UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
DEPARTAMENTO DE MATEMÁTICA ESTATÍSTICA E COMPUTAÇÃO
CURSO DE BACHARELADO EM INFORMÁTICA

Microcontrolador BDLC – Uma Implementação para Aplicação na Industria Automobilística

por

FELIPE DE SOUZA MARQUES

Projeto de conclusão submetida à avaliação, como
requisito parcial para a obtenção do título de
Bacharel em Informática

Prof. Msc. Marcello da Rocha Macarthy
Orientador

Prof. Dr. Fabian Vargas
Co-Orientador

Pelotas, dezembro de 2000

UNIVERSIDADE FEDERAL DE PELOTAS
Reitor: Profa. Inguelore Scheunemann de Souza
Pró-Reitor de Graduação: Prof. João Nelci Brandalise
Diretor do Instituto de Física e Matemática: Prof. Amauri de Almeida Machado
Coordenador do Curso de Informática: Prof. Gil Carlos Medeiros

Sumário

1. Introdução

Este trabalho apresenta um protótipo para a implementação do módulo de comunicação serial Byte Data Link Controller (BDLC), que é responsável pela transmissão de mensagens (utilizando protocolo SAE J1850) no Sistema Tolerante a Falhas Baseado em Microcontroladores Comerciais, segundo [RIB 00], demonstrando como será sua utilização neste e os benefícios que proporcionará para uma maior eficiência do sistema.

Para realizar o trabalho proposto é apresentado o BDLC, fazendo a descrição do componente e das estruturas que o formam, a partir desta é elaborado um diagrama hierárquico das estruturas na composição do módulo. Para viabilizar a implementação do projeto utiliza-se as vantagens oferecidas pela linguagem VHDL, no que refere-se a flexibilidade que esta oferece para migrar de uma ferramenta para outra. Além disso, são feitas análises funcionais e temporais da parte implementada do sistema, demonstrando os resultados obtidos pelos testes e ainda apresenta os benefícios da implementação em VHDL do microcontrolador (BDLC).

Os capítulos seguintes tratam, cada um deles, de assuntos que envolvem o desenvolvimento deste projeto, conforme a breve descrição que é a partir de agora apresentada.

O segundo capítulo trata especificamente da descrição do microcontrolador que é implementado neste trabalho, enfatizando principalmente sua estrutura para que se possa elaborar uma base de dados para o desenvolvimento do protótipo.

O terceiro capítulo aborda-se a linguagem de descrição de hardware (VHDL), com as estruturas básicas utilizadas no desenvolvimento do trabalho em questão, suas características léxicas, os tipos de dados que admite e seus comandos básicos, exemplificando seu funcionamento para facilitar a compreensão dos que venham a se utilizar deste material, visto ser esta uma das linguagens suportadas pela ferramenta MAX+PLUS II, usada nesta etapa, e por outras que são necessárias para a completa implementação do sistema em pauta.

O quarto capítulo, descreve o funcionamento, forma de utilização e características da ferramenta MAX+PLUS II, fazendo referência especial aos benefícios que ela oferece para o projeto específico.

O quinto capítulo apresenta uma solução para viabilizar a implementação do BDLC e as técnicas usadas neste processo. Como passo inicial implementa-se a estrutura do módulo, isto é, defini-se todas as entidades do microcontrolador, em seguida utiliza-se a VHDL para realizar a descrição comportamental de cada uma das entidades. Por fim utiliza-se o MAX+PLUS II para testar e sintetizar o projeto.

2. Apresentando o BDLC

O Byte Data Link Controller – BDLC, de acordo com [BDL 97], é um módulo de comunicação serial. Foi desenvolvido pela MOTOROLA e permite que o usuário envie e receba mensagens através do protocolo de comunicação serial SAE J1850, desenvolvido pela Society of Automotive Engineers – SAE. O BDLC provê o acesso a rede, tomada de decisões, formatação de mensagens e detecção de erros. O trabalho, aqui apresentado, visa implementar este módulo em VHDL para possibilitar que seja dada sequência, com facilidade, ao projeto proposto em [RIB 00], descrevendo a importância deste módulo dentro do sistema.

2.1 Diagrama de blocos do BDLC

O BDLC é composto por três blocos (fig. 2.1), são eles os seguintes:

- **Bloco de interface da CPU**: provê a transferência de informações entre a CPU e o BDLC. Este bloco é composto de cinco registradores, que são utilizados para configuração e controle de *status* do MCU.
- **Bloco manipulador de protocolo**: provê a transmissão e recepção do buffer que contém a mensagem, realiza tomada de decisão, gera os pacotes para transmissão, isto é, realiza a formatação das mensagens, gera e verifica CRC e realiza detecção de erros.
- **Bloco de interface do MUX**: este realiza a codificação e decodificação e também a filtragem digital de ruídos entre o manipulador de protocolo e a interface física.

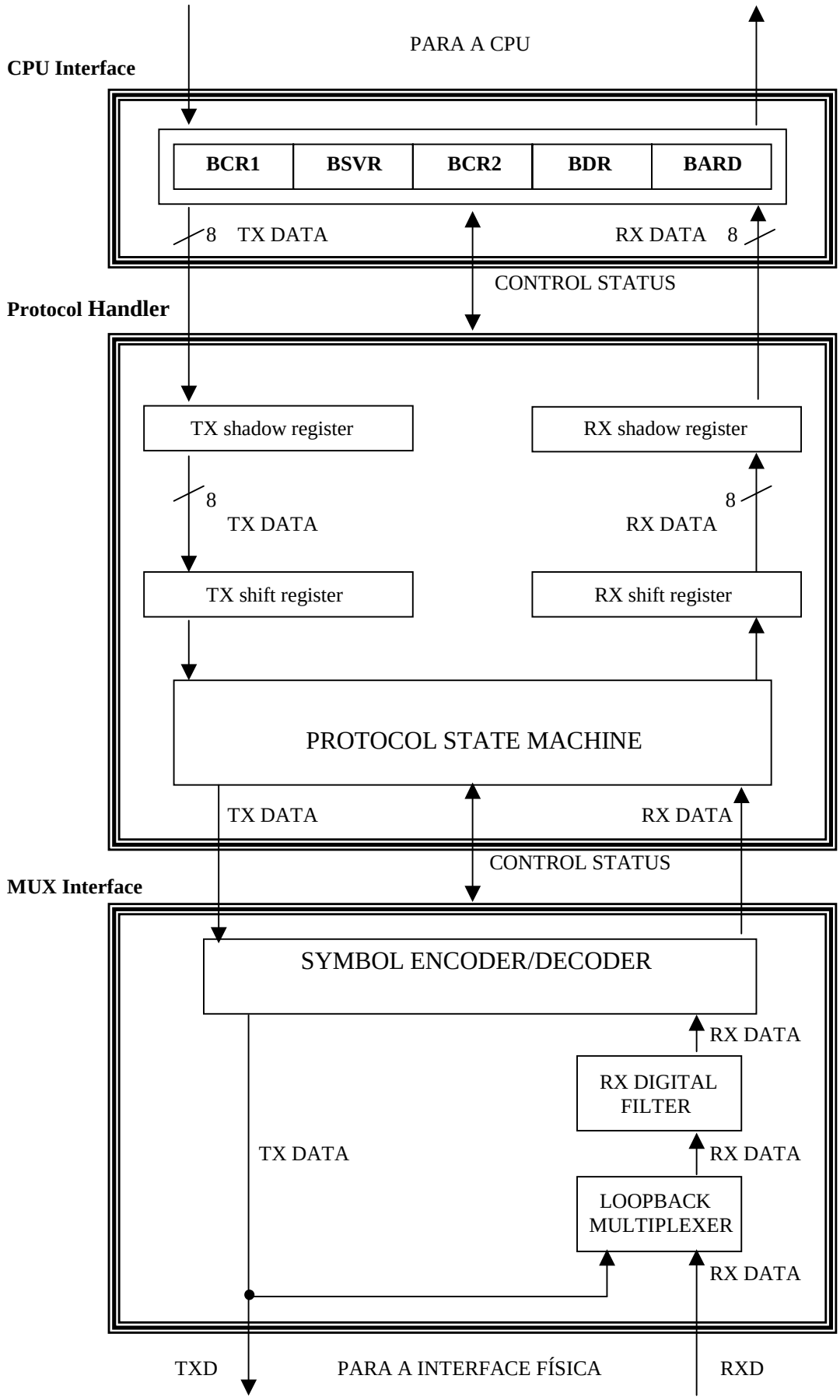

FIGURA 2.1 – Diagrama de blocos do BDLC.

2.2 Modos de operação do BDLC

Os principais modos de operação do BDLC são: *power off mode*, *reset mode*, *run mode*, *wait mode* e *stop mode*.

2.2.1 Power off mode

Sempre que a tensão do MCU for menor que o valor mínimo especificado para garantir o funcionamento do BDLC, o *power off mode* é ativado a partir do *reset mode*. Neste modo, as especificações dos pinos de entrada e saída não são garantidas.

2.2.2 Reset mode

O *reset mode* é iniciado pelo *power off mode* quando a tensão atinge o valor mínimo especificado e um *reset source* do MCU estiver *setado*[1]. Este modo também pode ser ativado a partir de qualquer outro modo, desde que um dos *reset sources* esteja *setado*.

2.2.3 Rum mode

No *rum mode* ocorrem as operações normais de rede. Pode-se assegurar que todas as transmissões do BDLC tenham sido cessadas antes da saída do *rum mode*.

Pode ser iniciado a partir do *reset mode*, do *wait mode* e do *stop mode*. É iniciado a partir do *reset mode* quando não houver *reset sources setados*. Já pelo o *wait mode*, é iniciado quando detecta alguma atividade no barramento e , ainda, pode ser iniciado a partir do *stop mode* quando alguma atividade é percebida.

2.2.4 Wait mode

Esse modo é iniciado a partir do *run mode* sempre que a CPU executar a instrução WAIT e se o bit WCM do registrador BCR1 for *resetado*[2] previamente. No *wait mode* não há transmissão.

2.2.5 Stop mode

É iniciado pelo *run mode* quando a CPU executar uma instrução STOP ou uma instrução WAIT e o bit WCM no registrador BCR1 é *setado* previamente. Nesse modo os relógios internos do BDLC permanecem parados. Para ativar o BDLC uma transição inativa para ativa deve ocorrer no barramento J1850.

2.3 Bloco de interface da CPU

O bloco de interface da CPU realiza a comunicação entre a CPU e o BDLC. Ele é composto por cinco registradores de 8 bits, estes são: BCR1, BSVR, BCR2, BDR e BARD.

---

[1] Indica a ativação de um bit, isto é, o bit assume o nível lógico um.
[2] O bit assume o nível lógico zero.

2.3.1 BDLC Control Register 1 – BCR1

O BCR1 configura e controla o BDLC. Ele possui algumas funções básicas, tais como habilitar interrupções, configurar os relógios do módulo e indicar as mensagens recebidas que podem ser ignoradas.

O registrador possui cinco bits de configuração (fig. 2.2), sendo que quatro (CLKS, RS [1:0] e WCM) podem ser *setados* apenas uma vez após o *reset* do MCU. O bit IE, que habilita interrupções, pode ser escrito a qualquer hora. O IMSG é um bit usado pelo software de comunicação do SAE J1850 e não é considerado um bit de configuração.

7	6	5	4	3	2	1	0
IMSG	CLKS	RS [1:0]		0	0	IE	WCM

FIGURA 2.2 – BDLC Control Register 1.

2.3.1.1 Ignore message bit – IMSG

É usado quando o usuário determina que uma mensagem recebida não é de seu interesse. O bit é *setado* e após isto não ocorrem mais interrupções de CPU enquanto um símbolo que indique o início de frame (SOF) seja detectado. Quando ocorre um SOF, as interrupções são reabilitadas. O bit *setado* com 0 indica que o receptor está habilitado e com 1 está desabilitado.

2.3.1.2 Clock select bit – CLKS

Permite a seleção de freqüências de relógio para operação de BDLC: 1.049 MHz ou 1.00 MHz. Isto permite uma flexibilidade na escolha de freqüências osciladoras binárias ou freqüências osciladoras inteiras. Este bit é utilizado em conjunto com a classe de bits selecionados para deslocar a freqüência do relógio de sistema do MCU (f_{sys}) para a freqüência do relógio(f_{bdcl}) de operação do BDLC. O 0 lógico indica freqüência inteira (1.00 MHz) e o 1 lógico, indica freqüência binária (1.049 MHz).

