WILLIAM BARCELLOS

Análise da implementação de núcleos de código aberto de microcontroladores PIC16 em FPGA

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos, da
Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em
Eletrônica

ORIENTADOR: Prof. Dr. Maximiliam Luppe

São Carlos
2010

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

Barcellos, William
B242a Análise da implementação de núcleos de código aberto de microcontroladores PIC16 em FPGA / William Barcellos ; orientador Maximiliam Luppe. -- São Carlos, 2010.

Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2010.

1. Circuitos FPGA. 2. Microcontroladores. 3. Núcleo de código aberto. I. Título.

FOLHA DE APROVAÇÃO

Nome: William Barcellos

Título: "Análise da Implementação e de Núcleos de Código Aberto de Microcontroladores PIC16 em FPGA"

Trabalho de Conclusão de Curso defendido e aprovado em 30 / 11 / 2010,

com NOTA 10,0 (Dez, zero), pela comissão julgadora:

Prof. Dr. Evandro Luís Linhari Rodrigues - EESC/USP

Profa. Dra. Maria Stela Veludo de Paiva - EESC/USP

Prof. Associado Homero Schiabel
Coordenador da CoC-Engenharia Elétrica
EESC/USP

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO 5

CAPÍTULO 2 - MICROCONTROLADORES 7

CAPÍTULO 3 - FPGAS 11

3.1 AS ESTRUTURAS DE UMA FPGA 11
3.2 NÚCLEO DE FPGA 13
3.3 DESENVOLVIMENTO DE PROJETOS EM FPGA 14
3.3.1 Design 15
3.3.1.1 Captura de esquemático 15
3.3.1.2 Linguagem de descrição de hardware 15
3.3.2 Simulação 15
3.3.3 Síntese 16
3.3.4 Implementação 16

CAPÍTULO 4 - IMPLEMENTAÇÃO DE NÚCLEOS DE CÓDIGO ABERTO DE MICROCONTROLADORES PIC16 EM FPGA 17

4.1 PESQUISA DOS NÚCLEOS DE CÓDIGO ABERTO 17
4.2 METODOLOGIA DE IMPLEMENTAÇÃO E VERIFICAÇÃO DOS NÚCLEOS 18
4.2.1 Compilação do núcleo 18
4.2.2 Geração do programa de verificação 18
4.2.3 Associação do código de verificação à memória ROM 18
4.2.4 Implementação 19
4.2.5 Verificação 19
4.2.6 Simulação 19
4.3 CQPIC 19
4.3.1 Processo de verificação do núcleo 20
4.3.2 Resultados 20
4.4 MINI-RISC 21
4.4.1 Processo de verificação do núcleo 21
4.4.2 Resultados 22
4.5 PPX16 23
4.5.1 Processo de verificação do núcleo 23
4.5.2 Resultados 24
4.6 RISC5X 26
4.6.1 Processo de verificação do núcleo 26
4.6.2 Resultados 26
4.7 RISC8 28
4.7.1 Processo de verificação do núcleo 28
4.7.2 Resultados 28
4.8 RISC16F84 29
4.8.1 Processo de verificação do núcleo 29
4.8.2 Resultados 30
4.9 SLC1657 31
4.9.1 Processo de verificação do núcleo 31
4.9.2 Resultados 32
4.10 CLAIRISC 34
4.10.1 Processo de verificação do núcleo 34
4.10.2 Resultados 35
4.11 UPEM 36
4.11.1 Processo de verificação do núcleo 36
4.11.2 Resultados 36
4.12 ANÁLISE DOS NÚCLEOS 37
4.13 APLICAÇÃO DO NÚCLEO 39

CAPÍTULO 5 - CONCLUSÕES.....41
REFERÊNCIAS BIBLIOGRÁFICAS.....43
APÊNDICE A – ATRIBUIÇÃO EM VERILOG 45

Capítulo 1 - Introdução

Nos últimos anos, o crescimento, tanto em diversidade, quanto em densidade, dos dispositivos reconfiguráveis, e de suas respectivas ferramentas de desenvolvimento, tem favorecido a implementação de sistemas complexos e completos em lógica integrada e programável (SoC – *System on Chip*) em um curto espaço de tempo. Empresas como Atmel [1], Altera [2], Lattice [3], Cypress [4], Anadigm [5], entre outras, são alguns exemplos de empresas que desenvolvem soluções na área de sistemas reconfiguráveis, tanto digitais, como analógicos, ou ambos.

Da mesma forma, graças à densidade crescente dos dispositivos reconfiguráveis e à disponibilidade de ferramentas de desenvolvimento para estes, tem crescido o número de núcleos para as mais diversas aplicações, desde blocos de memória até processadores, passando por controladores de periféricos e interfaces de comunicação. A vantagem na utilização de núcleos está na sua reutilização, visto que um mesmo núcleo pode ser utilizado em diversos projetos, sem custo adicional nem gasto com tempo de desenvolvimento, o que o torna mais econômico e versátil [6].

Existem diversas famílias de dispositivos reconfiguráveis de diversos fabricantes. Por exemplo, a Atmel, além das famílias AT40K e AT6000 de dispositivos reconfiguráveis tipo FPGA, também possui uma linha de dispositivos reconfiguráveis que integra um microcontrolador tipo RISC, da própria Atmel. Esta família, conhecida por FPSLIC (*Field Programmable System Level Integrated Circuit*) e que compreende a família AT94K [7], é a combinação do dispositivo FPGA AT40K (com capacidade de até 40000 portas lógicas) com o microcontrolador RISC de oito bits da família AVR (com periféricos do tipo UART, Watchdog e Temporizadores/Contadores), além de 32Kbytes de memória SRAM. O dispositivo AT40K possui uma arquitetura peculiar, especialmente desenvolvida para a implementação de multiplicadores de vetor e matrizes, ideal para o processamento de sinais.

Na mesma linha, a Altera possui a família Excalibur [8] de dispositivos reconfiguráveis tipo FPGA, que integra um microcontrolador ARM922T, além da parte reconfigurável ser baseada na família APEX20KE. Além deste, a Altera também possui um núcleo flexível de um microprocessador, mais conhecido por Nios [9], que pode ser reconfigurado conforme a necessidade, podendo ser facilmente implementado nos dispositivos Cyclone, da própria Altera. Sua reconfigurabilidade permite, entre outras, a implementação de novas instruções e a definição da largura de dados de 16 ou 32 bits.

Mais recentemente, a Cypress lançou uma família de dispositivos reconfiguráveis baseados em microcontroladores. Diferentemente dos demais exemplos, estes dispositivos, chamados de PSoC [10], são realmente microcontroladores, mas possuem partes reconfiguráveis, tanto digital, como analógica. A parte reconfigurável digital permite a implementação de módulos de Temporizadores, Contadores, Geradores de Seqüências Pseudo-aleatórias, PWM, UART, etc., enquanto que a parte analógica permite a implementação de Conversores Analógico para Digital (ADC) e Digital para Analógico

(DAC), Comparadores, Filtros, etc. Estes módulos já são pré definidos e podem ter seus parâmetros de funcionamento configurados conforme a necessidade.

Cada uma destas empresas também possui, além dos dispositivos, as suas respectivas ferramentas de desenvolvimento: IDS e ProChip Designer (Atmel), Quartus II e SOPC Builder (Altera), PSoC Designer (Cypress) e ispLEVER (Lattice).

Conforme podemos verificar, existe uma tendência de mercado no desenvolvimento de dispositivos reconfiguráveis associados a microcontroladores, destinados à implementação de sistemas SoC. Neste sentido, o presente trabalho teve como objetivo verificar as vantagens e desvantagens na utilização de núcleos de código aberto de microcontroladores em FPGA.

O trabalho é relevante porque mostra as dificuldades na implementação dos núcleos. Não se pode simplesmente implementar o núcleo sem conhecê-lo, é necessário verificar se as instruções e funções estão funcionando corretamente. Também é importante conhecer suas características, pois existem núcleos que não possuem todas as características que o microcontrolador no qual foi baseado possui. Ao mostrar o processo de implementação de nove núcleos de microcontroladores, o trabalho pode servir como base para avaliar se a utilização de um núcleo de microcontrolador é, ou não, interessante para um projeto especifico.

Nos Capítulos 2 e 3 são apresentados conceitos teóricos sobre microcontroladores e FPGAs para uma melhor compreensão do trabalho.

No Capítulo 4 o processo de implementação, verificação do correto funcionamento e a análise dos núcleos são descritos, assim como a utilização de um dos núcleos no controle de uma esteira industrial.

No Capítulo 5 são apresentadas as vantagens e desvantagens na utilização de núcleos de microcontroladores em FPGA.

Capítulo 2 - Microcontroladores

Com o desenvolvimento da tecnologia de integração de circuitos eletrônicos, foi possível integrar as partes básicas de um computador: unidade central de processamento, unidades de armazenamento de informação e unidades de entrada e saída. Assim, foi possível implementar computadores mais compactos, confiáveis e baratos, dando origem aos microcomputadores.

Os primeiros microcomputadores foram implementados fisicamente utilizando diversos circuitos integrados com funções específicas: microprocessador, memória não-volátil, memória volátil e portas de entrada e saída.

O primeiro microprocessador foi o 4004, de 4 bits da Intel. Com o sucesso deste microprocessador, os projetistas passaram a solicitar unidades com maiores capacidades de processamento para aplicações mais sofisticadas. Depois do 4004, a Intel lançou os microprocessadores 8008, 8080 e 8085, de 8 bits. Outros fabricantes também lançaram microprocessadores para concorrer com a Intel, como 6800 da Motorola e o Z80 da Zilog.

Com o desenvolvimento contínuo da microeletrônica, um número cada vez maior de transistores podia ser integrado por unidade de área. Em certo ponto, todos os blocos básicos de um computador puderam ser integrados em uma única pastilha de silício, dando origem aos microcontroladores.

Atualmente existem microcontroladores com processadores de 8, 16 e 32 bits, e integram além das partes básicas de um microcomputador, módulos de controle USB, I^2C e UART, comparadores, conversores A/D, PWM, oscilador interno, entre outras funções.

Por serem limitados com relação à memória de dados, os microcontroladores são geralmente utilizados em aplicações específicas como automação residencial (telefones, forno de microondas, máquinas de lavar), em automação predial (elevadores, controladores de energia elétrica), em automação industrial (robótica, controladores de acesso restrito, relógios de ponto) e automação embarcada (computadores de bordo, alarmes) [11].

Existem dois tipos de arquiteturas de microcontroladores/microprocesadores: von Neumann e Harvard. Microcontroladores que utilizam a arquitetura von Neumann possuem uma única estrutura de armazenamento, que compreende as memórias de dados e de programa, e as duas memórias compartilham os barramentos de dados e de endereço, como pode ser visto na Figura 1. Como os barramentos são compartilhados, a CPU só consegue ou ler uma instrução, ou ler/escrever um dado, limitando a largura de banda de operação, o que é conhecido como Gargalo de von Neumann.

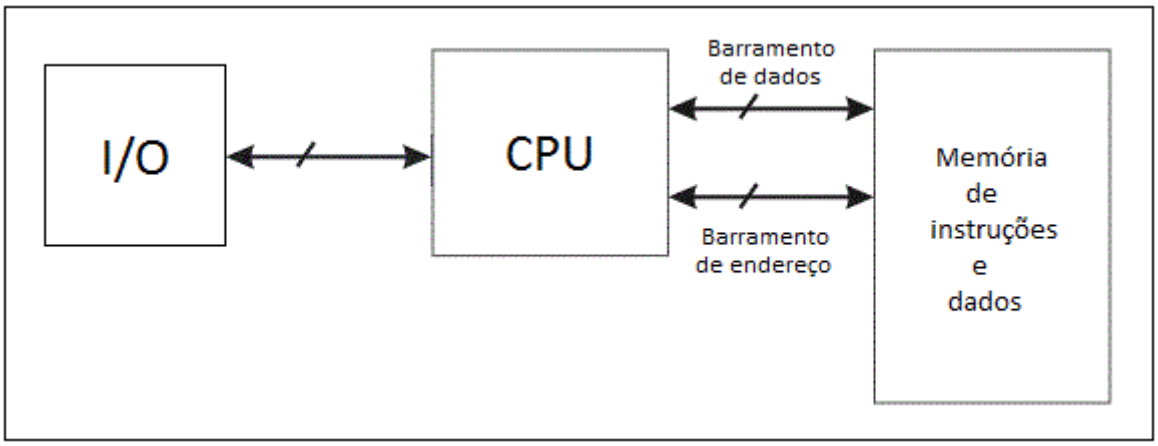

Figura 1 – Arquitetura von Neumann

Geralmente microcontroladores/microprocessadores com arquiterura von Neumann são também de arquiteturas do tipo CISC (*Complex Instruction Set Computer*). Na arquitetura CISC, as instruções podem ocupar espaços diferentes na memória de programa, podem ter durações diferentes e possuem muitas instruções, tornando os programas mais simples. Exemplos de microcontroladores com arquitetura von Neumann são: 4004, 8080, 8051, 8085 e Z80.

