ESCOLA POLITÉCNICA
DE PERNAMBUCO

IMPLEMENTAÇÃO DE UM ALGORITMO DE COMPRESSÃO EM HARDWARE

Trabalho de Conclusão de Curso

Engenharia da Computação

Tiago Lima de A. Fonseca

Orientador: Prof. Abel Guilhermino da Silva Filho

Recife, novembro de 2005

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Tiago Lima de A. Fonseca

IMPLEMENTAÇÃO DE UM ALGORITMO DE COMPRESSÃO EM HARDWARE

Resumo

Como previsto pela lei de Moore [7] (Gordon Moore,1965), o número de transistores em circuitos integrados, é dobrado a cada 18 meses. Isso faz com que aplicações digitais sejam capazes de agregar cada vez mais funcionalidade em equipamentos de menor volume, associados a menor custo, menor consumo de potência e maior eficiência. Aplicações que envolvem sensores inteligentes, controles com inteligência artificial (lógica *Fuzzy*, redes Neurais, etc), controles adaptativos, processamento digital de sinais, processamento de imagem, multimídia, etc, exigem alta velocidade de computação que, em muitos casos, não é suprida pelos sistemas embarcados convencionais, baseados apenas em microprocessadores.

A utilização de lógica reconfigurável tem sido uma tendência no campo da microeletrônica, tornando-se uma boa solução para o projetista que deseja prototipar, além de contribuir para um reduzido tempo de desenvolvimento de projetos, quando comparado com o desenvolvimento de chips dedicados, conhecidos como ASIC.

Há uma tendência dos sistemas embarcados seguirem arquiteturas que envolvem Componentes de *Hardware* Embarcados, baseados em lógica reconfigurável (FPGA), que permitem implementar funções com alto grau de paralelismo e com o desempenho desejado, ou funções que, além de exigir bom desempenho - não suprida por microprocessadores devem ser dinamicamente modificadas ao longo do funcionamento do sistema.

Neste trabalho apresentamos uma solução para compactação de arquivos, voltada para arquivos de imagem, usando *hardware* para realizar tal operação. Mostraremos um algoritmo implementado em VHDL para configuração de FPGAs. Como estudo de caso, aplicaremos o algoritmo implemetado, em uma dada figura.

Abstract

The Moore's law [7] (Gordon Moore,1965) foresaw that the number of transistors per square inch on integrated circuits doubled at least in 18 months. This implies that digital applications are able to include more functionality in components equipments with lower volume, lower cost, lower power consumption and more efficiency. Applications that involve intelligent sensors, artificial intelligence (Fuzzy logic, Neural networks) adaptative control, digital signal processing, image processing, multimedia etc, need high computational speed that, in many cases, is not supplied by the conventional embedded systems, based only in microprocessors.

The use of reconfigurable logic has been a tendency in the microelectronic systems, becoming a good solution for the designer that needs to build a prototype. In addiction, the use of reconfigurable logic contributes to reduce the amount of time required to develop projects, in comparison with the development of dedicated chips (ASICs -Application Specific Integrated Circuits). There is a tendency in the embedded systems to follow a architecture that involves Embedded *Hardware* Components, based on reconfigurable logic (FPGA - Field Programmable Gate Array). The FPGAs, for example, allows implementing functions with high degree of parallelism and with a desired performance. It also allows that changes be made dynamically, during the system execution. In this work we present a solution to compress files, specifically image files, using *hardware*. It will be shown an algorithm implemented in VHDL to configure FPGA's. This solution presents a better performance because it is implemented in *hardware*.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

ASIC – *Application-Specific Integrated Circuit*
Bitstream – Seqüência de *bits*
bps – *bits* por segundo
CAD – *Computer Aided Design*
CI – Circuito Integrado
CLB – *Common Logic Blocks*
FPGA – *Field-Programmable Gate Array*
NLA – Nível Lógico Alto
NLB – Nível Lógico Baixo
PLD – - *Programmable Logic Device*
RAM – *Random Access Memory*
RF – Rádio Freqüência
RLE – *Run-Length Encode*
TTM – *Time-To-Market*
VHDL – VHSIC Hardware Description Language
VHSIC – *Very High Speed Integrated Circuit*

Agradecimentos

Agradeço a minha família pelo sincero apoio dedicado, não apenas nos momentos da minha vida acadêmica, mas durante toda ela. Principalmente aos meus irmãos Humberto e Caio.

Aos meus amigos e colegas do CESAR, Centro de Estudos e Sistemas Avançados do Recife, pelo fornecimento de apoios técnicos. Essencialmente ao meu colega de turma, Rodrigo Gomes, pela atenção durante algumas tardes de sábado.

A Abel Guilhermino, meu orientador, graças ao incentivo e fornecimento das fontes de pesquisa e, aos engenheiros da computação, da 1ª Turma de Engenharia da Computação formada na Escola Politécnica de Pernambuco, “Os cobaias da Poli”.

E a minha namorada, Aracelli, pela essencial e contrária distância estabelecida nos últimos dias da produção desta monografia.

Capítulo 1

Introdução

O desenvolvimento de projetos envolvendo ASICs proporciona uma série de desvantagens que são consideráveis na tomada de decisão de projetos. As melhorias tecnológicas, que permitem a construção de transistores cada vez menores, tem tornado cada vez mais complexo o projeto das máscaras para a produção dos circuitos [2]. Isto tem elevado muito o custo de projeto. Como o FPGA é um circuito integrado já pronto e encapsulado, o projeto de sistemas embarcados com estes dispositivos não vai até o nível de definição de máscaras, limitando-se a definição de um padrão de configuração da lógica na forma de um *bitstream*, o que diminui sensivelmente a complexidade e custo de projeto. A Figura 1 demonstra a vantagem no desenvolvimento de projetos utilizando a tecnologia de FPGAs sobre os ASICs considerando as várias etapas de desenvolvimento de projetos (especificação, implementação, verificação, prototipação e avaliação) [2].

Apesar de, quando produzidos em larga escala, os ASICs serem mais econômicos que os FPGAs, dados atuais demonstram que o custo de FPGAs cai 15% ao ano enquanto o custo dos ASICs cai 3% ao ano [2]. Além disso, os FPGAs agregam a vantagem da capacidade de reconfiguração, o que permite que o mesmo *hardware* implemente múltiplas funcionalidades, que podem ser modificadas ao longo do tempo de operação do sistema.

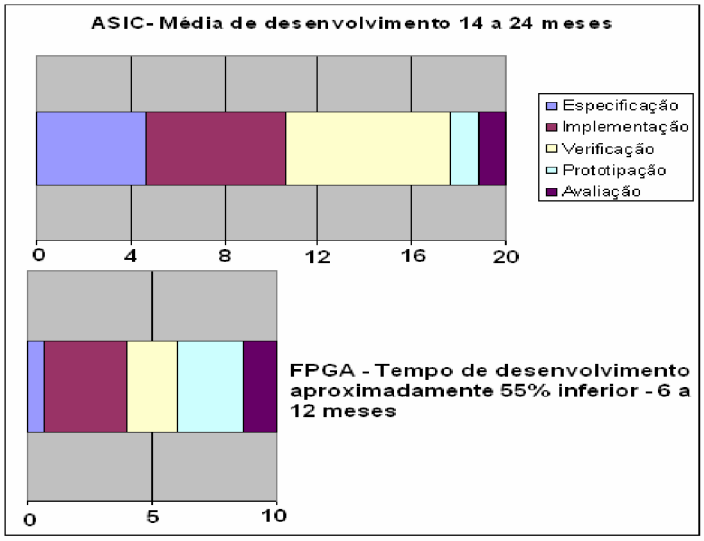

Figura 1. Diferenças de Complexidade nos projetos com ASIC e FPGAs

Uma outra abordagem que pode ser levada em consideração, é o uso de microcontroladores, que são circuitos integrados que executam uma seqüência de instruções. Esta é uma solução com baixo custo e versátil, porém não apresenta um bom desempenho, se comparada com FPGAs e ASICs.

O ponto de partida para a proposta deste trabalho é a utilização de uma arquitetura reconfigurável para desenvolvimento de projetos. Uma metodologia que auxilie o projetista no desenvolvimento de projetos de sistemas embarcados é necessária, além de ferramentas CAD que proporcionarão o suporte nas várias etapas de desenvolvimento de projetos.

Este trabalho tem como objetivo descrever a implementação de um algoritmo de compressão em hardware. Duas abordagens foram exploradas, uma considerando o algoritmo sendo implementado em *hardware*, e outra abordagem em *software*. A abordagem em hardware considera que o algoritmo será implementado usando componentes como FPGAs e a abordagem em software, usando componentes como microcontroladores. Um estudo comparativo entre as abordagens foi realizado, utilizando-se o algoritmo de compressão. Esse será descrito em maiores detalhes no decorrer desta monografia.

No capítulo 2, serão abordados alguns conceitos que dão suporte a um melhor entendimento dos assuntos encontrados nesse trabalho, tais como compressão de dados e tecnologias de *hardware*.

O capítulo 3 mostra a descrição das etapas do desenvolvimento deste trabalho, destacando etapas como observação do problema, escolha do algoritmo de compressão e implementação. Logo em seguida, no capítulo 4, é descrita toda a fase de implementação, mostrando como o sistema foi validado através de simulação.

O capítulo 5 apresenta um estudo de caso, onde consideramos uma pequena imagem para validar a implementação do algoritmo. Foram analisados resultados de desempenho considerando duas abordagens (*hardware e software*) que serão descritas no decorrer da monografia.

Finalizamos com o capitulo 6, mostrando a conclusão do trabalho e sugestões para trabalhos futuros.

Capítulo 2

Conceitos Básicos

Inicialmente alguns conceitos são importantes para fundamentar idéias encontradas nesse trabalho. Os assuntos levantados nesse capítulo serão utilizados, e estão relacionados à aplicação que será construída. Portanto, temos nesse capítulo, noções de Compressão de dados e algoritmos de compressão, bem como a arquitetura de FPGAs e de microcontroladores.

2.1 Compressão de dados

A compressão de dados não serve apenas para reduzir o tamanho dos dados, há várias outras aplicações que utilizam a compressão de dados. Obviamente, a **redução do espaço físico** utilizado é um elemento importante do conjunto de aplicações envolvendo compressão de dados, mas existem aplicações envolvendo **a** melhoria do **desempenho na transmissão de dados**. [9]

A redução do espaço físico é mais comumente utilizada em bancos de dados que, incorporando a compressão no projeto de seus registros, permitem um significativo ganho em termos de ocupação em disco e velocidade de acesso.

A compressão também é utilizada para aumentar a velocidade da transmissão de dados basicamente de duas formas: alterando a taxa de transmissão e permitindo o aumento no número de terminais de uma rede.

Como exemplo, pode-se ter a taxa de transmissão de um modem que transmite dados a 9400 bps, ampliada para uma transmissão real de 14400 bps, bastanto que os modems envolvidos na comunicação possuam algoritmos que possam compactar e descompactar os dados.

2.2 Tipos de Compressão

Essa seção mostra, além dos tipos de compressão, alguns algoritmos usados para compressão de dados.

2.2.1 Compressão Lógica

A compressão lógica refere-se ao projeto de representação otimizada de dados. Um exemplo clássico é o projeto de um banco de dados, que ao invés de representar os campos de dados

utilizando inteiros ou caracteres, utiliza seqüências de *bits* para a representação de campos de dados. No lugar de seqüências de caracteres ou inteiros, utilizam-se *bits*, reduzindo significativamente o espaço de utilização do banco de dados [9].

2.2.2 Compressão Física

A compressão física é aquela realizada sobre dados existentes, e a partir destes, dados que repetem-se, podem ser codificados. Existem dois tipos de técnicas para sinalizar a ocorrência de caracteres repetidos; Um deles indica o caracter (ou conjunto de caracteres) repetido através da substituição por um caracter especial. Outras técnicas indicam a freqüência de repetição de caracteres e representam isto através de seqüências de *bits*. [9]

A compressão física pode ser feita de duas maneiras: orientada a caracter e estática. Estas duas técnicas serão mostradas nas seções seguintes.

2.2.2.1 Compressão Orientada a Caracter

Antes da discussão das técnicas propriamente ditas, é necessário o esclarecimento de como são codificados os caracteres especiais a serem substituídos pelos caracteres repetidos neste tipo de compressão física. Há basicamente três formas de representação denominadas *run-length* , *run-length* estendido e inserção e deleção.

a) codificação *run-length* (RLE)

Quando temos um arquivo onde ocorre uma repetição contínua de determinado caracter, por exemplo, AAAAAAAAAAAA, é possível sua representação através da codificação *run-length* , da seguinte forma:

Ce 12 A

Onde A é o caracter repetido 12 vezes, o que é indicado pelo caracter especial Ce . Por exemplo, o caracter especial pode ser um dos caracteres não-imprimíveis do código ASCII,.

O RLE apresenta uma limitação de contagem de caracteres. Esse algoritmo reserva um *byte* para representação do número de repetições do caracter, isso faz com que haja um perda da eficiência, porque caso um caracter repita-se mais de 255 vezes sequencialmente, um novo padrão deverá ser criado.

b) codificação *run-length* estendido

Para resolver o problema de limitação do RLE, utiliza-se o *run-length* estendido. A diferença deste para o *run-length* simples, é que existem caracteres delimitadores no início e no fim da seqüência de codificação, isso possibilita a inserção de mais de um byte para representação do número de repetições do caracter:

SO R A 980 SI

Onde SO (*shift out*) é um caracter especial indicador de início de uma seqüência de caracteres definida pelo usuário até que SI (*shift in*) seja encontrado. Esta é uma propriedade de códigos de caracteres, como o ASCII. O caracter R indica a compressão *run-length*, onde o caracter A é indicado como repetido 980 vezes. Como o valor 980 ultrapassa o limite de 255 de um *byte*, torna-se necessária a utilização de mais um *byte* para a representação do valor.

c) codificação por inserção e remoção

Há determinados casos em que os caracteres especiais utilizados conflitam com outros aplicativos, por exemplo, aplicativos de transmissão de dados. Desta forma, utiliza-se um caracter convencional como indicador de compressão.

Como na língua portuguesa utilizam-se muito pouco as letras K, W, Y, pode-se, por exemplo, indicar a compressão de um arquivo de texto na forma:

K 12 A

2.2.2.2 Compressão estática

A codificação estatística tem como objectivo atribuir a cada símbolo um código com base na sua frequência de ocorrência, tal que símbolos mais frequentes recebem códigos com menor comprimento que os símbolos menos frequentes.

Algoritmo de Huffman

O Algoritmo de Huffman comprime arquivos, usando as probabilidades de ocorrência dos seus símbolos (caracteres, bytes, etc).

Primeiramente, os símbolos são listados em ordem decrescente de probabilidade. Em seguida, os dois símbolos de menor probabilidade são combinados em um novo símbolo com probabilidade igual à soma das duas probabilidades originais. A probabilidade do novo símbolo é reposicionada na lista de acordo com seu valor, como mostrado na Figura 2. Esse procedimento é repetido até que restem apenas dois símbolos. A esses dois símbolos, são associados os valores binários 0 e 1, como mostrado na Figura 3.

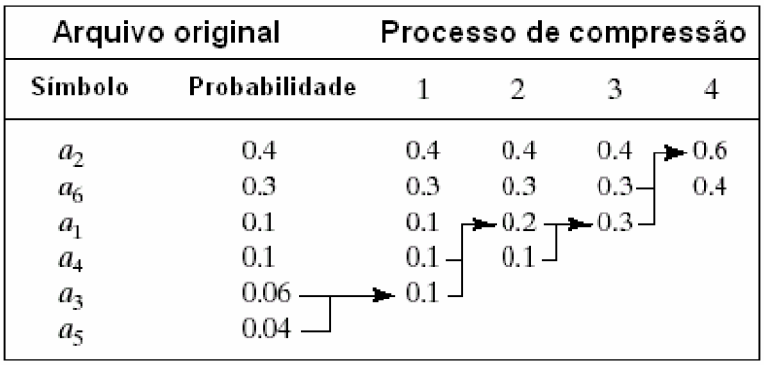

Figura 2. Processo de compressão usado pelo código de Huffman

Em seguida, novos códigos são gerados de acordo com as probabilidades somadas. Como resultado, tem-se que os símbolos com maior probabalidade terão um código menor, como ilustrado na Figura 3. Então no aquivo original, os símbolos são substituídos pelos códigos gerados.

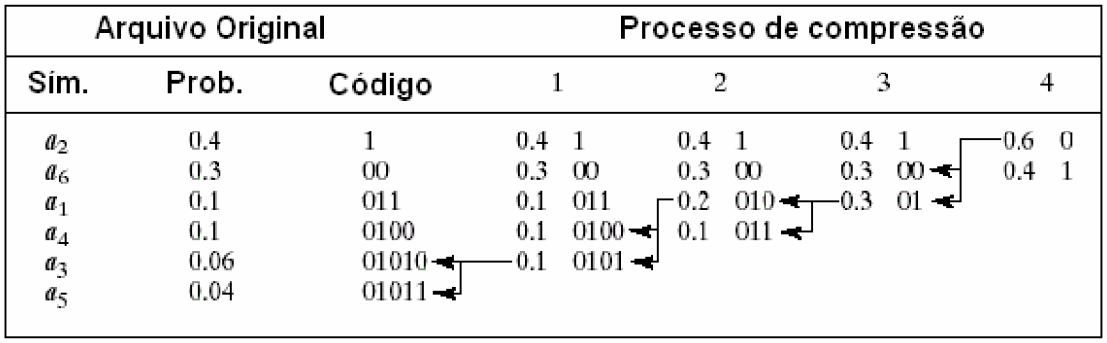

Figura 3. Associação entre os códigos e símbolos

2.3 FPGA

O FPGA faz parte de uma classe de dispositivos digitais chamados de PLDs. Estes são definidos como plataformas físicas configuráveis por *software*, ou seja, utiliza-se um programa para descrever a configuração que implementa um determinado circuito. Apresenta blocos lógicos idênticos simples, cuja interligação pode ser ou não implementada, resultando em circuitos complexos. O FPGA foi criado pela Xilinx Inc. em 1985, com o intuito de ser usado para propósito geral [3].

