔNICA DE TURBINAS

Neste capítulo apresenta-se uma proposta de um controle inteligente para sistemas de injeção eletrônica de turbinas (CISIET). Este atua no motor da bomba de combustível, respeitando parâmetros essenciais, adquiridos através de sensores que medem temperatura e rotações por minuto. O CISIET controla a turbina e seus limites de funcionamento, aumentando a vida útil e a segurança na manipulação do equipamento.

6.1 - INTRODUÇÃO

Geralmente uma turbina possui três fases de operação: (i) inicial, até conseguir o arranque, (ii) operação, onde sofre mudanças de velocidade para se adequar a uma carga de trabalho e (iii) final, para finalizar operação. Projetar adequadamente uma turbina é um trabalho difícil. O engenheiro Milton Souza Sanches conseguiu desenvolver uma turbina totalmente brasileira, a TS 65, após 10 anos de pesquisa. Apesar disso, o controle de operação ainda é manual e o correto funcionamento depende do bom senso e profundo conhecimento técnico da turbina.

Milton Souza Sanches vinha, desde fins da década de 1980, fazendo suas pesquisas e otimizando seu projeto. Mesmo com seus escassos recursos pessoais, fabricou os caros componentes individuais do primeiro protótipo da turbina. Apesar das dificuldades, o pesquisador percorreu um longo caminho de mais de dez anos de tentativas e erros, testando materiais, configurações e componentes.

Finalmente, em janeiro de 2000, a primeira turbina experimental Turbo Sanches (TS 65) funcionou com sucesso e apresentou um empuxo de 2800 N. Propelindo um aeromodelo em mais de cinqüenta vôos, demonstrando a viabilidade preliminar da turbina a jato genuinamente brasileira. Entretanto, trata-se de um protótipo experimental, com todo um caminho de ensaios e testes ainda a percorrer até a construção de uma versão comercial. O protótipo da turbina TS 65 pode ser visto na figura 6.1.

Nestes vôos preliminares, não existia controle e monitoramento eletrônico sobre o funcionamento geral da turbina. Desta maneira, sua vida útil fica bastante limitada, pois existem parâmetros cruciais de temperatura e rotação máxima que devem ser respeitados.

Figura 6.1 – Protótipo da turbina TS 65

Com um possível controle eletrônico, aumenta-se consideravelmente a vida útil da turbina, pois os limites são respeitados através do controle do fluxo de combustível. Esse controle é chamado de "CISIET", controle inteligente para sistema de injeção eletrônica de turbinas.

O CISIET utiliza um FPGA da família Virtex de 100K *gates*. Este dispositivo tem como característica uma grande capacidade física para abrigar circuitos lógicos complexos e alcança um alto desempenho temporal.

6.2 - DESCRIÇÃO GERAL DO FUNCIONAMENTO DA TURBINA

Partindo-se de uma primeira situação em que a turbina se encontra desligada, é necessário uma seqüência de passos para que esta venha a acionar. Esta seqüência deve ser respeitada. A seguir, a seqüência inicial de passos:

- acender uma vela de filamento, para queima do gás;
- liberar gás nos bicos injetores de combustível;
- acionar um "motor de arranque";
- certificar se ocorreu o acendimento inicial, senão retorna à fase preliminar;
- comutar o combustível gás para o combustível querosene;

Após o último passo, a turbina funciona em "marcha lenta" ou rotação mínima, e tem-se uma segunda situação, onde a preocupação principal é respeitar os limites de aceleração (injetando combustível rapidamente, há o risco de explosão da turbina) e desaceleração do equipamento (diminuindo-se a rotação da turbina rapidamente tem-se o risco da danificação dos rolamentos internos devido ao não resfriamento dos mesmos).

Esta situação é a mais complexa, pois devem ser considerados inúmeros parâmetros para o funcionamento correto da bomba de combustível, de maneira a ter um comportamento regular, isto é, requisições de mudanças bruscas de aceleração/desaceleração devem ser ignoradas pelo sistema e realizadas de modo suave.

Existe uma terceira e última situação quando se desliga a turbina, onde se deve acionar o "motor de arranque" novamente para proporcionar o resfriamento de suas partes mecânicas internas.

Portanto o CISIET foi idealizado para assumir todo o controle desde a situação da partida da turbina, o controle efetivo sobre a aceleração e desaceleração, até seu devido resfriamento. Sendo a segunda situação a mais importante do sistema, que será tratada neste capítulo.

6.3 - DESCRIÇÃO GERAL DO FUNCIONAMENTO DO CISIET

Para o correto funcionamento da turbina, faz-se necessário o controle e monitoramento eletrônico da temperatura, que pode chegar a 700 ºC e rotação, que atinge 120 mil rotações por minuto (RPM) conjugado à requisição de aceleração ou desaceleração imposta pelo usuário.

A aceleração e desaceleração da turbina estão diretamente ligadas, respectivamente, ao maior ou menor fluxo de combustível enviado à turbina através da bomba de combustível. Desta forma, o sistema atua por PWM (Pulse Wide Modulation) sobre a bomba, aumentando ou diminuindo o fluxo de combustível de acordo com as circunstâncias de funcionamento.

O CISIET está dividido em componentes ou módulos para facilitar a especificação, implementação e manutenção. Os componentes estão representados na figura 6.2, os quais serão descritos a seguir.

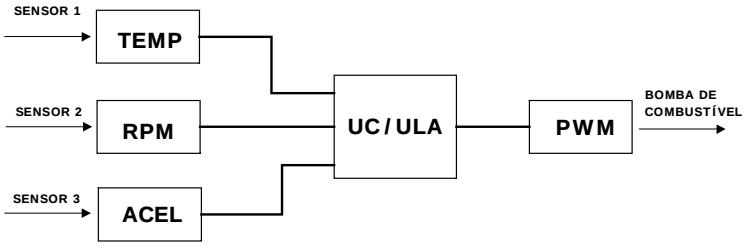


Figura 6.2 – Descrição geral dos componentes do CISIET

6.3.1 - COMPONENTE ACELERAÇÃO (ACEL)

Este componente tem a função de ler e linearizar as mudanças na requisição de aceleração. Quando o usuário requer mudanças suaves de aceleração, estas devem ser entregues à saída, pois assim esse processo não proporciona risco ao equipamento e ao próprio usuário. Porém, se o usuário requer mudanças severas (rápidas) é feito um

tratamento especial para adequar o pedido às possibilidades físicas de funcionamento da turbina garantindo a segurança.

O sinal da aceleração é composto por pulsos que variam de 1 até 2 milissegundos de duração, espaçados por uma distância de 18 milissegundos (período). Estes valores podem ser observados na figura 6.3.

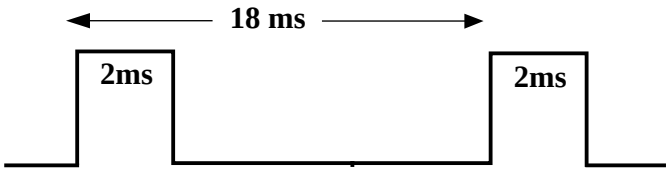


Figura 6.3 – Visualização da temporização do sinal de aceleração

A requisição de aceleração ou desaceleração é feita no rádio controle pelo usuário de forma aleatória e, na maioria das vezes, de forma brusca fazendo com que a turbina se danifique. As variações bruscas de requisição devem ser controladas, para adequar o pedido às possibilidades físicas de funcionamento da turbina garantindo a segurança.

Para garantir o controle da aceleração requisitada, existem dois registradores, os quais serão atualizados em um período fixo e conhecido de tempo, incrementando ou decrementando linearmente a aceleração atual da turbina.

O registrador de índice *RI* é responsável por capturar o sinal de aceleração a cada 0,2s. O RI funciona como uma amostra do sinal de aceleração requisitada, e é utilizado pelo registrador de sinal (*RS*) como base para detectar se a aceleração linear irá incrementar, decrementar ou manter.

O registrador de sinal *RS* é responsável por implementar a linearidade do sinal de aceleração. A sua função é incrementar, decrementar ou manter o sinal de aceleração linear a cada 0,2s. A decisão de modificar o sinal de aceleração linear é influenciada diretamente pelo RI. Se o valor do RI é maior que o valor do RS, o sinal de aceleração é incrementado; se o RI é menor que o valor do RS, o sinal é decrementado, senão o sinal de aceleração linear é mantido; lembrando que esta comparação é feita a cada 0,2s. Na figura 6.4, visualiza-se o funcionamento do cálculo de linearidade.

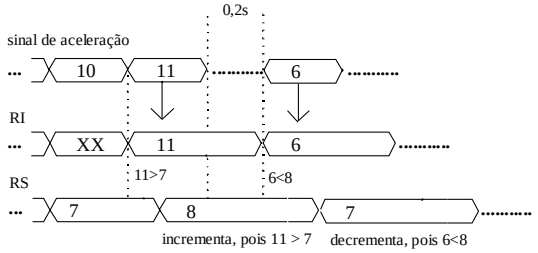


Figura 6.4 – Descrição do funcionamento do cálculo de linearidade

6.3.2 - COMPONENTE TEMPERATURA (TEMP)

Este componente tem a função de capturar os dados de um conversor A/D ligado a um termopar tipo K. As entradas do componente são: clock, entrada do A/D (8 bits) e pinos de controle do A/D. A saída do componente indica o valor da temperatura, a qual pode oscilar de 0 a 800ºC. Este componente é responsável por capturar a temperatura de funcionamento da turbina, para que esta seja utiliza pelo controle geral do CISIET.

Para o início do funcionamento é feita uma requisição ao A/D que está pré-configurado; assim o A/D responde e disponibiliza o valor da temperatura para captura. Isto leva um tempo para executar. Na figura 6.5, pode ser visualizada uma representação da requisição e captura do dado.

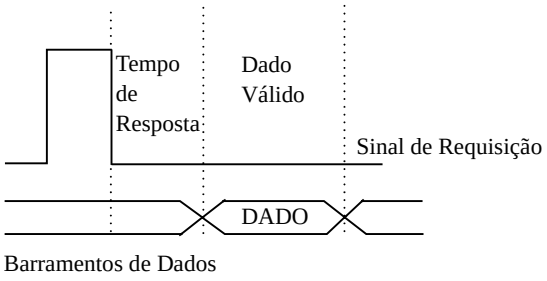


Figura 6.5 – Representação da requisição e captura do dado

Neste contexto, deve-se dar uma atenção especial ao tempo de resposta de uma requisição, para capturar o dado apenas quando ele for válido, para isto, deve-se calcular o tempo de resposta que geralmente se resume no tempo que o A/D leva para disponibilizar o dados convertidos.

Logo após definir o tempo de resposta de uma requisição (varia de A/D para A/D), deve-se apenas disparar um *timer* quando a requisição for realizada e, quando o *timer* finalizar o dado, está pronto para a captura, lembrando que o *timer* deve representar o tempo de resposta exato.

No mercado existem A/Ds que possuem a característica de conversão automática. Estes A/Ds utilizam um sinal de habilitação para indicar que o dado está pronto para captura. Esta característica elimina o *timer* e o sinal de requisição antes utilizados.

6.3.3 - COMPONENTE RPM

Este componente tem a função de ler pulsos de freqüência provenientes de um sensor externo que captura os ciclos de rotação da turbina, pois sabendo a freqüência, tem-se a rotação por minuto (RPM), bastando multiplicar a freqüência medida por 60.

Para o cálculo da freqüência, é utilizado o método de contagem de pulsos em uma base de tempo conhecida. Por exemplo: tendo-se uma base de tempo de 1 segundo, é só contar os pulsos a serem medidos, os quais correspondem à freqüência real, conforme ilustra a figura 6.6 .

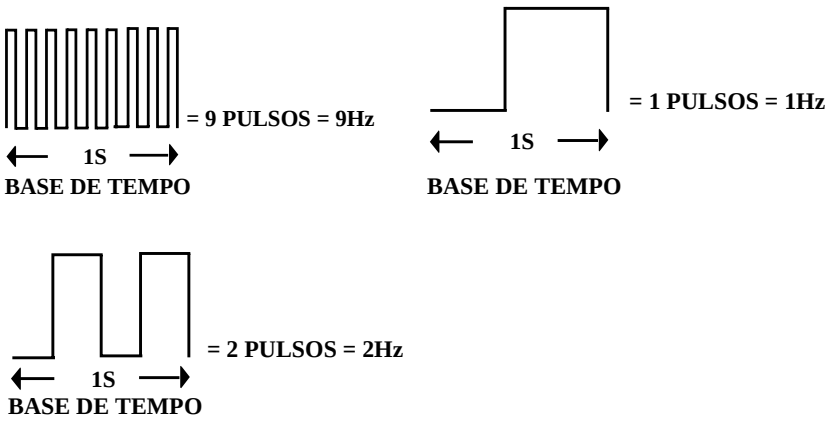


Figura 6.6 - Contagem de pulsos em uma base de tempo conhecida.

O problema é que esperar 1 segundo para se ter a resposta do valor da freqüência, pode ser um tempo muito longo. Neste contexto, a solução é reduzir o

tempo de espera, obtendo-se uma resposta mais rápida. Ao diminuir a base de tempo para 0,1 segundos, pode-se obter o valor da freqüência mais rapidamente.

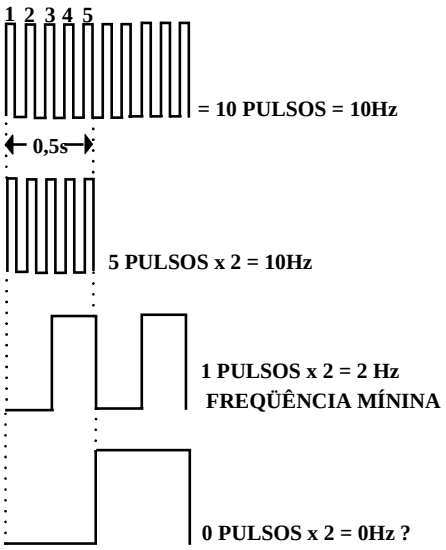


Figura 6.7 – Freqüência mínima medida.

Deve-se então, multiplicá-lo pelo fator de redução da base de tempo de 1 segundo. Por exemplo: 1 segundo foi dividido por 10 para se obter 0,1 segundos de amostra, e neste período foram contados 4 pulsos; multiplica-se então por dez: 10x4=40Hz.
Se necessário, pode-se reduzir mais ainda a base de tempo, para se obter uma resposta mais rápida ainda, lembrando que será necessário multiplicar os pulsos pelo fator de redução. Há um problema na redução do tempo de amostra: a mínima freqüência detectada medida é o valor da redução da base. Ver figura 6.7.

Para gerar uma base de tempo de 0,1 segundo, tendo o clock geral do sistema de 12MHz (12 milhões de pulsos por segundo) significa que 1,2 milhões de pulsos eqüivalem a 1/10 de segundo. Assim basta disparar um contador de pulsos do clock geral para gerar a base de tempo.

Existe um problema: a base de tempo da contagem não pode ser gerada continuamente, sendo necessário dispará-la ao primeiro pulso da freqüência a ser medida. Em uma base de tempo fixa, pode-se perder pulsos devido a falta de sincronismo, conforme a figura 6.8.

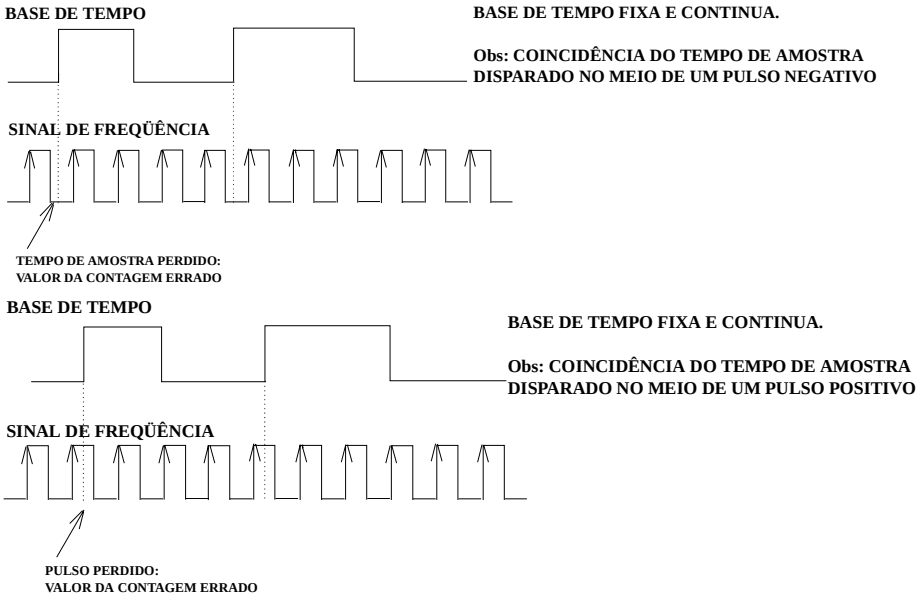


Figura 6.8 - Erro na contagem com base de tempo fixa.

Assim é necessário disparar a base de tempo ao mesmo instante em que for detectado um pulso na entrada. A função do componente RPM é capturar, detectar e sinalizar se a freqüência medida está abaixo ou acima do limite de 120 mil RPM ou 2000 HZ, gerando uma saída em nível 0 (zero) ou 1 (um) para representar esta situação. A seguir, o código VHDL deste componente.

Componente RPM

```
library ieee;
use ieee.std_logic_1164.all;

entity rpm is
   port (clk : in std_logic;
         rst : in std_logic;
         rpm : in std_logic;
         saida : out std_logic);
end rpm;

architecture rpm_arch of rpm is

constant reset : std_logic_vector(1 downto 0) :="00";
constant liberado : std_logic_vector(1 downto 0) :="01";
constant conta : std_logic_vector(1 downto 0) :="10";
constant mostra : std_logic_vector(1 downto 0) :="11";
```

```
     signal sum, enable , ssaida : std_logic;
signal cont_amostra : integer range 0 to 600000;
signal conta_rpm : integer range 0 to 10000;
signal estado : std_logic_vector(1 downto 0);

begin

 saida <=ssaida;

 process (clk, rst, estado)
 begin
        if rst ='1' then
            estado <= reset;
            elsif clk'event and clk='1' then
            case estado is
                  when reset =>
                        cont_amostra <=0;
                        sum <='0';
                        estado <= liberado;
                  when liberado =>
                        if enable ='1' then
                          estado<= conta;
                        else
                          estado<= liberado;
                       end if;
                  when conta =>
                       cont_amostra <= cont_amostra+1;
                       if cont_amostra <= 600000 then
                         sum <='1';
                         estado <= conta;
                       else
                         sum <='0';
                         estado <= mostra;
                      end if;
                  when mostra =>
                       if conta_rpm < 200 then
                         ssaida <='0';
                       else
                         ssaida <='1';
                       end if;
                       estado <= reset;
                   when others => estado <= reset;
             end case;
             end if;
  end process;

  process (rpm, rst, estado)
  begin
        if estado = reset then
          enable<='0';
          conta_rpm <=0;
        elsif rpm'event and rpm='1' then
            enable<='1';
            if sum = '1' then
              conta_rpm <= conta_rpm+1;
            end if;
        end if;
end process;
```

```
end rpm_arch;
```

6.3.4 - COMPONENTE DE CONTROLE – UC/ULA

Este componente tem a função avaliar e calcular qual será a potência a ser entregue à bomba de combustível, controlando a aceleração ou desaceleração segura do equipamento. Os limites máximos e de falha da turbina devem ser respeitados para que não haja complicações. A UC/ULA é o principal componente do sistema e recebe informações discretas de todos os módulos periféricos , atuando diretamente sobre o fluxo de combustível corrente na bomba. O componente UC/ULA é responsável por monitorar três situações distintas:

- Aceleração regular: nesta situação o usuário está acelerando ou desacelerando dentro dos limites do equipamento. O sistema detecta e não interfere na requisição imposta pelo usuário.
- Temperatura alta: A turbina atinge altas temperaturas durante seu funcionamento. O componente temperatura indica a situação corrente a UC/ULA, sendo que esta assume a responsabilidade de monitorar a temperatura. Em caso de temperaturas altas, diminui-se a potência entregue à bomba de combustível, com o intuito de diminuir a pontência da turbina e, consequentemente, a temperatura.
- Rotação excessiva: O componente RPM informa a UC/ULA o número de rotações por minuto que a turbina está desenvolvendo. O UC/ULA detecta o excesso de rotação, que pode danificar os rolamentos da turbina, e atua na bomba de combustível diminuindo a rotação da turbina.

O funcionamento, de forma simplificada, é a captura dos dados dos módulos aceleração, freqüência e temperatura, executando um algoritmo de filtro que, por sua vez, gera um valor de aceleração adequada para o componente PWM atuar diretamente sobre a bomba de combustível.

6.3.5 - COMPONENTE PWM

Antes de detalhar o componete PWM é interessante definir o que vem a ser o pulso PWM (*Pulse Wide Modulation*). Este tipo de sinal é bastante utilizado para controle de aceleração e desaceleração de motores e controle geral. Na figura 6.9 tem-

se um exemplo de um pulso PWM com a freqüência de 1 Hz. O pulso PWM (a) atuando em um motor poderia fazer este funcionar a metade (50%) de sua potência máxima de giro. De forma similar os pulsos PWM (b,c,d) fazem o motor funcionar com as respectivas potências, 25%, 0% e 100%.

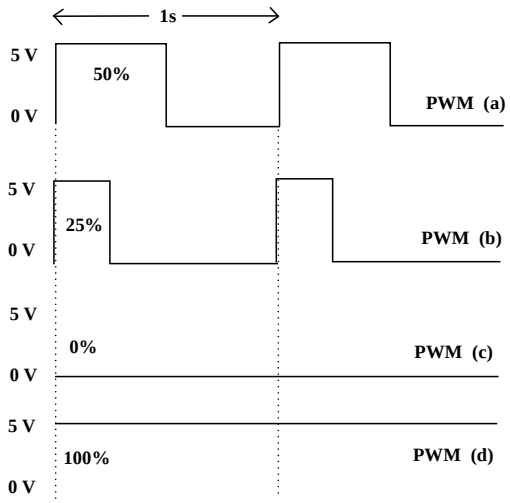


Figura 6.9 – Representação de pulsos PWM

O componente PWM tem a função de produzir um sinal de acordo com os parâmetros que receber do componente UC/ULA. No modo inoperante não produz PWM para a bomba de combustível; em modo de operação normal, o mínimo valor para o PWM deve ser correspondente ao fluxo mínimo de combustível necessário para o funcionamento da turbina em "marcha lenta", o que corresponde a 25% de sua pontência total.

O pulso PWM gerado para bomba de combustível possui algumas carcteristicas, visualizadas na figura 6.10:

- A base para os cálculos é uma onda de 100 MHz (A), clock máximo suportado pelo FPGA utilizado. Contando-se o número de pulsos desta onda é possível gerar diferentes PWMs;
- A freqüência do PWM deve ser de 2 KHz (B);
- A aceleração mínima corresponde a 25% do PWM. (C);

- Acima da aceleração mínima tem-se 27 estágios de aceleração até atingir a aceleração máxima. (D)

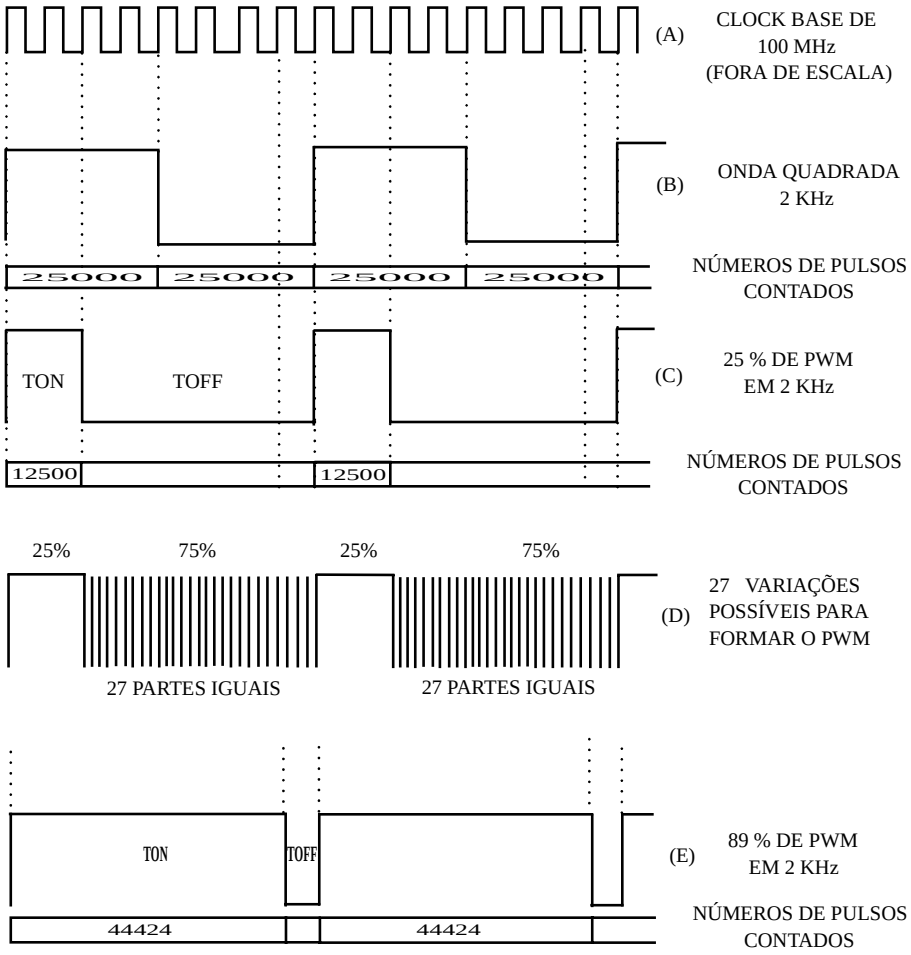


Figura 6.10 – Caracteristicas do PWM utilizado no CISIET

Na figura 6.10, pode-se visualizar dois exemplos de ondas de PWM, geradas através do CISIET. Em (C), é representada a onda mínima necessária para atuar sobre a bomba de combustível, fazendo com que esta trabalhe a 25% de sua potência. Com este pulso, a rotação da turbina se mantém estável em sua “marcha-lenta”. Em (E), visualiza-se o pulso que faz a bomba trabalhar a 89% de sua potência.

A aceleração máxima consome totalmente o pulso PWM, gerando um sinal contínuo. A seguir a descrição VHDL do componente PWM.

PWM

```
library ieee;
use ieee.std_logic_1164.all;

entity pwm is
port (peClk: in std_logic;
      peReset : in std_logic;
      pePWM  : in integer range 1 to 27;
      psPWM   : out std_logic);

    constant reset : std_logic_vector(1 downto 0):="00";
    constant calcular : std_logic_vector(1 downto 0):="01";
    constant gerar : std_logic_vector(1 downto 0):="10";

end pwm;

architecture comp of pwm is
signal estado :  std_logic_vector(1 downto 0);
signal timeacel:integer range 1 to 27;
signal um:integer;
signal scont: integer;
begin

process(peClk,pePWM,peReset)
begin
  if peReset ='1' then
     estado <= reset;
  elsif peClk'event and peClk='1' then
     case estado is
       when reset =>
          timeacel <= pePWM;
          um <= 0;
          estado<=calcular;
       when calcular =>
          um <= ((timeacel*1388)+12500);
          if timeacel=1 then
             um <=12500;
         end if;
          estado <= gerar;
       when gerar =>
          scont<= scont +1;
          if scont <= um then
            psPWM<='1';
            estado <=gerar;
         elsif scont<50000 then
            psPWM<='0';
           estado <=gerar;
         else scont <= 0;
.              estado <= reset;
```

```
        end if;
        when others => estado <= reset;
      end case;
   end if;
end process;
end comp;
```

6.4 - ESTATÍTICAS DE DESEMPENHO DO CISIET

Na tabela 6.1 tem-se as estatísticas de desempenho do sistema CISIET. A freqüência de aproximadamente 45MHz é suficiente para que o sistema tenha um bom desempenho, dado que esta aplicação não necessita de um processamento de alta performance.

Descrição do Recurso Utilizado	Recurso Usado	Utilização no FPGA
Número de Slices	724 de1,200	60 %
Número de Slice Flip Flops	174 de 2,400	7 %
Número total de LUTS 4 entradas	1.020 de 2.400	42 %
Número de IOBs	18 de 166	10 %
Número de GCLKs	1 de 4	25 %
Número de GCLKIOBs	1 de 4	25 %
Temporização	22,089 ns	**Frequência Máxima** 45.271MHz

Tabela 6.1 - Descrição dos recursos que o CISIET utiliza no FPGA VIRTEX 100K

CAPÍTULO VII – PROJETO E DESEMPENHO DE PROCESSADORES EM FPGAS

Neste capítulo é discutido um sistema digital complexo, uma unidade central de processamento (UCP) de 8 bits, a qual é implementada utilizando duas diferentes metodologias: microprogramada e com máquina de estados finita. Para cada UCP implementada são gerados dados estatísticos de desempenho com o propósito de comparar e eleger a melhor metodologia aqui adotada para o desenvolvimento de UCP. Ainda neste capítulo é comparado o desempenho de uma UCP de 8 com uma de 16 bits e também implementada a UCP do microcontrolador HC05 da Motorola.

7.1 - INTRODUÇÃO

Neste capítulo apresenta-se um sistema digital complexo, uma unidade central de processamento (UCP) de 8 bits, implementada utilizando duas diferentes metodologias:

- Microprogramada: descrição onde a arquitetura é controlada através de sinais ou "gatilhos", sendo estes responsáveis pelo fluxo de execução da UCP. O conjunto de sinais de controle é chamado de microinstruções, normalmente armazenadas em uma memória interna da UCP.
- Máquina de estados finita (FSM): este tipo de descrição possibilita ao projetista de hardware uma maior abstração na descrição do comportamento de um determinado circuito. Esta metodologia tem como base estados que representam todas as fases de uma lógica predeterminada de execução.

Para cada UCP implementada, são gerados dados estatísticos de desempenho com o propósito de comparar e eleger a melhor metodologia aqui adotadas para o desenvolvimento de UCP. Ainda neste capítulo, é comparado o desempenho de uma UCP de 8 com uma de 16 bits e também é implementada a UCP do microcontrolador HC05 da Motorola.

A UCP é o elemento principal do computador. Sua função é executar programas armazenados na memória principal, buscando, examinado e executando uma instrução após a outra. A UCP é composta por módulos distintos, comuns em qualquer arquitetura:

- Unidade de controle (UC): responsável por executar o ciclo de busca-decodificação-execução de instruções.
- Unidade lógica e aritmética (ULA) - faz operações de adição, subtrações, and, or, dentre outras necessárias para execução das instruções.
- Contador de programa (PC): é um índice de referência para a próxima instrução a ser executada.
- O Registrador de instrução (RI) – armazena temporariamente a instrução, que está

sendo executada.

EXECUÇÃO DE INSTRUÇÕES (PASSO A PASSO)

A seguir tem-se a execução, passo a passo, de instruções, desde a busca na memória principal até a decodificação e execução, armazenando o(s) resultado(s) em locais apropriados.

1. Busca a próxima instrução da memória e envia ao registrador de instrução (RI).
2. Atualiza o contador de programa (PC) para que este referencie a instrução seguinte.
3. Determina o tipo da instrução.
4. Se a instrução usa dados da memória, determina em que posições eles estão.
5. Busca os dados, se houver algum, e envia aos registradores internos da CPU.
6. Executa a instrução.
7. Armazena resultado(s) em determinadas posições de memória ou registradores da UCP.
8. Volta ao passo 1 para iniciar a execução da próxima instrução.

Esta seqüência de passos é freqüentemente referida como ciclo de **"busca-decodifica-executa"**. Este ciclo é o centro da operação de todos os computadores. Na figura 7.1, ilustra-se a representação deste.

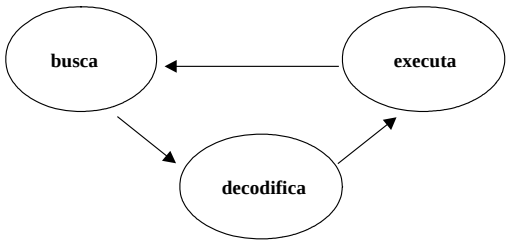


Figura 7.1 – Ciclo de busca-decodifica-executa instruções

Desenvolver um sistema complexo, como uma UCP em VHDL, exige desde um conhecimento aprofundado da linguagem até o conhecimento geral e específico do funcionamento de todo o sistema.