2.3.1.3 Rate select field – RS [1:0]

Esses bits são utilizados em conjunto com o CLKS para deslocar f_{sys} resultando em f_{bdcl}. O RS [1:0] (tab. 2.1 e tab. 2.2)provê o deslocamento necessário para dividir a freqüência de *clock* do sistema abaixo da requerida freqüência de operação do BDLC.

TABELA 2.1 – BDLC Rate Selection para freqüências binárias (CLKS = 1).

F_{sys}	R1	R0	Valor de deslocamento	f_{bdcl}
1.00 MHz	0	0	1	1.049 MHz
2.00 MHz	0	1	2	1.049 MHz
4.00 MHz	1	0	4	1.049 MHz
8.00 MHz	1	1	8	1.049 MHz

Fonte: [BDL 97]. Initialization. p. 4-3.

TABELA 2.2 – BDLC Rate Selection para freqüências inteiras (CLKS = 0).

F_{sys}	R1	R0	Valor de deslocamento	F_{bdcl}
1.00 MHz	0	0	1	1.00 MHz
2.00 MHz	0	1	2	1.00 MHz
4.00 MHz	1	0	4	1.00 MHz
8.00 MHz	1	1	8	1.00 MHz

Fonte: [BDL 97]. Initialization. p. 4-3.

2.3.1.4 Interrupt enable bit – IE

O IE habilita interrupções do BDLC para a CPU, isto é, se estiver *setado*, as interrupções do BDLC podem causar uma interrupção de CPU. Se estiver *resetado*, somente uma interrupção de *wakeup* do *wait mode* (com WCM = 1) ou *stop mode* pode causar uma interrupção de CPU. O bit *setado* com 0, indica que as requisições de interrupção estão desabilitadas. Do contrário se estiver *setado* com 1, indica que as requisições estão habilitadas.

2.3.1.5 Wait clock mode bit – WCM

O WCM determina a operação dos relógios internos do BDLC durante o *wait mode*. Se o bit estiver *setado* (WCM = 1), os relógios param quando uma instrução de WAIT é executada pela CPU. No entanto se não estiver *setado* (WCM = 0), o relógios continuam funcionando durante o *wait mode*.

2.3.2 BDLC State Vector Register – BSVR

O BSVR indica o *status* corrente do BDLC (fig. 2.3). Cada estado do registrador pode opcionalmente gerar uma interrupção de CPU, exceto para ativação (wake-up), na qual gera uma interrupção de CPU não mascarável.

7	6	5	4	3	2	1	0
0	0	ISF [3:0]				0	0

FIGURA 2.3 – BDLC State Vector Register.

A tabela abaixo (tab. 2.3) mostra a codificação dos *status* e a indicação de prioridade para todas as fontes de interrupção do BDLC.

TABELA 2.3 – Fontes de Interrupção do BSVR.

BSVR	**ISF3**	**ISF2**	**ISF1**	**ISF0**	**Fonte de Interrupção**	**Prioridade**
$00	0	0	0	0	Nenhuma interrupção pendente	0 (mais baixa)
$04	0	0	0	1	EOF recebido	1
$08	0	0	1	0	IFR byte recebido (RXIFR)	2
$0C	0	0	1	1	Registrador de dados Rx cheio (RDRF)	3
$10	0	1	0	0	Registrador de dados Tx vazio (TDRE)	4
$14	0	1	0	1	Perda de decisão	5
$18	0	1	1	0	Erro de CRC	6

$1C	0	1	1	1	Símbolo inválido ou fora da faixa	7
$20	1	0	0	0	Ativação (wake-up)	8 (mais alta)

Fonte: [BDL 97]. BDLC State Vector Register. p. 5-1.

2.3.3 BDLC Control Register 2 – BCR2

Esse registrador controla algumas operações de transmissão do BDLC (fig. 2.4). Tem como funções básicas o controle de entrada e saída de *loopback modes*, a configuração do bit de normalização IFR, o controle de entrada no modo de recepção 4X, a indicação para o BDLC quando o último byte de uma mensagem transmitida ou IFR tenha sido escrito no BDR.

O BCR2 possui três bits de controle (ALOOP, DLOOP e RX4XE) que normalmente são utilizados para indicar a operação em modos especiais. O quarto bit (NBFS) é apenas para configuração. O conjunto de bits restantes (TEOD, TSIFR, TMIFR1 e TMIFR0) é utilizado durante a operação normal.

7	6	5	4	3	2	1	0
ALOOP	DLOOP	RX4XE	NBFS	TEOD	TSIFR	TMIFR1	TMIFR0

FIGURA 2.4 – BDLC Control Register 2.

2.3.3.1 Analog loopback mode bit – ALOOP

O bit indica para o BDLC se o transceptor analógico foi colocado em *analog loopback mode*, onde seu controlador de saída é curto-circuitado (*by-passed*) passado internamente e o sinal transmitido é realimentado para o comparador receptor. Quando o bit é *setado*, o BDLC opera normalmente. Já quando é *resetado*, o BDLC espera que o barramento fique inativo por pelo menos um período que indique o final de um frame (EOF), para que comece a receber mensagens. Espera-se também, que o barramento fique inativo, por pelo menos um período de separação *inter-frame*, antes de começar a transmitir mensagens. Considerando o nível lógico desse bit, 0 indica que o transceptor analógico saiu do *loopback mode* e 1 indica que o transceptor analógico está no *loopback mode*.

2.3.3.2 Digital loopback mode bit – DLOOP

O DLOOP isola o transceptor analógico para determinar se um nó transceptor está causando uma interrupção no barramento. Quando o bit é *setado*, o tranceptor é curto-circuitado, com o sinal digital transmitido passado diretamente para o circuito receptor digital. Esse processo resulta nos dados colocados no BDR para transmissão sendo transmitido diretamente (back) no BDR para ser recebida.

Possui as mesmas propriedades do ALOOP quando *resetado*. Quando é *resetado* e o barramento tem estado inativo por um tempo de EOF, uma interrupção de EOF é gerada, indicando que o BDLC está pronto para receber mensagens. O nível lógico 0 indica que a saída de transmissão digital está conectada diretamente ao pino de transmissão, e o pino de recepção está conectado a entrada de recepção digital. O 1 lógico, indica que a saída de transmissão digital está conectada indiretamente a entrada de recepção digital, isto é, os pinos de recepção e transmissão não estão conectados.

2.3.3.3 Receive 4x mode enable bit – RX4XE

Permite que o BDLC receba mensagens transmitidas a quatro vezes a taxa normal (41.6 kbps). No modo 4X o BDLC só recebe mensagens, não pode transmiti-las. Se , no modo 4X, o BDLC receber um símbolo de BREAK, o bit RX4XE é *resetado* automaticamente e o BDLC retorna a operação normal. A indicação de que o BDLC esteja no modo 4X é o nível lógico 1, caso contrário (nível lógico 0), o BDLC recebe e transmite a 10.4 kbps.

2.3.3.4 Normalization bit format select bit – NBFS

O bit NBFS determina o formato do *normalization bit* usado pelo BDLC enquanto transmite e recebe respostas *in-frame* na rede. O *normalization bit* (NB) precede o primeiro byte de uma IFR. O NBFS estando *setado* (nível lógico 1), indica que o NB que é recebido ou transmitido é um 0 lógico quando a parte de resposta de um IFR termina com um byte de CRC. Do contrário, se o nível lógico do NBFS for 0, o NB é um 1 lógico.

2.3.4 BDLC Analog Roundtrip Delay Register – BARD

O BARD possui seis bits que são usados para controlar o BDLC durante a execução do protocolo (fig. 2.5). Esses bits são escritos apenas no reset do MCU e, após isto, tornam-se apenas bits para leitura.

7	6	5	4	3	2	1	0
ATE	RXPOL	0	0	BO [3:0]			

FIGURA 2.5 – BDLC Analog Roundtrip Delay Register.

2.3.4.1 Analog transceiver enable bit – ATE

O bit seleciona um tipo de protocolo: SAE J1850 *integrated* ou *stand-alone*. Para MCUs que contém transceptor analógico integrado, esse transceptor é selecionado com o bit *setado* (1 – *on-board*). Se o transceptor for externo, o bit é *resetado*(0 – *off-chip*). Caso não haja um transceptor integrado, esse bit não tem efeito sobre a operação do BDLC.

2.3.4.2 Receive polarity bit – RXPOL

O RXPOL seleciona a polaridade dos dados digitais recebidos vindos de um transceptor SAE J1850 externo. O 0 lógico seleciona polaridade invertida, onde um transceptor externo inverte o sinal recebido de um barramento SAE J1850. Já o 1 lógico seleciona a polaridade verdadeira, a qual não inverte o sinal recebido.

2.3.4.3 BDLC analog roundtrip delay offset field - BO[3:0]

O conjunto de bits BO ajusta o bit transmitido e símbolos de cronometragem para calcular os diferente *roundtrips delays* encontrados em diferentes transceptores analógicos SAE J1850. A faixa de *delay* permissível é de 9µs a 24 µs, com alvo nominal de 16µs (valor de reset).

A tabela abaixo (tab. 2.4) mostra os valores correspondentes do BO[3:0] para os esperados *delays* do transceptor.

TABELA 2.4 – BO[3:0] Offset Values.

BO3	BO2	BO1	BO0	Delay
0	0	0	0	9µs
0	0	0	1	10µs
0	0	1	0	11µs
0	0	1	1	12µs
0	1	0	0	13µs
0	1	0	1	14µs
0	1	1	0	15µs
0	1	1	1	16µs
1	0	0	0	17µs
1	0	0	1	18µs
1	0	1	0	19µs
1	0	1	1	20µs
1	1	0	0	21µs
1	1	0	1	22µs
1	1	1	0	23µs
1	1	1	1	24µs

Fonte: [BDL 97]. Initialization. p. 4-7.

2.4 Bloco manipulador de protocolo

O manipulador de protocolo é composto por dois registradores de deslocamento (TX – transmissão e RX - recepção), dois registradores shadow e a máquina de estados de protocolo.

2.4.1 Registrador de deslocamento RX

Esse registrador recebe os bits seriais do barramento pelo decodificador de símbolos da interface do MUX e os disponibiliza na forma paralela para o registrador shadow RX.

2.4.2 Registrador de deslocamento TX

O registrador de deslocamento TX disponibiliza serialmente os dados recebidos (na forma paralela) do registrador shadow TX para o codificador da interface do MUX.

2.4.3 Registrador shadow RX

O registrador shadow RX armazena os bytes recebidos do barramento, os quais podem ser lidos pelo BDR. Uma leitura do BDR sempre retorna o último byte escrito no registrador shadow RX.

2.4.4 Registrador shadow TX

O shadow TX é usado para armazenar os bytes escritos pelo BDR para transmissão. Imediatamente após um byte ser escrito nesse registrador, ele é transferido para o registrador de deslocamento TX.

2.4.5 Máquina de estados de protocolo

A máquina de estados de protocolo executa e controla as funções para que seja realizado o protocolo SAE J1850. Algumas destas funções são: prover o acesso ao barramento, formatação de mensagens, detecção de colisões, decisões e realizar o tratamento para CRC.

2.5 Bloco de interface do MUX

A interface do multiplexador do BDLC é composta por um codificador/decodificador de símbolos, um filtro digital RX e o multiplexador digital de loopback.

2.5.1 Codificador / decodificador de símbolos

O codificador recebe informações de formatação de mensagens e dados transmitidos serialmente do manipulador de protocolo e codifica-os no nível lógico de um símbolo de Variable Pulse Width – VPW (seqüência de bits de dados). Essa seqüência é transmitida para interface física via protocolo SAE J1850.

Da mesma forma o decodificador recebe a seqüência de dados da interface física e os decodifica para que possam ser repassados para o manipulador de protocolo.

2.5.2 Bloco do filtro digital RX

Esse componente é um filtro passa baixa digital, que serve para remover ruídos que surgem no barramento SAE J1850. A figura abaixo descreve o filtro (fig. 2.6).