Os benefícios do uso de FPGA são muitos quando comparado com os projetos CMOS VLSI: diminui o custo inicial e os riscos apresentados quando se usam máscaras convencionais "*gate array*". Os FPGAs podem ser configurados várias vezes, e aceitam *clock* acima de 50 MHz. São utilizados principalmente em prototipação para validação de projetos, pois não requerem um período muito longo de implementação se comparado com os ASICs. Uma grande desvantagem do uso dos PLDs de um modo geral, é que estes são grandes, caros, tem um alto consumo de potência e são lentos comparados aos ASICs.

2.3.1 Tecnologias de FPGA

Existem quatro principais tecnologias para construção de FPGAs, são elas: RAM estática, transistores de passagem, EPROM e EEPROM. O desempenho de cada uma depende da aplicação.

RAM estática

As conexões entre os blocos lógicos são implementadas através de portas de transmissão ou multiplexadores controlados por células SRAM, e tem como vantagem permitir a rápida reconfiguração, porém exige *hardware* auxiliar. Este é o tipo de tecnologia mais usada na

atualidade, podendo-se citar grandes fabricantes, como: Altera Corporation [4] e Xilinx Corporation [3], que detêm este tipo de tecnologia. A Figura 4 mostra o exemplo da arquitetura interna de um FPGA da Xilinx [3].

Transistores de Passagem

Transistores são dispositivos eletrônicos que podem funcionar como chaves, abrindo ou fechando circuitos. Portanto, eles pode ser usados para estabelecer conexões entre os blocos lógicos.

EPROM/EEPROM

Essa tecnologia é semelhante à encontrada nas memórias EPROM/EEPROM. Tem a vantagem de não perder a configuração quando interrompida a fonte de energia.

2.3.2 Arquitetura de um FPGA

A programação do FPGA é realizada através da leitura de uma matriz de dados, que pode ser via comunicação serial ou paralela. Os dados são fornecidos por uma memória externa auxiliar, que pode ser lida na inicialização do sistema.

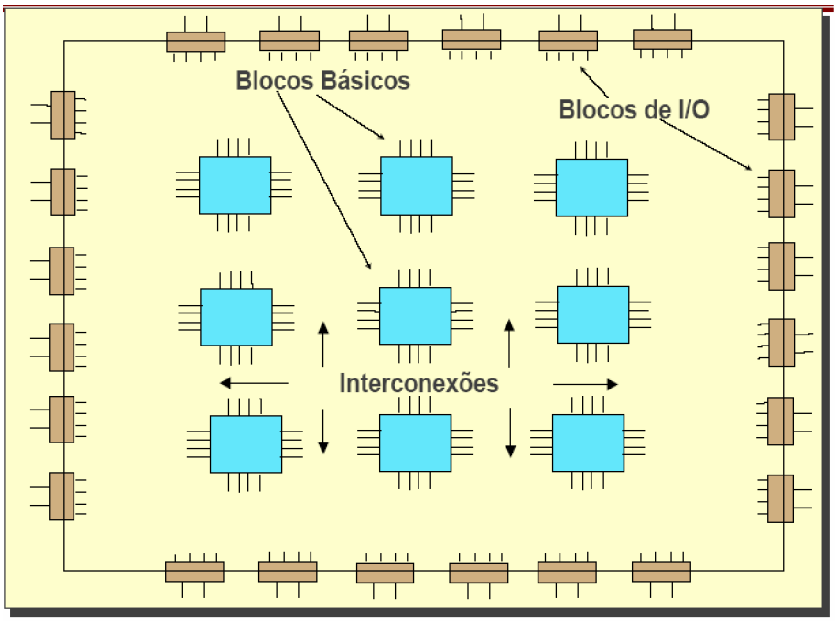

Figura 4. Arquitetura de um FPGA genérico.

O FPGA possui três conjuntos de elementos de configuração:

- CLB – Blocos lógicos configuráveis: São circuitos idênticos compostos por *flip-flops* e lógica combinacional. Os CLBs são compostos de LUTs (*Lookup Table*), que são estruturas que pode implementar qualquer função lógica com um número fixo de variáveis. Por exemplo, as LUTs podem ser programadas para executar funções de memória RAM e *dual port* RAM, usando lógica seqüencial ou combinacional. A Figura 5 mostra a estrutura interna de um CLB genérico. Para esta arquitetura mostrada, o CLB apresenta somente uma LUT, mas esse pode ser composto por mais LUTs.
- IOB – Input Output Blocks, são circuitos de interface dos CLBs com o exterior do FPGA, constituídos por buffers bidirecionais. No FPGA, os pinos dos IOBs podem ser programados como saída, entrada, bidirecional, tri-state, pull-up ou pull-down, etc.
- Interconexões – possuem trilhas para conectar as entradas e saídas dos CLBs e IOBs. Essas ligações geralmente são feitas por programação das células de memória estática. O processo de escolha das trilhas é chamado de roteamento.

As conexões internas são compostas por segmentos metálicos e pontos de chaveamento programável, para que o roteamento possa ser feito. Existe uma grande quantidade de possibilidades para se definir uma trilha de forma automática e eficiente. Tais trilhas podem ser classificadas em três grupos: linhas simples, linhas duplas e linhas longas. Existem ainda trilhas que possuem características especiais, apropriadas para transmitir sinais globais de alta freqüência. Estas trilhas podem ser utilizadas como via de transmissão de sinal de *clock* ou de controle geral de habilitação de leitura e escrita.

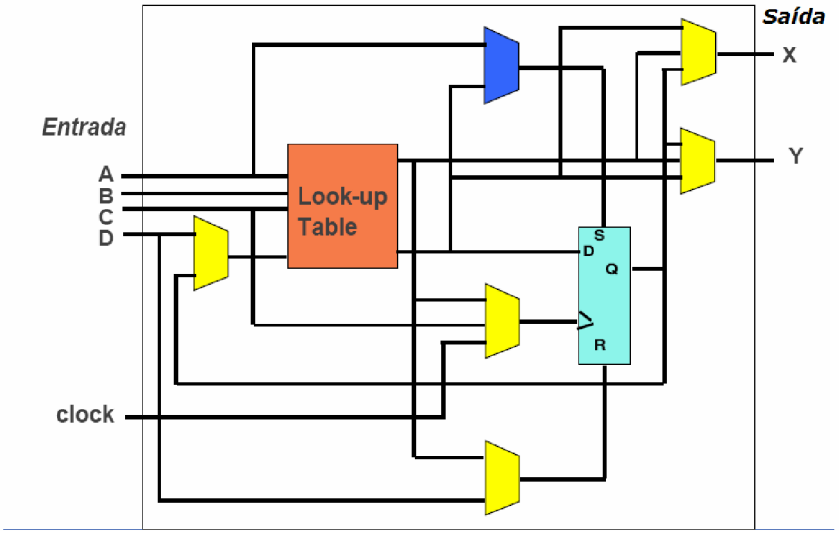

Figura 5. Estrutura interna de um bloco lógico configurável

Um CLB pode implementar uma ou duas funções booleanas com diferentes números de variáveis. A saída de cada CLB pode ser armazenada em *flip-flops*. Quando se utiliza *flip-flops*, o CLB pode executar lógicas seqüenciais, do contrário, apresenta apenas lógica combinacional. A escolha de se usar ou não, *flip-flops* pode ser feita via programação.

Programando o FPGA

É possível programar todo o projeto, utilizando a lógica dos CLBs, isto é, o programador especifica quais ligações serão feitas entre os CLBs, porém existem pacotes de desenvolvimento que permitem um projeto digital bem menos desgastante, utilizando-se editores esquemáticos, editores de diagrama de estados ou editores para linguagens descritivas como VHDL. Os editores esquemáticos muitas vezes apresentam blocos funcionais comuns como: registradores, *buffers*, contadores memórias, decodificadores, somadores, entre outros, que podem ser usados para construir circuitos maiores, de uma forma rápida e confiável, pois os circuitos básicos já foram exaustivamente testados.

Tais editores geram streams de *bits* (seqüências binárias) que são carregados no FPGA. Os procedimentos para carregamento desse programa são: carregamento através de sinais enviados por um PC, por exemplo, através da porta paralela; ou a seqüência de *bits* é inserida numa PROM, que envia seus dados para o FPGA após um pulso de inicialização. Essa forma é a mais indicada para a versão final da interface, ou seja, quando ela não sofrerá mais alterações. Nesse projeto utilizamos um editor de VHDL, no qual é possível obter resultados de validação através de simulação.

Modos de configuração do FPGA

Configurar um FPGA significa carregar o 'programa' no dispositivo. Alguns FPGAs utilizam cerca de 350 *bits* de configuração por CLB [2]. Cada *bit* é responsável por definir o estado de uma célula de memória estática que controla uma função booleana, um multiplexador de entrada ou um transistor de interconexão.

Para algumas famílias, os modos de operação dos FPGAs podem ser configurados através de 3 *bits*. Temos como exemplos de modo de operação dos FPGAs:

- **Modo mestre serial**: Nesse Modo, o sinal de *clock* (CLK) é gerado a partir de um oscilador interno que serve como sincronizador dos periféricos, por exemplo, uma memória SPROM, que pode armazenar o arquivo de configuração do FPGA. Um aspecto importante desse modo de operação é que este permite que os dados de configuração sejam automaticamente carregados no FPGA imediatamente após a energização do circuito.
- **Modo escravo serial**: Diferentemente o Modo Mestre Serial, o Modo Escravo Serial não apresenta um *clock* interno, mas é gerado por um circuito externo. Este modo geralmente é usado na fase de validação do projeto, onde a parte funcional é mais importante e deve ser testada.

A escolha do modo de configuração mais apropriado depende de fatores como [8]: simplicidade de interface, configuração rápida, possibilidade de serem feitas várias configurações através de um microcontrolador, dentre outras.

Para se descartar a configuração do FPGA utiliza-se um circuito interno que é ativado na inicialização do FPGA, ou seja, sempre que o circuito é ligado. Esse circuito mantém a memória de configuração limpa enquanto o pino *PROG do FPGA estiver em NLB.

2.4 Microcontroladores

Nessa seção, serão mostrados algumas características dos microcontroladores. Isto servirá de base para entendimento da comparação com FPGAs, feita nas seções seguintes.

2.4.1 O que são microcontroladores

Para automatizar um sistema de controle (ex. controles de impressora, acionadores de motores de passo, reguladores de velocidade) [10] podem ser encontrados vários circuitos, como mostrado na Figura 6, que juntos provêem as funcionalidades desejadas dos sistemas, tais como:

- CPU: Usada para controle;
- ROM: Usada para armazenar o programa de controle;
- RAM: usada para implementação da pilha e armazenamento de dados;
- Porta Paralela: Comunicação com periféricos;
- Porta Serial: Comunicação com periféricos;
- A/D e D/A: Conversores para interface com periféricos analógicos;
- *Timers*: Temporização.

Após isso ser observado, começou a surgir a idéia de colocar todo esses periféricos dentro do CI da CPU. Isto diminuiria o tamanho e o custo do circuito impresso além de aumentar a confiabilidade. Assim surgem os microcontroladores, circuitos que apresentam num só chip, unidade de processamento e demais periféricos.

Esses circuitos executam programas que são armazenados em suas memórias internas. Os programas podem ser escritos em duas linguagens: a linguagem de alto nível *C* e a linguagem de montagem *Assembly*. Mas linguagem de máquina é utilizada.

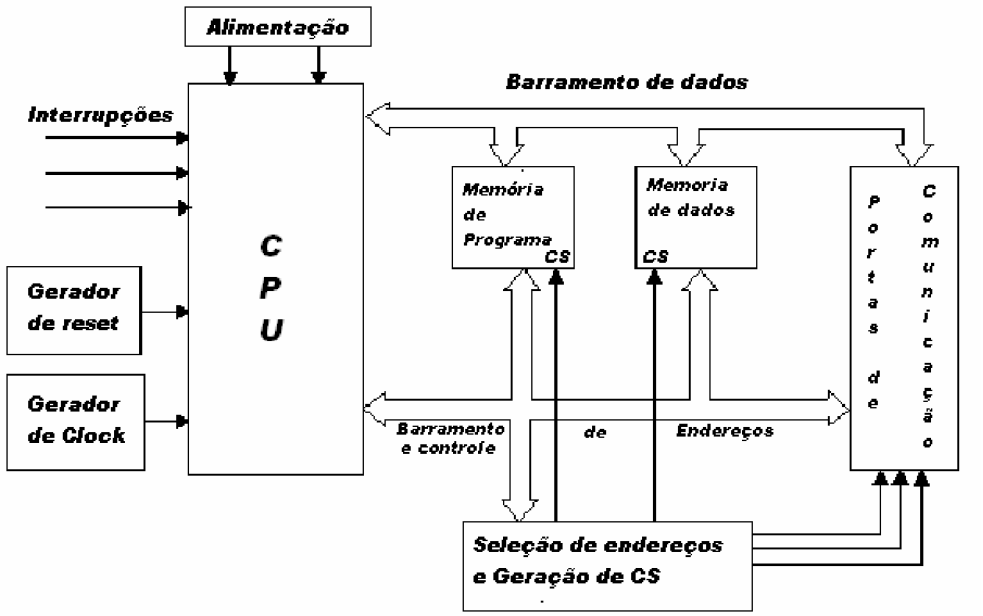

Figura 6. Arquitetura de um microcontrolador básico

A Figura 7 mostra um microcontrolador básico da família 8051.

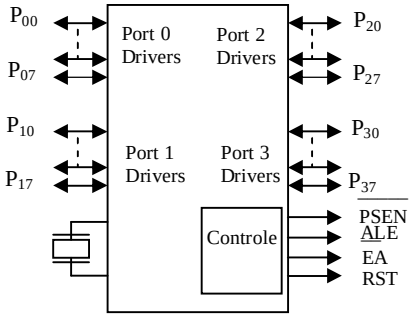

Figura 7. Microcontrolador básico da família 8051

O microcontrolador mostrado na Figura 7, possui 4 portas de comuincação de 8 bits, alguns sinais de controle (PSEN, ALE, EA e RST) e um cristal, que é usada para geração do *clock*.

Os microcontroladores são específicos para controle, não tem grande capacidade de processamento e por isso nunca haverá um computador pessoal cuja CPU seja um microcontrolador. Eles podem estar presentes num PC, mas apenas para controlar periféricos.

Usa-se o nome de Microcontrolador para designar dispositivos de uso genérico, mas existem vários microcontroladores que têm aplicações específicas, como por exemplo o controlador de teclado e o controlador de comunicações universal.

2.4.2 Componentes

Memórias

Usualmente, um dispositivo da família 8051 apresenta três tipos de memórias [11] e, para um programador de microcontroladores, é importante ter conhecimento básico sobre elas.
São elas:

- Memórias internas (*on-chip memories*): São as memórias localizadas dentro do chip, podendo ser de qualquer natureza (código, RAM e etc).
- Memória externa de código (*External Code Memory*): Esta memória está é uma extenção do chip e sua função é armazenar o programa que está sendo executado pelo 8051.
- RAM externa (*External RAM*): Está fora do chip e geralmente é representada por uma RAM padrão ou Flash RAM

Banco de registradores

Os 8051 básico possui 8 registradores que são usados em muitas de suas instruções. Estes registradores são numerados de 0 a 7 (R0, R1, R2, R3, R4, R5, R6, e R7). Estes são usados

geralmente ajudar na manipulação de valores e na movimentação de dados de uma posição de memória a outra.

Memória de *bit*

Os dispositivos da família 8051 são microcontroladores orientados a comunicação, portanto dão ao usuário a habilidade de acessar uma boa quantidade de variáveis de um *bit*.

É importante frisar que a memória de *bit* é realmente uma parte da RAM interna. De fato, as 128 variáveis de *bit* ocupam os 16 primeiros *byte*s da RAM interna.

Registradores de funções especiais (SFR)

Os registradores de funções especiais são áreas de memória que controlam uma funcionalidade específica do processador 8051. Por exemplo, quatro SFR permitem o acesso a 32 linhas de entrada/saída do 8051. Um outro SFR permite que um programa leia ou escreva na porta serial e assim por diante.

Timer

O 8051 básico possui 2 *timers/counters* internos, o TIMER0 e TIMER1, que são programados pelo software e trabalham de maneira independente dos demais sistemas do chip. Estes dois timers podem ser habilitados por software ou por hardware, seja em função dos bits em seus registros de controle, ou pelos pinos de interrupção.

Interrupções

No 8051 básico existem 5 maneiras de se gerar uma interrupção, são elas: pelas interrupções externas INT0 e INT1, pelos *timers* TIMER0 e TIMER1, e pelo canal de comunicação serial.

No 8051, cada interrupção pode ser individualmente habilitada ou não, e podemos também desabilitar todas de uma só vez. Para evitar que duas interrupções ocorram ao mesmo tempo, estas apresentam níveis de prioridade.

2.5 Comparação entre FPGA, ASIC e Microcontroladores

Com o grande avanço tecnológico alcançado atualmente, surgem novas tecnologias para resolver problemas de desempenho e custo do projeto, encontrados durante as fases de um projeto de *hardware*. Obviamente, os projetistas querem um alto desempenho a um custo baixo. Para isso, as opções tecnológicas disponíveis no mercado estão sempre inovando e melhorando o desempenho de seus *hardwares.*

Nessa seção faremos uma breve comparação entre as três abordagens que mais se destacam no mercado: FPGA, ASIC e Microcontroladores.