A arquitetura Harvard utiliza memórias fisicamente separadas para instruções e dados, requerendo barramentos de endereço e dados separados, como pode ser visto na Figura 2, o que possibilita a CPU ler uma instrução e ler/escrever um dado simultaneamente. Por possuir memórias fisicamente separadas, as caracteristicas das memórias de dados e programa podem ser diferentes, podendo ter barramentos de dados e/ou endereço diferentes, *timings* diferentes e tecnologias diferentes.

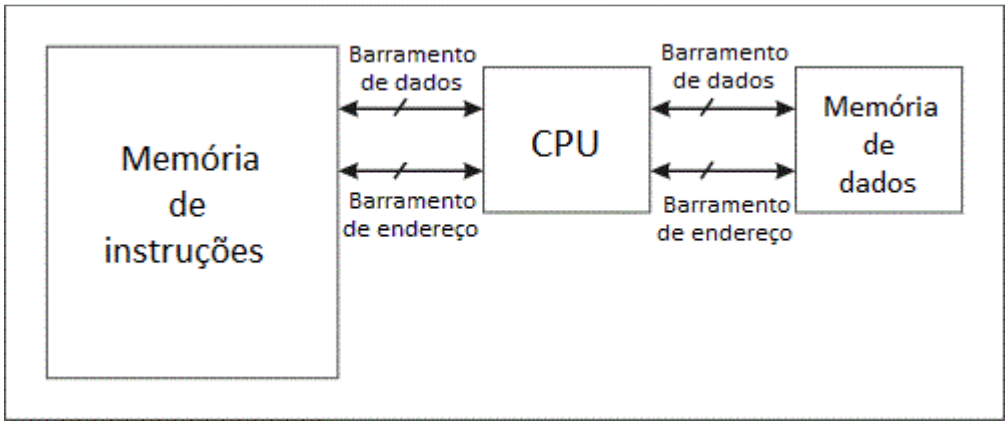

Figura 2 – Arquitetura Harvard

Instruções podem ser armazenadas em ROMs, enquanto dados requerem RAMs. Em alguns sistemas, a memória de instruções é muito maior que a memória de dados, então o barramento de endereço da memória de instruções é mais largo que o da memória de dados.

O armazenamento separado de instruções e dados possibilita que as memórias de dados e instruções possuam palavras de tamanhos diferentes, o que permite que uma instrução contenha um dado.

Geralmente microcontroladores com arquitetura Harvard são também de arquitetura do tipo RISC (*Reduced Instruction Set Computer*). Na arquitetura RISC, cada instrução ocupa o mesmo espaço na memória, e todas tem a mesma duração, com exceção das instruções que realizam salto. Por possuir poucas instruções, o programa torna-se mais complexo. Podem existir arquiteturas Von Neumann-RISC e Harvard-CISC, porém são menos comuns. Exemplos de microcontroladores/ microprocessadores com arquitetura Harvard são; 8086, 8088, PIC16F e PIC18F.

Capítulo 3 - FPGAs

Uma FPGA (*Field-Programmable Gate Array*) é um circuito integrado projetado para ser configurado pelo cliente após a sua fabricação. A configuração da FPGA é geralmente especificada utilizando Linguagem de Descrição de Hardware (HDL - Hardware Description Language), semelhante à utilizada para ASIC [12] (*Application-Specific Integrated Circuit*). FPGAs podem ser utilizadas para implementar qualquer lógica que um ASIC pode realizar.

A habilidade de atualizar suas funcionalidades após a venda, de reconfigurar partes do projeto, e o baixo custo não-recorrente de engenharia em relação a um projeto ASIC (apesar do custo unitário geralmente superior), oferece vantagens para muitas aplicações.

3.1 As estruturas de uma FPGA

Os três elementos básicos em uma FPGA são o bloco de lógica reconfigurável (CLB - *Configurable Logic Block*), as interconexões, e os blocos de entrada/saída (I/O - *Input/Output*). A estrutura básica de uma FPGA é ilustrada na Figura 3. Os blocos de I/O em todo o perímetro da estrutura fornecem acesso individual selecionável de entrada, saída ou bidirecional, ao mundo externo. A matriz distribuída de interconexões programáveis fornece interconexões aos CLBs e conexões aos blocos de I/O. FPGAs grandes podem ter dezenas de milhares de CLBs, além de memória e outros recursos.

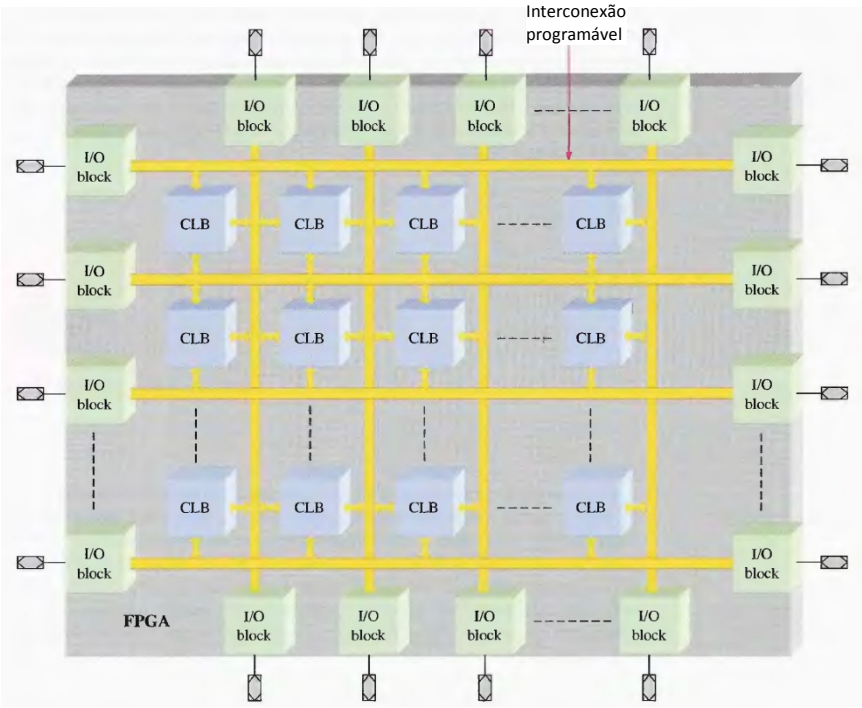

Figura 3 – Estrutura básica de uma FPGA (Floyd, 2006, p.629)

A maioria dos fabricantes de dispositivos de lógica programável produzem uma série de FPGAs que variam em densidade, consumo de energia, tensão de alimentação, velocidade, e até certo ponto variam em arquitetura. FPGAs são reprogramáveis e usam as tecnologias SRAM (volátil) ou antifuse (não-volátil) para programar as interconexões [13]. A densidade pode variar de centenas de CLBs a 180.000 CLBs, em encapsulamentos com até 1.000 pinos. A tensão de alimentação DC varia tipicamente de 1.2V a 2.5V, dependendo do dispositivo.

Normalmente um CLB consiste de vários módulos lógicos menores que são as unidades básicas de construção. A Figura 4 mostra um CLB dentro das linhas/colunas de interconexões programáveis globais que são utilizadas para conectar os CLBs. Cada CLB é constituído por múltiplos módulos lógicos menores e uma interconexão programável local que é utilizada para conectar os módulos lógicos dentro do CLB.

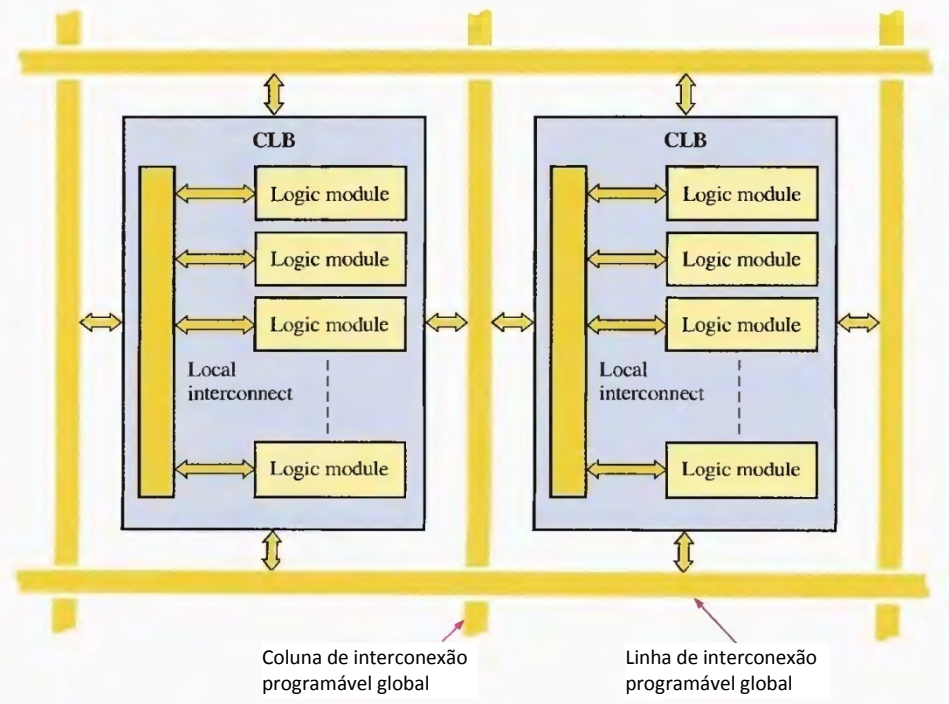

Figura 4 - configuração básica de um CLB dentro das linhas/colunas de interconexões programáveis globais (Floyd, 2006, p.629)

Um módulo lógico em um CLB pode ser configurado para lógica combinacional, lógica de registrador, ou uma combinação de ambos. Um flip-flop é parte da lógica associada e é utilizado para a lógica de registrador. O diagrama de blocos da Figura 5 mostra os componentes de um módulo lógico. O LUT (*Look-Up Table*) é um tipo de memória programável utilizada para gerar soma de produtos (SOP - *Sum Of Products*).

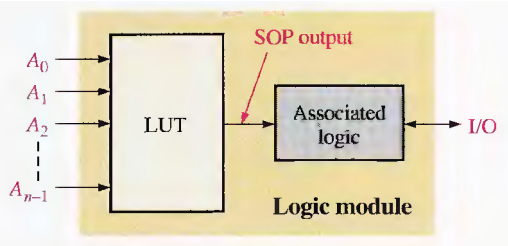

Figura 5 - Diagrama de bloco de um módulo lógico (Floyd, 2006, p.630)

3.2 Núcleo de FPGA

Existem dois tipos de núcleos de FPGA: núcleo rígido e núcleo flexível. Um núcleo rígido é uma parte de lógica em uma FPGA que é colocada pelo fabricante para fornecer uma função específica, e não pode ser reprogramada. Uma vantagem na utilização de núcleos rígidos é que um projeto implementado em uma FPGA utilizando um núcleo rígido ocupa menos espaço na FPGA do que o mesmo projeto implementado em campo por um usuário, resultando em uma economia de elementos lógicos e menor tempo de desenvolvimento. Além disto, as funções do núcleo rígido já foram exaustivamente testadas. A desvantagem dos núcleos rígidos é que as especificações são fixadas durante o processo de fabricação, e o consumidor não poderá alterar a lógica.

