UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL - CPGEI

Sibilla Batista da Luz

Método Complementar para Avaliação de um Microcontrolador de Código Aberto

DISSERTAÇÃO DE MESTRADO

CURITIBA
10 de fevereiro de 2009.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

---

DISSERTAÇÃO
apresentada a UTFPR
para obtenção do grau de

MESTRE EM CIÊNCIAS

por

SIBILLA BATISTA DA LUZ

MÉTODO COMPLEMENTAR PARA AVALIAÇÃO DE UM MICROCONTROLADOR DE CÓDIGO ABERTO

Banca Examinadora:

Presidente e Orientador:

PROF. DR. VOLNEI ANTONIO PEDRONI	**UTFPR**

Examinadores:

PROF. DR. ALTAIR OLIVO SANTIN	**PUCPR**
DR. CHRISTOPHE FREDERIC LUCIEN BRICOUT	**AXP MICROELETRÔNICA**
PROF. DR. CARLOS RAIMUNDO ERIG LIMA	**UTFPR**

Curitiba, Fevereiro de 2009.

SIBILLA BATISTA DA LUZ

MÉTODO COMPLEMENTAR PARA AVALIAÇÃO DE UM MICROCONTROLADOR DE CÓDIGO ABERTO

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do grau de "Mestre em Ciências" – Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Volnei A. Pedroni.
Co-Orientador: Dr. Christophe F. L. Bricout.

Curitiba
2009

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

L979m Luz, Sibilla Batista da
Método complementar para avaliação de um microcontrolador de código aberto / Sibilla Batista da Luz. – Curitiba : [s.n.], 2009.
xiii, 94 p. : il. ; 30 cm

Orientador: Volnei A. Pedroni
Co-orientador: Christophe F. L. Bricout
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração : Informática Industrial, Curitiba, 2009
Bibliografia: p. 93-94

1. Microprocessadores. 2. VHDL (Linguagem descritiva de hardware). 3. Ethernet (Sistema de rede local de computação). 4. Dispositivos lógicos programáveis. I. Pedroni, Volnei A. (Volnei Antonio), orient. II. Bricout, Christophe F. L., co-orient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração Informática Industrial. IV. Título.

CDD 621.3

AGRADECIMENTOS

Agradeço primeiramente a Deus, meu pai querido, que nunca me deixa só.

Ao meu orientador Prof. Volnei Pedroni, pelo apoio e incentivo, e principalmente por ter me dado à oportunidade de cursar este mestrado.

Ao meu co-orientador Christophe Bricout por todos os ensinamentos, dedicação, e paciência no desenvolvimento deste projeto.

Ao Ricardo Jasisnki que muitas vezes me ajudou com respostas para minhas dúvidas e dicas.

Aos meus pais que sempre torceram por mim e ao meu esposo pelo incentivo e amor.

Aos amigos do LME.

A AXP Microeletrônica pela parceria.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AVR	*Automatic Voltage Regulation*
BCI	Bloco de Interface e Controle
CI	Circuito Integrado
CISC	*Complex Instruction Set Computer*
CLB	*Configurable Logic Block*
CMOS	*Complementary-Symmetry Metal-Oxide Semiconductor*
CPLD	*Complex* PLD
CRC	C*yclic Redundance Check*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
DEC	*Digital Equipment Corporation*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
FPGA	*Field Programmable Gate Array*
GAL	*Generic* PAL
HSTL	*High-Speed Transceiver Logic*
IP	*Intellectual Property*
JMV	*Java Virtual Machine*
JTAG	*Joint Test Action Group*
LAN	*Local Area Network*
LUT	*Lookup Table*
LVCMOS	*Low Voltage* CMOS
LVDS	*Low-Voltage Differential Signaling*
LVPECL	*Low-Voltage Positive/Pseudo Emitter-Coupled Logic*
LVTTL	*Low Voltage* TTL
MAC	*Media Access Control*
PAL	*Programmable Array Logic*
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PPL	*Phased Lock Loop*
PROM	*Programmable Read-Only Memory*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*

SoC	*System-on-Chip*
SOP	*Sum-Of-Products*
SPLD	*Simple* PLD
SRAM	*static* RAM
SSTL	*Stub Series Terminated Logic*
VHSIC	*Very-High-Speed Integrated Circuit*
VHDL	VHSIC *Hardware Description Language*
VLSI	*Very Large Scale Integration*
TTL	*Transistor-Transistor Logic*
µC	Microcontrolador

RESUMO

O objetivo deste trabalho é desenvolver e implementar um método complementar para avaliação de um microcontrolador de código aberto. Para que, o mesmo possa ser usado, posteriormente, em projetos inovadores com diferentes aplicações, porém com algumas restrições para uso comercial.

Para o desenvolvimento deste projeto, o primeiro passo consistiu em pesquisar os vários microcontroladores disponíveis, para então selecionar o mais completo, contendo documentação, compilador gratuito, *testbench*, etc. O microcontrolador selecionado foi o ATmega103, de 8 bits, descrito em VHDL (VHSIC *Hardware Description Language*).

Para garantir que o *core* funciona corretamente é necessário realizar vários testes. Para isso foram criados códigos em C e Assembly, contendo instruções que foram gravadas na memória de programa do microcontrolador, o qual deve executá-las.

Com o objetivo de tornar esse método complementar independente da plataforma, fabricante e do *software* de *design* utilizado, um bloco Ethernet foi integrado ao projeto, onde os dados carregados em uma memória de execução são transmitidos através de frames pela rede local.

Para que a comunicação entre o bloco Ethernet e o microcontrolador fosse possível, desenvolveu-se um bloco em VHDL para realizar a interface e o controle entre eles. Este é responsável, entre outras coisas, por realizar a troca entre a memória de programa e a RAM (*Random Access Memory)* que contém o código a ser executado. Isso ocorre quando é detectada a chegada de um frame. Desta maneira, o microcontrolador passa a executar as instruções da memória que contém o frame e não mais da memória de programa antiga.

Com a utilização deste método complementar, uma grande quantidade de códigos distintos pode ser verificada em *hardware* com maior facilidade e posteriormente comparada com os resultados esperados ou obtidos em simulação.

PALAVRAS-CHAVE:

Avaliação, microcontrolador, código aberto, Ethernet, linguagem VHDL.

ABSTRACT

SUPPLEMENTARY METHOD FOR ASSESSMENT OF AN OPEN CORE MICROCONTROLLER

The objective of this project is to develop and implement a supplementary method for assessment of a free open core microcontoller to use it, later, in some innovative projects with different aplications and some restriction for commercial use.

For the development of this project, the first step was to list all the microcontrollers available, then select the best dressed and including documentation, free compiler, testbench, etc. The microcontroller selected was the Atmega103, with 8-bit, described in VHDL.

To ensure that the core functions properly it is necessary to perform various tests. For that, many C and Assembly codes were created and written into microcontroller's program memory for execution.

In order to have a generic method independent of platform, manufacturer and software design, an Ethernet block has been embedded to the FPGA, where the data loaded into the program memory are transmitted, in packages, through the local network.

For the communication between the Ethernet block and microcontroller, it has been developed a block in VHDL handling the transmission. The block is also responsible to switch properly the program memory and RAM that contains the code to be executed. The memory switch occurs when it detects a incoming package. Thus, the microcontroller starts to execute the instructions of memory that contains the package rather than the memory of program memory.

With the use of this method, a lot of different codes can be easy checked in hardware and then compared with results expected or obtained in simulation.

KEYWORDS:

Assessment, microcontroller, open core, Ethernet, VHDL.

CAPÍTULO 1

INTRODUÇÃO

1.1 MOTIVAÇÕES

Com o avanço da tecnologia, atualmente é possível construir circuitos integrados complexos capazes de conter todos os elementos de um produto completo. Este tipo de dispositivo é conhecido como SoC (*System-on-Chip*) [1].

Os SoCs contém componentes como processador, memória e dispositivos periféricos. São, na maioria das vezes, implementados como uma coleção de componentes reutilizáveis chamados IP (*Intellectual Property*) *core*.

Um IP *core* é um complexo módulo de *hardware* criado para ser reutilizado e que realiza algumas tarefas específicas.

Com base nesses conceitos, surgiu a proposta do projeto que visa desenvolver um método complementar para avaliação de um microcontrolador, através de um módulo de testes e um IP *Core* Ethernet, unidos em um único dispositivo.

O desenvolvimento de sistemas microprocessados em dispositivos lógicos programáveis apresenta várias vantagens quando comparados com os sistemas convencionais [1, 2]. Utilizando as FPGAs (*Field Programmable Gate Arrays*), é possível alterar a estrutura do *hardware* contida no dispositivo, ou seja, a configuração do mesmo. Assim, funções adicionais podem ser incrementadas ao sistema ou erros podem ser corrigidos após a implementação. Além disso, apresenta baixo custo de produção e desenvolvimento, bem como facilidade de prototipagem e portabilidade.

Vários projetos, dentre eles microcontroladores, são disponibilizados pelo site www.opencores.org. Esses projetos possuem o código aberto e estão disponíveis gratuitamente para os usuários. Muitos deles possuem um tipo de licença que permite que sejam utilizados livremente, sem a necessidade de pagar por isso, porém com algumas restrições para uso comercial.

No entanto, como os projetos não são versões comerciais, que passam por rigorosos testes, não é possível garantir que toda a implementação funciona corretamente. É necessário executar várias combinações de instruções para verificar se não há nenhum erro de execução. Depois de avaliado, o microcontrolador poderá ser utilizado em projetos futuros para diferentes aplicações, como sistemas dedicados de controle, monitoramento, etc.

A verificação functional de um microcontrolador é algo bastante complexo. Cerca de 60% dos resursos, humanos e computacionais, de um projeto são dedicados a este propósito [3, 4]. Se após a finalização do projeto determinadas características não funcionarem de acordo com as especificações da arquitetura, em alguns casos, principalmente em usos comerciais, o prejuízo financeiro pode ser altíssimo. Um exemplo disso ocorreu em 1994, quando um bug presente no processador Pentium gerou à Intel um prejuízo de aproximadamente $475,000,000 [5].

Normalmente, a verificação funcional inclui testes obtidos de diferentes fontes. Inicialmente, alguns testes são criados manualmente visando atingir problemas específicos. Em seqüência, verificações em simulação são realizadas com pacotes de testes prontos (conjuntos de testes comerciais, por exemplo). Após a conclusão do *hardware*, grandes aplicações são executadas (sistemas operacionais, por exemplo). Finalmente, são utilizados testes automáticos, geralmente pseudo-aleatórios [5]. Com a utilização de diferentes fontes de testes, é possível obter uma cobertura maior na verificação funcional do microcontrolador.

O método complementar proposto visa facilitar a verificação de um microcococontrolador de 8 bits de código aberto. Onde a confirmação de sucesso ou falha do teste é realizada através de *self-tests* descritos pelo usuário. Entretanto, outras técnicas de verificação como testes pseudo-aleátórios podem ser utilizados. Para isso, o usuário deve fornecer esses testes, e envia-los como dados de entrada para o circuito de testes.

1.2 OBJETIVOS

O objetivo principal deste projeto é desenvolver e implementar um método complementar para avaliação para microcontroladores de oito bits de código aberto. Assim, o microcontrolador poderá ser utilizado com confiança em projetos futuros. Além disso, este possui licença livre, podendo assim ser utilizado em aplicações gratuitamente, com algumas restrições para uso comercial.

A comunicação entre o computador e o circuito de teste deverá será feita através da rede Ethernet. Desta maneira, é possível enviar os dados que preencherão a memória de execução em frames através da rede. Após a recepção e a gravação dos dados em uma memória, o microcontrolador passará a executar as instruções contidas nela, como apresentado na figura 1.1.

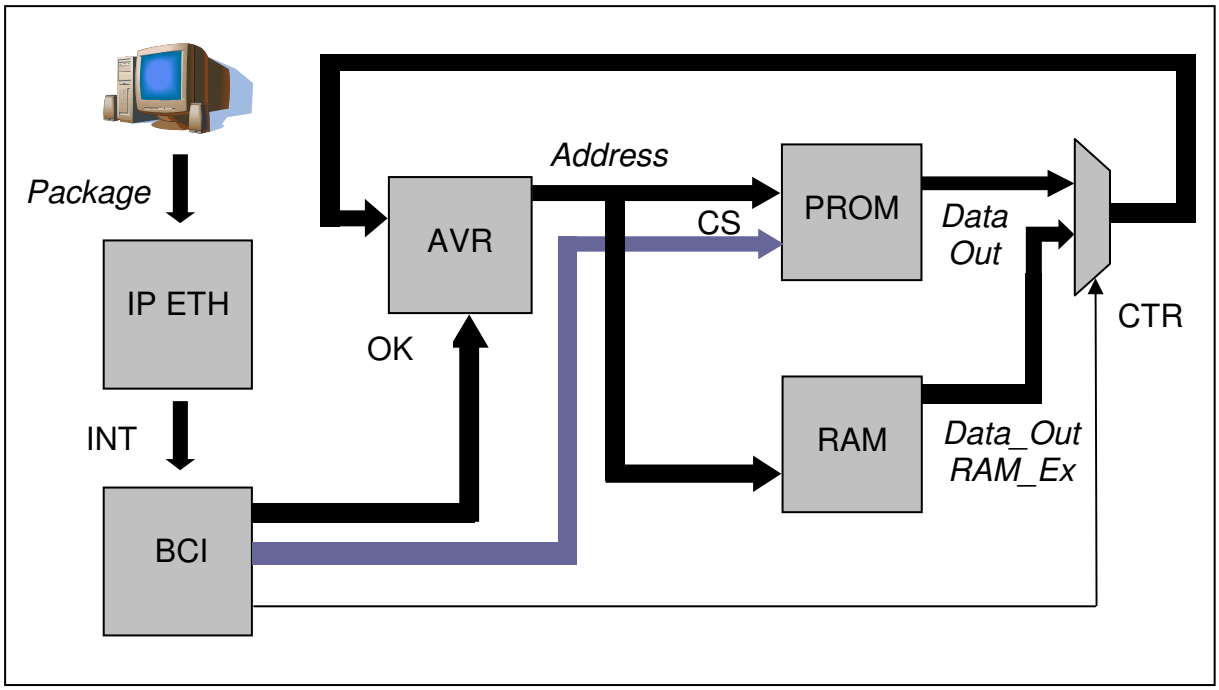

Figura 1.1: Diagrama em blocos do sistema.

Outro objetivo considerado é tornar este método complementar independente de sistema operacional ou *design software* utilizado. Assim o usuário tem a possibilidade de utilizar a ferramenta que considerar viável, levando em consideração principalmente custos. Através da comunicação Ethernet isso se torna possível, além de apresentar alta velocidade de transmissão.

1.3 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada da seguinte maneira:

Capítulo 2 - Apresenta uma revisão da literatura sobre os tópicos abordados neste documento, isto é, dispositivos lógicos programáveis, microcontroladores, rede Ethernet, estrutura do *frame* Ethernet, controle de acesso ao meio e, finalmente, sobre o protocolo WISHBONE.

Capítulo 3 – Este capítulo descreve o método complementar desenvolvido para a verificação e a avaliação de um microcontrolador de código aberto. Iniciando pela pesquisa realizada com todos no microcontroladores disponíveis no OpenCores para seleção de um único microcontrolador que atendesse as características desejadas. Em seguida, apresenta o protocolo utilizado pelo bloco de controle e interface para a gravação dos registradores Ethernet. Finalmente, a manipulação do frame recebido e a substituição da memória de programa pela memória RAM, para execução das instruções enviadas pelo usuário, visando a avaliação do microcontrolador.

Capítulo 4 – São apresentados os resultados obtidos em simulação, além de exemplos realizados com a utilização de alguns códigos simples em Assembly, além de ondas capturadas do osciloscópio.

Capítulo 5 – Neste capítulo são analisados os resultados obtidos, bem como, as dificuldades encontradas, conclusões finais e propostas para trabalhos futuros.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

2.1 INTRODUÇÃO

Este capítulo apresenta a fundamentação teórica dos tópicos abordados neste projeto. Na seção 2.2, são abordados os dispositivos lógicos programáveis e sua evolução durante os anos. A seção 2.3, apresenta os microcontroladores, incluindo sua utilização e tecnologia. A seção 2.4, apresenta as técnicas de verificação funcional existentes. Na seção 2.5, são apresentados, a rede Ethernet e seus princípios básicos, além de histórico, estrutura do *frame* e o protocolo de acesso ao meio. Finalmente, a seção 2.6 aborda o protocolo de interconexão WISHBONE.

2.2 DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Um PLD (*Programmable Logic Device*) é um CI (Circuito Integrado) que contém um grande número de portas lógicas, *flip-flops*, e outros subsistemas que são interconectados de forma programável no chip. Esses dispositivos permitem que o usuário final especifique a operação lógica do dispositivo por meio de um processo denominado "programação", assim a função específica do CI para dada aplicação pode ser determinada selecionando-se as conexões que devem ser abertas e quais devem ficar fechadas. Para isso, o usuário utiliza um computador com um *software* de desenvolvimento para PLD [6].

2.2.1 Histórico

Os dispositivos lógicos programáveis surgiram nos anos 70, onde a idéia era construir um circuito lógico combinacional que fosse programável. Desta maneira, o *hardware* não seria mais fixo, mas sim configurado para atender a cada especificação em particular [7].

Os primeiros PLDs eram chamados de PAL (*Programmable Array Logic*) ou PLA (*Programmable Logic Array*), empregavam somente portas lógicas convencionais, portanto serviam somente para a implementação de circuitos combinacionais.

No inicio dos anos 80, os PLDs evoluíram e passaram a incluir também *flip-flops* e multiplexadores. Continuaram sendo programáveis, o que permitia vários modos de operação. Além disso, o sinal de saída podia ser inserido novamente no circuito (*feedback*), o que trouxe

uma grande flexibilidade. Essa nova estrutura da PLD passou a ser chamada de GAL (*Generic* PAL).

Estes chips, PAL e GAL, foram classificados como SPLDs (*Simple* PLDs).

Em meados dos anos 80, as PLDs simples foram substituídas por CPLDs (*Complex* PLDs), obtidas pela construção e associação de vários GALs no mesmo chip. Estes dispositivos se comunicavam através de um sofisticado esquema de roteamento. Possuíam várias características adicionais, como suporte a JTAG (*Joint Test Action Group)*, diversidade de padrão de I/O, vasto número de pinos de I/O para o usuário e baixo consumo de energia.

As CPLDs tornaram-se bastante populares por apresentarem, em alguns casos, alta densidade, alta performance e baixo custo.

Foi nos anos 80 que também as FPGAs apareceram. Elas diferem das CPLDs na arquitetura, tecnologia, características de construção e custo. A FPGA é aplicada em sistemas complexos e de alta performance. Além disso, diferenciam-se na volatilidade, pois CPLDs utilizam memória EEPROM (*Electrically-Erasable Programmable Read-Only Memory)* ou *FLASH,* sendo, portanto, não voláteis, enquanto que as FPGAs são voláteis visto que utilizam memória SRAM (*static* RAM).

Para implementar um circuito em um dispositivo lógico programável, o circuito deve ser descrito em uma linguagem de descrição de *hardware*, normalmente VHDL ou Verilog, os quais são independentes de tecnologia ou fabricante do dispositivo [7].

2.2.2 FPGAs

Fisicamente, uma FPGA consiste em um arranjo bidimensional de blocos lógicos-programáveis, conectados entre si por meio de uma estrutura de interconexões (*Switch Matrix*), como mostra a figura 2.1.

A função lógica implementada por um elemento, definida pelos estados das chaves transistorizadas que compõem este mesmo elemento, bem como as conexões entre os blocos funcionais, são determinadas por vetores binários de configuração, que podem ser transferidos para a FPGA, repetidamente, a partir de uma fonte externa.

A estrutura interna de um CLB (*Configurable Logic Block*) é diferente de um PLD. Primeiro, ao invés de implementar expressões SOP (*Sum-Of-Products*) com portas AND seguidas de OR, o sistema é baseado em LUT (*Lookup Table*). Em uma FPGA o número de *flip-flops* é muito maior que em um CPLD, o que possibilita construir circuitos muito mais complexos.

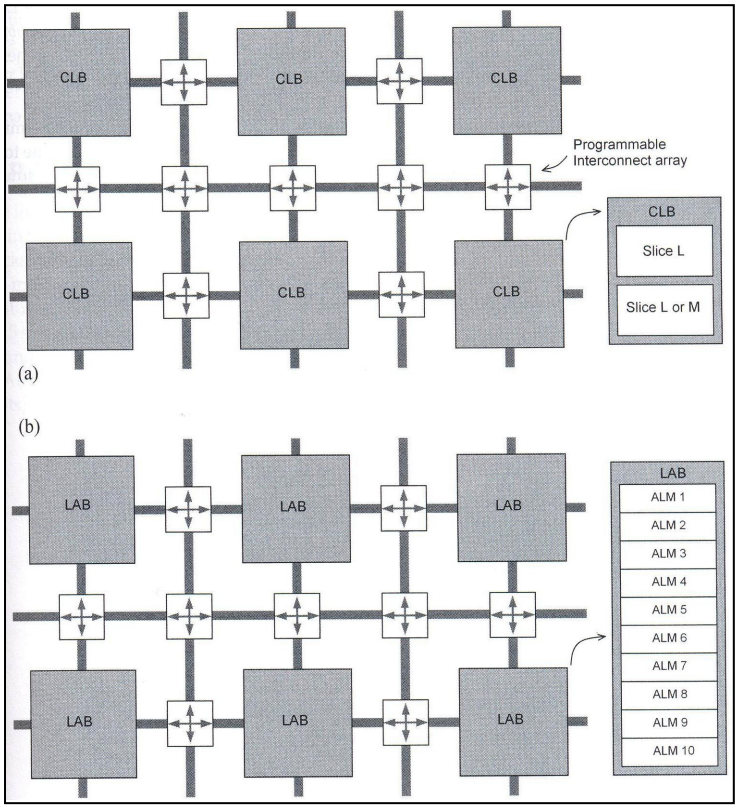

Figura 2.1: Arquitetura da FPGA [7].

Suportam diversos padrões de I/O. Para aplicações de velocidade relativamente baixa, os mais comuns são TTL (*Transistor-Transistor Logic*), LVTTL (*Low Voltage* TTL), CMOS (*Complementary Metal-Oxide Semiconductor*) e LVCMOS (*Low Voltage* CMOS).

Para velocidades mais altas e aplicações específicas, existem padrões mais complexos, como SSTL (*Stub Series Terminated Logic*), HSTL (*High-Speed Transceiver Logic*), LVPECL (*Low-Voltage Positive Emitter-Coupled Logic*) e LVDS (*Low-Voltage Differential Signaling*).

Algumas FPGAs possuem características adicionais como interface PCI *Express*, blocos Ethernet e transmissão serial de alta velocidade.

Várias companhias produzem FPGAs como Xilinx, Actel, Altera, QuickLogic e Atmel [2].

2.3 MICROCONTROLADORES

Um microcontrolador é popularmente conhecido como “computador de um só chip”. Ele possui, em um único encapsulamento, uma unidade central de processamento, memória de programa, memórias auxiliares, sistema de entrada/saída e vários periféricos que variam entre os modelos (por exemplo, conversores A/D, comparadores analógicos, etc.) [8].

2.3.1 Aplicações

Os microcontroladores estão presentes em praticamente todos os produtos modernos, desde fornos de microondas, televisores, aparelhos de som, telefones celulares e até relógios.

Um microcontrolador é utilizado quando é necessário que um circuito realize operações e procedimentos que variam conforme estímulos externos ou condições de um ambiente. Também são empregados em sistemas que efetuam comandos e procedimentos, ou ainda análises e correções. Possuem vantagens como baixo custo e baixo consumo, portabilidade e reconfiguração por *software* [8].

2.3.2 Tecnologias

Atualmente, pode-se encontrar duas categorias distintas de µCs (microcontroladores) que são RISC (*Reduced Instruction Set Computer*) e CISC (*Complex Instruction Set Computer*).

Nos últimos anos da década de 1970, a técnica dominada para a construção de processadores privilegiava as instruções muito complexas, cuja implementação era muito simples com o emprego do intepretador, os processadores CISC [8].

Em 1980, um grupo liderado por David Patterson e Carlo Séquin, em Berkeley, desenvolveu chips processadores VLSI (*Very Large Scale Integration*) que não usavam interpretação, os chamados processadores RISC. Este chip foi imediatamente seguido pelo RISC II, projetado pelos mesmos pesquisadores. Em 1981, John Henessy, pesquisador de Stanford, projetou um chip chamado MIPS, que transformou-se em um sucesso comercial, gerando produtos famosos como o chip SPARC e o MIPS. Estes processadores utilizavam instruções simples, executadas diretamente pelo *hardware*, nenhuma delas eram interpretadas, assim podiam ser executadas rapidamente.