Quando tratamos de soluções em *hardware* (FPGA e ASIC), damos destaque ao desempenho dessas abordagens, pois estas apresentam um circuito dedicado a funcionalidade

desejada. No caso do FPGA, este pode ser configurado segundo um 'programa' escrito em alguma linguagem de descrição de *hardware*. Já os ASICs apresentam uma configuração preestabelecida na fábrica.

Os microcontroladores apresentam um conjunto blocos funcionais interligados. Essas conexões entre eles não podem ser mudadas, diferentemente do FPGA. A grande vantagem disso é que apresentam uma tecnologia muito barata, porém estes demonstram um desempenho muito menor que os FPGAs ou ASICs. A figura abaixo mostra uma comparação de desempenho, Custo de reengenharia, custo unitário e *Time-to-Market*(TTM), considerando ASICs, FPGAs e Microcontroladores. O custo de reengenharia trata-se do impacto causado por um erro achado durante o projeto. O TTM é o tempo que um projeto gasta para ser posto no mercado.

Figura 8. ASICxFPGAxMicrocontroladores

2.6 Hardware x software

Há uma grande discussão de quando se deve usar as diferentes abordagens para implementações de sistemas digitais. Essa seção é dedicada à descrição de sistemas implementados tanto em hardware, quanto em software .

O grande benefício de se implementar em *software* é o nível de abstração que obtemos quando utilizamos linguagens de alto nível como a linguagem C. O que faz com que o tempo de desenvolvimento seja bastante reduzido.

Entretanto, implementação via *software* apresenta desvantagens, principalmente relacionados a desempenho. Isso ocorre porque a plataforma de desenvolvimento utiliza microcontroladores, que são *hardware*s de propósito geral. Como visto anteriormente os blocos funcionais dos microcontroladores são fixos, o que faz com que otimizações de desempenho sejam possíveis apenas no nível de código de programa ou estratégia de implementação.

Já implementações em *hardware* apresentam blocos funcionais que são interligados de tal forma que executem a função desejada, o que faz com que o sistema tenha um desempenho muito melhor, se comparado com implementações em *software*. Porém, nessa abordagem, temos que nos preocupar muito mais com os detalhes da implementação e a arquitetura do dispositivo de destino. Minúcias, como atrasos dos blocos e sincronização, devem ser criteriosamente observadas, para que haja sucesso na construção do sistema.

Capítulo 3

Etapas do projeto

3.1 Observação do problema

Nessa fase foi levantado o problema de ter que transmitir imagens sem sobrecarregar o tráfego de dados entre o transmissor e o receptor, além de diminuir o tempo gasto com manipulação dessas imagens.

3.2 Escolha do Algoritmo de compactação RLE

Como será visto, este algoritmo é eficiente quando o arquivo apresentar uma característica peculiar que é possuir seqüências de padrões repetidos. Então este pode ser aplicado satisfatoriamente a imagens, pois estas, na maioria dos casos, apresentam tal característica.

Além de atender às necessidades do projeto, o algoritmo de compactação RLE é bastante simples de implementar. Este servirá para análise comparativa entre as implementações em *hardware* e *software*.

3.3 Fase de Implementação

A aplicação escolhida consiste na implementação do algoritmo de compactação de imagens, o RLE, mostrado anteriormente. Essa implementação foi feita em VHDL para configurar um FPGA, e validada através de simulações. Aplicações como esta, justificam o uso dos recursos do FPGA, pois apresentam uma razoável massa de dados, além da necessidade de um bom desempenho.

Para compararmos o desempenho dessa abordagem (implementação em *hardware*), esse mesmo algoritmo foi implementado em *software* (linguagem C) para ser executado em um microcontrolador da família 8051.

Como já foi dito, utilizamos a linguagem VHDL para descrição de nosso *hardware*. Uma das vantagens da utilização de linguagens de descrição de *hardware*, é que muitas vezes podemos nos abstrair da arquitetura quando esta ainda não foi determinada.

O fluxo de projeto indicado na Figura 9 ilustra as principais etapas do desenvolvimento de um sistema digital e será detalhado nas seções a seguir.

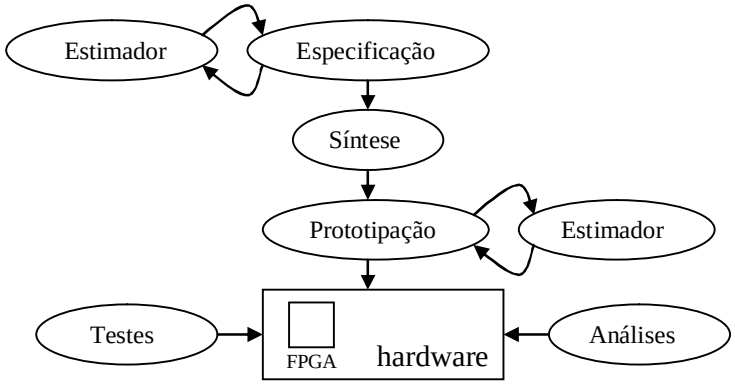

Figura 9. Fluxo de Projeto genérico de um projeto digital

3.3.1 Especificação

Inicia-se na especificação do sistema que se trata da descrição em alto nível do comportamento do sistema. Esta etapa é auxiliada pelas estimativas, podendo-se considerar restrições tais como área, potência e desempenho.

Nessa etapa do projeto, foram abordadas as possibilidades do uso de padrões para implementação do algoritmo em VHDL, a melhor abordagem foi modelar o sistema como uma máquina de estados de Mealy [11].

A máquina de estados é executada a partir de um sinal de *clock*, ou seja, a mudança de um estado para o outro é feita em sincronia com o pulso do *clock*. Cada estado apresenta uma configuração de sinais específica que podem ser descritos através de um diagrama de estados.

Durante esta etapa de projeto, observamos a necessidade de se separar o sistema em dois módulos para facilitar a implementação e futuramente possibilitar explorar paralelismo com inserção de novos módulos. Após essa divisão, passamos a pensar no comportamento de cada módulo, ou seja, definição dos sinais de controle, juntamente com entrada/saída, que seriam vistos como variáveis da máquina de estado. A máquina de estados foi definida com base no comportamento de cada módulo, bem como nos sinais gerados por ele.

3.3.2 Síntese

Logo após a especificação, vem a etapa de síntese que permite a passagem automática, através de ferramenta CAD adequada, de um nível de descrição para um outro nível inferior. Esta etapa normalmente pode ser agrupada em três níveis descendentes sucessivos: síntese comportamental, síntese lógica e síntese física. Na síntese comportamental é produzida uma descrição da funcionalidade do circuito no nível de blocos, a partir de uma descrição comportamental do mesmo. Esta etapa normalmente é dividida em escalonamento de operações, partilhamento de recursos e alocação de recursos. Na síntese lógica obtém-se uma descrição a nível lógico de um circuito, a partir de uma especificação funcional do mesmo. A síntese física permite a obtenção de um circuito (*layout*) a partir de uma descrição no nível lógico do mesmo. A etapa de síntese independe da arquitetura de FPGA que irá se utilizar.

3.3.3 Estimativas

Nesta etapa são levantadas algumas características, tais como, área, potência e desempenho visando atender as necessidades do projeto. Neste trabalho, foi realizada estimativa de área da lógica implementada no FPGA visando obter um circuito que não ultrapassasse a área máxima permitida para lógica do FPGA. Nesse trabalho fizemos o uso de ferramentas CAD comerciais para obtenção das estimativas. Normalmente, cada ferramenta CAD tem seu próprio ambiente de desenvolvimento de projeto.

3.3.4 Prototipação

Uma vez escolhida a família de dispositivo, o projeto passa para a etapa de prototipação que irá gerar uma descrição para uma determinada arquitetura. Pode-se, ainda nesta fase, fazer o uso dos estimadores para verificar se, após a síntese, para uma determinada arquitetura, o sistema ainda continua satisfazendo as restrições de projeto.

Em seguida, os *bitstreams* (arquivo de configuração do FPGA) são gerados por uma ferramenta CAD adequada, e finalmente o FPGA poderá ser configurado com a lógica implementada pelo projetista. Na verdade, o comportamento descrito em alto nível foi mapeado na arquitetura de FPGA e a plataforma foi configurada para que o *hardware* funcionasse com uma determinda especificação. Finalmente as fases de análise e testes, que são os responsáveis pela extração de informação do protótipo, e teste de funcionalidade, respectivamente.

Ordonez [2] sugere várias aplicações com a utilização de FPGAs da Xilinx baseados na plataforma da Xess Corporation [6]. As principais etapas serão consideradas para o desenvolvimento de nosso protótipo, sendo limitadas pela disponibilidade da ferramenta CAD que será utilizada.

3.3.5 Validação e Análises

A validação da nossa aplicação deu-se através de simulações, ou seja, usamos uma ferramenta CAD, que dá suporte a simulações e montamos todas as situações desejadas, a fim de obter as respostas esperadas.

As análises serão baseadas em estudo de caso, destacando o desempenho do sistema e comparando algumas abordagens de desenvolvimento de sistemas digitais.

Capítulo 4

Implementação

4.1 Visão geral do sistema

Consideramos na implementação do sistema, uma aplicação, ilustrada na Figura 10, que é composta basicamente de *host*, comunicação RF e controle de robô via plataforma reconfigurável. O robô é dotado de uma câmera, que capta imagens do ambiente, e as envia para a plataforma reconfigurável. Esta aplica um algoritmo de compressão de imagens, nas imagens fornecidadas pela câmera. Em seguida, a imagem compactada é enviada via RF

A figura mostra o sistema como um todo, mas nosso trabalho é focado basicamente no desenvolvimento de um algoritmo de compactação em hardware para a plataforma reconfigurável destacada na figura.

Uma vantagem de se usar *hardware* reconfigurável é que o sistema pode ser reconfigurado em tempo de execução, bastando que algum circuito externo o configure no instante correto. Neste caso, deve-se analisar o custo de reconfiguração do *hardware*, pois, o impacto pode ser relevante quando a quantidade de reconfigurações, para uma determinada aplicação, for muito alta.

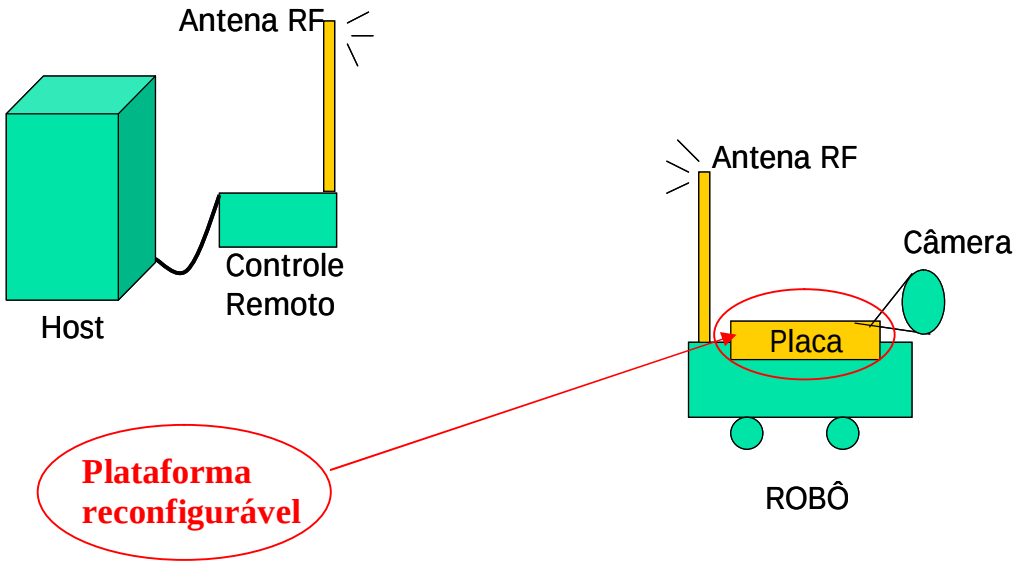

Figura 10. Sistema de captação de imagens através de um robô

Esse trabalho não cobre todas as operações demonstradas na Figura 10 acima, apenas o módulo em destaque, composto pelo FPGA e circuitos acessórios. Esse sistema pode ser estudado para futuros trabalhos.

4.2 Visão geral do Módulo Desenvolvido (RLE)

Nosso sistema apresenta dois módulos internos descritos em VHDL (Apêndice A) que serão apresentados adiante. Nessa sessão mostramos uma visão geral do sistema RLE, destacando a ligação entre os sinais de entrada e saída dos módulos e do sistema RLE como um todo.

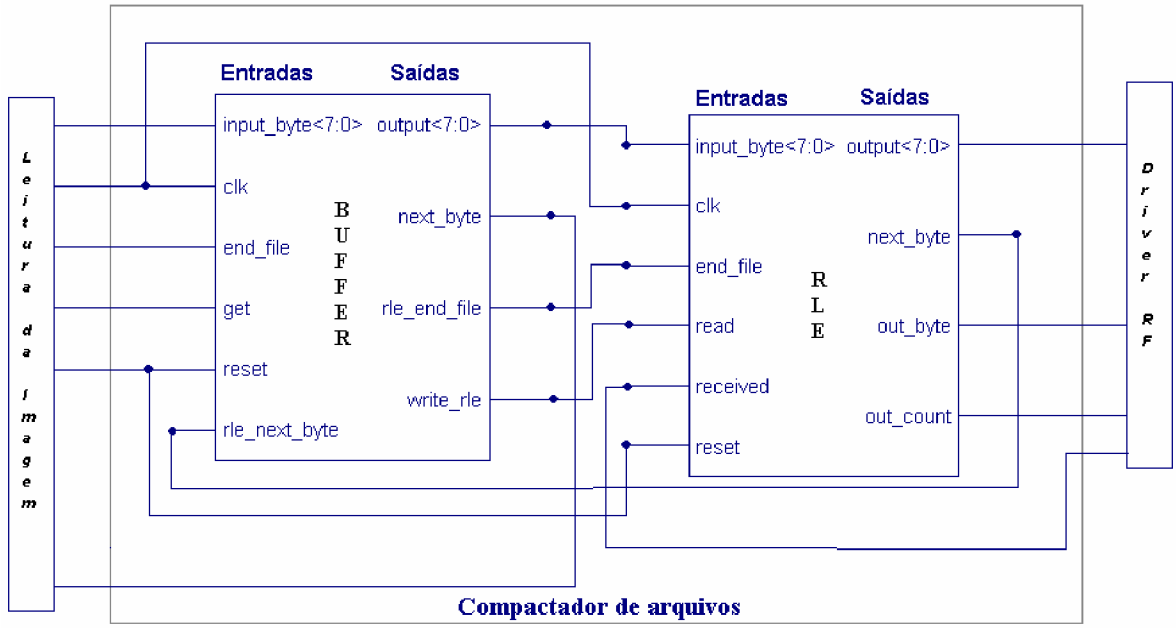

Figura 11. Visão geral do sistema composto pelos módulos *Buffer* e RLE

A partir da Figura 11 podemos que o módulo *Buffer* comunica-se com o módulo de leitura de imagem e o módulo de comunicação RF. O módulo de leitura de imagens está ligado à câmera e recebe as imagens capturadas por ela. Esse módulo envia os bytes para o módulo *buffer*. O *Driver RF* é responsável por enviar os *bytes* compactados via RF.

O *Buffer* apresenta uma entrada de 1 *byte* (*input_byte*), por onde os *bytes* são recebidos vindos do módulo de leitura de imagens. Outros sinais de comunicação serão detalhados nas próximas seções. Através desses sinais pode-se implementar todo o protocolo de comunicação entre os módulos.

4.3 Algoritmo RLE

O algoritmo de codificação RLE (*Run-Length Enco*ding) comprime/codifica dados baseados em ocorrências sucessivas do mesmo padrão. O Resultado da codificação de uma *string* de *bytes* é sempre uma palavra composta por dois *bytes*: um que indica o padrão de repetição e o outro *byte* que contém o número de vezes que deverá ser repetido esse padrão.

Daqui podemos concluir facilmente que este algoritmo só é aplicável para determinados tipos de informação: se não houver padrões repetidos então em vez de compressão de dados teremos expansão, uma vez que não havendo repetições vamos ter cada *byte* a ser codificado num padrão diferente, ou seja, vamos ter uma imagem com o *dobro* da informação.

Um exemplo típico de uma codificação RLE pode ser representado no seguinte esquema: considere a seqüência inicial “AABBBCCCC”, ela apresenta repetições de alguns *bytes*, por exemplo, o *byte* ‘A’ aparece duas vezes nessa seqüência, então para representarmos isso de forma codificada, usaremos dois *bytes*, um representará o *byte* da seqüência, o outro a quantidade de repetições do primeiro. Portanto para o *byte* ‘A’ repetindo-se duas vezes temos 2A.
Codificando toda a seqüência temos 2A3B4C, ou ainda podemos dividir em dois arrays de *bytes*

um com a quantidade de repetições e outro com os *byte*s apresentados na seqüência, ou seja, em um array temos 234 e no outro ABC.

Para esta seqüência a taxa de compressão é de 33% , pois houve uma redução de 9 *byte*s para 6 *byte*s.

4.4 Inicialização

Para que o sistema seja inicializado, deve-se gerar um pulso na entrada *reset*. Com isso, todos os módulos estarão prontos para iniciar suas atividades de acordo com os sinais de controle.

A partir daí, os sinais de controle podem ser gerados pelo módulo externo e recebidos pelo sistema.