A seguir os módulos principais da UCP de 8 bits proposta:

- Registrador de instrução (RI);
- Registradores de dados , A (RegA) e B (RegB);
- Decodificador de instruções (DEC);
- Multiplexadores (MUX) que auxiliam no controle do fluxo de informação;
- Unidade lógica e aritmética (ULA);
- Contador de programa (PC);
- Registrador de endereço (REND);
- Comparador (COMP);
- Unidade de controle (UC).

Cada um destes módulos unidos por uma lógica de controle compõe uma UCP. Inicialmente será apresentado o funcionamento geral do sistema e posteriormente apresenta-se o funcionamento detalhado e união de todos os módulos.

7.2 - INSTRUÇÕES

Esta UCP de 8 bits é relativamente simples, composta por 16 instruções básicas. A tabela 7.1 mostra as 16 instruções que a UCP é capaz de executar. Cada instrução possui um **opcode** ou código de operação. Tendo um conjunto de 16 instruções, são necessários 4 bits (2^4=16) de opcode para diferenciar cada uma delas.

Nº	Opcode	Instrução	Descrição
1	0000	MOV A,DADO	A recebe DADO
	0000	MOV B,DADO	B recebe DADO
	0000	MOV A,[REND]	A recebe o conteúdo do endereço [REND]
	0000	MOV B, [REND]	B recebe o conteúdo do endereço [REND]
	0000	MOV [REND],A	Endereço [REND] recebe A
	0000	MOV [REND],B	Endereço [REND] recebe B
2	0001	ADD A,B	A recebe A+B
3	0010	SUB A,B	A recebe A-B
4	0011	OR A,B	A recebe A or B
5	0100	AND A,B	A recebe A and B
6	0101	XOR A,B	A recebe A xor B
7	0110	NAND A,B	A recebe A nand B
8	0111	NOR A,B	A recebe A nor B
9	1000	XNOR A,B	A recebe A xnor B
10	1001	JUMP [REND]	JUMP para o endereço [REND]
11	1010	INC	Incrementa A
12	1011	DEC	Decrementa A
13	1100	CMP A, B	Compara A com B
14	1101	JG [REND]	JUMP se maior para o endereço [REND]

15	1110	JL [REND]	JUMP se menor para o endereço [REND]
16	1111	JZ [REND]	JUMP se zero para o endereço [REND]

Tabela 7.1 – Instruções da UCP

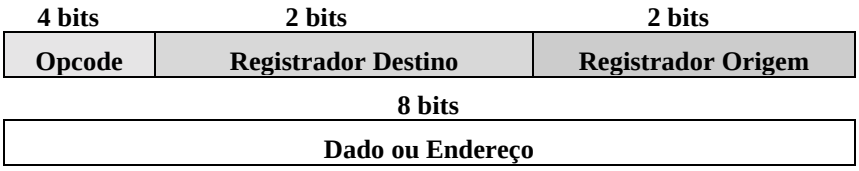
FORMATO DAS INSTRUÇÕES

Uma instrução é um conjunto de bits que são decodificados pela UCP a fim de identificar e executar uma determinada operação. Esse conjunto de bits pode ter um ou mais formatos. Nesta UCP, tem-se instruções de três formatos diferentes, apresentados a seguir:

- **Formato 1 –** Utiliza 16 bits, e todos são relevantes. Os primeiros 4 bits são destinados ao **opcode** da instrução, os próximos 2 bits representam o registrador destino e os últimos 2 bits o registrador origem. Esses registradores são identificados através de códigos, apresentados a seguir.

O registrador origem e destino são identificados pela seguinte codificação:

Registrador A → 00 **Registrador Rend → 10**
Registrador B → 01 **Dado → 11**



Este formato de instrução está ligado a operações de movimentação de dados entre a UCP e a memória. A seguir, as instruções que utilizam este formato, onde K é um dado de 8 bits e [REND] um endereço de memória.

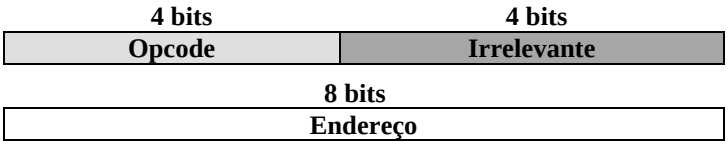
Listagem de instruções que utilizam o formato 1
MOV A,K MOV B,K MOV [REND],A MOV [REND],B MOV A,[REND] MOV B,[REND]

- **Formato 2 –** Utiliza 8 bits onde os 4 últimos bits são irrelevantes. Neste formato, é possível identificar e executar a instrução desejada com os primeiros 4 bits; os demais são irrelevantes e desprezados pela UCP.

Este formato de instrução está relacionado com operações lógicas e aritméticas executadas pela ULA.

Listagem de instruções que utilizam o formato 2
ADD A,B SUB A,B OR A,B AND A,B XOR A,B NAND A,B NOR A,B XNOR A,B INC DEC CMP A,B

Formato 3 – Utiliza 16 bits, onde os 4 primeiros determinam o opcode, os 4 seguintes são irrelevantes e os últimos 8 bits são destinados a um endereço de memória.



Este formato está relacionado a instruções de *jump* (salto) para um endereço específico.

Instrução que utiliza o formato 3
JMP [REND] JG [REND] JL [REND] JZ [REND]

Independente do formato da instrução o **opcode** está sempre presente, pois, através dele identifica-se a instrução que será executada. Os demais bits são utilizados para representar os registradores que serão utilizados ou os dados e endereço que fazem parte da instrução. Existe ainda bits **irrelevantes,** que são simplesmente desprezados.

7.3 – VISÃO GERAL DA ARQUITETURA

A figura 7.2 representa todos os módulos da UCP e suas interconexões necessárias para o correto funcionamento. É possível identificar todos os módulos através das abreviações. Na figura 7.2 os sinais de controle ou “gatilhos” estão

numerados, (Exemplo: A(0), A(9:6)*, etc.). Estes valores, são reais e utilizados pela unidade de controle (UC) para sincronizar o fluxo de informações e executar as tarefas desejadas.

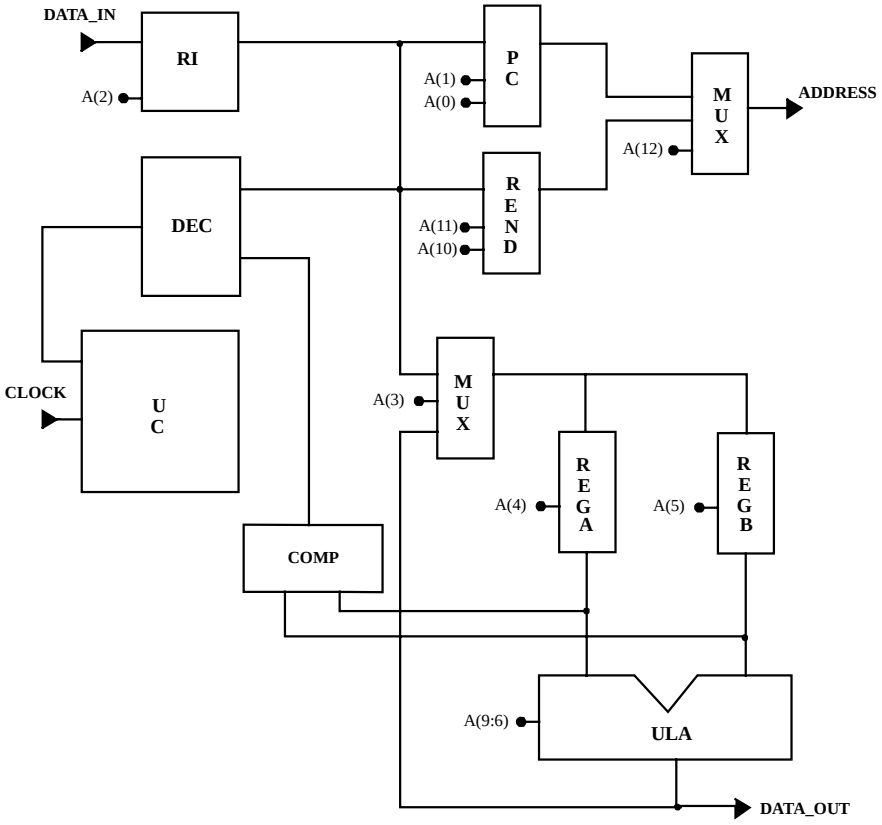


Figura 7.2 – Visão Geral da Arquitetura

*** A(9:6)** : significa que A é um vetor de 4 bits começando no bit 9 e terminado no bit 6.

PORTAS DE ENTRADA DA ARQUITETURA

- **data_in** → dados provenientes da memória principal.
- **clock** → clock geral do sistema

PORTAS DE SAÍDA DA ARQUITETURA

- **data_out →** dados que serão armazenados na memória principal
- **rw →** habilita leitura e escrita na memória principal
- **address →** acesso a um endereço da memória principal

A seguir a descrição do funcionamento básico da arquitetura, mostrando os passos necessários para execução de qualquer instrução nesta arquitetura (figura 7.2).

1. Busca a instrução apontada pelo contador de programa (PC), armazenado-a no registrador de instruções (RI).
2. Incrementa contador de programa, apontando para a próxima instrução.
3. Decodifica a instrução armazenada no registrador de instrução.
4. A unidade de controle recebe a instrução decodificada e dispara a execução.
5. Resultados são armazenados nos locais apropriados.
6. Retorna a busca de uma nova instrução (passo 1).

As seções 7.4 e 7.5 apresentam as metodologias, microprogramada e com máquinas de estados finita, para descrição da arquitetura anteriormente proposta.

7.4 – MÓDULOS DA UCP MICROPROGRAMADA

A UCP microprogramada pode ser dividida em módulos que se comunicam entre si para a realização de uma tarefa específica. Os módulos são interdependentes e podem ser controlados por um vetor de bits de controle que são distribuídos pela unidade central de processamento, o valor de cada bit de controle é determinado pelo módulo unidade de controle (UC) e o conjunto total desses valores determinam o funcionamento da UCP.

Os módulos que compõem a UCP são: Módulo Registrador, Mutiplexador (MUX), Comparador (COMP), Unidade Lógica e Aritmética (ULA), Contador, Unidade de Controle (UC) e Decodificador (DEC). A seguir, a definição funcional de cada módulo e sua respectiva descrição VHDL.

7.4.1- MÓDULO REGISTRADOR

Os registradores são utilizados para sincronizar o fluxo de dados armazenando-os no momento adequado. Na figura 7.3, as entradas do registrador de 8

bits são indicadas por *E1* e *CLK* e a saída por *S1*. A saída *S1* receberá o conteúdo de *E1* somente se o *CLK* for ativado. Três registradores de 8 bits são utilizados na UCP:

- **Registrador A** - Alimentação da ULA, armazena todos os resultados de operações realizadas pela unidade lógica e aritmética e dados provenientes da memória principal.
- **Registrador B** - Alimentação da ULA, armazena somente dados provenientes da memoria principal.
- **Registrador de Instrução** - Este registrador possui a instrução ou parte dela, que está sendo executada pela unidade central de processamento.

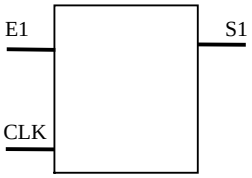


Figura 7.3 – Representação do registrador de 8 bits

A seguir a descrição VHDL de um registrador de 8 bits utilizado na UCP proposta.

Registrador de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
     port (e1 : in  std_logic_vector(7 downto 0);
           clk : in  std_logic;
           s1 : out std_logic_vector(7 downto 0));
end reg;

architecture reg_arch of reg is

begin
     process (clk)
     begin
       if (clk'event and clk='1')
          then s1 <= e1;
       end if;
   end process;

end reg_arch;
```

7.4.2 – MÓDULO MULTIPLEXADOR

O multiplexador 2x1 de 8 bits é utilizado para determinar qual informação deverá ser selecionada e desempenha um papel importante para o fluxo correto do processamento. Na figura 7.4 tem-se a representação do multiplexador 2x1 com suas portas de entrada *E1, E2* e *SEL* e saída *S1*.

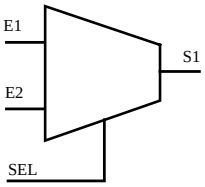


Figura 7.4 – Representação do multiplexador 2x1

O multiplexador funciona como um selecionador, onde um bit de controle *SEL* é testado e, conforme seu valor, uma das portas de entrada do multiplexador será atribuída à saída. Esse comportamento pode ser facilmente descrito em VHDL usando o comando condicional IF-THEN-ELSE para testar o bit de controle.

A seguir, a implementação VHDL de um multiplexador 2x1 de 8 bits. No capítulo III, tem-se diferentes descrições VHDL do multiplexador 2x1, qualquer uma delas pode ser adaptada e utilizada sem prejudicar o desempenho do sistema.

Multiplexador de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;

entity mux2 is
     port (e1 : in  std_logic_vector(7 downto 0);
           e2 : in   std_logic_vector(7 downto 0);
           sel : in   std_logic;
           s1 : out  std_logic_vector(7 downto 0));
end mux2;

architecture mux2_arch of mux2 is
begin
   process (sel,e1,e2)
   begin
      if (sel='0')
```

```
            then s1 <= e1;
            else s1 <= e2;
        end if;
    end process;
end mux2_arch;
```

7.4.3 – MÓDULO COMPARADOR

O comparador simplesmente compara os valores dos registradores A e B e retorna se A é maior, menor ou igual a B. É utilizado pela instrução CMP A, B. É importante observar que o comparador não possui *clock* de controle, basta que os valores de *E1* ou *E2* modifiquem para que uma nova comparação seja efetuada. Na figura 7.5, visualiza-se a representação do comparador com suas portas de entrada *E1* e *E2* e saída *S1*.

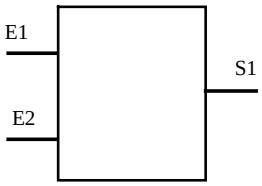


Figura 7.5 – Representação do comparador

A tabela 7.2 mostra as comparações que podem ser realizadas por este módulo. Como se pode notar, este é um comparador simplificado realizando apenas três comparações básicas.

Operação	Saída
E1>E2	**S1 recebe 00**
E1<E2	**S1 recebe 01**
E1=E2	**S1 recebe 10**

Tabela 7.2 – Comparações possíveis

A seguir, a descrição VHDL do comparador.

Comparador de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;
entity comp is
    port (e1: in std_logic_vector(7 downto 0);
```

```
            e2: in std_logic_vector(7 downto 0);
            s1: out std_logic_vector(1 downto 0));
end comp;
architecture comp_arch of comp is
begin
process (e1,e2)  begin
     if e1 = e2 then
        s1<="10";
     elsif e1 > e2 then
        s1<="00";
     else s1<="01";
     end if;
end process;
end comp_arch;
```

7.4.4 - MÓDULO ULA

A ULA é o módulo responsável por operações lógicas e aritméticas da UCP. A partir de um selecionador de operações *OP* é possível obter a operação desejada. A figura 7.6 representa uma unidade lógica e aritmética com suas portas de entradas *E1*, *E2* e *OP* e de saída *S1*.

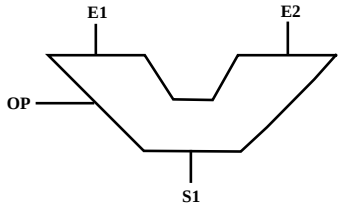


Figura 7.6 – Representação da ULA

A tabela 7.3 apresenta os códigos de operações (*opcode*) realizadas pela ULA e a operação correspondente. Esta ULA é capaz de executar 10 operações.

Nº	Opcode	Operação
Operações lógicas		
1	**0010**	**A** and **B**
2	**0011**	**A** or **B**
3	**0100**	**A** xor **B**
4	**0101**	**A** nand **B**
5	**0110**	**A** nor **B**
6	**0111**	**A** xnor **B**
Operações Aritméticas		
7	**0000**	**A** + **B**
8	**0001**	**A** – **B**

9	**1010**	**Incrementa A**
10	**1011**	**Decrementa A**

Tabela 7.3 – Operações Lógicas e Aritméticas

Unidade Lógica e Aritmética

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ula is
      port (e1 : in  std_logic_vector(7 downto 0);
            e2  : in  std_logic_vector(7 downto 0);
            op  : in  std_logic_vector(3 downto 0);
            s1 : out std_logic_vector(7 downto 0));
end ula;

architecture ula_arch of ula is

   -- constantes que representam as operacoes da ula
   constant addop  : std_logic_vector (3 downto 0) := "0000";
   constant subop  : std_logic_vector (3 downto 0) := "0001";
   constant andop  : std_logic_vector (3 downto 0) := "0010";
   constant orop   : std_logic_vector (3 downto 0) := "0011";
   constant xorop  : std_logic_vector (3 downto 0) := "0100";
   constant nandop : std_logic_vector (3 downto 0) := "0101";
   constant norop  : std_logic_vector (3 downto 0) := "0110";
   constant xnorop : std_logic_vector (3 downto 0) := "0111";
   constant incop  : std_logic_vector (3 downto 0) := "1010";
   constant decop  : std_logic_vector (3 downto 0) := "1011";

begin
    process (e1,e2,op)
    begin
        case op is
          -- selecao de operacao
          when addop => s1  <= e1  +  e2;
          when subop => s1  <= e1  -  e2;
          when andop => s1  <= e1  and  e2;
          when orop => s1   <= e1  or  e2;
          when xorop => s1  <= e1  xor  e2;
          when nandop => s1 <= e1  nand e2;
          when norop => s1  <= e1  nor  e2;
          when xnorop => s1 <= e1  xnor e2;
          when incop => s1 <= e1  + 1;
          when decop => s1 <= e1  - 1;
          when others => s1 <= (others =>'z');
       end case;
    end process;
end ula_arch;
```

A descrição VHDL da unidade lógica aritmética utiliza como base o comando CASE para selecionar a operação desejada. Quando um código de operação for inválido, é atribuído à saída da ULA uma seqüência de bits que representam um estado indefinido, implementado pelo comando **when others => s1 <= (others =>'Z')**.

7.4.5 - MÓDULO CONTADOR

O contador (*counter*) é um circuito sincronizado por um clock. A cada clock, sua saída é incrementada, gerando então uma seqüência de números (0,1,2,3,4..N). O contador implementado no projeto possibilita a contagem a partir de 0 (zero) ou de um número imposto pela UCP. A figura 7.7 ilustra o contador com suas portas de entrada e saída.

Os contadores são utilizados na UCP para fazer acesso a memória principal, tem-se dois contadores no projeto:

- **Contador de Programa (PC)** → Indica qual a próxima instrução a ser executada. O PC começa a incrementar a partir de 0 (zero) e vai até o fim da memória.
- **Contador REND** → utilizado pelas instruções que acessam informações em um endereço qualquer de memória. Ex: MOV A, [END] , MOV B, [END].

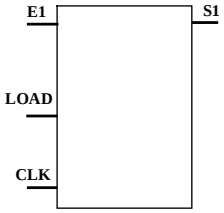


Figura 7.7 – Representação do contador

A seguir tem-se a descrição VHDL do contador de 8 bits.

Contador de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
    port (e1 : in  std_logic_vector(7 downto 0);
          clk  : in std_logic;
          load : in  std_logic;
```

```
            s1 : out std_logic_vector(7 downto 0));
end counter;

architecture counter_arch of counter is
signal cont : std_logic_vector(7 downto 0);
begin
    process (clk)
    begin
        if (clk'event and clk='1')
          then if (load = '0')
                then cont <= cont + 1;
                else cont <= e1;
              end if;
        end if;
    end process;
    s1 <= cont;
end counter_arch;
```

A descrição VHDL de um contador com opção de *LOAD*, exige a implementação de duas condições:

- a primeira condição é detectar o evento de borda de subida do clock do contador, através do comando clk'event and clk='1' .
- a segunda condição é testar o sinal de *LOAD*: se este for igual a 0 (zero), basta incrementar a saída, senão carrega (atribui) o valor da entrada do contador à saída.

7.4.6 – MÓDULO UNIDADE DE CONTROLE

Uma das partes da UCP que mais se destaca é a unidade de controle (UC). Ela é responsável pelo gerenciamento dos demais blocos tendo como lógica de base, microinstruções binárias, que têm a função de acionar cada bloco através de um vetor de bits de controle, fazendo com que todos os blocos se comuniquem em um determinado tempo, retornando a solução do problema imposto pela instrução. Na figura 7.8 tem-se a representação da unidade de controle com suas portas de entrada *E1* e *CLK* e saída *S1*.

A entrada *E1* é responsável por receber o endereço da microinstrução gerado pelo módulo decodificador. A entrada *CLK* da unidade de controle está diretamente ligada ao clock geral do sistema, e a saída *S1* representa o vetor de bits de controle.

Figura 7.8 – Representação da unidade de controle

A seguir, a descrição VHDL da unidade de controle.

Unidade de controle

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity control is
     port (clk      : in  std_logic;
           e1   : in  std_logic_vector(5 downto 0);
           s1     : out std_logic_vector(13 downto 0));
end control;

architecture control_arch of control is
signal i:integer;
type mem_rom is array (46 downto 0) of std_logic_vector (15 downto 0);
signal vd   : mem_rom;
begin

vd(00) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','1','0','0'); -- busca
vd(01) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','1'); -- decodifica
vd(02) <= ('1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0');
vd(03) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','1','1','0'); -- jmp [end]
vd(04) <= ('0','1','0','0','0','0','0','0','0','0','0','0','0','0','1','1');
vd(05) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','1','0','0'); -- mov a,dado
vd(06) <= ('0','1','0','0','0','0','0','0','0','0','1','0','0','0','0','1');
vd(07) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','1','0','0'); -- mov b,dado
vd(08) <= ('0','1','0','0','0','0','0','0','0','1','0','0','0','0','0','1');
vd(09) <= ('0','0','0','1','1','0','0','0','0','0','0','0','0','1','0','0'); -- mov a,[end]
vd(10) <= ('0','0','0','1','1','1','0','0','0','0','0','0','0','0','0','0');
vd(11) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','1','0','0');
vd(12) <= ('0','1','0','0','0','0','0','0','0','0','0','1','0','0','0','1');
vd(13) <= ('0','0','0','1','1','0','0','0','0','0','0','0','0','1','0','0'); -- mov b,[end]
vd(14) <= ('0','0','0','1','1','1','0','0','0','0','0','0','0','0','0','0');
vd(15) <= ('0','0','0','0','0','0','0','0','0','0','0','0','0','1','0','0');
vd(16) <= ('0','1','0','0','0','0','0','0','0','0','1','0','0','0','0','1');
vd(17) <= ('0','0','0','1','1','0','0','0','0','0','0','0','0','1','0','0'); -- mov [end],a
vd(18) <= ('0','0','0','1','1','1','1','0','0','0','0','0','0','0','0','0');
vd(19) <= ('0','0','1','1','0','0','1','0','0','0','0','0','0','0','0','0');
vd(20) <= ('0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','1');
vd(21) <= ('0','0','0','1','1','0','0','0','0','0','0','0','0','1','0','0'); -- mov [end],b
vd(22) <= ('0','0','0','1','1','1','1','0','0','1','0','0','0','0','0','0');
vd(23) <= ('0','0','1','1','0','0','1','0','0','1','0','0','0','0','0','0');
vd(24) <= ('0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','1');
vd(25) <= ('0','0','0','0','0','0','0','0','0','0','0','0','1','0','0','0'); -- add a,b
vd(26) <= ('0','1','0','0','0','0','0','0','0','0','0','1','1','0','0','0');
```

```
vd(27) <= ('0','0','0','0','0','0','0','0','0','1','0','0','1','0','0','0'); -- sub a,b
vd(28) <= ('0','1','0','0','0','0','0','0','0','1','0','1','1','0','0','0');
vd(29) <= ('0','0','0','0','0','0','0','0','0','1','0','0','0','1','0','0'); -- and a,b
vd(30) <= ('0','1','0','0','0','0','0','0','0','1','0','0','1','1','0','0');
vd(31) <= ('0','0','0','0','0','0','0','0','1','1','0','0','1','0','0','0'); -- or a,b
vd(32) <= ('0','1','0','0','0','0','0','0','1','1','0','1','1','0','0','0');
vd(33) <= ('0','0','0','0','0','0','0','0','1','0','0','0','0','1','0','0'); -- xor a,b
vd(34) <= ('0','1','0','0','0','0','0','0','1','0','0','0','1','1','0','0');
vd(35) <= ('0','0','0','0','0','0','0','0','1','0','1','0','0','1','0','0','0'); -- nand a,b
vd(36) <= ('0','1','0','0','0','0','0','0','1','0','1','0','1','1','0','0','0');
vd(37) <= ('0','0','0','0','0','0','0','0','1','1','0','0','0','1','0','0','0'); -- nor a,b
vd(38) <= ('0','1','0','0','0','0','0','0','1','1','0','0','0','1','1','0','0');
vd(39) <= ('0','0','0','0','0','0','0','0','1','1','1','0','0','1','0','0','0'); -- xnor a,b
vd(40) <= ('0','1','0','0','0','0','0','0','1','1','1','0','1','1','0','0','0');
vd(41) <= ('0','0','0','0','0','0','0','1','0','1','0','0','0','1','0','0','0'); -- inc a,b
vd(42) <= ('0','1','0','0','0','0','0','1','0','1','0','0','1','1','0','0','0');
vd(43) <= ('0','0','0','0','0','0','0','1','0','1','1','0','0','1','0','0','0'); -- dec a,b
vd(44) <= ('0','1','0','0','0','0','0','1','0','1','1','0','1','1','0','0','0');
vd(45) <= ('0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'); -- cmp a,b
vd(46) <= ('0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','1');

process (clk)
begin
   if (clk'event and clk = '1')
      then i <= i + 1;
      if (vd(i)(15) = '1') -- carrega nova instrucao
         then i <= conv_integer(e1);
      end if;
      if (vd(i)(14) = '1') -- novo ciclo de busca-decodifica
         then i <= 0;
      end if;
      s1 <= vd(i)(13 downto 0);
   end if;
end process;
end control_arch;
```

A unidade de controle como já foi citado é o módulo responsável pelo funcionamento do sistema. Isto é possível através de um vetor de bits de controle. Cada bit ou seqüência de bits está associado(a) a um módulo do sistema, viabilizando o controle do fluxo de informações dentro da UCP. A estrutura interna da UC é composta por uma memória onde estão armazenadas as microinstruções e uma lógica de controle interna alimentada pelo clock geral do sistema.

MEMÓRIA DE ARMAZENAMENTO DE MICROINSTRUÇÕES

Esta memória pode ser implementada utilizando uma estrutura de matriz, onde é definido um tipo **mem_rom,** sendo uma matriz de 47 linhas e 16 colunas. As 16 colunas representam o tamanho da microinstrução; neste caso, 16 bits e as 47 linhas

representam o número de microinstruções. Esta memória é descrita através do comando:

type mem_rom is array (46 downto 0) of std_logic_vector (15 downto 0);

Nesta memória encontram-se as microinstruções responsáveis pela busca e decodificação de instruções. A tabela 7.4 mostra o significado de cada bit de microinstrução utilizada neste projeto.

Bit	Significado
0	Clock PC
1	Load PC
2	Clock RI
3	Selecionador do multiplexador de acesso aos registradores da unidade lógica e aritmética
4	Clock RegA
5	Clock RegB
6	Selecionador de operação de ULA
7	
8	
9	
10	Clock REND
11	Load REND
12	Selecionador do multiplexador de acesso ao endereço da Memória Principal
13	Sinal de RW
14	Novo ciclo de busca-decodifica-executa
15	Carrega nova instrução

Tabela 7.4 – Definição de cada bit de uma microinstrução

Nota-se que no código VDHL da unidade de controle, apenas os bits de 0 a 13 são atribuídos à saída da UC, como visto no comando **s1 <= vd(i)(13 downto 0)**. Os bits 14 e 15 são bits de controle interno da unidade de controle, onde o bit 14 é responsável por iniciar um novo ciclo de busca de instrução. Isto é possível realizando um *jump* (salto) para a posição onde está localizada a microinstrução de busca; neste caso no endereço 0 (zero) da memória de microinstrução.

```
if (vd(i)(14) = '1')
   then i <= 0;
end if;
```

O bit 15 é responsável por carregar uma nova instrução na UC para ser executada. Toda instrução que atinge a UC passou por um processo de decodificação efetuada pelo módulo decodificador.

```
if (vd(i)(15) = '1')
    then i <= conv_integer(e1);    -- conversão std_logic_vector para inteiro
end if;
```

Note a necessidade de uma conversão de tipos, de *std_logic_vector* para inteiro, pois o acesso aos endereços de memória de microinstruções é feito através de um índice inteiro "**i**".

7.4.7 – MÓDULO DECODIFICADOR DE INSTRUÇÕES

O decodificador é responsável por receber uma determinada instrução externa e retornar para o módulo UC a posição exata da microinstrução correspondente à instrução decodificada. Este decodificador não depende de clock, isto é, basta uma nova instrução chegar para ser decodificada. O decodificador também utiliza a saída do bloco comparador para auxiliar na decodificação de algumas instruções. A figura 7.9 representa o decodificador de instruções com suas respectivas portas de entrada *A* e *B* e saída *S1*.

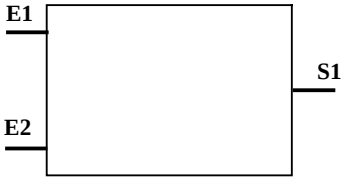


Figura 7.9 – Representação do decodificador de instruções

A seguir tem-se a descrição VHDL do decodificador de instruções.

Decodificador de instruções da UCP

```
library ieee;
use ieee.std_logic_1164.all;

entity decode is
    port (e1 : in  std_logic_vector(7 downto 0);
          e2 : in  std_logic_vector(1 downto 0);
          s1   : out std_logic_vector(5 downto 0));
```

```
end  decode;
architecture decode_arch of decode is

   constant mov    : std_logic_vector(3 downto 0) := "0000";
   constant add2   : std_logic_vector(3 downto 0) := "0001";
   constant sub2   : std_logic_vector(3 downto 0) := "0010";
   constant and2   : std_logic_vector(3 downto 0) := "0011";
   constant or2    : std_logic_vector(3 downto 0) := "0100";
   constant xor2   : std_logic_vector(3 downto 0) := "0101";
   constant nand2 : std_logic_vector(3 downto 0) := "0110";
   constant nor2   : std_logic_vector(3 downto 0) := "0111";
   constant xnor2 : std_logic_vector(3 downto 0) := "1000";
   constant jmp     : std_logic_vector(3 downto 0) := "1001";
   constant inc     : std_logic_vector(3 downto 0) := "1010";
   constant dec     : std_logic_vector(3 downto 0) := "1011";
   constant cmp     : std_logic_vector(3 downto 0) := "1100";
   constant jg      : std_logic_vector(3 downto 0) := "1101";
   constant jl      : std_logic_vector(3 downto 0) := "1110";
   constant jz      : std_logic_vector(3 downto 0) := "1111";

   -- registradores origem e destino da ucp --
   constant rga   : std_logic_vector(1 downto 0)   := "00"; -- registrador a
   constant rgb   : std_logic_vector(1 downto 0)   := "01"; -- registrador b
   constant rge   : std_logic_vector(1 downto 0)   := "10"; -- registrador rend
   constant nda   : std_logic_vector(1 downto 0)   := "11"; -- nenhum registrador
begin
    process (e1,e2)
    variable regdest,regorig: std_logic_vector(1 downto 0);
    variable code : std_logic_vector(3 downto 0);
    begin
        regorig := e1(1 downto 0); -- registrador origem
        regdest := e1(3 downto 2); -- registrador destino
        code    := e1(7 downto 4); -- opcode

        case code is
        when mov => case regdest is
                when rga =>  case regorig is
                        when nda => s1 <= "000101"; --mov a,dado
                        when rge => s1 <= "001001"; --mov a,[end]
                        when others => s1 <= "zzzzzz";
                        end case;
                when rgb =>  case regorig is
                        when nda => s1 <= "000111"; --mov b,dado
                        when rge => s1 <= "001101"; --mov b,[end]
                        when others => s1 <= "zzzzzz";
                        end case;
                when rge => case regorig is
                        when rga => s1 <= "010001"; --mov [end],a
                        when rgb => s1 <= "010101"; --mov [end],b
                        when others => s1 <= "zzzzzz";
                        end case;
                when others => s1 <= "zzzzzz";
                end case;
        when add2  => s1 <= "011001"; -- add  a,b
        when sub2  => s1 <= "011011"; -- sub  a,b
        when and2  => s1 <= "011101"; -- and  a,b
        when or2   => s1 <= "011111"; -- or   a,b
        when xor2  => s1 <= "100001"; -- xor  a,b
```

```
        when nand2 => s1 <= "100011"; -- nand a,b
        when nor2  => s1 <= "100101"; -- nor  a,b
        when xnor2 => s1 <= "100111"; -- xnor a,b
        when jmp   => s1 <= "000011"; -- jmp  rot
        when inc   => s1 <= "101001"; -- inc
        when dec   => s1 <= "101011"; -- dec
        when cmp   => s1 <= "101101"; -- cmp a,b
        when jg    => if e2="00" then
                         s1 <= "000011"; -- jg  rot
                      else s1 <= "101110";
                      end if;
         when jl   => if e2="01" then
                         s1 <= "000011"; -- jl  rot
                      else s1 <= "101110";
                      end if;
         when jz   => if e2="10" then
                         s1 <= "000011"; -- jz  rot
                      else s1 <= "101110";
                      end if;
        when others => s1 <= "zzzzzz";
      end case;
    end process;
end decode_arch;
```

A descrição VHDL do decodificador de instruções é realizada através de um seqüência de comandos CASE, que analisa o *opcode* e outras informações contidas na instrução.