Na época em que esses processadores simples estavam sendo projetados, o número de instruções chamava a atenção, em torno de 50, um número bem menor do que as 200 ou 300 dos processadores CISC.

Os processadores CISC têm a vantagem de reduzir o tamanho do código executável por já possuírem muito do código comum em vários programas, em forma de uma única instrução. Porém como as instruções nos processadores RISC são executadas diretamente pelo *hardware*, sendo que nenhuma delas é interpretada por micro instruções, a eliminação de um nível de interpretação resulta em uma velocidade de execução bastante rápida para a maioria das instruções [8].

2.4 TÉCNICAS DE VERIFICAÇÃO FUNCIONAL

Verificação é o processo de comparar se a inplementação de um projeto é equivalente a sua especificação. A verificação funcional é uma das tarefas que mais consume tempo no desenvolvimento de um projeto. Atualmente, um sistema completo pode ser integrado em um único chip, utilizando milhões de portas lógicas. Essa complexidade torna a verificação funcional um ponto crítico no projeto, consumindo mais da metade dos recursos disponíveis em sua implementação [3].

Devido a complexidade dos circuitos integrados digitais, não existe uma técnica de verificação aplicável a todas as classes de CIs. Assim, essas técnicas são divididas em dois grandes grupos: estático e dinâmico, também conhecido como técnicas de verificação formal e informal (ou baseada em simulação).

2.4.1 Verificação Estática ou Verificação Formal

Essa técnica utiliza métodos matemáticos formais para verificar a funcionalidade do projeto. É denominada estática, porque suas técnicas não requerem estímulos nem simulação para verificar o comportamento do projeto. Os engenheiros de verificação não presisam devenvolver um *testbench*. A desvantagem é que tem limitações e não pode substituir completamente a simulação, deve ser utilizada para complementá-la.

A verificação funcional formal é dividida em duas classes: uma que compara o projeto com sua especificação, e outra que verifica a equivalência entre um modelo já testado (*golden model*) e o modelo sob verificação. São chamadas de *intent verification* e *equivalence checking*.

2.4.1.1 *Intent Verification*

Duas técnicas conhecidas estão nesta classe que são *Model Checking* e *Theorem Proving*.

Model Checking gera todas as entradas possíveis ao modelo em verificação, sob todos os estados possíveis, e compara se as propriedades do modelo, originado a partir das especificações, são consistentes. Esta técnica é adquada para blocos pequenos devido ao crescimento exponecial de sua complexidade. Apresenta uma cobertura na verificação funcional eficaz, porém depende de propriedades que devem ser cuidadosamente construídas.

Theorem Proving descreve o modelo a ser verificado e a especificação de uma maneira formal, envolvendo axiomas, propriedades e regras de inferência. A verificação é atingida

provando as propriedades, usando para isso, os axiomas e regras de inferência. A eficácia deste método depende da qualidade na transcrição da especificação em propriedades. Este método não é limitado ao número de entradas ou estado possíveis.

2.4.1.2 E*quivalence Checking*

Compara a equivalência entre modelo sob verificação e o *golden model,* que é o modelo já verificado. Esta técnica converte ambos para uma representação lógica formal e compara se as representações são iguais. Não existem propriedades a serem criadas a partir da especificação, ao invés disso o *golden model* é usado. O método de *equivalence checking* é disponibilizado por diversas ferramentas comerciais, como o Texudo.

2.4.2 Verificação Dinâmica

Esta técnica verifica o projeto através de simulação utilizando um conjunto de testes. Os resultados obtidos são comparados com os valores esperados (determinados como corretos). Utiliza *testbench*, estímulos e resultados para comparação, criados manualmente pelo engenheiro de verificação. A verificação dinâmica também é dividida em *intent verification* e *equivalence verification*.

2.4.2.1 *Intent Verification*

A especificação é utilizada como referencia para a criação de padrões de verificação. É um processo demorado, pois cada declaração na especificação é verficada por algum padrão de teste, e mesmo que o número de padrões examinados cresça rapidamente isso não significa que o projeto está totalmente verificado.

2.4.2.2 E*quivalence Checking*

Compara a equivalência entre os resultados obtidos no modelo sob verificação e os obtidos do *golden model*. Neste caso para aumentar a cobertura, estímulos peseudo-aleatórios podem ser utilizados para complementar os estímulos determinísticos. É comum ser utilizada para determinar problemas específicos no projeto. Algumas vezes o modelo é construído em *hardware* através de FPGAs para a verificação aleatória mais rápida, contudo a maioria dos

métodos convencionais basea-se apenas em *testbench*. Normalmente *equivalence checking* é utilizado para verificação de níveis mais baixos (*gate-level*), comparando com o *golden model* em RTL.

2.5 ETHERNET

Ethernet é uma tecnologia de interconexão para redes locais ou LAN (*Local Area Network*), baseada no envio de quadros.

2.5.1 Princípios Básicos

Atualmente, a camada de rede mais popular para LAN é a Ethernet (IEEE 802.3), que pode ser apresentada de muitos modos e formatos. Essa rede pode ter uma topologia em estrela, anel (*token ring*) ou em barramento, pode usar para transmissão cabo coaxial, fio de cobre de par trançado ou fibra ótica. Além disso, pode transmitir dados em diferentes velocidades, 10 Mbps, 100 Mbps e 1 Gbps [10].

O IEEE 802.3 define a banda máxima possível, as interfaces físicas e o formato do *frame* Ethernet. A rede é dividida em quatro subcamadas (figura 2.2):

- Especificação de mídia;
- Subcamada física;
- Subcamada de controle de acesso ao meio;
- Subcamada de controle lógico do link.

A Ethernet utiliza um método de controle de acesso ao meio, com detecção de colisão, chamado CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) [11].

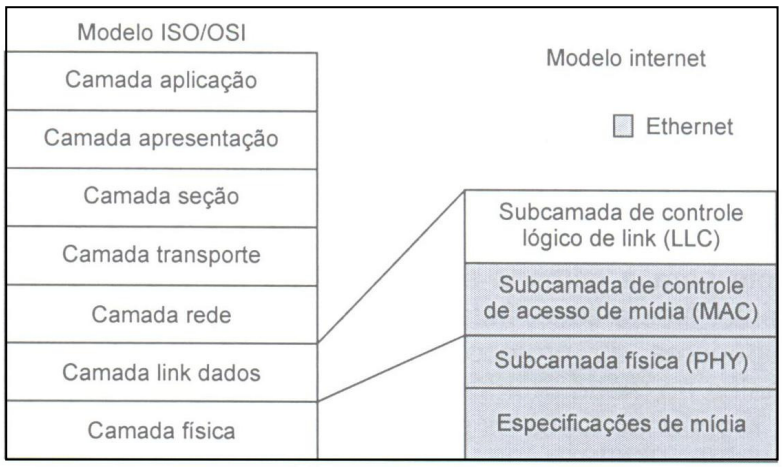

Figura 2.2 – Subcamadas da camada de rede [11].

2.5.2 Histórico

A Ethernet foi criada pela Xerox *Corporation*, nos anos 70. Uma breve descrição apareceu no artigo *Ethernet: Distributed Packet Switching for Local Computer Networks*, em julho de 1976, em uma publicação da *Association for Computing Machinery*. O artigo foi escrito por Robert M. Metcalfe e David R. Boggs, da Xerox.

Em 1980, a DEC (*Digital Equipment Corporation*), a Intel Corporation e a Xerox Corporation formalizaram as descrições da Ethernet em um documento chamado *The Ethernet, a Local Area Network: Data Link Layer and Physical Layer Specifications*.

Outro nome para esse padrão é DIX Ethernet, referente às iniciais das empresas envolvidas.

Em 1985, o IEEE realizou sua própria edição sobre o padrão, que é bastante similar à interface DIX e compatível com a mesma [12].

2.5.3 Quadros

Todos os dados em uma rede Ethernet trafegam em estruturas chamadas *frames*. Um *frame* Ethernet tem campos definidos para os dados, destino, origem e informação para determinação da integridade dos dados. Os campos em um IEEE 802.3 *frame* Ethernet são mostrados na tabela 1.

Tabela 2.1: Campos do *frame* Ethernet [10].

Preâmbulo	Endereço de destino	Endereço de origem	Tamanho ou Tipo	Dados	CRC
8 bytes	6 bytes	6 bytes	2 bytes	46 a 1500 bytes	4 bytes

- Preâmbulo: O quadro Etherenet começa com um campo preâmbulo de 8 bytes. Cada um dos primeiros 7 bytes do preâmbulo tem um valor de 10101010, que servem para “despertar” os adaptadores receptores (A) e sincronizar o relógio deles com o relógio do remetente (B), para que as velocidades de transmissão sejam compatíveis. O último byte é 10101011, onde os últimos dois bits são 1s para alertar o adaptador B que algo relevante está chegando; quando o hospedeiro B recebe os dois 1s consecutivos, ele sabe que os próximos 6 bytes são o endereço de destino.

- Endereço de destino: Esse campo de 6 bytes contêm o endereço do adaptador de destino. Se o adaptador que recebe o quadro verifica que não é seu endereço nem o endereço de *broadcast* da LAN, ele descarta o quadro.
- Endereço de origem: Esse campo de 6 bytes contém o endereço de LAN do adaptador que transmite o quadro para a LAN.
- Tamanho ou Tipo: Se o valor for 500 (05DCh) ou menos, o campo se refere ao tamanho do campo de dados em bytes. Se for 1536 (0600h) ou mais, refere-se ao protocolo utilizado no campo de dados.
- Dados: Pode conter entre 46 e 1500 bytes. Se houver menos que 46 bytes, então o campo será preenchido de modo a completar o valor mínimo. Esse campo também carrega o datagrama IP (quando o protocolo IP é utilizado), que indica se o pacote deve ser fragmentado ou não.
- CRC (*Cyclic Redundance Check*): A finalidade desse campo de 4 bytes é permitir que o adaptador receptor detecte se algum erro foi introduzido no quadro. Esses erros podem ocorrer devido à atenuação do sinal ou à interferência eletromagnética. A detecção do erro é feita da seguinte forma: o hospedeiro A calcula o CRC, com base nos bits contidos no quadro, e então faz o envio do mesmo. Quando o hospedeiro B recebe o quadro, calcula novamente o CRC e compara com o valor indicado no campo CRC. Se forem diferentes é porque houve algum erro durante a transmissão [10, 12].

2.5.4 Controle de Acesso ao Meio

Conforme mencionado anteriormente, em redes que usam interfaces half-duplex, uma interface de cada vez pode transmitir. Por isso, as interfaces precisam de uma maneira de decidir quem irá transmitir. O MAC (*Media Access Control*) é responsável por esse controle de transmissão.

A Ethernet usa um método de controle de acesso ao meio chamado CSMA/CD, que é um protocolo de acesso múltiplo. Esse método permite que qualquer interface transmita quando a rede estiver ociosa. Se duas interfaces tentam transmitir ao mesmo tempo, ambas devem esperar um tempo aleatório (*Backoff*) e então tentam transmitir novamente. No caso de colisão durante o envio, assim que detectada a colisão, a transmissão é interrompida e ambas devem esperar o tempo de *Backoff* para retransmitir, como mostrado no fluxograma da figura 2.3 [12].

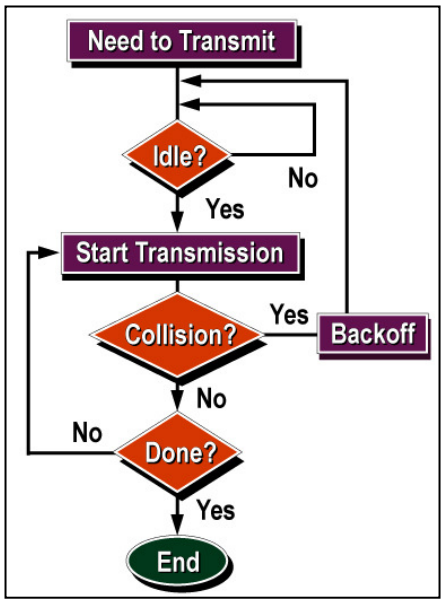

Figura 2.3: Fluxograma do CSMA/CD.

2.6 WISHBONE - SoC *Interconnection Architecture for Portable IP Cores*

WISHBONE é uma metodologia desenvolvida para ser utilizada com IP *cores* semicondutores. Tem o objetivo de promover a reutilização do projeto, diminuindo problemas na integração de SoC. Isto é realizado através da criação de uma interface comum entre IP *cores*. Assim obtem-se maior portabilidade e confiabilidade no sistema, além do usuário conseguir finalizar seu sistema em menos tempo [13].

Anteriormente, IP *cores* não utilizavam esquemas de interconexão padrão, o que dificultava a integração, tendo surgido então a necessidade de criar uma lógica para conectá-los.

Este padrão não requer uma ferramenta ou *hardware* específico para seu desenvolvimento. Além disso, é compatível com todas as ferramentas de síntese.

O WISHBONE não possui direitos autorais, é de domínio público. Assim pode ser livremente copiado e utilizado em projetos sem encargos.

A arquitetura WISHBONE foi fortemente influenciada por três fatores. O primeiro, a necessidade de uma boa solução para a integração confiável entre SoCs. Segundo, a necessidade de uma especificação de interface comum para facilitar o *design* estruturado em grandes projetos conectados. Terceiro, foi influenciada por soluções tradicionais de integração de sistemas disponibilizadas para microcomputadores, como barramento PCI e VMEbus.

Este padrão para conectar *cores* é feito para que *desingners* combinem projetos em Verilog, VHDL ou qualquer outra linguagem de descrição de *hardware*.

Possui arquitetuta *Master/Slave* para que os projetos conectados tenham maior flexibilidade. Pode ser definido para que o barramento tenha 8, 16, 32 ou 64 bits. WISHBONE é destinado a ser um "barramento lógico". Não especifica informação elétrica ou topologia do barramento. Ao invés disso, a especificação é escrita em termos de sinais, ciclos de *clock* e níveis lógicos, alto e baixo.

Os sinais utilizados no protocolo de interconexão são especificados no anexo 1. Alguns desses sinais são opcionais, e podem ou não estar presentes em uma interface. A figura 2.4 apresenta um diagrama com a conexão em o módulo *Master* e *Slave*.

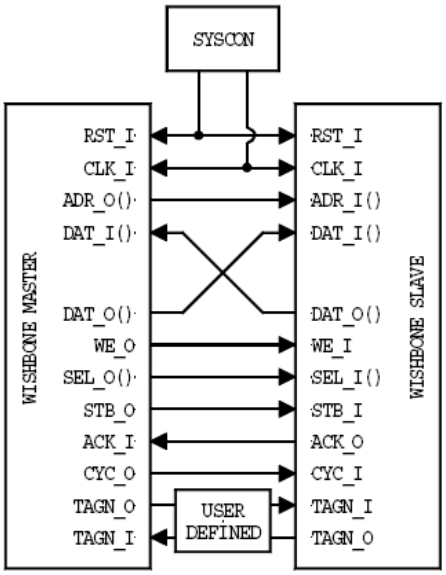

Figura 2.4: Conexão padrão WISHBONE [13].

CAPÍTULO 3

METODOLOGIA

Este capítulo descreve o método complementar desenvolvido para a verificação e a avaliação de um microcontrolador de código aberto.

Inicialmente, os microcontroladores disponíveis foram pesquisados para que uma seleção fosse realizada. Posteriormente, desenvolveu-se um bloco em linguagem de descrição de *hardware* para fazer a interface e o controle entre o microcontrolador e um IP *Core* Ethernet.

O protocolo implementado nesse bloco realiza a configuração dos registradores Ethernet para estabelecimento de uma conexão. Os *frames* recebidos pelo IP *Core* Ethernet são armazenados em uma memória, que posteriormente será utilizada como memória de programa.

3.1 PESQUISA E SELEÇÃO DO MICROCONTROLADOR

Na primeira parte do projeto foi realizada uma pesquisa no site do OpenCores para selecionar o microcontrolador a ser utilizado.

O OpenCores disponibiliza vários projetos, como microcontroladores, memórias, *system controllers, video controllers*, SoC, entre outros, que tem o código aberto descrito em VHDL ou Verilog.

O objetivo era selecionar o microcontrolador de oito bits mais completo que possuísse um componente equivalente no mercado para comparação do funcionamento, documentação descrevendo as funções implementadas e as limitações do projeto, *testbench*, *software* para conversão dos dados que serão colocados na memória de programa, preferencialmente descrito em VHDL, pois se tinha mais conhecimento da linguagem, e finalmente que fosse um projeto com atualizações recentes. Nas tabelas abaixo são apresentados todos os projetos pesquisados e suas descrições.

Na tabela 3.1 são listados os 33 microcontroladores pesquisados e seus respectivos modelos comerciais. Porém, em alguns casos o µC não possuía um modelo equivalente no mercado como o projeto JOP, por exemplo, que é a implementação de uma JMV (*Java Virtual Machine*) como uma máquina em *hardware*.

O projeto CPUgen gera processadores RISC customizáveis. Ele permite a determinação do tamanho dos barramentos de endereços, dados e instruções. Alem disso, é possível decidir se o processador suportará interrupção e endereçamento indireto.

Além destes, é possível observar outros microcontroladores sem modelos comerciais compatíveis como o Marca, Mlite, Tiny64 e OpenRisc1000.

Tabela 3.1: Arquitetura, modelo comercial equivalente e fabricante (do componente de mercado) dos microcontroladores pesquisados no OpenCores.

Projeto	Bits	Modelo Comercial	Fabricante
68hc05	8	mc68hc05	Motorola
68hc08	8	mc68hc08	Motorola/ Freescale
AVR_CORE	8	ATmega103	Atmel
JOP	-	-	-
RISCMCU	8	90S1200	Atmel
System09	8	6809	Motorola
System11	8	68HC11	Motorola
System68	8	6800	Motorola
T400	4	COP420/421 e COP410L/411L	National Semiconductor
Marca	16	-	-
Risc5x	12	PIC 12 bit opcode	Microchip
Mlite	8	-	-
CPUgen	-	-	-
T51	8	8052/8032	Intel
T48 µController	8	MCS-48	Intel
Tiny64	64	-	-
MiniMIPS	32	MIPS I	-
T80	8	Z80 e 8080	Zilog/Intel
AX8	8	90S1200 e 90S2313	Atmel
Risc16f84	8	PIC 16f84	Microchip
PPX16 mcu	8	PIC 16C55 e 16F84	Microchip
TV80	8	Z80	Zilog
AE18	8	PIC18	Microchip
C16	8	8080	Intel
Wishbone *High Performance* Z80	8	Z80	Zilog
Aquarius	16	CPU RISC SuperH-2 ISA	Motorola
OpenRISC 1000	32/64	-	-
8051 *core*	8	8051	Intel
MiniRisc	8	PIC 16C57	Microchip
YACC	32	MIPS I	-
S1 *Core*	64	Reduced version of the OpenSPARC T1	Sun Microsystems
CPU8080	8	8080	Intel
aeMB	32	Microblaze	Xilinx

Um dos principais critérios considerados para escolha do microcontrolador diz respeito à sua documentação. Era importante conhecer detalhes sobre o projeto, como limitações, diferenças do modelo comercial, informações sobre os blocos implementados, etc. Alguns projetos não disponibilizavam documentação, enquanto outros apresentavam documentação completa. Em outros casos, o projeto continha apenas o conjunto de instruções suportadas pelo microcontrolador, como, por exemplo, o T51 (tabela 3.2).

Outra informação importante é o *testbench*, pois desta maneira é possível entender melhor o funcionamento do *core*, visualizar as ondas de simulação e verificar sinais e valores.

Apesar da maioria dos projetos possuírem *testbench,* alguns são implementados em *Assembly* e não em linguagem de descrição de *hardware*. Estes realizam testes como impressão de mensagens na porta serial, calculadoras, leitura de uma seção de memória, etc., como é o caso dos projetos JOP, RISCMCU, System09, T80, AX8, entre outros.

Também havia preferência por um projeto descrito em VHDL, pois tinha-se mais conhecimento da linguagem comparando-se com Verilog.

Tabela 3.2: Documentação e *testbench* dos microcontroladores pesquisados no OpenCores**.**

Projeto	Documentação	*Testbench*	Linguagem
68hc05	Não	Não	VHDL
68hc08	Não	Não	VHDL
AVR_CORE	Sim	Não	VHDL
JOP	Sim	Sim	VHDL
RISCMCU	Sim	Sim	VHDL
System09	Sim	Sim	VHDL
System11	Não	Sim	VHDL
System68	Sim	Sim	VHDL
T400	Sim	Sim	VHDL
Marca	Sim	Não	VHDL
Risc5x	Sim	Sim	VHDL
Mlite	Não	Sim	VHDL
CPUgen	Sim	Sim	VHDL
T51	Sim	Sim	VHDL
T48 µController	Sim	Sim	VHDL
Tiny64	Não	Sim	VHDL
MiniMIPS	Sim	Sim	VHDL
T80	Não	Sim	VHDL
AX8	Não	Sim	VHDL
Risc16f84	Sim	Não	Verilog
PPX16 mcu	Não	sim	Verilog
TV80	Sim	Sim	Verilog
AE18	Sim	Sim	Verilog

Projeto	Documentação	*Testbench*	Linguagem
C16	Sim	Sim	Verilog
Wishbone *High Performance* Z80	Sim	Sim	Verilog
Aquarius	Sim	Sim	Verilog
OpenRISC 1000	Sim	Sim	Verilog
8051 *core*	Sim	Sim	Verilog
MiniRisc	Sim	Sim	Verilog
YACC	Sim	Sim	Verilog
S1 *Core*	Sim	Sim	Verilog
CPU8080	Sim	Sim	Verilog
aeMB	Sim	Sim	Verilog

A tabela 3.3 apresenta os projetos que possuem *software* para conversão do código implementado, já no formato hexadecimal, em um formato compatível para compor a memória de programa. Em alguns projetos, como o AVR_CORE, por exemplo, o conversor gera, a partir de um arquivo hexadecimal, um arquivo em VHDL contendo a entidade e a arquitetura da memória PROM.

As atualizações realizadas nos projetos também foram consideras. Foi dada preferência aos projetos mais modernos. Algumas atualizações eram referentes a correções de erros e outras a incrementos na implementação dos blocos.

Tabela 3.3: *Software* de conversão e atualizações dos microcontroladores pesquisados no OpenCores.