4.5 Funcionamento do sistema

Inicialmente, os dois primeiros *bytes* do *stream* retirado do arquivo a ser compactado são lidos pelo módulo *buffer* e encaminhados para o módulo RLE, onde são comparados. Sendo eles padrões repetidos, o contador é incrementado e o *buffer* de entrada solicita o próximo *byte* para ser lido. Se os *bytes* comparados não forem iguais ou se o contador ultrapassar o seu valor máximo, o *byte* anteriormente lido é colocado na saída, seguido da sua quantidade de repetições. O valor do contador passa a ser 1 e o novo *byte* será comparado com um próximo que será buscado.

O diagrama a seguir reproduz a estrutura do algoritmo do RLE através de diagrama de blocos:

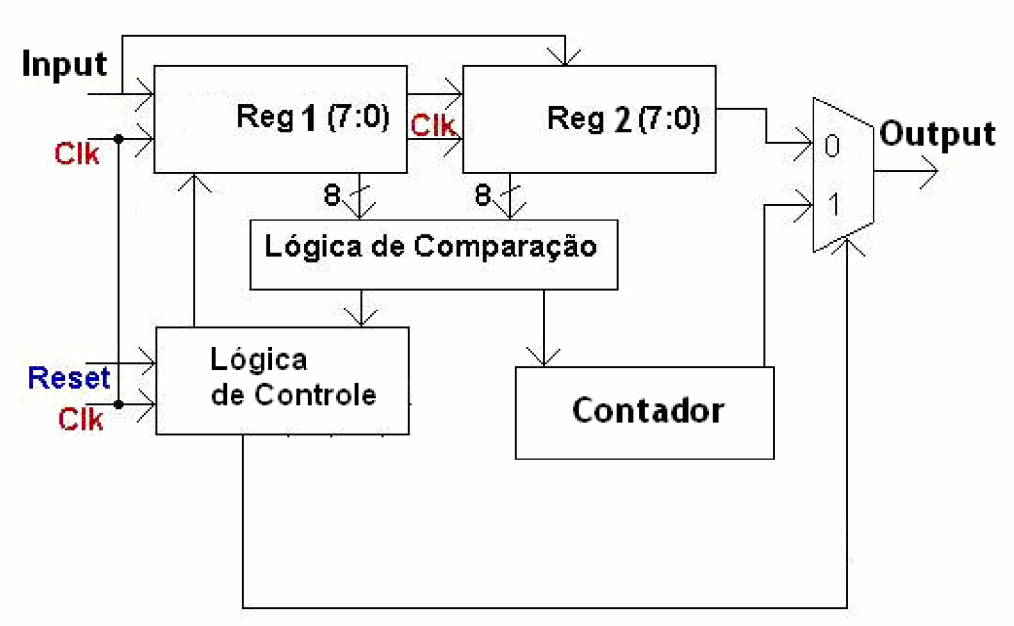

Figura 12. Diagrama de blocos do algoritmo.

4.5.1 Módulo Buffer de entrada

Esse módulo é responsável por receber os dados enviados pelo módulo externo, cujos pinos estão mapeados em pinos do FPGA. Essa comunicação é estabelecida seguindo um protocolo de comunicação que foi idealizado para garantir sincronização entre os módulos e a interface externa.

Considere a interface externa como sendo um *driver* para comunicação que pode ser PCI, ISA, serial, paralela, dentre outros. Em uma primeira implementação de nossa abordagem, o módulo Buffer foi pensado para ser o *driver* de comunicação para um barramento ISA, tendo em vista que foi montado uma plataforma baseada em FPGA que poderia ser conectado a um barramento ISA. Mas, não foi possível continuar com a idéia, tendo em vista que o fabricante do FPGA não fornecia mais suporte para o dispositivo que estávamos trabalhando. Então, o módulo Buffer foi adaptado para se comunicar através de um protocolo específico, não considerando nenhum tipo de barramento de comunicação.

Considerando um FPGA da Altera, da família *flex 10k*, a configuração desse módulo ocupou 54 células lógicas de 576 existentes e 18 pinos de entrada/saída de 53 existentes. Esses dados foram retirados do relatório que foi gerado na compilação do projeto (*Compiler Report File*). A Figura 13 ilustra o protocolo de comunicação implementado que é usado entre o módulo Buffer e a interface externa.

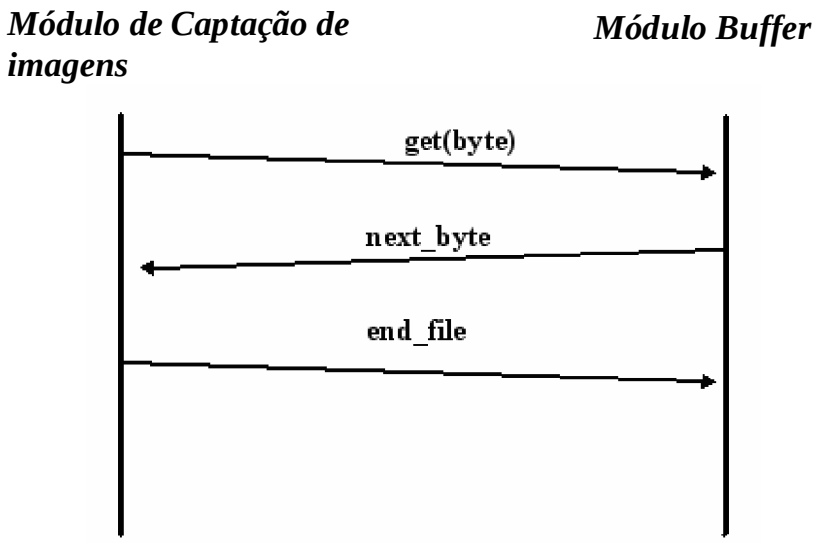

Figura 13. Protocolo de comunicação entre o módulo de captação de imagens e o *Buffer*.

Outros sinais do módulo Buffer são usados internamente, por isso não são mostrados no protocolo de comunicação mostrado na Figura 13. A seguir será indicada a lista de sinais de controle usados em nossa abordagem proposta com suas respectivas funcionalidades.

Sinais de controle

- get: Este sinal informa ao módulo *Buffer* que o próximo *byte* está disponível para ser lido na entrada.
- end_file: Sinaliza ao *Buffer* o fim da seqüência de *byte*s, ou seja, indica que o *buffer* deve

finalizar suas operações.

- reset: prepara o módulo para o início das operações, colocando os devidos valores nas variáveis internas do módulo.
- rle_next_byte: É um sinal interno de comunicação entre os módulos *Buffer* e RLE. Ele indica que o módulo RLE está pronto para receber o próximo *byte* a ser codificado.
- next_byte: Indica ao módulo externo que o *Buffer* está pronto para receber o próximo *byte*.
- write_rle: Sinal interno de comunicação entre os módulos, informa ao módulo RLE que o próximo *byte* está disponível no *Buffer*.
- rle_end_file: Sinaliza ao RLE o fim das operações.

Como dito anteriormente, a implementação desse módulo (*Buffer*) foi baseada no conceito de máquina de estados que é melhor ilustrado através da Figura 14. A funcionalidade de cada estado, da máquina de estados usada neste módulo também é apresentada a seguir.

- **Estado A**: Prepara a máquina de estados para iniciar a operação. Nesse estado, o sinal de controle end_file é verificado, caso esteja em NLA, a máquina segue para o **estado E**, onde finaliza suas operações, do contrário segue para o **estado B**.
- **Estado B**: Nesse estado, o *buffer* aguarda o envio do próximo *byte*. Aqui, o sinal de controle end_file é novamente verificado, caso esteja em NLA, o sinal write_rle é colocado para NLB e a saída é zerada, em seguida, a máquina vai para o **estado E**, onde finaliza suas operações. Estando o sinal end_file em NLB, e o sinal de controle get em NLA, o valor da entrada é lida e armazenada numa variável interna, o sinal write_rle é zerado e a máquina segue para o **estado C**. Caso o sinal de controle get esteja em NLB, ela permanece nesse estado.
- **Estado C**: Esse estado verifica se o próximo módulo (Módulo RLE) está solicitando o próximo *byte*, caso o sinal de controle rle_next_byte estiver em NLA, o *byte* armazenado no estado anterior é posto na saída e os sinais next_byte e write_rle são colocados em NLB e NLA, respectivamente, e a máquina de estados segue para o **estado D**. Caso contrario a máquina permanece no estado corrente.
- **Estado D**: O estado **D** serve para que o *Buffer* certifique-se de que o módulo RLE recebeu o byte enviado no estado **C.** Caso o sinal rle_next_byte esteja em NLB a máquina permanece nesse estado, caso contrário, o sinal de controle next_byte é posto em NLA, o sinal write_rle é colocado em NLB e a máquina volta para o estado é o **B**.
- **Estado E**: Estado final da máquina de estados; o sinal rle_end_file é colocado em NLA e máquina permanece nesse estado até que o sinal de reset esteja em NLA.

A figura a seguir mostra o diagrama da máquina de estados do módulo *Buffer*:

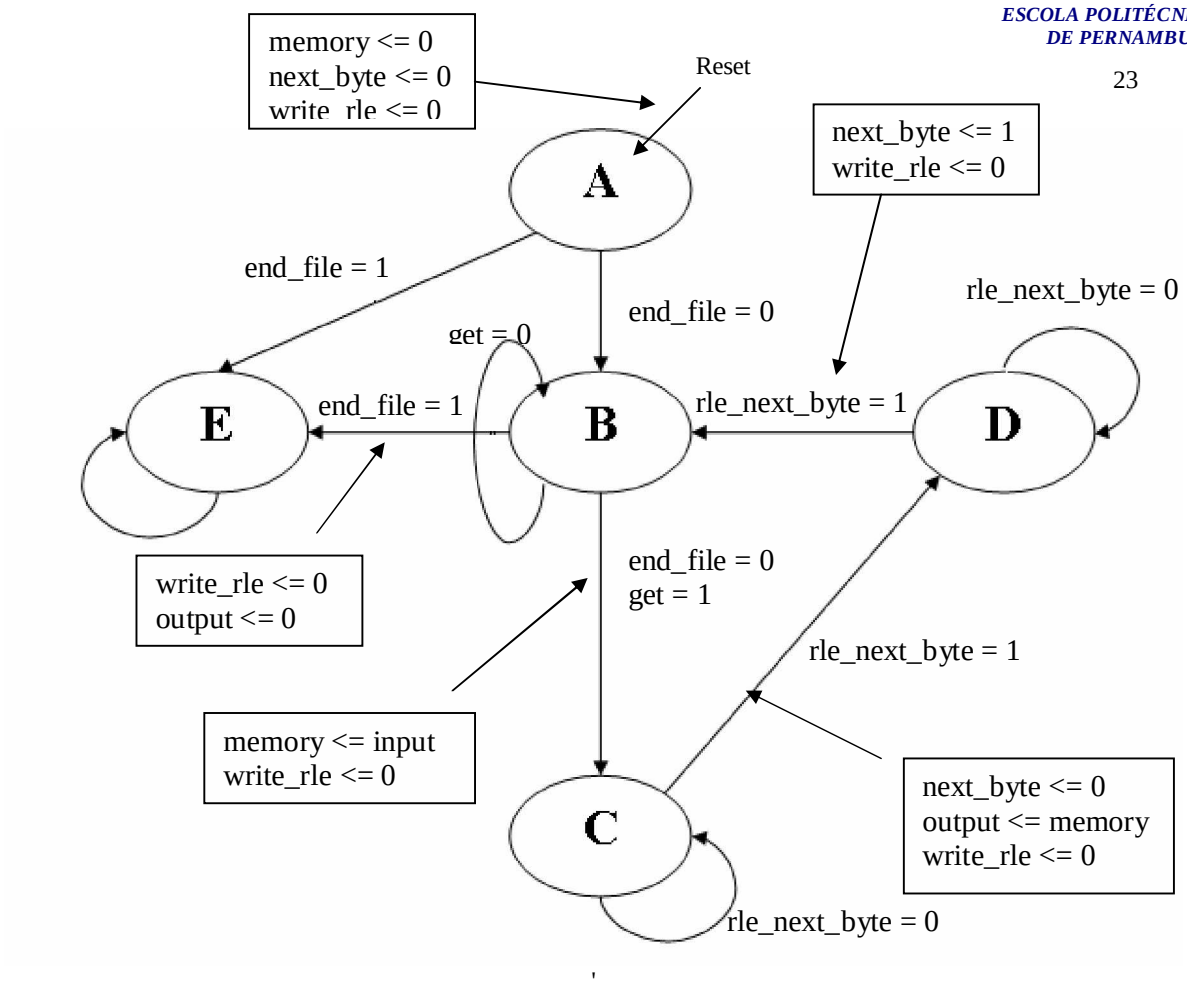

Figura 14. Diagrama de estados do *Buffer*

Na Figura 14, o símbolo "<=" significa que o sinal está recebendo o valor especificado após o símbolo.

4.5.2 Módulo RLE

Este módulo é responsável por executar o algoritmo de compactação RLE, além de se comunicar com o módulo externo, enviando os dados compactados. A Figura 15 ilustra os sinais de controle que são usados na comunicação entre o módulos Buffer e o RLE.

O FPGA da família *flex 10k (Altera)*, possui 576 células lógicas e 53 pinos de entrada/saída. A ferramenta CAD usada, fornece uma estimativa de quantas células lógicas pinos de entrada e saída foram usados para implementar a funcionalidade desejada. Então a configuração desse módulo ocupou 149 células lógicas e 18 pinos de entrada/saída.

Sinais de controle entre os módulos, *Buffer* de entrada e RLE

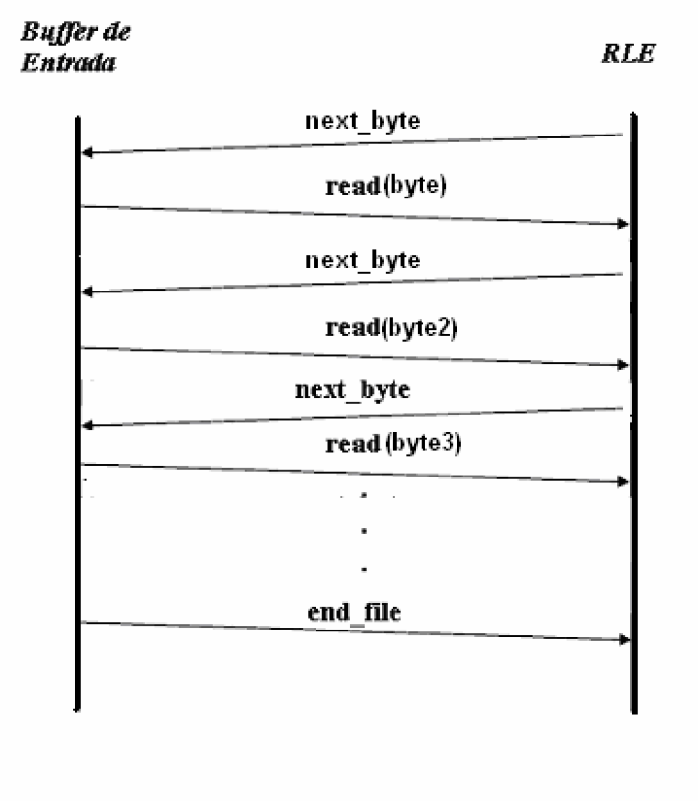

Figura 15. Sinais entre *Buffer* e RLE

Sinais de controle

- read: É um sinal de entrada que diz ao módulo RLE que o próximo *byte* já pode ser lido do *buffer* de entrada.

Os outros sinais de controle foram explicados na sessão sobre o módulo Buffer. A Figura 16 mostra a interface entre o módulo RLE e o Módulo *Buffer*, através dos sinais de controle usados.

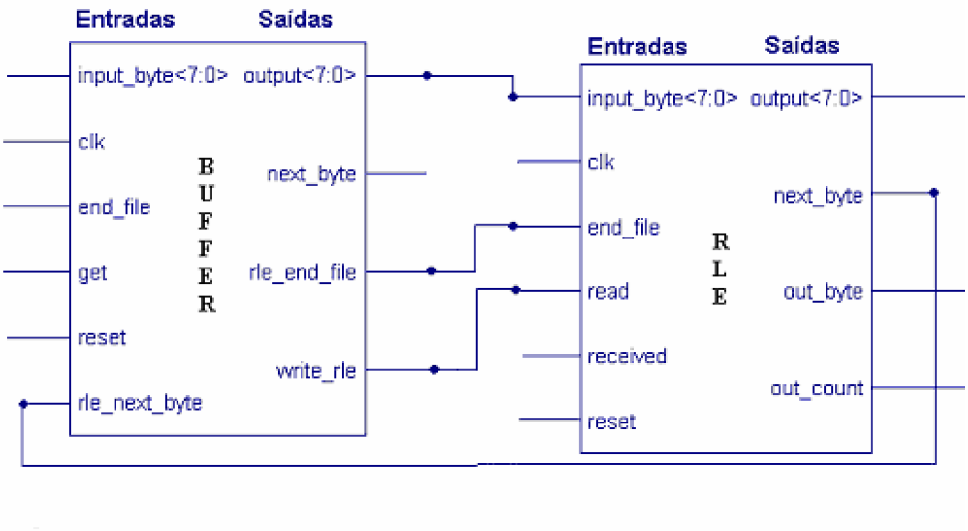

Figura 16. Sinais de interface

Como no módulo *Buffer*, o módulo RLE também é baseado no conceito de máquina de estados e apresenta os seguintes estados descritos a seguir e ilustrados na Figura 18.