7.4.8 – UNIDADE CENTRAL DE PROCESSAMENTO MICROPROGRAMADA

Após definir todos os módulos da UCP, basta definir as interconexões entre eles. Isto é possível através de sinais (*signal*) e portas de comunicação externa, como por exemplo o clock geral do sistema e interfaces com a memória principal. Na figura 7.2 tem-se a visão geral da arquitetura com suas portas de entrada *clk* e *din* e saída *dout*, *rw* e *adr*. A seguir, a descrição VHDL da unidade central de processamento de 8 Bits.

Código VHDL que integra todos os módulos da UCP

```
library ieee;
use ieee.std_logic_1164.all;
entity cpu is
   port (clk   : in std_logic;
         din   : in std_logic_vector(7 downto 0);
         dout  : out std_logic_vector(7 downto 0);
         rw    : out std_logic;
         adr   : out std_logic_vector(7 downto 0));
end cpu;
```

```
architecture cpu_arch of cpu is
-- declaracoes de componentes utilizados na ucp
component reg
port (e1 : in  std_logic_vector(7 downto 0);
            clk : in  std_logic;
            s1 : out std_logic_vector(7 downto 0));
end component;

component mux2
     port (e1 : in  std_logic_vector(7 downto 0);
          e2 : in   std_logic_vector(7 downto 0);
          sel : in   std_logic;
          s1 : out  std_logic_vector(7 downto 0));
end component;

component decode
     port (e1 : in  std_logic_vector(7 downto 0);
           e2 : in  std_logic_vector(1 downto 0);
           s1   : out std_logic_vector(5 downto 0));
end component;

component counter
     port (e1 : in  std_logic_vector(7 downto 0);
          clk   : in std_logic;
          load    : in  std_logic;
          s1   : out std_logic_vector(7 downto 0));
end component;

component ula
     port (e1    :  in  std_logic_vector(7 downto 0);
           e2    :  in  std_logic_vector(7 downto 0);
           op      :  in  std_logic_vector(3 downto 0);
           s1 : out std_logic_vector(7 downto 0));
end component;

component control
     port (clk       : in  std_logic;
           e1   : in  std_logic_vector(5 downto 0);
           s1      : out std_logic_vector(13 downto 0));
end component;

component comp
 port (e1: in std_logic_vector(7 downto 0);
           e2: in std_logic_vector(7 downto 0);
           s1: out std_logic_vector(1 downto 0));
end component;

signal a: std_logic_vector(13 downto 0);
signal riout,ulaout,ulareg,regaout,regbout,pcout,rendout: std_logic_vector(7 downto 0);
signal instcod: std_logic_vector(5 downto 0);
signal compout: std_logic_vector(1 downto 0);
begin
       ri      : reg port map (din,a(2),riout);
       ulamux  : mux2 port map (riout,ulaout,a(3),ulareg);
       rega      : reg port map (ulareg,a(4),regaout);
       regb      : reg port map (ulareg,a(5),regbout);
       ula1      : ula port map (regaout,regbout,a(9 downto 6),ulaout);
       cmp       : comp port map (regaout,regbout,compout);
```

```
    dec      : decode port map (riout,compout,instcod);
    pc       : counter port map (riout,a(0),a(1),pcout);
    rend     : counter port map (riout,a(10),a(11),rendout);
    memmux : mux2 port map (pcout,rendout,a(12),adr);
    uc       : control port map (clk,instcod,a);
    rw <= a(13);
  dout <= ulaout;
end cpu_arch;
```

7.5 – IMPLEMENTAÇÃO DA UCP COM FSM

A maioria das UCP descritas em VHDL utiliza-se a metodologia de desenvolvimento com máquina de estados finita. Além de ser uma metodologia que pode gerar um circuito mais otimizado, é facil de ser interpretada por outros projetistas, pois os estados são bem definidos, únicos e interligados, facilitando a visão do fluxo de dados no sistema.

A máquina de estados finita (FSM) é sincronizada pelo clock geral do sistema, isto é, a cada pulso do relógio, o estado atual é atualizado, determinando a dinâmica do sistema. Neste caso, a implementação da máquina de estado é limitada a apenas um processo. Neste, descreve-se o sincronismo com o relógio, o fluxo de dados e a arquitetura do sistema.

A implementação com (FSM) possibilita a descrição de todos os módulos da UCP juntamente com a descrição do fluxo de dados e sincronismo do sistema, eliminando sinais ou canais de fluxo de dados que interligam os diversos módulo como na UCP microprogramada.

CARACTERÍSTICAS DA DESCRIÇÃO EM VHDL.

O código VHDL da UCP de 8 bits implementada com uma máquina de estados finita pode ser dividido em três partes principais:

- **Primeira parte:** declaração de portas de entradas e saídas da UCP, sinais e tipos do sistema. As entradas e saídas da UCP com máquina de estados finita são basicamente as mesmas da implementação microprogramada, com exceção da entrada de reset, pois é importante que toda FSM tenha um reset. Na declaração de sinais e tipos encontram-se todos os registradores e sinais de controle da UCP, além da definição dos possíveis estados da UCP.

- **Segunda parte:** Decodificação da instrução. A decodificação das instruções é uma parte que envolve um lógica que afeta de forma intensa o desempenho do circuito. Assim, determinar como esta parte será descrita é importante, pois influencia diretamente no desempenho do circuito, por isso, deve ser bem elaborada.

  Uma forma bastante otimizada de desenvolvimento é descrever a decodificação de instruções de maneira concorrente, otimizando o tempo de execução do circuito consideravelmente. Esta metodologia possui um impacto no espaço físico do circuito que aumenta. Uma outra forma é a decodificação seqüencial. Esta produz um circuito fisicamente menor, mas o tempo de execução do mesmo é bem maior. Comparando as duas metodologias, neste caso, é mais vantajoso descrever a decodificação das instruções de maneira concorrente, pois a relação espaço/tempo é mais vantajosa.
- **Terceira parte:** Busca e execução de instruções. Nesta parte, é feito o tratamento do sinal de reset, a busca e execução de instruções através de uma máquina de estados finita.

A seguir, o código VHDL da UCP de 8 bits implementada com um máquina de estados finita.

UCP de 8 bits (FSM)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- primeira parte - declaracao das portas de i/o,  sinais e tipos

entity cpu is
  port (clk: in std_logic; -- clock geral
        rst: in std_logic; -- sinal de reset
        din: in std_logic_vector(7 downto 0); -- entrada de dados vindos da ram
        rw: out std_logic; -- sinal de escrita/leitura na ram
        addr: out std_logic_vector(4 downto 0); -- enderecamento da ram
        dout: out std_logic_vector(7 downto 0)); -- saida de dados para ram
end cpu;
architecture cpu_arch of cpu is
-- estados da cpu
type estado is (reset1, reset2, busca, executa, movad, movbd, movae,
                movbe, movex, sync_grav, jmp, resultado);

signal estado_atual: estado;

signal sdout,       -- alimenta a entrada da ram
       reg_inst1,  -- registrador de instrucao 1
```

```
        reg_inst2,  -- registrador de instrucao 2
        rega,       -- alimentacao da ula (registrador a)
        regb,       -- alimentacao da ula (registrador b)
        reg_ula:    -- armazena resultados de operacoes da ula
        std_logic_vector (7 downto 0);
signal pc: std_logic_vector (4 downto 0); -- contador de programa
signal comp: std_logic_vector (1 downto 0); -- armazena resultado do
                                            -- comparador

-- sinais que indicam qual instrucao deve ser executada
signal inst_movad, inst_movbd, inst_movea, inst_moveb, inst_movae,
        inst_movbe: std_logic;
signal inst_add, inst_sub, inst_and, inst_or, inst_xor, inst_nor,
        inst_nand, inst_xnor: std_logic;
signal inst_inc, inst_dec, inst_cmp, inst_jg, inst_jl, inst_jz, inst_jmp
        :std_logic;

begin

   -- segunda parte - decodificao da instrucao (trecho concorrente)

   inst_movad <= '1' when reg_inst1(7 downto 0) ="00000011" else '0';
   inst_movbd <= '1' when reg_inst1(7 downto 0) ="00000111" else '0';
   inst_movea <= '1' when reg_inst1(7 downto 0) ="00001000" else '0';
   inst_moveb <= '1' when reg_inst1(7 downto 0) ="00001001" else '0';
   inst_movae <= '1' when reg_inst1(7 downto 0) ="00000010" else '0';
   inst_movbe <= '1' when reg_inst1(7 downto 0) ="00000110" else '0';
   inst_add    <= '1' when reg_inst1(7 downto 4) ="0001" else '0';
   inst_sub    <= '1' when reg_inst1(7 downto 4) ="0010" else '0';
   inst_and    <= '1' when reg_inst1(7 downto 4) ="0011" else '0';
   inst_or     <= '1' when reg_inst1(7 downto 4) ="0100" else '0';
   inst_xor    <= '1' when reg_inst1(7 downto 4) ="0101" else '0';
   inst_nor    <= '1' when reg_inst1(7 downto 4) ="0110" else '0';
   inst_nand   <= '1' when reg_inst1(7 downto 4) ="0111" else '0';
   inst_xnor   <= '1' when reg_inst1(7 downto 4) ="1000" else '0';
   inst_jmp    <= '1' when reg_inst1(7 downto 4) ="1001" else '0';
   inst_inc    <= '1' when reg_inst1(7 downto 4) ="1010" else '0';
   inst_dec    <= '1' when reg_inst1(7 downto 4) ="1011" else '0';
   inst_cmp    <= '1' when reg_inst1(7 downto 4) ="1100" else '0';
   inst_jg     <= '1' when reg_inst1(7 downto 4) ="1101" else '0';
   inst_jl     <= '1' when reg_inst1(7 downto 4) ="1110" else '0';
   inst_jz     <= '1' when reg_inst1(7 downto 4) ="1111" else '0';

   addr <= pc;
   dout <= sdout;

   -- terceira parte -  busca e execucao ( fsm)

   process (clk,rst)
   begin
     if rst='1' then
       estado_atual <= reset1;
     elsif clk'event and clk='1' then
       rw <= '0';
       case estado_atual is

          -- inicializa registradores
          when reset1 =>
```

```
    reg_inst1 <= "00000000";
    reg_inst2 <= "00000000";
    rega <= "00000000";
    regb <= "00000000";
    reg_ula <= "00000000";
    sdout <= "00000000";
    estado_atual <= reset2;

-- inicializa pc (contador de programa)
when reset2 =>
    pc <= "00000";
    estado_atual <= busca;

-- busca instrucao apontado pelo pc
when busca =>
    reg_inst1 <= din;
    pc <= pc + 1;
    estado_atual <= executa;

-- analisa sinais resultantes da decodificacao do opcode e executa
-- operacao ou parte dela.
when executa =>
    if inst_movad='1' then
      reg_inst2 <= din;
      estado_atual <= movad;
    elsif inst_movbd='1' then
      reg_inst2 <= din;
      estado_atual <= movbd;
    elsif inst_movae='1' then
      reg_inst2(4 downto 0) <= pc + 1;
      pc <= din(4 downto 0);
      estado_atual <= movae;
    elsif inst_movbe='1' then
      reg_inst2(4 downto 0) <= pc + 1;
      pc <= din(4 downto 0);
      estado_atual <= movbe;
    elsif inst_movea='1' then
      sdout <= rega;
      reg_inst2(4 downto 0) <= pc + 1;
      pc <= din(4 downto 0);
      rw <= '1';
      estado_atual <= sync_grav;
    elsif inst_moveb='1' then
      sdout <= regb;
      reg_inst2(4 downto 0) <= pc + 1;
      pc <= din(4 downto 0);
      rw <= '1';
      estado_atual <= sync_grav;
    elsif inst_add='1' then
      reg_ula <= rega + regb;
      estado_atual <= resultado;
    elsif inst_sub='1' then
      reg_ula <= rega - regb;
      estado_atual <= resultado;
    elsif inst_and='1' then
      reg_ula <= rega and regb;
      estado_atual <= resultado;
    elsif inst_or='1' then
```

```
      reg_ula <= rega or regb;
      estado_atual <= resultado;
    elsif inst_xor='1' then
      reg_ula <= rega xor regb;
      estado_atual <= resultado;
    elsif inst_nor='1' then
      reg_ula <= rega nor regb;
      estado_atual <= resultado;
    elsif inst_nand='1' then
      reg_ula <= rega nand regb;
      estado_atual <= resultado;
    elsif inst_xnor='1' then
      reg_ula <= rega xnor regb;
      estado_atual <= resultado;
    elsif inst_inc='1' then
      reg_ula <= rega + "00000001";
      estado_atual <= resultado;
    elsif inst_dec='1' then
      reg_ula <= rega - "00000001";
      estado_atual <= resultado;
    elsif inst_jmp='1' then
      reg_inst2 <= din;
      estado_atual <= jmp;
    elsif inst_cmp='1' then
      if rega>regb then
        comp<="00";
      elsif rega<regb then
        comp<="01";
      else
        comp<="10";
      end if;
      estado_atual <= busca;
    elsif inst_jg='1' then
      if comp="00" then
        reg_inst2 <= din;
        estado_atual <= jmp;
      else
        pc <= pc + "00001";
        estado_atual <= busca;
      end if;
    elsif inst_jl='1' then
      if comp="01" then
        reg_inst2 <= din;
        estado_atual <= jmp;
      else
        pc <= pc + "00001";
        estado_atual <= busca;
      end if;
    elsif inst_jz='1' then
      if comp="10" then
        reg_inst2 <= din;
        estado_atual <= jmp;
      else
        pc <= pc + "00001";
        estado_atual <= busca;
      end if;
    end if;
  -- parte final da execucao da instrucao mov a,dado
```

```
            when movad =>
                rega <= reg_inst2;
                pc <= pc + "00001";
                estado_atual <= busca;

            -- parte final da execucao da instrucao mov b,dado
            when movbd =>
                regb <= reg_inst2;
                pc <= pc + "00001";
                estado_atual <= busca;

            -- parte final da execucao da instrucao mov a,[end]
            when movae =>
                rega <= din;
                pc <= reg_inst2(4 downto 0);
                estado_atual <= busca;

            -- parte final da execucao da instrucao mov b,[end]
            when movbe =>
                regb <= din;
                pc <= reg_inst2(4 downto 0);
                estado_atual <= busca;

            -- sincronizacao da gravacao de dados na ram
            when sync_grav =>
                estado_atual <= movex;
            -- parte final da execucao da instrucao mov [end],a e mov [end],b
            when movex =>
                pc <= reg_inst2(4 downto 0);
                estado_atual <= busca;
            -- parte final da execucao da instrucao jmp [end]
            when jmp =>
                pc <= reg_inst2(4 downto 0);
                estado_atual <= busca;