Núcleos rígidos são geralmente disponíveis para funções que são usualmente utilizadas como microprocessadores, interface padrão de I/O, e processadores digitais de sinais. Mais de um núcleo rígido pode ser programado em uma FPGA. A Figura 6 ilustra o conceito de um núcleo rígido rodeado por elementos lógicos programáveis pelo usuário.

Núcleo rígido: parte de lógica programada pelo fabricante para uma função específica

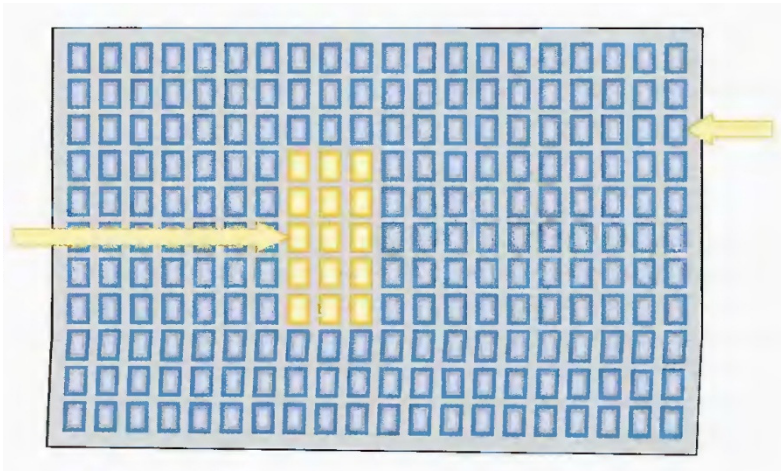

Figura 6 - Modelo de um núcleo rígido embutido em uma FPGA (Floyd, 2006, p.632)

Núcleos rígidos são geralmente desenvolvidos pelos fabricantes de FPGA. Projetos de propriedade do fabricante são denominados de IP (*Intellectual Property*). A empresa geralmente relaciona em seu *site* os tipos de IP que estão disponíveis.

Um núcleo flexível é uma parte de lógica que é programada pelo usuário e pode ser modificada. Projetos de núcleos flexíveis podem ser encontrados prontos ou podem ser desenvolvidos. As desvantagens na sua utilização são a maior utilização da capacidade da FPGA e o maior tempo de desenvolvimento para o usuário, pois mesmo que o projeto esteja pronto, pode ser necessária a realização de testes e modificações. A sua vantagem está na sua flexibilidade por poder ser modificado.

3.3 Desenvolvimento de projetos em FPGA

Ao longo dos últimos anos, o uso de FPGAs tem aumentado muito nos produtos comerciais e militares. Eles podem ser encontrados em radares, na comunicação via satélite, produção, indústria automotiva, e muitos outros tipos de produtos. Independentemente do produto final, projetistas de FPGA seguem um mesmo processo básico. Os estágios de desenvolvimento em FPGA são *design*, simulação, síntese e implementação, como mostrado na Figura 7.

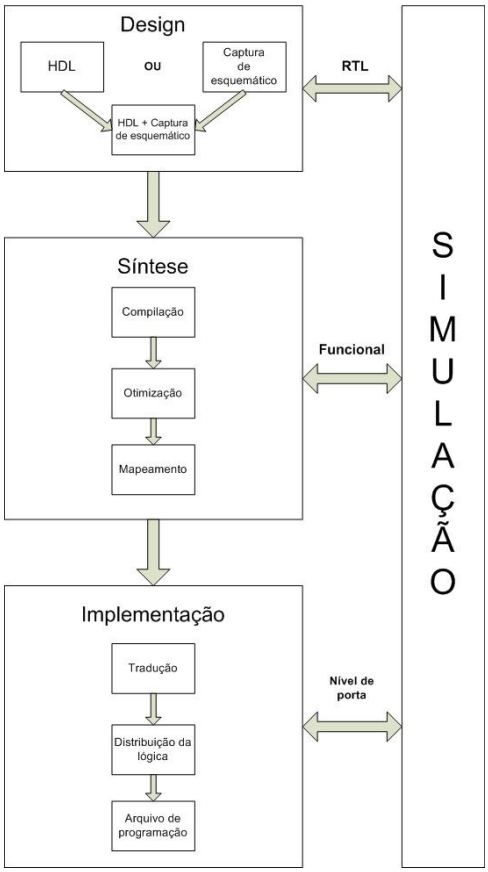

Figura 7 - Fluxo de desenvolvimento de projeto em FPGA

3.3.1 Design

O processo de *design* envolve a conversão dos requisitos para um formato que representa a função digital desejada. Formatos de concepção mais comuns são a captura de esquemático, a linguagem de descrição de hardware (HDL), ou uma combinação dos dois. Embora cada método tenha suas vantagens e desvantagens, HDLs geralmente oferecem grande flexibilidade de *design*.

3.3.1.1 Captura de esquemático

A captura de esquemático, uma representação gráfica de um projeto digital, mostra a interligação efetiva entre as portas lógicas que produzem as funções de saída desejadas. Muitas destas funções envolvem informações proprietárias, e são fornecidas ao projetista apenas através da biblioteca do fornecedor específico da FPGA. A natureza exclusiva deste tipo de projeto faz com que seja dependente do fornecedor, e o processo de *design* inteiro deve ser repetido se um fornecedor diferente é usado.

Exemplos de ferramentas de captura de esquemáticos são Viewlogic's ViewDraw e HDL's EASE. A principal vantagem da utilização da captura de esquemático é que a representação gráfica é fácil de entender. No entanto, sua maior desvantagem é um aumento no custo e tempo para reproduzir um projeto para diferentes fornecedores, devido à natureza proprietária do desenho.

3.3.1.2 Linguagem de descrição de hardware

Linguagens de descrição de *hardware* (HDLs) utilizam código para representar funções digitais. A utilização de HDLs é uma abordagem mais comum e popular para projetos em FPGA. Pode-se criar o código-fonte com qualquer editor de texto, ou através de editores especiais como o CodeWright e o Scriptum, que oferecem recursos como modelos e destaque das palavras reservadas não encontradas em editores de texto comuns. As HDLs podem ser genéricas, como Verilog ou VHDL, podendo ser utilizadas por várias ferramentas de desenvolvimento de projeto, ou específicas, como a AHDL (Altera Hardware Description Language), que só é reconhecida pela ferramenta de desenvolvimento de projeto da Altera.

3.3.2 Simulação

A simulação é o ato de verificar o *design* antes da validação do hardware real. As características dos sinais de entrada do circuito são descritos em HDL ou graficamente, e então são aplicados ao design. Isso permite que o responsável pela verificação do código possa observar o comportamento das saídas.

Na simulação, os sinais de entrada do circuito, também chamados de estímulos, imitam sinais de entrada de um circuito real. Os estímulos forçam o circuito a operar sob várias circunstâncias e estados. O maior benefício dos estímulos é a habilidade de aplicar uma vasta gama de sinais com características válidas e não válidas. Assim, pode-se variar parâmetros de sinais de entrada e testar os limites do circuito, observando o comportamento das saídas sem causar dano ao hardware.

Existem três níveis de simulação: Nível de transferência de registrador (RTL - *Register Transfer Level*), funcional e nível de porta. Cada um ocorre em um lugar específico no processo de desenvolvimento. O nível de simulação RTL segue a fase de *design*, o nível funcional segue a síntese e após a conclusão da implantação, é feita a simulação em nível de porta. Geralmente, o estímulo desenvolvido para a simulação RTL é reutilizável sem modificações para cada nível de simulação.

A simulação inicial realizada imediatamente após a fase de *design* é a simulação RTL. Ela envolve diretamente a aplicação dos estímulos ao *design*. A simulação RTL apenas permite que os projetistas verifiquem se a lógica está correta.

A aplicação de estímulo de teste ao design sintetizado é uma simulação funcional. Nesta simulação é verificado se o processo de síntese não alterou o *design*.

Na simulação em nível de porta, todos os atrasos de tempo internos são incluídos, tornando as saídas da simulação mais precisas.

3.3.3 Síntese

O primeiro passo no processo de síntese é compilar o *design* em elementos estruturais. A etapa seguinte envolve a otimização do design, tornando-o menor e mais rápido, removendo lógicas desnecessárias e permitindo que os sinais cheguem às entradas ou saídas mais rapidamente.

O passo final no processo de síntese envolve o mapeamento ou associação do design à arquitetura específica de um fornecedor. O processo de mapeamento define as conexões do design na FPGA, usando a arquitetura do fabricante específico.

3.3.4 Implementação

A última fase no processo de desenvolvimento de projeto em FPGA é a implementação, também conhecida como PAR (*Place And Route*). O primeiro passo no processo de implementação é chamado tradução. A tradução envolve a verificação da coerência entre a síntese do *design* e a FPGA alvo. Alguns exemplos de incoerência são: dois sinais diferentes serem atribuídos ao mesmo pino, atribuição de um sinal ao pino Vcc ou terra ou atribuir um sinal a um pino não existente. O segundo passo é a distribuição da lógica do *design* na FPGA. O último passo é gerar o arquivo de programação, que pode ser gravado em uma memória flash, PROM ou diretamente na FPGA, dependendo do fabricante.

Capítulo 4 - Implementação de núcleos de código aberto de microcontroladores PIC16 em FPGA

Todo o desenvolvimento das atividades foram realizadas nas ferramenta computacionais Quartus II Web Edition v8.0 SP1 [14], da Altera, ModelSim-Altera 6.1g [15], da Mentor Graphics [16], e MPLAB, da Microchip. O uso da ferramenta computacional Quartus II 8.0 se deve ao fato deste trabalho ser baseado no uso de dispositivo reconfigurável da Altera, a utilização do simulador ModelSim 6.1 se deve ao fato deste ter uma versão licenciada e disponibilizada pela Altera, e a utilização do MPLAB se deve ao fato de que o núcleos estudados utilizarem instruções de microcontroladores da Microchip.

Para realizar a análise dos núcleos, foram utilizadas as linguagens de descrição de hardware VHDL e Verilog, conforme a necessidade.

Após a definição dos núcleos que seriam analisados, foram realizados testes para verificar a correta execução de todas as suas instruções e funções (Timer, Sleep, Reset, Interrupções, etc...) através de simulações no software ModelSim-Altera 6.1g e da implementação dos núcleos no kit de desenvolvimento DK-CYCII-2C20N [17] que possui, além da FPGA Cyclone II EP2C20F484C7N [18], chaves, LEDs, push-buttons e displays de 7 segmentos.

Os núcleos foram então analisados, e um deles foi escolhido para substituir um PIC16F84 [19] em um modelo de esteira industrial.

4.1 Pesquisa dos núcleos de código aberto

A pesquisa dos núcleos de código aberto foi iniciada no *site* OpenCores [21], onde os usuários publicam núcleos flexíveis de código aberto. Na seção de processadores podem ser encontrados núcleos de código aberto de microcontroladores das famílias 8051 (Intel), PIC16 (Microchip), AVR (Atmel), Z80 (Zilog), MC68HC (Motorola), e alguns núcleos que foram desenvolvidos como estudo.

Os núcleos de código aberto de microcontroladores da família PIC16 foram escolhidos como foco deste trabalho por terem um conjunto de instruções reduzido e também porque o desenvolvedor deste projeto possuía conhecimento prévio desta família de microcontroladores.

No *site* OpenCores foram encontrados os núcleos Mini-RISC, PPX16, RISC5x, risc16f84 e ClaiRISC. Em uma busca mais aprofundada na Internet, foram encontrados os núcleos CQPIC, RISC8, SLC1657 e UPEM.

4.2 Metodologia de implementação e verificação dos núcleos

A metodologia de implementação e verificação dos núcleos é composta dos seguintes processos: Compilação do núcleo, geração do programa de verificação, associação do programa de verificação à memória ROM, implementação, verificação, e simulação.

4.2.1 Compilação do núcleo

O processo de compilação do núcleo, realizado no Quartus II, é necessário porque os arquivos gerados neste processo são utilizados no processo de implementação do núcleo.

Apenas um núcleo foi compilado com sucesso na primeira tentativa, porque a maior parte dos autores não fornece o código das memórias RAM e ROM. Em alguns casos a memória RAM era fornecida, descrita através de matriz. Nos caos em que os códigos das memórias não foram fornecidos, eles foram gerados através da função Mega Wizard do Quartus II

Em alguns núcleos, as memórias eram acessadas como se fossem periféricos. Nestes casos foi necessário escrever um código para fazer a interligação entre as memórias e o núcleo. Este código é chamado de **arquivo TOP**.