- **Estado A**: Simplesmente encaminha a máquina de estados ao próximo estado, que será o **estado B**;
- **Estado B**: Esse estado inicializa os valores dos sinais internos. Nele, o valor do contador é alterado, recebendo o valor 1, isso quer dizer que o próximo *byte* terá o valor mínimo de ocorrência obviamente igual a 1. Daqui a máquina passa para o **estado C**;
- **Estado C**: Nesse estado o próximo *byte* é lido. Os sinais out_byte e out_count são colocados em NLB, sinalizando que não há dado na saída ainda. Então o valor do sinal 'end_file' é testado, caso esteja em NLA, a máquina seta o valor da variável 'last', que serve para indicar que os últimos valores (valores do *byte* do arquivo e da contagem) devem ser colocados na saída, e a máquina deve seguir para o seu estados final. Caso o 'end_file' esteja em NLB e sinal 'read' esteja em NLA, o segundo *byte* é lido pelo módulo, pois o primeiro já foi lido no **estado G**. Além de ler o segundo *byte*, o sinal next_*byte* é colocado em NLB, isso por que há um *delay* na leitura do *byte* enviado pelo módulo *Buffer*. Passado esse delay, o sinal será colocado em NLA novamente (isso será feito em outro estado).
  Após o *byte* ter sido lido, a máquina de estados segue para o **estado H**. Caso nenhuma das situações citadas anteriormente aconteça, a máquina permanece no **estado C**.
- **Estado D**: Para que não aconteça *overflow* e a contagem de *byte*s siga errada, nesse estado o contador é verificado. Caso seu valor seja maior que 255 (valor máximo para 8 *bits*), o sinal de *overflow* é colocado em NLA e a máquina de estados segue para o **estado J**. Caso não aconteça o overflow, o sinal next_*byte* é colocado em NLA para sinalizar que o módulo *Buffer* já pode disponibilizar o próximo *byte*; então a máquina de estados volta ao **estado C**.
- **Estado E**: Nesse estado o sinal out_byte é colocado em NLA, indicando que será colocado na saída o *byte* original do arquivo, pois o resultado desse algoritmo são dois vetores de *byte*s, como foi dito anteriormente, o primeiro com as amostras (*byte*s do arquivo original), e o segundo com a quantidade de repetições consecutivas daqueles

*byte*s. Então nesse estado o primeiro *byte* é colocado na saída e a máquina permanece nesse estado até que o sinal 'received' esteja em NLA. Então o novo *byte* é inserido no reg1 (ver figura diagrama de blocos), ou seja, será comparado com o próximo enviado pelo módulo *Buffer*. Feito isso, a máquina de segue para o **estado F**.

- **Estado F**: Como dito anteriormente, dois vetores de *byte*s devem ser colocados na saída, os *byte*s do vetor de amostras ao colocados na saída no estado E, e os *byte*s de contagem são colocados nesse estado. Esse estado também é responsável por colocar os sinais next_*byte* e out_count em NLA, sinalizando ao *buffer* para enviar o próximo *byte* e que o *byte* que está na saída é um *byte* indicando a quantidade de repetições do primeiro. A máquina permanece nesse estado até que o sinal received esteja em NLA.
  Daqui, então, a máquina de estados segue para o estado J, ou segue para o estado I se a variável last estiver setada.
- **Estado G**: O primeiro *byte* do arquivo é lido e inserido no registrador 1 (reg1) para ser posteriormente comparado com os *byte*s seguintes. Nesse estado também o sinal next_*byte* é colocado em NLA. Então a máquina de estados segue para o **estado C**.
- **Estado I**: Estado final da máquina de estados; a máquina permanece nesse estado até que o sinal de reset esteja em NLA.
- **Estado J**: O contador recebe o valor 1, e a máquina segue para o **estado C**.

A figura abaixo mostra a interface entre o sistema RLE completo visto na Figura 11 e o módulo externo, através dos sinais de controles usados:

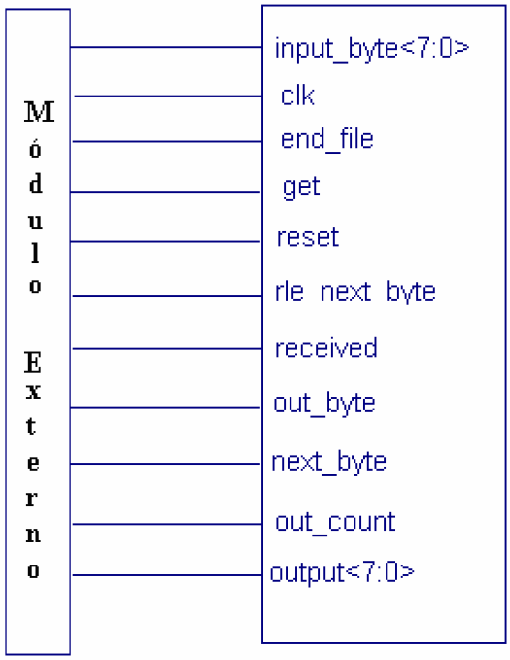

Figura 17. Sinais de interface com o módulo externo

Para um melhor entendimento da máquina de estados, seu diagrama de estados é mostrado na Figura 18.

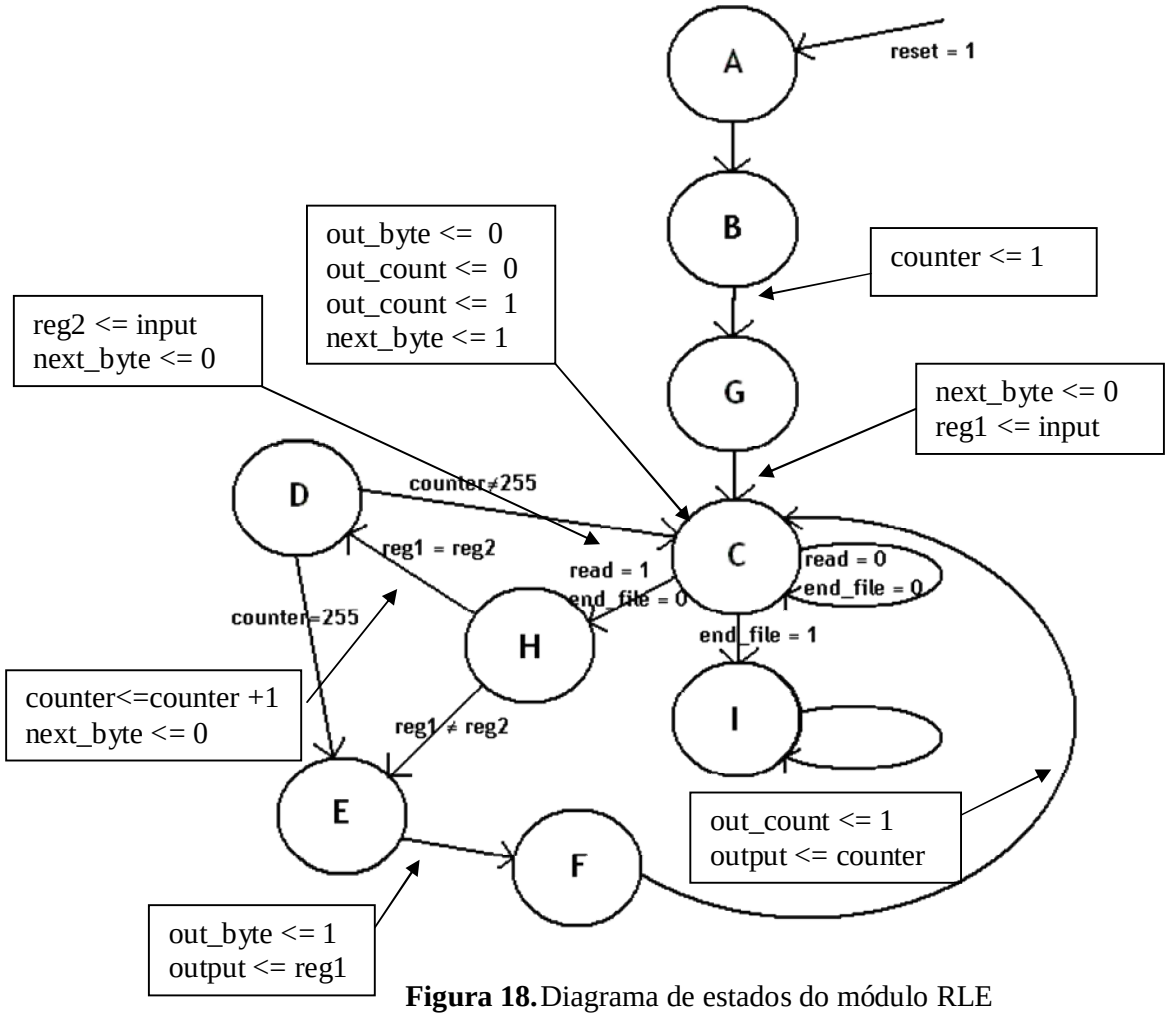

Figura 18. Diagrama de estados do módulo RLE

Na Figura 18, o símbolo "<=" significa que o sinal está recebendo o valor especificado após o símbolo.

4.6 Validação

O algoritmo desenvolvido foi validado via simulação. O conjunto de ferramentas utilizadas para simulação do comportamento dos sinais no FPGA foi o MAX+PLUS II versão 10.2, da Altera Corporation. Os códigos VHDL podem ser vistos no Apêndice A.

Com esta ferramenta foi possível compilar o arquivo .vhd (descrição comportamental), e através da ferramenta Waveform Editor, disponibilizada pelo MAX+PLUS, foram retirados os valores para os sinais ao longo do tempo, como mostrado na sessão seguinte.

4.6.1 Módulo Buffer de entrada

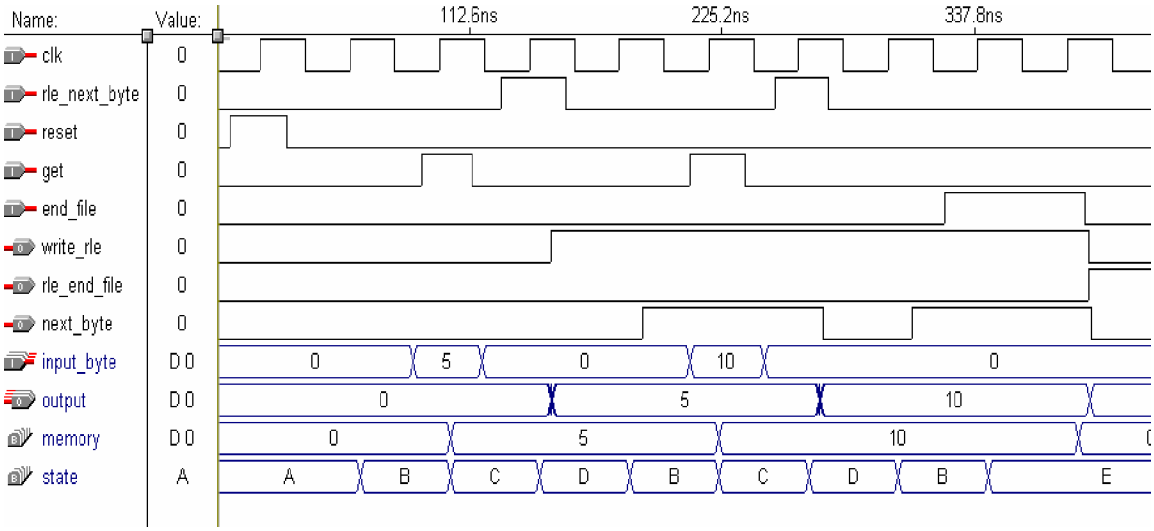

Figura 19. Sinais do módulo *Buffer*

Os sinais gerados pelo módulo *Buffer* de entrada estão mostrados na Figura 19. Sua funcionalidade é receber os *byte*s do arquivo a ser compactado e passá-los para o módulo RLE, que irá codificá-los.

A seguir, todos os sinais anteriormente explicados foram validados por simulação e são mostrados através das formas de onda extraídas através da ferramenta CAD utilizada. Para melhorar o entendimento do andamento da máquina de estados alguns sinais interno de controle, tais como, memory, onde o *byte* é temporariamente armazenado e o state, que guarda os estados da máquina de estados são também ilustrados nas formas de onda.

Reset

Para que o sistema inicie suas operações, é necessário que os módulos recebam um sinal de Reset, que faz com que todas as variáveis internas do sistema ganhem valores esperados. A Figura 20 mostra o inicio da operação do *buffer*, destacando o pulso de reset.

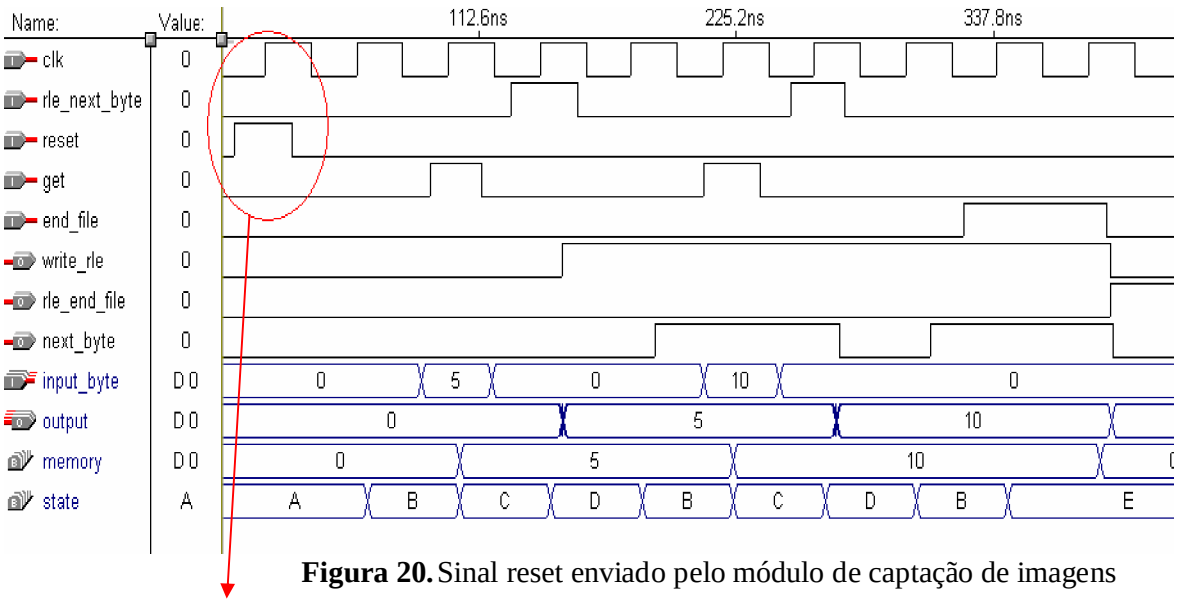

Figura 20. Sinal reset enviado pelo módulo de captação de imagens

Observemos que a partir do pulso de reset, a máquina de estados começa a transitar entre os estados, realizando, assim, sua tarefa.

Embora o pulso de reset apareça durante a transição de subida do pulso de *clock*, não significa que o reset é apenas permitido nessa situação. O reset pode ser percebido tanto na transição de subida quanto na de descida do pulso de *clock* (clk).

Sinal 'Get'

Após o reset, a máquina está pronta para iniciar suas operações. Então a máquina de estados permanece no estado B até que o controle externo envie um sinal 'get' indicando que já existe disponível um *byte* na entrada do *buffer* para ser armazenado pelo mesmo.

A figura abaixo destaca o sinal 'get' , com pulsos ao longo do tempo, e mostra o *byte* na entrada e posteriormente, o *byte* contido na memória do *buffer*.

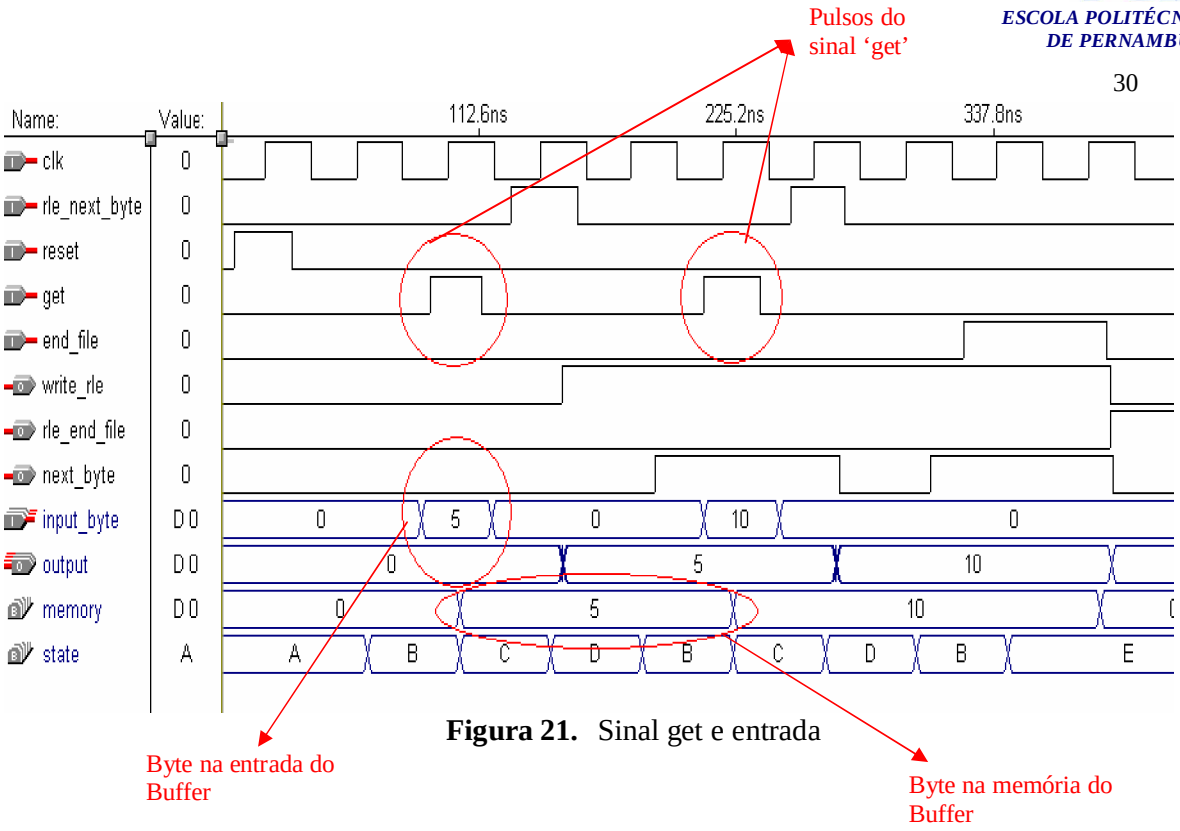

Figura 21. Sinal get e entrada

O sinal 'get', diferente do sinal de reset, só será sentido na transição de subida do pulso de *clock*. Portanto o *byte* deve ser colocado na entrada do *Buffer* e deve ser gerado um pulso no sinal 'get' para que o *Buffer* armazene exatamente o *byte* solicitado. Então o *Buffer* armazena o *byte* da entrada e a máquina de estados segue para o estado C, como observamos na figura acima.