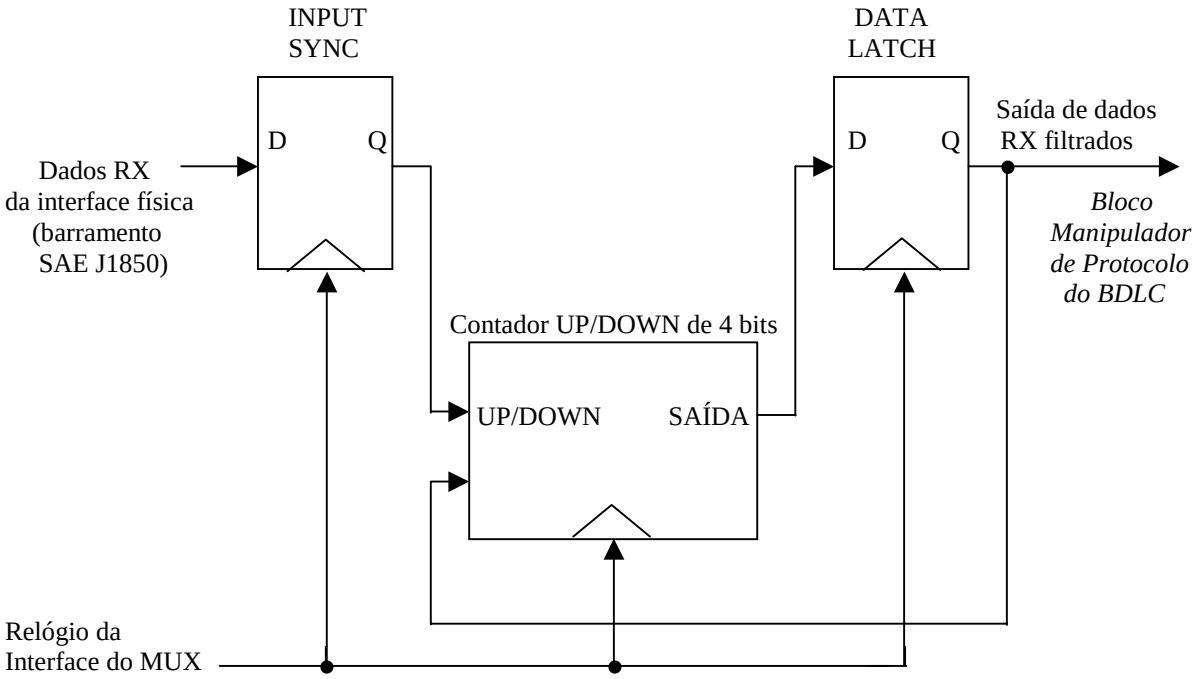

FIGURA 2.6 – Filtro Digital RX

2.6 Ciclo de desenvolvimento do projeto

Com a descrição do módulo parte-se para a solução da sua concreção seguindo o ciclo de desenvolvimento do projeto (fig. 2.7), segundo [AIR 99], que se inicia com a modelagem de alto nível, quando é feita uma descrição do sistema de forma textual mesclada com esquemas, de maneira a tornar fácil a compreensão deste, oferecendo uma referência a ser seguida na próxima etapa de desenvolvimento.

Segue-se com a formação do projeto eletrônico, isto é, são feitas as especificações funcionais de todos os componentes do BDLC, passando os requisitos da descrição do sistema para uma descrição formal, com o uso do VHDL aplicado ao MAX+PLUS II.

Com o uso da ferramenta obtém-se a síntese do circuito automaticamente após o processo de compilação. Pode-se conceituar síntese como sendo: “a tradução de uma descrição comportamental para uma descrição estrutural, em que cada elemento representa um dispositivo eletrônico pré-definido” [AIR 99].

Após o processo de compilação dá-se inicio a simulações sobre o protótipo implementado, para análises temporais e funcionais validando o sistema.

Finalmente, parte-se para a representação física do componente, no caso deste trabalho utiliza-se os dispositivos programáveis oferecidos pela arquitetura Altera, que para tanto usa a interface de programação do MAX+PLUS II.

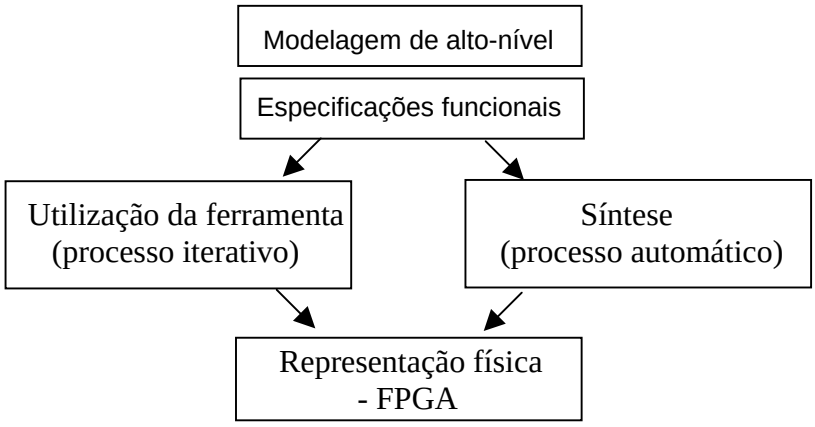

FIGURA 2.7 – Ciclo de desenvolvimento do projeto.

2.7 Diagrama hierárquico da estrutura do BDLC

Nesta seção é colocada a descrição do BDLC partindo-se de um diagrama hierárquico (fig. 2.8) com uma abordagem voltada a estrutura de formação do MCU. Cada uma das estruturas mencionadas no diagrama será implementada com a VHDL através da ferramenta MAX+PLUS II.

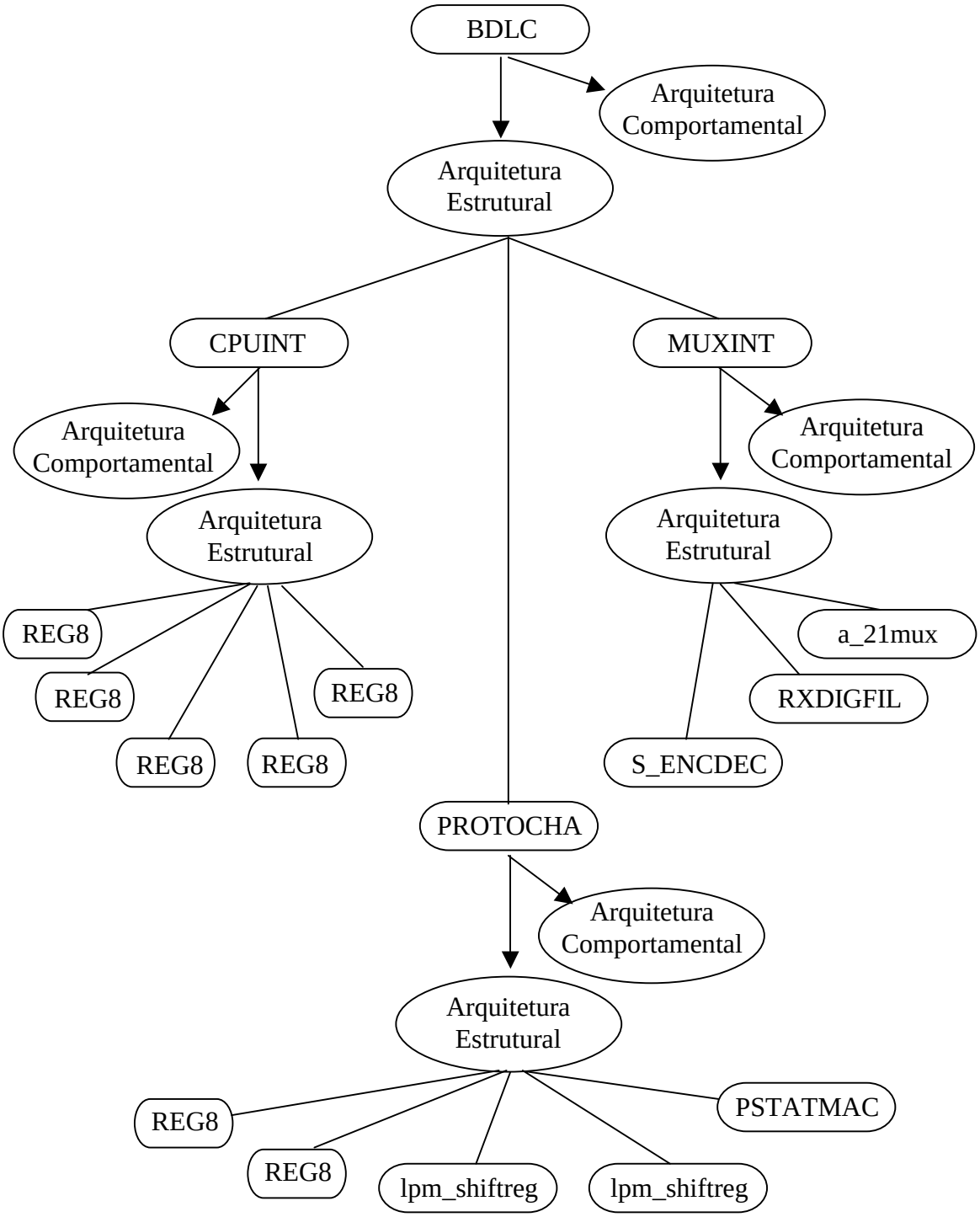

FIGURA 2.8 – Diagrama Hierárquico do BDLC.

3. VHDL – A linguagem de descrição de hardware

Neste capítulo são apresentados os componentes para a realização de um projeto em VHDL, a semântica da linguagem e objetos utilizados para desenvolver um projeto. O objetivo é facilitar a compreensão da linguagem utilizada para a execução do projeto, tendo em vista a flexibilidade oferecida pela linguagem em termos de plataformas e ferramentas.

VHDL é uma linguagem de descrição de hardware que representa entradas e saídas, comportamento e funcionalidade de circuitos. Atualmente faz parte da maioria das ferramentas de projetos eletrônicos.

3.1 Histórico

Em 1981, devido a necessidade de criar uma linguagem de programação padronizada específica para o trabalho com hardware, iniciou-se o desenvolvimento da VHSIC Hardware Description Language, por um programa do Departamento de Defesa Americano, o *Very High Speed Integrated Circuits* (VHSIC), para ser usada como uma linguagem padrão pela comunidade de microeletrônica.

Já em julho de 1983, uma equipe de desenvolvedores da Intermetrics, IBM e Texas Intruments são contratadas para desenvolver uma nova linguagem e implementar um software que necessitasse do uso desta linguagem. Três anos depois, em julho de 1986, a Intermetrics entregou uma série inicial do software e em fevereiro de 1987, uma ferramenta completamente revisada surgia sob a versão 7.2.

Por fim, a linguagem foi definida por duas padronizações sucessivas pelo *Institute of Electrical and Electronics Engineers* (IEEE), uma em 1987 – IEEE Std 1076-1987 (chamada de “VHDL 1987”) – e a outra em 1993 – IEEE Std 1076-1993 (chamada de “VHDL 1993”).

3.2 Componentes de um projeto

A estrutura de um programa VHDL é baseada em blocos, que serão apresentados brevemente nesta seção, com a finalidade de expor os recursos que a linguagem oferece para a realização de um projeto qualquer, em suas diversas etapas, demonstrando como são utilizados.

Os principais blocos, segundo [LIP 90] são: a declaração de entidades (entity), a arquitetura (architecture), os sub-programas, a declaração de pacotes (package) e o corpo do pacote (package body), que serão comentadas nos itens a seguir. Convém, no entanto, chamar a atenção para os requisitos que compõem um projeto segundo [AIR 99], apresentado abaixo (fig. 3.1):

	Primários	**Secundários**
Hierarquia de Hardware	ENTITY CONFIGURATION	ARCHITECTURE
Hierarquia de Software	PACKAGE	PACKAGE BODY

FIGURA 3.1 - Blocos principais de um projeto.

3.2.1 Declaração de entidades

A declaração de uma entidade serve para descrever singularmente um componente de hardware, dando inicio a formação do mesmo, com o número de entradas e o número de saídas, isto é, define as portas do componente. Segundo [PER 93], as entidades equivalem a símbolos gerados por ferramentas como o MAX+PLUS II.

Uma forma geral de definição (fig. 3.2), conforme [LIP90], é:

```
ENTITY nome_da_entidade IS
        GENERIC( parametro1 : string :=  valor;
                        Parametro2 : integer:=  valor);
        PORT(
                entrada1 : IN  STD_LOGIC;
                Entrada2  :  IN  STD_LOGIC_VECTOR(max_valor  downto
min_valor);
                E_s : INOUT  STD_LOGIC;
                Saida1, saida2 : OUT  STD_LOGIC);
END nome_da_entidade;
```

FIGURA 3.2 – Modelo de declaração de entidade.