4.2.2 Geração do programa de verificação

Após conseguir compilar o núcleo, um programa era escrito em assembly, no MPLAB, para verificar se uma instrução ou função estava funcionando corretamente. No MPLAB deve ser selecionado como microcontrolador alvo o microcontrolador no qual o núcleo foi baseado. Ao compilar o programa de verificação no MPLAB, é gerado um arquivo .hex, no formato Intel HEX 32.

4.2.3 Associação do código de verificação à memória ROM

No Mega Wizard, no processo de definição dos atributos da memória ROM a ser gerada, é possível associar à memória um arquivo contendo os valores que estarão na memória quando esta for implementada na FPGA. Este arquivo deve ser de extensão **.mif**, é chamado Memory Initialization File, e possui uma formatação específica.

Para que o núcleo rode o programa de verificação, é preciso converter o arquivo **.hex**, gerado na compilação do programa de verificação, em um arquivo **.mif**, e então associar este arquivo **.mif** à memória ROM utilizada pelo núcleo. Para os núcleos de PIC16F84, a conversão foi feita através de um programa escrito pelo autor do CQPIC. Para os núcleos de PIC16C5X e PIC16F5X, a conversão foi feita através de um programa escrito pelo autor do RISC8. Ambos os programas foram escritos em linguagem C. No Mega Wizard é possível editar os atributos da memória ROM gerada no processo de

compilação. Portanto não é necessário gerar nova memória ROM para fazer a associação com o arquivo **.mif**.

4.2.4 Implementação

No processo de implementação, o núcleo, juntamente com suas memórias, é gravado na FPGA através do Quartus II. Após a implementação, o núcleo passa a funcionar na FPGA, rodando o programa de verificação.

4.2.5 Verificação

A verificação do correto funcionamento da instrução ou função é feita através dos LEDs, displays de sete segmentos, chaves e push-buttons do kit de desenvolvimento. Por exemplo, para testar a instrução SUBWF pode-se escrever um programa que coloque um valor pré-definido em F, pegue um valor definido por oito chaves (cada chave um bit) e passe para W, e então execute a instrução SUBWF, mostrando o resultado da operação em oito LEDs (cada LED um bit).

Se a verificação foi feita com sucesso, um novo programa de verificação é feito, e os processos são repetidos para todas as instruções e funções dos núcleos.

4.2.6 Simulação

Quando alguma instrução ou função não é executada corretamente, o núcleo, juntamente com suas memórias, é simulado no ModelSim. Na simulação é possível ver o comportamento dos sinais internos do núcleo para descobrir o motivo do não funcionamento da instrução ou função.

Após descobrir o motivo do não funcionamento da instrução ou função, as alterações necessárias são realizadas para que a instrução ou função funcione corretamente, e então o núcleo, juntamente com suas memórias, é novamente implementado para realizar novamente a verificação.

4.3 CQPIC

O núcleo CQPIC foi desenvolvido por Sumio Morioka e é uma implementação em VHDL de um microcontrolador PIC16F84. O núcleo foi publicado em dezembro de 1999 na revista Transistor Gijutsu, uma das revistas mais famosas do Japão relacionadas a hardware de computadores.

O autor mantém um *site* [22] onde a última versão dos arquivos (v1.00d) pode ser encontrada. A última atualização foi feita em 19/07/2004.

4.3.1 Processo de verificação do núcleo

O autor fornceu as memórias RAM e ROM, do tipo LPM [23], que não são suportadas pela FPGA utilizada. Por este motivo, as memórias foram substituídas por outras geradas através da função Mega Wizard do Quartus II.

Durante o processo de verificação das instruções do núcleo foi constatado erro no funcionamento do flag de estouro de WDT. Através de simulações, observou-se que o **bit4** do **status_reg** não passava para nível baixo quando havia estouro de WDT. O erro foi corrigido passando o valor da *variable* **wdtreset_node** para um *signal* (incluído no código), **WDT_reset_flag**, e então utilizando o *signal* como flag de estouro de WDT no lugar da *variable*.

4.3.2 Resultados

O núcleo CQPIC possui Watchdog Timer (WDT), Timer (TMR0), Sleep, interrupções externas (pinos RB0 e RB4 à RB7), memória RAM de 512 bytes e memória ROM de 8192 bytes. Possui dois PORTs com entradas e saídas independentes (PORTA e PORTB), sendo a primeira de 6 bits (RA0 a RA5), e a segunda de 8 bits (RB0 a RB7). O núcleo original possui a lógica para leitura e escrita de EEPROM, porém esta foi retirada pois não foi utilizada.

Através da compilação verificou-se que o núcleo CQPIC consumiu 768 elementos lógicos (4% dos elementos lógicos), 118.784 bits de memória (50% dos bits de memória), utilizou 48 pinos (15% dos pinos), e a freqüência máxima de operação é 99.59 MHz.

A Figura 8 apresenta a hierarquia dos códigos que compõem o núcleo CQPIC.

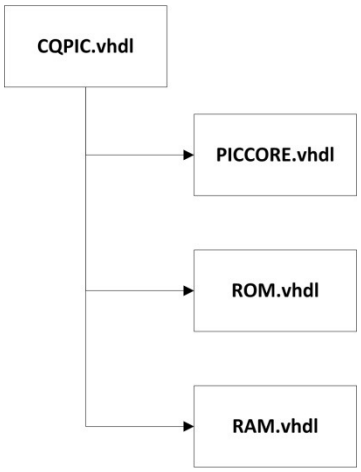

Figura 8 - Hierarquia dos códigos do núcleo CQPIC

A Tabela 1 apresenta a comparação entre as características de um PIC16F84 e as do núcleo CQPIC.

Tabela 1 - Comparação entre as características de um PIC16F84 e as do núcleo CQPIC

Característica	PIC16F84	CQPIC
Freqüência de Operação	20 MHz	99.59 MHz
PORTs	PORTA, 6 bits PORTB, 8 bits	PORTA, 6 bits PORTB, 8 bits
Watchdog Timer	Sim	Sim
Sleep	Sim	Sim
Pilha	8 níveis	8 níveis
Timer	Sim	Sim
Interrupções	RB0/INT TMR0 PORTB<7:4> EEPROM	RB0/INT TMR0 PORTB<7:4>
EEPROM	64 bytes	Não possui
Memória de dados (RAM)	68 bytes	512 bytes
Memória de programa (ROM)	1024 bytes	8192 bytes

4.4 Mini-RISC

O núcleo Mini-RISC foi desenvolvido por Rudolf Usselmann e é uma implementação em Verilog de um microcontrolador PIC16C57. O núcleo foi publicado no *site* OpenCores em 25 de Setembro de 2001 e sua última atualização foi feita em 01/10/2002 [24].

4.4.1 Processo de verificação do núcleo

As memórias RAM e ROM não foram fornecidas, por isto foram geradas através da função Mega Wizard do Quartus II.

Durante o processo de testes, foi constatado que a instrução CLRWDT zera o WDT, porém não altera o estado de TO ou PD. Quando há estouro de WDT, nada acontece porque o autor não desenvolveu uma estrutura para alterar TO e PD. Por este motivo, o WDT foi considerado como não existente.

A instrução Sleep não funciona. O núcleo apenas reconhece a instruções, mas não há uma estrutura para realizar o Sleep. Portanto o Sleep foi considerado como não existente.

O bit C não estava sendo colocado em 1 quando a operação retornava um resultado positivo ou zero, e não estava sendo zerado quando a operação retornava um resultado negativo, porque o carry e não o borrow estava sendo passado para o bit C. As alterações necessárias para corrigir o problema foram realizadas.

Para fazer a contagem, o Timer utiliza seu estado anterior. Como nenhum valor era definido na inicialização, o Timer permanecia em estado indefinido e por isto não havia contagem. O funcionamento do Timer foi corrigido inicializando-o com o valor 0.

4.4.2 Resultados

O núcleo Mini-RISC possui três PORTs de 8 bits cada (PORTA, PORTB, PORTC), Timer (TMR0), memória RAM de 128 bytes e ROM de 512 bytes. Instruções que alteram o registrador PC utilizam 4 ciclos de máquina ao invés de 2 como no PIC16C57. Por não possuírem a lógica completa para o funcionamento, o Watchdog Timer (WDT) e o Sleep foram considerados como não existentes.

Através da compilação verificou-se que o núcleo Mini-RISC consumiu 507 elementos lógicos (3% dos elementos lógicos), 25.600 bits de memória (11% dos bits de memória), utilizou 28 pinos (9% dos pinos), e a freqüência máxima de operação é 72.71MHz.

A Figura 9 apresenta a hierarquia dos códigos que compõem o núcleo Mini-RISC.

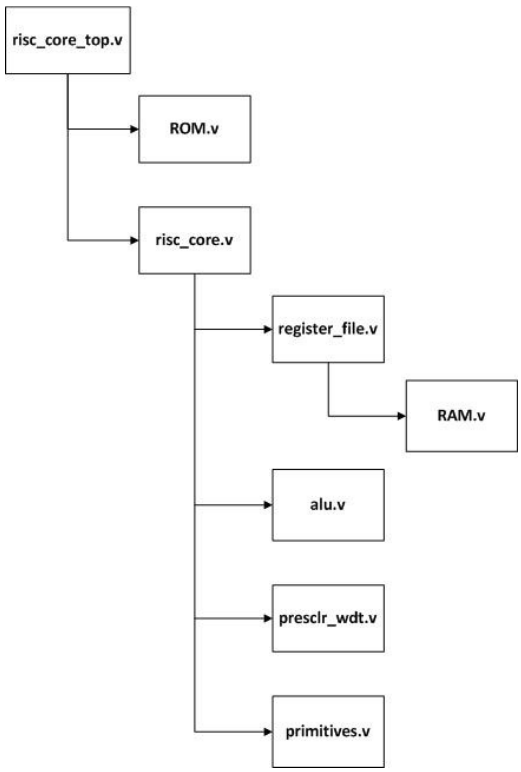

Figura 9 - Hierarquia dos códigos do núcleo Mini-RISC

A Tabela 2 apresenta a comparação entre as características de um PIC16C57 e as do núcleo Mini-RISC.

Tabela 2 - Comparação entre as características de um PIC16C57 e as do núcleo Mini-RISC

Característica	PIC16C57	Mini-RISC
Freqüência de Operação	40 MHz	72.71 MHz
PORTs	PORTA, 4 bits PORTB, 8 bits PORTC, 8 bits	PORTA, 8 bits PORTB, 8 bits PORTC, 8 bits
Watchdog Timer	Sim	Não
Sleep	Sim	Não
Pilha	2 níveis	4 níveis
Timer	Sim	Sim
Memória de dados (RAM)	72 bytes	128 bytes
Memória de programa (ROM)	2048 bytes	512 bytes

4.5 PPX16

O núcleo PPX16 foi desenvolvido por Daniel Wallner e é uma implementação em VHDL que possui duas versões, uma para o PIC16F84 (P16F84) e outra para o PIC16C55 (P16C55). O núcleo foi publicado no *site* OpenCores em 14 de Maio de 2002 [25].

4.5.1 Processo de verificação do núcleo

A memória ROM não foi fornecida pelo autor, então foi gerada através da função Mega Wizard do Quartus II.

A memória RAM utilizada foi a fornecida pelo autor do núcleo e foi implementada através de *array* [26].

Os registradores INTCON e OPTION estavam sendo alterados sempre que seus endereços estavam presentes no duto de endereço da memória RAM. Para controlar o momento certo para alterar o valor dos registradores, o flag de controle de escrita da memória RAM foi utilizado na lógica de controle dos registradores.

A instrução SUBLW subtrai W (subtraendo) de k (minuendo) através do método **complemento de dois**. No entanto, o OpenCore realizava a instrução utilizando k como subtraendo e W como minuendo. As alterações necessárias foram realizadas para inverter a lógica de execução

Todos os problemas descritos anteriormente foram encontrados tanto na versão P16F84 como na a versão P16C55.