Projeto	*Software* de Conversão	*Última atualização
68hc05	Não	1 mês
68hc08	Não	1 mês
AVR_CORE	Sim	1 mês
JOP	Sim	1 semana
RISCMCU	Sim	2 anos
System09	Sim	1 ano
System11	Sim	3 anos
System68	Sim	1 ano
T400	Sim	1 ano
Marca	Não	3 meses
Risc5x	Sim	2 anos
Mlite	Sim	1 mês
CPUgen	Não	3 anos
T51	Sim	2 meses
T48 µController	Sim	6 meses
Tiny64	Sim	1 mês

Projeto	*Software* de Conversão	*Última atualização
MiniMIPS	Sim	1 ano
T80	Sim	1 ano
AX8	Sim	1 ano
Risc16f84	Sim	1 ano
PPX16 mcu	Sim	5 meses
TV80	Não	1 ano
AE18	Não	1 mês
C16	Sim	1 ano
Wishbone *High Performance* Z80	Não	2 anos
Aquarius	Sim	2 anos
OpenRISC 1000	Sim	4 meses
8051 *core*	Sim	2 anos
MiniRisc	Sim	1 mês
YACC	Sim	2 anos
S1 *Core*	Sim	1 mês
CPU8080	Sim	7 meses
aeMB	Não	1 mês

* A pesquisa no site OpenCores foi realizada nos meses de abril e maio de 2007

Entre todos os *cores* pesquisados, foram selecionados quatro modelos, sendo que para cada modelo existe um ou mais projetos correspondentes. Fez-se uma pesquisa mais detalhada na documentação, código e um breve teste de funcionamento, para só então escolher um único microcontrolador. Os quatro selecionados foram:

- Intel 8051
- Zilog Z80
- Motorola 6805
- Atmel AVR

3.1.1 Intel 8051 - Projetos T51 e 8051 *core* (figura 3.1).

- Características [14]:
  - Arquitetura: 8-bit
  - Memórias:
    - 4 Kbytes de memória ROM interna
    - 128 bytes de memória RAM interna
    - 8 Kbytes de memória PROM interna

- Até 64Kbytes de memória RAM externa
- Até 64Kbytes de memória ROM externa

o Freqüência de Operação: 1-12MHz

o Interrupções: 5 fontes de interrupção com 2 níveis de prioridade

o 2 temporizadores/contadores de 8/16 bits com 4 modos de operação cada um

o 32 linhas de E/S bidirecionais e individualmente endereçáveis

- Modos de endereçamento:
  - o Direto
  - o Indireto
  - o Por registradores
  - o Por registrador específico
  - o Imediato
- Número de instruções: 111
- Compilador: SDCC

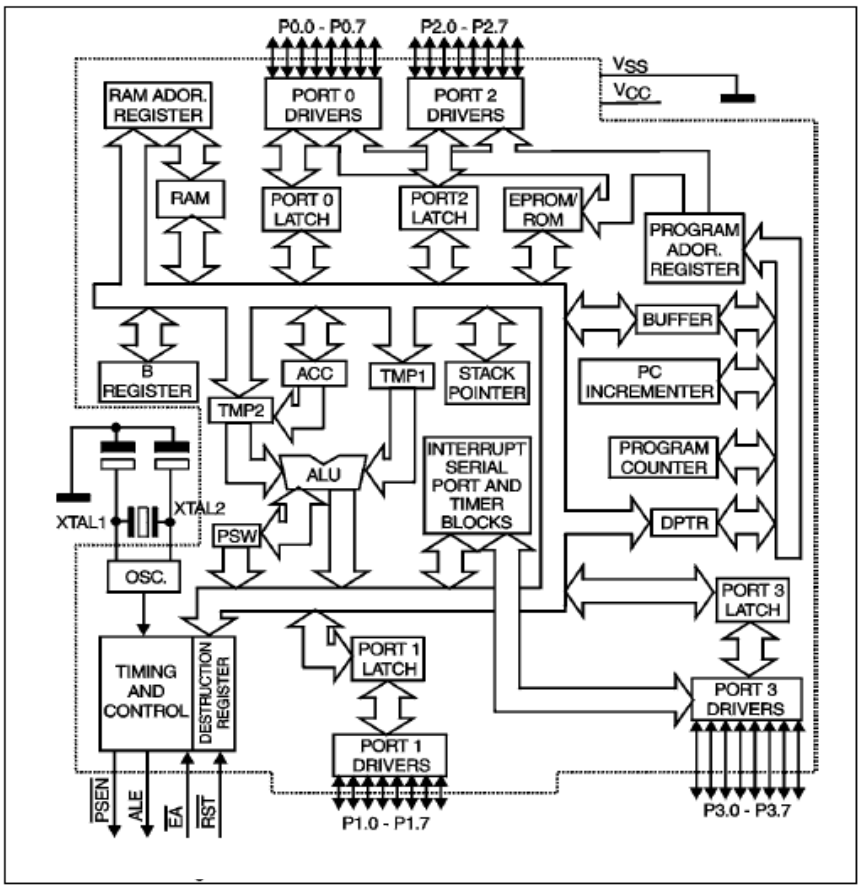

Figura 3.1: Arquitetura interna do modelo comercial do microcontrolador 8051.

3.1.2 Zilog Z80 – Projetos T80 e TV80 (figura 3.2).

- Características [15]:

- Arquitetura: 8-bit
- Memórias:
  - 32 Kbytes de memória ROM
  - 32 Kbytes de memória RAM
- Freqüência de Operação: 6-20 MHz
- Interrupções: 3 fontes de interrupção
- 17 registradores internos

- Modos de endereçamento:
  - Direto
  - Indireto
  - Imediato
  - Indexado
- Número de instruções: 158
- Compilador: SDCC

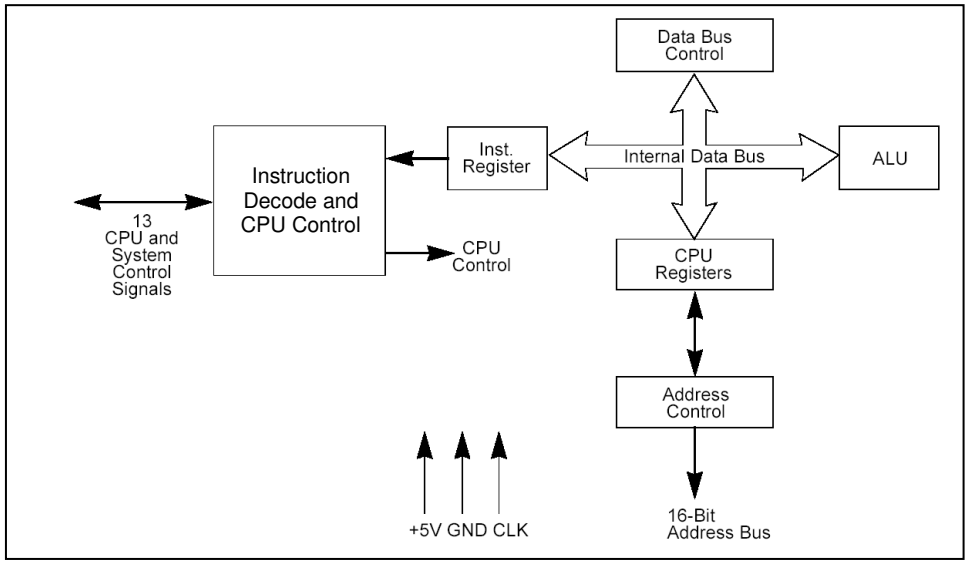

Figura 3.2: Arquitetura interna do modelo comercial do microcontrolador Z80.

3.1.3 Motorola 6805 – Projeto 68hc05 (figura 3.3)

- Características:
  - Arquitetura: 8-bit [16]
  - Memórias:
    - 5936 bytes de memória ROM
    - 176 bytes de memória RAM
    - 256 bytes de memória EEPROM

- Freqüência de Operação: 2.1 MHz
- Interrupções: 5 fontes de interrupção
- Interface de comunicação serial
- *Watchdog timer*
- Conversor A/D

- Modos de endereçamento:
  - Direto
  - Imediato
  - In,dexado
  - Estendido
- Número de instruções: 62
- Compilador: Hi-tech

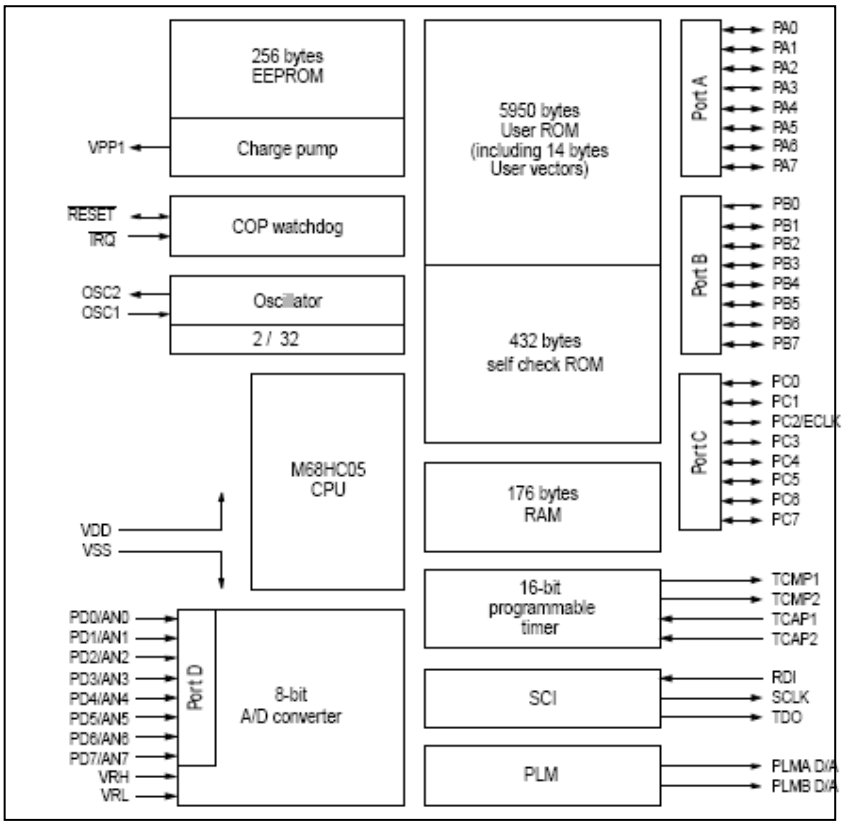

Figura 3.3: Arquitetura interna do modelo comercial do microcontrolador 6805.

3.1.4 Atmel AVR (*Automatic Voltage Regulation*) – Projetos RISCMCU, AX8 e AVR_CORE.

- Características do 90S1200 [17]:
  - Arquitetura: 8-bit (figura 3.4)
  - Memórias:
    - 1 Kbyte de memória FLASH

        - 64 bytes de memória EEPROM
    - Freqüência de Operação: 1-12 MHz
    - Interrupções: interna e externa
    - 1 temporizador/contador de 8 bits
    - Interface de comunicação serial
    - *Watchdog timer*
- Modos de endereçamento:
    - Direto, único registrador
    - Direto, dois registradores
    - Indireto
- Número de instruções: 89
- Compilador: AVR_GCC

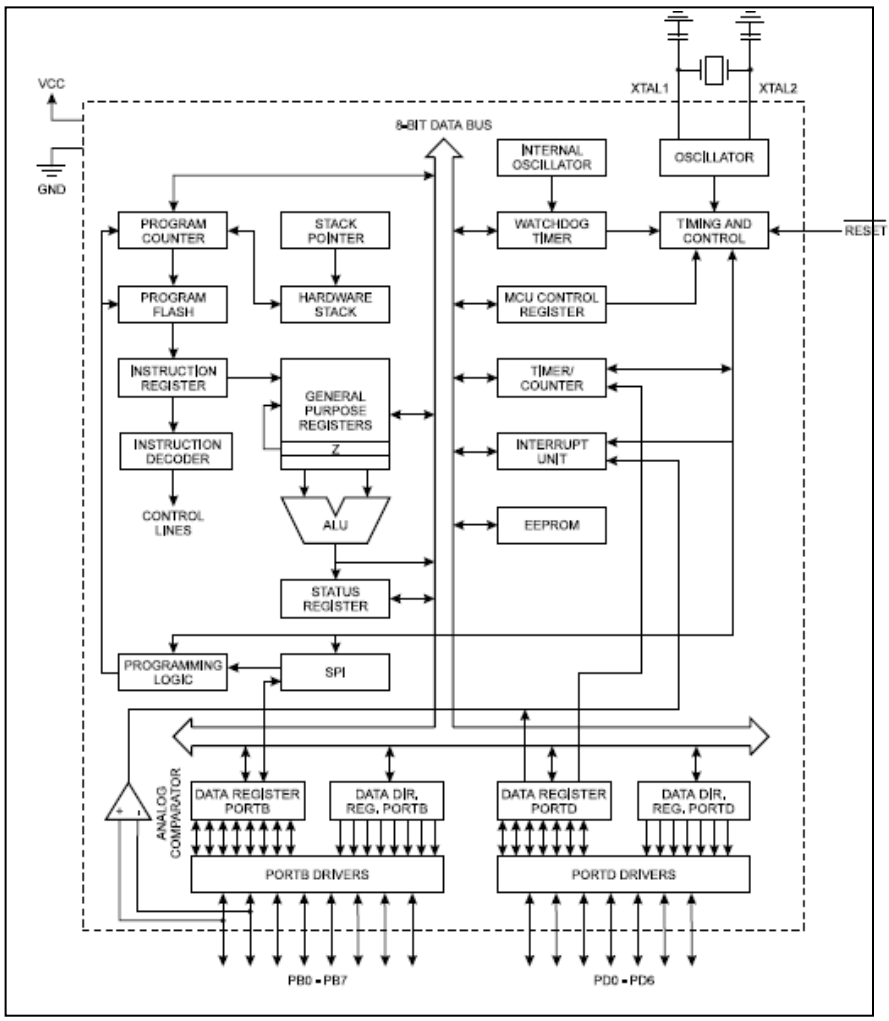

Figura 3.4: Arquitetura interna do modelo comercial do microcontrolador 90S1200**.**

- Características do 90S2313 [18]:
  - Arquitetura: 8-bit (figura 3.5).
  - Memórias:
    - 2 Kbytes de memória FLASH
    - 128 bytes de memória EEPROM
    - 128 bytes de memória RAM interna
  - Frequencia de Operação: 1-10 MHz
  - Interrupções: internas e externas
  - 1 temporizador/contador de 8 bits
  - 1 temporizador/contador de 16 bits
  - Interface de comunicação serial
  - *Watchdog timer*
  - PWM
- Modos de endereçamento:
  - Direto, único registrador
  - Direto, dois registradores
  - Direto, I/O
  - Indireto
  - Relativo
- Número de instruções: 118
- Compilador: AVR_GCC

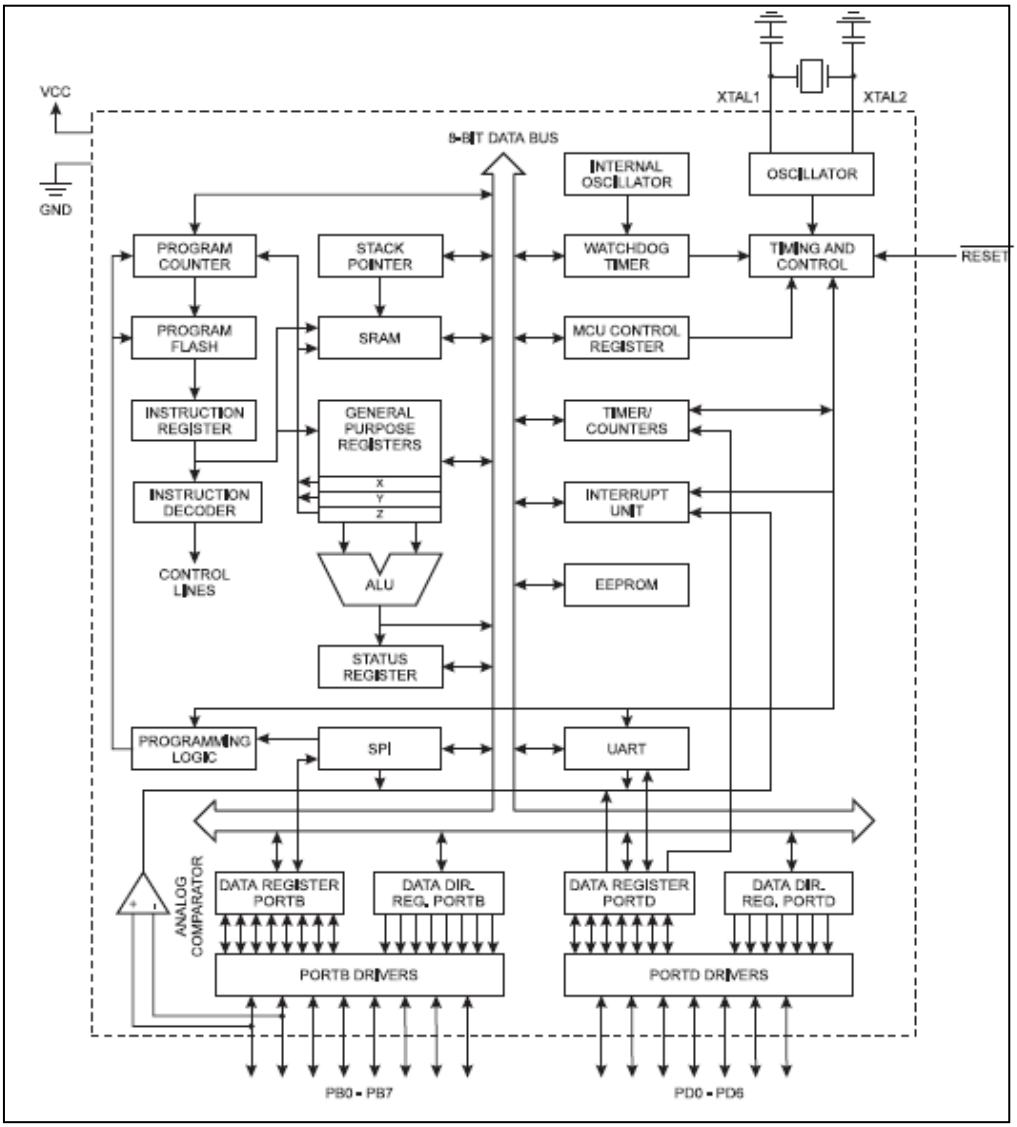

Figura 3.5: Arquitetura interna do modelo comercial do microcontrolador 90S2313.

- Características do ATmega103 [19]:
  - Arquitetura: 8-bit (figura 3.6).
  - Memórias:
    - 128 Kbytes de memória FLASH
    - 4 Kbytes de memória EEPROM
    - 4 Kbytes de memória SRAM interna
  - Freqüência de Operação: 1-10 MHz
  - Interrupções: internas e externas
  - 1 temporizador/contador de 8 bits
  - Interface de comunicação serial
  - *Watchdog timer*
- Modos de endereçamento:

- Direto, único registrador
- Direto, dois registradores
- Direto, I/O
- Indireto
- Relativo

- 32 registradores internos
- Número de instruções: 121
- Compilador: AVR_GCC

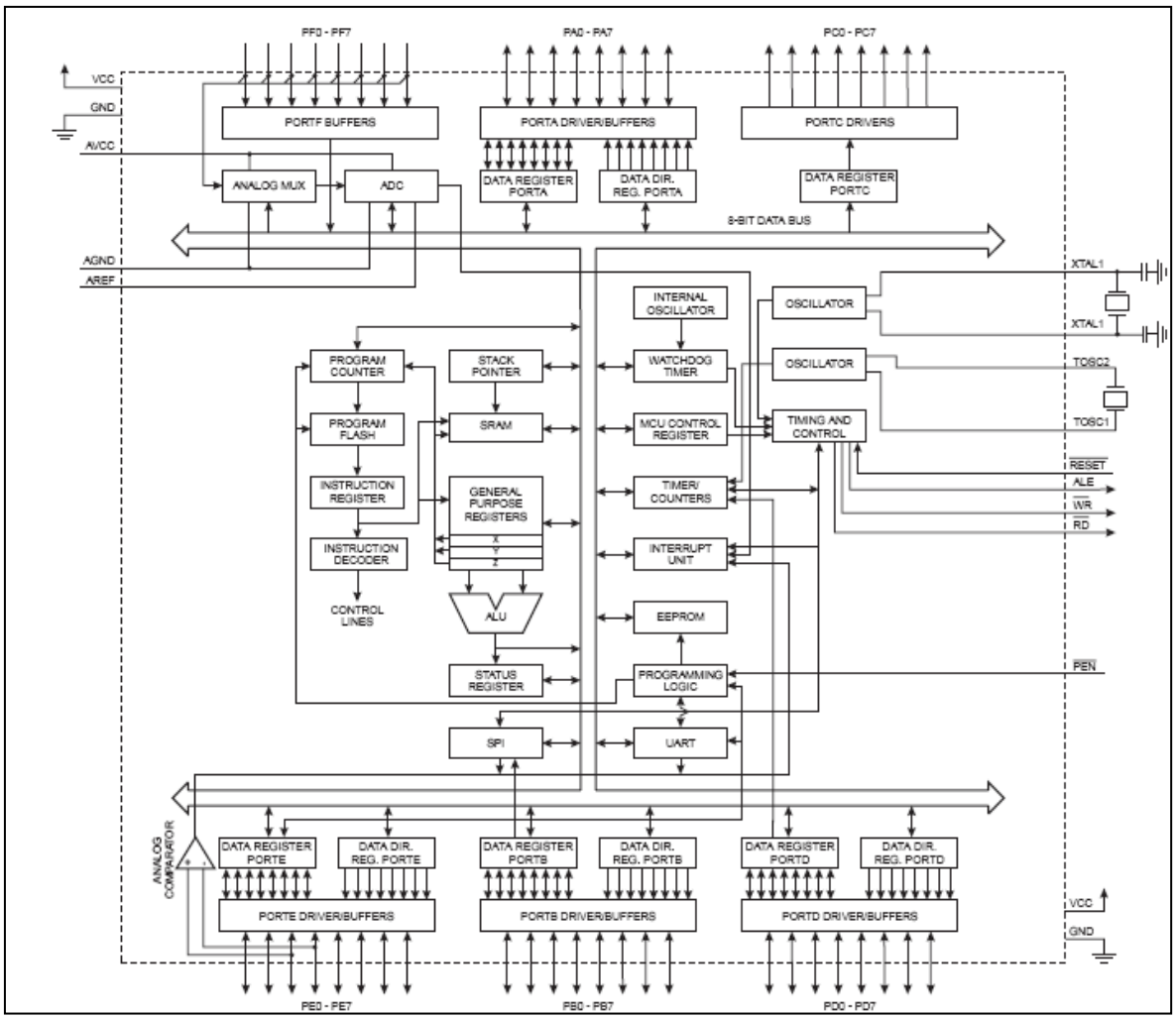

Figura 3.6: Arquitetura interna do modelo comercial do microcontrolador ATmega103.

A tabela 3.4 apresenta um resumo dos projetos pré-selecionados, mostrando informações sobre modelo comercial, documentação, *testbench*, linguagem, *software* de conversão, última atualização e características sobre cada projeto.

Tabela 3.4: Resumo dos projetos pré-selecionados**.**

Projeto	T51	8051 Core	T80	TV80	68hc05	RISCMCU	AX8	AVR_CORE
Arquitetura	8-bit							
Modelo Comercial	8051		Z80		mc68hc05	90S1200	90S2313	ATmega103
Fabricante	Intel		Zilog		Motorola	Atmel		
Documentação	Sim	Sim	Não	Sim	Não	Sim	Não	Sim
Testbench	Sim	Sim	Sim	Sim	Não	Sim	Sim	Não
Software de Conversão	Sim	Sim	Sim	Não	Não	Não	Sim	Sim
Última Atualização	2 meses	2 anos	1 ano	1 ano	1 mês	2 anos	1 ano	1 mês
Memórias	4 Kbytes de ROM interna 128 bytes de RAM interna 8 Kbytes de PROM interna 64Kbytes de RAM externa 64Kbytes de ROM externa		32 Kbytes de memória PROM 32 Kbytes de memória RAM		5936 bytes de memória ROM 176 bytes de memória RAM 256 bytes de memória EEPROM	1 Kbyte de FLASH 64 bytes de EEPROM	2 Kbytes de FLASH 128 bytes de EEPROM 128 bytes de RAM interna	128 Kbytes de FLASH 4 Kbytes de EEPROM 4 Kbytes de SRAM interna
Freqüência de Operação	1-12 MHz		6-20 MHz		2.1 MHz	1-12 MHz	1-10 MHz	
Interrupções	5 estruturas de interrupção		3 estruturas de interrupção		5 estruturas de interrupção	Estruturas de interrupção interna e externa		
Modos de Endereçamento	Direto Indireto Por registradores Por registrador específico Imediato		Direto Indireto Imediato Indexado		Direto Imediato Indexado Estendido	Direto, único registrador Direto, dois registradores Indireto	Direto, único registrador Direto, dois registradores Direto, I/O Indireto Relativo	
Número de Instruções	111		158		62	89	118	121
Compilador	SDCC		SDCC		Hi-tech	AVR_GCC		
Conjunto de Instruções	CISC		CISC		RISC	RISC		
MIPS	1 MIPS/MHZ		1.6 MIPS/MHZ		1 MIPS/MHZ	1 MIPS/MHZ		

O projeto escolhido foi o AVR_CORE, que mesmo não possuindo *testbench*, é um projeto recente, descrito em VHDL, que utiliza pouco espaço da FPGA, possui um compilador gratuito para C e Assembly e um *software* para converter o arquivo em hexadecimal, gerado pelo compilador, em um arquivo em VHDL contendo o código da memória de programa já preenchida.

3.2 AVR_CORE

3.2.1 Estrutura

O projeto é composto de seis blocos principais [20]:

- AVR *core*
- *Program memory*
- *Data memory*
- UART
- *Timer/Counter*
- PORTA e PORTB.

Abaixo são descritas as funcionalidades de cada bloco (figura 3.7):
AVR *core*:
avr_core – Entidade superior do projeto AVR *core*.
alu_avr - Unidade lógica aritmética (ALU).
bit_processor - Alguns bits de operação.
reg_file - Registros.
pm_fetch_dec - Principal parte do *core* (decodificação das instruções, memória e interfaces I/O, PC, etc.).
io_reg_file - Registros de I/O implementados dentro do *core* (SREG, RAMP, SPH, SPL).
io_adr_dec - Decodificador de endereços e multiplicação do barramento de dados para os registros de I/O implementados dentro do *core*.

Microcontrolador:
top_avr_core_sim – Entidade superior do projeto do microcontrolador.
AVRuCPackage - Constantes e tipos.
external_mux – Multiplexador do barramento de dados.
RAMDataReg – Registrador do barramento de dados.

PROM.VHD – Memória de programa .

DataRAM – Memória RAM de dados

portx – *Ports* paralelos (PORTA, PORTB).