3.2.2 Blocos de arquitetura

Esta declaração define o comportamento do componente, a sua funcionalidade, ou seja, se constrói um modelo comportamental do dispositivo. Através deste modelo é que se pode definir a finalidade do componente dentro de um projeto. Dentro de um bloco de arquitetura, de acordo com [PER 97], ainda definem-se os componentes de uma arquitetura, podendo-se dizer, em um nível mais abstrato, que a mesma equivale a um projeto esquemático.

A declaração de uma arquitetura (fig. 3.3), conforme [LIP 90], pode ser:

```
ARCHITECTURE corpo OF nome_da_entidade IS
        SIGNAL s1 : STD_LOGIC;
        SIGNAL s2 : STD_LOGIC;
BEGIN
        -- Segmentos, processos, chamadas de procedimentos, ...
END corpo;
```

FIGURA 3.3 – Modelo de declaração de uma arquitetura.

3.2.3 Sub-programas

Os sub-programas são uma seqüência de declarações que pode ser invocadas repetidamente em diferentes partes de uma descrição VHDL. Estes sub-programas são: procedimentos e funções, que são seguidamente utilizados na tarefa de programar.

Um exemplo de um sub-programa (fig. 3.4):

```
FUNCTION max (a, b : INTEGER) RETURN INTEGER IS
BEGIN
    IF a > b THEN
        RETURN a;
    ELSE
        RETURN b;
    END IF;
END max;
```

FIGURA 3.4 – Exemplo da utilização de sub-programas.

3.2.4 Declaração de pacotes e corpo de pacotes

Um pacote envolve declaração de constantes, tipos de dados e sub-programas, o que viabiliza a reutilização de código. Eles podem ser divididos entre definição e corpo; e podem ser usados por mais de uma entidade e arquiteturas.

Pode-se dizer, resumidamente, que um pacote seria uma definição de um conjunto de dados utilizada como uma biblioteca. O corpo do pacote contém a implementação dos sub-programas utilizados por uma arquitetura.

Para exemplificar um pacote (fig. 3.5), será utilizada a função (*function max(a, b : integer))* definida na seção anterior:

```
PACKAGE exemplo IS                    -- Definição do pacote
    TYPE bit_8 is BIT_VECTOR (7 DOWNTO 0);
    CONSTANT tamanho : INTEGER := '10';
    FUNCTION max (a, b : INTEGER) RETURN INTEGER;
END exemplo;
PACKAGE BODY exemplo IS               -- O corpo do pacote
    FUNCTION max (a, b : INTEGER) RETURN INTEGER IS
    BEGIN
        IF a > b THEN
            RETURN a;
        ELSE
            RETURN b;
        END IF;
    END max;
END exemplo;
```

FIGURA 3.5 – Exemplo da composição de um pacote.

3.3 Características da Linguagem

Nesta seção são abordadas características em relação à semântica da linguagem VHDL, de acordo com [LIP 90]. Deseja-se neste item apenas apresentar noções básicas, tornando possível a compreensão da maneira como se trabalha com a linguagem, sem a pretensão de abranger toda sua sintaxe.

3.3.1 Comentários

Os comentários da linguagem servem para que o programador faça “anotações” sobre a programação em certos pontos do código, e na VHDL são indicados pela seqüência de dois hífens (--). Estende-se até o fim da linha, significa dizer que para continuar o comentário na próxima linha é necessário fazer nova indicação.

3.3.2 Operadores

Os operadores da linguagem (tab. 3.1) são:

TABELA 3.1 – Operadores da linguagem VHDL

Tipos	Operadores
Lógicos	**and, or, nand, nor, xor, xnor** e **not**
Numéricos	soma (+), subtração (-), multiplicação (*), divisão (/), módulo (mod), remanescente (rem), expoente (**), valor absoluto (abs)
Relacionais	igual (=), diferente (/=), maior que (>), menor que (<), maior ou igual (>=), menor ou igual (<=)

3.3.3 Identificadores

Os caracteres aceitos em identificadores são todos os alfanuméricos e o caracter *underline* (_). Os identificadores devem iniciar por uma letra e não fazem distinção a letras maiúsculas e minúsculas. Não é permitido o uso do caracter *underline* no fim de um identificador e nem a utilização de dois *underlines* seguidos.

Exemplos de identificadores:
entrada_padrão, DATA, d1, saida2, ent_1_A.

3.3.4 Tipos de dados

Abaixo (tab. 3.2) serão definidos os tipos que implicitamente são usados por todas as entidades de um projeto. Estes tipos fazem parte do pacote STANDARD, definido pela padronização do IEEE.

TABELA 3.2 – Tipos de dados que compõem o pacote *standard.*

Tipos	Valores
BOOLEAN	pode assumir valores *true* ou *false*
BIT	assume valores ‘0’ e ‘1’
CHARACTER	pode assumir valores da lista de caracteres descrita abaixo: NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FSP, GSP, RSP, USP, ‘ ‘, ‘!’, ‘”’, ‘#’, ‘$’, ‘%’, ‘&’, ‘’’, ‘(‘, ‘)’, ‘*’, ‘+’, ‘,’, ‘-‘, ‘.’, ‘/’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘:’, ‘;’, ‘<’, ‘=’, ‘>’, ‘?’, ‘@’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, ‘Z’, ‘[‘, ‘\’, ‘]’, ‘^’, ‘_’, ‘`’, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’, ‘l’, ‘m’, ‘n’, ‘o’, ‘p’, ‘q’, ‘r’, ‘s’, ‘t’, ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, ‘z’, ‘{‘, ‘|’, ‘}’,

	'~', DEL
SEVERITY_LEVEL	este tipo é usado para tratamento de erro, pode assumir os valores *NOTE, WARNING, ERROR* e *FAILURE*
INTEGER	números inteiros, não se pode fazer conversão sobre inteiros
REAL	números reais, sempre acompanhados pelo ponto decimal
TIME	representa medias de tempo. Estes podem ser: **fs**; femtosecond **ps** = 1000 fs; picosecond **ns** = 1000 ps; nanosecond **us** = 1000 ns; microsecond **ms** = 1000 us; millisecond **sec** = 1000 ms; second **min** = 60 sec; minute **hr** = 60 min; hour
STRING	representa um vetor de caracteres
BIT_VECTOR	representa um vetor de bits
NATURAL	representa números inteiros positivos incluindo '0'
POSITIVE	representa números inteiros positivos

Outros tipos (tab. 3.3), que não fazem parte do pacote STANDARD, podem ser utilizados:

TABELA 3.3 – Tipos de dados que não fazem parte do pacote *standard.*

Tipos	Descrição
RANGE	serve para determinar intervalos para serem utilizados por um determinado tipo
ARRAY	é definido como uma coleção de dados de um determinado tipo
RECORD	é uma coleção de elementos de tipos diferentes

3.4 Objetos

Objetos podem ser constantes, variáveis e sinais. Estes podem ser definidos como vetores ou assumirem valores escalares. Para fazer referência a vetores, segue-se o exemplo: **v** é um vetor; **v** (2) é o elemento 2 do vetor; também podem ser referenciadas "fatias" de um vetor, **v** (1 to 5) são referenciados os elementos de 1 a 5 do vetor **v**.

Estes objetos possuem as mesmas características dos identificadores tratando-se de questões léxicas, isto é, devem obrigatoriamente iniciar por letras e, depois, podem ser seguidos por letras e dígitos (e também o caracter ' _ ').

3.4.1 Constantes

Uma constante serve para armazenar valores fixos. Consistem de um nome, um tipo e um valor que pode ser opcional, desde que haja uma declaração posterior. É conveniente declarar constantes que são utilizadas com freqüência em um *package*.

Sintaxe: CONSTANT id : TIPO [:= expressão].
Exemplo: CONSTANT tamanho_ROM : INTEGER := 16#FFFF#.

3.4.2 Sinais

São utilizados para comunicação entre módulos. Devem ser declarados em *entity*, *architecture* ou *package*. Não podem ser declarados em processos. Os sinais são temporizados.

Sintaxe: SIGNAL id (s) : TIPO [restrição] [:= expressão].
Exemplos: SIGNAL clk, load : BIT := ‘0’;
SIGNAL bus : BIT_VECTOR;
SIGNAL aux : INTEGER RANGE 10 DOWNTO 1.

3.4.3 Variáveis

As variáveis são utilizadas para atribuição imediata de valores. Elas podem ser utilizadas em processos, sem temporização. Normalmente são utilizadas na modelagem comportamental de alto-nível.

Sintaxe: VARIABLE id : TIPO [restrição] [:=expressão]
Exemplos: VARIABLE end : INTEGER RANGE 0 TO tamanho_ROM;
VARIABLE buf : BIT_VECTOR (0 TO 31);

3.5 Comandos seqüenciais

3.5.1 Atribuições a variáveis

O símbolo utilizado para atribuições é “**:=**”. Estas variáveis não passam valores fora do processo na qual foram declaradas, isto é, elas são locais.

Exemplo (fig. 3.6):

PROCESS (a, b)
VARIABLE c : BIT;
BEGIN
c := a AND b; -- Atribuição
END PROCESS;

FIGURA 3.6 – Exemplo de uma atribuição em um processo

3.5.2 Atribuições a sinais

Para atribuição a sinais, utiliza-se o símbolo “<=”.

Exemplo:
A <= ‘1’, ‘0’ AFTER 64 ns;

3.5.3 Condicionais

IF-THEN-ELSE

Este comando também contém a cláusula (ELSIF), podendo ser utilizado, segundo [LIP 90], como no modelo abaixo (fig. 3.7):

IF *condição1* THEN
<comandos1>
ELSIF *condição2* THEN
<comandos2>
ELSE
<comandos3>
END IF;

FIGURA 3.7 – Modelo da estrutura do IF – THEN - ELSE

CASE

Este seleciona uma execução a partir de uma lista de opções, como demonstrado no exemplo abaixo (fig. 3.8):

PROCESS (a, b, c)
BEGIN
CASE a IS
WHEN “00” => x <= b AND c
WHEN “01” => x <= b OR c
WHEN “10” => x <= b NAND c
WHEN “11” => x <= b NOR c
END CASE;
END PROCESS;

FIGURA 3.8 – Exemplo da utilização do CASE

3.5.4 Laços de repetição

LOOP-FOR

Este laço não é condicional, ele possui um número de iterações fixo (fig. 3.9).

Exemplo:

FOR i IN 0 TO 4 LOOP
IF (a = b) THEN
c (i) := ‘1’;
END IF;
END LOOP;

FIGURA 3.9 – Exemplo da utilização do LOOP-FOR

No exemplo acima, o número de iterações seria cinco (5).

LOOP-WHILE

Este é laço condicional, isto é, a execução permanece no corpo do laço enquanto a condição for verdadeira (fig. 3.10), diferente do LOOP-FOR que possui um número pré-determinado de iterações.

Exemplo:

```
WHILE status = '1' LOOP
        Conta := conta + 1;
        IF (conta = 16) THEN
                status := '0';
        END IF;
END LOOP;
```

FIGURA 3.10 – Exemplo da utilização do LOOP-WHILE

3.5.5 Outros comandos

WAIT

Este comando causa a suspensão de um processo ou um procedimento. Há quatro formas de utilização deste comando:

- WAIT UNTIL <*condição*>
  WAIT UNTIL status = '1'; o processo fica suspenso até que status seja igual a 1.
- WAIT ON <lista de sinais>
  WAIT ON a, b, c; suspende até que a, b e c sejam iguais a 1.
- WAIT FOR <tempo>
  WAIT FOR 50 ns; suspende a operação por 50 nanosegundos.
- WAIT
  WAIT; suspende uma operação.

EXIT

É utilizado para encerrar um *loop*. Também pode ser utilizado combinado com a cláusula "WHEN" e, neste caso, só é executado se a condição for verdadeira.

Exemplo:

```
EXIT label_loop WHEN a = b;
```

NULL

Este é freqüentemente utilizado em segmentos "CASE". Algumas vezes deseja-se que certas opções não realizem uma ação e para este propósito utiliza-se o "NULL". Pode-se dizer que ele é semelhante ao "*null*" da linguagem C.

ASSERT

Este seguimento é utilizado para fins de simulação. Uma condição é testada durante a simulação e se não for válida, uma mensagem é enviada a um dispositivo de saída. Este segmento apresenta-se, conforme [LIP 90], como no modelo abaixo:

ASSERT <*condição*>
REPORT < *mensagem*>
SEVERITY <*nível*>

O "*nível*" utilizado na cláusula "SEVERITY" é do tipo "SEVERITY_LEVEL", tipo este que pertence ao pacote padrão.