4.5.2 Resultados

O núcleo PPX16-P16F84 possui dois PORTS de 8 bits cada (PORTA e PORTB) e Timer (TMR0). O Watchdog Timer (WDT) e o Sleep não foram implementados. Interrupções externas foram implementadas, com a diferença de que a interrupção do pino RB0 ser feita pela entrada INT, e as outras interrupções são feitas através dos pinos RB4 a RB7, como no PIC16F84. O núcleo utiliza memória RAM de 128 bytes e ROM de 1024 bytes.

O núcleo PPX16-P16C55 é similar ao PPX-P16F84, porém, com um PORT a mais de 8 bits (PORTC), memória RAM de 32 bytes, memória ROM de 512 bytes, e não possui interrupções.

Através da compilação verificou-se que o núcleo PPX-P16F84 consumiu 661 elementos lógicos (4% dos elementos lógicos), consumiu 14.976 bits de memória (6% dos bits de memória), utilizou 20 pinos (6% dos pinos), e a freqüência máxima de operação é 56.09 MHz. O núcleo PPX16-P16C55 consumiu 442 elementos lógicos (2% dos elementos lógicos), consumiu 6.400 bits de memória (3% dos bits de memória), utilizou 27 pinos (9% dos pinos), e a freqüência máxima de operação é 58.15 MHz.

A Figura 10 apresenta a hierarquia dos códigos que compõem o núcleo PPX16.

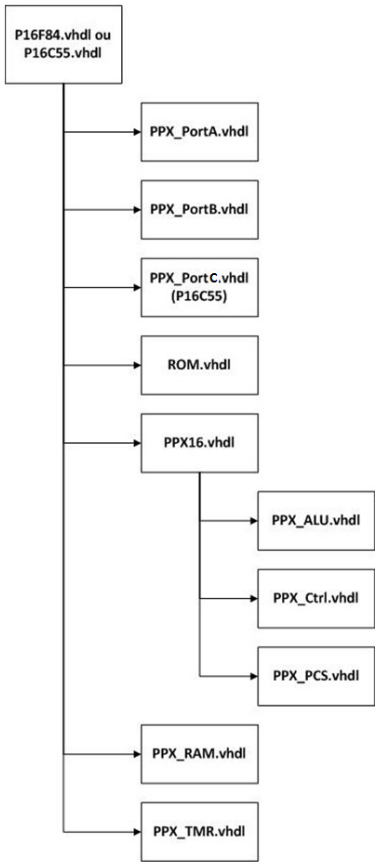

Figura 10 - Hierarquia dos códigos do núcleo PPX16

A Tabela 3 apresenta a comparação entre as características de um PIC16F84 e as do núcleo PPX16-P16F84.

Tabela 3 - Comparação entre as características de um PIC16F84 e as do núcleo PPX16-P16F84

Característica	PIC16F84	PPX16-P16F84
Freqüência de Operação	20 MHz	56.09 MHz
PORTs	PORTA, 5 bits PORTB, 8 bits	PORTA, 8 bits PORTB, 8 bits
Watchdog Timer (WDT)	Sim	Não possui
Sleep	Sim	Não possui
Pilha	8 níveis	8 níveis
Timer	Sim	Sim
Interrupções	RB0/INT TMR0 PORTB<7:4> EEPROM	INT TMR0 PORTB<7:4>
EEPROM	64 bytes	Não possui
Memória de dados (RAM)	68 bytes	128 bytes
Memória de programa (ROM)	1024 bytes	1024 bytes

A Tabela 4 apresenta a comparação entre as características de um PIC16F84 e as do núcleo PPX16-P16C55.

Tabela 4 - Comparação entre as características de um PIC16C55 e as do núcleo PPX16-P16C55

Característica	PIC16C55	PPX16-P16C55
Freqüência de Operação	40 MHz	58.15 MHz
PORTs	PORTA, 4 bits PORTB, 8 bits PORTC, 8 bits	PORTA, 8 bits PORTB, 8 bits PORTC, 8 bits
Watchdog Timer (WDT)	Sim	Não
Sleep	Sim	Não
Pilha	2 níveis	2 níveis
Timer	Sim	Sim
Memória de dados (RAM)	24 bytes	128 bytes
Memória de programa (ROM)	512 bytes	512 bytes

4.6 RISC5x

O núcleo RISC5x foi desenvolvido por Mike Johnson e é uma implementação em VHDL, de um microcontrolador PIC16C57. O núcleo foi publicado no *site* OpenCores em 17 de Janeiro de 2002 [27].

4.6.1 Processo de verificação do núcleo

A memória ROM não foi fornecida pelo autor, e por isto foi gerada através da função Mega Wizard do Quartus II.

A memória RAM utilizada foi a fornecida pelo autor do núcleo e foi implementada através de *array*.

Todas as funções e instruções foram testadas e executaram corretamente.

4.6.2 Resultados

O núcleo RISC5x possui três PORTS de 8 bits cada (PORTA, PORTB, PORTC). O Watchdog Timer (WDT), o Timer (TMR0) e o Sleep não foram implementados. O núcleo utiliza memória RAM de 128 bytes e ROM de 2048 bytes.

Através da compilação verificou-se que o núcleo RISC5x consumiu 1.759 elementos lógicos (9% dos elementos lógicos), consumiu 24.576 bits de memória (10% dos bits de memória), utilizou 26 pinos (8% dos pinos), e a freqüência máxima de operação é 46.87 MHz.

A Figura 11 apresenta a hierarquia dos códigos que compõem o núcleo RISC5x.

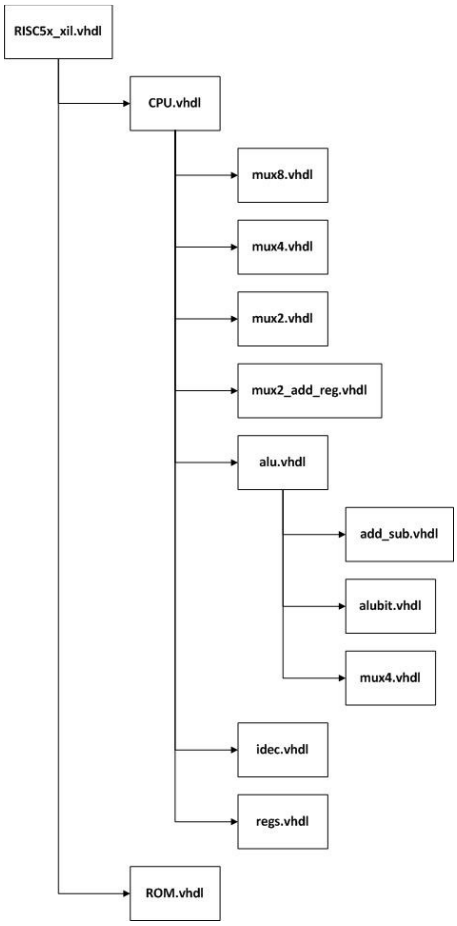

Figura 11 - Hierarquia dos códigos do núcleo RISC5x

A Tabela 5 apresenta a comparação entre as características de um PIC16C57 e as do núcleo RISC5x.

Tabela 5 - Comparação entre as características de um PIC16C57 e as do núcleo RISC5x

Característica	PIC16C57	RISC5x
Freqüência de Operação	40 MHz	46.87 MHz
PORTs	PORTA, 4 bits PORTB, 8 bits PORTC, 8 bits	PORTA, 8 bits PORTB, 8 bits PORTC, 8 bits
Watchdog Timer (WDT)	Sim	Não
Sleep	Sim	Não
Pilha	2 níveis	2 níveis
Timer	Sim	Não
Memória de dados (RAM)	72 bytes	128 bytes
Memória de programa (ROM)	2048 bytes	2048 bytes

4.7 RISC8

O núcleo RISC8 foi desenvolvido por Tom Coonan e é uma implementação em Verilog, de um microcontrolador PIC16C57. O núcleo foi publicado na página pessoal do autor [28] e sua última atualização feita em 26 de Outubro de 2002.

4.7.1 Processo de verificação do núcleo

As memórias RAM e ROM não foram fornecidas pelo autor.

A memória ROM foi gerada através da função Mega Wizard do Quartus II.

A memória RAM gerada através da função Mega Wizard do Quartus II não funciono. Então a memória foi implementada utilizando-se uma matriz de vetores [29].

Todas as funções e instruções foram testadas e executaram corretamente.

4.7.2 Resultados

O núcleo RISC8 possui três PORTS de 8 bits cada (PORTA, PORTB e PORTC) e Timer (TMR0). O Watchdog Timer (WDT) e o Sleep não foram implementados. O núcleo utiliza memória RAM de 128 bytes e ROM de 2048 bytes.

Através da compilação verificou-se que o núcleo RISC8 consumiu 1.777 elementos lógicos (9% dos elementos lógicos), consumiu 24.576 bits de memória (10% dos bits de memória), utilizou 26 pinos (8% dos pinos), e a freqüência máxima de operação é 40.13 MHz.

A Figura 12 apresenta a hierarquia dos códigos que compõem o núcleo RISC8.

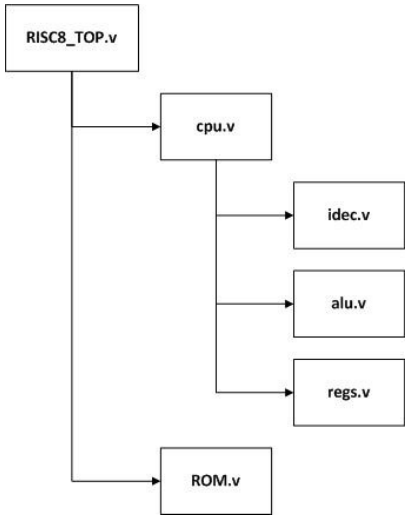

Figura 12 - Hierarquia dos códigos do núcleo RISC8

A Tabela 6 apresenta a comparação entre as características de um PIC16C57 e as do núcleo RISC8.

Tabela 6 - Comparação entre as características de um PIC16C57 e as do núcleo RISC8

Característica	PIC16C57	RISC8
Freqüência de Operação	40 MHz	40.13 MHz
PORTs	PORTA, 4 bits PORTB, 8 bits PORTC, 8 bits	PORTA, 8 bits PORTB, 8 bits PORTC, 8 bits
Watchdog Timer (WDT)	Sim	Não
Sleep	Sim	Não
Pilha	2 níveis	2 níveis
Timer	Sim	Sim
Memória de dados (RAM)	72 bytes	128 bytes
Memória de programa (ROM)	2048 bytes	2048 bytes

4.8 Risc16f84

O núcleo Risc16f84 é uma implementação em Verilog, de um microcontrolador PIC16F84 e possui 4 versões: risc16f84, risc16f84_lite, risc16f84_small, risc16f84_clk2x.

A versão risc16f84 é a mais fiel ao PIC16F84. Na versão risc16f84_lite a interface com EEPROM foi retirada. Na versão risc16f84_small além da interface com EEPROM, as interrupções foram retiradas. A versão risc16f84_clk2x não possui interface com EEPROM, interrupções ou PORTS. Um duto de dados auxiliar foi implementado para acesso de 64Kbytes de registradores, PORTS e periféricos. A versão escolhida para o estudo foi a risc16f84_lite, pois apenas a interface com EEPROM foi retirada, e esta interface não seria utilizada.

O núcleo foi desenvolvido por John Clayton e é uma tradução para Verilog do núcleo CQPIC. Ele foi publicado no *site* OpenCores em 7 de Maio de 2002 [30].

4.8.1 Processo de verificação do núcleo

As memórias RAM e ROM não foram fornecidas, por isto foram geradas através da função Mega Wizard do Quartus II.

A instrução MOVF não funcionava corretamente porque o endereço correto da memória RAM a ser lido estava aparecendo com um *clock* de atraso. As alterações necessárias foram feitas de modo que o endereço correto da memória RAM a ser lido fosse utilizado.

Muitas instruções não executavam corretamente devido a atrasos de sinais. Estes atrasos ocorriam porque o autor utilizava a atribuição "<=" ao invés da atribuição "=". As operações que utilizam a atribuição "<=" são executadas paralelamente e por este motivo ocorriam os atrasos. Substituindo a atribuição "<=" pela atribuição "=", as operações são realizadas seqüencialmente e as instruções são executadas corretamente (vide APÊNDICE A – Atribuição em Verilog).