            -- parte final da execucao da instrucao logicas e aritmetica da ula
            when resultado =>
                rega <= reg_ula;
                estado_atual <= busca;
             when others=> estado_atual <= busca;
        end case;
      end if;
    end process;
end cpu_arch;
```

7.6 – ANÁLISE DO DESEMPENHO DAS METODOLOGIA DE DESENVLOVIMENTO UCP

A seguir, apresenta-se as estatísticas da UCP implementada, microprogramada e com máquinas de estados finita (FSM). Para gerar as estatísticas de desempenho a memória RAM responsável pelo armazenamento das instruções foi excluída, pois a intenção é avaliar o desempenho da UCP independente da memória utilizada.

Estatísticas de recursos utilizados da FPGA				
UCP	**FF**	**LUTs 4**	**LUTs 3**	**CLBs**
Microprogramada	51	237	39	126
FSM	59	232	25	130
Temporização da UCP				
UCP	**TL**	**TR**	**AT**	
Microprogramada	11.079ns	25.950ns	37.029ns	
FSM	7.289ns	19.526ns	26.815ns	

Tabela 7.5 - Estatísticas de desempenho em FPGA das UCPs de 8 bits.

É interessante notar que, apesar da UCP implementada com máquina de estados finita utilizar um pouco mais de recursos do FPGA, seu atraso total é consideravelmente menor em relação à implementação microprogramada.

Em circuitos de baixa complexidade, a diferença no atraso total normalmente não ultrapassa 2 ou 3 ns, parecendo não ser muito significativa, mas quando se tem um circuito um pouco mais complexo como uma UCP, essa diferença é notada com mais facilidade.

7.7 – UCP DE 16 BITS

A UCP de 16 bits proposta é capaz de executar operações lógicas e aritméticas e comparações de 16 bits, portanto sua ULA e o módulo comparador são mais robustos. Os registradores RI, REGA e REGB são capazes de armazenar 16 bits de informação. Módulos como o multiplexador também são modificados para selecionarem barramentos de 16 bits.

A UCP de 16 bits obedece à mesma arquitetura selecionada para o UCP de 8 bits. Foi adotada a metodologia de implementação com máquina de estados finita (FSM), já que esta obteve um melhor desempenho em UCP de 8 bits, como já citado anteriormente. A tabela 7.7 mostra a temporização e a ocupação espacial da UCP de 16 bits no FPGA.

Nota-se que o tempo e o espaço da UCP de 16 bits implementado em FPGA aumentaram em relação à estatística de desempenho da UCP de 8 bits. Isto parece lógico, pois toda a arquitetura ficou mais robusta. Agora a pergunta chave é: Qual a vantagem de implementar a arquitetura de 16 bits? Mesmo tendo um tempo total de propagação e ocupação maior.

A UCP de 16 bits tem um boa vantagem quando é analisado o desempenho funcional. Para entender melhor, observa-se o exemplo a seguir.

Exemplo: Uma operação comum em UCPs é a adição de 32 bits. A tabela 7.6 demostra os passos necessários para execução desta operação nas UCPs 8 e 16 bits.

UCP de 8 bits	**Ciclos**	**Tempo Gasto**
Busca 8 bits	2	56,706ns
Decodifica Executa	2	56,706ns
Armazena Parcial 1	2	56,706ns
Busca 8 bits	2	56,706ns
Decodifica Executa	2	56,706ns
Armazena Parcial 2	2	56,706ns
Busca 8 bits	2	56,706ns
Decodifica Executa	2	56,706ns
Armazena Parcial 3	2	56,706ns
Busca 8 bits	2	56,706ns
Decodifica Executa	2	56,706ns
Armazena Parcial 4	2	56,706ns
Total	**24**	**680,472ns**

UCP de 16 bits	**Ciclos**	**Tempo Gasto**
Busca 16 bits	3	93,135ns
Decodifica Executa	2	62,09ns
Armazena Parcial 1	3	93,135ns
Busca 16 bits	3	93,135ns
Decodifica Executa	2	62,09ns
Armazena Parcial 2	3	93,135ns
Total de ciclos	**16**	**496,72ns**

Tabela 7.6 – Passo a passo da execução de uma adição de 32 bits

Com os resultados obtidos na tabela 7.6, é fácil notar, onde a UCP de 16 bits faz a diferença. Enquanto uma instrução de adição de 32 bits é executada em 496,720ns pela UCP de 16 bits, a de 8 bits levou 680,472ns para executá-la. É uma diferença significativa e até perceptível quando, por exemplo, milhares de adições forem executadas por ambas as UCPs.

Outra coisa interessante é que apesar da UCP de 8 bits ser mais rápida, quando considerado o tempo de cada ciclo de instrução, mesmo assim não supera a eficiência do processamento de 16 bits. A tabela 7.7 apresenta as estatísticas de desempenho

temporal e espacial da UCP de 16 bits.

Recursos utilizados				
	FF	**LUTs 4**	**LUTs 3**	**CLBs**
UCP 16 bits	80	350	45	183
Temporização				
	Tempo de lógica	**Tempo de roteamento**		**Atraso total**
UCP 16 bits	12,015ns	19,030ns		31,045ns

Tabela 7.7 - Estatísticas de desempenho em FPGA da UCP de 16 bits.

7.8 – UCP HC05

A unidade central de processamento HC05 implementada realiza as mesmas instruções e tem os mesmos opcodes que a UCP contida no microcontrolador MC68**HC05**J5A da Motorola. A capacidade de mapeamento de memória é de 4K-bytes, onde apenas 64 bytes são para pilha, variando do endereço 00C0h até 00FFh

A UCP HC05 possui uma arquitetura completa, realizando instruções como: instruções lógicas, aritméticas, de movimentação de dados e *jumps* condicionais e incondicionais. Instruções dedicadas aos dispositivos de entrada e saída do microcontrolador não foram descritas nessa versão. O comportamento de todos os registradores está descrito a seguir.

7.8.1 – REGISTRADORES

O processador contém cinco registradores internos e outros que estão localizados na memória. Esses cinco registradores internos são apresentados na figura 7.10, descrevendo os seus nomes e quantidade de bits.

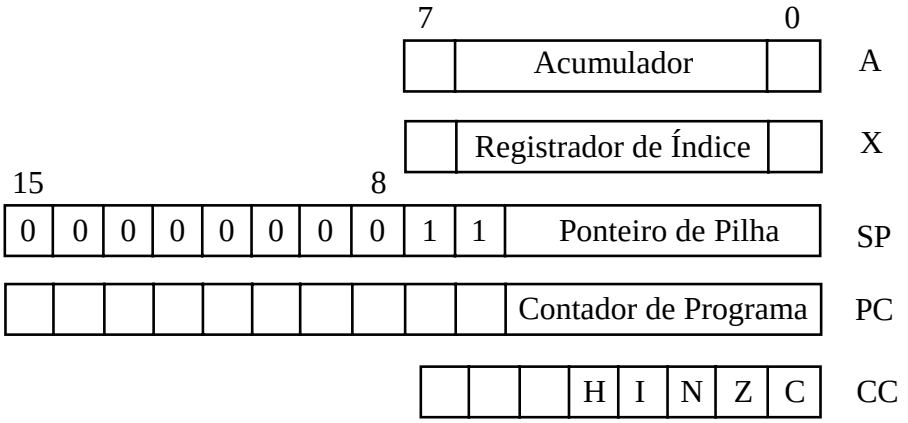


Figura 7.10 – Registradores internos do HC05

ACUMULADOR (A)

O acumulador é um registrador de propósito geral de 8 bits, usado para armazenar resultados de diversas operações. Este registrador não é afetado pelo *reset* do sistema.

REGISTRADOR DE ÍNDICE (X)

O registrador de índice é de 8 bits e pode ser utilizado para duas funções principais:

- Endereçamento indexado
- Armazenamento temporário

São três os modos de endereçamento indexado:

- Modo de endereçamento indexado sem parâmetro, onde o registrador de índice contém, no meio byte menos significativo, o endereço, e no meio byte mais significativo assume 0000_B, gerando o endereço final.
- Modo de endereçamento indexado com 8 bits de parâmetro: o endereço final é gerado a partir da soma do conteúdo o registrador de índice com o conteúdo dos 8 bits imediatos.

- Modo de endereçamento indexado com 16 bits de parâmetro: o endereço final é gerado a partir da soma do conteúdo o registrador de índice com o conteúdo dos 16 bits imediatos.

O registrador de índice também pode auxiliar o registrador acumulador para armazenamento temporário. Esse registrador não é afetado pelo *reset* do sistema.

PONTEIRO DE PILHA (SP)

O ponteiro de pilha é de 16 bits. Esse registrador contém o próximo endereço de pilha livre. Durante o *reset* do sistema ou quando é executada a instrução RSP, o ponteiro de pilha é inicializado com o valor 00FFh. Quando um dado é inserido na pilha, o ponteiro de pilha é decrementado e quando um dado é retirado, o ponteiro é incrementado.

Os dez bits mais significativos sempre contêm o valor binário 0000000011b. Isso é feito para que o intervalo da pilha na memória não ultrapasse de 00FF à 00C0. Se exceder aos 64 bytes de pilha, os dados serão sobrescritos.

CONTADOR DE PROGRAMA (PC)

O registrador contador de programa é de 16 bits, sendo capaz de endereçar 64K-bytes, onde os endereços gerados acima do limite da memória são ignorados. Esse registrador aponta a próxima instrução ou operando a ser buscado.

REGISTRADOR DE CÓDIGO DE CONDIÇÃO (CC)

Apesar de ser um registrador de 8 bits apenas 5 são utilizados para indicar características do resultado de instruções que acabaram de ser executadas. Cada bit tem sua função. Com exceção do bit de interrupção (I), todos os outros foram implementados. A seguir, a descrição de cada bit do registrador CC.

- Half Carry (H) – Este bit é setado quando ocorre um *overflow* do meio byte mais significativo da operação de adição (ADD/ADC).

- Negativo (N) – Este bit é setado quando o bit 7 do resultado da última operação lógica, aritmética ou de manipulação de dados assumir "1".
- Zero (Z) - Este bit é setado quando o resultado da última operação lógica, aritmética ou de manipulação de dados resultar em zero.
- Carry/Borrow (C) – Este bit é setado quando ocorre um *overflow* ou *underflow* do resultado de algumas operações.

7.8.2 - MODOS DE ENDEREÇAMENTO.

A UCP HC05 utiliza oito modos de endereçamento facilitando e otimizando o acesso aos dados desejados. O modo de endereçamento define a maneira como o processador buscará os dados requeridos para execução. A seguir, os modos de endereçamento.

- Inerente
- Indexado sem parâmetro
- Indexado com 16 bits de parâmetro
- Imediato
- Direto
- Indexado com 8 bits de parâmetro
- Relativo
- Estendido

INERENTE

As instruções que utilizam endereçamento inerente são aquelas que não possuem operando como a instrução SEC, que seta o bit de *carry* e INCA, que incrementa o acumulador. Esse tipo de endereçamento não requer endereço de memória ou dados adicionais. São instruções de um byte.

IMEDIATO

Imediatamente após o *opcode* da instrução, tem-se o dado que deve ser utilizado na operação com o registrador acumulador ou de índice. Esse modo de endereçamento não requer endereço de memória e são instruções de dois bytes. Onde o primeiro byte é o *opcode* da instrução, e o segundo, o operando.

DIRETO

No endereçamento direto, o dado requisitado está localizado nas primeiras 256 posições da memória. Para gerar o endereço correspondente, concatena-se 00h com o próximo byte imediato, buscando o dado desejado. Esse modo de endereçamento

requer endereço de memória e são instruções de dois bytes. Onde o primeiro byte é o *opcode* da instrução e o segundo, o valor do endereço em que está localizado o dado.

ESTENDIDO

O modo de endereçamento estendido utiliza três bytes, onde o primeiro é o *opcode* e o segundo e terceiro representam o endereço completo de onde está localizado o dado desejado. Esse tipo de endereçamento possibilita a localização do dado desejado em qualquer posição de memória existente.

INDEXADO SEM PARÂMETRO

No modo de endereçamento indexado sem parâmetro, o endereço de localização do dado desejado é gerado a partir da concatenação de 00h com o conteúdo do registrador de índice (X). Sendo assim, o endereço gerado está localizado no intervalo de 0000h a 00FFh.

INDEXADO COM 8 BITS DE PARÂMETRO

No modo de endereçamento indexado com 8 bits de parâmetro, o endereço de localização do dado desejado é gerado a partir da soma do conteúdo do registrador de índice (X) com o byte imediato ao *opcode* da instrução. Sendo assim, o endereço gerado está localizado no intervalo de 0000h à 01FEh.

INDEXADO COM 16 BITS DE PARÂMETRO

No modo de endereçamento indexado com 16 bits de parâmetro, o endereço de localização do dado desejado é gerado a partir da soma do conteúdo do registrador de índice (X) com os bytes imediatos ao *opcode* da instrução. Sendo assim, o endereço gerado pode acessar qualquer posição de memória disponível.

RELATIVO

O modo de endereçamento relativo é utilizado por instruções de desvio condicional (*branch*). Se a condição de desvio for verdadeira, o valor do endereço relativo é adicionado ou subtraído do contador de programa (PC), desviando o fluxo de execução do programa. O bit mais significativo define se o desvio será positivo ou

negativo. Se o bit 7 for 1 (um), então o desvio é positivo, isto é, o endereço relativo será adicionado ao PC, senão o desvio é negativo, subtraindo o valor do endereço relativo do PC. Assim o intervalo de acesso à memória é de –128 a +127 em relação ao PC.

7.8.3 – TIPOS DE INSTRUÇÃO

As instruções podem ser divididas em cinco categorias:

- Instruções de registrador/memória
- Desvio/Desvio condicional
- Instruções de controle
- Instruções de leitura, modificação e escrita
- Instruções de manipulação de Bit

INSTRUÇÕES DE REGISTRADOR/MEMÓRIA

São instruções que utilizam dois operandos. Um operando está contido no registrador acumulador ou de índice e o outro está localizado na memória. A tabela 7.8 apresenta o mnemônico e a descrição das instruções de registrador/memória.

Mnemônico	Descrição
ADC	Operação de adição do byte da memória, carry bit e acumulador
ADD	Operação de adição do byte da memória e acumulador
AND	Operação de AND do byte da memória e acumulador
BIT	Testa o acumulador com byte da memória
CMP	Compara acumulador com byte de memória
CPX	Compara registrador de índice com byte de memória
EOR	Operação de XOR do byte da memória e acumulador
LDA	Carrega o acumulador com o byte de memória
LDX	Carrega o registrador de índice com o byte de memória
MUL	Multiplicação
ORA	Operação de OR do byte da memória e acumulador
SBC	Operação de subtração do byte da memória, carry bit e acumulador
STA	Armazena acumulador na memória
STX	Armazena registrador de índice na memória
SUB	Operação de subtração do byte da memória e acumulador

Tabela 7.8 - Instruções de registrador/memória

INSTRUÇÕES DE LEITURA, MODIFICAÇÃO E ESCRITA.

Estas instruções fazem a leitura da memória ou de um registrador, modifica a informação, e escreve novamente no local de origem. A exceção é a instrução TST, pois não realiza a escrita. Na tabela 7.9, tem-se o mnemônico e a descrição das instruções de leitura, modificação e escrita.

Mnemônico	Descrição
ASL	Deslocamento aritmético à esquerda
ASR	Deslocamento aritmético à direita
BCLR	Zera um bit na memória
BSET	Seta um bit na memória
CLR	Zera
COM	Complemento de um
DEC	Decrementa
INC	Incrementa
LSL	Deslocamento lógico à esquerda
LSR	Deslocamento lógico à direita
NEG	Complemento de dois
ROL	Rotação à esquerda com carry
ROR	Rotação à direita com carry
TST	Teste para negativo ou zero

Tabela 7.9 - Instruções de leitura, modificação e escrita

DESVIO INCONDICIONAL /DESVIO CONDICIONAL

O desvio incondicional é executado pelas instruções JMP e JSR. Neste tipo de desvio, a seqüência de execução do programa é alterada sem depender de uma condição previamente estabelecida.

Já o desvio condicional, utilizado pelas demais instruções de desvio (*branch*), só altera o curso normal de execução de um programa se uma condição previamente estabelecida seja satisfeita, senão o curso de execução continua o mesmo. O desvio condicional utiliza-se do modo de endereçamento relativo, onde o bit mais significativo do endereço identifica se o desvio é positivo ou negativo. A tabela 7.10 mostra o mnemônico e a descrição das instruções desvio condicional e incondicional.

Mnemônico	Descrição
BCC	Desvia se CC Carry é zero
BCS	Desvia se CC Carry é um
BEQ	Desvia se igual
BHCC	Desvia se CC Half-Carry é zero
BHCS	Desvia se CC Half-Carry é um
BHS	Desvia se maior igual
BLO	Desvia se menor
BLS	Desvia se menor igual
BMI	Desvia se negativo
BNE	Desvia se não for igual
BPL	Desvia se CC negativo é zero
BRA	Desvia sempre
BRCLR	Desvia se bit é zero
BRN	Nunca desvia

BRSET	Desvia se bit é um
BSR	Desvia para subrotina
JMP	Salto incondicional
JSR	Salto para subrotina

Tabela 7.10 – Desvio incondicional/Desvio condicional

INSTRUÇÕES DE MANIPULAÇÃO DE BIT

Esse tipo de instrução manipula apenas um bit que deve estar localizado nos primeiros 256 bytes de memória. A manipulação é do tipo leitura ou escrita de valor de um determinado bit. O modo de endereçamento utilizado é direto. Na tabela 7.11 tem-se o mnemônico e a descrição das instruções de manipulação de bit.

Mnemônico	Descrição
BCLR	Zera bit
BRCLR	Desvia se bit é zero
BRSET	Desvia se bit é um
BSET	Seta bit

Tabela 7.11 - Instruções de manipulação de bit

INSTRUÇÕES DE CONTROLE

São instruções que auxiliam no fluxo de informações durante a execução do programa, realizando operações de transferência de dados entre registradores e mudanças no registrador de código de condição (CC). A tabela 7.12 apresenta o mnemônico e a descrição das instruções de controle.

Mnemônico	Descrição
CLC	Zera Carry Bit
NOP	Nenhuma operação
RSP	Inicializa ponteiro de pilha
RTS	Retorna de subrotina
SEC	Seta Carry Bit
TAX	Transfere conteúdo do acumulador para registrador de índice
TXA	Transfere conteúdo do registrador de índice para acumulador

Tabela 7.12 – Instruções de controle

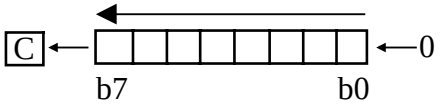
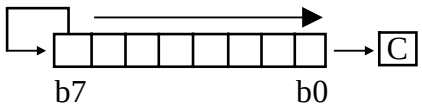
7.8.4 – DESCRIÇÃO COMPLETA DO CONJUNTO DE INSTRUÇÕES

Na tabela 7.13 tem-se a descrição completa do conjunto de instruções do processador HC05. Nessa tabela, está descrita as seguintes informações:

- **Formato** - O formato da instrução em cada modo de endereçamento, mostrando o seu mnemônico e operandos.
- **Descrição** - Mostra a operação realizada e os registradores envolvidos.
- **CC -** Nesse campo é descrito a situação dos f*lags* do registrador de código de condição para cada instrução, onde "X" indica que o *flag* pode sofrer alterações e "|" indica que o *flag* não é alterado.
- **Modo de endereçamento –** Descreve os modos de endereçamento que cada instrução pode trabalhar.
- **Opcode –** O valor em hexadecimal de cada instrução do processador HC05. Esse valores são reais.

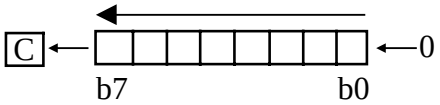
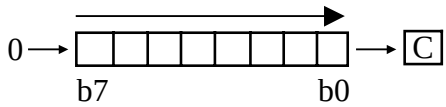
A seguir uma lista de abreviações utilizadas na tabela 7.13.

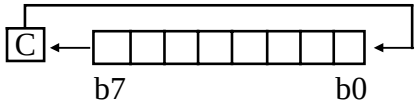
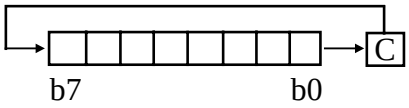
A = Acumulador
X = Registrador de índice
M = Byte de memória
C = Bit de carry (CC)
N = Bit de Negativo (CC)
H = Bit de half-carry (CC)
Z = Bit de Zero (CC)
PC = Contador de programa
PCL = Contador de Programa byte menos significativo
PCH = Contador de Programa byte mais significativo
SP = Ponteiro de pilha
n = Algum bit
op = Operando
INR = Modo de endereçamento inerente
IME = Modo de endereçamento imediato
DIR = Modo de endereçamento direto
EST = Modo de endereçamento estendido
IX2 = Modo de endereçamento indexado com 16 bits parâmetro.
IX1 = Modo de endereçamento indexado com 8 bits parâmetro.
IX = Modo de endereçamento indexado sem parâmetro.
REL = Modo de endereçamento Relativo ou endereço relativo.

Descrição do conjunto de instruções do HC05							
Formato	Descrição	CC				Modo de Endereçamento	Opcode
		H	N	Z	C		
ADC #op	A← (A) + (M) + (C)	X	X	X	X	IME	A9
ADC op						DIR	B9
ADC op						EST	C9
ADC op,X						IX2	D9
ADC op,X						IX1	E9
ADC X						IX	F9
ADD #op	A← (A) + (M)	X	X	X	X	IME	AB
ADD op						DIR	BB
ADD op						EST	CB
ADD op,X						IX2	DB
ADD op,X						IX1	EB
ADD X						IX	FB
AND #op	A← (A) AND (M)	|	X	X	|	IME	A4
AND op						DIR	B4
AND op						EST	C4
AND op,X						IX2	D4
AND op,X						IX1	E4
AND X						IX	F4
ASL op		|	X	X	X	DIR	38
ASLA						INR	48
ASLX						INR	58
ASL op,X						IX1	68
ASL X						IX	78
ASR op		|	X	X	X	DIR	37
ASRA						INR	47
ASRX						INR	57
ASR op,X						IX1	67
ASR X						IX	77
BCC	PC← (PC) + 2 + REL? C=1	|	|	|	|	REL	24

BCLR n op	M(n) ← 0	|	|	|	|	DIR(0) DIR(1) DIR(2) DIR(3) DIR(4) DIR(5) DIR(6) DIR(7)	11 13 15 17 19 1B 1D 1F
BCS rel	PC←(PC) +2 + REL ? C=1	|	|	|	|	REL	25
BEQ rel	PC←(PC) +2 + REL ? Z=1	|	|	|	|	REL	27
BHCC	PC←(PC) +2 + REL ? H=0					REL	28
BHCS	PC←(PC) +2 + REL ? H=1	|	|	|	|	REL	29
BHI	PC←(PC) +2 + REL ? C OR Z=1	|	|	|	|	REL	22
BHS	PC←(PC) +2 + REL ? C=0	|	|	|	|	REL	24
BIT	(A) AND (M)	|	X	X	|	IME DIR EST IX2 IX1 IX	A5 B5 C5 D5 E5 F5
BLO	PC←(PC) +2 + REL ? C=1	|	|	|	|	REL	25
BLS	PC←(PC) +2 + REL ? C AND Z=1	|	|	|	|	REL	23
BMI	PC←(PC) +2 + REL ? N=1	|	|	|	|	REL	2B
BNE	PC←(PC) +2 + REL ? Z=0	|	|	|	|	REL	26
BPL	PC←(PC) +2 + REL ? N=0	|	|	|	|	REL	2^A
BRA	PC←(PC) +2 + REL ? 1=1	|	|	|	|	REL	20
BRCLR n op rel	PC←(PC) +2 + REL ? M(N)=0	|	|	|	|	DIR(0) DIR(1) DIR(2) DIR(3) DIR(4) DIR(5) DIR(6) DIR(7)	01 03 05 07 09 0B 0D 0F
BRSET	PC←(PC) +2 + REL ? M(N)=1	|	|	|	|	DIR(0) DIR(1) DIR(2) DIR(3) DIR(4) DIR(5) DIR(6) DIR(7)	00 02 04 06 08 0^A 0C 0E
BRN	PC←(PC) +2 + REL ? 1=0	|	|	|	|	REL	21

BSET	M(N) ← 1	|	|	|	|	DIR(0) DIR(1) DIR(2) DIR(3) DIR(4) DIR(5) DIR(6) DIR(7)	10 12 14 16 18 1^A 1C 1E
BSR	PC←(PC) +2; PUSH (PCL); SP←SP-1; PC←(PC) +2; PUSH (PCL); SP←SP-1; PC←(PC) + REL					REL	AD
CLC	C←0	|	|	|	0	INR	98
LR op CLRA CLRX CLR op,X CLR X	M← $00 A← $00 X← $00 M← $00 M← $00	|	0	1	|	DIR INR INR IX1 IX	3F 4F 5F 6F 7F
CMP #op CMP op CMP op CMP op,X CMP op,X CMP X	(A) – (M)	|	X	X	X	IME DIR EST IX2 IX1 IX	A1 B1 C1 D1 E1 F1
COM op COM A COM X COM op,X COM X	M← $FF - (M) A← $FF - (M) X← $FF - (M) M← $FF - (M) M← $FF - (M)	|	X	X	1	DIR INR INR IX1 IX	3F 4F 5F 6F 7F
CPX #op CPX op CPX op CPX op,X CPX op,X CPX X	(X) – (M)	|	X	X	X	IME DIR EST IX2 IX1 IX	A3 B3 C3 D3 E3 F3
DEC op DEC A DEC X DEC op,X DEC X	M← (M) – 1 A← (A) – 1 X← (X) – 1 M← (M) – 1 M← (M) – 1	|	X	X	|	DIR INR INR IX1 IX	3F 4F 5F 6F 7F
EOR #op EOR op EOR op EOR op,X EOR op,X EOR X	A← (A) XOR (M)	|	X	X	|	IME DIR EST IX2 IX1 IX	A8 B8 C8 D8 E8 F8

INC op	M← (M) + 1					DIR	3C
INC A	A← (A) + 1					INR	4C
INC X	X← (X) + 1	|	X	X	|	INR	5C
INC op,X	M← (M) + 1					IX1	6C
INC X	M← (M) + 1					IX	7C
JMP op						DIR	BC
JMP op						EST	CC
JMP op,X	PC ← ENDEREÇO DE SALTO	|	|	|	|	IX2	DC
JMP op,X						IX1	EC
JMP X						IX	FC
JSR op	PC ← (PC) +1; PUSH(PCL);					DIR	BD
JSR op	SP←SP –1; PC ← (PC) +1;					EST	CD
JSR op,X	PUSH(PCH); SP←SP –1;	|	|	|	|	IX2	DD
JSR op,X	PC←ENDEREÇO CODICIONAL					IX1	ED
JSR X						IX	FD
LDA #op						IME	A6
LDA op						DIR	B6
LDA op	A ← (M)	|	X	X	|	EST	C6
LDA op,X						IX2	D6
LDA op,X						IX1	E6
LDA X						IX	F6
LDX #op						IME	AE
LDX op						DIR	BE
LDX op	X ← (M)	|	X	X	|	EST	CE
LDX op,X						IX2	DE
LDX op,X						IX1	EE
LDX X						IX	FE
LSL op						DIR	38
LSL A						INR	48
LSL X		|	X	X	X	INR	58
LSL op,X						IX1	68
LSL X						IX	78
LSR op						DIR	34
LSR A						INR	44
LSR X		|	X	X	X	INR	54
LSR op,X						IX1	64
LSR X						IX	74
MUL	X:A← (X) x (A)	0	|	|	0	INR	42
NEG op	M← $00 - (M)					DIR	3F
NEG A	A← $00 - (M)					INR	4F
NEG X	X← $00 - (M)	|	X	X	X	INR	5F
NEG op,X	M← $00 - (M)					IX1	6F
NEG X	M← $00 - (M)					IX	7F
NOP	NENHUMA OPERAÇÃO	|	|	|	|	INR	9D

ORA #op ORA op ORA op ORA op,X ORA op,X ORA ,X	A← (A) OR (M)	|	X	X	|	IME DIR EST IX2 IX1 IX	A8 B8 C8 D8 E8 F8
ROL op ROL A ROL X ROL op,X ROL ,X		|	X	X	X	DIR INR INR IX1 IX	39 49 59 69 79
ROR op ROR A ROR X ROR op,X ROR ,X		|	X	X	X	DIR INR INR IX1 IX	36 46 56 66 76
RSP	SP←$00FF	|	|	|	|	INR	9C
RST	SP←SP + 1; POP(PCH) SP←SP + 1; POP(PCL)	|	|	|	|	INR	81
SBC #op SBC op SBC op SBC op,X SBC op,X SBC ,X	A← (A) - (M) – (C)	|	X	X	X	IME DIR EST IX2 IX1 IX	A2 B2 C2 D2 E2 F2
SEC	C←1	|	|	|	1	INR	99
STA op STA op STA op,X STA op,X STA X	M ← (A)	|	X	X	|	DIR EST IX2 IX1 IX	B7 C7 D7 E7 F7
STX op STX op STX op,X STX op,X STX X	M ← (X)	|	X	X	|	DIR EST IX2 IX1 IX	BF CF DF EF FF
SUB #op SUB op SUB op SUB op,X SUB op,X SUB ,X	A← (A) - (M)	|	X	X	X	IME DIR EST IX2 IX1 IX	A0 B0 C0 D0 E0 F0
TAX	X←(A)	|	|	|	|	INR	97

TST op TST A TST X TST op,X TST X	(M) - $00	|	X	X	|	DIR INR INR IX1 IX	3F 4F 5F 6F 7F
TXA	A←(X)	|	|	|	|	INR	9F

Tabela 7.13 - Descrição completa do conjunto de instruções do HC05

7.8.5 – ESTATÍSTICAS DE DESEMPENHO DO PROCESSADOR HC05

A tabela 7.14 mostra as estatísticas de desempenho do processador HC05 implementado em um FPGA Virtex de 100K *gates*. Com base nos números gerados, pode-se notar a robustez deste processador. Observa-se que foi utilizado 850 CLBs. Isto significa que aproximadamente 70% do FPGA foi ocupado, lembrando que foi implementado apenas o processador. Sua arquitetura complexa e o grande número de instruções influenciam diretamente no desempenho temporal do processador implementado. O tempo total de aproximadamente 50ns é relativamente bom considerando a robustez do circuito.

Não foi possível implementar o processador HC05 no FPGA XC4000XL por limitação de espaço físico.

Recursos utilizados do FPGA				
	FF	**LUTs 4**	**LUTs 3**	**CLBs**
Processador HC05	300	800	400	850
Temporização do processador				
	Tempo de lógica	**Tempo de roteamento**		**Tempo total**
Processador HC05	20.040ns	32.031ns		52.071ns

Tabela 7.14 - Estatísticas de desempenho em FPGA do processador HC05

7.8.6 – CÓDIGO VHDL DO PROCESSADOR HC05

HC05

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
```

```
entity hc05 is
port (clk : in  std_logic;
      rst : in  std_logic;
      ce  : out std_logic;
      oe  : out std_logic;
      we  : out std_logic;
      data : inout std_logic_vector(7 downto 0);
      addr : out std_logic_vector(15 downto 0));
end entity hc05;

architecture comp_hc05 of hc05 is

-- registradores

signal a : std_logic_vector(7 downto 0); -- acumulador
signal x : std_logic_vector(7 downto 0); -- indexador
signal sp : std_logic_vector(15 downto 0); -- ponteiro de pilha
signal pc : std_logic_vector(15 downto 0); -- contador de programa
signal cc : std_logic_vector(7 downto 0); -- codigo de condicao

-- sinais auxiliares

signal aux_pc: std_logic_vector(15 downto 0); -- armazena o endereco do pc;
signal state : std_logic_vector(2 downto 0); -- estado
signal opcode : std_logic_vector(7 downto 0); -- opcode
signal inst_opcode : std_logic_vector(4 downto 0); -- inst_opcode
alias  opclass : std_logic_vector(2 downto 0) is opcode(6 downto 4); -- opclass
alias  code : std_logic_vector(4 downto 0) is inst_opcode(4 downto 0); -- code
signal mdr:std_logic_vector(7 downto 0);
signal f_rw:std_logic;
constant zbit : integer:=1;
constant nbit : integer:=2;
constant cbit : integer:=0;
constant hbit : integer:=4;

-- estados

constant reset  : std_logic_vector(2 downto 0):="000";
constant inic   : std_logic_vector(2 downto 0):="001";
constant busca  : std_logic_vector(2 downto 0):="010";
constant exec   : std_logic_vector(2 downto 0):="011";

--  instrucoes

constant lda    : std_logic_vector(4 downto 0):="10110";
constant ldx    : std_logic_vector(4 downto 0):="11110";
constant inc    : std_logic_vector(4 downto 0):="01100";
constant dec    : std_logic_vector(4 downto 0):="01010";
constant neg    : std_logic_vector(4 downto 0):="00000";
constant asl    : std_logic_vector(4 downto 0):="01000";
constant lsl    : std_logic_vector(4 downto 0):="01000";
constant asr    : std_logic_vector(4 downto 0):="00111";
constant lsr    : std_logic_vector(4 downto 0):="00100";
constant rola   : std_logic_vector(4 downto 0):="01001";
constant rora   : std_logic_vector(4 downto 0):="00110";
constant com    : std_logic_vector(4 downto 0):="00011";
constant clr    : std_logic_vector(4 downto 0):="01111";
constant sta    : std_logic_vector(4 downto 0):="10111";
```

```
constant stx    : std_logic_vector(4 downto 0):="11111";
constant tst    : std_logic_vector(4 downto 0):="01101";
constant and2   : std_logic_vector(4 downto 0):="10100";
constant ora    : std_logic_vector(4 downto 0):="11010";
constant eor    : std_logic_vector(4 downto 0):="11000";
constant add    : std_logic_vector(4 downto 0):="11011";
constant adc    : std_logic_vector(4 downto 0):="11001";
constant sub    : std_logic_vector(4 downto 0):="10000";
constant sbc    : std_logic_vector(4 downto 0):="10010";
constant cmp    : std_logic_vector(4 downto 0):="10001";
constant cpx    : std_logic_vector(4 downto 0):="10011";
constant bit    : std_logic_vector(4 downto 0):="10101";
constant jmp    : std_logic_vector(4 downto 0):="11100";
constant jsr    : std_logic_vector(4 downto 0):="11101";
constant nop    : std_logic_vector(7 downto 0):="10011101";
constant sec    : std_logic_vector(7 downto 0):="10011001";
constant tax    : std_logic_vector(7 downto 0):="10010111";
constant txa    : std_logic_vector(7 downto 0):="10011111";
constant clc    : std_logic_vector(7 downto 0):="10011000";
constant bcc,bhs : std_logic_vector(7 downto 0):="00100100";
constant bcs,blo : std_logic_vector(7 downto 0):="00100101";
constant beq    : std_logic_vector(7 downto 0):="00100111";
constant bhcc   : std_logic_vector(7 downto 0):="00101000";
constant bhcs   : std_logic_vector(7 downto 0):="00101001";
constant bhi    : std_logic_vector(7 downto 0):="00100010";
constant bls    : std_logic_vector(7 downto 0):="00100011";
constant bmi    : std_logic_vector(7 downto 0):="00101011";
constant bne    : std_logic_vector(7 downto 0):="00100110";
constant bpl    : std_logic_vector(7 downto 0):="00101010";
constant bra    : std_logic_vector(7 downto 0):="00100000";
constant brn    : std_logic_vector(7 downto 0):="00100001";
constant brclr  : std_logic_vector(7 downto 0):="00100001";
constant rsp    : std_logic_vector(7 downto 0):="10011100";
constant rts    : std_logic_vector(7 downto 0):="10000001";
constant bsr    : std_logic_vector(7 downto 0):="10101101";
constant fib    : std_logic_vector(7 downto 0):="00101110";-- fim das instrucoes de branch

-- enderecamento

constant imm  : std_logic_vector(2 downto 0):="010";
constant dir  : std_logic_vector(2 downto 0):="011";
constant ext  : std_logic_vector(2 downto 0):="100";
constant ix2  : std_logic_vector(2 downto 0):="101";
constant ix1  : std_logic_vector(2 downto 0):="110";
constant ix   : std_logic_vector(2 downto 0):="111";
constant inha : std_logic_vector(2 downto 0):="100";
constant inhx : std_logic_vector(2 downto 0):="101";

begin

-- armazena o valor puro do opcode
inst_opcode <= opcode(7)&opcode(3 downto 0);
addr<=pc;

-- tratamento com a memória RAM externa
with f_rw select
data <= mdr when '1',
"zzzzzzzz" when others;
```

```
oe<='0' when f_rw='0' else '1';
we<='1' when f_rw='0' else '0';
ce<='0';

-- processo principal

process(clk,rst)

variable f_inic     : std_logic_vector(2 downto 0);
variable hi_nibble  : std_logic_vector(7 downto 0);
variable lo_nibble  : std_logic_vector(7 downto 0);
variable loop_state : std_logic_vector(2 downto 0);
variable reg_aux    : std_logic_vector(8 downto 0);

-- procedimento incrementa pc

procedure inc_pc is
begin
        pc<=pc+1;
end procedure inc_pc;

-- procedimento para setar o valor do flag Z do registrador CC

procedure zerobit (reg_bit:std_logic_vector(8 downto 0))  is
begin
  if reg_bit="000000000" then
    cc(zbit)<='1';
  else
    cc(zbit)<='0';
  end if;
end procedure;

-- procediemnto que realiza as operações de ULA

procedure ula (op1,op2:std_logic_vector(8 downto 0);tipo:std_logic_vector(4 downto 0)) is
begin
   case tipo is
      when and2|ora|eor|bit =>
         if tipo=and2 or tipo=bit then
            reg_aux := op1 and op2;
         elsif tipo=ora then
            reg_aux := op1 or op2;
         else
            reg_aux := op1 xor op2;
          end if;
       if tipo/=bit then
          a <= reg_aux(7 downto 0);
       end if;
       cc(nbit)<=reg_aux(7);
       zerobit(reg_aux);
    when add|adc|inc=>
         if tipo=add or tipo=inc then
            reg_aux := op1 + op2;
         else
            reg_aux := op1 + op2 + cc(cbit);
         end if;
         if tipo=inc then
```

```
      mdr<=reg_aux(7 downto 0);
      cc(nbit)<=reg_aux(7);
      zerobit(reg_aux);
    else
      a <= reg_aux(7 downto 0);
      cc(cbit)<=reg_aux(8);
      cc(hbit)<=reg_aux(5);
      cc(nbit)<=reg_aux(7);
      zerobit(reg_aux);
    end if;
when sub|sbc|cmp|cpx|dec|neg|com|tst =>
    if tipo=sub or tipo=cmp or tipo=cpx or tipo=dec or tipo=neg or tipo=com or tipo=tst then
        reg_aux := op1 - op2;
    else
        reg_aux := op1 - op2 - cc(cbit);
    end if;
    if tipo=sub or tipo=sbc then
        a <= reg_aux(7 downto 0);
    elsif tipo=dec or tipo=neg or tipo=com then
          mdr<=reg_aux(7 downto 0);
    end if;
    if tipo = com then
      cc(cbit)<='1';
    end if;
    if tipo/=dec or tipo/=tst  then
      cc(cbit)<=reg_aux(8);
    end if;
    cc(nbit)<=reg_aux(7);
    zerobit(reg_aux);
when clr =>
    if tipo = clr then
       mdr<=(others=>'0');
       cc(nbit)<='0';
       cc(zbit)<='1';
    end if;
when asl|asr|lsr|rola|rora => --lsl
    if tipo=asl then
       for i in 0 to 6 loop
          reg_aux(i+1) := op1(i);
       end loop;
    reg_aux(0):='0';
    cc(cbit)<=op1(7);
    elsif tipo=asr then
      for i in 6 downto 0 loop
         reg_aux(i) := op1(i+1);
       end loop;
     cc(cbit)<=op1(0);
     elsif tipo=lsr then
        for i in 6 downto 0 loop
           reg_aux(i) := op1(i+1);
        end loop;
     reg_aux(7):='0';
     cc(cbit)<=op1(0);
     elsif tipo=rola then
       reg_aux(0):=cc(cbit);
       for i in  0 to 6 loop
          reg_aux(i+1) := op1(i);
       end loop;
```

```
            cc(cbit)<=op1(7);
          elsif tipo=rora then
            reg_aux(7):=cc(cbit);
            for i in 6 downto 0 loop
               reg_aux(i) := op1(i+1);
            end loop;
            cc(cbit)<=op1(0);
        end if;
     mdr <= reg_aux(7 downto 0);
     zerobit(reg_aux);
     if tipo/=lsr then
        cc(nbit)<=reg_aux(7);
     else
        cc(nbit)<='0';
     end if;
     when others => null;
   end case;
end procedure;

procedure define_op is
variable tipo:std_logic_vector(4 downto 0);
begin
    case code is
     when and2  => ula ('0'&a,'0'&data,and2);
     when ora   => ula ('0'&a,'0'&data,ora);
     when eor   => ula ('0'&a,'0'&data,eor);
     when add   => ula ('0'&a,'0'&data,add);
     when adc   => ula ('0'&a,'0'&data,adc);
     when sub   => ula ('0'&a,'0'&data,sub);
     when sbc   => ula ('0'&a,'0'&data,sbc);
     when cmp   => ula ('0'&a,'0'&data,cmp);
     when cpx   => ula ('0'&a,'0'&data,cpx);
     when inc   => ula ('0'&data,"000000001",inc);
     when dec   => ula ('0'&data,"000000001",dec);
     when neg   => ula ("000000000",'0'&data,neg);
     when com   => ula ("111111111",'0'&data,com);
     when clr   => ula ("xxxxxxxxx","xxxxxxxxx",clr);
     when asl   => ula ('0'&data,"000000000",asl);
     when asr   => ula ('0'&data,"000000000",asr);
     when lsr   => ula ('0'&data,"000000000",lsr);
     when rola  => ula ('0'&data,"000000000",rola);
     when rora  => ula ('0'&data,"000000000",rora);
     when tst     => ula ('0'&data,"000000000",tst);
     when others => null;
end case;
end procedure;

begin
   if rst='0' then
     state <= reset;
   elsif clk'event and clk='1' then
   case state is

      -- estado reset da UCP

      when reset =>
         f_rw <='0';
         pc     <= "1111111111111110"; -- fffe
```

```
      a      <= (others=>'0');
      x      <= (others=>'0');
      sp     <= "0000000011111111"; -- 00ff
      cc     <= (others=>'0');
      f_inic := (others=>'0');
      loop_state:= (others=>'0');
      state <= inic;

-- estado que localiza onde começa o programa na memória RAM

when inic =>
      if f_inic ="00" then
          f_rw <='0';
          pc <= "1111111111111110"; -- fffe
          f_inic:="001";
      elsif f_inic="001" then
          hi_nibble:=data;
          pc <= "1111111111111111"; -- ffff
          f_inic:="010";
      elsif f_inic="010" then
          pc<=hi_nibble&data;
          f_inic:="000";
          state<=busca;
      end if;

-- estado de busca de instruções na memória

when busca =>
   f_rw<='0';
   opcode<=data;
   inc_pc;
   state<=exec;

-- decodificação e execução da instruções

when exec =>
   if opcode(7 downto 5)="000" then
      if loop_state="000" then
         aux_pc<=pc+2;
         reg_aux(7 downto 0):=data;
         inc_pc;
         loop_state:="001";
      elsif loop_state="001" then
          pc<="00000000"&reg_aux(7 downto 0);
          if opcode(4)/='1' then
              reg_aux(7 downto 0):=data;
          end if;
          loop_state:="010";
          elsif loop_state="010" then
          case opcode(4 downto 0) is
-- brclr
when "00001"=> if data(0)='0' then loop_state:="100"; else loop_state:="011"; end if;
when "00011"=> if data(1)='0' then loop_state:="100"; else loop_state:="011"; end if;
when "00101"=> if data(2)='0' then loop_state:="100"; else loop_state:="011"; end if;
when "00111"=> if data(3)='0' then loop_state:="100"; else loop_state:="011"; end if;
when "01001"=> if data(4)='0' then loop_state:="100"; else loop_state:="011"; end if;
when "01011"=> if data(5)='0' then loop_state:="100"; else loop_state:="011"; end if;
when "01101"=> if data(6)='0' then loop_state:="100"; else loop_state:="011"; end if;
```

```
      when "01111"=> if data(7)='0' then loop_state:="100"; else loop_state:="011"; end if;
      -- brset
      when "00000"=> if data(0)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "00010"=> if data(1)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "00100"=> if data(2)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "00110"=> if data(3)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "01000"=> if data(4)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "01010"=> if data(5)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "01100"=> if data(6)='1' then loop_state:="100"; else loop_state:="011"; end if;
      when "01110"=> if data(7)='1' then loop_state:="100"; else loop_state:="011"; end if;
      -- bset
      when "10000"=> mdr<=data(7 downto 1)&'1'; loop_state:="101";
      when "10010"=> mdr<=data(7 downto 2)&'1'&data(0); loop_state:="101";
      when "10100"=> mdr<=data(7 downto 3)&'1'&data(1 downto 0); loop_state:="101";
      when "10110"=> mdr<=data(7 downto 4)&'1'&data(2 downto 0); loop_state:="101";
      when "11000"=> mdr<=data(7 downto 5)&'1'&data(3 downto 0); loop_state:="101";
      when "11010"=> mdr<=data(7 downto 6)&'1'&data(4 downto 0); loop_state:="101";
      when "11100"=> mdr<=data(7)&'1'&data(5 downto 0); loop_state:="101";
      when "11110"=> mdr<='1'&data(6 downto 0); loop_state:="101";
      --bclr
      when "10001"=> mdr<=data(7 downto 1)&'0'; loop_state:="101";
      when "10011"=> mdr<=data(7 downto 2)&'0'&data(0); loop_state:="101";
      when "10101"=> mdr<=data(7 downto 3)&'0'&data(1 downto 0); loop_state:="101";
      when "10111"=> mdr<=data(7 downto 4)&'0'&data(2 downto 0); loop_state:="101";
      when "11001"=> mdr<=data(7 downto 5)&'0'&data(3 downto 0); loop_state:="101";
      when "11011"=> mdr<=data(7 downto 6)&'0'&data(4 downto 0); loop_state:="101";
      when "11101"=> mdr<=data(7)&'0'&data(5 downto 0); loop_state:="101";
      when "11111"=> mdr<='0'&data(6 downto 0); loop_state:="101";
      when others=> null;
   end case;
elsif loop_state="100" then
    if reg_aux(7)='1' then
       pc <= pc + reg_aux(6 downto 0);
    else
    pc <= pc - reg_aux(6 downto 0);
    end if;
    loop_state:="000";
    state<=busca;
elsif loop_state="011" then
    pc <= aux_pc;
    loop_state:="000";
    state<=busca;
elsif loop_state="101" then
    f_rw<='1';
    loop_state:="110";
elsif loop_state="110" then
    f_rw<='0';
    pc <= aux_pc -1;
    state<=busca;
end if;
    else
    case opcode is
      when bsr=>
       if loop_state="000" then
          aux_pc<= pc+1;
          lo_nibble:=data;
             pc <= sp;
             loop_state:="001";
```

```
        elsif loop_state="001" then
            sp<=sp-1;
            mdr <= aux_pc(7 downto 0);
            loop_state:="010";
        elsif loop_state="010" then
            f_rw<='1';
            f_inic:="011";
        elsif loop_state="011" then
            f_rw<='0';
            pc <= sp;
            loop_state:="100";
        elsif loop_state="100" then
           sp<=sp-1;
           mdr <= aux_pc(15 downto 8);
           loop_state:="101";
        elsif loop_state="101" then
           f_rw<='1';
           loop_state:="110";
        elsif loop_state ="110" then
           f_rw<='0';
           loop_state:="111";
        else
           if lo_nibble(7)='1' then
               pc <= pc + lo_nibble(6 downto 0);
           else
               pc <= pc - lo_nibble(6 downto 0);
           end if;
           loop_state:="000";
           state<=busca;
         end if;
when rts=>
      if loop_state="000" then
        pc <= sp+1;
        loop_state:="001";
      elsif loop_state="001" then
        reg_aux(7 downto 0):=data;
        sp<=sp+1;
        loop_state:="010";
      elsif loop_state="010" then
        pc <= sp+1;
        loop_state:="011";
      else
        pc<=reg_aux(7 downto 0)&data;
        sp<=sp+1;
        loop_state:="000";
        state<=busca;
      end if;
when rsp =>
      sp <= "0000000011111111";
      state<= busca;
when tax =>
      x<=a;
      state<=busca;
when txa =>
      a<=x;
      state<=busca;
when clc =>
     cc(cbit)<='0';
```

```
      state<=busca;
when nop|brn =>
      state<=busca;
when sec =>
      cc(cbit)<='1';
      state<=busca;
when bcc => -- igual a bhs
      if cc(cbit)='0' then
          opcode <= fib;
      end if;
when bcs => -- igual a blo
      if cc(cbit)='1' then
         opcode <= fib;
      end if;
when beq=>
      if cc(zbit)='1' then
        opcode <= fib;
      end if;
when bhcc=>
      if cc(hbit)='0' then
        opcode <= fib;
      end if;
when bhcs=>
      if cc(hbit)='1' then
        opcode <= fib;
      end if;
when bhi=>
      if cc(cbit)='0' and cc(zbit)='0' then
         opcode <= fib;
      end if;
when bls=>
      if cc(cbit)='1' or cc(zbit)='1' then
         opcode <= fib;
      end if;
when bmi=>
      if cc(nbit)='1' then
         opcode <= fib;
      end if;
when bne=>
      if cc(zbit)='0' then
        opcode <= fib;
      end if;
when bpl=>
      if cc(nbit)='0' then
         opcode <= fib;
      end if;
when bra=>
      opcode <= fib;
when fib =>
      if data(7)='1' then
         pc <= pc + data(6 downto 0);
      else
         pc <= pc - data(6 downto 0);
      end if;
      state<=busca;
when others =>
case code is
    when lda|ldx|jmp|sta|stx|jsr =>
```

```
case opclass is
  when imm =>
    if code=lda then
        a<=data;
    else
        x<=data;
    end if;
    inc_pc;
    state<=busca;
  when dir =>
    if loop_state="000" then
      if code=jsr and f_inic/="111" then
        if f_inic="000" then
            aux_pc<= pc+1;
            lo_nibble:=data;
            pc <= sp;
            f_inic:="001";
        elsif f_inic="001" then
            sp<=sp-1;
            mdr <= aux_pc(7 downto 0);
            f_inic:="010";
        elsif f_inic="010" then
            f_rw<='1';
            f_inic:="011";
        elsif f_inic="011" then
            f_rw<='0';
            pc <= sp;
            f_inic:="100";
        elsif f_inic="100" then
            sp<=sp-1;
            mdr <= aux_pc(15 downto 8);
            f_inic:="101";
        elsif f_inic="101" then
            f_rw<='1';
            f_inic:="110";
        else
            f_rw<='0';
            f_inic:="111";
        end if;
      else
        if code/=jsr then
            aux_pc<=pc+1;
            pc <= "00000000"&data;
        else
            pc<="00000000"&lo_nibble;
        end if;
        if code=jmp or code=jsr then
            f_inic:="000";
            state<= busca;
        else
            loop_state:="001";
        end if;
      end if;
  elsif loop_state="001" then
    if code=sta or code=stx then
       if f_inic="00" then
     if code=sta then
        mdr<=a;
```

```
            else
               mdr<=x;
            end if;
            f_inic:="001";
        elsif f_inic="001" then
            f_rw<='1';
            f_inic:="010";
        else
            f_rw<='0';
            f_inic:="000";
            pc <= aux_pc;
            loop_state:="000";
            state<=busca;
        end if;
      elsif code=lda then
        a<=data;
        pc <= aux_pc;
        loop_state:="000";
        state<=busca;
      else
        x<=data;
        pc <= aux_pc;
        loop_state:="000";
        state<=busca;
      end if;
    end if;
  when ext =>
     if loop_state="000" then
         hi_nibble:=data;
         inc_pc;
         loop_state:="001";
     elsif loop_state="001" then
       if code=jsr and f_inic/="111" then
         if f_inic="000" then
            aux_pc<= pc+1;
            lo_nibble:=data;
            pc <= sp;
            f_inic:="001";
         elsif f_inic="001" then
           sp<=sp-1;
           mdr <= aux_pc(7 downto 0);
           f_inic:="010";
         elsif f_inic="010" then
           f_rw<='1';
           f_inic:="011";
         elsif f_inic="011" then
          f_rw<='0';
          pc <= sp;
          f_inic:="100";
         elsif f_inic="100" then
          sp<=sp-1;
          mdr <= aux_pc(15 downto 8);
          f_inic:="101";
        elsif f_inic="101" then
          f_rw<='1';
          f_inic:="110";
        else
          f_rw<='0';
```

```
          f_inic:="111";
      end if;
    else
      if code/=jsr then
        aux_pc<=pc+1;
        pc <= hi_nibble&data;
      else
        pc<=hi_nibble&lo_nibble;
      end if;
      if code=jmp or code=jsr then
        f_inic:="000";
        loop_state:="000";
        state<= busca;
      else
        loop_state:="010";
      end if;
end if;
elsif loop_state="010" then
      if code=sta or code=stx then
       if f_inic="000" then
         if code=sta then
           mdr<=a;
         else
           mdr<=x;
         end if;
         f_inic:="001";
       elsif f_inic="001" then
         f_rw<='1';
         f_inic:="010";
      else
         f_rw<='0';
         f_inic:="000";
         pc <= aux_pc;
         loop_state:="000";
         state<=busca;
      end if;
    elsif code=lda then
      a<=data;
      pc <= aux_pc;
      loop_state:="000";
      state<=busca;
    else
     x<=data;
     pc <= aux_pc;
     loop_state:="000";
     state<=busca;
    end if;
   end if;
when ix2 =>
      if loop_state="000" then
         hi_nibble:=data;
         inc_pc;
         loop_state:="001";
      elsif loop_state="001" then
         if code=jsr and f_inic/="111" then
            if f_inic="000" then
           aux_pc<= pc+1;
           lo_nibble:=data;
```

```
        pc <= sp;
        f_inic:="001";
    elsif f_inic="001" then
        sp<=sp-1;
        mdr <= aux_pc(7 downto 0);
        f_inic:="010";
    elsif f_inic="010" then
        f_rw<='1';
        f_inic:="011";
    elsif f_inic="011" then
        f_rw<='0';
        pc <= sp;
        f_inic:="100";
    elsif f_inic="100" then
        sp<=sp-1;
        mdr <= aux_pc(15 downto 8);
        f_inic:="101";
    elsif f_inic="101" then
        f_rw<='1';
        f_inic:="110";
    else
        f_rw<='0';
        f_inic:="111";
    end if;
  else
    if code/=jsr then
        aux_pc<=pc+1;
        pc <= x+(hi_nibble&data);
    else
        pc <= x+(hi_nibble&lo_nibble);
    end if;
    if code=jmp or code=jsr then
        f_inic:="000";
        loop_state:="000";
        state<= busca;
    else
        loop_state:="010";
    end if;
  end if;
  else
  if code=sta or code=stx then
     if f_inic="000" then
        if code=sta then
           mdr<=a;
        else
           mdr<=x;
        end if;
        f_inic:="001";
    elsif f_inic="001" then
        f_rw<='1';
        f_inic:="010";
    else
       f_rw<='0';
       f_inic:="000";
       pc <= aux_pc;
       loop_state:="000";
      state<=busca;
    end if;
```

```
    elsif code=lda then
       a<=data;
       pc <= aux_pc;
       loop_state:="000";
       state<=busca;
    else
       x<=data;
       pc <= aux_pc;
       loop_state:="000";
       state<=busca;
    end if;
  end if;
when ix1 =>
    if loop_state="000" then
      if code=jsr and f_inic/="111" then
         if f_inic="000" then
            aux_pc<= pc+1;
            lo_nibble:=data;
            pc <= sp;
            f_inic:="001";
         elsif f_inic="001" then
            sp<=sp-1;
            mdr <= aux_pc(7 downto 0);
            f_inic:="010";
         elsif f_inic="010" then
            f_rw<='1';
            f_inic:="011";
         elsif f_inic="011" then
            f_rw<='0';
            pc <= sp;
            f_inic:="100";
         elsif f_inic="100" then
            sp<=sp-1;
            mdr <= aux_pc(15 downto 8);
            f_inic:="101";
         elsif f_inic="101" then
            f_rw<='1';
            f_inic:="110";
         else
            f_rw<='0';
            f_inic:="111";
         end if;
      else
         if code/=jsr then
            aux_pc<=pc+1;
            pc <= (x + "0000000000000000") + (data);
         else
            pc<=(x + "0000000000000000") + (lo_nibble);
         end if;
         if code=jmp or code=jsr then
            f_inic:="000";
            state<= busca;
         else
            loop_state:="001";
         end if;
      end if;
    else
      if code=sta or code=stx then
```

```
        if f_inic="000" then
          if code=sta then
            mdr<=a;
          else
            mdr<=x;
          end if;
          f_inic:="001";
       elsif f_inic="001" then
          f_rw<='1';
          f_inic:="010";
       else
          f_rw<='0';
          f_inic:="000";
          pc <= aux_pc;
          loop_state:="000";
          state<=busca;
       end if;
     elsif code=lda then
        a<=data;
        pc <= aux_pc;
        loop_state:="000";
        state<=busca;
     else
        x<=data;
        pc <= aux_pc;
        loop_state:="000";
        state<=busca;
     end if;
   end if;
when ix  =>
    if loop_state="000" then
      if code=jsr and f_inic/="111" then
        if f_inic="000" then
           aux_pc<= pc;
           pc <= sp;
           f_inic:="001";
       elsif f_inic="001" then
           sp<=sp-1;
           mdr <= aux_pc(7 downto 0);
           f_inic:="010";
       elsif f_inic="010" then
           f_rw<='1';
           f_inic:="011";
       elsif f_inic="011" then
           f_rw<='0';
           pc <= sp;
           f_inic:="100";
       elsif f_inic="100" then
           sp<=sp-1;
           mdr <= aux_pc(15 downto 8);
           f_inic:="101";
       elsif f_inic="101" then
           f_rw<='1';
           f_inic:="110";
       else
           f_rw<='0';
           f_inic:="111";
        end if;
```

```
      else
        if code/=jsr then
           aux_pc<=pc;
           pc <= "00000000"&x;
        else
           pc<="00000000"&x;
        end if;
        if code=jmp or code=jsr then
            f_inic:="000";
            state<= busca;
        else
           loop_state:="001";
        end if;
    end if;
   else
     if code=sta or code=stx then
       if f_inic="000" then
         if code=sta then
           mdr<=a;
         else
           mdr<=x;
         end if;
         f_inic:="001";
       elsif f_inic="001" then
         f_rw<='1';
         f_inic:="010";
       else
         f_rw<='0';
         f_inic:="000";
         pc <= aux_pc;
         loop_state:="000";
         state<=busca;
      end if;
   elsif code=lda then
      a<=data;
      pc <= aux_pc;
      loop_state:="000";
      state<=busca;
   else
      x<=data;
      pc <= aux_pc;
      loop_state:="000";
      state<=busca;
   end if;
end if;
when others=> null;
end case;
if code=lda or code=sta then
     cc(nbit)<=a(7);
     zerobit('0'&a);
elsif code=ldx or code=stx then
     cc(nbit)<=x(7);
     zerobit('0'&x);
end if;
when inc|dec|neg|asl|asr|lsr|rola|rora|com|clr|tst =>
    case opclass is
     when dir =>
       if loop_state="000" then
```

```
          aux_pc<=pc+1;
          pc <= "00000000"&data;
          loop_state:="001";
      elsif loop_state="001" then
          define_op;
          if code/=tst then
              loop_state:="010";
          else
              loop_state:="100";
          end if;
      elsif loop_state="010" then
          f_rw<='1';
          loop_state:="011";
      elsif loop_state="011" then
          f_rw<='0';
          loop_state:="100";
      else
          pc <= aux_pc;
          loop_state:="000";
          state<=busca;
      end if;
when inha =>
      define_op;
      if code/=tst then
          a<=mdr;
      end if;
      state<=busca;
when inhx =>
      define_op;
       if code/=tst then
           x<=mdr;
       end if;
       state<=busca;
when ix1 =>
       if loop_state="000" then
           aux_pc<=pc+1;
           pc <= (x + "0000000000000000") + (data);
           loop_state:="001";
      elsif loop_state="001" then
          define_op;
          if code/=tst then
              loop_state:="010";
          else
              loop_state:="100";
          end if;
      elsif loop_state="010" then
          f_rw<='1';
          loop_state:="011";
      elsif loop_state="011" then
          f_rw<='0';
          loop_state:="100";
      else
          pc <= aux_pc;
          loop_state:="000";
          state<=busca;
      end if;
when ix  =>
       if loop_state="000" then
```

```
          aux_pc<=pc;
          pc <= "00000000"&x;
          loop_state:="001";
      elsif loop_state="001" then
          define_op;
          if code/=tst then
              loop_state:="010";
          else
              loop_state:="100";
          end if;
      elsif loop_state="010" then
          f_rw<='1';
          loop_state:="011";
      elsif loop_state="011" then
          f_rw<='0';
          loop_state:="100";
      else
          pc <= aux_pc;
          loop_state:="000";
          state<=busca;
      end if;
   when others=> null;
   end case;
  when and2|ora|eor|add|sub|adc|sbc|cmp|cpx =>
    case opclass is
      when imm =>
        define_op;
        inc_pc;
        state<= busca;
      when dir =>
         if loop_state="000" then
            aux_pc<=pc+1;
            pc <= "00000000"&data;
            loop_state:="001";
        else
           define_op;
           pc <= aux_pc;
           loop_state:="000";
           state<=busca;
        end if;
when ext =>
      if loop_state="000" then
          hi_nibble:=data;
          inc_pc;
          loop_state:="001";
     elsif loop_state="001" then
          aux_pc<=pc+1;
          pc <= hi_nibble&data;
          loop_state:="010";
     else
         define_op;
         pc <= aux_pc;
         loop_state:="000";
         state<=busca;
     end if;
when ix2 =>
      if loop_state="000" then
          hi_nibble:=data;
```

```
            inc_pc;
            loop_state:="001";
        elsif loop_state="001" then
            aux_pc<=pc+1;
            pc <= x+(hi_nibble&data);
            loop_state:="010";
        else
            define_op;
            pc <= aux_pc;
            loop_state:="000";
            state<=busca;
      end if;
    when ix1 =>
      if loop_state="000" then
          aux_pc<=pc+1;
          pc <= (x + "0000000000000000") + (data);
          loop_state:="001";
      else
          define_op;
          pc <= aux_pc;
          loop_state:="000";
          state<=busca;
      end if;
    when ix  =>
      if loop_state="000" then
          aux_pc<=pc;
          pc <= "00000000"&x;
          loop_state:="001";
      else
          define_op;
          pc <= aux_pc;
          loop_state:="000";
          state<=busca;
      end if;
    when others=> null;
    end case;
  when others => null;
  end case;
  end case;
  end if;
  when others => null;
  end case;
  end if;
  end process;
  end comp_hc05;
```