3.6 Exemplo

A seguir será apresentado um exemplo (fig. 3.11), retirado da coleção que acompanha a ferramenta MAX+PLUS II, que descreve uma máquina de estados.

Nota-se que há quatro pinos de entrada/saída, definidos no seguimento "entity". Há um tipo que define os estados da máquina ("state_type"). O processo define o comportamento da máquina de estados.

```
ENTITY statmach IS
        PORT(
                clk     : IN    BIT;
                input   : IN    BIT;
                reset   : IN    BIT;
                output  : OUT   BIT );
END statmach;

ARCHITECTURE a OF statmach IS
        TYPE STATE_TYPE IS (s0, s1);
        SIGNAL state    : STATE_TYPE;
BEGIN
        PROCESS (clk)
        BEGIN
                IF reset = '1' THEN
                        state <= s0;
                ELSIF (clk'EVENT AND clk = '1') THEN
                        CASE state IS
                                WHEN s0=>
                                        state <= s1;
                                WHEN s1=>
                                        IF input = '1' THEN
                                                state <= s0;
                                        ELSE
                                                state <= s1;
                                        END IF;
                        END CASE;
                END IF;
        END PROCESS;
        output <= '1' WHEN state = s1 ELSE '0';
END a;
```

FIGURA 3.11 – Exemplo de um componente descrito em VHDL.

4. A Ferramenta MAX+PLUS II

Este capítulo tem por objetivo apresentar o estudo realizado sobre o software MAX+PLUS II, que é uma ferramenta CAD, desenvolvido pela *Altera Corporation*. Com o MAX+PLUS II podem ser criados projetos eletrônicos de circuitos integrados para uma determinada aplicação.

A abordagem inicial, será desenvolvida de forma a mostrar os aspectos mais relevantes para a utilização desta ferramenta, colocando alguns itens de maior importância para o desenvolvimento deste projeto, são eles: entrada de dados, capacidade de processamento e poder de simulação que proporciona, o que a caracteriza como indispensável para o trabalho de testes.

4.1 Aplicações da ferramenta

Segundo [MAX 97], o Altera Multiple Array MatriX Programmable Logic User System (MAX+PLUS II) é um sistema multiplataforma, que provê uma arquitetura independente, podendo ser facilmente adaptado a aplicações diversas. Além disso a ferramenta oferece uma entrada de dados fácil, um processamento rápido e uma programação direta de dispositivos, características que a torna adequada ao trabalho previsto.

O MAX+PLUS II provê uma variedade de métodos de entrada para o desenvolvimento de projetos, grande capacidade de síntese lógica, simulação funcional, simulação ligada a multi-dispositivos, análise temporal, localização automática de erros e dispositivos de programação e verificação. O sistema contém onze aplicações completamente integradas que facilitam no processo de criação do projeto, já que evitam erros decorrentes de tarefas que elas substituem. A facilidade de uso que a ferramenta oferece por ser totalmente integrada possibilita que seja efetuada a simulação dos componentes do sistema de maneira mais simplificada que utilizando outros recursos.

4.2 Entrada de dados

O software possui vários mecanismos para entrada de dados, facilitando e agilizando o projeto. A entrada pode ser: por meio de esquemas, construídos através do editor gráfico do software, ou importar projetos via Orcad; por meio textual, utilizando a linguagem da Altera (AHDL) ou o VHDL; por meio de formas de onda; e ainda permite que se importe qualquer arquivo padrão EDIF.

A possibilidade de entrar com dados utilizando a linguagem VHDL, que foi a escolhida pelas necessidades que o sistema apresenta, foi fator relevante para sua utilização no trabalho, pois possibilitou o desfrute das vantagens desta ferramenta sem abrir mão da flexibilidade oferecida pela linguagem citada. Além disso, torna viável que seja utilizado os

códigos gerados em VHDL, no editor gráfico, através de simbologias, o que facilita a visualização do projeto como um todo e dispensa o trabalho de criar módulos escritos para ligar diversos componentes.

Abaixo (tab. 4.1) tem-se uma descrição das principais extensões dos arquivos de entrada de dados:

TABELA 4.1 – Extensões de arquivos de entrada de dados do MAX+PLUS II.

Extensão	Descrição
GDF	(graphic design file) – Arquivo gráfico gerado a partir do editor gráfico
SCH	(Orcad schematic file) – Arquivo gráfico gerado a partir do Orcad
WDF	(waveform design file) – Arquivo de forma de ondas
TDF	(text design file) - Arquivo texto descrito em AHDL
VHD	(VHDL design file) - Arquivo texto descrito em VHDL
EDF	(Edif input file) – Arquivo texto, formato EDIF 290 ou 300, importado de outras ferramentas CAE
XNF	(Xilinx Netilist Format file) – Arquivo texto, formato Xilinx
ADF	(Altera design File) - Arquivo texto, gerado pelo software Aplus da Altera
SMF	(State Machine Files) – Arquivo texto, gerado pelo software SAM+PLUS da Altera

4.3 Descrição dos aplicativos

O MAX+PLUS II contém módulos que são gerenciados pelo MAX+PLUS II Manager. Estes módulos são todos interligados e viabilizam desde a descrição dos componentes, usando principalmente o editor de textos para construção de código VHDL e também do editor gráfico para a visão geral do trabalho executado, até a simulação do projeto como um todo.

Estes aplicativos que o constituem são apresentados nos itens a seguir.

4.3.1 Visualizador hierárquico (Hierarchy Display)

Este aplicativo apresenta a hierarquia de arquivos através de árvores hierárquicas com ramos representando subprojetos, que são componentes agregados a um projeto específico. Possibilita que sejam abertos a partir deste aplicativo, arquivos de entrada, tais como: projetos esquemáticos, projetos textuais (AHDL ou VHDL), projetos de formas de ondas (*waveform*) e também arquivos gerados pelo próprio compilador (fig. 4.1).

Sua a utilização torna possível a visualização da estrutura do projeto, facilitando a compreensão de dependências entre os módulos que o compõe. Partindo-se de uma visão geral do projeto pode-se chegar a detalhamentos deste, com acesso direto às fontes de criação dos módulos.

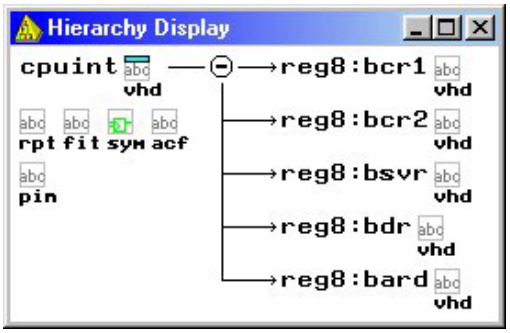

FIGURA 4.1 – MAX+PLUS II Hierarchy Display.

4.3.2 Editor gráfico (Graphic Editor)

Com o editor gráfico podem ser gerados esquemas a partir de componentes lógicos para a construção de um projeto (fig. 4.2). A ferramenta possui uma série de componentes implementados que podem ser utilizados em qualquer espécie de aplicação, enquadrando-se bem ao perfil do trabalho pretendido que utiliza alguns dos componentes já implementados, sendo que a descrição do comportamento e pinagem destes componentes estão disponíveis na documentação oferecida pelo software. O aplicativo ainda proporciona a utilização de símbolos configuráveis as necessidades de uso, podendo também gerar símbolos a partir de módulos escritos tanto em AHDL como em VHDL, recurso muito utilizado para a execução deste projeto, visto que o mesmo apresenta a necessidade de unir alguns módulos implementados, trabalho este que torna-se bem menos árduo com a utilização dos esquemas gerados.

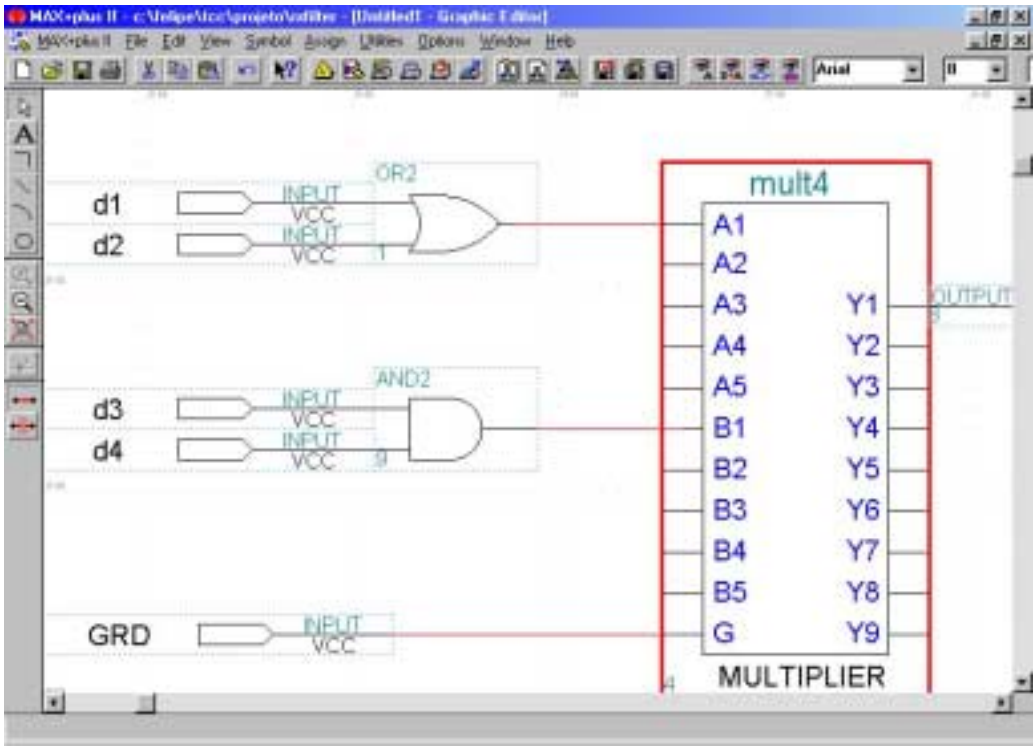

FIGURA 4.2 – MAX+PLUS II Graphic Editor.

Convém salientar que a partir de um arquivo esquemático, pode-se gerar um arquivo de saída para ser utilizado por simuladores da ferramenta. Por oferecer esta

possibilidade é que a ferramenta foi escolhida, neste trabalho, para criar projetos esquemáticos provenientes de módulos implementados utilizando a linguagem VHDL.

4.3.3 Editor de símbolos (Symbol Editor)

O editor de símbolos torna possível a visualização, criação e edição de símbolos que representam um circuito lógico, sendo útil para a elaboração do projeto esquemático (fig. 4.3). Estes símbolos podem ser criados a partir de um projeto, isto é, por uma descrição de hardware, no trabalho em questão todo e qualquer símbolo gerado tem sua origem nas descrições feitas em VHDL. O compilador gera automaticamente estes símbolos, a partir dos arquivos de entrada, permitindo a edição dos mesmos para melhor adapta-los a projetos esquemáticos.

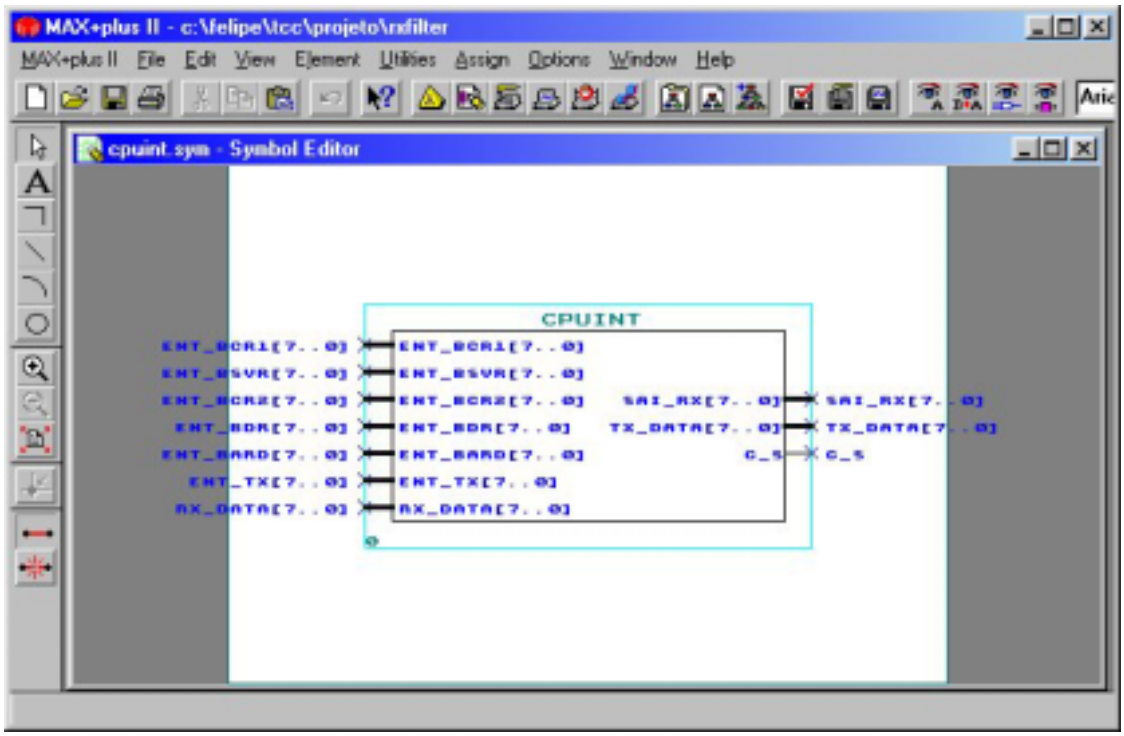

FIGURA 4.3 – MAX+PLUS II Symbol Editor.