Timer_Counter – Temporizador/Contador.

uart – UART.

simple_timer – Temporizador.

Service_Module – Alguns registros de controle adicionais (MCUCR, MCUSR, XDIV, EIMSK, EIFR, EICR).

CPUWaitGenerator – Coloca, automaticamente, o CPU em espera.

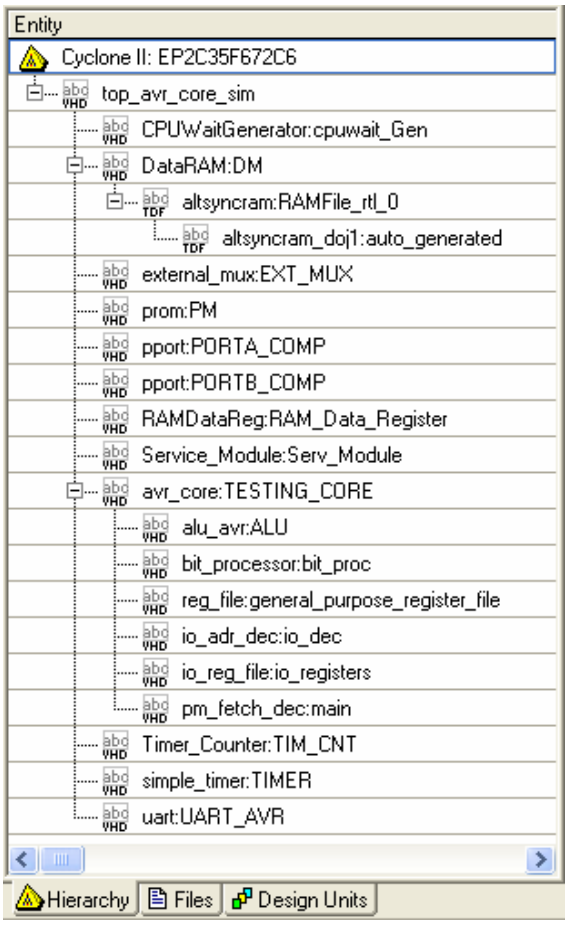

Figura 3.7: Hierarquia dos blocos no AVR_CORE.

3.2.2 Limitações

- O *core* não suporta as instruções SLEEP e CLRWDT.
- Possui somente o *Timer/Counter*0 e *o Timer/ Counter*2 implementados.
- O *Timer/Counter*0 emula o modo síncrono (o *Timer/Counter*0 do microcontrolador

ATmega103 opera com dois sinais de *clock* separados).

- O *Timer/Counter*0 (*Timer/Counter2*) suporta mudanças (*toggling*) do OC0/PWM0 (OC0/PWM2) *output line*, mas não suporta set/reset dessa linha.
- PORTA e PORTB opera somente como interface paralela, e não suporta funções adicionais (OCx/PWMx , #INTx, etc.).

3.2.3 Testes

Foram realizados vários testes no microcontrolador, para os quais utilizou-se diferentes códigos escritos em C e outros em Assembly. A maioria das instruções suportadas foram testadas. Além disso, a seqüência de instruções de um alguns códigos foram executadas e, verificadas, e posteriormente, alteradas para novos testes. Desta maneira, seria possível determinar se alguma combinação de instruções provocaria algum erro durante a execução.

Primeiramente, em um editor de texto, o código em C ou Assembly foi escrito e compilado no AVR_GCC. O arquivo hexadecimal gerado pela compilação foi utilizado como entrada em um conversor que disponibilizava um arquivo em VHDL contendo a entidade e a arquitetura da memória PROM já preenchida. O projeto foi atualizado e a memória regravada, para que o microcontrolador executasse as instruções contidas na mesma, como mostra o modelo da figura 3.8.

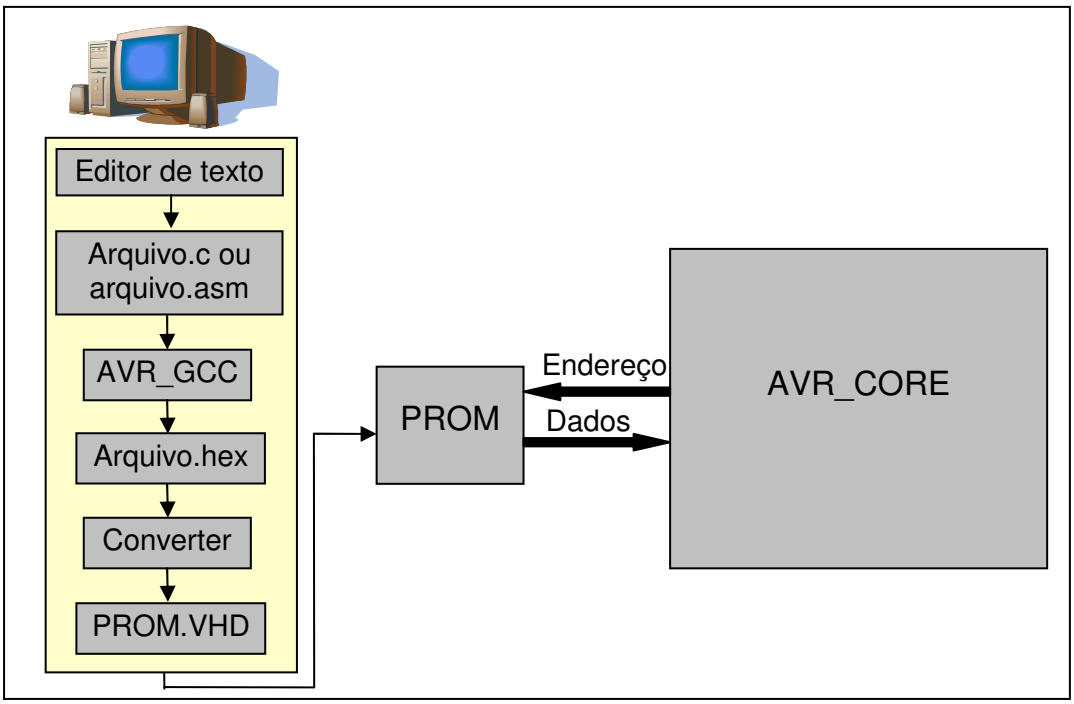

Figura 3.8: Modelo inicial.

Para tornar esses testes independentes de sistema operacional, *design software* e fabricante do dispositivo lógico, utilizou-se uma comunicação Ethernet. A FPGA é gravada uma única vez.

O novo código a ser executado pelo microcontrolador é enviado em *frames* pela interface Ethernet e armazenado em uma memória RAM. Após receber todo o arquivo, o microcontrolador passa a executar o conteúdo desta memória, como mostrado na figura 3.9.

A memória PROM não tem seu conteúdo alterado, sendo que a mesma contém apenas os dados de inicialização para configuração da conexão Ethernet.

Para isso, utilizou-se um IP *Core* Ethernet também disponibilizado pelo OpenCores, que possui o protocolo Ethernet implementado em Verilog.

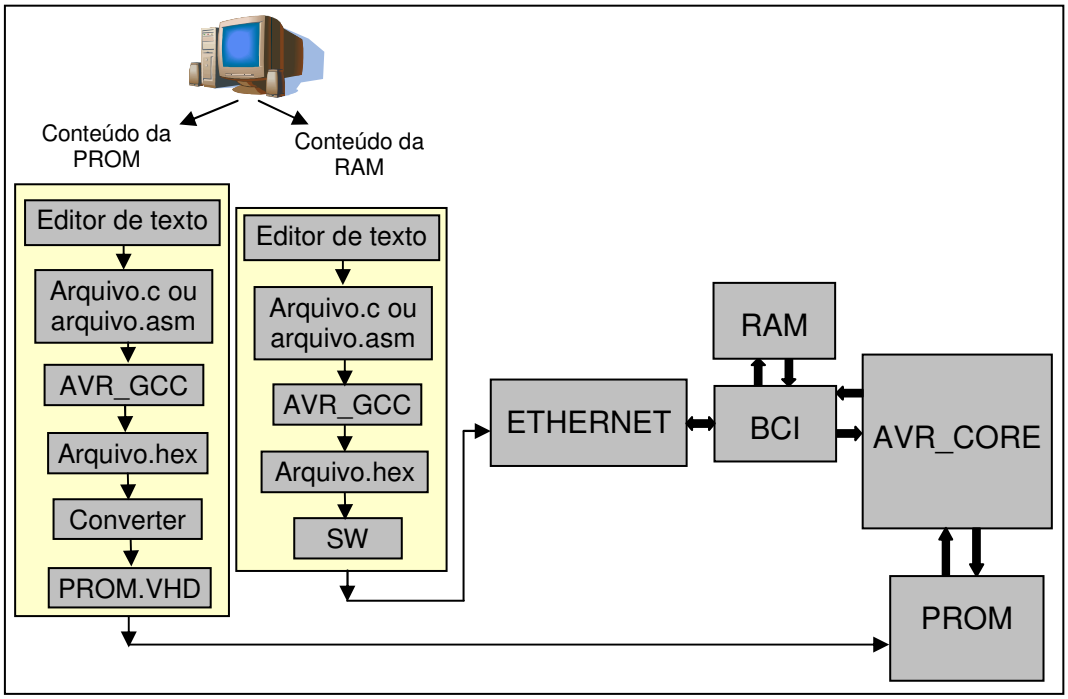

Figura 3.9: Modelo proposto.

A seqüência de operações realizadas é a seguinte: inicialmente o código da memória de programa é implementado. Nele, como mencionado anteriormente, são descritas instruções que realizam a configuração dos registradores Ethernet. Todos os passos para obtenção do arquivo PROM.VHD são os mesmos citados para o modelo inicial, mostrado na figura 3.8.

O projeto completo, contendo o microcontrolador, o bloco BCI (Bloco de Interface e Controle) e o IP *Core* Ethernet são gravados na FPGA.

O bloco BCI faz a interface entre o microcontrolador e o IP *Core* Ethernet, para que estes possam se comunicar adequadamente. Além disso, controla a recepção de *frames*.

Após o estabelecimento da conexão, o AVR_CORE aguarda o recebimento de um *frame*, que será enviado pelo usuário através da interface Ethernet, como apresentado no fluxograma da figura 3.10.

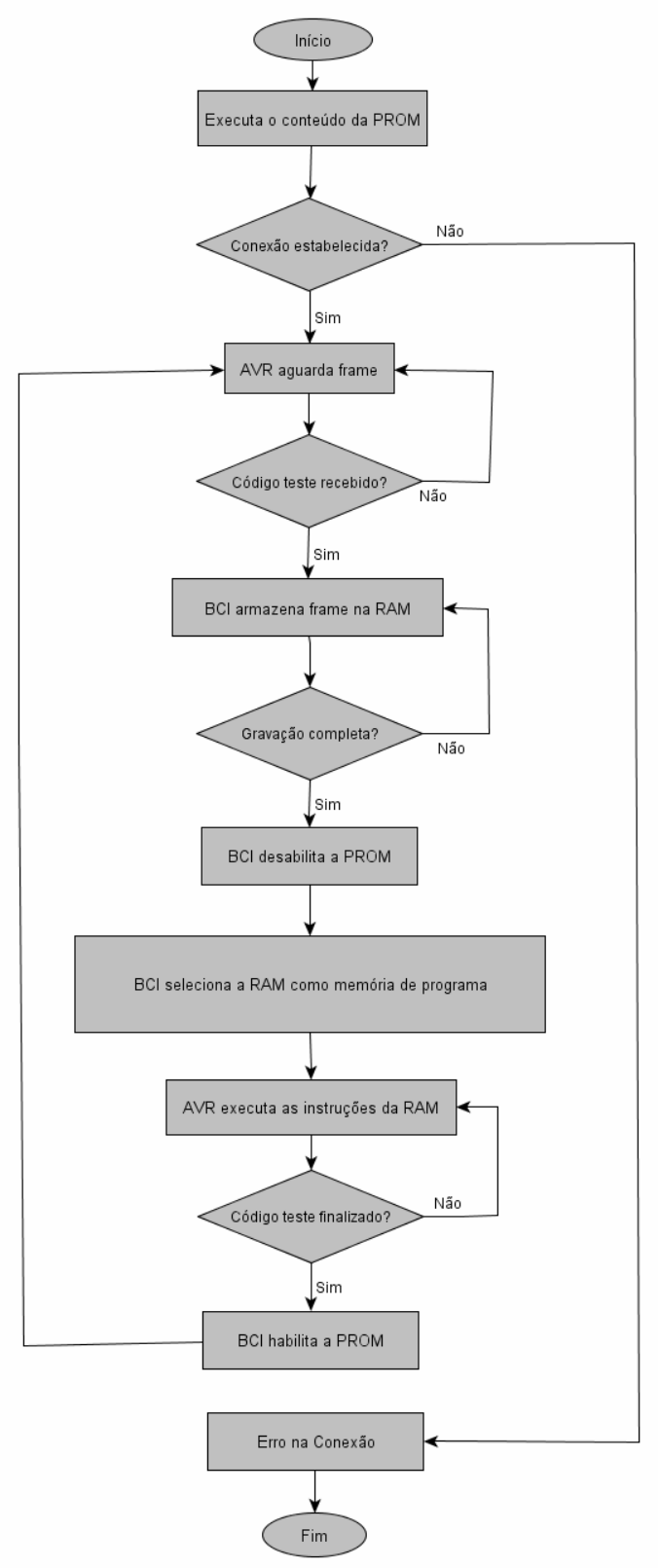

Figura 3.10: Fluxograma do sistema.

O usuário deve seguir a mesma seqüência de operações descrita anteriormente para compilação e conversão do código teste implementado. Após a conversão do arquivo, o usuário utilizará um *software* para realizar o envio, onde será necessário informar endereço MAC de destino e origem.

Quando o *frame*, contendo o código do usuário, for recebido, o BCI realiza o armazento do mesmo em uma memória RAM. Ao final da gravação desta memória, o BCI desabilita a PROM e o AVR passa a executar as instruções gravadas na RAM. Ao fim da execução do código teste, a memória PROM é novamente habilitada.

3.3 IP *Core* Ethernet

O IP *Core* Ethernet é um controlador de acesso ao meio (MAC). Ele conecta o chip PHY Ethernet e o barramento WISHBONE SoC. Este *core* foi projetado para oferecer a maior flexibilidade possível para todos os tipos de aplicação [21].

Características:

- Desempenha as funções da camada MAC do IEEE 802.3 Ethernet
- Geração e checagem automática do CRC de 32 bits
- Geração e remoção do preâmbulo
- Preenchimento automático de *frames*, com tamanho menor que o limite, na transmissão
- Detecção de frames com tamanhos superiores ou inferiores aos limites
- Possibilidade de transmissão de frames acima do tamanho limite
- Suporta *full-duplex*
- Baseado no protocolo CSMA/CD
- Suporta taxas de 10 e 100 Mbps
- Rejeição automática de frames
- Controle de fluxo e geração automática de *frames* de controle no *modo full-duplex*
- Detecção de colisão e auto-retransmissão no modo *half-duplex* (IEEE 802.3x)
- *Status* completo de frames TX/RX
- IEEE 802.3 *Media Independent Interface* (MII)
- Interconexão com WISHBONE SoC
- RAM interna para armazenar 128 TX/RX *buffer descriptors*
- Geração de interrupção em todos os eventos

O IP *Core* Ethernet, figura 3.11, é constituído de sete blocos principais:

Interface WISHBONE: É composta pelas interfaces *master* e *slave* e conecta-se o *core* ao barramento WISHBONE. A interface *master* é utilizada para armazenar os dados do *frame* recebido na memória e carregar os dados, que precisam ser enviados, da memória para o Ethernet *core*.

Módulo de Transmissão: Realiza todas as operações relacionadas à transmissão (geração do preâmbulo, *padding*, CRC, etc.).

Módulo de Recepção: Realiza todas as operações relacionadas à recepção (remoção do preâmbulo, checagem do CRC, etc.).

Módulo de Controle: Desempenha as operações de controle de fluxo quando o Ethernet é usado no modo *full-duplex*.

Módulo MII (*Media Independent Module*): Provê uma interface *Media Independent* para o ch*IP Core Ethernet* PHY externo.

Módulo de *Status*: Guarda diferentes *status* do *buffer descriptor* correspondente ou pode ser utilizado por outro módulo.

Módulo de Registro: Registros que são usados para operações do Ethernet MAC estão neste módulo.

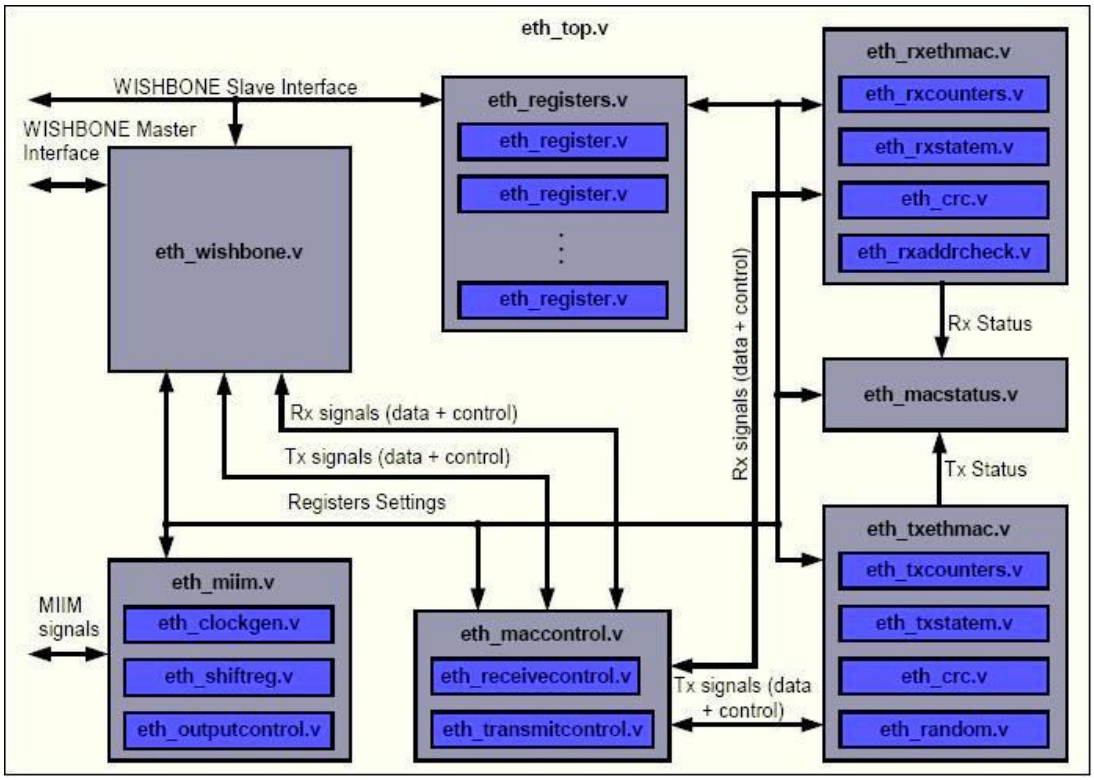

Figura 3.11: Principais blocos do *IP Core Ethernet* [21].

3.3.1 Interfaces do *IP Core Ethernet*

O IP *Core* Ethernet usa três tipos de sinais para conectar ao meio [22]:

- Sinais WISHBONE para conectar a interface servidora
- Sinais MII *Management* para conectar ao PHY
- Sinais de Reset (para reiniciar diferentes partes do IP *Core*)

As tabelas apresentadas no anexo 2 contém os *ports* que conectam o IP *Core* Ethernet à interface servidora, que é o WISHBONE e os sinais de interface com o PHY. Todos os sinais são ativos em nível lógico alto. È apresentada, também no anexo 2, os registros de controle e *status* do IP *Core* Ethernet.

Todos os detalhes de especificação e *design* do IP *Core* Ethernet estão documentados e disponibilizados pelo OpenCores junto com projeto correspondente.

3.4 BCI – Bloco de Controle e Interface

O bloco BCI faz a interface entre o AVR_CORE e o IP *Core* Ethernet. Neste está implementado o protocolo de comunicação entre os blocos, pois o microcontrolador AVR não possui interface WISHBONE.

É também responsável por realizar a configuração inicial dos registradores do IP *Core* Ethernet (velocidade de transmissão/recepção, limites de tamanho dos frames, interrupções, rejeição de frames, etc.). Quando um frame é recebido, efetua a gravação dos dados em uma memória RAM e então faz a troca entre as memórias para que o código do usuário seja executado.

3.4.1 Configuração dos registradores Ethernet

A memória PROM contém o código com os valores a serem gravados nos registradores Ethernet. O *port* A, mostrado na tabela 3.5, é utilizado para controle, e contém os sinais utilizados para o protocolo de escrita/leitura dos registros. O *port* B é para dados.

Tabela 3.5: *Port* de Controle

Start	Fim do programa do usuário	Sinalização de execução correta do código do usuário				Operação	*Stop*
7	6	5	4	3	2	1	0

Como os *ports* possuem 8 bits e os registradores Ethernet são de 32 bits, é necessário passar os valores em quatro partes, como mostra a tabela 3.6. Para isso, foi implementado um protocolo onde o sinal de *start* indica o início da seqüência de dados. Quando é detectado um pulso positivo, sabe-se que os próximos dados disponibilizados no *port* B formaram o endereço, e que as próximas quatro partes, do mesmo *port*, serão referentes ao valor do dado, que será gravado no endereço previamente disponibilizado.

O bit de operação pertencente ao *port* A (*port* de controle) é utilizado para determinar a operação que será realizada. Para escrita, o valor deve ser 1, e para leitura, 0.

Para efetuar a gravação de cada registrador é necessário que o protocolo descrito acima seja repetido, primeiramente com o pulso de start, posteriormete indicação de operação e os valores de endereço e dados no *port* B.

Ao final da gravação de todos os registradores um sinal de *stop* indica a finalização da operação. A partir deste momento, já com a conexão estabelecida, o processador aguarda a chegada de um frame.

Os bits 5 e 6 são de utilização do usuário e serão comentados adiante.

Tabela 3.6: Seqüência mínina para a gravação de um registro Ethernet.

Port A	*Start*	
	Operação	
Port B	*Address* – 1 / 4	MSB
	Address – 2 / 4	
	Address – 3 / 4	
	Address – 4 / 4	LSB
	Data – 1 / 4	MSB
	Data – 2 / 4	
	Data – 3 / 4	
	Data – 4 / 4	LSB
Port A	*Stop*	

O bloco BCI possui uma rotina onde recebe os valores dos *ports* e os coloca em um único registro de 32 bits. Após os registros de endereço e dados serem preenchidos, os sinais de CYC e STB recebem nível lógico alto, indicando o inicio da gravação do registrador Ethernet que, por sua vez, responde com um ACK, levando os sinais de CYC e STB novamente para nível lógico zero e deixando o sistema apto a uma nova operação.

Abaixo são mostradas as ondas de simulação, obtidas utilizando a ferramenta ModelSim – 6.1g *Web Edition*. Porém, nesta versão gratuita não é possível simular um projeto em duas linguagens distintas e neste caso o AVR e o BCI estão descritos em VHDL e o IP *Core* Ethernet, em Verilog. Assim, são apresentadas nas figuras 3.11 e 3.12, duas simulações, uma para cada linguagem.

Na figura 3.12, são apresentados os sinais do protocolo de gravação dos registradores Ethernet. O sinal *vector_ctrl* é referente ao *port* A e *vector* ao *port* B. Inicialmente o valor apresentado no vector_ctrl é 82H, indica um start e operação de escrita. Em seguida, no *port* B, estão os valores referentes a endereço e dados, sinais *addr* e *dat* respectivamente. O sinal *sel* indica que o endereço é de 32 bits. Os sinais CYC e STB são levados para alto no momento em que o endereço e os dados estão disponíveis, porém não retornam para nível lógico baixo, pois não há alterações no sinal ACK proveniente do IP *Core* Ethernet.