CAPÍTULO VIII – PROJETO E DESEMPENHO DE MICROCONTROLADOR EM FPGAS

Este capítulo irá expor uma noção inicial de assembler para o M68HC11, da Motorola. Será visto o formato de algumas instruções básicas, os passos que a UCP toma para executá-las e uma visão dos valores correntes na memória RAM, antes e depois de cada instrução. Com o auxílio de um programa de simulação, será exposto alguns pequenos programas de teste, que irão auxiliar na compreensão do M68HC11 e do protótipo desenvolvido.

8.1 - INTRODUÇÃO

Microcontroladores estão sendo cada vez mais utilizados nas indústrias e em projetos que envolvem informática e eletrônica. Um microcontrolador possui características de um sistema computacional completo em um único *chip*: memória RAM; memória EPROM ou EEPROM; unidade central de processamento (UCP); portas de comunicação I/O selecionáveis por software; bloco de comunicação com algum protocolo conhecido; timer gerador de sinais para interrupções; conversor A/D e outras características que variam de fabricante a fabricante.

Dentre os microcontroladores mais conhecidos do mercado pode-se citar: PIC (Microchip), 8051 (vários fabricantes), TMS (Texas Instruments) e COP8 (National Semiconductor). Este capítulo apresenta um protótipo do M68HC11 da Motorola.

O M68HC11 possui 3 registradores de 8 bits A, B, e CCR e 4 registradores de 16 bits: X, Y, SP e PC. Existe ainda um outro registrador "virtual" de 16 bits, o registrador D, que não existe fisicamente, sendo formado pela união de A e B. A figura 8.1, representa os registradores internos à UCP do M68HC11.

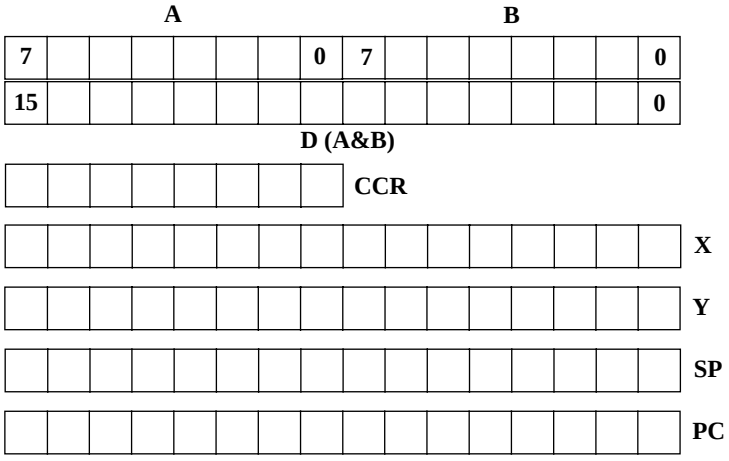


Figura 8.1 – Representação dos registradores internos à UCP do M68HC11

O Microcontrolador M68HC11 possui arquitetura interna bastante interessante, como visto na figura 8.2, na forma original.

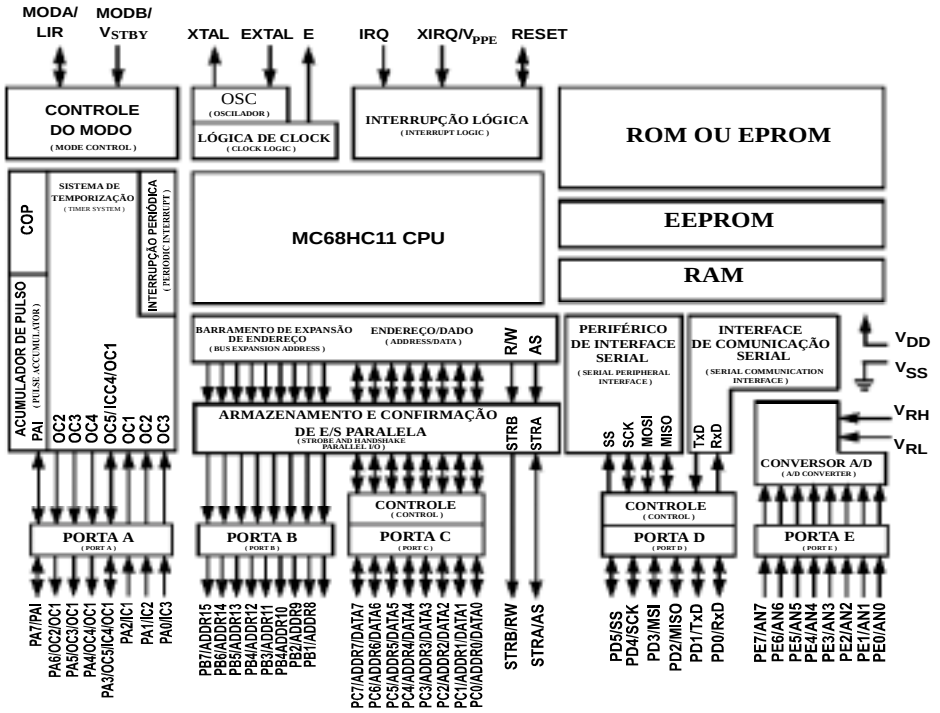


Figura 8.2: Estrutura interna do microcontrolador M68HC11.

Neste capítulo, serão descritos alguns programas do assembler para o M68HC11, com as devidas explicações passo-a-passo das instruções utilizadas, *debug* da RAM e dos *opcodes* criados pelo montador assembler, bem como a organização do programa e sua estrutura.

Os códigos são apresentados partindo de programas simples e aumentando a complexidade ao decorrer do capítulo. Nos exemplos iniciais, todas as instruções estão comentadas, sendo esta prática indicada num primeiro contato com uma linguagem. Por outro lado, a prática de comentários torna a leitura do código cansativa, principalmente quando já se está familiarizado com as instruções. Sendo assim, os comentários tornam-se mais escassos e objetivos, à medida que aumenta a complexidade dos programas.

Para realização do estudo do assembler do M68HC11, utiliza-se o programa Shadow11, versão 0.8, *freeware*. Este é um excelente simulador de programas assembler para o M68HC11, onde é possível acompanhar a execução de programas

passo a passo, visualizar os valores de todos os registradores internos à UCP, uma "RAM virtual" com os valores hexadecimais correspondentes aos mnemônicos assembler, as "pilhas" de dados montadas pela UCP na memória RAM, entre outros.

Para o correto funcionamento do Shadow11, é necessário um montador assembler, o ASM11, sendo utilizada a versão 1.84, *freeware*. A interface do programa Shadow11, pode ser visualizada na figura 8.3.

Com o uso da interface proporcionada pelo software Shadow11, visualiza-se com facilidade a forma e a disposição em que o programa escrito será "montado" na memória RAM, através de valores hexadecimais para o *chip*.

Será descrito um pequeno resumo sobre o "problema" que o programa deverá solucionar, permitindo uma compreensão do propósito do código.

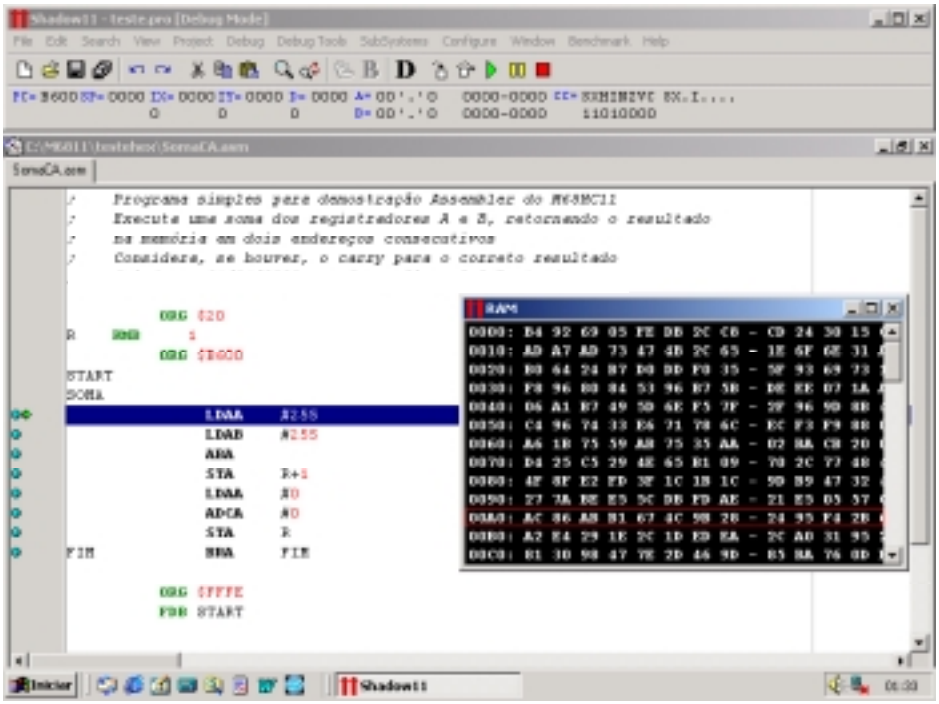


Figura 8.3- Interface do programa de simulação Shadow11

8.2 - MODOS DE ENDEREÇAMENTO DO M68HC11

O M68HC11 possui seis modos de endereçamento diferentes, ou seja, seis formas de acesso à memória RAM, sendo estes: *Immediate* - IMM (imediato), *Direct* –

DIR (direto), *Extended* – EXT (estendido), *Indexed* – IND (indexado), *Inherent* – INH (inerente) e *Relative* – REL (relativo). No *opcode*, há informação da localidade dos dados necessários à execução da operação, bastando identificar o modo que a memória deverá ser acessada.

Uma única instrução, como por exemplo a **ADDA,** que instrui à UCP realizar uma soma do registrador A com uma posição de memória, pode ser representada através dos *opcodes* 8B, 9B, BB, AB e o par "18 AB". Recebendo qualquer um destes *opcodes*, a UCP irá realizar a instrução **ADDA**. Porém, cada um instrui buscar o dado na memória de forma (endereços) diferente.

Nos modos DIR, EXT e IND ocorre um salto (*jump*) no endereçamento da RAM. O valor para qual será destinado o salto é o enfoque destes modos de operação da instrução.

Nos exemplos, são tomadas por base duas instruções **ADDA** e **ADDD**. Ambas tomadas aleatoriamente, sendo que a primeira trabalha em 8 bits e a segunda em 16 bits. Todos os valores de endereços e valores contidos em registradores são meramente ilustrativos, podendo assumir quaisquer outros valores, salvo exceções comentadas.

IMM: *IMMEDIATE* – MODO IMEDIATO

Neste modo, o dado a ser considerado está imediatamente após a instrução, e pode ser formado por um ou dois bytes, dependendo do registrador que será armazenado.

Exemplo 1: A instrução **ADDA**, faz com que a UCP realize uma soma do dado contido no registrador A com um dado da memória, e retorne o resultado no registrador A. Como o registrador A é de oito bits (1 byte), o dado a ser somado também deve ser de 1 byte e, no modo imediato, estará localizado na posição de endereço seguinte ao *opcode*.

Supõe que o registrador A possua o valor 26_H. Após a UCP receber a seqüência de dados $8B_H$ -15_H, o registrador A irá possuir $26_H + 15_H = 3B_H$, conforme visto na tabela 8.1.

Endereço (exemplo)	Memória	Descrição	Valor em A
0020	8B	ADDA – IMM	26_H
0021	15	Dado 15, que será somado ao valor de A	$26_H + 15_H$
-	-	Soma completa	$3B_H$

Tabela 8.1 - Instrução **ADDA** Modo Imediato (IMM)

Exemplo 2: A instrução **ADDD** faz com que a UCP faça uma soma do valor contido no registrador D com um dado da memória. Sendo o registrador D uma "união" do registrador A com o registrador B (D = A & B), portanto de 16 bits (2 bytes), é necessário o envio de 2 bytes para serem somados. O dado completo estará localizado nos 2 endereços consecutivos ao *opcode*.

A tabela 8.2 supõe que o registrador D possua 1500_H; após a seqüência $C3_H$ - 02_H -14_H, o registrador D irá conter o valor $1500_H + 0214_H = 1714_H$.

Endereço (exemplo)	Memória	Descrição	Valor em D
0020	C3	ADDD – IMM	1500_H
0021	02	Dado 02	1500_H
		A UCP automaticamente gera o próximo endereço (0022_H), pois D é de 16 bits.	
0022	14	Dado 14	$1500_H + 0214_H$
-	-	Soma completa	1714_H

Tabela 8.2: Instrução ADDD Modo Imediato (IMM)

DIR: *DIRECT* – MODO DIRETO

No modo direto, um único byte após o *opcode* formará o endereço do dado na RAM. Com apenas um byte é possível somente referenciar endereços de 0000_H até $00FF_H$, sendo esta faixa de memória suficiente para referenciar dados na maioria das aplicações. No modo direto, a parte alta do endereço é sempre zero.

Exemplo 3: Instrução ADDA no modo direto: o dado a ser somado está no endereço referenciado pelo byte após o *opcode*.

A tabela 8.3 supõe que o registrador A possua o valor 26_H, após a UCP receber a seqüência de dados $9B_H$ -18_H, gera-se o endereço 0018_H e assim encontra-se o dado 36_H a ser somado em A. O registrador A irá possuir $26_H + 36_H = 5C_H$.

Endereço	Memória	Descrição	Valor em A
0020	9B	ADDA – DIR	26_H
0021	18	Endereço do dado a ser somado com o valor de A	26_H
-	-	Novo endereço: 0018_H	26_H
0018	36	Dado 36	$26_H + 36_H$
-	-	Soma completa	$5C_H$

Tabela 8.3: Instrução ADDA modo direto (DIR)

Exemplo 4: Instrução **ADDD** no modo direto: de forma similar ao exemplo anterior, o endereço do dado a ser somado está referenciado por 1 byte após o *opcode*, sendo a parte alta do endereço, considerada como 00_H e com a diferença de que o dado

a ser somado será de 2 bytes. Desta forma, ao receber a seqüência $D3_H$ -40_H, a UCP irá buscar o primeiro byte no endereço 0040_H e o segundo byte no endereço 0041_H.

A tabela 8.4 supõe que o registrador D contenha o valor 1060_H, após a seqüência de instruções descrita, D irá conter 2185_H.

Endereço (exemplo)	Memória	Descrição	Valor em D
0020	D3	ADDD – DIR	1060_H
0021	40	Endereço inicial do dado a ser somado com o valor de D (o dado será formado pelos endereços 0040 e 0041)	1060_H
-	-	Novo endereço: 0040_H	1060_H
0040	11	Dado 11	1060_H
-	-	A UCP automaticamente gera o próximo endereço, pois D é de 16 bits.	-
0041	25	Dado 25	1060_H + 1125_H
-	-	Soma completa	2185_H

Tabela 8.4:ADDD modo direto (DIR)

EXT: *EXTENDED* – MODO ESTENDIDO

O endereço efetivo do dado está em 2 bytes após o *opcode*, desta forma pode-se referenciar qualquer endereço da memória, de 0000_H à $FFFF_H$. Qualquer instrução no modo estendido será formada por no mínimo 3 bytes (1 byte do *opcode*, 2 bytes do endereço)

Exemplo 5: O dado a ser somado está no endereço de 16 bits referenciado por 2 bytes após o *opcode*. A tabela 8.5 supõe que o registrador A possua o valor 26_H. Após a UCP receber a seqüência de dados BB_H - 15_H - 77_H, gera-se o endereço 1577_H diretamente, e assim encontra-se o dado 64_H a ser somado ao registrador A. Dessa maneira, o registrador A irá possuir a soma de 26_H com 64_H, que corresponde a $8A_H$.

Endereço (exemplo)	Memória	Descrição	Valor em A
0020	BB	ADDA – EXT	26_H
0021	15	Parte alta do endereço que contém o dado	26_H
0022	77	Parte baixa do endereço que contém o dado	26_H
-	-	Novo endereço: 1577_H	26_H
1577	64	Dado 64	$26_H + 64_H$
-	-	Soma completa	$8A_H$

Tabela 8.5: ADDA modo estendido (EXT)

Exemplo 6: ADDD modo EXT: o dado está no endereço composto por 2 bytes após o *opcode*. A tabela 8.6 mostra da instrução ADDD no modo EXT.

Endereço	Memória	Descrição	Valor em D
0020	F3	ADDD – EXT	1060_H
0021	51	Parte alta do endereço do dado a ser somado com o valor de D	1060_H
0022	80	Parte baixa do endereço do dado a ser somado com o valor de D	1060_H
-	-	Novo endereço: 5180_H	1060_H
5180	72	Dado 72	1060_H
-	-	A UCP gera o próximo endereço, pois D é de 16 bits.	1060_H
5181	13	Dado 13	1060_H + 7213_H
-	-	Soma completa	8273_H

Tabela 8.6: Instrução ADDD modo direto (EXT)

IND: *INDEXED* – MODO INDEXADO

No modo indexado, um segmento de 8 bits contido na instrução é adicionado ao valor contido em um registrador de índice (X ou Y). A soma é o endereço efetivo. Este modo de endereçamento permite referenciar qualquer endereço num espaço de 64k (16 bits de endereçamento).

Exemplo 7: ADDA modo IND: um endereço de memória será calculado com base nos registradores de índice (**X** ou **Y**). A UCP soma o byte em seqüência ao *opcode* com o valor de **X** ou **Y**. As tabelas 8.7 e 8.8 exemplificam o exposto.