4.3.4 Editor de texto (Text Editor)

Neste editor podem ser criados e editados arquivos textos baseados em projetos lógicos escritos em AHDL, VHDL e Verilog HDL. Com este editor ainda podem ser visualizados e editados, arquivos ASCII criados pelo MAX+PLUS II após uma compilação (fig. 4.4). O aplicativo traz facilidades de editoração, pois possui um sistema de ajuda integrado, sintaxe colorida e esquemas de blocos (*templates*) em AHDL, VHDL e Verilog HDL.

O editor de textos da ferramenta tem grande valor no trabalho que é proposto aqui, visto o fato de que todos os módulos implementados utilizam-se de uma descrição comportamental elaborada em VHDL.

```
cpuint.vhd - Text Editor
-- Modulo "CPU INTERFACE"  do BDLC
package types is
    subtype bit_8 is bit_vector (7 downto 0);
    component reg8
        port (  d : in bit_vector (7 downto 0);
                clk : in bit;
                q : out bit_vector (7 downto 0)
                );
    end component;
end types;

use work.types.all;
entity cpuint is
    port (  clk : in bit;
            -- Entrada de dados dos Registradores
            ent_bcr1 : in bit_8;
            ent_bsvr : in bit_8;
            ent_bcr2 : in bit_8;
            ent_bdr  : in bit_8;
            ent_bard : in bit_8;
Line  21   Col  1   INS
```

FIGURA 4.4 – MAX+PLUS II Text Editor.

4.3.5 Editor de formas de onda (Waveform Editor)

A função deste editor é criar descrições de formas de onda, baseado-se em um projeto lógico. As descrições são canais de entrada e/ou saída que viabilizam as simulações. Um projeto em forma de onda é criado especificando combinações de níveis de entradas lógicas e especificando as saídas de forma gráfica (fig. 4.5), com esta habilidade, que o MAX+PLUS II disponibiliza, torna-se mais fácil a tarefa de execução de testes, oferecendo ainda a vantagem de possibilitar uma visualização gráfica dos resultados que são obtidos.

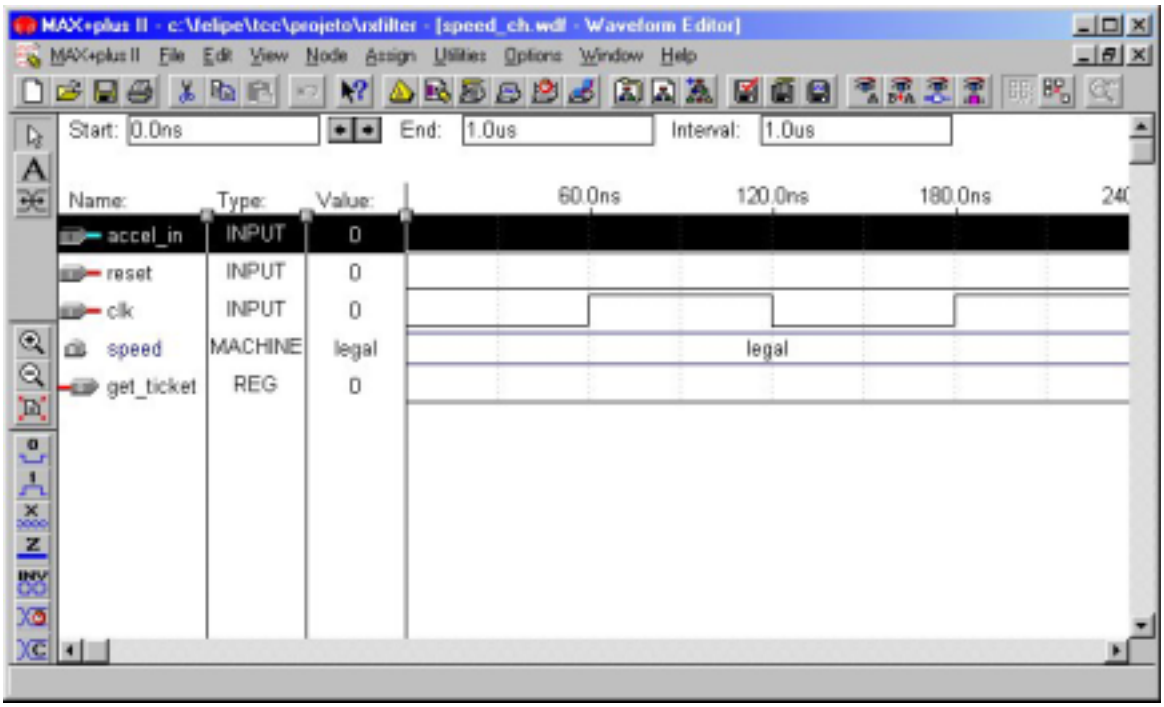

FIGURA 4.5 – MAX+PLUS II Waveform Editor.

4.3.6 Editor f*loorplan* (Floorplan Editor)

É um ambiente gráfico que permite a configuração dos pinos de um dispositivo físico e recursos de células lógicas. A configuração destes pinos faz-se necessária, quando se deseja direcionar um projeto a um dispositivo programável específico. Pode-se editar a localização dos pinos visualizando um dispositivo e designar sinais para células lógicas individuais em uma visão detalhada de um bloco lógico (fig. 4.6).

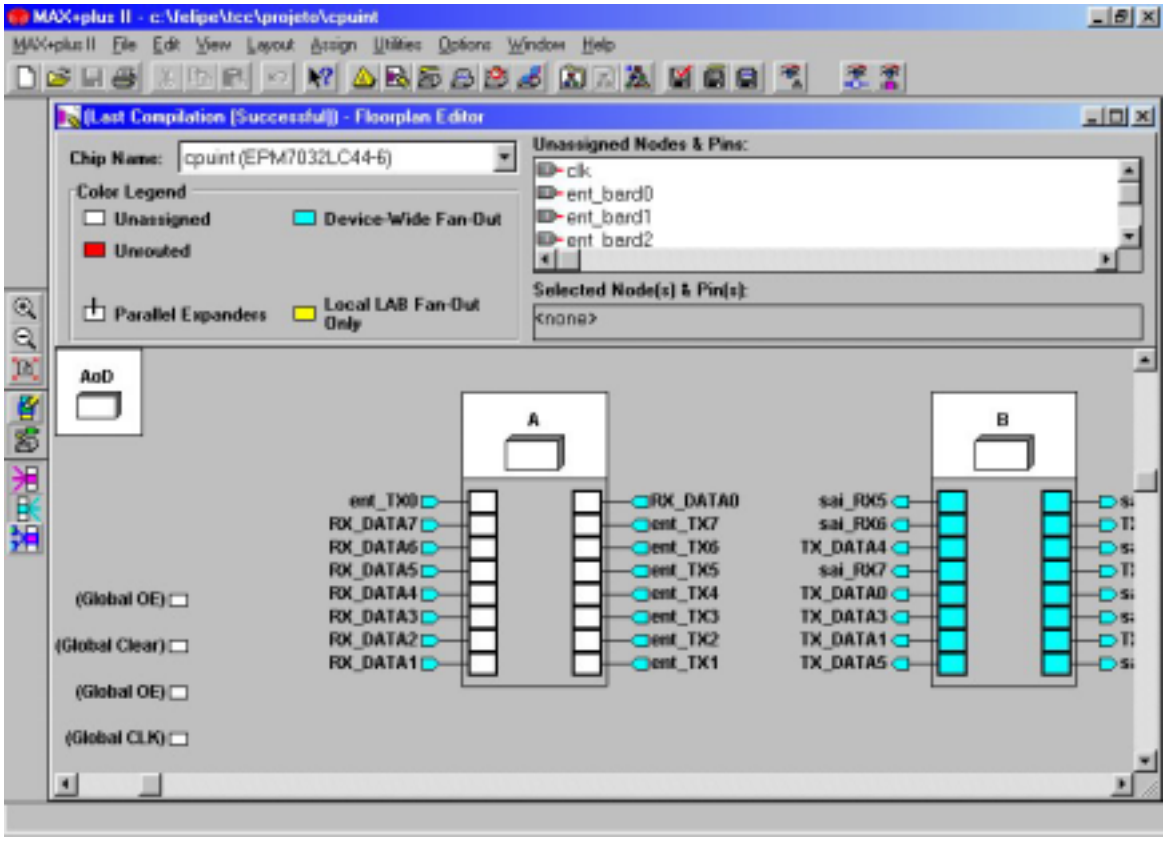

FIGURA 4.6 – MAX+PLUS II Floorplan Editor.

4.3.7 Compilador (Compiler)

O compilador é responsável pela ligação dos aplicativos de entrada – os editores já apresentados – aos aplicativos de simulação, sendo eles o analisador temporal, o simulador e o programador de dispositivos (fig. 4.7). Este utilitário possui uma série de processos que detectam os erros no projeto, fazem a síntese lógica, formatam o projeto para que possa ser colocado nos dispositivos, em suma sintetiza e otimiza o circuito lógico, gerando arquivos para simulação, análise temporal e programação de dispositivos.

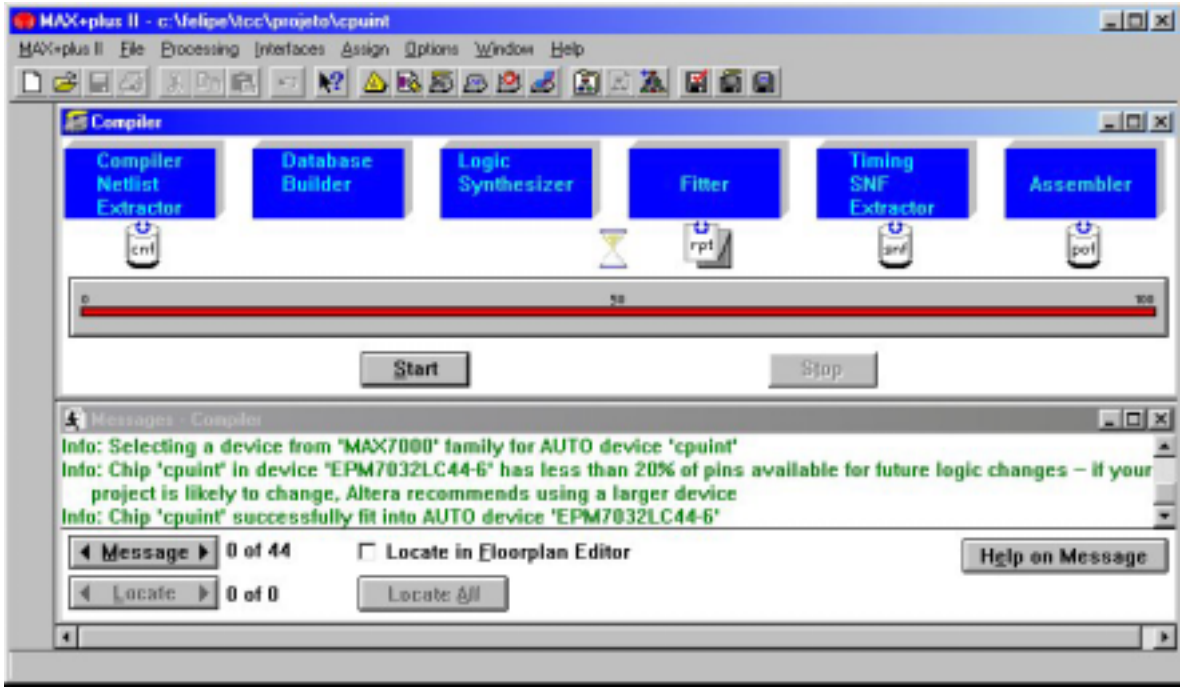

FIGURA 4.7 – MAX+PLUS II Compiler.