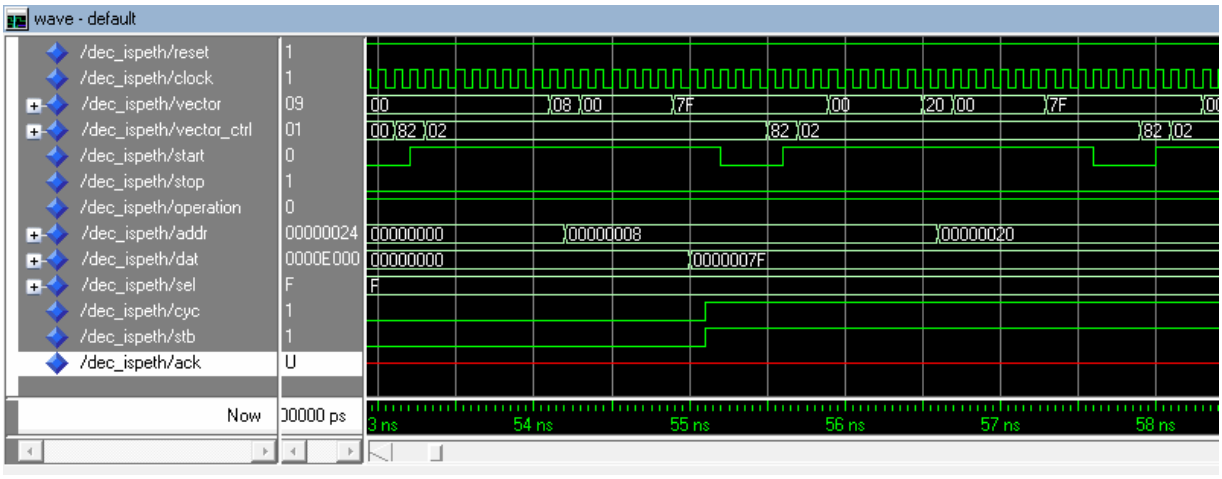

Figura 3.12: Ondas de simulação do bloco BCI para gravação dos registradores Ethernet.

Na figura 3.13, são apresentadas as ondas de simulação do *testbench* do IP *Core* Ethernet, apenas para mostrar como os sinais de CYC, STB e ACK deveriam se comportar. Os sinais utilizados no protocolo são os mesmos apresentados na figura anterior, endereço, dados, seleção, CYC, STB e ACK.

Para o *design* utilizou-se o Quartus II 8.0 *Web Edition*, no qual é possível utilizar diferentes linguagens em um mesmo projeto. Para verificação dos sinais do protocolo utilizou-se o osciloscópio, e posteriormente todos os registradores foram lidos e comparados com os valores desejados.

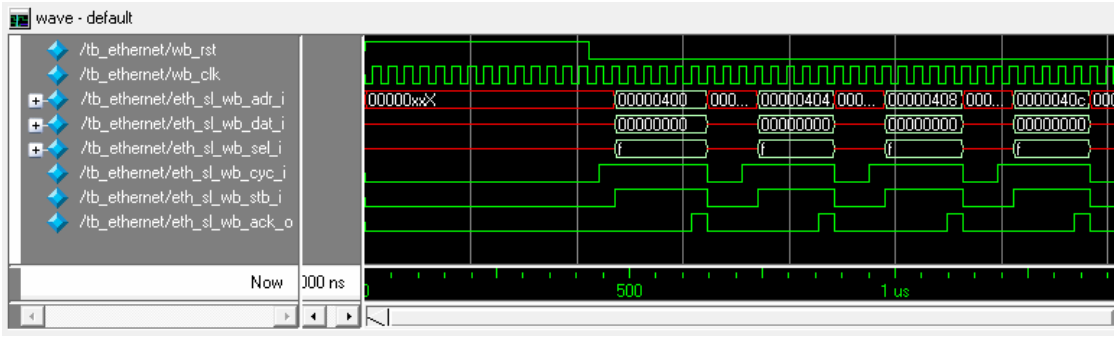

Figura 3.13: Ondas de simulação do *testbench* do *IP Core Ethernet*.

O IP *Core* Ethernet possibilita o envio e o recebimento de dados através da rede, mas utilizou-se apenas o recebimento de frames. Desta maneira, apenas alguns registros foram inicializados, sendo que para os demais foram mantidos os valores padrão. No registrador TX_BD_NUM foi indicado que dos 128 *buffers descriptors* um seria para recepção e os demais, para transmissão. Os registros e seus respectivos valores constam na tabela 3.10.

Tabela 3.10: Valores gravados nos registradores Ethernet.

Registrador	Endereço	Conteúdo
MODER	0x00000000	0x00000461
INT_MASK	0x00000008	0x0000007F
IPGT	0x0000000C	0x00000015
TX_BD_NUM	0x00000020	0x0000007F
CTRLMODER	0x00000024	0x00000001
MAC_ADDR0	0x00000040	0xA3201300
MAC_ADDR1	0x00000044	0x0000CDAB
BD	0x000007F8	0x0000E000
BD+4	0x000007FC	0x00000000

O modelo arquitetural utilizado no projeto é apresentado na figura 3.14. Neste modelo é possível visualizar a comunicação entre os blocos. O AVR passa os valores a serem gravados nos registradores Ethernet através dos *ports* A e B. O bloco BCI recebe essas informações e efetua a configuração dos registradores e dos *buffers descriptors*. O IP *Core* Ethernet não se comunica diretamente com o AVR. Quando ocorre a recepção de um pacote, o IP *Core* Ethernet gera uma interrupção indicando ao BCI o evento. O BCI por sua vez, indica ao AVR, através do *port* D, que o *program counter* deve apontar para a RAM, que implica em desabilitar a PROM e tornar os dados de saída da RAM nas instruções a serem executadas pelo processador.

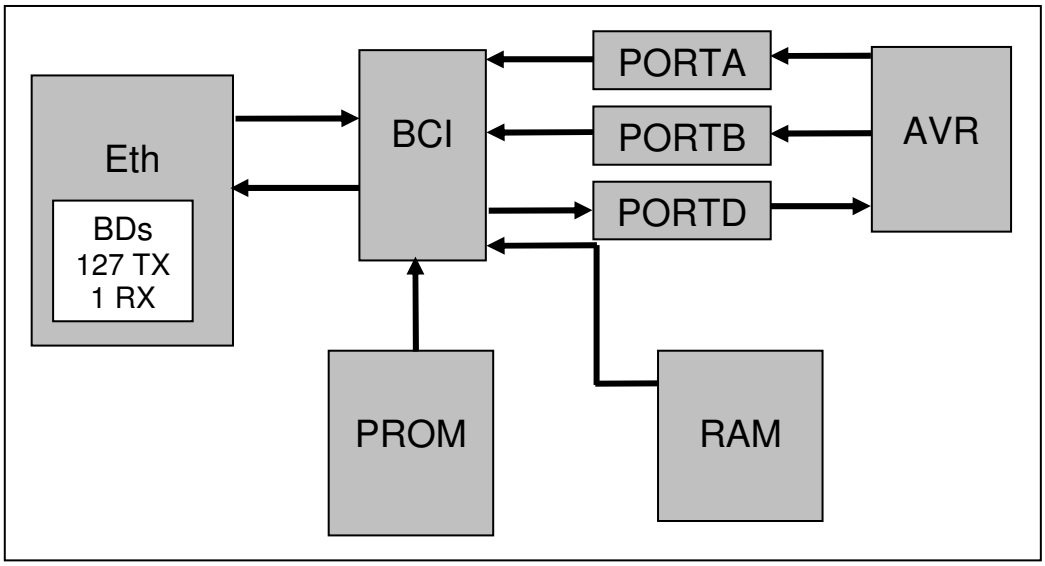

Figura 3.14: Arquitetura do projeto.

3.4.2 Controle das Memórias

A memória PROM contém o código inicial que o AVR deverá executar. Nela estão, como já mencionado anteriormente, todos os valores que configuraram a conexão de rede entre o kit que contém a FPGA e o computador.

Esta memória é preenchida a partir do endereço 0200H (figura 3.15). O microcontrolador executa seu conteúdo e configura os registros do bloco Ethernet. Após verificar o sinal de *stop,* o processador entra em *loop* e espera a indicação de que um frame foi recebido, através de uma interrupção gerada pelo IP *Core* Ethernet.

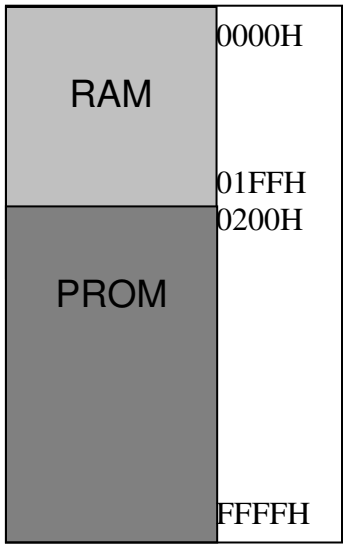

Figura 3.15: Organização das memórias.

O multiplexador recebe os dados de saída das duas memórias e libera o valor de uma ou de outra, dependendo do endereço e do sinal de CTRL. Se o PC (*Program Counter)* é maior que 0200H ou o sinal CTRL é igual a um, o multiplexador libera os dados da PROM. Se o PC está entre 0008H e 01FFF e o CTRL é igual à zero (indicando que o código do usuário ainda não terminou), os dados da RAM é que são repassados para o AVR_CORE.

Ao final do código que será enviado através da Ethernet, o usuário deve escrever 1 no bit 6 do *port* A, assim o bloco BCI saberá o código do usuário acabou e passará a executar o conteúdo da memória PROM novamente. O usuário deverá escrever ao final de seu código as seguintes instruções:

“ldi r19, $40

out $1B, r19”

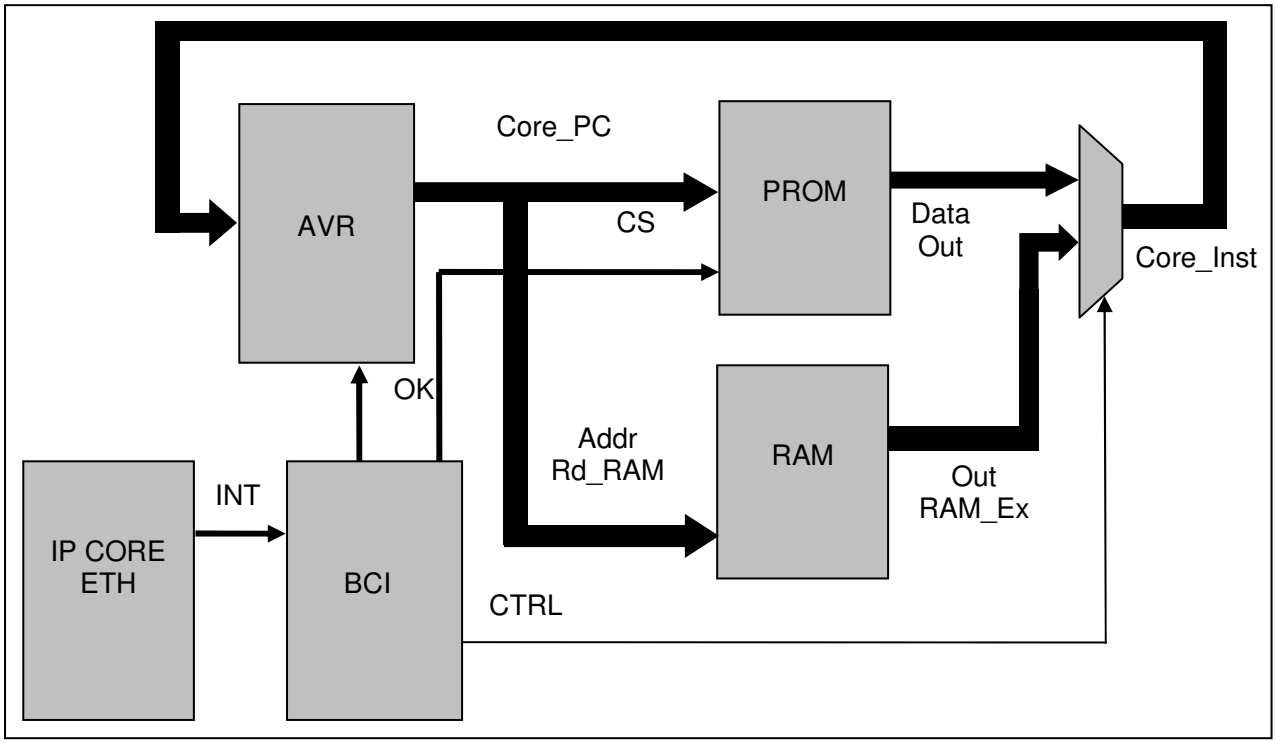

Figura 3.16: Diagrama em blocos do controle das Memórias.

O bit 5 do port A também é dedicado ao usuário. Este é utilizado para sinalizar que o código executado correspondeu ao esperado, ou seja, que o microcontrolador executou as instruções corretamente. É o usuário que vai definir a rotina de teste e o resultado esperado. A seguir é mostrado um exemplo:

```
.include "riscmcu.inc"
.def TEMP=r19
.org $0008
        ser TEMP
        out DDRA, TEMP; inicialização dos ports
        out DDRB, TEMP

        ldi r20, $03
        ldi r18, $06
l1:     dec r20          ;conta até que r20 seja igual a $00
        inc r18
        out PORTB, r18
        cpi r20, $00
```

```
        brne l1
        out PORTB, r18
        cpi r18, $09
        breq ok         ;compara r18 com $09, se for igual salta para ok
        ldi r19,$40     ;se r18 for difentente de $09, o bit 6 do port A recebe 1, o programa do
        out PORTA, r19 ;usuário é finalizado e a PROM é habilitada.

ok:     ldi r19,$20     ; o bit 5 do port A recebe 1, para indicar o sucesso da execução
        out PORTA, r19
        ldi r19,$40     ;o programa do usuário é finalizado e a PROM é habilitada.
        out PORTA, r19
```

O código em *Assembly* apresentado acima realiza três vezes o incremento do valor de um registrador, iniciando em 06H e finalizando com o 09H. Além disso, imprime esses valores no *port* B.

Se o microcontrolador executar corretamente as instruções, é esperado que no final o r18 tenha o valor 09H. Se isso for verdadeiro, é disponibilizado no *port* A o valor 20H, indicando que o código do usuário foi executado com sucesso. Caso contrário, a execução é finalizada e o microcontrolador volta a buscar as instruções na memória PROM.

Desta maneira, é possível verificar se ocorreu algum erro durante a execução, pois é possível fazer uma rotina de teste com verificações em vários registradores, sendo que se o primeiro teste estiver correto, faz-se um salto para o próximo e se ao final todos os testes forem exatos o valor 20H é impresso no *port* A.

3.5 Organização dos Blocos

Os blocos são organizados da seguinte maneira: o BCI_top é o entidade superior do projeto, sendo nele instanciados o BCI e o reset_gen (figura 3.17).

O reset_gen tem a de função capturar o sinal de reset gerado pelo botão do kit contendo a FPGA e manter esse sinal ativo por 50 ms, para garantir que todos os sinais e registros foram corretamente reinicializados. A freqüência de *clock* utilizada foi 50 MHz.

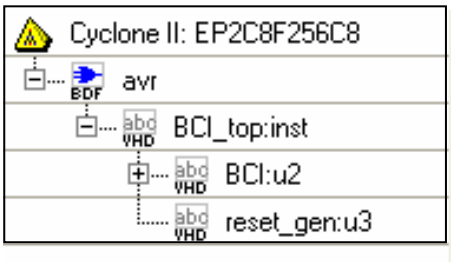

Figura 3.17: Organização dos blocos BCI_top, BCI e reset_gen.

O bloco BCI contém o AVR_CORE, o IP *Core* Ethernet, o CTRL_ACK e o CTRL_MEM_EX, como apresentado na figura 3.18.

O CTRL_ACK é responsável por disponibilizar, durante um ciclo de *clock*, um sinal confirmando a gravação do frame recebido em uma memória RAM, com dados de 32 bits. Isso faz com que os sinais de m_wb_cyc_o e m_wb_stb_o recebam nível lógico baixo e fiquem disponíveis para que a próxima parte do frame, com mais 32 bits, gravada na posição de memória seguinte.

Cada vez que o CTRL_ACK gera um *Acknowledgment*, o bloco CTRL_MEM_EX utiliza a saída da memória RAM, com dados de 32 bits, como entrada para a gravação de outra memória RAM de 16 bits, que posteriormente será a memória de execução. Este bloco é responsável por dividir os dados de entrada em duas partes e guardar em endereços separados. Este procedimento é necessário, pois as instruções executadas pelo microcontrolador devem ser de 16 bits, como acontece na PROM.

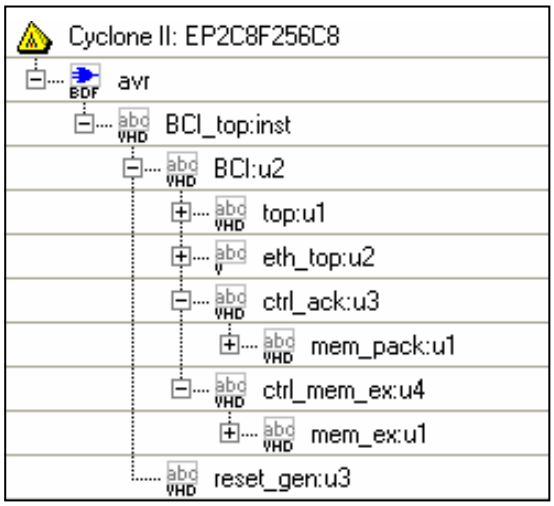

Figura 3.18: Blocos contidos no BCI.

O bloco top_avr_core_sim do AVR_CORE inicialmente instanciava as memórias PROM e DataRAM (memória de dados do AVR), mas para que houvesse o controle da memória de programa, o bloco *control* foi criado. Ele agora passa a instanciar as duas memórias (figura

3.19). Quando este bloco recebe um sinal de controle, desabilita o *chip select* da PROM e faz com que as instruções passem a ser o conteúdo de saída da memória RAM de execução. O microcontrolador volta a executar as instruções contidas na PROM assim que verifica o valor 40H no *port* A.

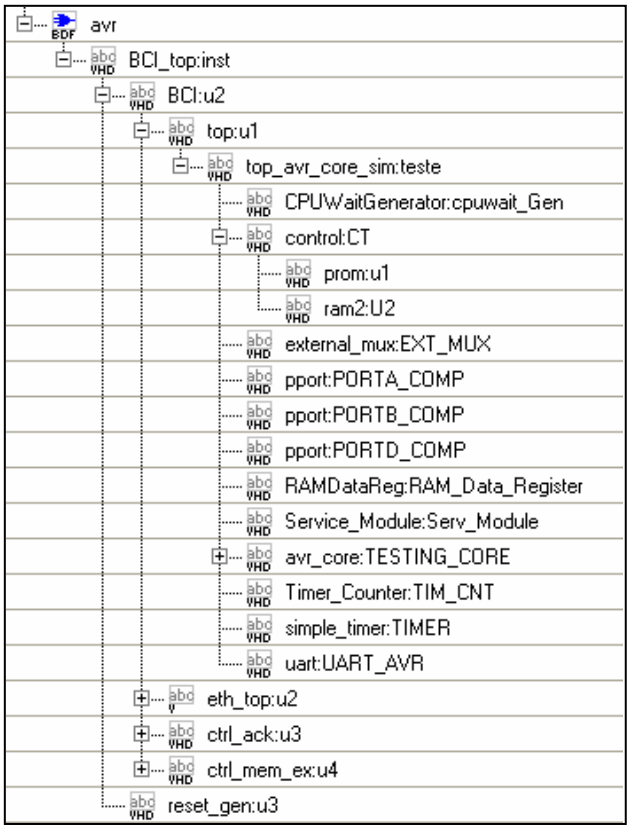

Figura 3.19: Organização de todos os blocos.

3.6 Transmissões dos dados

Para enviar os dados para o IP *Core* Ethernet usa-se um programa chamado Bittwist. Este programa utiliza um arquivo com formato pcap (*packet capture*) obtido através do programa WinDump [23], que é um analisador de tráfego de rede que pode capturar e salvar em disco os pacotes da rede [24].

Para editar o conteúdo do pacote capturado utiliza-se o programa Bittwiste [25]. Com este editor é possível alterar o *payload*, especificar a camada desejada, alterar o cabeçalho, inserir endereço de destino e origem dos dados, entre outras opções.

Neste projeto definiu-se que toda a comunicação seria feita no nível da camada de enlace, ou seja, utilizando a Ethernet já que o IP *Core* Ethernet não tem o protocolo TCP/IP implementado.

Abaixo, na figura 3.20, é apresentado um exemplo da edição de um pacote capturado, que ainda possiu cabeçalho TCP/IP. O *payload* é o programa que será armazenado na memória e posteriormente executado.

Parâmetros do Bittwiste:
-I *input*: Arquivo pcap de entrada.
-O *output*: Arquivo de saída.
-R *range*: Salva apenas a faixa de pacotes especificada.
-X *payload*: Anexa o payload em hexadecimal ao fim de cada pacote.
-L *layer*: Copia a camada especificada e descarta os dados restantes.
-T *header*: Altera apenas o cabeçalho especificado.
-d dmac: Endereço MAC de destino.
-s smac: Endereço MAC de origem.
- t len: Indica que o tamanho dos dados deve ser calculo e anexado ao pacote.

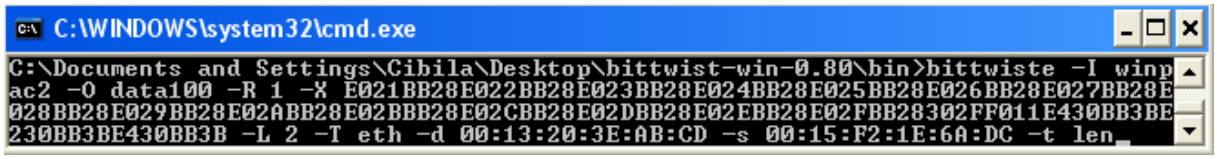

Figura 3.20: Edição do pacote através do Bittwiste.

Após o pacote ser editado, está pronto para ser envido, utilizando, para isso, o programa Bittwist (figura 3.21).

-i *interface*: Interface de rede utilizada, neste caso 2 é referente à placa de rede.

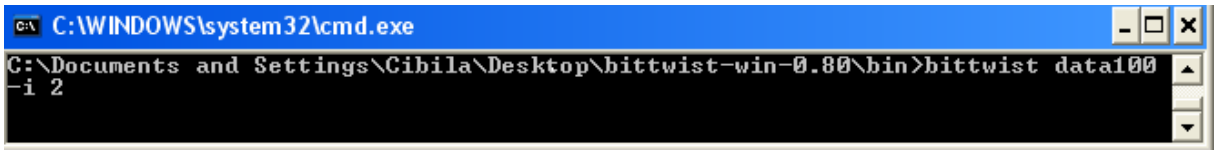

Figura 3.21: Envio do frame através do Bittwist.

O *software* Wireshark (figura 3.22), é um analisador de tráfego de rede, e foi utilizado para controle dos pacotes enviados. Com esta ferramenta e possível visualizar todo o conteúdo enviado pela rede.

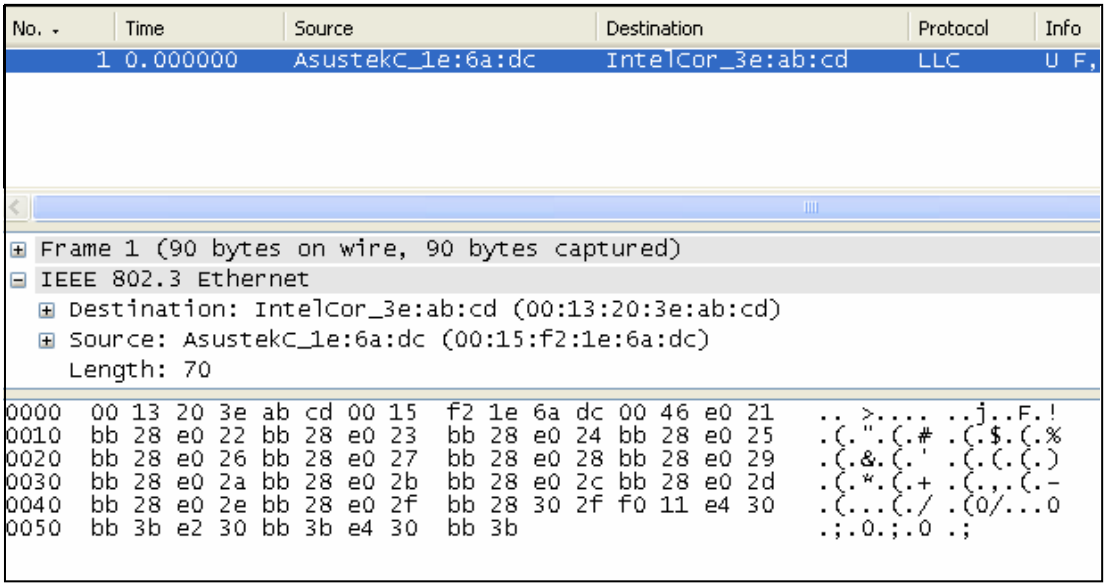

Figura 3.22: Conteúdo do pacote enviado.

CAPÍTULO 4

RESULTADOS

O projeto foi dividido em qvárias etapas. Os resultados obtidos em cada etapa eram pré-requisitos para a próxima. Desta maneira, a seqüência das atividades só acontecia quando os resultados eram adequados.