Sinais rle_next_*byte* e write_rle

Como visto anteriormente, o *Buffer* armazena o *byte* que será passado para o módulo RLE. O sinal rle_next_*byte* é responsável por solicitar ao módulo *Buffer* que coloque em sua saída o *byte* armazenado, pois o módulo RLE já está apto a receber e codificar mais um *byte.*

Caso o *Buffer* não receba um pulso do sinal rls_next_byte, a máquina de estados permanecerá no estado C, ou seja, esperando que o módulo RLE retire o *byte* da memória do *Buffer* e esse possa solicitar outro *byte*.

Nem sempre o *Buffer* pode estar com o *byte* disponível para o módulo RLE, por isso o sinal write_rle foi criado, para indicar ao módulo RLE que o *Buffer* já dispõe do *byte* em sua memória, e pode colocá-lo em sua saída.

Na figura a seguir, observamos estes sinais em destaque, além da saída apresentando o *byte* que estava contido na memória do *Buffer*.

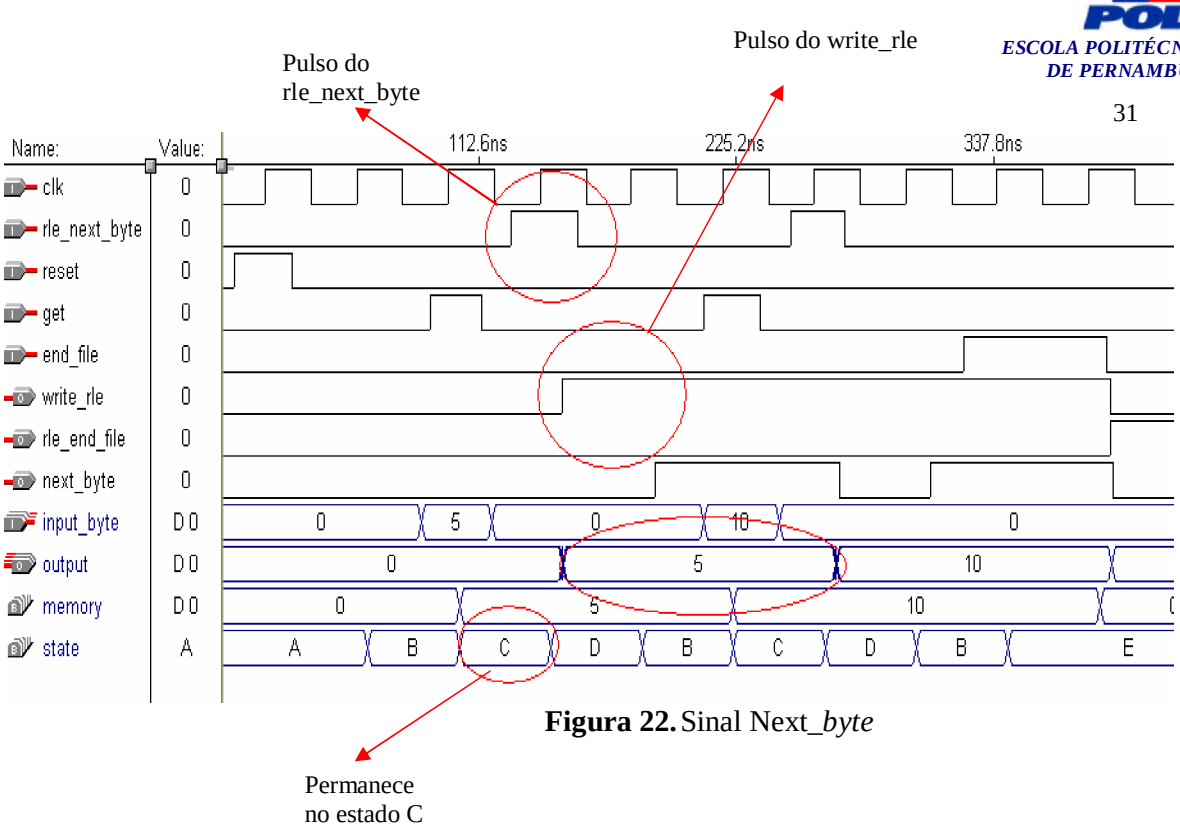

Figura 22. Sinal Next_*byte*

Após o módulo RLE ter retirado o *byte* da saída do *Buffer*, a máquina segue para o estado D, que coloca o sinal 'next_*byte*' em NLA indicando que o *Buffer* já pode receber o próximo *byte* para armazenar. Daqui a máquina segue para o estado B, onde permanece esperando um sinal 'get' ou um 'end_file' indicando que não há *byte*s a serem armazenados. Caso chegue um pulso 'end_file' a máquina segue para o estado E, onde permanece até que um sinal de reset seja enviado.

4.6.2 Módulo RLE

A figura abaixo mostra os sinais internos e externos do módulo RLE, que, como já foi dito, é responsável por receber os *byte*s enviados pelo módulo *Buffer*, compactá-los e enviá-los para saída.

Os sinais de interface, ou seja, entradas e saídas desse módulo são:

Entradas:

Clk (*clock*), Reset, Read, End_file, Received, Input_byte.

Saídas:

Next_byte, Output, Out_count, Out_byte

Este também apresenta sinais internos que são:

- Reg1
- Reg2
- Counter
- State

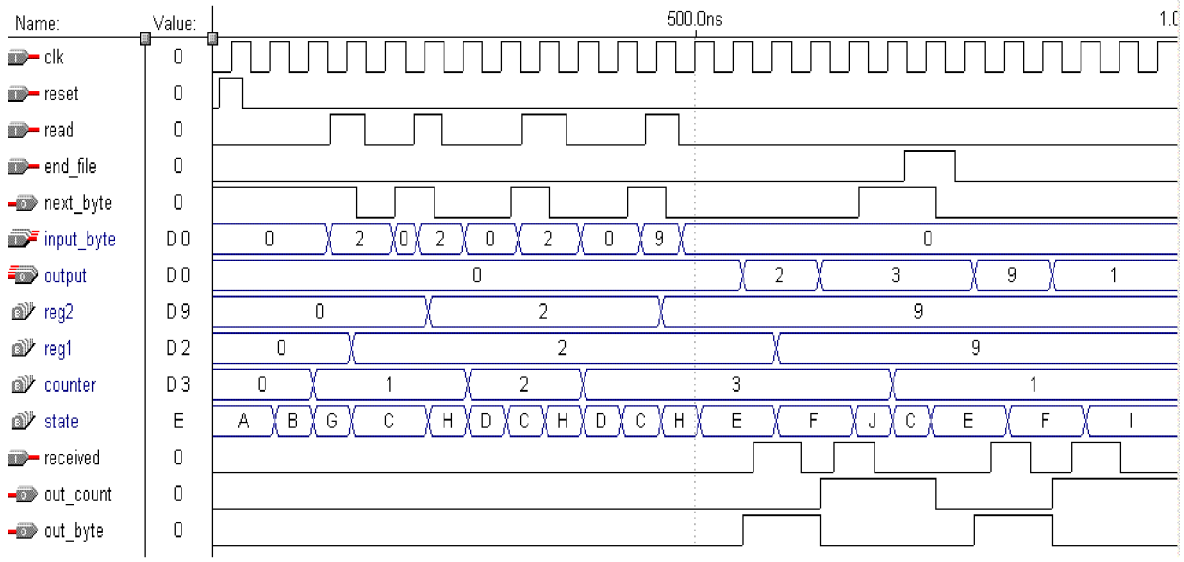

Figura 23. Sinais do RLE

Reset

Como foi dito antes, para que os módulos iniciem suas operações, é preciso que um pulso de reset seja recebido por eles.

A figura abaixo demonstra o início das operações do módulo RLE.

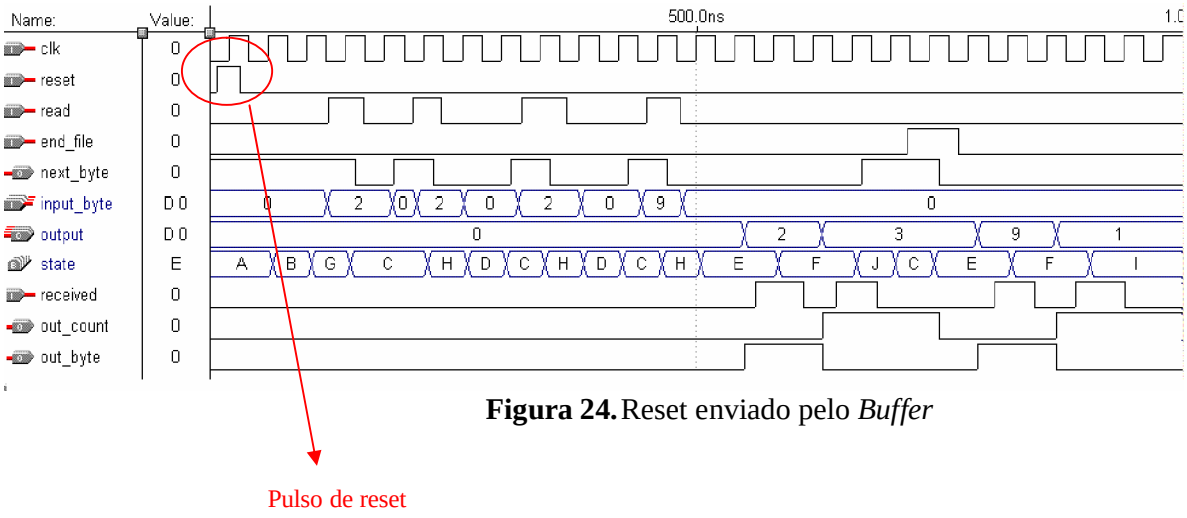

Figura 24. Reset enviado pelo *Buffer*

Da mesma forma que no módulo *Buffer*, o reset também pode ser percebido pelo sistema, tanto na transição de subida quanto na de descida do sinal de *clock*.

Sinal Read

Para que o módulo RLE leia o *byte* que se encontra na saída do *Buffer* (entrada do módulo RLE), é necessário que o módulo *Buffer* envie um sinal indicando que tal *byte* está disponível. Isso é

feito através do sinal ‘read’, ou seja, quando o sinal ‘read’ estiver em NLA, o módulo RLE lerá de sua entrada o *byte* a ser codificado.

A figura a seguir mostra o sinal ‘read’ (pulsos ao longo do tempo) e os valores da entrada do módulo.

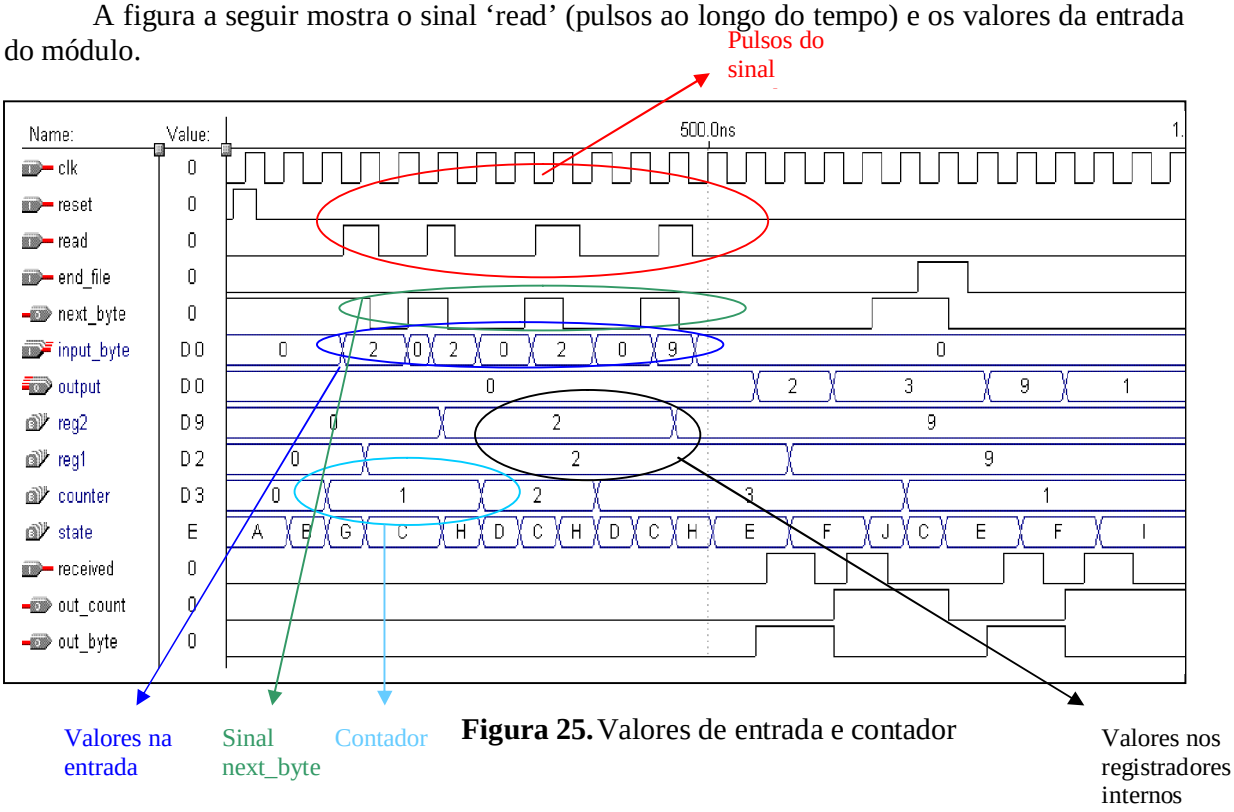

Figura 25. Valores de entrada e contador

Podemos observar também, que os valores da entrada são armazenados em registradores internos, e o valor do contador ao longo do tempo é colocado em 1, quando o padrão de repetição cessar. E quando isso acontece o *byte* é colocado na saída seguido pela quantidade de repetições, como pode ser visto na próxima sessão. É importante observarmos também, através da Figura 25, o comportamento do sinal next_*byte*. Quando o módulo RLE não está pronto para receber outro *byte*, esse sinal permanece em NLB, então vemos na figura que, logo após o módulo receber um pulso do sinal read, o sinal next_*byte* vai para NLB, sinalizando que o módulo está manipulando o *byte* colocado em sua entrada (imput_byte).

Sinal input_byte e output

Na Figura 26, vemos que quando a seqüência de *byte*s iguais e consecutivos é quebrada, o *byte* anterior é colocado na saída, seguido da quantidade de repetição, como proposto pelo algoritmo RLE.

Observamos também que quando o sinal end_file é colocado em NLA, o ultimo *byte* e sua quantidade de repetições são colocados na saída e a máquina de estados vai para o estado I, onde finaliza suas operações, retomando apenas após um pulso do sinal reset.

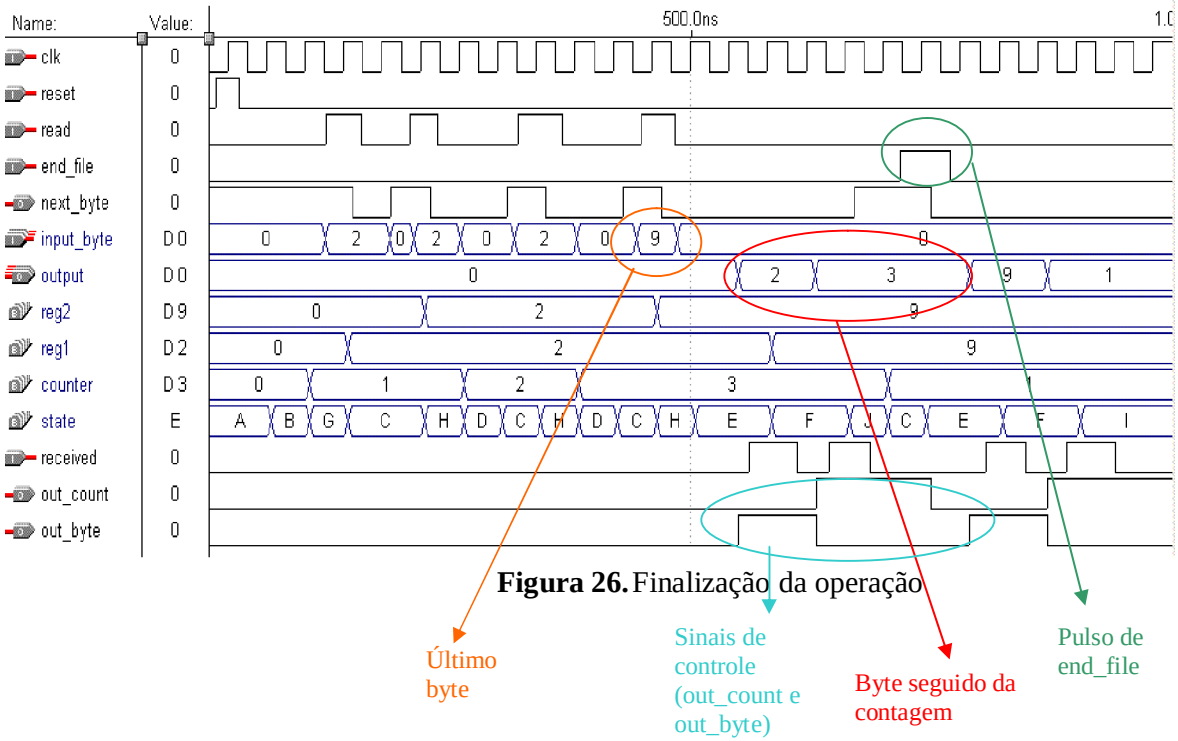

Figura 26. Finalização da operação

Podemos observar também através da Figura 26 que os sinais out_count e out_*byte* indicam o que está na saída naquele instante, ou seja, se o que está na saída for o *byte* codificado, o sinal out_*byte* ficará em NLA, caso seja o *byte* de contagem o sinal out_count ficará em NLA. Assim que o módulo externo recebe um dos *byte*s, esse gera um pulso no sinal received para informar ao módulo RLE que este deve prosseguir sua atividade. Se nada relevante estiver na saída, estes dois sinais permanecem em NLB.