4.3.8 Simulador (Simulator)

O aplicativo simulador faz testes sobre a lógica operacional e a temporização interna de um circuito lógico. São disponibilizadas as simulações funcionais, temporais e a simulação ligada a multi-dispositivos, tornando possível uma visão do andamento e funcionamento global do trabalho (fig. 4.8).

A partir deste utilitário, torna-se possível uma análise sobre a viabilidade de um projeto, levando em conta uma análise temporal sobre o circuito implementado.

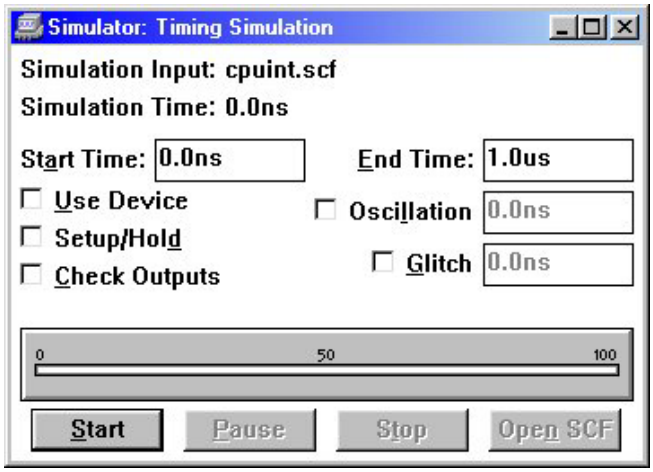

FIGURA 4.8 – MAX+PLUS II Simulator.

4.3.9 Analisador temporal (Timing Analyzer)

É um utilitário que analisa a performance de um circuito lógico após ele ter sido sintetizado e otimizado pelo compilador (fig. 4.9), a função básica dele é oferecer subsídios para que se tenha uma visão crítica do projeto e possa com isso rever o trabalho em pontos onde a performance de velocidade não está conforme o esperado, pois possibilita que sejam

traçados todos os caminhos de sinais em um projeto, determinando a velocidade de caminhos críticos e limitá-los para uma melhor performance.

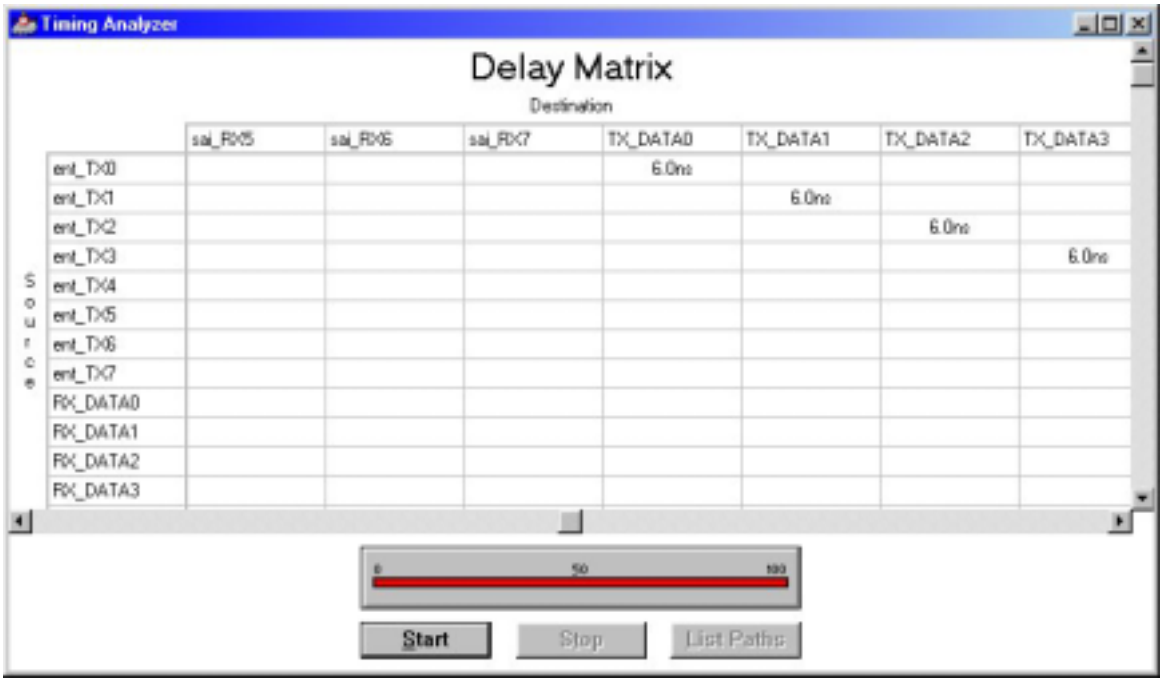

FIGURA 4.9 – MAX+PLUS II Timing Analyzer.

4.3.10 Programador de dispositivos (Programer)

No programador de dispositivos são utilizados os programas gerados pelo compilador para utilizar os dispositivos Altera (fig. 4.10), habilitando os testes das estruturas implementadas neste trabalho. Ele possibilita que se programe, configure, verifique, examine e teste a funcionalidade dos dispositivos implementados, permite assim que se tomem conclusões sobre o projeto quanto a sua viabilidade.

FIGURA 4.10 – MAX+PLUS II Programer.

4.3.11 Processador de mensagens (Message Processor)

O aplicativo mostra mensagens indicando erros, alertas e informações sobre o estado do projeto e ainda localiza a origem da mensagem automaticamente, com isso facilita o trabalho de encontrar o problema, tornando mais rápido o processo de depuração. A origem das mensagens pode ser os arquivos de entrada.

5. Implementação do BDLC

Este capítulo visa apresentar a implementação do BDLC, abordando as principais etapas no desenvolvimento de um projeto. Procura-se fazer, a partir da modelagem de alto nível do projeto apresentada no segundo capítulo, a composição estrutural do circuito. A partir da modelagem faz-se as especificações funcionais dos componentes do projeto e, logo após, aplica-se processos para sintetizar e testar o protótipo implementado. A última etapa seria a representação física do componente.

5.1 Composição estrutural

As entidades do projeto podem conter uma descrição comportamental (arquitetura comportamental) e/ou uma descrição estrutural (arquitetura estrutural). Na arquitetura comportamental define-se a funcionalidade de uma entidade (componente), já na arquitetura estrutural referencia-se os componentes utilizados por uma entidade.

A seguir serão explanadas as estruturas dos blocos que compõe o BDLC.

5.1.1 Interface da CPU

O bloco de interface da CPU é representado pela entidade CPUINT. A arquitetura estrutural desta entidade, referencia cinco outros componentes (registradores de 8 bits - REG8) que são instanciados dentro de um bloco de arquitetura, isto é, tratando-se de VHDL as referências a componentes são feitas em blocos *architecture*. Estes registradores referenciados são entidades que já foram definidas e, para que possam ser instanciadas, deve-se declará-las como um componente (cláusula *component*) na própria arquitetura em que é usada ou em um pacote. No caso do trabalho, o componente REG8 foi declarado em um pacote (*package componentes*), definido junto a entidade CPUINT, visto que o mesmo é utilizado por outras arquiteturas de componentes do BDLC.

A implementação estrutural dos componentes REG8 (fig. 5.1) é simples, pois tratam-se de registradores ativados por borda alta de relógio, e portanto a pinagem é simplificada, onde CLK é um sinal de relógio, neste caso é o mesmo sinal provido à CPU. A entrada de dados é feita pela porta denominada D e a saída é denominada como Q. Tanto a entrada como a saída de dados é feita de forma paralela (8 bits), isto é, os dados são palavras de 1 byte.

FIGURA 5.1 – Símbolo do registrador de 8 bits.

A entidade CPUINT (fig. 5.2) foi definida a partir da descrição apresentada no capítulo anterior. Os principais conjuntos de pinos deste componente são denominados: CPU_TX, que está ligado a CPU e é um canal para transmitir as mensagens; CPU_RX, também está ligado a CPU e é o canal de recepção de mensagens; TX_DATA, faz a conexão com o módulo manipulador de protocolo, levando a mensagem a ser transmitida; RX_DATA, recebe as mensagens do manipulador de protocolo, passando-as ao CPU_RX; C_S, este é um canal bidirecional que provê informações de controle sobre o BDLC, isto é, define o *status* do MCU. Todos os dados que trafegam por estas portas são disponibilizados de forma paralela, ou seja, a declaração é constituída por um vetor de 8 bits utilizando o tipo de lógica padrão (*std_logic_vector*) definido pelo IEEE. Este tipo foi adotado para tornar os componentes aqui implementados compatíveis aos componentes parametrizados da ferramenta MAX+PLUS II.

Além das portas citadas anteriormente, existem os pinos para entrada de dados nos registradores de controle do BDLC. Para cada registrador, BCR1, BCR2 e BARD, existem bits de configuração que devem ser escritos pelo usuário do MCU. No BCR1 os bits com acesso externo são IMSG, CLKS, RS[1:0] e WCM. Para prover este acesso definiu-se as portas *bcr1*_imsg, *bcr1_clks* e *bcr1_wcm* como entradas do tipo *std_logic* (apenas um bit em cada porta) e a porta *bcr1_rs* que é uma entrada do tipo *std_logic_vector (1 downto 0)*, ou seja, é um conjunto de dois bits. O registrador BCR2 possui apenas um bit de configuração externo, NBFS, definido pela porta *bcr2_nbfs* do tipo *std_logic*. O BARD possui portas referentes aos bits ATE e RXPOL e ao campo BO[3:0].Para os bits ATE e RXPOL são especificadas, respectivamente, *bard_ate* e *bard_rxpol* como entradas do tipo *std_logic*. O campo BO[3:0] é definido por uma porta de entrada do tipo *std_logic_vector (3 downto 0)*. Os registradores BDR e BSVR são para uso interno do BDLC, e portanto não tem comunicação externa.

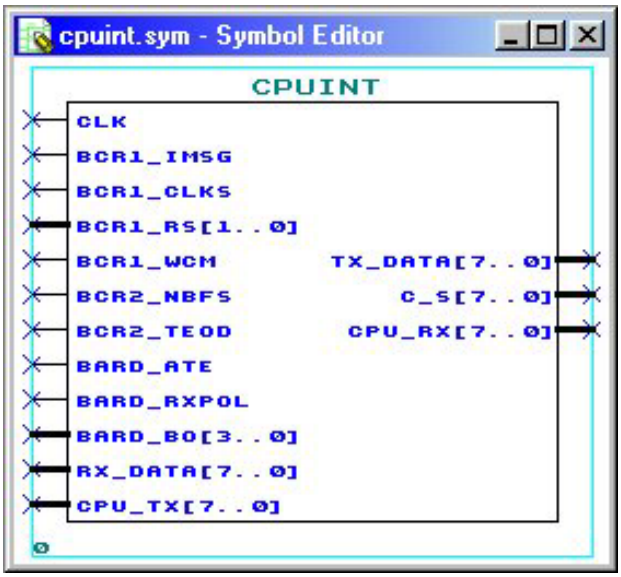

FIGURA 5.2 – Símbolo da entidade CPUINT.

5.1.2 Manipulador de protocolo

O módulo manipulador de protocolo do BDLC, representado pela entidade PROTOCHA, é composto por dois tipos de entidades e uma função genérica (*lpm_shiftreg*), que faz parte de uma biblioteca de funções oferecida pelo MAX+PLUS II. Uma das entidades é a REG8, instanciada para representar os registradores shadow RX e shadow TX.

A outra é a PSTATMAC, que consiste na maquina de estados de protocolo do BLDC. A função LPM_SHIFTREG é facilmente adaptada para utilização neste trabalho, portanto foi empregada para representar os registradores de deslocamento RX e TX.

O projeto esquemático abaixo (fig. 5.3) representa a integração dos componentes da entidade PROTOCHA, para elaborá-lo foi utilizando o editor gráfico do MAX+PLUS II.