O primeiro passo após a escolha do microcontrolador utilizado no projeto foi descobrir mais sobre seu funcionamento. Como o AVR_CORE não possuía *testbench,* foi necessário realizar alguns testes em simulação para isso. Testou-se a maioria das instruções suportadas por um AVR, através de códigos escritos em C e em Assembly. O microcontrolador apresentou um bom resultado nestes testes. A figura 4.1 mostra os registradores internos e a saída da ULA. Todos os sinais foram analisados para conferir a execução.

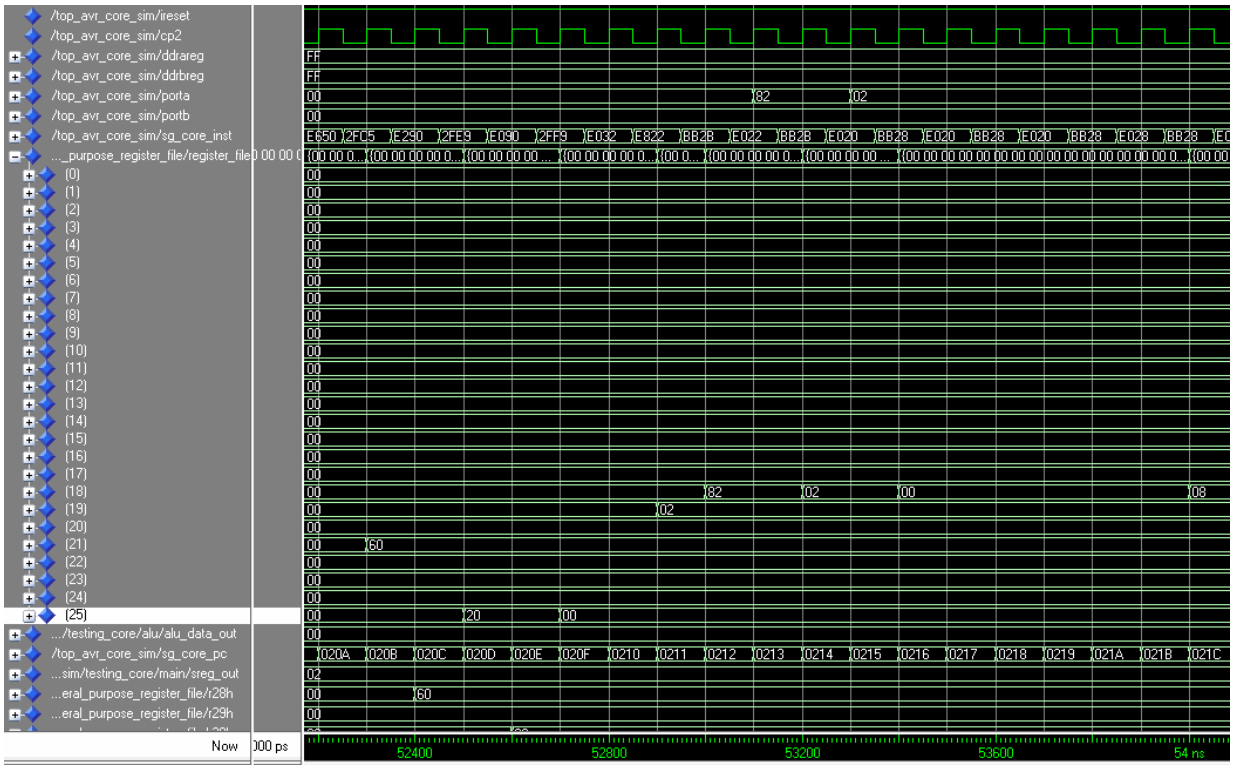

Figura 4.1: Sinais internos do AVR.

Desenvolveu-se então o bloco BCI e o protocolo de gravação dos registros. Porém, o IP *Core* Ethernet ainda não havia sido conectado. Todas as verificações foram feitas através de simulação.

A implementação das trocas de memórias foi inicialmente realizada, apenas com o BCI e o AVR, onde o conteúdo da memória RAM era proveniente de uma cópia de parte da memória PROM. Esta copia era realizada quando o microcontrolador estava executando o conteúdo da memória de programa.

A RAM tinha seus valores gravados a partir do endereço 60H. Ao final de sua gravação, havia um salto para o endereço 0210H. Neste momento, havia a troca entre as memórias e o microcontrolador passava a executar o conteúdo da RAM, como mostra a figura 4.2.

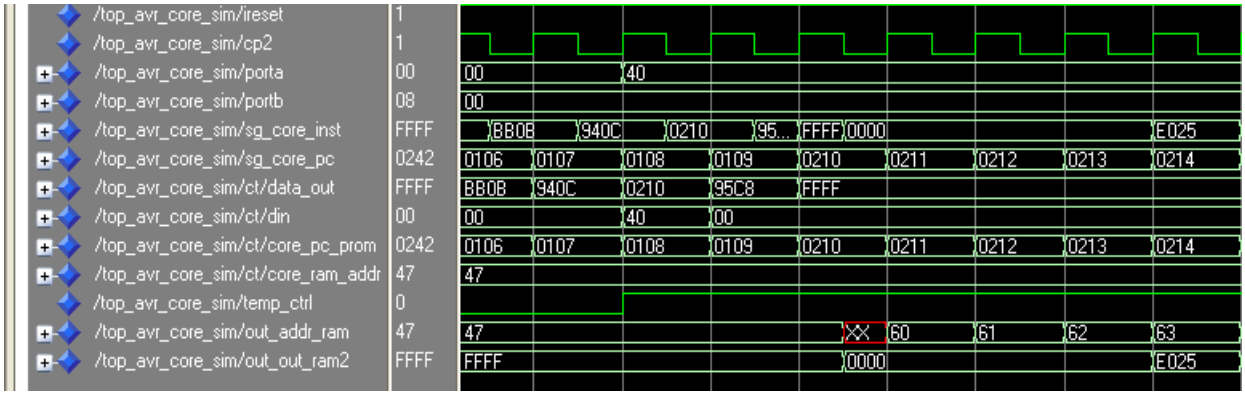

Figura 4.2: Controle das memórias.

Só então o IP *Core* Ethernet foi inserido ao projeto e a gravação dos registros efetivamente realizada.

Inicialmente, utilizou-se o kit Altera NIOS II *Development Kit*, porém com a inclusão do *IP Core Ethernet* foi necessário substituí-lo, pois este kit possui o componente Intel LX971A (*Fast Ethernet PHY Transceiver*) que provê uma interface independente ao meio (MII) e controle de acesso ao meio (MAC), e necessitávamos de um componente que disponibilizasse apenas o MII já que o MAC já estava implementado no IP *Core* Ethernet. Utilizou-se então uma placa desenvolvida no laboratório de microeletrônica para o projeto MicroVote, figura 4.3, que possui o componente Teridian 78Q2123, FPGA Cyclone II EP2C8F256C8, interface serial e PS2.

Precisava-se de um *software* que realizasse o envio de quadros através da rede Ethernet, no nível da camada Ethernet, ou seja, sem protocolo TCP/IP, pois o IP *Cor*e Ethernet não possui esses protocolos implementados. Como já mencionado anteriormente, o *software* selecionado foi o Bittwist, o qual realiza o envio e possui um editor para frames capturados.

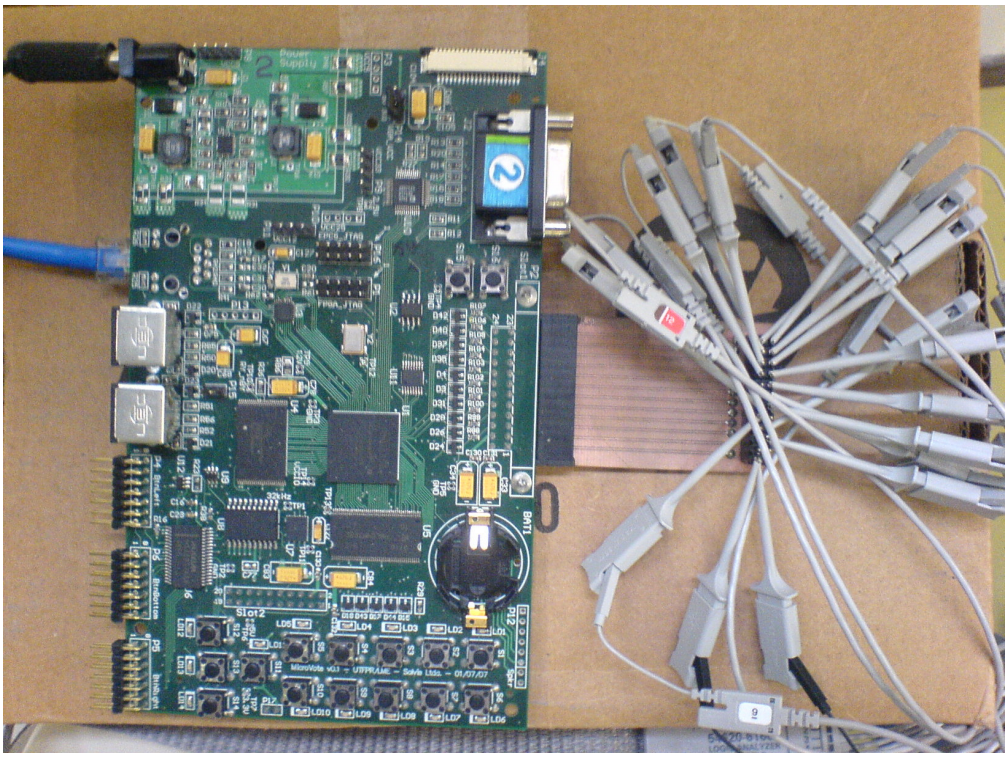

Figura 4.3: Placa MicroVote v0.1 – UTFPR/LME – Solvis LTDA.

Abaixo estão alguns exemplos de programas enviados pela Ethernet e executados pelo microcontrolador.

No primeiro exemplo (tabela 4.1), o objetivo é verificar a instrução de salto (jmp). Neste código, os valores 01H e 02H são impressos no *port* B. Em seguida, a instrução *jump* indica um salto para o endereço 15H, e o valor 05H deve ser impresso no *port* B. A parte do código onde os valores 03H e 04H é carregada e disponibilizada, não deve ser executada.

Tabela 4.1: Exemplo de código do usuário 1.

Endereço	Opcode	Instrução Assembly
000008	ef3f	ser TEMP
000009	bb3a	out $1A, TEMP
00000a	bb37	out DDRB, TEMP
00000b	e021	ldi r18, $01
00000c	bb28	out PORTB, r18
00000d	e022	ldi r18, $02
00000e	bb28	out PORTB, r18
00000f	940c	jmp l1
000010	0015	

Endereço	Opcode	Instrução Assembly
000011	e023	ldi r18, $03
000012	bb28	out PORTB, r18
000013	e024	ldi r18, $04
000014	bb28	out PORTB, r18
000015	e025	l1: ldi r18, $05
000016	bb28	out PORTB, r18
000017	E026	ldi r18, $06
000018	bb28	out PORTB, r18
000019	e027	ldi r18, $07
00001a	bb28	out PORTB, r18
00001b	e028	ldi r18, $08
00001c	bb28	out PORTB, r18
00001d	e029	ldi r18, $09
00001e	bb28	out PORTB, r18
00001f	e02a	ldi r18, $0A
000020	bb28	out PORTB, r18
000021	e02b	ldi r18, $0B
000022	bb28	out PORTB, r18
000023	e02c	ldi r18, $0C
000024	bb28	out PORTB, r18
000025	e02d	ldi r18, $0D
000026	bb28	out PORTB, r18
000027	e02e	ldi r18, $0E
000028	bb28	out PORTB, r18
000029	e02f	ldi r18, $0F
00002a	bb28	out PORTB, r18

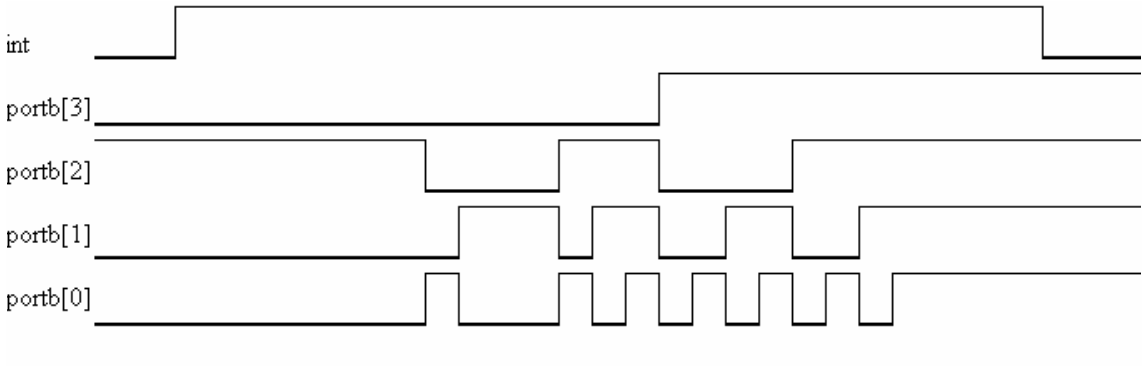

Figura 4.4: Ondas capturadas pelo osciloscópio para o código do usuário 1.

No segundo exemplo (tabela 4.2), a instrução de desvio, breq, é verificada. Neste código, os valores 01H e 02H são impressos no *port* B. Posteriormente, há uma comparação entre o registrador 18 e o valor absoluto 02H. Como os valores são iguais, há um desvio para o endereço 1DH, onde o valor 09H é carregado e disponibilizado no *port* B.

Tabela 4.2: Exemplo de código do usuário 2.

Endereço	Opcode	Instrução Assembly
000008	ef3f	ser TEMP
000009	bb3a	out $1A, TEMP
00000a	bb37	out DDRB, TEMP
00000b	e021	ldi r18, $01
00000c	bb28	out PORTB, r18
00000d	e022	ldi r18, $02
00000e	bb28	out PORTB, r18
00000f	3022	cpi r18, $02
000010	f061	breq l1
000011	e023	ldi r18, $03
000012	bb28	out PORTB, r18
000013	e024	ldi r18, $04
000014	bb28	out PORTB, r18
000015	e025	ldi r18, $05
000016	bb28	out PORTB, r18
000017	E026	ldi r18, $06
000018	bb28	out PORTB, r18
000019	e027	ldi r18, $07

Endereço	Opcode	Instrução Assembly
00001a	bb28	out PORTB, r18
00001b	e028	ldi r18, $08
00001c	bb28	out PORTB, r18
00001d	e029	l1:ldi r18, $09
00001e	bb28	out PORTB, r18
00001f	e02a	ldi r18, $0A
000020	bb28	out PORTB, r18
000021	e02b	ldi r18, $0B
000022	bb28	out PORTB, r18
000023	e02c	ldi r18, $0C
000024	bb28	out PORTB, r18
000025	e02d	ldi r18, $0D
000026	bb28	out PORTB, r18
000027	e02e	ldi r18, $0E
000028	bb28	out PORTB, r18
000029	e02f	ldi r18, $0F
00002a	bb28	out PORTB, r18

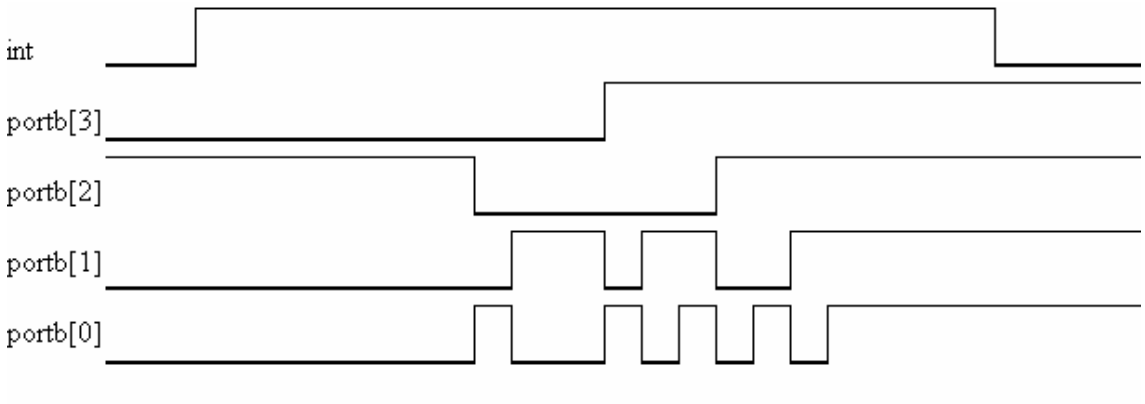

Figura 4.5: Ondas capturadas pelo osciloscópio para o código do usuário 2.

No terceiro exemplo (tabela 4.3), a instrução de desvio, brne, é testada. Como no exemplo anterior, os valores 01H e 02H são impressos no *port* B. Em seqüência, há uma comparação entre o registrador 18 e o valor absoluto 03H. Como os valores são diferentes, há

um desvio para o endereço 1BH, onde o valor 08H é carregado e disponibilizado no *port* B e em seguida as próximas instruções são executadas.

Tabela 4.3: Exemplo de código do usuário 3.

Endereço	Opcode	Instrução Assembly
000008	ef3f	ser TEMP
000009	bb3a	out $1A, TEMP
00000a	bb37	out DDRB, TEMP
00000b	e021	ldi r18, $01
00000c	bb28	out PORTB, r18
00000d	e022	ldi r18, $02
00000e	bb28	out PORTB, r18
00000f	3023	cpi r18, $03
000010	f451	brne l1
000011	e023	ldi r18, $03
000012	bb28	out PORTB, r18
000013	e024	ldi r18, $04
000014	bb28	out PORTB, r18
000015	e025	ldi r18, $05
000016	bb28	out PORTB, r18
000017	e026	ldi r18, $06
000018	bb28	out PORTB, r18
000019	e027	ldi r18, $07
00001a	bb28	out PORTB, r18
00001b	e028	l1: ldi r18, $08
00001c	bb28	out PORTB, r18
00001d	e029	ldi r18, $09
00001e	bb28	out PORTB, r18
00001f	e02a	ldi r18, $0A
000020	bb28	out PORTB, r18
000021	e02b	ldi r18, $0B
000022	bb28	out PORTB, r18
000023	e02c	ldi r18, $0C

Endereço	Opcode	Instrução Assembly
000024	bb28	out PORTB, r18
000025	e02d	ldi r18, $0D
000026	bb28	out PORTB, r18
000027	e02e	ldi r18, $0E
000028	bb28	out PORTB, r18
000029	e02f	ldi r18, $0F
00002a	bb28	out PORTB, r18

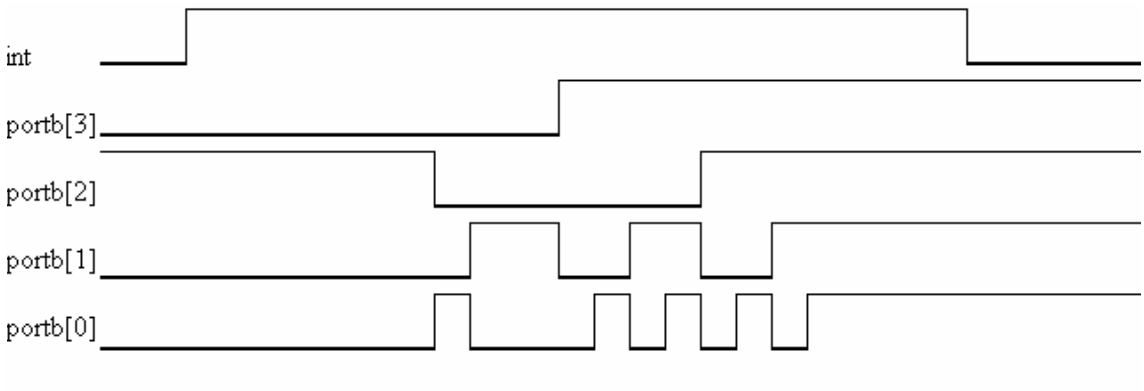

Figura 4.6: Ondas capturadas pelo osciloscópio para o código do usuário 3.

No último exemplo (tabela 4.4), os valores de 01H a 0FH são carregados no registrador 18 e disponibilizados no *port* B. Este exemplo mostra um *selftest* que o usuário pode escrever para verificar se as instruções foram executadas corretamente. Como já comentado anteriormente, o valor de alguns registradores podem ser comparados com os valores esperados para avaliação. Neste caso, espera-se que o último valor carregado no registrador 18 seja 0FH (endereço 29H). É realizada então, uma comparação entre o conteúdo desse registrador e o valor absoluto 0FH, se isso for verdadeiro, o *program counter* é incrementado e passa a apontar para o endereço 2DH, e o bit 5 do *port* A recebe o valor 1, indicando que o código foi executado com sucesso. Caso contrário, houve algum problema durante a execução, e o bit 5 não é levado para nível lógico alto. O programa é finalizado, quando é o bit 6 do *port* A recebe o valor 1.

Tabela 4.4: Exemplo de código do usuário 4.

Endereço	Opcode	Instrução Assembly
000008	ef3f	ser TEMP
000009	bb3a	out $1A, TEMP
00000a	bb37	out DDRB, TEMP
00000b	e021	ldi r18, $01
00000c	bb28	out PORTB, r18
00000d	e022	ldi r18, $02
00000e	bb28	out PORTB, r18
00000f	e023	ldi r18, $03
000010	bb28	out PORTB, r18
000011	e024	ldi r18, $04
000012	bb28	out PORTB, r18
000013	e025	ldi r18, $05
000014	bb28	out PORTB, r18
000015	E026	ldi r18, $06
000016	bb28	out PORTB, r18
000017	e027	ldi r18, $07
000018	bb28	out PORTB, r18
000019	e028	ldi r18, $08
00001a	bb28	out PORTB, r18
00001b	e029	l1:ldi r18, $09
00001c	bb28	out PORTB, r18
00001d	e02a	ldi r18, $0A
00001e	bb28	out PORTB, r18
00001f	e02b	ldi r18, $0B
000020	bb28	out PORTB, r18
000021	e02c	ldi r18, $0C
000022	bb28	out PORTB, r18
000023	e02d	ldi r18, $0D
000024	bb28	out PORTB, r18
000025	e02e	ldi r18, $0E
000026	bb28	out PORTB, r18

Endereço	Opcode	Instrução Assembly
000027	e02f	ldi r18, $0F
000028	bb28	out PORTB, r18
000029	302f	cpi r18, $0F
00002a	f011	breq ok
00002b	e430	ldi r19, $40
00002c	bb3b	out $1B, r19
00002d	e230	ok: ldi r19, $20
00002e	bb3b	out $1B, r19
00002f	e430	ldi r19, $40
000030	bb3b	out $1B, r19

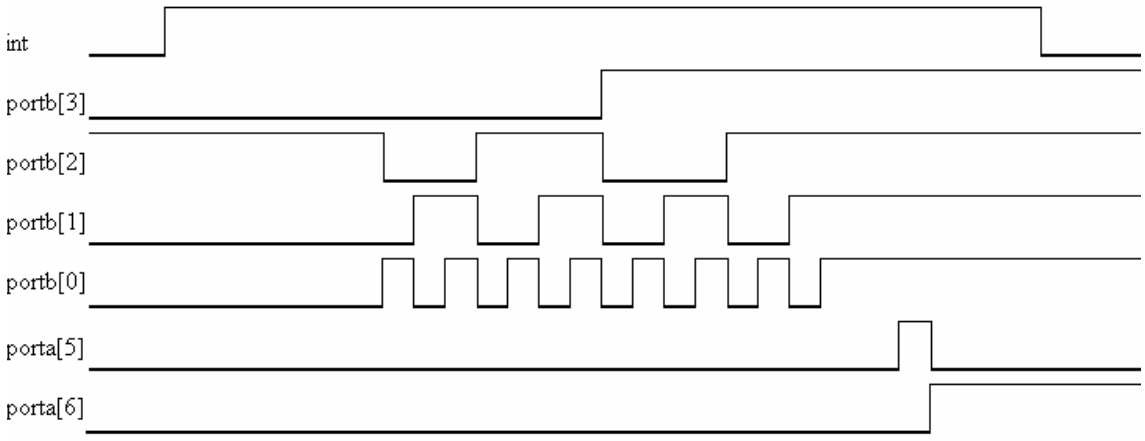

Figura 4.7: Ondas capturadas pelo osciloscópio para o código do usuário 4.

Na placa utilizou-se três LEDs para sinalização, um indicando se o código do usuário foi executado corretamente, outro mostrando se o código do usuário chegou ao fim e um último informando qual memória está sendo utilizada (figura 4.8).