Endereço (exemplo)	Memória	Descrição	Valor em A (exemplo)	Valor em X (exemplo)	Valor em Y (exemplo)
0500	AB	ADDA – IND	26_H	8006_H	0150_H
0501	70	Dado a ser somado com o valor de X, - pois não houve pré-byte – para o cálculo do novo endereço	26_H	8006_H	0150_H
-	-	A soma em X é efetuada	26_H	$8006_H + 70_H$	0150_H
-	-	Novo endereço: 8076_H	26_H	8006_H	0150_H
8076	61	Dado 61	$26_H + 61_H$	8006_H	0150_H
-	-	Soma completa	87_H	8006_H	0150_H

Tabela 8.7: Instrução ADDA modo indexado (IND), sem pré-byte (18_H)

Endereço (exemplo)	Memória	Descrição	Valor em A (exemplo)	Valor em X (exemplo)	Valor em Y (exemplo)
04FF	18	O pré-byte (18) anterior a instrução, define que será utilizado o valor do registrador Y para o próximo cálculo, independente de qual for a próxima instrução	26_H	8006_H	0150_H
0500	AB	ADDA – IND	26_H	8006_H	0150_H
0501	70	Dado a ser somado com o valor de Y, - pois houve pré-byte - para o cálculo do novo endereço	26_H	8006_H	0150_H
		A soma em Y é efetuada		8006_H	$0150_H + 70_H$
-	-	Novo endereço: $01C0_H$	26_H	8006_H	0150_H
01C0	61	Dado 61	$26_H + 61_H$	8006_H	0150_H
-	-	Soma completa	87_H	8006_H	0150_H

Tabela 8.8: Instrução ADDA Modo Indexado (IND), com pré-byte (18_H)

Exemplo 8: ADDD Modo Indexado – Similarmente a execução da instrução ADDA descrita anteriormente, ADDD no modo indexado procede da mesma maneira, com a diferença de que o dado final a ser somado estará em dois endereços, pois D é de 16 bits. Nas tabelas 8.9 e 8.10, exemplifica-se a instrução ADDD no modo IND.

Endereço	Memória	Descrição	Valor em D (exemplo)	Valor em X (exemplo)	Valor em Y (exemplo)
0500	E3	ADDD – IND	2045_H	8006_H	0150_H
0501	9A	Dado a ser somado com o valor de X, - pois não houve pré-byte - para o cálculo do novo endereço	2045_H	8006_H	0150_H
		A soma em X é efetuada		$8006_H + 9A_H$	0150_H
-	-	Novo endereço: $80A0_H$	2045_H	8006_H	0150_H
80^A0	74	Dado 74	2045_H	8006_H	0150_H
-	-	A UCP automaticamente gera o próximo endereço, pois D é de 16 bits.	2045_H	8006_H	0150_H
80A1	91	Dado 91	2045_H	8006_H	0150_H
-	-	Efetuando a soma	$2045_H + 7491_H$	8006_H	0150_H
-	-	Soma completa	$94D6_H$	8006_H	0150_H

Tabela 8.9: Instrução ADDD modo indexado (IND), sem pré-byte (18_H)

Endereço	Memória	Descrição	Valor em A (exemplo)	Valor em X (exemplo)	Valor em Y (exemplo)
04FF	18	O pré-byte anterior a instrução, define que será utilizado o valor do registrador Y para o próximo cálculo, independente de qual for a próxima instrução	2045_H	8006_H	0150_H
0500	E3	ADDD – IND	2045_H	8006_H	0150_H
0501	9A	Dado a ser somado com o valor de Y, - pois houve pré-byte - para o cálculo do novo endereço	2045_H	8006_H	0150_H
-	-	A soma em Y é efetuada	2045_H	8006_H	$0150_H + 9A_H$
-	-	Novo endereço: $01EA_H$	2045_H	8006_H	0150_H
01EA	14	Dado 14	2045_H	8006_H	0150_H
-	-	A UCP automaticamente gera o próximo endereço, pois D é de 16 bits.	2045_H	8006_H	0150_H
01EB	21	Dado 21	2045_H	8006_H	0150_H
-	-	Efetuando a soma	$2045_H + 1421_H$	8006_H	0150_H
-	-	Soma completa	3466_H	8006_H	0150_H

Tabela 8.10 - Instrução ADDA Modo Indexado (IND), com pré-byte (18H)

8.3 - ESTUDOS DE PROGRAMAS COMPLETOS

Para facilitar o estudo do assembler, os códigos foram escritos com atenção especial para que utilizem somente os modos IMM e INH, os quais possuem uma estrutura de fácil compreensão.

8.3.1 - COMO SOMAR DOIS NÚMEROS ?

Solução: Para somar dois números, deve-se inicialmente carregá-los nos registradores internos do M68HC11, efetuar a instrução de soma efetivamente e logo após retornar o resultado na memória RAM.

Como todo programa, são utilizadas variáveis para o desenrolar do processamento. Este programa simples em assembler começa alocando posições de memória às variáveis, em seguida, define-se em que posição de memória o programa será armazenado e suas instruções. A última linha (FIM BRA FIM) faz com que a UCP entre em *loop* infinito. A instrução BRA (*Branch Aways*) informa à UCP que esta deverá "saltar" para o rótulo logo após a instrução BRA, no caso, FIM. Assim a UCP retorna ao rótulo FIM e novamente é conduzida a ele, entrando em um *loop* (laço) eterno. Por fim, define-se o vetor de *reset* (endereço que a UCP irá iniciar).

Programa 1 – Soma os registrador es A e B

```
;   Programa simples para demonstração do Assembler do M68HC11
;   Executa uma soma dos registradores A e B, retornando o resultado  na memória
;   Não considera o carry

            ORG $20         ; As variáveis começam a serem armazenadas
                            ; na posição de memória 0020H
R           RMB 1           ; Reserva uma posição de memória para a variável R
            ORG $B600       ; O programa terá início no endereço de memória B600
START   ; Um "rótulo" que informa que o programa está começando
SOMA    ; Um "rótulo" para referenciar um trecho de código
            LDAA #14        ; Carrega o valor 14₁₀ (0EH) no Registrador A
            LDAB #22        ; Carrega o valor 22₁₀ (16H) no Registrador B
            ABA             ; Soma os Registradores A e B e retorna o resultado em A
            STA R           ; Armazena A, (soma efetiva) na posição de memória de R
FIM         BRA FIM         ; loop infinito
            ORG $FFFE
            FDB START       ; "Leva" a UCP a começar a execução no rótulo START
```

No programa 1, não é considerado o *carry*, ou seja, o "vai-um", sendo o valor máximo de resultado limitado à FF_H (255_{10}). No exemplo 2 é descrito como corrigir este problema do *carry*, podendo ser efetuada a operação $FF_H+FF_H = 1FE_H$ ($255_{10} + 255_{10} = 510_{10}$) e obter-se o resultado válido.

8.3.2 - COMO SOMAR DOIS NÚMEROS COM *CARRY* ?

Primeiramente, observa-se que os números a serem somados são de 8 bits, e para se obter um resultado válido, é necessário que o resultado possua mais de 8 bits. A operação de soma com ou sem *carry* é realizada da mesma forma, sendo que a soma com *carry* pode resultar em um valor que ocupa 16 bits de memória, modificando, assim, a forma de armazenar o resultado.

Neste contexto, o programa armazena o valor final esperado ($1FE_H$) em duas posições consecutivas de memória (0020_H e 0021_H). Por fim, o programa 2 termina com as mesmas instruções do programa 1.

Programa 2 – Soma registradores A e B, com carry

```
;   Programa simples para demonstração assembler do M68HC11
;   Soma os registradores A e B, retornando o resultado na memória em dois endereços consecutivos
;   Considera, se houver, o carry para o correto resultado

            ORG $20         ; As variáveis começam a serem armazenadas
                            ; na posição de memória 0020H
```

```
R       RMB 1          ; Reserva uma posição de memória para a variável R
        ORG $B600      ; O programa terá início no endereço de memória B600
START   ; Um "rótulo" que informa que o programa está começando
SOMA    ; Um "rótulo" para referenciar um trecho de código
        LDAA #255      ; Carrega o valor 255₁₀ (FF_H) no Registrador A
        LDAB #255      ; Carrega o valor 255₁₀ (FF_H) no Registrador B
                       ; Estes valores são os maiores possíveis (para A e B)
        ABA            ; Soma os Registradores A e B e retorna o resultado em A
        STA R+1        ; Armazena o resultado parcial na posição de memória seguinte à
                       ; reservada para R, o valor "parcial" é o próprio valor contido em A
        LDAA #0        ; Carrega A com o valor 0 (ou "limpa" o registrador A)
        ADCA #0        ; Soma ao registrador A, o valor do carry, retornando o resultado em A
        STA R          ; Armazena na memória o valor final, o qual está em A
FIM     BRA FIM        ; loop infinito
        ORG $FFFE
        FDB START      ; "Leva" a UCP a começar a execução no rótulo START
```

A execução pode ser visualizada na tabela 8.11.

Instrução	Valor dos registradores		
	A = 00	B = 00	*Carry = 0*
LDAA #255	A = FF	B = 00	*Carry = 0*
LDAB #255	A = FF	B = FF	*Carry = 0*
ABA	A = FE	B = FF	*Carry = 1*

Tabela 8.11 - Valores dos registradores de acordo com cada instrução

Não é possível mover diretamente o valor do registrador de *carry* para a memória. Deve-se então, armazenar o valor de A na memória, "limpar" o valor de A e somar o *carry* ao A, com o objetivo de se ter o valor do *carry* em A.

A título de exemplo, a tabela 8.12 mostra passo-a-passo o que ocorre na memória RAM, após processadas as instruções que a acessam.

Variável	Endereço	Valor	Instrução
R	0020	00	
R+1	0021	FE	STA R+1
R	0020	01	STA R
R+1	0021	FE	

Tabela 8.12 - Alteração dos dados na RAM, a cada instrução STR

8.3.3 - INSTRUÇÕES E *OPCODES* UTILIZADOS NOS PROGRAMAS 1 E 2

As instruções e seus respectivos formatos descritos na tabela 8.13 são referentes ao programa 1, criadas pelo montador assembler na memória RAM.

Instrução	Descrição	*Opcode*
LDAA	M => A	86_H
LDAB	M => B	$C6_H$
ABA	A + B => A	$1B_H$
STA	A => M	97_H
BRA	?1=1	20_H

Tabela 8.13 - Descrição do formato e opcode das instruções do programa 1

Um padrão adotado para representar dados em uma memória pode ser visualizado na figura 8.4. Este padrão obedece à seguinte sintaxe: primeiramente tem-se o endereço inicial ($B600_H$). Logo após, tem-se os dados armazenados a partir deste endereço.

Desta forma, o dado 86_H está no endereço $B600_H$, o dado $0E_H$ no $B601_H$, $C6_H$ no $B602_H$ e assim por diante, conforme visto na tabela 8.14.

```
B600: 86 0E C6 16 1B 97 20 20 FE
```

Figura 8.4 -: Padrão utilizado por vários programas, ao referenciar dados na RAM.

END	B600	B601	B602	B603	B604	B605	B606	B607	B608
DADO	86	0E	C6	16	1B	97	20	20	FE

Tabela 8.14 - Endereços e dados, na RAM, para o programa 1

A seguir a descrição detalhada da execução e uma interpretação dos *opcodes* gerados pelo montador para execução dos programas 1 e 2.

O programa 1 inicia-se na posição de memória $B600_H$, com a instrução 86_H. O valor hexadecimal 86_H foi criado pelo assembler e representa a primeira instrução do programa: LDAA. Ao receber o *opcode* 86_H a UCP entende que o programa pede para mover um dado para o registrador A. Mas qual dado ?

Quando a UCP recebe o *opcode* 86_H, ela busca o dado na próxima posição de memória, ou seja, busca no endereço $B601_H$ o dado. No programa 1, o dado encontrado é $0E_H$. Na tabela 8.15, visualiza-se o exposto.

Endereço	*Opcode*	Instrução
B600	86	LDAA
B601	0E	DADO

Tabela 8.15 - Movendo o dado 0E para o registrador A

Após mover $0E_H$ para o registrador A, a UCP gera o endereço $B602_H$ e encontra a instrução $C6_H$. O *opcode* $C6_H$ representa a instrução LDAB, a qual instrui à UCP mover um dado para o registrador B. A UCP busca no endereço $B603_H$ o dado 16_H e o move para B. Na tabela 8.16, visualiza-se o exposto.

Endereço	***Opcode***	**Instrução**
B602	C6	LDAB
B603	16	DADO

Tabela 8.16 - Movendo o dado 22 para o registrador B

Repetindo o ciclo de procura da UCP, esta novamente irá gerar o endereço $B604_H$, e encontra o *opcode* $1B_H$, correspondente à instrução ABA. Ao receber o *opcode* $1B_H$ (ABA), a UCP soma o registrador A com o registrador B, retornando o resultado em A. A instrução ABA não faz nenhum acesso à memória, sendo sua atuação interna a UCP.

No próximo endereço, $B605_H$, é encontrado o *opcode* 97_H, correspondente à instrução STA. Esta instrução indica que a UCP deve armazenar o valor do registrador A em um endereço de memória. Mas qual endereço ?

A instrução STA representada por 97_H, instrui à UCP buscar o endereço que deverá gravar o valor de A, logo em seqüência ao 97_H (no caso, 20_H). Na tabela 8.17, visualiza-se o exposto.

Endereço	***Opcode***	**Instrução**
B605	97	STA
B606	20	DADO

Tabela 8.17 - Gerando o endereço de memória 0020

Ao encontrar o dado 20_H, a UCP gera o endereço 0020_H, o qual foi destinado a variável R no início do programa. Sabendo o endereço, a UCP grava o valor contido em A (24_H) no endereço 0020_H. A RAM irá conter após a execução do programa 1, a soma no endereço 0020_H, como visualizado na tabela 8.18.

Endereço	**Dado**
0020	24

Tabela 8.18- Resultado da soma, na RAM

É importante observar que o *opcode* 97_H, representando a instrução STA, assume que o byte alto do endereço como 00, podendo ser formados endereços de 0000_H a $00FF_H$.

A última instrução do programa é representada pelo *opcode* 20_H, a instrução BRA - *Branch Always* (ramifique sempre). Esta instrução é uma versão reduzida do JUMP, faz com que a execução do programa na UCP salte para um determinado endereço onde, através de um calculo, a UCP gera o endereço para o acesso à RAM. O cálculo é um pouco complexo e exige uma atenção especial para sua compreensão.

A instrução BRA, só pode referenciar até 128 endereços para trás e 127 endereços à frente do endereço corrente. A UCP soma um valor positivo de até 127 ($+7F_H$) ao endereço corrente, se for requisitado um salto à frente; soma um valor negativo de até 128 (-80_H) ao endereço corrente se for requisitado um salto para trás.

Um exemplo prático pode ser visualizado na tabela 8.19. No exemplo, a UCP encontra a instrução BRA no endereço $B600_H$ e busca o valor do salto no endereço seguinte, $B601_H$, e neste endereço encontra o valor 03_H. O valor 03_H é então somado ao valor do endereço corrente, B602, obtendo-se o valor B605, pois $B602_H + 0003_H = B605_H$.

Endereço	Dado	Descrição
B600	20	BRA
B601	03	"Salte 3 endereços"
B605	-	Endereço destino

Tabela 8.19 - Exemplo da instrução BRA (*jump* à frente)

De forma similar, a instrução BRA também pode indicar para que a execução do programa volte alguns endereços. Um exemplo prático pode ser visualizado na tabela 8.20. Neste exemplo, no endereço $B603_H$ a UCP encontra a instrução BRA e o valor FB_H no endereço seguinte, $B604_H$. Porém, o valor FB_H não representa o endereço destino do salto, sendo necessário o cálculo descrito na figura 8.5.

(1) 00FF + (-FB) = 4.
(2) B604 – 4
(3) B600

Figura 8.5 - Cálculo realizado pela instrução BRA, para um *jump* para traz.

Na figura 8.5, em (1), o valor $00FF_H$ é padrão para o cálculo e ($-FB_H$) foi a referência encontrada após a instrução BRA. Em (2), subtrai-se do endereço corrente, o valor encontrado em (1). Em (3), o valor do endereço correto para o salto.

Para voltar 3 posições de endereço para trás da instrução BRA, é gerado o número 4. Isto serve para ignorar a posição ocupada pela própria instrução BRA, e por este motivo exprime-se 128 posições para trás e 127 à frente. Na realidade tanto à frente quanto para trás da instrução BRA, referencia-se no máximo 127 posições. Os maiores saltos possíveis podem ser visualizados nas figuras 8.6 e 8.7.

ENDEREÇO	DADO	DESCRIÇÃO
B600	-	Endereço destino
B601	01	NOP
B602	01	NOP
B603	20	BRA
B604	FB	"Volte" 3 endereços, sem contar a própria instrução BRA

Tabela 8.20 - Exemplo da instrução BRA (*jump* à trás)

B600	**20**	**7F**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B610	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B620	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B640	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B650	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B650	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B660	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B660	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B670	**01**	**86**	**0E**	**C6**	**16**	**97**	**20**	**8C**	**6C**	**DF**	**52**	**5A**	**59**	**8D**	**3D**	**0E**

Figura 8.6 - BRA para frente (maior salto possível à frente – 127 endereços)

B600	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B610	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B620	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B630	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B640	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B650	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B660	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**
B670	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**20**	**80**

Figura 8.7 - BRA para trás (maior salto possível à trás – "128" endereços)

A função da instrução BRA no programa 1 pode ser facilmente compreendida. A tabela 8.21 e a figura 8.8 mostram o cálculo realizado.

Endereço	*Opcode*	Instrução
B607	20	BRA
B608	FE	DADO

Tabela 8.21 - Instrução BRA no programa 1

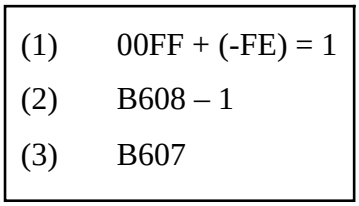


Figura 8.8 - Cálculo realizado para execução de BRA, no programa 1.

Consegue-se então um loop (laço), onde a UCP é conduzida ao endereço $B607_H$, que novamente executa o BRA e retorna no endereço $B607_H$, infinitamente.

Em resumo, a instrução BRA faz a UCP monitorar o primeiro bit do dado seguinte a instrução: sendo "0", (de 00000000_B a 01111111_B) soma-se o dado ao endereço corrente; sendo "1", (de 11111110_B a 10000000_B) subtrai-se o dado do endereço corrente.

8.3.4 - COMO MULTIPLICAR DOIS ALGARISMOS DE 16 BITS ?

Solução: O programa deve utilizar duas posições de memória para cada operando. No programa 3, são multiplicados os valores 1234_{10} e 5000_{10}.

Programa 3 (Parte 1)

```
            ORG $0
RESULT      RMB     4
OPERAND1    RMB     2
OPERAND2    RMB     2
            ORG$B600
START
MUL_16_16   LDD     #1234       ; Carrega o valor 04D2H (1234₁₀) no registrador
                                ; D, 04H, no registrador A e D2H no registrador B
            STD OPERAND1        ; Armazena o valor registrador  em D (04D2H) na
                                ; memória: 04H na posição 0004 e D2H na posição 0005
            LDD     #5000       ; Carrega o valor 1388H (5000₁₀) no registrador D, 13H no
                                ; registrador A e 88H (136₁₀) no registrador B
            STD OPERAND2        ; Armazena o valor registrado em D (1388H) na memória:
                                ; A=>M, B=>M+1, ou seja, 13H em uma posição da RAM
                                ; e 88H na próxima
```

Os valores hexadecimais produzidos pelo montador assembler, referente ao trecho de código do programa 3 (parte 1) são visualizadas na figura 8.9.

```
B600: CC 04 D2 DD 04 CC 13 88 DD 06
```

Figura 8.9 - Valores hexadecimais, referente ao programa 3

Como já visto anteriormente, o *opcode* CC representa a instrução LDD no modo IMM, ou seja, a UCP transfere os valores consecutivos à CC_H (04_H $D2_H$) aos registradores A e B.

O valor DD_H, representa a instrução STD no modo IMM, informando que a UCP deve gravar o valor contido em D (A e B) na posição de memória 0004_H. Cabe aqui lembrar que a parte alta do endereço é assumida como 00_H. O valor 04_H foi gerado automaticamente pelo montador assembler, pois as quatro primeiras posições da memória foram reservadas para a variável RESULT.

O par LDD/STD se faz necessário, pois não há uma instrução que permita mover valores de um endereço da RAM, diretamente a outro.

Continuando a execução, a UCP encontra novo CC_H e grava o valor 1388_H no registrador D e nos próximos endereços encontra DD_H 06_H, grava o valor de D na posição de memória 0006_H.

Após esta execução parcial, a área de dados da memória RAM, localizada no endereço 0000_H, como reservado no início do programa (instrução ORG $0), recebe os primeiros dados correspondentes aos dois operandos da multiplicação, destacados na figura 8.10.

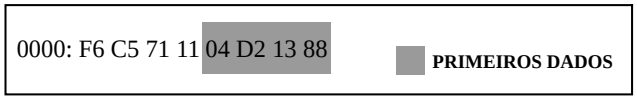


Figura 8.10: Primeiros dados a serem processados no programa 3.

Observa-se que os quatro primeiros valores, ou endereços, são irrelevantes por enquanto, pois ainda não foram acessados.

Retornando ao estudo do código, deve-se multiplicar a parte baixa dos dois operandos, sendo a primeira operação visualizada na figura 8.11.

Programa 3 (Parte 2)

```
LDAA OPERAND1+1          ; Low x Low
LDAB OPERAND2+1
MUL
STD RESULT+2
```

	0 4	D 2
X	1 3	8 8
	6 F	9 0

Figura 8.11: Primeira operação do programa 3

Na memória RAM, o primeiro valor parcial do resultado aparece, na região da memória RAM, reservada para RESULT, visualizado em destaque na figura 8.12.

0000: F6 C5 6F 90 04 D2 13 88 — 1º RESULTADO PARCIAL

Figura 8.12 - Primeiro resultado parcial do programa 3

Agora o programa multiplica a parte baixa do operando 1 pela parte alta do operando 2, sendo a segunda operação visualizada na figura 8.13.

Programa 3 – (Parte 3)

```
LDAA OPERAND1+1          ; Low x High
LDAB OPERAND2
MUL
ADDB RESULT+2
ADDB RESULT+2
STD RESULT+1
```

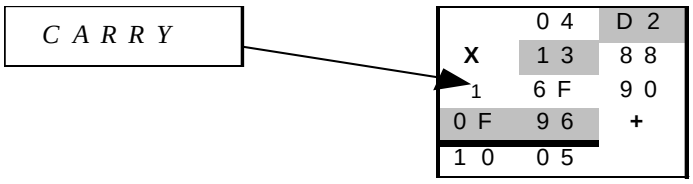


Figura 8.13 - Segunda operação do programa 3

Na memória RAM, o resultado é novamente atualizado, na região reservada para RESULT, visualizado em destaque na figura 8.14.

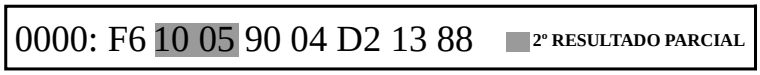


Figura 8.14 - Segundo resultado parcial do programa 3.

Multiplica-se a parte alta do operando 1 pela parte baixa do operando 2, sendo a terceira operação visualizada na figura 8.15.

Programa 3 – (Parte 4)

```
LDAA OPERAND1          ; High x Low
LDAB OPERAND2 + 1
MUL
ADDB RESULT+2
ADCA RESULT+1
STD RESULT+1
```

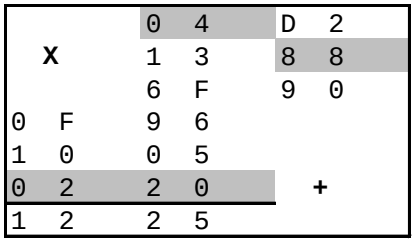


Figura 8.15 - Terceira operação do programa 3

Na memória RAM, o resultado é novamente atualizado, na região reservada para RESULT, em destaque na figura 8.16.

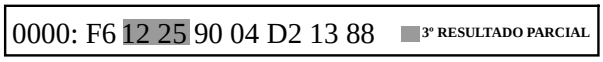


Figura 8.16 - Terceiro resultado parcial do programa 3.

Por fim, o último trecho de código do programa 3 multiplica as partes altas dos operandos, sendo a operação final visualizado na figura 8.17.

Programa 3 – (Parte 5)

```
LDAA OPERAND1          ; High x High
LDAB OPERAND2
MUL
ADDB RESULT+1
ADCA #0
STD RESULT
```

				0	4	D	2
			X	1	3	8	8
				6	F	9	0
		0	F	9	6		
		1	0	0	5		
		0	2	2	0		
		1	2	2	5		
0	0	4	C	+			
0	0	5	E				

Figura 8.17 - Última operação realizada pelo programa 3

Na figura 8.18, visualiza-se toda a seqüência de operações e seus respectivos valores intermediários, realizadas pelo programa 3, e na figura 8.19, o valor final da multiplicação de $04D2_H$ X 1388_H, em destaque.

				0	4	D	2
			X	1	3	8	8
				6	F	9	0
		0	F	9	6		
		1	0	0	5		
		0	2	2	0		
		1	2	2	5		
0	0	4	C				
0	0	5	E				
Valor Final		0	0	5	E	2	5	9	0

Figura 8.18 - Seqüência realizada pelo programa 3, para realizar a multiplicação

```
0000: 00 5E 25 90 04 D2 13 88
```
RESULTADO FINAL

Figura 8.19 - Resultado final da multiplicação de 1388 X 04D2.

Na figura 8.20, pode-se visualizar como ficam os dados na memória RAM, após sua execução.

0000	**00**	**5E**	**25**	**90**	**04**	**D2**	**13**	**88**	**D9**	**A1**	**44**	**AF**	**35**	**5F**	**E5**	**E6**
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
B600	**CC**	**04**	**D2**	**DD**	**04**	**CC**	**13**	**88**	**DD**	**06**	**96**	**05**	**D6**	**07**	**3D**	**DD**
B610	**02**	**96**	**05**	**D6**	**06**	**3D**	**DB**	**02**	**89**	**00**	**DD**	**01**	**96**	**04**	**D6**	**07**
B620	**3D**	**DB**	**02**	**99**	**01**	**DD**	**01**	**96**	**04**	**D6**	**06**	**3D**	**DB**	**01**	**89**	**00**
B630	**DD**	**00**	**20**	**FE**	-	-	-	-	-	-	-	-	-	-	-	-

Figura 8.20 - Disposição dos dados na memória RAM, após a execução do programa 3.

Multiplicação de 16 bits X 16 bits (completo)

```
                ORG $0
RESULT          RMB     4
OPERAND1        RMB     2
OPERAND2        RMB     2
                RMB     2
                ORG$B600
START
MUL_16_16       LDD     #1234       ; Carrega o valor 04D2H (1234₁₀) no registrador
                                    ; D, 04H, no registrador A e D2H no registrador B
                STD OPERAND1        ; Armazena o valor registrador  em D (04D2H) na
                                    ; memória: 04H  na posição 0004 e D2H na posição 0005
                LDD     #5000       ; Carrega o valor 1388H (5000₁₀) no registrador D, 13H no
                                    ; registrador A e 88H (136₁₀) no registrador B
                STD OPERAND2        ; Armazena o valor registrado em D (1388H) na memória:
                                    ; A=>M, B=>M+1, ou seja, 13H em uma posição da RAM
                                    ; e 88H na próxima
                LDAA OPERAND1+1     ; Low x Low
                LDAB OPERAND2+1
                MUL
                STD RESULT+2
                LDAA OPERAND1+1     ; Low x High
                LDAB OPERAND2
                MUL
                ADDB RESULT+2
                ADDB RESULT+2
                STD RESULT+1
                LDAA OPERAND1       ; High x Low
                LDAB OPERAND2 + 1
                MUL
                ADDB RESULT+2
                ADCA RESULT+1
                STD RESULT+1
                LDAA OPERAND1       ; High x High
                LDAB OPERAND2
                MUL
                ADDB RESULT+1
                ADCA #0
                STD RESULT
FIM             BRA FIM             ; loop infinito
                ORG $FFFE
                FDB START           ; "Leva" a UCP a executar no rótulo START
```

8.4 – O µHC11

O HC11CORE é uma descrição VHDL do microcontrolador M68HC11, escrita por Scott Thibault. Esta descrição demostra como a linguagem VHDL pode ser utilizada para um projeto de um microcontrolador. Neste livro é descrita uma versão simplificada do HC11CORE, entitulada µHC11, com intuito de demostrar de forma simplificada o funcionamento e arquitetura básica do microcontrolador M68HC11.

A implementação do µHC11 respeita as limitações de tamanho e impossibilidade da implementação física no FPGA de dois periféricos:

- memória EPROM – dispositivos FPGA não possuem a capacidade de retenção de dados caso sua alimentação seja desligada, característica imprescindível de uma memória EPROM.
- conversor A/D – todo conversor A/D é dotado de um comparador de tensão para determinar o nível de tensão em sua entrada e, logo após representá-lo sob a forma de bits em sua saída. Os FPGAs não possuem estes comparadores de tensão, impossibilitando mapeá-los com um código de um A/D.

Existem algumas implementações de conversores A/D e D/A em FPGAs, porém, além de ser um código VHDL extenso, o que consome grande parte dos recursos do FPGA, todas essas descrições utilizam comparadores de tensão.

A descrição VHDL do µHC11 contém 16 instruções nos seis modos de endereçamento que o M68HC11 é capaz de realizar. Também oferece parte do periférico de comunicação serial, onde foi implementado somente a transmissão (TxD).

O capítulo focaliza principalmente os problemas detectados no seu desenvolvimento e na experiência adquirida. Além disso, mostra-se como é possível realizar sistemas complexos, usando-se da integração de conceitos de hardware (VHDL, FPGAs e eletrônica) e software (interface Delphi e programação geral).

O µHC11 é apresentado seguindo uma escala de evolução que parte de uma versão básica totalmente simulada em software (VHDLStudio) evoluindo até a versão final implementada em FPGA, onde se tem uma interface entre o µHC11 e o PC.

8.4.1 – PRIMEIRA VERSÃO DO µHC11

A primeira versão do µHC11 desenvolvida no software VHDLStudio é basicamente composta por dois módulos distintos :

- uma memória ROM, que contém o programa a ser executado pela UCP e;
- a UCP µHC11 em si.

Nesta versão inicial, ainda que totalmente implementada, depurada e validada em software, não há nenhuma gravação de dados na memória, pois esta é uma memória ROM (somente leitura). A movimentação dos dados é feita somente da memória ROM para os registradores A e B e entre eles próprios. A arquitetura do µHC11, primeira versão, pode ser visualizada na figura 8.21.

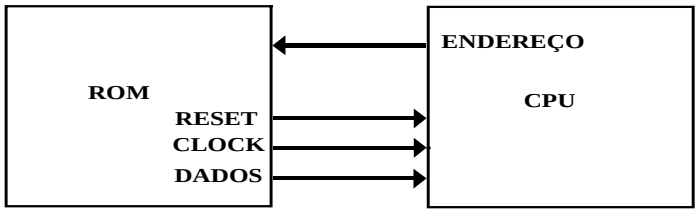


Figura 8.21 - Arquitetura do µHC11, primeira versão

O código VHDL, o qual implementa a ROM de 60 x 8 bits, não possui entidade (*test bench*) e envia dados à UCP através da porta dados, de acordo com o endereço recebido. Também é responsável por enviar o estímulo de *clock* a cada 50ns à UCP, bem como um sinal de *reset*.

A UCP simplificada implementada tem seu conjunto de instruções sumariado na tabela 8.22, o qual é bem menor que o conjunto de instruções implementados no HC11CORE.

No HC11CORE, existem dois processos distintos que perfazem o processamento dos endereços e operações de ULA. Estes processos executados em paralelo diminuem o tempo necessário à execução da instrução. A UCP do µHC11 possui uma união entre a Unidade de Controle e ULA, realizando as funções de endereçamento e operações de ULA em um único processo. Diferente da UCP do HC11CORE, esta metodologia de união torna a compreensão do código mais fácil, porém diminui o paralelismo, o que torna a UCP mais lenta. Como o livro visa somente

demonstrar um CORE que execute algumas instruções do M68HC11, respeitando o formato original, dados temporais de execução foram desprezados.

No software de simulação VHDLStudio, a depuração da execução de cada instrução é uma tarefa muito fácil, onde pode-se visualizar os valores dos registradores, o valor do PC, o *opcode* corrente, entre outros, e também o tempo decorrido entre a compilação do código e a simulação é bem pequeno (cerca de 10 segundos, em um Pentium II 400Mhz, para o código do µHC11), sendo erros de lógica facilmente encontrados e corrigidos.