4.6.3 Módulo completo

Mostraremos aqui a integração dos dois módulos apresentados anteriormente. Nessa simulação veremos a troca de sinais entre o módulo *Buffer* e o módulo RLE, além dos sinais entre o sistema e o módulo externo.

A Figura 27 mostra os sinais do módulo completo.

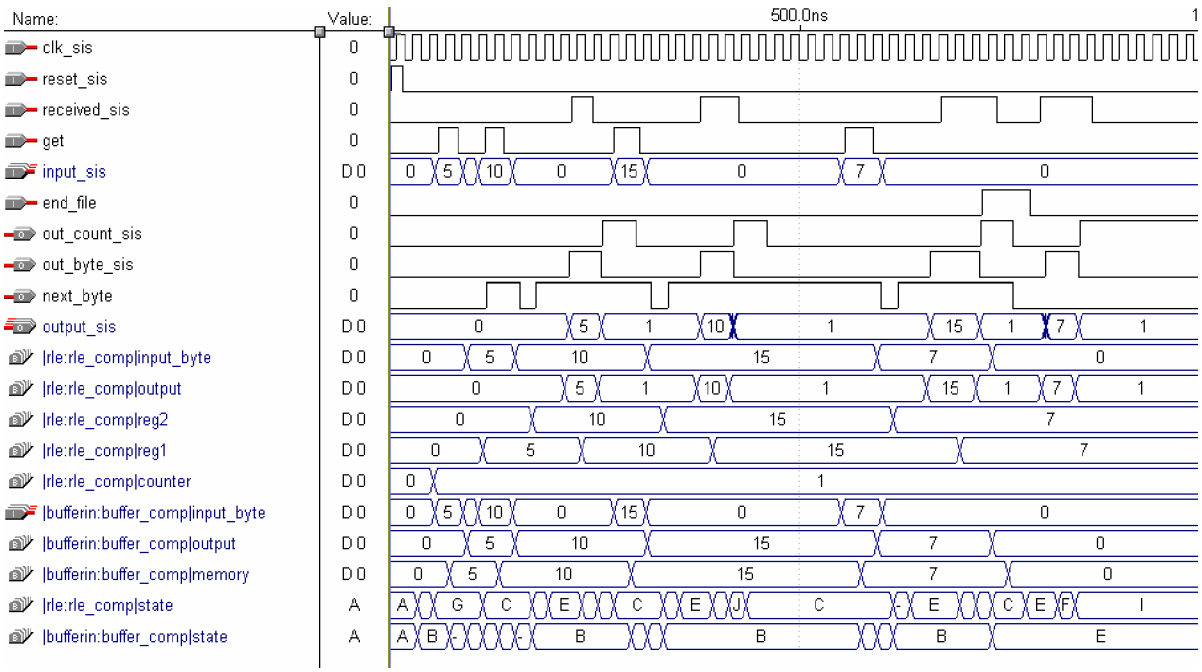

Figura 27. Sinais do Módulo completo

Como mostrado anteriormente o RLE apresenta os sinais:

Entradas:

- clk_sis: *clock* do sistema;
- reset_sis: reset ligado aos dois módulos;
- received_sis: indica ao sistema que o módulo externo recebeu o dado;
- get: indica ao sistema que o próximo *byte* está disponível na entrada (input_sis);
- input_sis: recebe os *byte*s a serem compactados;
- End_file: indica ao sistema o fim do processo.

Saídas:

- out_count_sis: indica que o *byte* que encontra na saída do sistema é um *byte* de contagem;
- out_byte_sis: indica que o *byte* que encontra na saída do sistema é o *byte* codificado;
- next_*byte*: indica ao módulo externo que já pode disponibilizar o próximo *byte* a ser codificado.
- output_sis: saída do sistema, envia tanto o *byte* codificado, quanto sua contagem para o módulo externo.

Faremos agora uma explicação sobre o comportamento do sistema, visualizando os valores dos sinais em cada instante.

Sinais reset e get

Um pulso do sinal reset significa que o sistema deve iniciar suas operações. A figura a seguir

destaca o pulso do reset e os pulsos do sinal get.

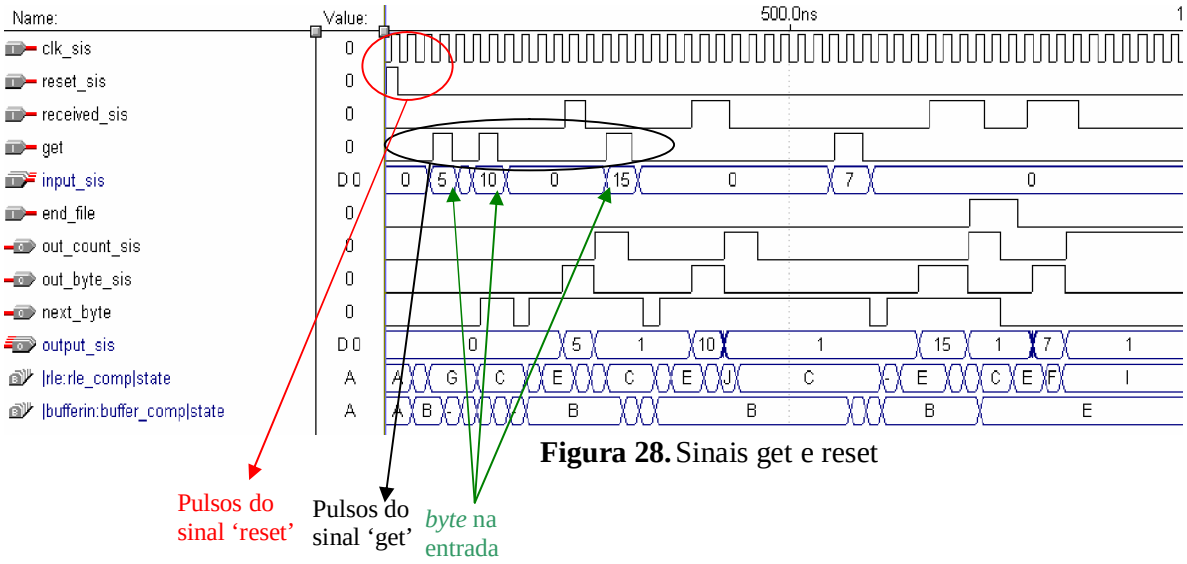

Figura 28. Sinais get e reset

Podemos observar a partir da Figura 28 que, após o pulso de reset, os sinais internos do sistema são zerados, no caso, *counter* e *memory*. Além disso, a máquina de estados passa para o estado A, nos dois módulos.

Também é importante vermos que no momento em que há um pulso do sinal get, um dado válido está disponível na entrada do sistema. Será esse o valor que é dado como entrada para o algoritmo RLE.

Sinais out_count_sis, out_byte_sis e received_sis

Vejamos o comportamento dos sinais out_count_sis, out_byte_sis, que servem para indicar o que está sendo colocado na saída.

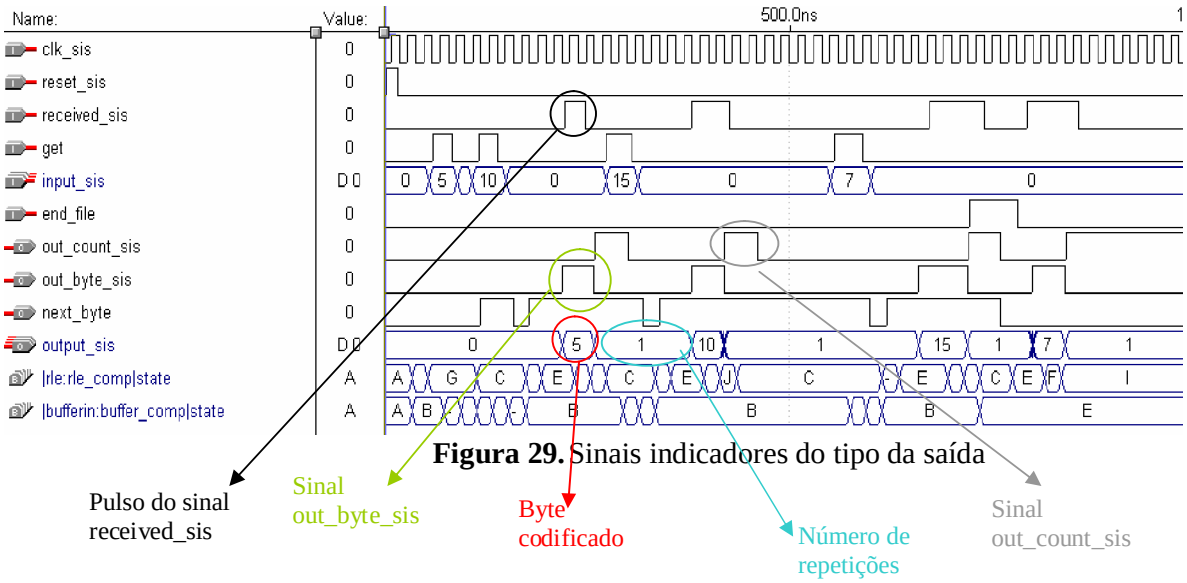

Figura 29. Sinais indicadores do tipo da saída

Podemos então verificar, através da Figura 29, que no instante em que o *byte* codificado está na saída (output_sis) o sinal out_*byte*_sis está em NLA e o out_count_sis em NLB. Já quando é o *byte* de contagem que está na saída, o out_count_sis está em NLA, enquanto o out_*byte*_sis em NLB.

Para que o nosso sistema assegure-se de que o módulo externo recebeu o *byte* codificado, é necessário que o módulo gere um pulso no sinal received_sis.

Sinal end_file

Quando o sistema percebe um pulso desse sinal, ele finaliza suas operações, mas antes disso o ultimo *byte* recebido é colocado na saída, juntamente com a contagem.

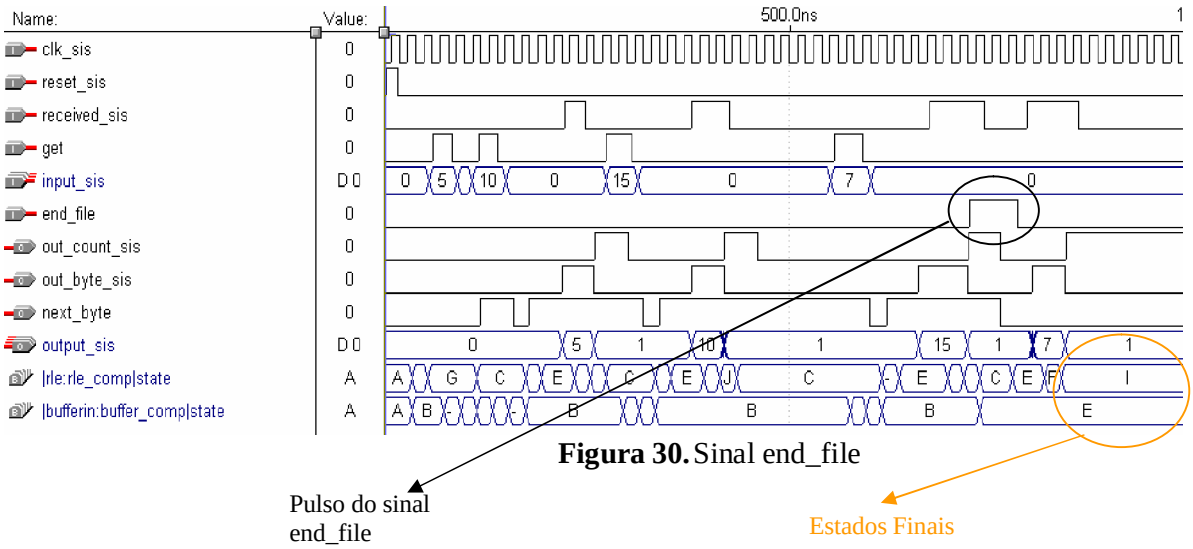

Figura 30. Sinal end_file

Na Figura 30, vemos o pulso do sinal end_file, que será gerado pelo módulo externo. Após esse pulso, os dois módulos vão para seus estados finais – o RLE vai para o estado I e o *buffer* para o estado E.

Capítulo 5

Estudo de Caso

Nesta seção, será descrito um estudo de caso, considerando o algoritmo de compactação RLE implementado em hardware. Consideramos uma pequena imagem ilustrada na Figura 31 que foi usada para análise do algoritmo de compactação. Visando motivar nosso trabalho, comparamos a abordagem implementada em hardware com uma outra abordagem em software, também desenvolvida neste trabalho. Resultados são indicados no decorrer desta seção.

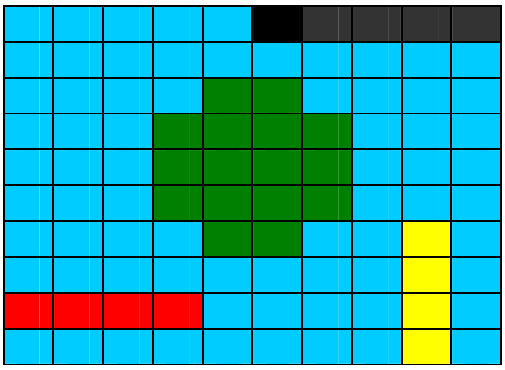

Figura 31. Exemplo de figura

5.1 Dados resultantes da simulação

Mostraremos aqui, alguns dados que podem ser retirados da simulação do algoritmo RLE, tanto para a implementação em *hardware*, quanto para implementação em *software*.

Neste estudo de caso, foi usado para validação do algoritmo de compactação a Figura 31. A figura usada foi manipulada, tendo em vista que o foco de nosso trabalho foi validar a implementação do algoritmo, sem considerar o tratamento de uma imagem real (*.bmp. *.jpg, *.gif, etc). Para um melhor entendimento do algoritmo, a figura foi dividida em células, a fim de possibilitar uma melhor visualização dos seus *bytes.* A dimensão dessa imagem é 10x10.

Considere a figura acima onde a cor de cada célula do *grid* pode ser representada por um *byte*. De acordo com a legenda abaixo, a imagem apresenta os *bytes* mostrados na Tabela 2 .

15	15	15	15	15	**0**	**0**	**0**	**0**	**0**
15	15	15	15	15	15	15	15	15	15
15	15	15	15	20	20	15	15	15	15
15	15	15	20	20	20	20	15	15	15
15	15	15	20	20	20	20	15	15	15
15	15	15	20	20	20	20	15	15	15
15	15	15	15	20	20	15	15	5	15
15	15	15	15	15	15	15	15	5	15
10	10	10	10	15	15	15	15	5	15
15	15	15	15	15	15	15	15	5	15

Tabela 1. Representação fictícia dos *bytes* da imagem

A Tabela 2 abaixo apresenta valores fictícios para as cores apresentadas na Figura 31, ou seja, os valores atribuídos não fazem parte de nenhum tipo de codificação de imagens (Ex. BMP, JPG, GIF, etc).

Cores	Valor
	10
	5
	15
	0
	20

Tabela 2. Valores fictícios para os *bytes*

5.1.1 Resultados da simulação da implementação em hardware

Usamos como base para implementação em *hardware* a família *flex 10K* que apresenta de 10.000 a 250.000 portas [1], o que é suficiente para o projeto, inclusive para colocarmos mais módulos executando em paralelo, como proposto na sessão de conclusão, o que apresentaria um ganho de desempenho.

Através da simulação, pudemos colher dados de desempenho da implementação, ou seja, o total de tempo gasto pelo sistema para compactar a imagem dada como exemplo.

Para efeitos de comparação das abordagens (hardware e software), elas devem ser analisadas sob as mesmas condições de freqüência (44Mhz). Como o simulador da Altera apresenta uma limitação temporal de 1 µs para amostragem da simulação, tivemos que particionar a imagem, e verificar o tempo para cada parte.

A Figura 32 mostra a codificação dos 7 primeiros *bytes* da imagem da Figura 31. Observamos através daquela, que o tempo gasto para que os *bytes* sejam codificados e colocados

na saída (output_sis) é 10,34 µs. Vemos que nesse instante, que o *byte* 0 e sua contagem (2) são colocados sequencialmente na saída, finalizando, assim, a codificação desses primeiros *bytes*.