A interrupção da entrada RB6 e a instrução CLRWDT não eram executadas corretamente devido a erro de digitação do autor. Os erros de digitação foram corrigidos.

O núcleo não saía do Sleep por interrupção quando o bit GIE não estava em 1. As alterações necessárias foram feitas para que as interrupções funcionassem durante um Sleep, mesmo com o GIE zerado.

4.8.2 Resultados

O núcleo risc16f84_lite possui dois PORTs, sendo um de 5 bits (PORTA) e um de 8 bits (PORTB), possui Timer (TMR0), Watchdog Timer (WDT) e Sleep. As interrupções, que no PIC16F84 são feitas através dos pinos RB0 e RB4 à RB7, neste núcleo são feitas através de pinos dedicados. O núcleo utiliza memória RAM de 512 bytes e ROM de 8192 bytes.

Através da compilação verificou-se que o núcleo risc16f84_lite consumiu 702 elementos lógicos (4% dos elementos lógicos), consumiu 118.784 bits de memória (50% dos bits de memória), utilizou 27 pinos (9% dos pinos), e a freqüência máxima de operação é 93.8 MHz.

A Figura 13 apresenta a hierarquia dos códigos que compõem o núcleo risc16f84_lite.

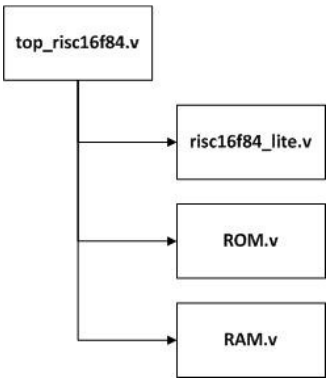

Figura 13 - Hierarquia dos códigos do núcleo risc16f84_lite

A Tabela 7 apresenta a comparação entre as características de um PIC16C57 e as do núcleo risc16f84_lite.

Tabela 7 - Comparação entre as características de um PIC16F84 e as do núcleo risc16f84_lite

Característica	PIC16F84	risc16f84_lite
Freqüência de Operação	20 MHz	93.8 MHz
PORTs	PORTA, 5 bits PORTB, 8 bits	PORTA, 5 bits PORTB, 8 bits
Watchdog Timer (WDT)	Sim	Sim
Sleep	Sim	Sim
Pilha	8 níveis	8 níveis
Timer	Sim	Sim
Interrupções	RB0/INT TMR0 PORTB<7:4> EEPROM	INT0, INT4, INT5, INT6, INT7 TMR0
EEPROM	64 bytes	Não possui
Memória de dados (RAM)	68 bytes	512 bytes
Memória de programa (ROM)	1024 bytes	8192 bytes

4.9 SLC1657

O núcleo SLC1657 [31] é uma implementação em VHDL de um microcontrolador PIC16C57. O núcleo foi desenvolvido por uma empresa chamada Silicore e o responsável pelo projeto foi Wade D. Peterson. O desenvolvimento do núcleo foi terminado em 14 de Agosto de 2001 e em 03 de Setembro de 2003 a Silicore lançou o SLC1657 sob licença do tipo LGPL [32].

4.9.1 Processo de verificação do núcleo

A memória ROM não foi fornecida pelo autor, então foi gerada através da função Mega Wizard do Quartus II.

A memória RAM foi fornecida pelo autor e foi implementada através de *array*.

A memória ROM, o Timer e o WDT utilizam sinais que não eram inicializados, e por isto iniciavam com o valor ‘U’ (*Uninitialized*) [33]. Desta forma, a memória ROM, o Timer e o WDT não funcionavam. Após definir valores iniciais para os sinais que iniciavam com o valor ‘U’, a memória ROM o Timer e o WDT passaram a funcionar corretamente.

A instrução RETLW não funcionava corretamente porque o endereço armazenado na Pilha estava errado. As alterações necessárias para que a Pilha armazenasse o endereço correto foram feitas.

Quando a instrução DECFSZ era executada e o resultado do decremento era 0, o núcleo realizava dois decrementos, porque ao invés de substituir a instrução seguinte por uma instrução NOP, estava substituindo por DECFSZ. As alterações necessárias foram realizadas e a instrução DECFSZ passou a ser executada corretamente.

O Prescaler estava utilizando como clock base o sinal 'MCLK_4', sendo que deveria utilizar o sinal 'MCLK'. As alterações necessárias foram realizadas para que o Prescaler utilize o sinal 'MCLK' como clock.

O WDT deveria utilizar como clock a saída do Prescaler, no entanto, estava utilizando como clock o sinal 'MCLK_16'. As alterações necessárias foram feitas para utilizar a saída do Prescaler como clock do WDT.

4.9.2 Resultados

O núcleo SLC1657 possui três PORTs de 8bits cada (PORT0, PORT1, PORT2), Timer (TMR0), Watchdog Timer (WDT) e Sleep. O núcleo utiliza memória RAM de 128 bytes e ROM de 2048 bytes.

Através da compilação verificou-se que o núcleo SLC1657 consumiu 1.288 elementos lógicos (7% dos elementos lógicos), consumiu 24.576 bits de memória (10% dos bits de memória), utilizou 31 pinos (10% dos pinos), e a freqüência máxima de operação é 25.79 MHz.

A Figura 14 apresenta a hierarquia dos códigos que compõem o núcleo SLC1657.

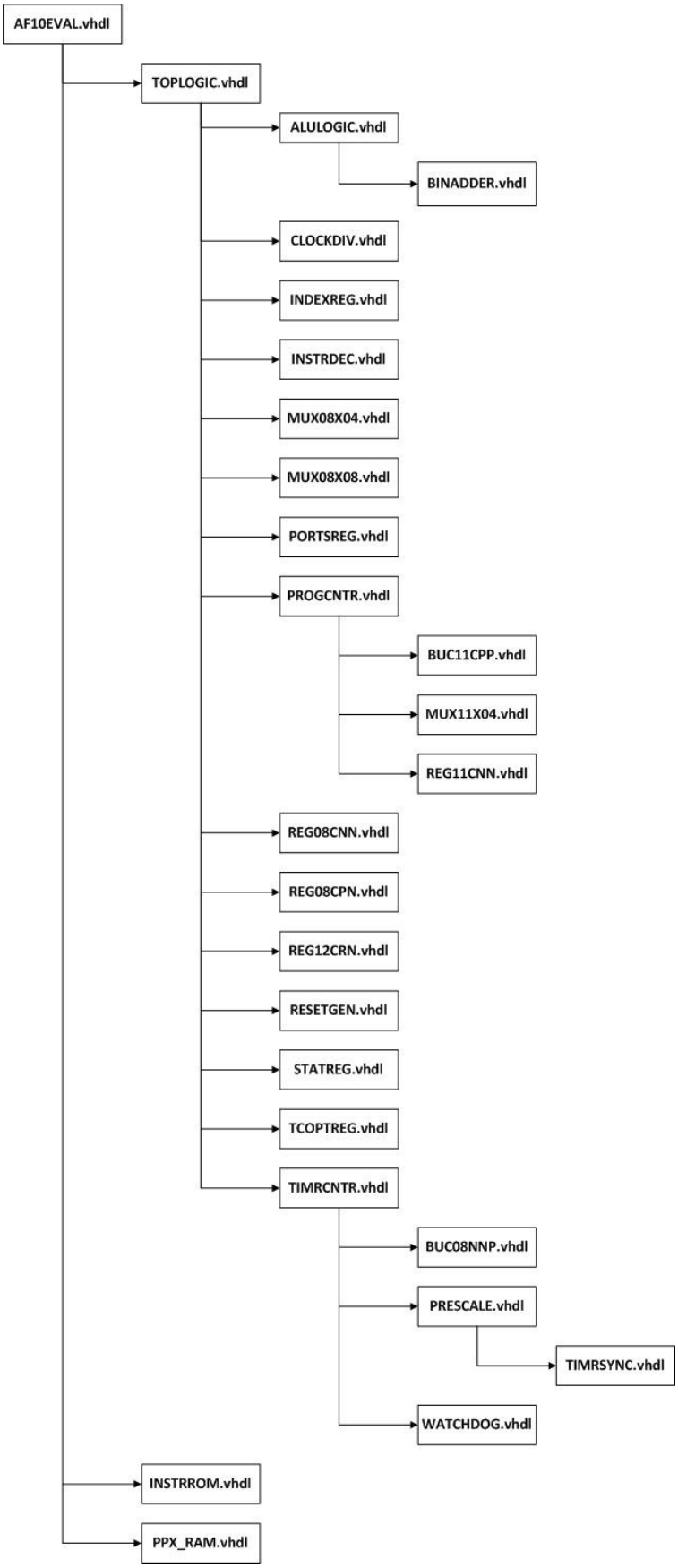

Figura 14 - Hierarquia dos códigos do núcleo SLC1657

A Tabela 8 apresenta a comparação entre as características de um PIC16C57 e as do núcleo SLC1657.

Tabela 8 - Comparação entre as características de um PIC16C57 e as do núcleo SLC1657

Característica	PIC16C57	SLC1657
Freqüência de Operação	40 MHz	25.79 MHz
PORTs	PORTA, 4 bits PORTB, 8 bits PORTC, 8 bits	PORTA, 8 bits PORTB, 8 bits PORTC, 8 bits
Watchdog Timer (WDT)	Sim	Sim
Sleep	Sim	Sim
Pilha	2 níveis	2 níveis
Timer	Sim	Sim
Memória de dados (RAM)	72 bytes	128 bytes
Memória de programa (ROM)	2048 bytes	2048 bytes

4.10 ClaiRISC

O núcleo ClaiRISC foi desenvolvido por Li Wei e é uma implementação em Verilog de um microcontrolador PIC16F57. O núcleo foi publicado no *site* OpenCores em 08 de Fevereiro de 2008 [34].

4.10.1 Processo de verificação do núcleo

As memórias RAM e ROM não foram fornecidas pelo autor, portanto foram geradas pela função Mega Wizard do Quartus II.

Ao voltar de um Reset, a instrução da posição 1 da memória ROM não era executada devido à lógica de definição do valor do *Program Counter* (PC) após um Reset. As alterações necessárias foram realizadas e a instrução da posição 1 passou a ser executada após um Reset.

As instruções ANDWF, IORWF e BCF não executavam corretamente devido a erro de digitação. Os erros de digitação foram arrumados e as instruções passaram a executar corretamente.

As instruções CALL e GOTO não executavam corretamente porque a lógica da execução destas operações estava errada. A lógica foi corrigida e as instruções passaram a ser executadas corretamente.

A instrução RETLW não funcionava corretamente porque o endereço salvo na pilha estava errado. As alterações necessárias foram realizadas e a instrução passou a ser executada corretamente.

4.10.2 Resultados

O núcleo ClaiRISC possui dois PORTs de 8 bits cada (PORTB, PORTC). O Timer (TMR0), o Watchdog Timer (WDT) e o Sleep não foram implementados. O núcleo utiliza memória RAM de 128 bytes e ROM de 2048 bytes.

Através da compilação verificou-se que o núcleo ClaiRISC consumiu 381 elementos lógicos (2% dos elementos lógicos), consumiu 25.600 bits de memória (11% dos bits de memória), utilizou 34 pinos (8% dos pinos), e a freqüência máxima de operação é 64.81 MHz.

A Figura 15 apresenta a hierarquia dos códigos que compõem o núcleo ClaiRISC.

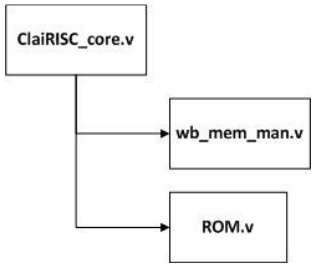

Figura 15 - Hierarquia dos códigos do núcleo ClaiRISC

A Tabela 9 apresenta a comparação entre as características de um PIC16F57 e as do núcleo ClaiRISC.