Nesta versão, validada em software, foi implementado um programa na memória ROM, escrito especificamente para testar todas as instruções descritas na tabela 8.22, em todos os modos de endereçamento.

Instrução	**Descrição**	**MODO**	***Opcode***
ABA	A + B => A	INH	1B
LDAA	M => A	IMM	86
		DIR	96
		EXT	B6
		IND X	A6
		IND Y	18, A6
LDAB	M => B	IMM	C6
		DIR	D6
		EXT	F6
		IND X	E6
		IND Y	18, E6
LDD	M => A, M +1 => B	IMM	CC
		DIR	DC
		EXT	FC
		IND X	EC
		IND Y	18, EC
ANDA	A and M => B	IMM	86
		DIR	96
		EXT	B6
		IND X	A6
		IND Y	18, A6
ANDB	B and M => B	IMM	86
		DIR	96
		EXT	B6
		IND X	A6
		IND Y	18, A6
JUMP	M, M + 1 => PC	EXT	7E

Tabela 8.22 - Instruções implementadas e validadas em µHC11 V0.1

Como o intuito principal é implementar e testar o código *fisicamente* em um FPGA, é necessário converter o projeto a outra ferramenta CAD de desenvolvimento

de hardware que permita a compilação do código e seu correto mapeamento em um FPGA. A ferramenta CAD utilizada e o FPGA foram, respectivamente, Xilinx Foundation e XC4010XL.

8.4.2 - SEGUNDA VERSÃO DO µHC11

Nesta versão, o µHC11 está descrito na ferramenta CAD, Xilinx Foundation, e adaptado à placa de prototipação XS40, da XESS, a qual contém um FPGA XC4010XL. O único *display* de 7 segmentos disponível é utilizado para depuração das operações realizados pelo µHC11.

Esta conversão, em um primeiro momento, parece ser uma tarefa simples, bastando adicionar os códigos obtidos na versão anterior, à ferramenta CAD da Xilinx, levando-se em conta que a linguagem VHDL é padronizada. A incompatibilidade entre os softwares de síntese torna necessário a modificação de trechos de código VHDL preservando a integridade e funcionalidade lógica do µHC11. A arquitetura desta segunda versão do µHC11, pode ser visualizada na figura 8.22.

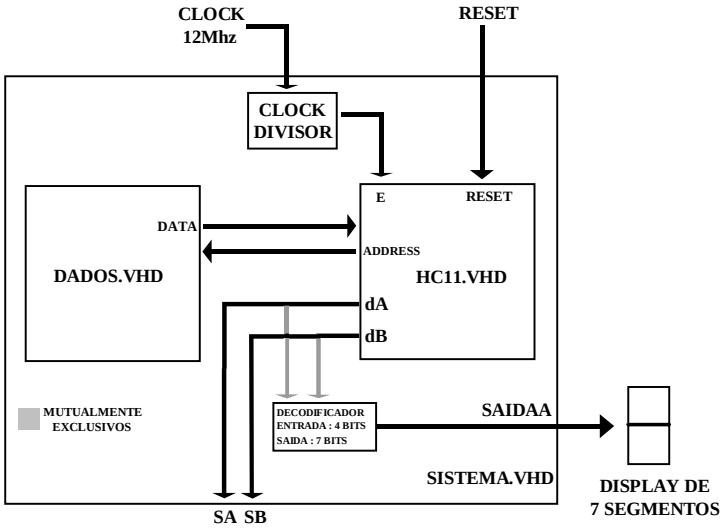


Figura 8.22 - Arquitetura do µHC11 V0.2

O µHC11 versão 0.2, é composto por três módulos VHDL distintos:

- **dados.vhd**: Este código VHDL representa uma ROM, de 60 x 8 bits, que contém os dados a serem entregues à UCP. Seu propósito é bem parecido com o VHDL da ROM implementada no HC11CORE, mas com uma grande diferença: o código da ROM implementada na primeira versão, tinha a função de entregar dados, gerar o *clock* base e *reset* à UCP. No VHDLStudio gerar o *clock* e o *reset* para a UCP é uma tarefa fácil, através dos comandos *after* e *wait for*. Estes comandos tem restrições de uso na ferramenta da Xilinx, pois o software de síntese não dá suporte a implementação desses comandos. Neste contexto, o módulo **dados.vhd** possui nesta segunda versão, a única função de enviar dados à UCP, de acordo com o endereço que esta lhe requisita.

- **Hc11.vhd**: Este código VHDL descreve a UCP inicial já adaptada à plataforma Xilinx.

- **sistema.vhd**: A função principal de sistema.vhd é oferecer o *debug* indispensável à validação do código, unindo Hc11.vhd e dados.vhd, monitora os valores correntes no registrador A e B, sendo o valor decodificado e mostrado no *display* de 7 segmentos disponível na placa de prototipação. Possui também a função de reduzir o *clock* de 12Mhz, disponível na placa, para um *clock* de 1Hz, resultando em um ciclo de execução por segundo, facilitando a depuração real dos resultados.

O conjunto de instruções implementado nesta versão é reduzido, pois as instruções LDD, ANDA e ANDB foram retiradas, dado o espaço ocupado no FPGA. Lembrando que agora os testes são físicos e limitados à capacidade do FPGA utilizado, ao contrário da implementação simulada por software (VHDLStudio);

Nesta versão do µHC11, a maior dificuldade está no *debug* de funcionamento do código VHDL, pois há somente um único *display* de LED disponível na placa de prototipação. Através de um decodificador, é possível a visualização do meio byte menos significativo e meio byte mais significativo dos registradores A e B. Este método de *debug*, apesar de ser dispendioso, comprova o correto funcionamento das instruções implementadas.

A seqüência de *debug* utilizada pode ser visualizada na figura 8.23.

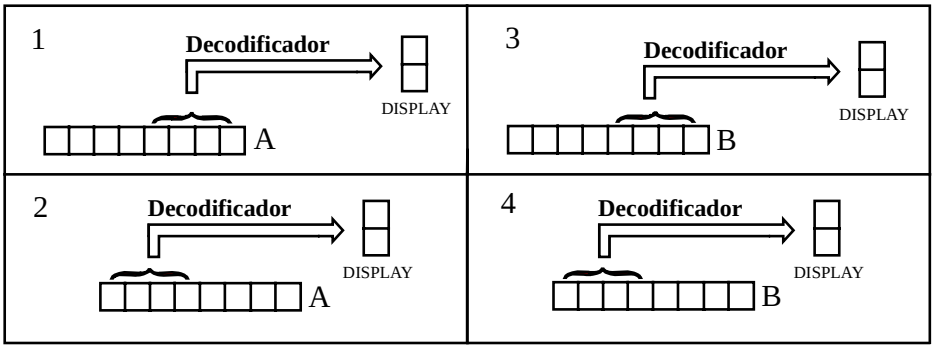


Figura 8.23 - Seqüência e método de *debug*, nas versões de 0.2 do µHC11

O µHC11 versão 0.2 utiliza cerca de 97% dos recursos disponível do FPGA utilizado, o que impossibilitava o desenvolvimento de novas instruções. Uma solução é retirar do código a memória ROM interna ao FPGA e substituí-la por uma memória RAM externa ao FPGA, disponível na placa de prototipação.

Com esta substituição, o FPGA tem sua capacidade de utilização ampliada em cerca de 40%, caindo de 97 para 57% de utilização de seus recursos. Nas seções seguintes são discutidos as características da memória RAM disponível na placa de prototipação da Xess e como utilizá-la no projeto.

8.5 - UTILIZAÇÃO DA MEMÓRIA RAM DISPONÍVEL NA PLACA DE PROTOTIPAÇÃO XS 40, DA XESS.

As placas de prototipação da empresa Xess dispõe de uma memória RAM que podem ser de 32K x 8 bits ou 128K x 8 bits. A placa de prototipação utilizada para implementação do µHC11 versão 0.2 possui uma memória de 32K x 8 bits, sendo incompatíveis com os interesses do projeto, pois o M68HC11 original da Motorola é capaz de endereçar uma memória externa de 64K x 8.

Uma solução alternativa é substituição da memória de 32k x 8 por uma de 64k x 8. Na prática, a diferença entre o chip de 32K x 8 disponível na placa de prototipação e o chip necessário é somente um pino de endereçamento, pois 15 bits endereçam 32K e 16 bits endereçam 64K.

Na figura 8.24 podem ser visualizadas, em (B) e (C), as opções de chips de memórias disponíveis nas placas da XESS, de 32K x 8 e 128K x 8, respectivamente.
Em (A), memória utilizada no projeto µHC11. A figura 8.24 ilustra também os encapsulamentos e as pinagens dos chips de memória RAM, segundo os datasheets do fabricante de memórias Winbond.

8.5.1 - O CHIP DE MEMÓRIA RAM DE 64K X 8 E À PLACA XS40

Adaptar o chip de 64k x 8 no lugar do original de 32K x 8 parece, em um primeiro momento, uma tarefa difícil, pois os chips tem tamanhos e pinagens fisicamente diferentes, conforme visualizado na figura 8.25. Na figura 8.26 visualiza-se, em (a) o chip de 64K x 8 e em (b) o chip de 32K x 8 percebe-se que os chips tem quase as mesmas funções entre seus pinos. A seguir, as alterações físicas necessárias para adequar a memória de 64K à placa de prototipação da Xess.

No chip original da placa, 32K, o pino 28 corresponde à VDD (+5V), e no chip a ser adaptado de 64K o pino nesta posição é CS2, cuja função é habilitar o segundo banco de memória do chip, já que este é formado por dois bancos de 32K. Para habilitar o segundo banco, este pino, CS2, deve receber alimentação positiva (+5V), sendo disponibilizada para alimentar o chip original, o que pode ser visualizado à esquerda da figura 8.27 onde, em (A), é representada a memória RAM utilizada no projeto, de 64K x 8, e em (B) a memória RAM original da placa.

Com 5V em CS2, para habilitar o segundo banco de memória do chip, o primeiro problema está solucionado, faltando alimentar o VDD do chip de 64K para que este funcione corretamente. Com uma ligação entre o pino 30 e 32, a alimentação do chip foi solucionada, como pode ser visualizado à direita da figura 8.27. A figura 8.28 ilustra a modificação física real.

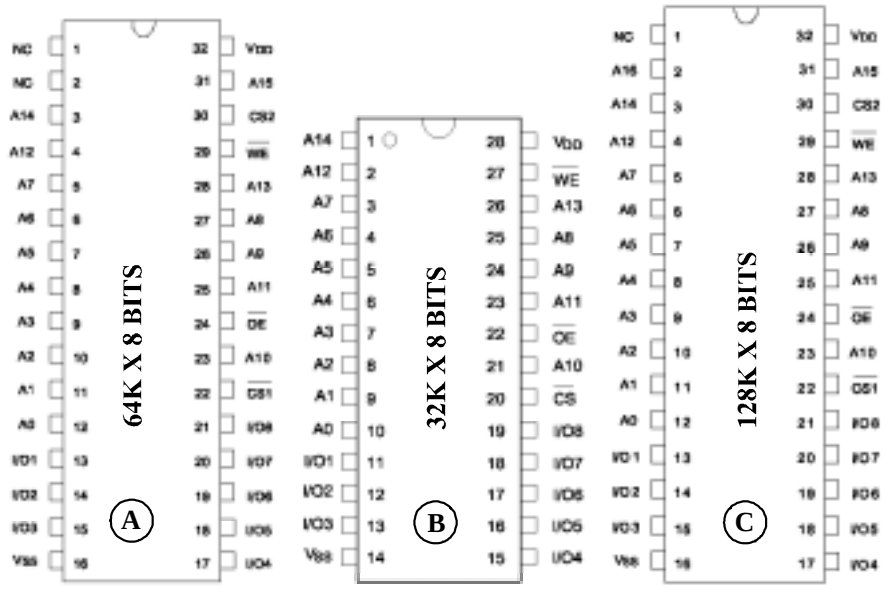


Figura 8.24 - Chips de memória RAM estudados no projeto

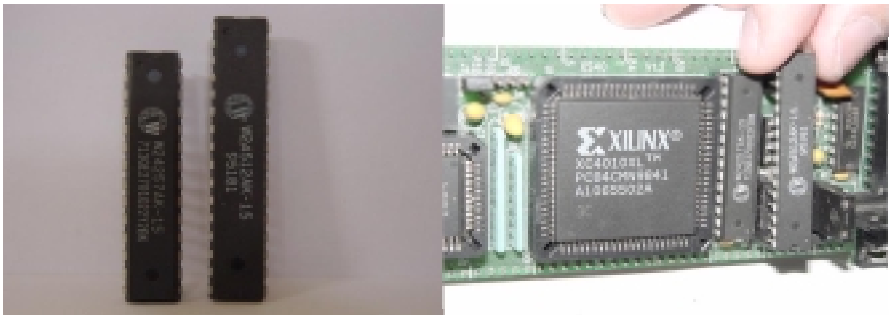


Figura 8.25 - Tamanhos fisicamente diferentes

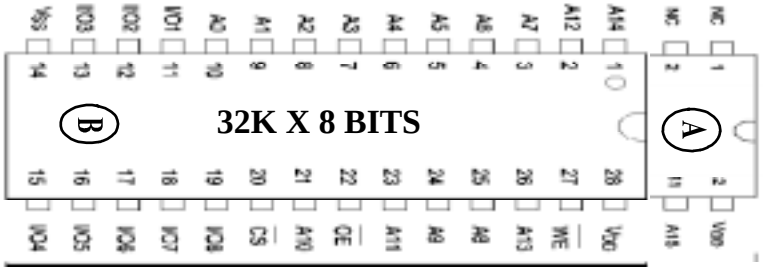


Figura 8.26 - Compatibilidade de funções quase total entre os chips de 32K e 64K.

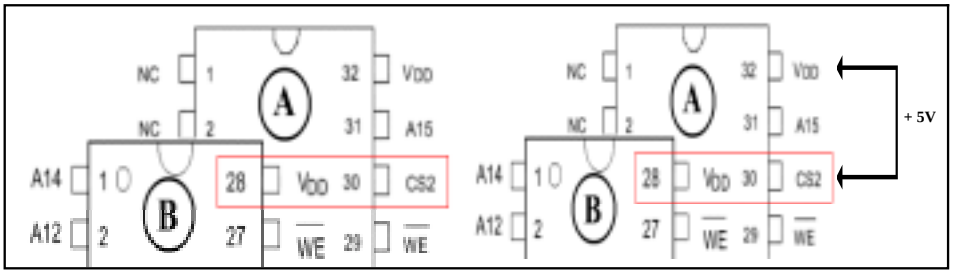


Figura 8.27 - À esquerda, pinos 28 e 30 recebem +5V; À direita, interligação entre os pinos 30 e 32 para alimentar o chip de 64K x 8.

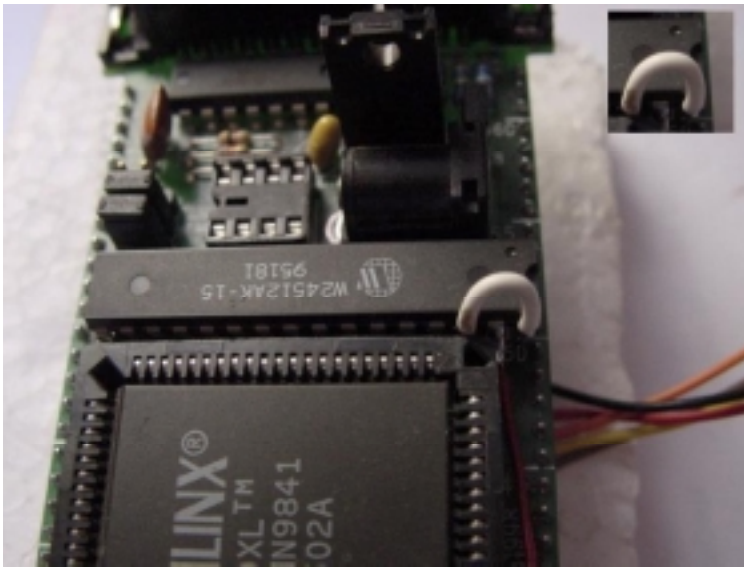


Figura 8.28 - No detalhe, ligação física necessária para alimentar o chip de 64K

Por fim, deve-se interligar o pino 31, ou pino de endereçamento A15, ao pino 28 do FPGA, sendo esta última alteração feita na placa de prototipação, onde pode ser visualizada na figura 8.29.

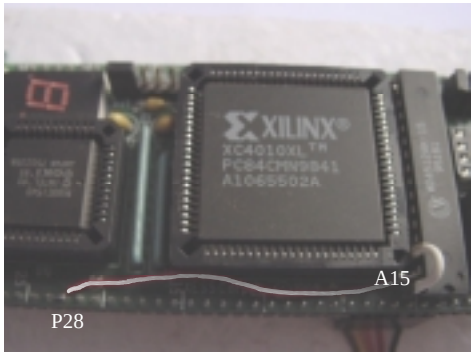


Figura 8.29 - Esquema da placa de testes e sua respectiva alteração

A placa de prototipação XS40 com sua memória original de 32K x 8 e mesma placa com a memória adaptada de 64K x 8, pode ser visualizada figura 8.30.

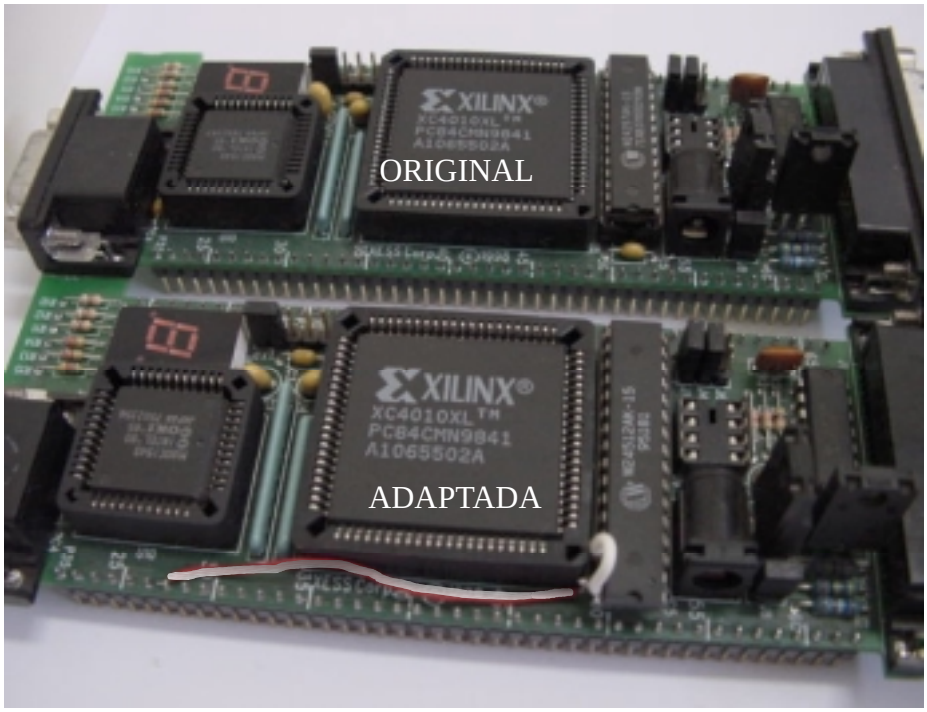


Figura 8.30 - Placa de prototipação utilizada

Somente ler os valores da RAM, sem conhecê-los, de nada adianta. Para ter controle sobre os dados enviados à memória, é necessário conhecer os padrões e os formatos de arquivos com a extensão “hex”.

O método utilizado para gravação na RAM externa pode ser visualizado na figura 8.31. Neste formato, basta iniciar cada linha com “–10“ seguido do endereço desejado e dezesseis valores hexadecimais. O valor digitado do endereço inicial ou final é aleatório, bastando apenas tomar o cuidado de se manter a distância de 10_H entre os valores de endereçamento, quando há mais de uma linha de texto.

PADRÃO	ENDEREÇO (n BITS)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
- 10	XXXX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
- 10	X+10_H	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX

EXEMPLO

PADRÃO	ENDEREÇO (n BITS)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
- 10	B600	10	25	A0	40	83	64	F1	A5	C1	D9	83	71	12	10	10	30
- 10	B610	15	20	11	45	44	29	B0	22	34	55	80	71	47	12	22	0A

Figura 8.31 - Formato HEX

Adequar o novo chip de memória à placa, sem testes funcionais que comprovam seu correto endereçamento, de nada adianta. Sendo assim, após realizar as alterações eletrônicas, é necessário alguns testes, com códigos VHDL e arquivos do formato .hex escritos especificamente para validar a nova memória de 64K x 8.

O código VHDL desenvolvido para comprovar o funcionamento da RAM de 64K x 8 tem a função de gerar endereços com valores entre 0000_H e $FFFF_H$, ou seja, toda a capacidade de endereçamento do µHC11. O valor do endereço é incrementado e entregue a RAM, à uma velocidade de aproximadamente 1714 vezes por segundo.

Simultaneamente, a parte inferior do dado gravado na RAM é decodificada e mostrada no *display* de 7 segmentos da placa. Desta forma, é possível ler todo o conteúdo da RAM em aproximadamente 40 segundos. A arquitetura para validar a nova memória pode ser visualizada na figura 8.32.

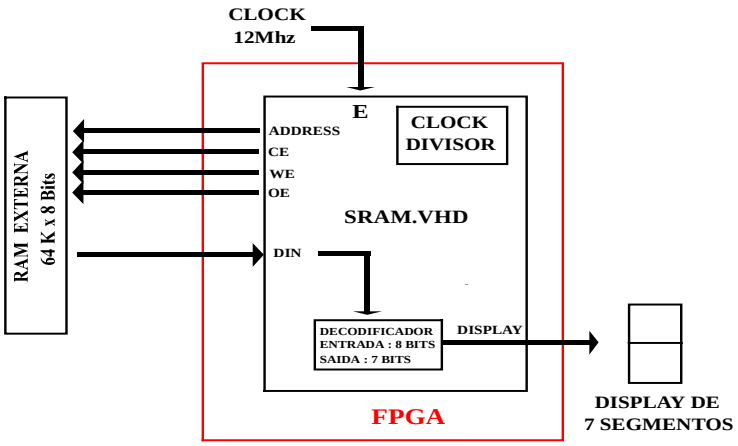


Figura 8.32 - Arquitetura para validar a memória externa, sram.vhd.

O arquivo hex criado contém dados conhecidos que serão armazenados em toda a extensão da memória RAM. O arquivo gerado é extenso. Na tabela 8.23, tem-se a visualização simplificada deste, onde do endereço 0000 a 7FF0 está armazenado o valor 0A e do endereço 8000 a FFFF o valor 0B.

Arquivo Hex (teste RAM)
-10 0000 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A
-10 0010 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A
-10 0020 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A
-10 0030 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A
.
.
.
-10 7FF0 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A
-10 8000 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
-10 8010 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
-10 8020 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
-10 8030 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
-10 8040 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
.
.
-10 FFF0 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B

Tabela 8.23 - Arquivo HEX, gerado para validação da RAM de 64K

Mapeando o FPGA com o arquivo de teste da memória RAM poderá ser visualizado no *display*, durante aproximadamente 20 segundos, a letra “A” e, por mais 20 segundos, a letra “B”. Na seção seguinte, tem-se a versão final do µHC11.

8.6 – TERCEIRA E ÚLTIMA VERSÃO DO µHC11

Esta última versão do µHC11 dispõe de um método de *debug* através da comunicação serial com o PC, onde este realiza uma depuração completa e altamente confiável, para se verificar os valores correntes nos registradores implementados no µHC11.

O teste inicial de comunicação serial utiliza um acessório do Windows, o programa Hyper Terminal, que permite monitorar e gerenciar conexões seriais. Antes da comunicação ser efetivamente estabelecida, é necessário configurá-lo para que receba os bits na ordem e velocidade que forem enviados pelo transmissor. O programa Hyper Terminal e sua interface de configuração pode ser visualizado na figura 8.33.

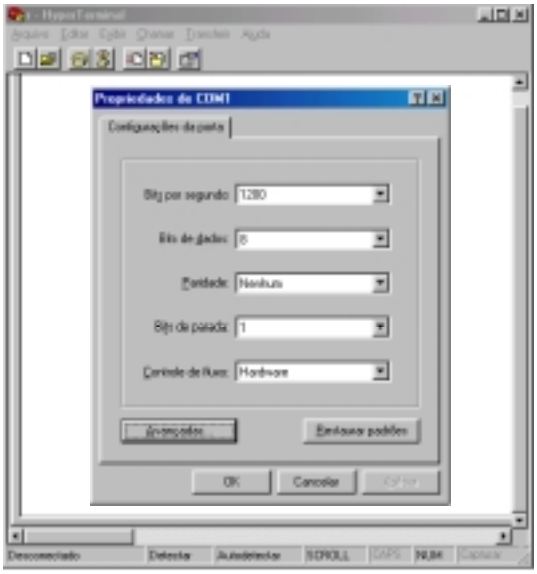


Figura 8.33- Programa Hyper Terminal e respectiva interface de configuração.

A comunicação serial ocorre fundamentalmente entre dois agentes: (a) um transmissor e (b) um receptor, como pode ser visualizado na figura 8.34. Antes de iniciar a transmissão, transmissor e receptor devem ser ajustados para que utilizem o mesmo formato e a mesma velocidade de transmissão de dados, com um único meio físico de comunicação.

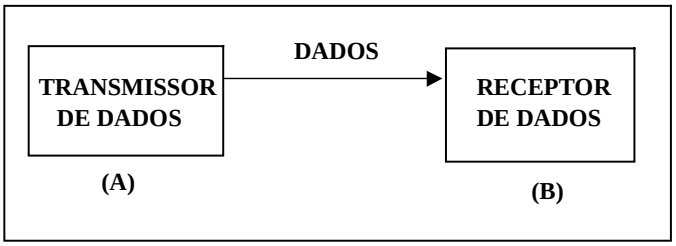


Figura 8.34 - Componentes básicos da comunicação serial

A comunicação é concluída após o receptor de dados receber uma seqüência de bits, previamente estabelecida entre os dois componentes. O transmissor e receptor devem definir a taxa de transferência de bits por segundo e o tamanho da palavra a ser transmitida (quantidade de bits). Os padrões de transmissões seriais são compostos por:

- uma taxa de transferência de bits por segundo, conhecida por todos os comunicantes;
- um bit inicial (*start bit*), que indica que uma palavra será transmitida;
- um bit de parada (*stop bit*), que indica que o envio da palavra terminou;
- opcionalmente, um bit de verificação de todos os dados recebidos, que pode ser paridade par ou ímpar.

O padrão sem bit de verificação pode ser visualizado na figura 8.35, e na figura 8.36, o padrão com bit de verificação.

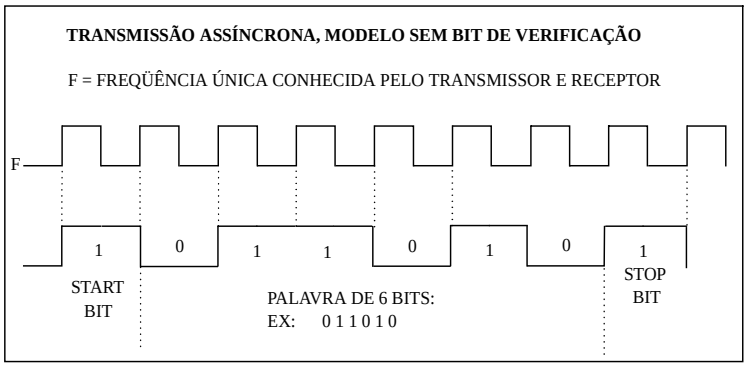


Figura 8.35 - Padrão para transmissão serial, sem bit de verificação de erro.

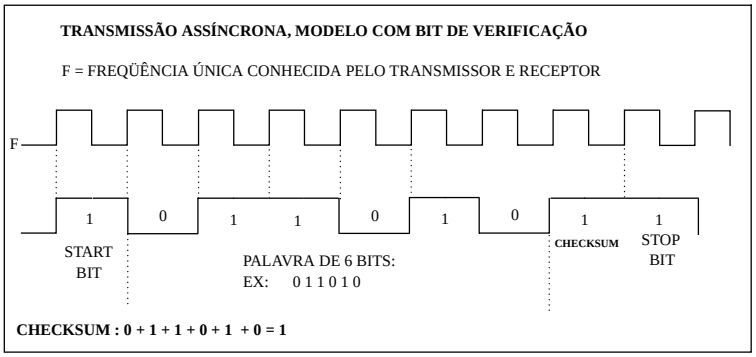


Figura 8.36 - Padrão para transmissão serial, com bit de verificação de erros

O padrão descrito na figura 8.36, com bit de verificação, é adotado para transmissões seriais com base em altas taxas de transferência de bits por segundo. No projeto, foi adotado o modelo sem bit de verificação e uma taxa de transmissão de 300 bits por segundo, considerada baixa, com palavra de tamanho de 8 bits.

8.6.1 PRIMEIRO TESTE DE COMUNICAÇÃO SERIAL

Para o primeiro teste de uma transmissão serial básica entre o programa Hyper Terminal e FPGA, tem-se um código VHDL que envia a letra “A” para o Hyper Terminal. Este código VHDL envia uma seqüência de bits contendo o bit de *start*, a seqüência de bits que corresponde a letra “A” da tabela ASCII e o bit de *stop*. O dado em si foi definido como sendo de 8 bits, a taxa de transferencia de 300 Bps e a porta de comunicação no PC, a COM1, como visualizado na figura 8.37.

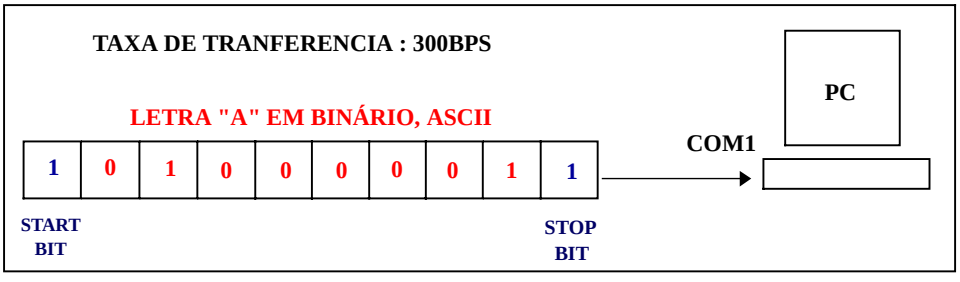


Figura 8.37 - Primeiro teste de comunicação serial, entre FPGA e o PC.

A seguir o código VHDL para o primeiro teste de comunicação serial.