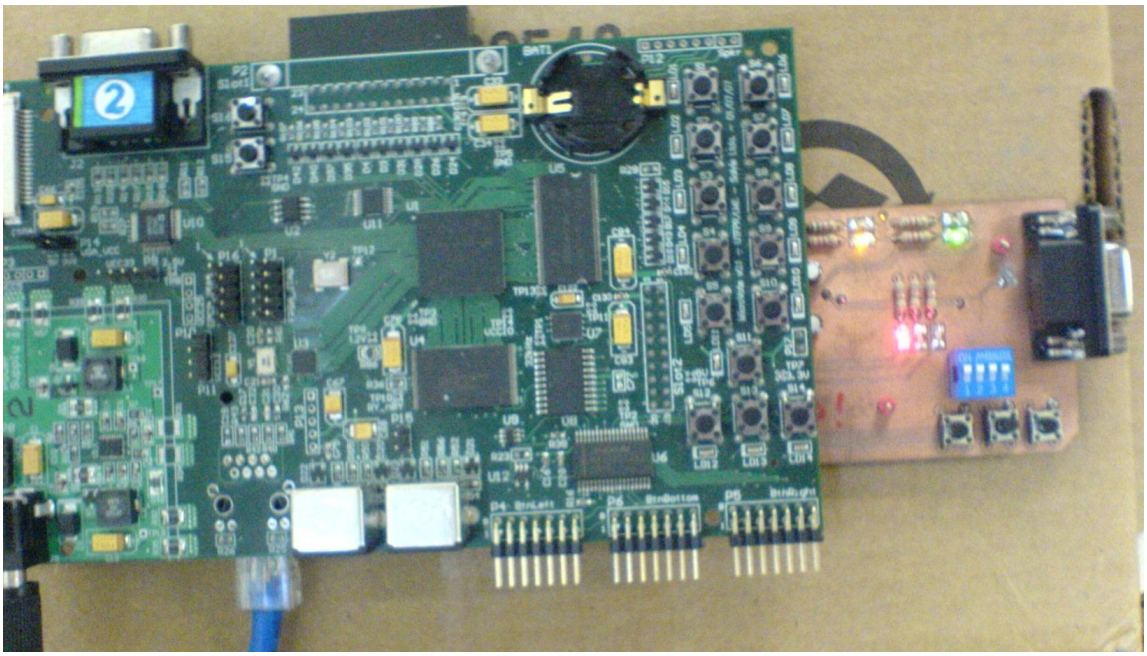

Figura 4.8: LEDs de sinalização.

CAPÍTULO 5

DISCUSSÃO E CONCLUSÕES

5.1 ANÁLISE DOS RESULTADOS

O método complementar desenvolvido neste trabalho propôs tornar os testes realizados para avaliação de um microcontrolador independente de plataforma, fabricante e *software* de *design.* Para isso, incluiu-se ao projeto o IP *Core* Ethernet. Assim, o código implementado pode ser enviado facilmente através da interface de rede Ethernet.

Os resultados obtidos foram satisfatórios e correspondentes ao esperado inicialmente. Como o projeto foi desenvolvido em etapas, os problemas encontrados foram resolvidos antes do início da próxima fase, impedindo assim que erros fossem adiados até o fim do trabalho. Primeiramente, os blocos foram implementados e testados através de um simulador, sendo possível executar o código, passo a passo, e analisar a simulação. Posteriormente, o projeto foi gravado na FPGA, e os resultados práticos comparados com os obtidos anteriormente.

Ao longo do trabalho foram encontradas algumas dificuldades, que, conseqüentemente, exigiram que detalhes de implementação dos blocos fossem descobertas. Encontraram-se problemas também para fazer o envio dos frames utilizando o IP *Core* Ethernet, requerendo fazer um estudo detalhado para que os dados fossem enviados corretamente.

Outra dificuldade encontrada foi que a licença do *software* de simulação permitia que apenas uma linguagem de descrição de *hardware* fosse utilizada no projeto. Como os blocos BCI e AVR estão em VHDL e o IP *Core* Ethernet em Verilog, muitas simulações tiveram que ser realizadas separadamente. Nos sinais que comunicavam um bloco ao outro, os valores eram forçados, pois já se sabia previamente como o sinal se comportava analisando o *testbench* do IP *Core* Ethernet.

Os resultados apresentados no capítulo anterior mostram exemplos simples de códigos em Assembly para demonstrar que o método complementar é capaz de realizar o que foi proposto inicialmente. Antes da conexão do IP *Core* Ethernet ao AVR foram testados códigos mais elaborados e o AVR obteve resultados positivos.

5.2 CONCLUSÕES

O desempenho do bloco BCI foi satisfatório, pois conseguiu-ae efetuar todas as atividades necessárias, além de não haver atraso na disponibilidade da instrução quando há a troca entre as memórias. Com a utilização de um IP *Core* Ethernet para recebimento dos testes através da rede, o tempo dedicado às verificações reduz consideravelmente.

O envio dos frames através do *software* de transmissão e a recepção pelo IP *Core* Ethernet, após alguns ajustes, ocorreram perfeitamente.

O principal objetivo do trabalho foi o desenvolvimento do método complementar que pode ser utilizado para avaliação de outro microcontrolador de oito bits. Para isso o microcontrolador deve possuir pelo menos três *ports,* utilizados para controle e dados, como mencionado na sessão 3.4.1. Não é necessário que esses *ports* sejam de uso exclusivo deste método, pois são utilizados apenas no inicio da execução, durante a configuração dos registradores Ethernet.

O método complementar para avaliação pode ser utilizado para microcontroladores CISC e RISC. É possível também avaliar MCUs de 12, 16, 32 ou 64 bits bastando apenas que algumas alterações sejam realizadas no código.

O microcontrolador AVR foi verificado, embora não se considere a freqüência de operação. Todos os testes foram realizados com freqüência de clock de 50 MHz, desta maneira, não é possível garantir a partir de que freqüência o microcontrolador passará a apresentar problemas. O foco do projeto foi o desenvolvimento do método complementar, sendo assim, demais testes no microcontrolador podem ser realizados futuramente.

Este método complementar deve ser utilizado em conjunto com testes realizados em simulação, assim uma melhor cobertura na verificação funcional é obtida.

5.3 TRABALHOS FUTUROS

Para um trabalho futuro seria interessante que o IP *Core* Ethernet realizasse também o envio de frames como uma resposta ao usuário. Este bloco já tem funções de transmissão implementadas. Desta maneira, o usuário faria o envio de seu código e ao invés de visualizar nos LEDs o status da execução, receberia através da interface Ethernet essa e outras informações, como o conteúdo de alguns registros, indicação de troca entre as memórias, disponibilidade para o envio do próximo frame, etc.

Futuramente alterações na implementação poderiam ser realizadas permitindo que vários *frames* sejam recebidos e executados em seqüência, assim o usuário conseguiria enviar testes mais extensos, pois se ultrapassasem o tamanho máximo de um *frame,* os dados seriam fragmentados e enviados em quantos *frames* fossem necessários.

Em função do tempo, testes em um kit da Xilinx e utilização do sistema operacional UNIX ficaram como trabalhos futuros.

ANEXO 1 - WISHBONE

Tabela 6.1 - Descrição dos Sinais WISHBONE.

Module or interface	*Signal*	*Description*
SYSCON *module signals*	CLK_O	*The system clock output* [CLK_O] *is generated by the* SYSCON *module. It coordinates all activities for the internal logic within the* WISHBONE *interconnect. The* INTERCON *module connects the* [CLK_O] *output to the* [CLK_I] *input on master and slave interfaces.*
	RST_O	*The reset output* [RST_O] *is generated by the* SYSCON *module. It forces all* WISHBONE *interfaces to restart. All internal self-starting state machines are forced into an initial state. The* INTERCON *connects the* [RST_O] *output to the* [RST_I] *input on master and slave interfaces.*
Signals common to master and slave interfaces	CLK_I	*The clock input* [CLK_I] *coordinates all activities for the internal logic within the* WISHBONE *interconnect. All* WISHBONE *output signals are registered at the rising edge of* [CLK_I]. *All* WISHBONE *input signals are stable before the rising edge of* [CLK_I].
	DAT_I()	*The data input array* [DAT_I ()] *is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g.* [DAT_I (63..0)]).
	DAT_O()	*The data output array* [DAT_O ()] *is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 4-bits* (*e.g.* [DAT_I (63..0)]).
	RST_I	*The reset input* [RST_I] *forces the* WISHBONE *interface to restart. Furthermore, all internal self-starting state machines will be forced into an initial state. This signal only resets the* WISHBONE *interface. It is not required to reset other parts of an* IP *core although it may be used that way).*
	TGD_I()	*Data tag type* [TGD_I ()] *is used on master and slave interfaces. It contains information that is associated with the data input array* [DAT_I ()], *and is qualified by signal* [STB_I]. *For example, parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification.*
	TGD_O()	*Data tag type* [TGD_O ()] *is used on master and slave interfaces. It contains information that is associated with the data output array* [DAT_O ()], *and is qualified by signal* [STB_O]. *For example, parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification.*
	ACK_I	*The acknowledge input* [ACK_I], *when asserted, indicates the normal termination of a bus cycle.*
	ADR_O()	*The address output array* [ADR_O ()] *is used to pass a binary*

Master signals		*address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size and granularity. For example the array size on a 32-bit data port with byte granularity is* [ADR_O (n..2)]. *In some cases (such as* FIFO *interfaces) the array may not be present on the interface.*
	CYC_O	*The cycle output* [CYC_O]*, when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a block transfer cycle there can be multiple data transfers. The* [CYC_O] *signal is asserted during the first data transfer, and remains asserted until the last data transfer. The* [CYC_O] *signal is useful for interfaces with multi-port interfaces (such as dual port memories). In these cases, the* [CYC_O] *signal requests use of a common bus from an arbiter.*
	ERR_I	*The error input* [ERR_I] *indicates an abnormal cycle termination. The source of the error, and the response generated by the master is defined by the* IP *core supplier. Also see the* [ACK_I] *and* [RTY_I] *signal descriptions.*
	LOCK_O	*The lock output* [LOCK_O] *when asserted, indicates that the current bus cycle is uninterruptible. Lock is asserted to request complete ownership of the bus. Once the transfer has started, the INTERCON does not grant the bus to any other master, until the current master negates* [LOCK_O] *or* [CYC_O].
	RTY_I	*The retry input* [RTY_I] *indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the* IP *core supplier.*
	SEL_O()	*The select output array* [SEL_O ()] *indicates where valid data is expected on the* [DAT_I ()] *signal array during read cycles, and where it is placed on the* [DAT_O ()] *signal array during write cycles. The array boundaries are determined by the granularity of a port. For example, if 8-bit granularity is used on a 64-bit port, then there would be an array of eight select signals with boundaries of* [SEL_O (7..0)]. *Each individual select signal correlates to one of eight active bytes on the 64-bit data port.*
	STB_O	*The strobe output* [STB_O] *indicates a valid data transfer cycle. It is used to qualify various other signals on the interface such as* [SEL_O ()]. *The slave asserts either the* [ACK_I], [ERR_I] *or* [RTY_I] *signals in response to every assertion of the* [STB_O] *signal.*
	TGA_O()	*Address tag type* [TGA_O ()] *contains information associated with address lines* [ADR_O ()]*, and is qualified by signal* [STB_O]. *For example, address size (24-bit, 32-bit etc.) and memory management (protected vs. unprotected) information can be attached to an address. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is defined by this specification.*
	TGC_O()	*Cycle tag type* [TGC_O ()] *contains information associated with bus cycles, and is qualified by signal* [CYC_O]. *For example, data transfer, interrupt acknowledge and cache control cycles can be uniquely identified with the cycle tag. They can also be used to discriminate between* WISHBONE *single, block and RMW cycles.*

		These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is defined by this specification.
	WE_O	*The write enable output* [WE_O] *indicates whether the current local bus cycle is a read or write cycle. The signal is negated during read cycles, and is asserted during write cycles.*
Slave signals	ACK_O	*The acknowledge output* [ACK_O]*, when asserted, indicates the termination of a normal bus cycle.*
	ADR_I()	*The address input array* [ADR_I ()] *is used to pass a binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size. For example the array size on a 32-bit data port with byte granularity is* [ADR_O (n..2)]. *In some cases (such as* FIFO *interfaces) the array may not be present on the interface.*
	CYC_I	*The cycle input* [CYC_I], *when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a block transfer cycle there can be multiple data transfers. The* [CYC_I] *signal is asserted during the first data transfer, and remains asserted until the last data transfer.*
	ERR_O	*The error output* [ERR_O] *indicates an abnormal cycle termination. The source of the error, and the response generated by the master is defined by the* IP *core supplier.*
	LOCK_I	*The lock input* [LOCK_I], *when asserted, indicates that the current bus cycle is uninterruptible. A slave that receives the lock* [LOCK_I] *signal is accessed by a single master only, until either* [LOCK_I] *or* [CYC_I] *is negated.*
	RTY_O	*The retry output* [RTY_O] *indicates that the indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the* IP *core supplier.*
	SEL_I()	*The select input array* [SEL_I ()] *indicates where valid data is placed on the* [DAT_I ()] *signal array during write cycles, and where it should be present on the* [DAT_O ()] *signal array during read cycles. The array boundaries are determined by the granularity of a port. For example, if 8-bit granularity is used on a 64-bit port, then there would be an array of eight select signals with boundaries of* [SEL_I (7..0)]. *Each individual select signal correlates to one of eight active bytes on the 64-bit data port.*
	STB_I	*The strobe input* [STB_I], *when asserted, indicates that the slave is selected. A slave shall respond to other* WISHBONE *signals only when this* [STB_I] *is asserted (except for the* [RST_I] *signal which should always be responded to). The slave asserts either the* [ACK_O], [ERR_O] *or* [RTY_O] *signals in response to every assertion of the* [STB_I] *signal.*
	TGA_I	*Address tag type* [TGA_I ()] *contains information associated with address lines* [ADR_I ()], *and is qualified by signal* [STB_I]. *For example, address size (24-bit, 32-bit etc.) and memory management (protected vs. unprotected) information can be attached to an address. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification.*

	TGC_I()	*Cycle tag type* [TGC_I ()] *contains information associated with bus cycles, and is qualified by signal* [CYC_I]. *For example, data transfer, interrupt acknowledge and cache control cycles can be uniquely identified with the cycle tag. They can also be used to discriminate between* WISHBONE *single, block and RMW cycles. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification.*
	WE_I	*The write enable input* [WE_I] *indicates whether the current local bus cycle is a read or write cycle. The signal is negated during read cycles, and is asserted during write cycles.*

ANEXO 2 – IP CORE ETHERNET

Tabela 6.2 - Sinais de Interface com o servidor [22].

Port	*Width*	*Direction*	*Description*
CLK_I	1	I	*Clock Input*
RST_I	1	I	*Reset Input*
ADDR_I	32	I	*Address Input*
DATA_I	32	I	*Data Input*
DATA_O	32	O	*Data Output*
SEL_I	4	I	*Select Input Array - Indicates which bytes are valid on the data bus. Whenever this signal is not 1111b during a valid access, the ERR_O is asserted.*
WE_I	1	I	*Write Input - Indicates a Write Cycle when asserted high or a Read Cycle when asserted low.*
STB_I	1	I	*Strobe Input - Indicates de beginning of a valid transfer cycle.*
CYC_I	1	I	*Cycle Input - Indicates that a valid bus cycle is in progress.*
ACK_O	1	O	*Acknowledgment Output - Indicates a normal Cycle termination.*
ERR_O	1	O	*Error Acknowledgment Output - Indicates an abnormal cycle termination.*
INTA_O	1	O	*Interrupt Output A*
M_ADDR_O	32	O	*Address Output*
M_DATA_I	32	I	*Data Input*
M_DATA_O	32	O	*Data Output*
M_SEL_O	4	O	*Select Output Array - Indicates which bytes are valid on the data bus. Whenever this signal is not 1111b during a valid access, the ERR_I is asserted.*
M_WE_O	1	O	*Write Output - Indicates a Write Cycle when asserted high or a Read Cycle when asserted low.*
M_STB_O	1	O	*Strobe Output - Indicates de beginning of a valid transfer cycle.*
M_CYC_O	1	O	*Cycle Output - Indicates that a valid bus cycle is in progress.*
M_ACK_I	1	I	*Acknowledgment Input - Indicates a normal Cycle termination.*
Port	Width	Direction	*Description*
M_ERR_I	1	I	*Error Acknowledgment Input - Indicates an abnormal cycle termination.*

Tabela 6.3 - Sinais de interface com o PHY [22].

Port	*Width*	*Direction*	*Description*
MTxClk	1	I	*Transmit Nibble or Symbol Clock. The PHY provides the MTxClk signal. It operates at a frequency of 25 MHz (100 Mbps) or 2.5 MHz (10 Mbps). The clock is used as a timing reference for the transfer of MTxD[3:0], MtxEn, and MTxErr.*
MTxD[3:0]	4	O	*Transmit Data Nibble. Signals are the transmit data nibbles. They are synchronized to the rising edge of MTxClk. When MTxEn is asserted, PHY accepts the MTxD.*
MTxEn	1	O	*Transmit Enable. When asserted, this signal indicates to the PHY that the data MTxD[3:0] is valid and the transmission can start. The transmission starts with the first nibble of the preamble. The signal remains asserted until all nibbles to be transmitted are presented to the PHY. It is deasserted prior to the first MTxClk, following the final nibble of a frame.*
MTxErr	1	O	*Transmit Enable. When asserted, this signal indicates to the PHY that the data MTxD[3:0] is valid and the transmission can start. The transmission starts with the first nibble of the preamble. The signal remains asserted until all nibbles to be transmitted are presented to the PHY. It is deasserted prior to the first MTxClk, following the final nibble of a frame.*
MRxClk	1	I	*Transmit Enable. When asserted, this signal indicates to the PHY that the data MTxD[3:0] is valid and the transmission can start. The transmission starts with the first nibble of the preamble. The signal remains asserted until all nibbles to be transmitted are presented to the PHY. It is deasserted prior to the first MTxClk, following the final nibble of a frame.*
MRxDV	1	I	*Receive Data Valid. The PHY asserts this signal to indicate to the Rx MAC that it is presenting the valid nibbles on the MRxD[3:0] signals. The signal is asserted synchronously to the MRxClk. MRxDV is asserted from the first recovered nibble of the frame to the final recovered nibble. It is then deasserted prior to the first MRxClk that follows the final nibble.*
MRxD[3:0]	4	I	*Receive Data Nibble. These signals are the receive data nibble. They are synchronized to the rising edge of MRxClk. When MRxDV is asserted, the PHY sends a data nibble to the*

			Rx MAC. For a correctly interpreted frame, seven bytes of a preamble and a completely formed SFD must be passed across the interface.
MRxErr	1	I	*Receive Error. The PHY asserts this signal to indicate to the Rx MAC that a media error was detected during the transmission of the current frame. MRxErr is synchronous to the MRxClk and is asserted for one or more MRxClk clock periods and then deasserted.*
MColl	1	I	*Collision Detected. The PHY asynchronously asserts the collision signal MColl after the collision has been detected on the media. When deasserted, no collision is detected on the media.*
MCrS	1	I	*Carrier Sense. The PHY asynchronously asserts the carrier sense MCrS signal after the medium is detected in a non-idle state. When deasserted, this signal indicates that the media is in an idle state (and the transmission can start).*
MDC	1	O	*Management Data Clock. This is a clock for the MDIO serial data channel.*
MDIO	1	I/O	*Management Data Input/Output. Bi-directional serial data channel for PHY/STA communication.*

Tabela 6.4 - Lista de registros [22].

Name	*Address*	*Width*	*Access*	*Description*
MODER	0x00	32	RW	*Mode Register*
INT_SOURCE	0x04	32	RW	*Interrupt Source Register*
INT_MASK	0x08	32	RW	*Interrupt Mask Register*
IPGT	0x0C	32	RW	*Back to Back Inter Packet Gap Register*
IPGR1	0x10	32	RW	*Non Back to Back Inter Packet Gap Register 1*
IPGR2	0x14	32	RW	*Non Back to Back Inter Packet Gap Register 2*
PACKETLEN	0x18	32	RW	*Packet Length (minimum and maximum) Register*
COLLCONF	0x1C	32	RW	*Collision and Retry Configuration*
TX_BD_NUM	0x20	32	RW	*Transmit Buffer Descriptor Number*

Name	*Address*	*Width*	*Access*	*Description*
CTRLMODER	0x24	32	RW	*Control Module Mode Register*
MIIMODER	0x28	32	RW	*MII Mode Register*
MIICOMAND	0x2C	32	RW	*MII Commend Register*
MIIADDRESS	0x30	32	RW	*MII Address Register - Contains the PHY address and the register within the PHY address*
MIITX_DATA	0x34	32	RW	*MII Transmit Data The data to be transmitted to the PHY*
MIIRX_DATA	0x38	32	RW	*MII Receive Data The data received from the PHY*
MIISTATUS	0x3C	32	RW	*MII Status Register*
MAC_ADDR0	0x40	32	RW	*MAC Individual Address0 The LSB four bytes of the individual address are written to this register*
MAC_ADDR0	0x44	32	RW	*MAC Individual Address1 The LSB four bytes of the individual address are written to this register*
ETH_HASH0_ADR	0x48	32	RW	*HASH0 Register*
ETH_HASH1_ADR	0x4C	32	RW	*HASH1 Register*
ETH_TXCTRL	0x50	32	RW	*Transmit Control Register*

ANEXO 3 – CÓDIGO FONTE

Bloco BCI_top – *Top Level* do projeto

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

use WORK.AVRuCPackage.all;

entity BCI_top is
	port
	(
		reset, clock: in std_logic;
		clk_tx, clk_rx: in std_logic;
		reset_phy: out std_logic;
		MRxD: in std_logic_vector(3 downto 0);
		MRxDV, MCrs, MColl, MRxErr, md_pad_i: in std_logic;
		MTxEn, Mdc_O, out_int_o, md_pad_o, md_padoe_o: out std_logic;
		MTxD: out std_logic_vector(3 downto 0);
		led3, led5, st, start_out, ok : out std_logic