Tabela 9 - Comparação entre as características de um PIC16F57 e as do núcleo ClaiRISC

Característica	PIC16F57	ClaiRISC
Freqüência de Operação	20 MHz	64.81 MHz
PORTs	PORTA, 4 bits PORTB, 8 bits PORTC, 8 bits	PORTB, 8 bits PORTC, 8 bits
Watchdog Timer (WDT)	Sim	Não
Sleep	Sim	Não
Pilha	2 níveis	2 níveis
Timer	Sim	Não
Memória de dados (RAM)	72 bytes	128 bytes
Memória de programa (ROM)	2048 bytes	2048 bytes

4.11 UPEM

O núcleo UPEM é uma implementação em VHDL de um microcontrolador PIC16F84. Este núcleo foi retirado do livro "Microcontroladores e FPGAs, aplicações em automação" [35] e é uma simplificação do núcleo CQPIC.

4.11.1 Processo de verificação do núcleo

A memória ROM utilizada pelo autor era implementada através de um case, o que dificultaria a alteração dos dados da ROM, portanto, decidiu-se utilizar uma memória ROM gerada pela função Mega Wizard do Quartus II.

A memória RAM utilizada foi a fornecida pelo autor do núcleo e foi implementada através de *array*.

Todas as instruções e funções foram testadas e funcionaram corretamente.

4.11.2 Resultados

O núcleo UPEM possui um PORT de 8 bits (PORTB). As funções Timer (TMR0), Watchdog Timer (WDT), Sleep e interrupções não foram implementadas. O núcleo utiliza memória RAM de 64 bytes e ROM de 8192 bytes.

Através da compilação verificou-se que o núcleo UPEM consumiu 1.482 elementos lógicos (8% dos elementos lógicos), consumiu 114.688 bits de memória (48% dos bits de memória), utilizou 11 pinos (3% dos pinos), e a freqüência máxima de operação é 109.7 MHz.

A Figura 16 apresenta a hierarquia dos códigos que compõem o núcleo UPEM.

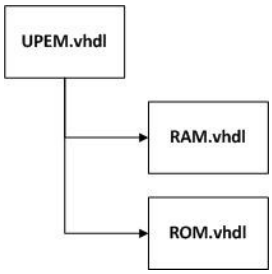

Figura 16 - Hierarquia dos códigos do núcleo UPEM

A Tabela 10 apresenta a comparação entre as características de um PIC16F84 e as do núcleo UPEM.

Tabela 10 - Comparação entre as características de um PIC16F84 e as do núcleo UPEM

Característica	PIC16F84	UPEM
Freqüência de Operação	20 MHz	109.7 MHz
PORTs	PORTA, 5 bits PORTB, 8 bits	PORTB, 8 bits
Watchdog Timer (WDT)	Sim	Não possui
Sleep	Sim	Não possui
Pilha	8 níveis	8 níveis
Timer	Sim	Não possui
Interrupções	RB0/INT TMR0 PORTB<7:4> EEPROM	Não possui
EEPROM	64 bytes	Não possui
Memória de dados (RAM)	68 bytes	64 bytes
Memória de programa (ROM)	1024 bytes	8192 bytes

4.12 Análise dos núcleos

Após a verificação e implementação dos núcleos, estes foram analisados com relação ao consumo de elementos lógicos, freqüência máxima de operação, suas funções e a estrutura do código, para selecionar um núcleo para substituir um PIC16F84 responsável pelo controle de uma esteira industrial. O consumo de bits de memória não foi considerado na análise dos núcleos por que este parâmetro pode ser variado sem alterar os outros parâmetros.

A Tabela 11 apresenta as características dos núcleos de PIC16F84 importantes na análise dos núcleos.

Tabela 11 - Características dos núcleos de PIC16F84

	Freqüência de Operação	WDT	Sleep	Timer	Interrupções	Consumo de elementos lógicos
UPEM	109,7 MHz	Não	Não	Não	Não	1.482
CQPIC	99,59 MHz	Sim	Sim	Sim	Sim	768
risc16f84	93,8MHz	Sim	Sim	Sim	Sim	702
PPX16-P16F84	56,09MHz	Não	Não	Sim	Sim	661

Dentre os núcleos de PIC16F84, o núcleo CQPIC aparenta ser o mais interessante.

O CQPIC possui freqüência de operação levemente inferior à do UPEM, porém, o UPEM consome o dobro de elementos lógicos e não possui nenhuma função que o CQPIC possui.

O risc16f84 é bastante similar ao CQPIC, o que é coerente, pois é uma tradução do CQPIC, descrito em VHDL, para Verilog. Neste caso o CQPIC é mais interessante apenas por ser o original.

O PPX16-P16F84, apesar de ter um consumo de elementos lógicos levemente inferior ao CQPIC, possui freqüência de operação inferior e não possui Sleep e WDT.

A Tabela 12 apresenta as características dos núcleos de PIC16C5X e PIC16F5X importantes na análise dos núcleos. Os dois tipos de núcleo foram comparados juntamente porque os microcontroladores nos quais são baseados só diferenciam na freqüência de operação, com o PIC16C5X rodando a 40MHz e o PIC16F5X rodando a 20MHz.

Tabela 12 - Características dos núcleos de PIC16C5X e PIC16F5X

	Freqüência de Operação	WDT	Sleep	Timer	Consumo de elementos lógicos
Mini-RISC	72,71MHz	Não	Não	Sim	507
ClaiRISC	64,81MHz	Não	Não	Não	381
PPX16-P16C55	58,15 MHz	Não	Não	Sim	442
RISC5x	46,87MHz	Não	Não	Não	1.759
RISC8	40,13MHz	Não	Não	Sim	1.777
SLC1657	25,79MHz	Sim	Sim	Sim	1.288

Dentre os núcleos de PIC16C5X e PIC16F5X, o núcleo ClaiRISC aparenta ser o mais interessante.

O SLC1657 apesar de possuir WDT, Sleep e Timer, ele apresenta freqüência de operação muito baixa, sendo inferior a freqüência de operação do microcontrolador no qual foi baseado, e o consumo de elementos lógicos é elevado em relação ao ClaiRISC.

O RISC5x e o RISC8 possuem consumo de elementos lógicos elevado, em relação ao ClaiRISC, e freqüência de operação inferior ao ClaiRISC.

O PPX16-P16C55, apesar de possuir Timer, possui consumo de elementos lógicos levemente superior ao ClaiRISC e freqüência de operação levemente inferior ao ClaiRISC.

O Mini-RISC possui frequencia de operação levemente superior ao ClaiRISC, e apesar de possuir Timer, possui consumo de elementos lógicos levemente superior ao ClaiRISC.

Para a simples substituição do PIC16F84, poderia ser utilizado o CQPIC, porém, o núcleo ClaiRISC foi escolhido porque é o núcleo com o menor consumo de elementos lógicos, e apesar de não possuir WDT, Sleep e Timer, possui um código extremamente simplificado, facilitando alterações necessária para aplicações específicas.

4.13 Aplicação do núcleo

A esteira industrial, apresentada na Figura 17, serve para rejeitar peças metálicas e/ou não metálicas. Quando a esteira é ligada, ela deve se movimentar para a esquerda, e o pistão (6) deve recolher sua haste. Ao colocar uma peça na esteira, esta se movimenta para a esquerda, em direção ao sensor óptico (3). Quando a peça atinge o sensor óptico, a esteira deve parar por alguns segundos. A esteira então deve se movimentar para a direita, fazendo a peça passar pelo sensor indutivo (4) e em seguida o sensor capacitivo (5). O sensor indutivo detecta apenas peças metálicas, e o sensor capacitivo detecta peças de qualquer material. Quando a peça passa pelos sensores, se ambos detectarem sua presença, a peça é metálica, caso contrário é não metálica. Se a peça deve ser rejeitada, a esteira para com a peça em frente ao pistão, e o pistão empurra a peça para fora da esteira. Se a peça não deve ser rejeitada, a esteira continua movimentando-se para a direita até que a peça caia no fim dela. Na disciplina "SEL0338 - Tópicos Especiais em Sistemas Digitais" [36] o autor realizou o controle da esteira utilizando um kit com um PIC16F84.

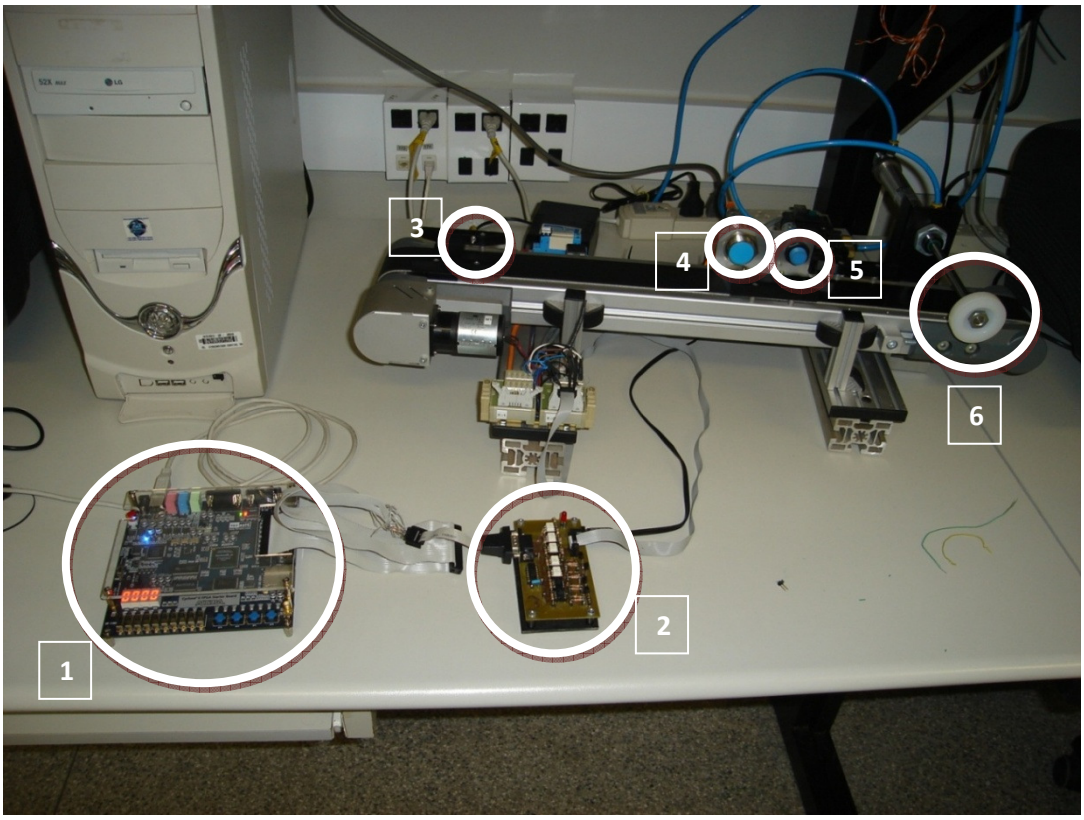

Figura 17 - Modelo de esteira industrial utilizando o núcleo ClaiRISC

O kit com um PIC16F84 foi substituído pelo kit de desenvolvimento DK-CYCII-2C20N (1) com o núcleo ClaiRISC implementado. O kit foi então conectado a um circuito (2) que faz a interface entre o kit e a esteira, sensores e pistão.

Os displays de 7 segmentos do kit de desenvolvimento foram utilizados para mostrar o número de peças metálicas e não metálicas que passaram na esteira. Para controlar os displays e realizar a contagem das peças, um módulo foi implementado na FPGA. Este módulo funciona como um periférico, e é conectado em dois pinos de I/O do ClaiRISC, um correspondendo às peças metálicas e outro correspondendo às peças não metálicas. Sempre que o ClaiRISC detecta o tipo da peça presente na esteira, ele gera um pulso no pino de I/O correspondente. O módulo implementado detecta este pulso, incrementa um contador interno e mostra o valor da contagem nos displays.

Para realizar o controle da esteira, foi necessária a utilização de um temporizador. Como o ClaiRISC não possui Timer, um módulo temporizador foi implementado na FPGA. O módulo temporizador funciona como um periférico do PIC, sendo controlado através de um PORT adicionado ao núcleo. Um dos pinos do PORT adicionado liga ou desliga o temporizador. O contador do temporizador é incrementado a cada 0,5 segundo, e quatro pinos do PORT adicionado definem o valor máximo do contador do temporizador, podendo temporizar até 6 segundos. Quando a contagem termina um pino de entrada é colocado em 1 para que o núcleo detecte a fim da contagem. Não há lógica de detecção automática, o núcleo deve verificar o pino.