Teste simples de comunicação serial

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rs232 is
   port ( e : in std_logic;
        rst : in std_logic;
        s_dado : out std_logic);
end rs232;
architecture rs232_arch of rs232 is                --i87654321f
constant dado : std_logic_vector(9 downto 0):="1101111101";
                                                   -- i= inicial (start bit)
                                                   -- f= final (stop bit)
signal s_e  : std_logic;
signal conta : integer range 0 to 40000;
signal contb : integer;
begin
--- redução do clock de 12mhz ---
process (e)
  begin
   if e'event and e='1' then
            conta <= conta + 1;
     if conta <   20000 then
        s_e <= '0';
      else
        s_e <= '1';
      end if;
      if conta =  40000 then
        conta <= 0;
      end if;
   end if;
  end process;
--- envio serial em si ---
process (s_e,rst)
  begin
    if s_e'event and s_e='0' then
      contb <= contb +1;
      if contb < 10 then
              s_dado<=dado(contb);
      end if;
    end if;
  end process;
end rs232_arch;
```

Para conclusão do primeiro teste, é necessário a confecção de um cabo, o qual interliga o kit de prototipação e a porta COM1 do PC. Basicamente, o cabo é composto por um conector compatível com os pinos da placa de teste em uma ponta e um conector fêmea DB9. Detalhes de confecção do cabo podem ser visualizados na figura 8.38.

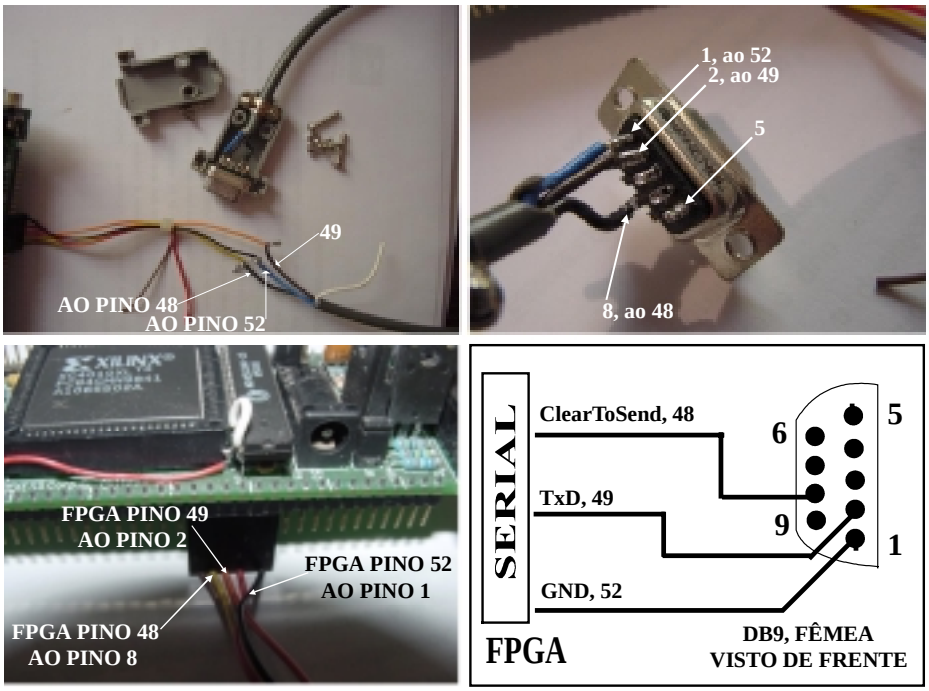


Figura 8.38 - Interligação do cabo confeccionado à placa de testes

8.6.2 - SEGUNDO TESTE DE COMUNICAÇÃO SERIAL

O segundo teste implementa um código VHDL, destinado a transmitir toda tabela ASCII, ou seja, valores binários entre 00000000 e 11111111, mantendo a taxa de transmissão fixa em 300 Bps, utilizando-se o mesmo cabo do primeiro teste de comunicação serial e o programa Hyper Terminal.

Segundo teste de comunicação serial

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rs232 is
   port ( e : in std_logic;
          rst : in std_logic;
          s_dado : out std_logic;
          cltosend : out std_logic    );
end rs232;
```

```
architecture rs232_arch of rs232 is


constant reset : std_logic_vector(1 downto 0):="00";
constant monta : std_logic_vector(1 downto 0):="01";
constant envia : std_logic_vector(1 downto 0):="10";
constant espera : std_logic_vector(1 downto 0):="11";


signal estado : std_logic_vector(1 downto 0);
signal dado   : std_logic_vector(11 downto 0);
signal parte   : std_logic_vector(7 downto 0);
signal s_e  : std_logic;
signal conta : integer range 0 to 10010;
signal contb : integer range 0 to 14;


begin


-- 156 a 312 38400 bps
-- 312 a 624 19200 bps
-- 441 a 882 14400 bps
-- 625 a 1250 9600 bps
-- 2500 a 5000 2400 bps
-- 5000 a 10000 1200 bps
-- 10000 a 20000 600 bps
-- 20000 a 40000 300 bps


process (e)

begin
  if e'event and e='1' then
      conta <= conta + 1;
      if conta <   5000 then
         s_e <= '0';
       else
         s_e <= '1';
       end if;
       if conta =  10000 then
         conta <= 0;
       end if;
   end if;
end process;


process (s_e,rst)

begin
            if rst ='0' then
                        estado <= reset;
            elsif s_e'event and s_e='0' then
            if estado = reset then
                        dado <= (others => '0');
                        contb <= 0;
                        cltosend <='0';
```

```
                    estado <= monta;
                    parte <= "00000000";
            elsif estado = monta then
                     parte <= parte +1;
                    dado <= "01" & not (parte) &"10";
                    estado <= envia;
            elsif estado = envia then
                    if contb = 0 then
                      cltosend <='1';
                    else
                      cltosend <='0';
                    end if;
                    contb <= contb +1;
                    if contb > 0 and contb < 13 then
                      s_dado<=dado(contb-1);
                    end if;
                    if contb = 13 then
                       estado <= monta;
                       contb <=0;
                    end if;
            end if;
  end if;
end process;

end rs232_arch;
```

8.6.3 - O PROGRAMA DE *DEBUG* EM DELPHI

O programa de *debug* facilita a depuração das instruções implementadas, onde os valores correntes em cada registrador do µHC11 no FPGA podem ser visualizados. Sendo interpretados da esquerda para a direita: registrador A (8 bits), registrador B (8 bits), registrador X (16 bits), registador Y (16 bits), o estado da UCP (8 bits), o PC (contador de programa) e o *opcode* corrente.

Na figura 8.39, pode-se visualizar a interface do programa de *debug*, em funcionamento, desenvolvido em Delphi 6. Os dados representados pelo programa são os valores reais correntes no FPGA, dos registradores do µHC11.

Com a união do código do µHC11, o acesso à memória RAM e o periférico de transmissão serial tem-se um sistema, o qual pode ser visualizado na figura 8.40.

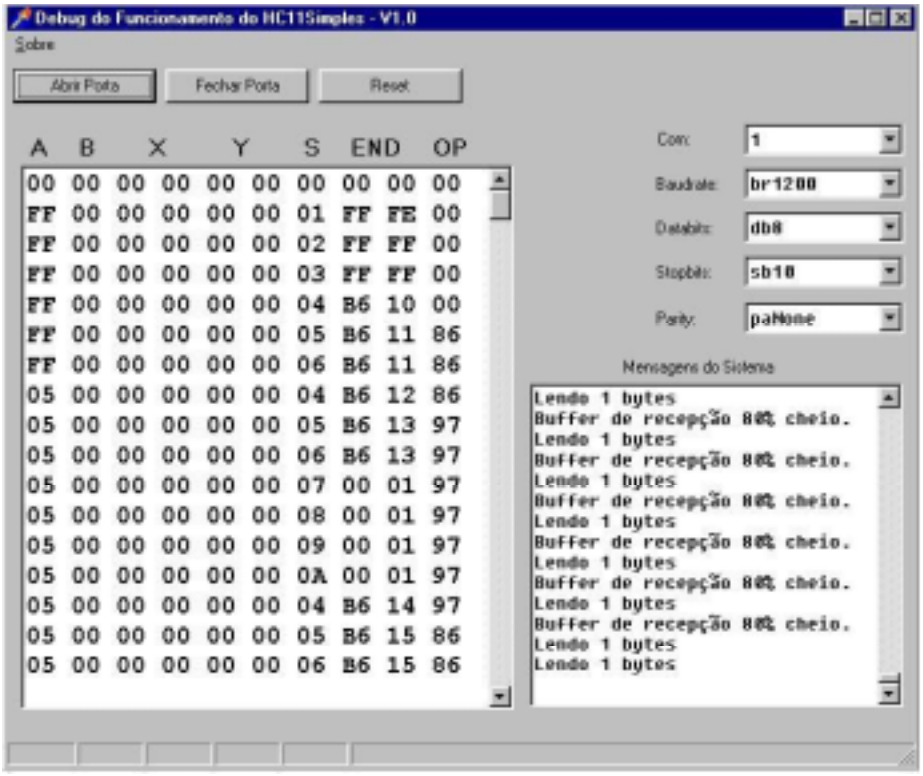


Figura 8.39 - Programa de *debug* e depuração do funcionamento do VHDL_HC11

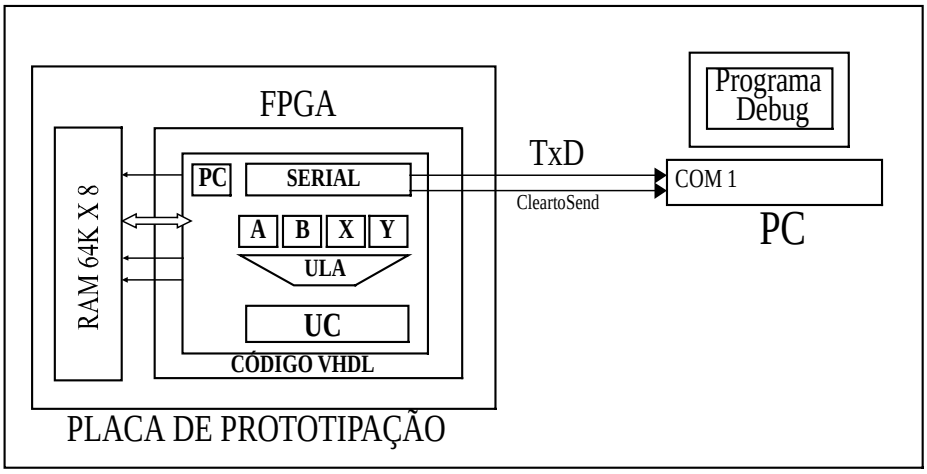


Figura 8.40 - Visão modular do sistema completo

Nas figuras 8.41 e 8.42, o sistema completo pode ser visualizado, formando uma interface amigável e completando e finalizando o sistema proposto.

Figura 8.41 - Integração entre µHC11, e o PC.

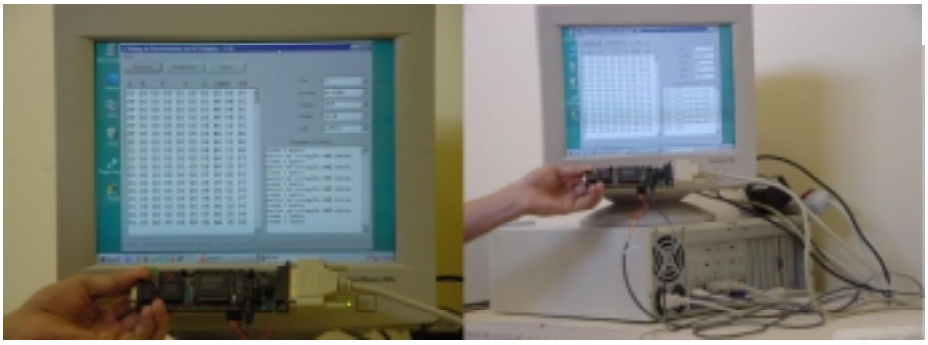


Figura 8.42 - Comprovação do funcionamento do µHC11, com o uso do *debug* no PC.

Este capítulo demostra como é possível realizar sistemas complexos (como é o caso de um microcontrolador) usando-se da integração de conceitos de hardware (VHDL, FPGAs e eletrônica) e software (interface Delphi e programação geral).

O protótipo desenvolvido é de caráter acadêmico, portanto útil aos interessados no estudo e descrição VHDL de microcontroladores. Neste sentido, o µHC11 poderá ser ampliado, descrevendo mais instruções e periféricos do M68HC11.

O protótipo desenvolvido não pode ser comparado, em nenhum aspecto, ao M68HC11 original, sendo que este possui muito mais instruções e periféricos.

A seguir, o código VHDL do µHC11 versão final, onde o conjunto de instruções descritas são: ABA, ABX, ABY, ADDA, ADDB, ANDA, ANDB, LDAA, LDAB, EORA, EORB, STAA, STAB, LDD e JUMP, respeitando todos os modos de endereçamento do M68HC11 original.

µHC11

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity hc11 is
port (e, reset : in std_logic;
      data : inout std_logic_vector (7 downto 0);
      ce: out std_logic;
      we: out std_logic;
      oe: out std_logic;
      address  : out std_logic_vector (15 downto 0);
      cltosend : out std_logic;
      dado_serial : out std_logic);

constant aba  : std_logic_vector (7 downto 0) := "00011011"; -- 1b
constant abx  : std_logic_vector (7 downto 0) := "00111010"; -- 3a
constant adca : std_logic_vector (7 downto 0) := "10001001"; -- 89
constant adcb : std_logic_vector (7 downto 0) := "11001001"; -- c9

constant adda : std_logic_vector (7 downto 0) := "10001011"; -- 8b
constant addb : std_logic_vector (7 downto 0) := "11001011"; -- cb
constant anda : std_logic_vector (7 downto 0) := "10000100"; -- 84
constant andb : std_logic_vector (7 downto 0) := "11000100"; -- c4
constant eora : std_logic_vector (7 downto 0) := "10001000"; -- 88
constant eorb : std_logic_vector (7 downto 0) := "11001000"; -- c8

constant bra : std_logic_vector (7 downto 0) := "00100000"; -- 20
constant ldaa : std_logic_vector (7 downto 0) := "10000110"; -- 86
constant ldab : std_logic_vector (7 downto 0) := "11000110"; -- c6
constant lddd : std_logic_vector (7 downto 0) := "11001100"; -- cc
constant jmp : std_logic_vector (7 downto 0) := "01001110"; -- 4e
constant ldd : std_logic_vector (7 downto 0) := "11001100"; -- cc
constant mul : std_logic_vector (7 downto 0) := "00111101"; -- 3d
constant staa : std_logic_vector (7 downto 0) := "10000111"; -- 87
constant std : std_logic_vector (7 downto 0) := "11001101"; -- cd

constant imm : std_logic_vector (1 downto 0) := "00";
constant dir : std_logic_vector (1 downto 0) := "01";
constant ind : std_logic_vector (1 downto 0) := "10";
constant ext : std_logic_vector (1 downto 0) := "11";

-- constantes necessarias a maquina de envio serial
constant rst       : std_logic_vector(2 downto 0):="000";
constant monta     : std_logic_vector(2 downto 0):="001";
constant envia     : std_logic_vector(2 downto 0):="010";
constant espera    : std_logic_vector(2 downto 0):="011";
end;
```

```
architecture comportamental of hc11 is

-- definição dos estados da unidade de controle para execução das instruções acima

constant zero   : std_logic_vector (3 downto 0) := "0000";
constant um     : std_logic_vector (3 downto 0) := "0001";
constant dois   : std_logic_vector (3 downto 0) := "0010";
constant tres   : std_logic_vector (3 downto 0) := "0011";
constant quatro : std_logic_vector (3 downto 0) := "0100";
constant cinco  : std_logic_vector (3 downto 0) := "0101";
constant seis   : std_logic_vector (3 downto 0) := "0110";
constant sete   : std_logic_vector (3 downto 0) := "0111";
constant oito   : std_logic_vector (3 downto 0) := "1000";
constant nove   : std_logic_vector (3 downto 0) := "1001";
constant dez    : std_logic_vector (3 downto 0) := "1010";
constant onze   : std_logic_vector (3 downto 0) := "1011";

-- definicao das operaoes da ula --
constant alu_pass : std_logic_vector (1 downto 0) := "00";
constant alu_or : std_logic_vector (1 downto 0) := "01";
constant alu_and : std_logic_vector (1 downto 0) := "10";
constant alu_add : std_logic_vector (1 downto 0) := "11";

-- definição da estrutura basica da ucp
signal  pc  : std_logic_vector(15 downto 0);
signal  x   : std_logic_vector(15 downto 0);
signal  y   : std_logic_vector(15 downto 0);
signal  a   : std_logic_vector(7 downto 0);
signal  b   : std_logic_vector(7 downto 0);

-- definição de todos os registradores auxiliares necessarios à execução

signal  dout      : std_logic_vector(7 downto 0);
signal  datain    : std_logic_vector(7 downto 0);
signal  opcode    : std_logic_vector(7 downto 0);
signal  opclass   : std_logic_vector(7 downto 0);
signal  hi     : std_logic_vector(7 downto 0);
signal  lo     : std_logic_vector(7 downto 0);
signal  addr      : std_logic_vector(15 downto 0);
signal  state     : std_logic_vector(3 downto 0);
signal  y_prefix  : std_logic;

-- sinais para decodificacao serial
signal  estado : std_logic_vector(2 downto 0);
signal  d_e, ser_e, f_grava : std_logic;
signal  cont : integer range 0 to 12000010;          -- gera clk 1 segundo para a ucp
signal  conta : integer range 0 to 10010;            -- gera clk 1200 bps para a transmissao serial
signal  contb : integer range 0 to 11;               -- utilizado para o deslocamento serial
signal  passo : integer range 0 to 40;               -- utilizado para montar o valor em dado
signal  f_envio : std_logic;      -- signal utilizado para liberar a maquina de estados de envio
signal  dado : std_logic_vector(11 downto 0);  -- armazena o dado a ser transmitido serialmente

alias mode : std_logic_vector(1 downto 0) is opcode(5 downto 4);

begin

--  leitura e escrita no chip ram
```

```
--  ce = 0, habilita o chip ram
--  oe = 0, habilita saida de dados da ram
--  we = 1, habilita gravacao de dados da ram
    ce <= '0';
    oe <= '0' when  f_grava ='0' else '1';
    we <= '1' when  f_grava ='0' else '0';

  address <= pc;

  -- controle da e/s da ram

  with f_grava select
      data <= dout  when '1',
              "zzzzzzzz"   when others;

  ---------------------------------------------------------------------------------------------
  --  processo de reducao de clock para debug e transmissao serial
  -- entradas   e : std_logic   => entrada de clock de 12mhz
  -- passo : integer => monitoramento da maquina de envio serial
  --
  -- saidas     d_e : std_logic   => clock reduzido para a ucp, 1 ciclo por segundo
  -- ser_e : std_logic => clock para maquina de envio
  --f_envio : std_logic => libera ou suspende a maquina de envio dos
-- caracteres seriais
  -- controles  cont  : integer => contador para reducao do clock de 12mhz
  -- conta :integer => contador para gerar a frequencia exata de transmissao serial txd
  -- valores com um clock de 12 Mhz
  --          156 a 312 38400 bps
  --          312 a 624 19200 bps
  --          441 a 882 14400 bps
  --          625 a 1250 9600 bps
  --          2500 a 5000 2400 bps
  --          5000 a 10000 1200 bps
  --          10000 a 20000 600 bps
  --          20000 a 40000 300 bps
  ---------------------------------------------------------------------------------------------

  process (e)
   begin
    if e'event and e='0' then
      cont <= cont + 1;
      conta <= conta + 1;
      if cont <  6000000 then -- 1 ciclo a cada segundo
        d_e <= '0';
      else
        d_e <= '1';
      end if;
      if cont = 12000000 then
        cont <=0;
      end if;
      if conta <  5000  then -- 1200 bps
        ser_e <= '0';
      else
        ser_e <= '1';
      end if;
      if conta = 10000 then
        conta <=0;
      end if;
```

```
    if cont =  7000000 then -- apos 1 ciclo da ucp, libera maquina de envio txd
      f_envio <= '1';
    end if;
    if passo = 10 then      -- se enviou todos os caracteres, suspende a maquina de envio
      f_envio <= '0';
    end if;
 end if;
end process;
-------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------
--              processo de transmissao serial txd
-------------------------------------------------------------------------------------------
 process (ser_e,reset)
 begin
    if reset ='0' then
       estado <= rst;
    elsif ser_e'event and ser_e='1' then
      if estado = rst then
         dado <= (others => '0');
         cltosend <='0';
         passo <= 0;
         contb <= 0;
         estado <= espera;
      elsif estado = espera then
         if f_envio ='1' then      -- se a ucp gerou um ciclo libera montar o dado a ser enviado
            estado <= monta;
         end if;
      elsif estado = monta then
       case passo is
           when 0 => dado <= "01" & not (a) & "10" ;  -- registrador a
                   estado <= envia; -- a
            when 1 => dado <= "01" & not (b) & "10" ;   -- registrador b
                   estado <= envia; -- b
            when 2 => dado <= "01" & not  (x(15 downto 8)) & "10" ;  -- inicia  x
                   estado <= envia; -- high x
           when 3 => dado <= "01" & not (x(7 downto 0)) & "10" ;    -- termina  x
                   estado <= envia; -- low x
           when 4 => dado <= "01" & not (y(15 downto 8)) & "10" ;   -- inicia  y
                   estado <= envia;
           when 5 => dado <= "01" & not (y(7 downto 0)) & "10" ;    -- termina  y
                   estado <= envia;
           when 6 => dado <= "01" & not ("0000" & state) &"10" ;    -- envia o estado da ucp
                   estado <= envia;
           when 7 => dado <= "01" & not (pc(15 downto 8)) & "10" ;  -- inicia o endereco
                   estado <= envia;
            when 8 => dado <= "01" & not (pc(7 downto 0)) & "10" ;   -- termina o endereco
                   estado <= envia;
           when 9 => dado <= "01" & not (opcode) & "10" ;
                   estado <= envia;
           when 10 => estado <= espera; -- se o passo igual a 10, retorna a espera
                   passo  <= 0;
           when others => null;
      end case;
      passo <= passo +1;
   elsif estado = envia then
           if contb = 0 then
             cltosend <='1';
```

```
            else
              cltosend <='0';
            end if;
            contb <= contb +1;
            if contb > 0 and contb < 13 then
              dado_serial<=dado(contb-1);
            end if;
            if contb = 13 then
               estado <= monta;
               contb <=0;
            end if;
            end if;
    end if;
   end process;

  ------------------------------------------------------------------------------------------------
--                                  processo ucp completa
  ------------------------------------------------------------------------------------------------
  process (reset,d_e)
    procedure inc_pc is
    begin
     pc <= pc + 1;
     state <= cinco;
    end;

    function ula ( op1,op2 : std_logic_vector(7 downto 0); ula_op : std_logic_vector(1 downto 0))
return std_logic_vector is
    variable ula_out : std_logic_vector(7 downto 0);
    begin
    case ula_op is
          when alu_pass =>          ula_out := op2;
          when alu_or   =>          ula_out := op1 or op2;
          when alu_and  =>          ula_out := op1 and op2;
          when alu_add  =>          ula_out := op1 + op2;
          when others => null;
    end case;
    return (ula_out);
    end;

    procedure decode is
    begin
     if  opclass = ldaa then
        a <= ula (a, datain, alu_pass);
     elsif opclass = ldab then
        b <= ula (b, datain, alu_pass);
     elsif opclass = eora then
        a <= ula (a, datain, alu_or);
     elsif opclass = eorb then
        b <= ula (b, datain, alu_or);
     elsif opclass = anda then
        a <= ula (a, datain, alu_and);
     elsif opclass = andb then
        b <= ula (b, datain, alu_and);
     elsif opclass = adda then
        a <= ula (a, data, alu_add);
     elsif opclass = addb then
        b <= ula (b, data, alu_add);
     end if;
```

```
  end;

begin
  if reset='0' then
    state <= zero;
  elsif d_e'event and d_e='0' then
    if (state = zero) then
      pc <= "1111111111111110";
      a   <= "00000000";
      b   <= "00000000";
      x   <= "0000000000000000";
      y   <= "0000000000000000";
      y_prefix <='0';
      f_grava <='0';
      state <= um;
    elsif (state = um) then
      hi <= data;
      pc <= "1111111111111111";
      state <= dois;
    elsif (state = dois) then
      lo <= data;
      state <= tres;
    elsif (state = tres) then
      pc <= hi & lo;
      state <= quatro;
    elsif state = quatro then
     opcode  <= data;
     opclass <= data and "11001111";
     inc_pc;
     state <= cinco;
    elsif state = cinco then
      if opcode = "00011000" then -- se encontrar o opcode 18, direciona y
        y_prefix <='1';
        state <= quatro;         -- pode retornar ao estado inicial de busca
      else
        datain <= data;
        state  <= seis;
      end if;
   else
    case opcode is
    when aba =>  a <= ula(a,b,alu_add);
                 state <= quatro;
    when abx =>  if y_prefix ='0' then x <= x + ("00000000" & b); else y <= y + ("00000000" & b);
    end if;
    when others =>
      case opclass is
          when  adda | addb | anda | andb | ldaa | ldab | eora | eorb  =>
            case mode is
                when imm =>
                  decode;
                  inc_pc;
                  state <= quatro;
            when dir =>
                  if state = seis then
                   addr <= pc+1;
                   pc <= "00000000" & datain;
                   state <= sete;
                elsif state = sete then
```

```
            decode;
            pc <= addr;
            state  <= quatro;
          end if;
        when ext =>
          if state = quatro then
            inc_pc;
            state <= cinco;
          elsif state = cinco then
            inc_pc;
            state <= seis;
          elsif state = seis then
            inc_pc;
            hi <= data;
            state <= sete;
          elsif state = sete then
            lo <= data;
            state <= oito;
          elsif state = oito then
            addr <= pc+1;
            pc <= hi & lo;
            state <= nove;
          elsif state = nove then
            decode;
            pc <= addr;
            state <= quatro;
          end if;
        when ind =>
          if state = seis then
            if y_prefix='1' then
              y <= std_logic_vector( y +  data);
            else
              x <= x +  data;
            end if;
            state <= sete;
          elsif state = sete then
            addr <= pc+1;
            if y_prefix='1' then
              pc <= y;
            else
              pc <= x ;
            end if;
            state <= oito;
          elsif state = oito then
            decode;
            pc <= addr;
            y_prefix <='0';
            state <= quatro;
          end if;
        when others => null;
        end case;
      when staa  =>
          case mode is
            when dir => if state = seis then
                          addr <= pc+1;
                          pc <= "00000000" & datain;
                        state <= sete;
             elsif state = sete then
```

```
                dout <= a;
                state <= oito;
                    elsif state = oito then
                      f_grava <='1';
                        state <= nove ;
                    elsif state = nove then
                      f_grava <='0';
                      state <= dez ;
                    elsif state = dez then
                      pc <= addr;
                      state <= quatro;
                   end if;
        when others => null;
       end case;
when ldd =>
  case mode is
   when imm =>
      if state = seis then
        a <= data;
        inc_pc;
        state <= sete;
      elsif state = sete then
        b <= data;
        inc_pc;
        state <= quatro;
     elsif state = oito then
        b <= data;
        state <= quatro;
      end if;
   when dir =>
      if state = seis then
         addr <= pc+1;
        pc <= "00000000" & datain;
         state <= sete;
      elsif state = sete then
         a <= data;
         pc <= pc+1;
        state <= oito;
      elsif state = oito then
         b <= data;
         pc <= addr;
         state <= quatro;
      end if;
    when ext =>
       if state = seis then
         addr(15 downto 8)<= data;
         inc_pc;
         state <= sete;
       elsif state = sete then
         addr (7 downto 0) <= data;
         state <= oito;
       elsif state = oito then
         addr <= pc+1;
         pc <= addr;
         state <= nove;
      elsif state = nove then
         a <= data;
         inc_pc;
```

```
                    state <= dez;
                  elsif state = dez then
                    b <= data;
                    pc <= addr;
                    state <= quatro;
                  end if;
                when ind =>
                    if state = seis then
                     if y_prefix ='1' then
                      y <= y + data;
                     else
                      x <= x + data;
                     end if;
                     state <= sete;
                    elsif state = sete then
                      addr <=pc+1;
                      if y_prefix='1' then
                       pc <= y;
                      else
                       pc <= x ;
                      end if;
                      state <= oito;
                    elsif state = oito then
                      a <= data;
                      inc_pc;
                      state <= nove;
                    elsif state = nove then
                      b <= data;
                      pc <= addr;
                      y_prefix <= '0';
                      state <= quatro;
                    end if;
                 when others => null;
                 end case;
               when jmp =>
                 case mode is
                 when ext =>
                    if state = seis then
                      inc_pc;
                      state <= sete;
                    elsif state = sete then
                      pc <= datain & data;
                      state <=quatro;
                    end if;
                 when others => null;
                 end case;
             when others => state <= quatro;
        end case;
       end case;
    end if;
  end if;
end process;
end;
```

É importante salientar que no desenvolvimento de um projeto, seja este em qualquer área, demanda muito estudo e força de vontade, para se obter resultados satisfatórios.

CAPÍTULO IX - REFERÊNCIAS BIBLIOGRÁFICAS

LIVROS

John F. Wakerly, " Digtal Design Principles & Practices", Prentice Hall, 2000.

Thomas Richard McCalla, "Digital Logic and Computer Design", Macmillan Publishing Company, 1992.

Pak K. Chan, Samiha Mourad, "digital design using Field Programmable Gate Array", Pretince Hall, 1994.

Perry D. L. " VHDL Third Edition", McGraw-Hill,Inc. 1998.

Peter J. Ashenden, " Then Student's Guide to VHDL", Morgan Kaufmann Publishers, 1998.

Dezso Sima, Terence Fountain , Peter Kacsuk, " Advanced Computer Architectures A design Space Approach", Addison-Wesley, 1997.

Kai Hwang, Fayé A. Briggs "Computer Architecture and Parallel Processing", McGraw-Hill, 1984

Patterson Hennesy, "Computer Organization", Morgan Kaufmann

Kevin Skahill. VHDL for Programmable Logic. Editora Addison-Wesley

Douglas J. Smith - HDL Chip Design / A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog – Doone Publications.1997.

Stephen Brown, Zvonko Vranesic, Fundaments of Digital Logic With VHDL Design, 2000

John V. Oldfield, Richard C. Dorf , Field Programmable Gate Array, 1995

James Bignell, Robert Donovan. Eletrônica Digital: Lógica Combinacional. Editora Makron Books, 1995.

James Bignell, Robert Donovan. Eletrônica Digital: Lógica Seqüêncial. Editora Makron Books, 1995.

Luigi Carro – "Projeto e Prototipação de Sistemas Digitais" – Ed. Universitaria, 2001

Ronald J. Tocci – Sistemas Digitais Princípios e Aplicações Prentice-Hall do Brasil,1996.

Edward Moreno, Jorge Luis e Silva. Computação Reconfigurável: Experiências e Perspectivas. Editora FEESR, Agosto, 2000.

Anderson Royes Terroso, semana da computação PUCRS, "Dispositivos Lógicos Programáveis (FPGA) e Linguagem de Descrição de Hardware (VHDL)", PUCRS – 1998.

Andrew S. Tanenbaum - Organização Estruturada de Computadores – Terceira Edição - Editora Prentice-Hall do Brasil LTDA,1992

Dave Van den Bout, The Pratical Xilinx Designer Lab Book, 1997

Pedro A. Medoe; Curso Básico de Telefonia Editora Saber LTDA, Maio/2000.

LINKS

Telefonia

Telecon (http://www.telecon.com.br)

Instituto Nacional de Telecomunicações (INTEL) http://www.intel.com.br

Projeto Voice Recognition Telephone Dialer. Tim Golding, Eric Cheung, Felicia Cheng, Wilson (LATA) Kwan, Davi Li.
http://www.ee.ualberta.ca/~elliott/ee552/projects/2000_w/VRTD/FinalReport.htm

Turbina

IOPE – Instrumentos de Precisão – http://www.iope.com.br/p_temperatura.htm

The Wren MW-54 ECU - http://www.5bears.com/ecu.htm

Turbina KJ66 – Paul's Microjets - http://www.microjets.co.uk/

Informações sobre o Microcontrolador M68HC11

http://www.motorola.com/mcu/hc11/

http://home1.gte.net/tdickens/68hc11/docs_references.htm

http://www.hc11.demon.nl/thrsim11/68hc11/

ftp://nyquist.ee.ualberta.ca/pub/motorola/68hc11/

http://www.gmvhdl.com/downloads.html

http://www.gmvhdl.com/about.html

http://www.motorola.com

Microcontroladores em geral

http://www.microcontrolador.com.br

http://dec1.wi-inf.uni-essen.de/~astephan/links/links_micro.htm

http://ee.cleversoul.com/

http://www.microchip.com/1010/pline/picmicro/families/16f8x/devices/16f84/index.htm

http://developer.intel.com/design/MCS51/MANUALS/27238302.pdf

http://www-us7.semiconductors.philips.com/acrobat/datasheets/
8XC51_8XC52_6.pdf

http://www.atmel.com

http://focus.ti.com/docs/prod/folders/print/tms370c777a.html

http://www.national.com/pf/CO/COP820C_840C.html

Software de simulação VHDL, VhdlStudio

http://www.gmvhdl.com/*download*s.html

Fabricantes de FPGAs

http://www.xilinx.com

http://www.altera.com

http://www.actel.com

Simulador do M68HC11, Shadow11

http://www.geocities.com/SiliconValley/Peaks/4125

Montador ASM11

http://www.aspisys.com

Empresa Xess

http://www.xess.com

http://www.xess.com/manuals/xs40-manual-v1_4.pdf

http://www.xess.com/manuals/xstools-v4_0.pdf

Winbond, fabricante de chips de memória RAM

http://www.winbond.com

Documento para definição do formato Hex "Intel Hex Format"

http://www.keil.com/support/docs/1584.htm

A comunicação serial e o protocolo RS232

http://www.bb-europe.com/welcome.html

Tabela ASCII

http://www.e-kit.com.br/tabela_ascii.asp?topo=n