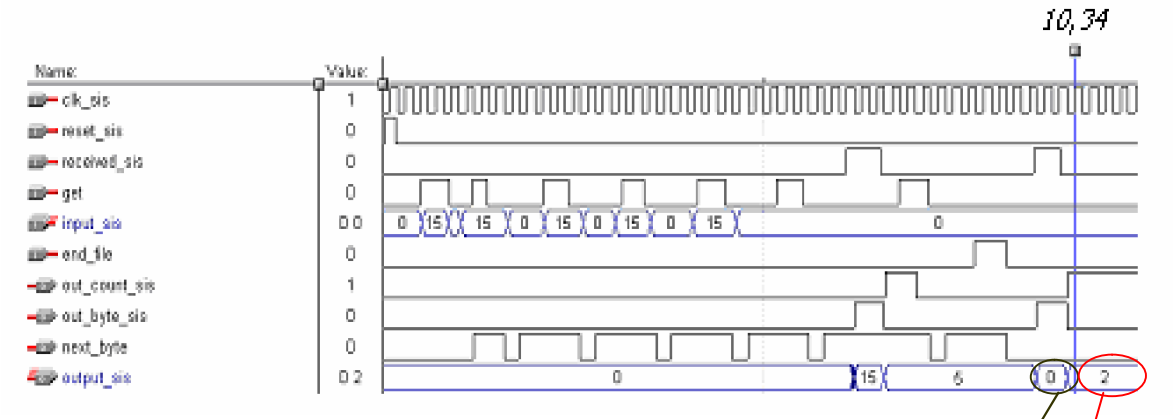

Figura 32. Compactação dos primeiros bytes da imagem

Byte 0, representando a cor preta.

Contagem do *byte* 0

Esse processo não será mostrado através de imagens como feito para os primeiros bytes, tendo em vista que as análises para os outros bytes são similares, porém os tempos para cada fragmento da imagem são mostrados através da tabela seguir.

Fragmentos da imagem	Tempo para codificação (µs)
15 15 15 15 15 0 0	10,34
0 0 0 15 15 15 15	9,66
15 15 15 15 15 15 15	9,89
15 15 20 20 15 15 15	10,57
15 15 15 15 20 20 20	10,11
20 15 15 15 15 15 15	10,11
20 20 20 20 15 15 15	10,11
15 15 15 20 20 20 20	10,11
15 15 15 15 15 15 15	9,89
20 20 15 15 5 15 15	11,25
15 15 15 15 15 15 15	9,88
5 15 10 10 10 10 15	11,02
15 15 15 5 15 15 15	10,34
15 15 15 15 15 15 5 15	11,36
Tempo total	**144,66**

Tabela 3. Tempos parciais de codificação

O tempo total gasto na compactação de todo o arquivo é a soma dos resultados parciais mostrados na Tabela 3. Como podemos ver, o tempo total é 144,66 µs.

5.1.2 Resultados da simulação da implementação em software

Para a simulação da implementação em *software*, os dados foram obtidos utilizando-se o simulador keil µVision v2.40a. Essa foi realizada com base no microcontrolador T80C51 da Atmel que opera a uma freqüência de 44Mhz.

Descrição obtida na base de dispositivos da ferramenta da keil:

- *8051 based CMOS controller* - Controlador CMOS baseado no 8051;
- *32 I/O lines* – 32 pinos de entrada/saída*;*
- *2 Timers/Counters* – 2 temporizadores/contadores*;*
- *5 Interrupts/2 priority levels 4K ROM* – 5 interrupções com 2 níveis de prioridade e ROM de 4K*;*
- *128 Bytes on-chip RAM* – RAM de 128 *Bytes.*

Neste abordagem, como pode ser visto no apêndice B, o software apresenta um *array* que guarda os *bytes* da imagem, portanto a execução foi feita com todo o arquivo de uma só vez, diferentemente da simulação da implementação em *hardware*, que devido a limitações no simulador, tivemos que dividir a imagem em conjuntos de 7 *bytes.* Isso não afeta substancialmente a comparação, porque na simulação da implementação em *hardware*, os tempos para todas as partes foram medidos.

Nessa ferramenta, o tempo de execução é informado, além de outros parâmetros, como pode ser visto na figura a seguir:

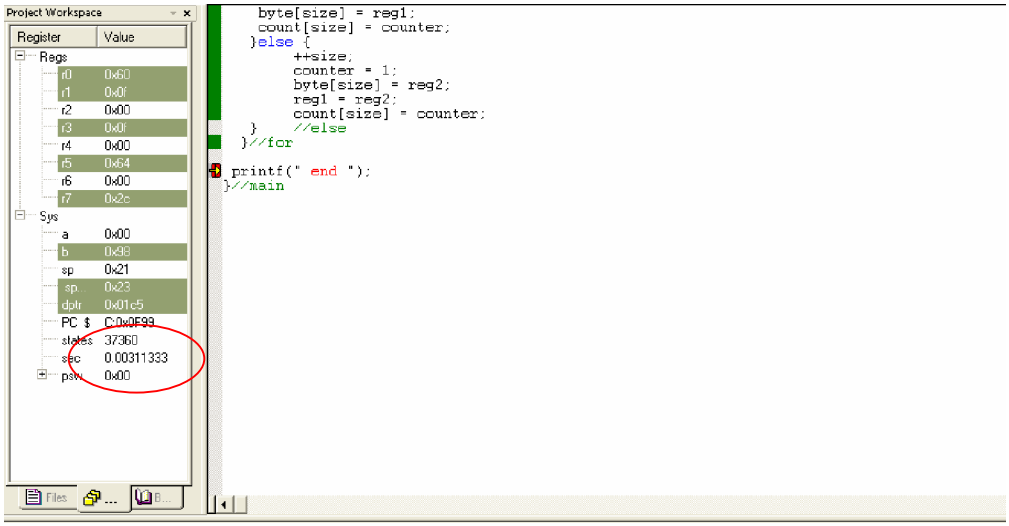

Figura 33. Tempo em segundos - µVision da keil

Usamos também como entrada para a simulação, a imagem mostrada na Figura 31.

	Tempo de Execução	***Espaço Físico Ocupado***	***Nº de Linhas de código***	***Clock (MHz)***	**Preços**
Hardware (FPGA)	144,66 µs	201 logic *Cells* de 576 disponíveis	223	44	R$ 152 (FPGA XC4010E-4)
Software (Microcontrolador)	3,1133 ms	1.105 *byte*s	25	44	R$ 30 (Chip Atmel 89S8252)

Tabela 4. Comparação de resultados encontrados para FPGA e Microcontrolador

Através desses resultados podemos observar que o tempo de execução do algoritmo RLE no FPGA é cerca de 21 vezes menor que o tempo de execução da implementação em software (microcontroladores). Neste caso, isso comprova que a implementação do algoritmo RLE é mais satisfatória, em termos de desempenho, ser implementada em hardware.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho, foi implementado um o algoritmo de compactação de arquivos em *hardware*, usando a linguagem de descrição de *hardware* VHDL.

A escolha de uma implementação em *hardware* pode nos indicar que é possível obter ganhos satisfatórios de desempenho quando comparados com uma abordagem em *software* baseada em microcontroladores.

O algoritmo proposto para ser implementado foi o RLE (Run-length *encode*) que agrupa os *byte*s repetidos consecutivos apresentados por um arquivo. Na implementação desse algoritmo, usamos o conceito de máquina de estados, que facilitou o esforço de implementação. Resultados validados via simulação, mostram que o desempenho da abordagem baseada em FPGA foi cerca de 21 vezes melhor que a solução baseada em microcontroladores 8051 com programação em linguagem C. O simulador da keil apresenta algumas opções de otimização do código C para microcontroladores. Nesse projeto foi usado o maior nível de otimização de código, mesmo assim, a solução baseada em microcontroladores apresenta um desempenho inferior ao da solução em FPGAs.

Uma possível opção para melhorar o desempenho da solução em microcontroladores seria implementar o algoritmo usando a linguagem Assembly, porque esta pode apresentar um nível maior de otimização de código. Contudo, a otimização feita com o código em Assembly pode não tornar o desempenho da solução muito melhor do que o que foi mostrado anteriormente.

O escopo desse trabalho não cobre a transmissão do arquivo através dos módulos externos (captação de imagens e *Driver* RF), porém, o algoritmo reage a comandos e dados externos, isto é, ele está totalmente preparado para comunicação com o módulo externo. Considerando a visão geral do projeto descrito no capítulo 3, de um sistema composto basicamente por robô guiado por comunicação RF (rádio freqüência), *host* e plataforma reconfigurável, propomos como trabalhos futuros: (a) a implementação do circuito de transmissão e recepção de uma comunicação RF; (b) a implementação de um protocolo de comunicação seguro para controle do robô remotamente; (c) Implementação de uma aplicação remota para *web* que possa controlar o robô; (d) Implementação de um circuito de captação de imagem baseada em FPGAs; (e) Estender o trabalho explorando mais o paralelismo do FPGA através da criação de novas instâncias do algoritmo RLE. a fim de melhorar mais ainda o desempenho do sistema. (e) Implementação de outras funcionalidades além de compactação de imagem.

Bibliografia

[1] MENDONÇA, A.; ZELENOVSKY, R. Monte seu protótipo ISA Controlado por FPGA, MZ Editora, 2001

[2] ORDONEZ, E. D. M.; PEREIRA, F. D.; PENTEADO, C. G.; PERICINI, R. de A. Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs), Editora Bless, 2003

[3] Xilinx, Inc site. Disponível em: <www.xilinx.com>. Acesso nov. 2005.

[4] Altera Corporation site. Disponível em: <www.altera.com>. Acesso nov. 2005.

[5] ORDONEZ, Edward David Moreno, SILVA, Jorge Luiz e (ed.). Computação reconfigurável: experiências e perspectivas. 2000.

[6] Xess Corp. Disponível em: <www.xess.com>. Acesso nov. 2005.

[7] MOORE, G. E. Cramming more components onto integrated circuits. Disponível em <ftp://download.intel.com/research/silicon/moorespaper.pdf>. Acesso nov 2005

[8] The book of Programmable Logic, 1998, Xilinx Inc.

[9] J. Gomes e L. Velho - Computação Gráfica: Imagem, IMPA, 1995

[10] JUNIOR, V. P. da S. Aplicações Práticas do Microcontrolador 8051, 7. Ed., São Paulo: Érica, 1998.

[11] Finite State Machines with Output (Mealy and Moore Machines). Disponível em: <http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Seq/fsm.html>. Acesso nov. 2005

Apêndice A

Código VHDL

Módulo *Buffer*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY bufferIn IS
        PORT(
                clk, get, end_file                      : IN     STD_LOGIC;
                input_byte                              : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                reset, rle_next_byte                    : IN     STD_LOGIC;
                next_byte, write_rle, rle_end_file      : OUT   STD_LOGIC;
                output                                  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
                );
END bufferIn;

ARCHITECTURE rle_st_machine OF bufferIn IS
        TYPE STATE_TYPE IS (A, B, C, D, E);
        SIGNAL state : STATE_TYPE;
        SIGNAL memory       : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
        PROCESS (clk)
                BEGIN
                IF reset = '1' THEN
                        state <= A;
                        memory <= "00000000";
                        next_byte <= '0';
                        write_rle <= '0';
                ELSIF (clk'EVENT AND clk = '1') THEN
                        CASE state IS
                                WHEN A=>
                                        IF end_file = '1' then
                                                state <= E;
                                        ELSE state <= B;
                                        END IF;
                                WHEN B=>
                                        if end_file = '1'  then
```

```
                        write_rle <='0';
                        output <= "00000000";
                        state <= E;
                elsif get = '0' then
                        state <= B;
                else
                        memory <= input_byte;
                        state <= C;
                        write_rle <= '0';
                end if;
        WHEN C=>
                        if rle_next_byte = '1' then
                                next_byte <= '0';
                                output <= memory;
                                write_rle <= '1';
                                state <= D;
                        else state <= C;
                        end if;
        WHEN D=>
                if rle_next_byte = '0' then
                        state <= D;
                else
                        next_byte <= '1';
                        write_rle <= '0';
                        state <= B;
                end if;
        WHEN E=>
                memory <= "00000000";
                next_byte <= '0';
                write_rle <= '0';
                rle_end_file <= '1';
                output <="00000000";
                state <= E;
            END CASE;
    END IF;
END PROCESS;
END rle_st_machine;
```

Módulo RLE

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


ENTITY rle IS
        PORT(
                clk                                                     : IN     STD_LOGIC;

                input_byte                                              : IN   STD_LOGIC_VECTOR(7
DOWNTO 0);
                reset, read, end_file, received    : IN     STD_LOGIC;
                next_byte, out_byte, out_count  : OUT   STD_LOGIC;
```

```
        output                                          : OUT
    STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END rle;

ARCHITECTURE rle_st_machine OF rle IS
    TYPE STATE_TYPE IS (A, B, C, D, E, F, G, H, I, J);      -- G pega proxima entrada
    SIGNAL state                    : STATE_TYPE;
    SIGNAL overflow                 : STD_LOGIC;
    SIGNAL reg1, reg2           : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
    PROCESS (clk)
        variable counter : unsigned(7 downto 0);
        variable last: STD_LOGIC;
        BEGIN
        IF reset = '1' THEN
            state <= A;
            out_byte <= '0';
            out_count <= '0';
            reg2 <= "00000000";
            reg1 <= "00000000";
            last := '0';
            next_byte <= '1';
        ELSIF (clk'EVENT AND clk = '1') THEN
            CASE state IS
                WHEN A=>
                    state <= B;
                WHEN B=>
                    counter := "00000001";
                    state <= G;
                WHEN C=>
                    out_byte <= '0';
                    out_count <= '0';
                    next_byte <= '1';
                    if(end_file = '1')          then
                        last := '1';
                        next_byte <= '0';
                        state <= E;
                    elsif(read = '1') then
                        reg2 <= input_byte;
                        next_byte <= '0';
                        state <= H;
                    else state <= C;
                    end if;
                WHEN H=>
                    if(reg1 = reg2) then
                        counter := counter + 1;
                        state <= D;
                    else
                        state <= E;
                    end if;
                WHEN D=>
```

```
                                        if (counter=255)  then overflow<='1';
                                                                state<=E;

                                else next_byte <= '1';
                                        state<=C;
                                end if;
                                        WHEN E=>
                                                out_byte <= '1';
                                                output <= reg1;          -- coloca o byte na saida;
                                        if(received = '1') then
                                                        state <= F;
                                                        reg1 <= reg2;
                                                else
                                                        state <= E;
                                                end if;
                                        WHEN F=>
                                                out_byte <= '0';
                                                out_count <= '1';
output <= std_logic_vector(counter); -- coloca a
--contagem na saída
                                                if (received /= '1') then
                                                        state <= F;
                                                else
                                                        if(last /= '1') then
                                                                state <= J;
                                                                next_byte <= '1';
                                                        else
                                                                next_byte <= '0';
                                                                state <= I;
                                                        end if;
                                                end if;

                                        WHEN G=>
                                                if(read = '1') then
                                                        next_byte <= '0';
                                                        reg1 <= input_byte;
                                                        state <= C;

                                                else state <= G;
                                                end if;
                                        WHEN I =>
                                                state <= I;
                                        WHEN J =>
                                                counter := "00000001";
                                                state <= C;
                                END CASE;
                        END IF;
                END PROCESS;
END rle_st_machine;
```

Módulo de mapeamento dos sinais de comunicação entre o módulo *Buffer* e módulo RLE

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY rle_comp IS
PORT(
		clk_sis, reset_sis                          : IN    STD_LOGIC;
		get, end_file, received_sis                 : IN    STD_LOGIC;
		input_sis                                   : IN    STD_LOGIC_VECTOR(7 downto 0);
		output_sis                                  : OUT   STD_LOGIC_VECTOR(7 downto 0);
		next_byte, out_byte_sis, out_count_sis      : OUT   STD_LOGIC);
END rle_comp;


ARCHITECTURE rle_structural OF rle_comp IS
	SIGNAL next_byte_signal, read_signal, end_file_signal     : STD_LOGIC;
	SIGNAL input_byte_signal                                  :STD_LOGIC_VECTOR(7 downto 0);


COMPONENT rle

	PORT(
		clk                             : IN     STD_LOGIC;
		input_byte                      : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
		reset, read, end_file, received : IN     STD_LOGIC;
		next_byte, out_byte, out_count  : OUT   STD_LOGIC;
		output                          : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0)
		);

END COMPONENT;

COMPONENT bufferin
PORT(
		clk, get, end_file                  : IN STD_LOGIC;
		input_byte                          : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
		reset, rle_next_byte                : IN STD_LOGIC;
		next_byte, write_rle, rle_end_file  : OUT STD_LOGIC;
		output                              : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
		);
END COMPONENT;

BEGIN
   buffer_comp: bufferin   port map(clk_sis, get, end_file, input_sis, reset_sis, next_byte_signal, next_byte,
read_signal, end_file_signal, input_byte_signal);
rle_comp: rle  port map(clk_sis, input_byte_signal, reset_sis, read_signal, end_file_signal, received_sis,
next_byte_signal,out_byte_sis, out_count_sis,output_sis);

END rle_structural;
```

Apêndice B

Código C

Código do RLE implementado na Linguagem C para o microcontrolador 8051

```
#include <reg51.h>                  /* define 8051 registers */
#include <stdio.h>                  /* define I/O functions */


#define SIZE_FILE 4

void main (void)  {                 /* main program */


  int counter = 1;
  int i = 0;
  int bytesCount = 0;
  int a = 0;
  int size = 0;
  unsigned char file [SIZE_FILE] = {'a', 'b', 'c', 'd'};
  unsigned char byte [SIZE_FILE] = {' ', ' ', ' ', ' '};
  int count [10] = 0;

  unsigned char reg1 = file[0];
  unsigned char reg2 = ' ';
```

```
for(i = 1; i<SIZE_FILE; i++) {
 reg2 = file[i];
 if(reg1 == reg2){
 ++counter;
        byte[size] = reg1;
        count[size] = counter;
 }else {
                ++size;
                counter = 1;
                byte[size] = reg2;
           reg1 = reg2;
                count[size] = counter;
 }      //else
 }//for
}//main
```