Terminado o desenvolvimento do controle da esteira, o núcleo consumiu 512 elementos lógicos (3% dos elementos lógicos).

Capítulo 5 - Conclusões

Através da verificação, implementação e análise dos núcleos de código aberto de microcontroladores PIC16, verificamos que eles possuem vantagens sobre os microcontroladores PIC16. Com exceção do núcleo SLC1657, todos os outros núcleos analisados podem ser utilizados com freqüência de operação superior à freqüência de operação do microcontrolador PIC no qual foi baseado. As memórias podem ser implementadas com o tamanho desejado, de acordo com a necessidade. O número de pinos de entrada/saída podem ser alterados conforme a aplicação. Funções como timers e interrupções podem ser adicionadas ao núcleo. Outros núcleos de código aberto, como controladores USB, I2C e Ethernet, podem ser utilizados juntamente com os núcleos de microcontroladores. Todas estas características tornam a utilização de núcleos de microcontroladores implementados em FPGAs interessante em aplicações específicas.

A FPGA Cyclone II EP2C20F484C7N, utilizada neste trabalho, possui 18.752 elementos lógicos e 239.616 bits de memória. O núcleo ClaiRISC consumiu 318 elementos lógicos e 25.600 bits de memória. Na FPGA utilizada neste trabalho podem ser implementados 9 núcleos ClaiRISC (núcleo de microcontrolador PIC16F57), consumindo 18% dos elementos lógicos e 96% dos bits de memória. Em 21/10/2010, uma FPGA Cyclone II EP2C20F484C7N custava 56,30US$ (preço verificado no *site* da Altera) e um microcontrolador PIC16F57 custava 0,94US$ (preço verificado no *site* da Microchip). Portanto, cada núcleo ClaiRISC implementado em uma FPGA Cyclone II EP2C20F484C7N teria um custo de 6,25US$. Neste caso, o custo por microcontrolador seria 6 vezes mais caro, porém ainda restam 82% de elementos lógicos na FPGA para serem utilizados. Uma abordagem interessante seria utilizar uma FPGA de menor custo, menor número de elementos lógicos e com mais bits de memória, já que o fator limitante na FPGA Cyclone II EP2C20F484C7N foram os bits de memória. Uma FPGA Cyclone III EP3C5E144C8, por exemplo, possui 5.136 elementos lógicos e 423.936 bits de memória, e nela podem ser implementados até 13 núcleos ClaiRISC, valor limitado pelo número de elementos lógicos. Em 21/10/2010, uma FPGA Cyclone III EP3C5E144C8 custava 12,80US$ (preço verificado no *site* da Altera), resultando num custo de 0,98US$ por núcleo implementado. Neste caso, o preço por núcleo é bem próximo do preço de um PIC. Então, a utilização de núcleos de microcontroladores implementados em FPGA pode ser interessante na substituição de um número grande de microcontroladores.

Foi constatado através deste trabalho que os núcleos de código aberto de microcontroladores devem ser utilizados com cuidado porque a maioria apresenta algum tipo de erro durante a execução, ou não possui a lógica necessária para a execução de alguma função. Portanto, antes da utilização de algum núcleo, é necessário fazer a análise e verificação do funcionamento deste.

Trabalhos futuros podem abordar os seguintes pontos:

- Avaliar o que é mais vantajoso, a implementação de memórias utilizando elementos de memória da FPGA ou utilizando elementos lógicos, através de matrizes (array);
- Verificar o funcionamento dos núcleos utilizados neste trabalho em FPGAs de outros fabricantes;
- Verificar se os núcleos funcionam na freqüência de operação que o Quartus II diz ser a máxima;
- Verificar se núcleos são mais ou menos susceptíveis a ruído do que microcontroladores;
- Utilizar os núcleos em aplicações complexas, fazendo modificações no núcleo e utilizando-o com outros núcleos como controladores USB, I2C e Ethernet;
- Implementar novas instruções para o núcleo;
- Explorar a utilização de vários núcleos em uma FPGA para realizar multiprocessamento.

Referências Bibliográficas

[1] Atmel Corporation. Diponível em <www.atmel.com>. Acesso em: 21/10/2010.

[2] FPGA, CPLD and ASIC from Altera. Disponível em <www.altera.com>. Acesso em: 21/10/2010.

[3] Lattice Semiconductor. Disponível em <www.latticesemi.com>. Acesso em: 21/10/2010.

[4] Cypress Semiconductor. Disponível em <www.cypressmicro.com>. Acesso em: 21/10/2010.

[5] Anadigm, The dpASP company. Disponível em <www.anadigm.com>. Acesso em: 21/10/2010.

[6] Thomas, T.. Technology for IP Reuse and portability, IEEE Design & Test of Computers, Volume 16, Issue 4, pp. 7–13, 1999.

[7] Atmel Products – FPSLIC (AVR with FPGA). Disponível em <http://www.atmel.com/products/fpslic/default.asp>. Acesso em: 21/10/2010.

[8] About Excalibur Embedded Processor Solution. Disponível em <http://www.altera.com/products/devices/excalibur/exc-index.html>. Acesso em: 21/10/2010.

[9] Embedded Systems. Disponível em <http://www.altera.com/products/ip/processors/nios/nio-index.html>. Acesso em: 21/10/2010.

[10] Programable System-on-Chip - PSoC. Disponível em <http://www.cypress.com/?id=1353>. Acesso em: 21/10/2010.

[11] Gimenez, Salvador Pinillos. Microcontroladores 8051: Teoria do hardware e do software / Aplicações em controle digital / Laboratório e simulação. São Paulo: Pearson Prentice Hall, 2002.

[12] ASIC definition. Disponível em <http://www.vlsichipdesign.com/index.php/Chip-Design-Articles/asic-definition.html>. Acesso em: 21/10/2010.

[13] Design Security in Nonvolatile Flash and Antifuse FPGAs. Disponível em <http://www.design-reuse.com/articles/6529/design-security-in-nonvolatile-flash-and-antifuse-fpgas.html>. Acesso em: 21/10/2010.

[14] Quartus II Web Edition v8.0 SP1. Disponível em <https://www.altera.com/download/quartus-ii-we/dnl-quartus_we-v80.jsp>. Acesso em: 21/10/2010.

[15] ModelSim-Altera 6.1g Software. Disponível em <https://www.altera.com/download/modelsim/dnl-msim-61g-qii80.jsp>. Acesso em: 21/10/2010.

[16] The EDA Technology Leader – Mentor Graphics. Disponível em <http://www.mentor.com>. Acesso em: 21/10/2010.

[17] FPGA Board, FPGA Development Kit. Disponível em <http://www.altera.com/products/devkits/altera/kit-cyc2-2C20N.html>. Acesso em: 21/10/2010.

[18] Buy On-Line – BuyAltera.com. Disponível em <http://www.buyaltera.com/scripts/partsearch.dll?Detail&name=544-1669-ND>. Acesso em: 21/10/2010.

[19] PIC16F84A. Disponível em <http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010230>. Acesso em: 21/10/2010.

[21] OpenCores. Diponível em <www.opencores.org>. Acesso em: 21/10/2010.

[22] CQPIC support page. Disponível em <http://www002.upp.so-net.ne.jp/morioka/cqpic.html>. Acesso em: 21/10/2010.

[23] library of parameterized modules (LPM) Definition. Disponível em <http://quartushelp.altera.com/9.1/mergedProjects/reference/glossary/def_lpm.htm>. Acesso em: 21/10/2010.

[24] Mini-Risc core. Disponível em <http://www.opencores.org/?do=project&who=minirisc>. Acesso em: 21/10/2010.

[25] PPX16 mcu. Disponível em <http://www.opencores.org/?do=project&who=ppx16>. Acesso em: 21/10/2010.

[26] Amore, Robertod'. VHDL: Descrição e Síntese de Circuitos Digitais. Rio de Janeiro: LTC, 2005. p. 198-203

[27] RISC5x. Disponível em <http://www.opencores.org/?do=project&who=risc5x>. Acesso em: 21/10/2010.

[28] Free RISC8 (Verilog). Disponível em <http://www.mindspring.com/~tcoonan/newpic.html>. Acesso em: 21/10/2010.

[29] Chapter 3: Verilog Syntax Details. Disponível em <http://www.verilogtutorial.info/chapter_3.htm>. Acesso em: 21/10/2010.

[30] risc16f84. Disponível em <http://www.opencores.org/?do=project&who=risc16f84>. Acesso em: 21/10/2010.

[31] PIC FPGA Cores. Disponível em <http://www.embeddedtronics.com/pic_core.html>. Acesso em: 21/10/2010.

[32] LGPL – Definição de LGPL. Disponível em <http://www.guiadohardware.net/termos/lgpl>. Acesso em: 21/10/2010.

[33] Ashenden, Peter J.. The designer`s guide to VHDL. California; Morgan Kaufmann Publishers, Inc.. p. 42.

[34] ClaiRISC – runs 12bit opcode PIC family . Disponível em <http://www.opencores.com/?do=project&who=lwrisc>. Acesso em: 21/10/2010.

[35] Moreno, E. M.;Penteado, C. G.; Rodrigues, A. C. Microcontroladores e FPGAs: Aplicações em automação. São Paulo: Novatec Editora Ltda., 2005, Capítulo 15.

[36] Tópicos Especiais em Sistemas Digitais. Disponível em <http://iris.sel.eesc.usp.br/sel338/ >. Acesso em: 21/10/2010.

APÊNDICE A – Atribuição em Verilog

Na linguagem Verilog existem dois tipos de atribuição, que são "=" e "<=". A atribuição "=" é chamada de blocking assignment e a atribuição "<=" é chamada nonblocking assignment. A diferença entre os dois tipos de atribuição é que ao utilizar blocking assignment, o código dentro de um begin/end é executado sequencialmente, e ao utilizar nonblocking assignment, o código dentro de um begin/end é executado paralelamente.

A diferença entre os dois tipos de atribuição pode ser entendida acompanhando o modo como são executados os Códigos 1 e 2.

```
always@(posedge clk)
  begin
    a = in;
    b = a;
  end
```

Código 1 - blocking assignment (seqüencial)

Se na subida do sinal clk, in=1, o Código 1 é executado da seguinte maneira:

- Realizo a atribuição "a=in", portanto a=1;
- Na seqüência, após a primeira atribuição, realizo a atribuição "b=a", portanto b=1.

Conclusão: após a subida do sinal clk, utilizando blocking assignment (seqüencial), temos a=1 e b=1.

```
always@(posedge clk)
  begin
    c <= in;
    d <= c;
  end
```

Código 2 - nonblocking asignment (paralelo)

Se na subida do sinal clk, in=1 e c=0, o Código 2 é executado da seguinte maneira:

- Realizo a atribuição "c<=in", portanto c=1;
- Em paralelo (no mesmo instante), realizo a atribuição "d<=c", portanto d=0.

Conclusão: após a subida do sinal clk, utilizando nonblocking assignment (paralelo), temos c=1 e d=0.

Os comportamentos dos códigos, descritos anteriormente, podem ser visualizados no instante 10 da Figura A.1.

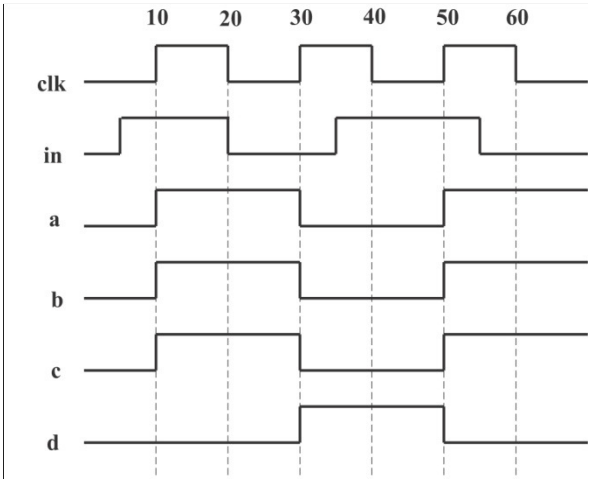

Figura A.1 – Formas de onda dos códigos 1 e 2