	);
end BCI_top;

architecture rtl of BCI_top is

signal vector_ctrl, vector, vector_d: std_logic_vector(7 downto 0);
signal reset_out: std_logic;


component BCI is
	port
	(
		reset, clock, clk_tx, clk_rx : in std_logic;
		vector, vector_ctrl, vector_d : inout std_logic_vector(7 downto 0);
		MRxD: in std_logic_vector(3 downto 0);
		MRxDV, MCrs, MColl, MRxErr, md_pad_i: in std_logic;
		MTxEn, Mdc_O, out_int_o, md_pad_o, md_padoe_o: out std_logic;
		MTxD: out std_logic_vector(3 downto 0);
		led3, led5, st, start_out, ok: out std_logic
	);
end component;

component reset_gen is
	port
	(
```

```
                reset, clock: in std_logic;
                reset_out: out std_logic
        );
end component;

begin

u1: component BCI
        port map
        (
                reset => reset_out,
                clock => clock,
                clk_tx => clk_tx,
                clk_rx => clk_rx,
                vector =>  vector,
                vector_ctrl =>  vector_ctrl,
                vector_d => vector_d,
                led3 => led3,
                led5 =>  led5,
                st => st,
                start_out => start_out,
                MRxD => MRxD,
                MRxDV => MRxDV,
                MCrs => MCrs,
                MColl => MColl,
                MRxErr => MRxErr,
                MTxEn => MTxEn,
                Mdc_O => Mdc_O,
                MTxD => MTxD,
                out_int_o => out_int_o,
                md_pad_i =>  md_pad_i,
                md_pad_o =>  md_pad_o,
                md_padoe_o => md_padoe_o,
                ok => ok
        );

u2: component clock_div
port map
         (
        reset =>  reset,
        clock => clock,
        reset_out => reset_out
        );

reset_phy<=reset;

end rtl;
```

Bloco BCI

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

use WORK.AVRuCPackage.all;
entity BCI is
	port
	(
		reset, clock, clk_tx, clk_rx : in std_logic;
		vector, vector_ctrl, vector_d : inout std_logic_vector(7 downto 0);
		MRxD: in std_logic_vector(3 downto 0);
		MRxDV, MCrs, MColl, MRxErr, md_pad_i: in std_logic;
		MTxEn, Mdc_O, out_int_o, md_pad_o, md_padoe_o: out std_logic;
		MTxD: out std_logic_vector(3 downto 0);
		led3, led5, st, start_out, ok: out std_logic
	);
end BCI;

architecture rtl of BCI is

	signal out_cs_prom: std_logic;
	signal dat, addr, size_i, m_wb_dat_i, sg_wb_dat_o, out_eth_ma_bw_dat_o, data_in_ram:
	std_logic_vector (31 downto 0);
	signal sel, m_wb_sel_o: std_logic_vector(3 downto 0);
	signal operation: std_logic;
	signal start, stop, cyc, stb, ack, m_wb_we_o, m_wb_cyc_o, m_wb_stb_o, m_wb_err_i,
	mtx_clk_pad_i, mtxen_pad_o, mtxerr_pad_o: std_logic;
	signal wb_dat_o, m_wb_adr_o, m_wb_dat_o: std_logic_vector(31 downto 0);
	signal int_o, wb_err_o: std_logic;
	signal reset_n: std_logic;
	signal m_wb_ack_i, v, sn: std_logic;
	signal f: integer:=1;
	signal ptr: std_logic_vector(6 downto 0);
	signal m_wb_cyc_o_ant, fim, cl_bd: std_logic;
	signal out_ram_ex, ram_ex: std_logic_vector(15 downto 0);
	signal ct2, h: integer:=0;
	signal cp_sg_core_pc: std_logic_vector(15 downto 0);
	signal aux_portd: std_logic_vector (7 downto 0);

	component top is
		port
		(
			eset : in std_logic;
			clk   : in std_logic;
			fim: in std_logic;
			top_porta: inout std_logic_vector(7 downto 0);
			top_portb: inout std_logic_vector(7 downto 0);
```

```
                top_portd: inout std_logic_vector(7 downto 0);
                aux_portd: in std_logic_vector (7 downto 0);
                ram_ex: in std_logic_vector(15 downto 0);
                out_cs_prom: out std_logic
        );
end component;

component eth_top is
        port
        (
                -- WISHBONE common
                wb_clk_i, wb_rst_i: in std_logic;
                wb_dat_i: in std_logic_vector(31 downto 0);
                wb_dat_o: out std_logic_vector(31 downto 0);
                wb_err_o: out std_logic;

                -- WISHBONE slave
                wb_adr_i: in std_logic_vector(11 downto 2);
                wb_sel_i: in std_logic_vector( 3 downto 0);
                wb_we_i, wb_cyc_i, wb_stb_i: in std_logic;
                wb_ack_o: out std_logic;

                -- WISHBONE master
                m_wb_adr_o: out std_logic_vector(31 downto 0);
                m_wb_sel_o: out std_logic_vector(3 downto 0);
                m_wb_we_o: out std_logic;
                m_wb_dat_i: in std_logic_vector(31 downto 0);
                m_wb_dat_o: out std_logic_vector(31 downto 0);
                m_wb_cyc_o, m_wb_stb_o: out std_logic;
                m_wb_ack_i, m_wb_err_i: in std_logic;

                -- Tx
                mtx_clk_pad_i: in std_logic;
                mtxd_pad_o: out std_logic_vector(3 downto 0);
                mtxen_pad_o, mtxerr_pad_o: out std_logic;

                -- Rx
                mrx_clk_pad_i: in std_logic;
                mrxd_pad_i: in std_logic_vector( 3 downto 0);
                mrxdv_pad_i, mrxerr_pad_i: in std_logic;

                -- Common Tx and Rx
                mcoll_pad_i, mcrs_pad_i, md_pad_i: in std_logic;
                mdc_pad_o, md_pad_o, md_padoe_o, int_o: out std_logic
        );
end component;

component ctrl_ack is
        port
        (
                clock, reset, ramwe, v: in std_logic;
```

```
                eth_ma_bw_dat_o: in std_logic_vector(31 downto 0);
                address: in std_logic_vector(6 downto 0);
                out_eth_ma_bw_dat_o: out std_logic_vector(31 downto 0);
                ack_out: out std_logic
        );
    end component;

    component ctrl_mem_ex is
        port
        (
                clock, reset, ack_i: in std_logic;
                out_eth_ma_bw_dat_o: in std_logic_vector(31 downto 0);
                address_rd: in std_logic_vector(15 downto 0);
                out_ram_ex: out std_logic_vector(15 downto 0)
        );
    end component;

begin

    u1:component top  port map
    (
        reset => reset,
        clk => clock,
        top_porta => vector_ctrl,
        top_portb =>  vector,
        top_portd => vector_d,
        aux_portd => aux_portd,
        fim => fim,
        ram_ex => ram_ex,
        out_cs_prom => out_cs_prom
    );

    u2:component eth_top  port map
    (
        -- WISHBONE common
        wb_clk_i =>  clock,
        wb_rst_i => reset_n,
        wb_dat_i => dat,
        wb_dat_o => wb_dat_o,
        wb_err_o => wb_err_o,

        -- WISHBONE slave
        wb_adr_i => addr(11 downto 2),
        wb_sel_i => sel,
        wb_we_i => operation,
        wb_cyc_i => cyc,
        wb_stb_i => stb,
        wb_ack_o => ack,

        -- WISHBONE master
        m_wb_adr_o => m_wb_adr_o,
```

```
        m_wb_sel_o => m_wb_sel_o,
        m_wb_we_o => m_wb_we_o,
        m_wb_dat_i => m_wb_dat_i,
        --m_wb_dat_i => wb_dat_o,
        m_wb_dat_o => m_wb_dat_o,
        m_wb_cyc_o => m_wb_cyc_o,
        m_wb_stb_o => m_wb_stb_o,
        m_wb_ack_i => m_wb_ack_i,
        --m_wb_ack_i => ack,
        m_wb_err_i => m_wb_err_i,
        --m_wb_err_i => err,

        -- Tx
        mtx_clk_pad_i => clk_tx, --
        mtxd_pad_o => MTxD,
        mtxen_pad_o => MTxEn,
        mtxerr_pad_o => mtxerr_pad_o,

        -- Rx

        mrx_clk_pad_i => clk_rx,
        mrxd_pad_i =>  MRxD,
        mrxdv_pad_i =>  MRxDV,
        mrxerr_pad_i => MRxErr,

        -- Common Tx and Rx
        mcoll_pad_i => MColl,
        mcrs_pad_i =>  MCrs,
        md_pad_i => md_pad_i,
        mdc_pad_o => Mdc_O,
        md_pad_o => md_pad_o,
        md_padoe_o => md_padoe_o,
        int_o => int_o
);

u3: component ctrl_ack port map
(
        clock => clock,
        reset => reset,
        ramwe => m_wb_we_o,
        v => v,
        eth_ma_bw_dat_o => m_wb_dat_o,
        address => ptr,
        out_eth_ma_bw_dat_o => out_eth_ma_bw_dat_o,
        ack_out => m_wb_ack_i
);

u4: component ctrl_mem_ex port map
(
        clock => clock,
        reset => reset,
```

```
        ack_i => m_wb_ack_i,
        out_eth_ma_bw_dat_o => data_in_ram,
        address_rd => cp_sg_core_pc,
        out_ram_ex => out_ram_ex
);

process(reset, clock)
variable i: integer;
variable t: std_logic_vector(7 downto 0);
variable disp: std_logic;
begin
        if (reset='0') then
                reset_n<=not reset;
                i:=0;
                f<=1;
                led3<='1';
                led5<='1';
                start<='0';
                stop<='0';
                sn<='0';
                fim<='0';
                addr<=X"00000000";
                dat<= X"00000000";
                operation<='1';
                sel<=X"F";
                v<='0';
                ptr<="0000000";
                cl_bd<='0';
                disp:='0';
                ct2<=0;
                aux_portd<="00000000";
                data_in_ram<=X"00000000";
                 h<=0;
                ok<='1';
         elsif (clock'event and clock='1') then
                reset_n<=not reset;
                sel<=X"F";
                if(vector_ctrl(7)='1') then
                        start<='1';
                        stop<='0';
                        f<=1;
                        if(vector_ctrl(1)='1') then
                                operation<='1';
                        elsif(vector_ctrl(1)='0') then
                                operation<='0';
                        end if;
                elsif(vector_ctrl(0)='1')then  --stop
                        start<='0';
                        stop<='1';
                end if;
```

```
m_wb_cyc_o_ant<=m_wb_cyc_o;
if(m_wb_cyc_o='1')then
        v<='1';
else
        v<='0';
end if;

if(m_wb_cyc_o='1' and m_wb_cyc_o_ant='0') then
        ptr<=ptr+1;
end if;

if(m_wb_ack_i = '1') then
        data_in_ram<=out_eth_ma_bw_dat_o;

end if;

if(out_cs_prom='1') then
        led5<='0';
else
        led5<='1';
end if;

if(vector_ctrl(5)='1') then
        ok<='0';
end if;
if(vector_ctrl(6)='1') then
        led3<='0';
        addr<=X"00000004"; --
        dat<=X"00000004";   --
        operation<='1';                 --
        sn<='1';          -- sequencia para limpar a interrupção
        cl_bd<='1';               -- seta limpeza do bd
        fim<='0';
        ptr<="0000000";
else
        led3<='1';
end if;


if(cl_bd='1') then                --limpa o bd
        if(ct2=0) then
                addr<=X"000007F8";
                dat<=X"00004000";
                operation<='1';
                sn<='1';
                ct2<=ct2+1;
        elsif(ct2=1) then
                addr<=X"000007F8";
                dat<=X"00006000";
                operation<='1';
                sn<='1';
```

```
                ct2<=ct2+1;
            elsif(ct2=2) then
                addr<=X"000007F8";
                dat<=X"0000E000";
                operation<='1';
                sn<='1';
                ct2<=ct2+1;
            elsif(ct2=3) then
                addr<=X"000007FC";
                dat<=X"00000000";
                operation<='1';
                sn<='1';          --seta o cyc e stb
                ct2<=0;
                cl_bd<='0';
            end if;
        else
            sn<='0';

        end if;

 if(int_o='1') then
        aux_portd<=X"FF";
        if(cp_sg_core_pc=X"0008") then
            disp:='1';
        end if;

        if(disp='1') then
            disp:='1';
            fim<='1';
        end if;
end if;


if(start='1') then
        i:=i+1;
        if(i=2) then
            i:=0;
            if(operation='1') then
                if(f=1) then
                    f<=f+1;
                elsif(f=2) then
                    addr(31 downto 24)<=vector;--4/4 endereço
                    f<=f+1;
                elsif(f=3) then
                    addr(23 downto 16)<=vector;--3/4 endereço
                    f<=f+1;
                elsif(f=4) then
                    addr(15 downto 8)<=vector;--2/4 endereço
                    f<=f+1;
                elsif(f=5) then
                    addr(7 downto 0)<=vector;--1/4 endereço
```

```
                                        f<=f+1;
                                elsif(f=6) then
                                        dat(31 downto 24)<=vector;--4/4 endereço
                                        f<=f+1;
                                elsif(f=7) then
                                        dat(23 downto 16)<=vector;--3/4 endereço
                                        f<=f+1;
                                elsif(f=8) then
                                        dat(15 downto 8)<=vector;--2/4 endereço
                                        f<=f+1;
                                elsif(f=9) then
                                        dat(7 downto 0)<=vector;--1/4 endereço
                                        f<=f+1;
                                elsif(f=10) then
                                        start<='0';

                                        f<=f+1;
                                end if;
                        elsif(operation='0') then
                                if(f=1 ) then
                                        f<=f+1;
                                elsif(f=2) then
                                        addr(31 downto 24)<=vector;--4/4 endereço
                                        f<=f+1;
                                elsif(f=3) then
                                        addr(23 downto 16)<=vector;--3/4 endereço
                                        f<=f+1;
                                elsif(f=4) then
                                        addr(15 downto 8)<=vector;--2/4 endereço
                                        f<=f+1;
                                elsif(f=5) then
                                        addr(7 downto 0)<=vector;--1/4 endereço
                                        f<=f+1;
                                elsif(f=6) then
                                   start<='0';
                                  f<=f+1;
                                end if;
                        end if;
                end if;


        end if;

end process;

process(reset, clock, vector_ctrl, ack, f)

begin
                if(reset='0') then
                        cyc<='0';
```

```
                stb<='0';
            elsif(clock 'event and clock='1') then
                if (ack='1') then
                    cyc<='0';
                    stb<='0';
                    sg_wb_dat_o<=wb_dat_o;
                elsif((operation='1' and f=10) or (operation='0' and f=6) or(sn='1'))then
                    cyc<='1';
                    stb<='1';
                end if;
            end if;

end process;

st<=stop;
start_out<=start;
out_int_o<=int_o;
ram_ex<=out_ram_ex;

end rtl;
```

Bloco reset_gen

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

use WORK.AVRuCPackage.all;


entity reset_gen is
    port(
        reset, clock: in std_logic;
        reset_out: out std_logic
    );
end reset_gen;

architecture rtl of clock_div is
    signal temp: std_logic:='1';
    signal i, j: integer:=0;
    signal k: std_logic:='0';

begin
    process (reset, clock)
    variable aux: std_logic:='1';
    begin
        if(reset= '0') then
            i<=0;
            j<=0;
            temp<='0';
```

```
                k<='1';
                reset_out<='0';
        elsif(clock 'event and clock='1') then
                if(k='1') then
                        j<=j+1;
                        reset_out<='0';
                        if(j=2500000) then
                                reset_out<='1';
                                k<='0';
                        end if;
                else
                        j<=0;
                        reset_out<=reset;
                end if;
        end if;
end process;

end rtl;
```

Bloco ctrl_ack

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

use WORK.AVRuCPackage.all;

entity ctrl_ack is
        port(
                clock, reset, ramwe, v: in std_logic;
                eth_ma_bw_dat_o: in std_logic_vector(31 downto 0);
                address: in std_logic_vector(6 downto 0);
                out_eth_ma_bw_dat_o: out std_logic_vector(31 downto 0);
                ack_out: out std_logic
        );
end ctrl_ack;

architecture rtl of ctrl_ack is
        signal temp_v: std_logic;
        component mem_pack is
                port(
                        cp2: in std_logic;
                        address : in  std_logic_vector (6 downto 0);
                        ramwe   : in  std_logic;
                        din     : in  std_logic_vector (31 downto 0);
                        dout    : out     std_logic_vector (31 downto 0)
                );
        end component;
```

```
begin
        u1: component mem_pack port map
        (
                cp2 => clock,
                address => address,
                ramwe => ramwe,
                din => eth_ma_bw_dat_o,
                dout => out_eth_ma_bw_dat_o
        );

        process(reset, clock, v)
        variable j: integer:=0;
        begin
                if(reset='0') then
                        j:=0;
                         ack_out<='0';
                elsif(clock 'event and clock='1') then

                        if(v='1' and temp_v='0') then
                                ack_out<='0';
                                j:=j+1;
                        elsif(j=1)then
                                ack_out<='1';
                                j:=j+1;
                        elsif(j=2) then
                                ack_out<='0';
                                j:=0;
                        else
                                ack_out<='0';
                        end if;

                        temp_v<=v;
                end if;
        end process;

end rtl;
```

Bloco mem_pack

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

use WORK.AVRuCPackage.all;

entity mem_pack is
```

```
	port(
		cp2: in std_logic;
		address: in  std_logic_vector (6 downto 0);
		ramwe : in  std_logic;
		din: in  std_logic_vector (31 downto 0);
		dout: out std_logic_vector (31 downto 0)
	);
end mem_pack;

architecture Beh of mem_pack is
	type RAMFileType is array(511 downto 0) of std_logic_vector(din'range);
	signal RAMFile : RAMFileType := (others  =>  x"00000000");

begin

	DataWrite:process(cp2)
	begin
		if cp2='1' and cp2'event then		-- Clock
			if ramwe='1' then			-- Clock enable
				RAMFile(CONV_INTEGER(address)) <= din;
			end if;
		end if;
	end process;
	dout <= RAMFile(CONV_INTEGER(address));
end Beh;
```

Bloco mem_ex

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

use WORK.AVRuCPackage.all;

entity mem_ex is
	port(
		cp2: in std_logic;
		address, address_rd : in  std_logic_vector ( 7 downto 0);
		ramwe: in  std_logic;
		din: in  std_logic_vector (15 downto 0);
		dout: out std_logic_vector (15 downto 0)
	);
end mem_ex;

architecture Beh of mem_ex is
	type RAMFileType is array(511 downto 0) of std_logic_vector(din'range);
	signal RAMFile : RAMFileType := (others  =>  x"0000");

begin
```

```
	DataWrite:process(cp2)
	begin
		if cp2='1' and cp2'event then		-- Clock
			if ramwe='1' then			-- Clock enable
				RAMFile(CONV_INTEGER(address)) <= din;
			end if;
		end if;
	end process;
	dout <= RAMFile(CONV_INTEGER(address_rd));

end Beh;
```

Bloco control

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

use WORK.AVRuCPackage.all;

entity control is generic(RAMSize : positive := 128);
	port(
		reset, clock, fim: in std_logic;
		core_pc_prom, ram_ex: in std_logic_vector(15 downto 0);
		core_ram_addr: in std_logic_vector(6 downto 0);
		inst_out_out: out std_logic_vector(15 downto 0);
		data_out: buffer std_logic_vector(15 downto 0);
		din: in std_logic_vector(7 downto 0);
		dout: out std_logic_vector(7 downto 0);
		ctrl, ramwe: in std_logic;
		out_state: out state_ram;
		out_cs_prom: out std_logic;
		out_addr_ram: out std_logic_vector(6 downto 0)
	);
end control;

architecture rtl of control is

	component PROM is port (
		address_in: in  std_logic_vector (15 downto 0);
		cs_prom: in std_logic;
		data_out, inst_out: out std_logic_vector (15 downto 0));
	end component;

	component ram2 is
		 generic(RAMSize :positive);
```

```
                port (
                        reset, cp2: in std_logic;
                        address: in  std_logic_vector (LOG2(RAMSize)-1 downto 0);
                        ramwe : in  std_logic;
                        din: in  std_logic_vector (7 downto 0);
                        dout: out std_logic_vector (7 downto 0));
        end component;

        signal cs_prom: std_logic;
        signal sg_prom_addr, temp, out_ram2, inst_out, out_fsm: std_logic_vector(15 downto
        0);
        signal sg_ram_addr: std_logic_vector(6 downto 0);
        signal sg_ram_addr_mux: std_logic_vector(6 downto 0);

begin

        u1:component PROM port map(
                address_in  =>  sg_prom_addr,
                cs_prom =>  cs_prom,
                data_out =>  data_out,
                inst_out => inst_out
        );

        u2:component ram2
                generic map(RAMSize  =>  RAMSize)
                port map(
                        reset => reset,
                        cp2 => clock,
                        address => sg_ram_addr,
                        ramwe => ramwe,
                        din => din,
                         dout => dout
);

        process(clock, reset, core_pc_prom, ctrl)
                variable inst: std_logic_vector(15 downto 0);
                begin
                        if(reset='0') then
                                inst:=x"0000";
                        end if;

                        if(core_pc_prom >= x"0200" or ctrl = '1')then
                                cs_prom<='1';
                                sg_prom_addr<=core_pc_prom;
                                inst:=data_out;
                        elsif(core_pc_prom >= x"0000" and core_pc_prom < x"0200" and ctrl
                = '0') then
                                cs_prom<='0';
                                inst:=ram_ex;
                        end if;
```

```
                inst_out_out<=inst;
                out_cs_prom<=cs_prom;

        end process;
end rtl;
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] MORAES, F., CALAZANS, N., MÖLLER, L., BRIÃO, E., CARVALHO, E. **"Dynamic and Partial Reconfiguration in FPGA SoCs:Requirements Tools and Case Study"**. In: W. Rosenstiel (Org.) Reconfigurable Computing. Chapter approved to be published in Kluwer Academic Press Book, 2005.

[2] DEHON, A., WAWRZYNEK, J. **"Reconfigurable Computing: What, Why, and Implications for Design Automation"**, 36th Design Automation Conference. New Orleans, 1999, p. 610-615.

[3] ENCINAS, W. S. Jr., DUEÑAS, C. A. **"Functional Verification in 8-bit Microcontrollers: A case Study"**. XVI SBMicro, 2001.

[4] SILVA, K. R. G., MELCHER, E. U. K., ARAÚJO, G. C. S., PIMENTA, V. A. **"An Automatic Testbench Generation Tool for a SystemC Functional Verification Methodology"**. SBCCI 2004, PE, Brasil, p.66-70.

[5] FOURNIER, L., ARBETMAN, Y., LEVINGER, M. **"Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator"**. Design, Automation and Test in Europe Conference, Munich, March 1999, p. 434-441.

[6] TOCCI, R.J., Widmer, N. S. **Sistemas Digitais: Princípios e Aplicações**, 8 Ed p. 134-135. São Paulo: Prentice Hall, 2003.

[7] PEDRONI, V. A. **Digital Electronics and Design with VHDL**. Burlington: Morgan Kaufmann, 2008, p. 467-480.

[8] SCHUNK, L. M., LUPPI, A. **Microcontroladores AVR: Teoria e Aplicações Práticas**, p. 19-20. São Paulo: Érica, 2001.

[9] TANENBAUM, A. S. **Organização Estruturada de Computadores**. Rio de Janeiro: LTC, 1999, p. 27-28.

[10] KUROSE, J. F., Ross, K. W. **Redes de Computadores e a Internet: Uma Nova Abordagem**. São Paulo: Addison Wesley, 2003, p. 324-330.

[8] MOKARZEL, M. P., CARNEIRO, K. P. M. **Internet Embedded TCP/IP para Microcontroladores**. São Paulo: Editora Érica, 2004, p. 26-28.

[9]AXELSON, J. **Embedded Ethernet and Internet Complete: Designing and Programming Small devices for Networking**. Madison: Lakeview Research LLC, 2003, p. 20-28.

[10] **WISHBONE SoC Architecture Specification, Revision B.3.** OpenCores, 2002.

[11] **Datasheet Intel 8051 8-BIT Control-Oriented Microcontrollers**.

[12] **Datasheet Zilog Z80 Microcontroller**, 2004.

[13] **Datasheet Motorola 6805 Microcontroller**, 1995.

[14] **Datasheet Atmel AVR 90S1200 Microcontroller**, 2002.

[15] **Datasheet Atmel AVR 90S2313 Microcontroller**.

[16] **Datasheet Atmel AVR ATmega103 Microcontroller**, 2007.

[17] **AVR_CORE Description**, OpenCores, 2002.

[18] MOHOR, I. ***IP Core Ethernet*** **Core Design Document**, 2002.

[19] MOHOR, I. ***IP Core Ethernet*** **Core Specification**, 2002.

[20] Degioanni, L., Varenni, G., Risso, F. Bruno, J. **WinDump: tcpdump for Windows**, 2006. Disponível em: http://www.winpcap.org/windump

[21] HENG, A. Y. C. **Bittwist**. Faculty of Information Technology, Multimedia University, 2006. Disponível em: http://bittwist.sourceforge.net

[22] HENG, A. Y. C. **Bittwiste**. Faculty of Information Technology, Multimedia University, 2006. Disponível em: http://bittwist.sourceforge.net

RESUMO:

O objetivo deste trabalho é desenvolver e implementar um método complementar para avaliação de um microcontrolador de código aberto. Para que, o mesmo possa ser usado, posteriormente, em projetos inovadores com diferentes aplicações, porém com algumas restrições para uso comercial.

Para o desenvolvimento deste projeto, o primeiro passo consistiu em pesquisar os vários microcontroladores disponíveis, para então selecionar o mais completo, contendo documentação, compilador gratuito, testbench, etc. O microcontrolador selecionado foi o ATmega103, de 8 bits, descrito em VHDL.

Para garantir que o *core* funciona corretamente é necessário realizar vários testes. Para isso foram criados códigos em C e Assembly, contendo instruções que foram gravadas na memória de programa do microcontrolador.

Com o objetivo de tornar essa metodologia independente de plataforma, fabricante e do *software* de *design* utilizado, um bloco Ethernet foi integrado ao projeto, onde os dados carregados em uma memória de execução são transmitidos através de pacotes pela rede local.

Para que a comunicação entre o bloco Ethernet e o microcontrolador fosse possível, desenvolveu-se um bloco em VHDL para realizar a interface e o controle entre eles. Este é responsável, entre outras coisas, por realizar a troca entre a memória de programa e a RAM que contém o código teste a ser executado. Isso ocorre quando é detectada a chegada de um pacote. Desta maneira, o microcontrolador passa a executar as instruções da memória que contém o pacote e não mais da memória de programa antiga.

Com a utilização deste método, uma grande quantidade de códigos distintos pode ser verificada em hardware com maior facilidade e posteriormente comparada com os resultados esperados ou obtidos em simulação.

PALAVRAS-CHAVE

Avaliação, microcontrolador, código aberto, Ethernet, linguagem VHDL.

ÁREA/SUB-ÁREA DE CONHECIMENTO

3.04.00.00 – 7 Engenharia Elétrica

1.03.04.01 – 0 Hardware

1.03.03.00 – 6 Metodologias e Técnicas da Computação

Ano 2009

N°: 494