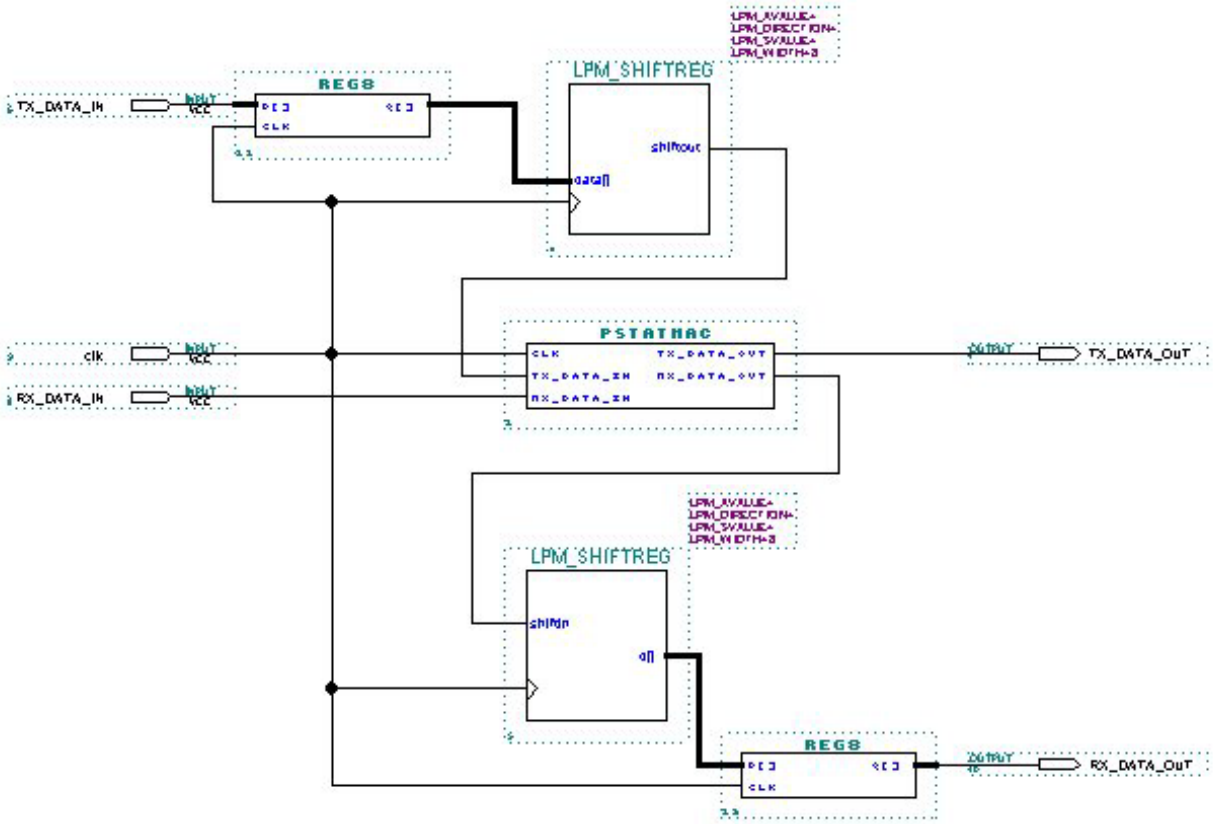

FIGURA 5.3 – Esquema do bloco manipulador de protocolo.

A estrutura da entidade REG8 já foi descrita no item anterior. Para ser utilizada neste módulo, é necessário que se faça referência ao pacote *componentes* utilizando a cláusula **use**, visto que o componente REG8 faz parte do mesmo.

Outro componente de extrema importância é a máquina de estados de protocolo, que é representada pela entidade PSTATMAC (fig. 5.4). Este componente possui seis definições de portas. Uma delas é a porta CLK, que recebe um sinal de relógio. Há mais duas portas de entrada, TX_DATA_IN e RX_DATA_IN. A porta TX_DATA_IN recebe sinais à serem transmitidos do registrador de deslocamento TX (TX_shift). Já na porta RX_DATA_IN, trafegam os sinais recebidos pelo barramento. Além das portas de entrada há duas portas de saída, TX_DATA_OUT, que passa os sinais que devem ser transmitidos ao barramento, e RX_DATA_OUT, que passa os sinais recebidos para o registrador de deslocamento RX (RX_shitf). Finalmente, tem-se a porta C_S, que é definida como sendo de entrada e saída (*inout*), cuja finalidade é prover informações para controle de *status* do BDLC.

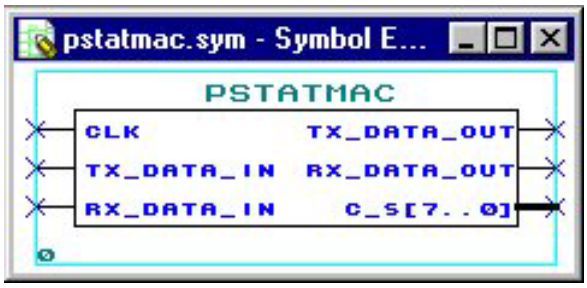

FIGURA 5.4 – Símbolo da entidade PSTATMAC.

Por fim, faz-se uso da função parametrizada *lpm_shiftreg*, que representa um registrador de deslocamento configurável, isto é, a pinagem do componente é definida de acordo com a aplicação. Neste trabalho a função é utilizada para definir dois componentes, TX_shift e RX_shift, possuindo uma configuração distinta, realizada de acordo com os parâmetros passados à mesma. Dentre estes parâmetros há definições gerais e definições quanto a pinagem utilizada. As definições gerais são feitas através da cláusula *generic map*. No caso desta função, define-se por exemplo o tamanho da entrada ou saída de dados de forma paralela (LPM_WIDTH), ou ainda a direção de deslocamento dos bits (LPM_DIRECTION). Nesta implementação, utilizou-se um único parâmetro geral, LPM_WIDTH = 8, para a configuração de ambos registradores, visto que os mesmos só recebem ou disponibilizam palavras de 8 bits.

A definição dos pinos empregados é realizada por referências as portas definidas na função, utilizando-se a cláusula *port map*. Neste caso, as configurações dos registradores são diferentes; o TX_shift, segundo [BDL 97], deve receber dados do registrador shadow TX na forma paralela (8 bits) e disponibilizá-los de forma serial, para tanto são utilizadas as portas, CLOCK, que recebe sinal de relógio, DATA[LPM_WIDTH – 1..0], que recebe os dados na forma paralela, e SHIFTOUT, que retorna os bits serializados. No caso do RX_shift, os dados são recebidos na forma serial e devem ser disponibilizados na forma paralela. Portanto, a porta de entrada de dados é a SHIFTIN, a de saída é a Q[LPM_WIDTH-1..0], e além destas, utiliza-se também a porta CLOCK.

5.1.3 Interface do MUX

A interface do MUX é definida pela entidade MUXINT (fig. 5.5). Este módulo possui as mesmas definições de portas da entidade PSTATMAC, ou seja, as entradas CLK, TX_DATA_IN e RX_DATA_IN, as saídas TX_DATA_OUT e RX_DATA_OUT, e ainda a porta bidirecional C_S.

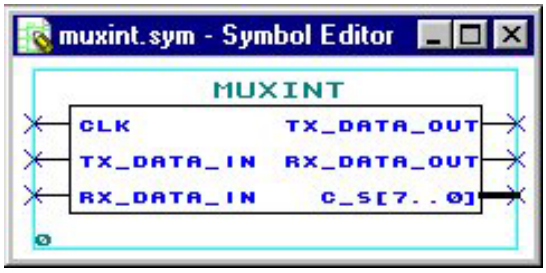

FIGURA 5.5 – Símbolo da entidade MUXINT.

A arquitetura estrutural da entidade MUXINT é composta por três componentes. Um destes componentes é o S_ENCDEC, que consiste no codificador e decodificador de

símbolos. Outro seria o filtro digital RX, representado pela entidade RXDIGFIL. E ainda um multiplexador, que é implementado por uma macrofunção da biblioteca *altera*, a_*21mux*.

A entidade S_ENCDEC (fig. 5.6) possui as mesmas especificações de portas definidas na entidade MUXINT que encapsula este componente. A única porta que não está diretamente ligada aos sinais provenientes das portas da entidade MUXINT é a RX_DATA_IN, que recebe o sinal de saída do RXDIGFIL (fig. 5.7).

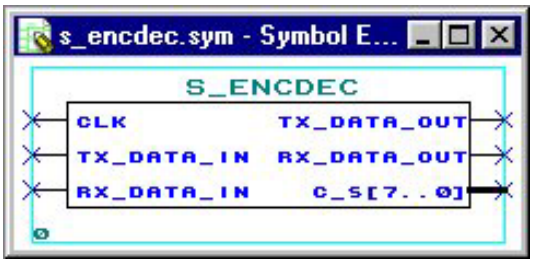

FIGURA 5.6 – Símbolo da entidade S_ENCDEC.

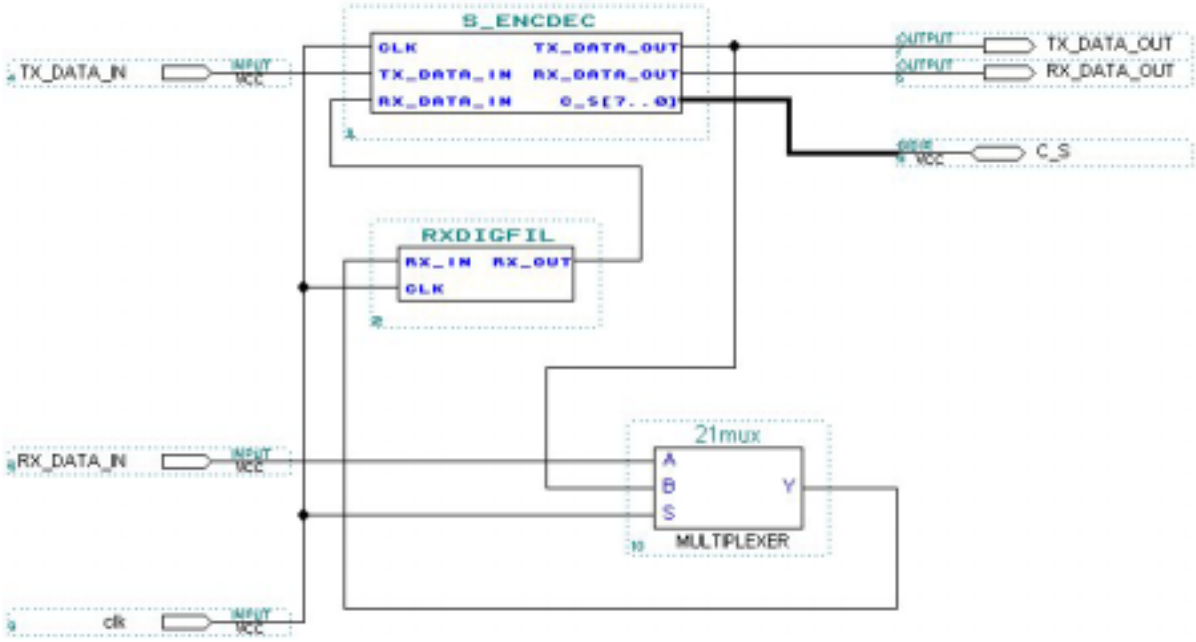

FIGURA 5.7 – Estrutura da entidade MUXINT.

O RXDIGFIL possui uma definição de portas simplificada (fig. 5.8), porém sua estrutura é mais complexa que os demais componentes do bloco de interface do MUX. A declaração das portas é composta por um sinal de relógio, CLK, um sinal de entrada a ser filtrado, RX_IN, e um sinal resultante do processo de filtragem, RX_OUT.

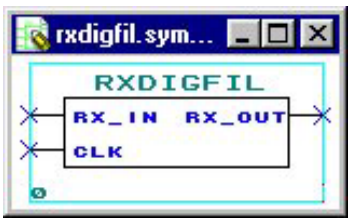

FIGURA 5.8 – Símbolo da entidade RXDIGFIL.

A arquitetura estrutural do RXDIGFIL (fig. 5.9) é composta por três componentes: um *flip-flop*, um contador de 4 bits e um *latch*. Estes componentes são definidos utilizando as funções parametrizadas do MAX+PLUS II. Como parâmetro geral emprega-se somente o LPM_WIDTH = 1, definindo assim o tamanho das entradas e saídas de dados dos mesmos. No *flip-flop* (*lpm_dff*) são utilizadas as portas de entrada de dados, DATA, de saída de dados, Q, e a porta CLOCK, que recebe um sinal de relógio. O contador (*lpm_counter*) possui a mesma pinagem do *flip-flop* (DATA, Q e CLOCK) e um pino adicional (UPDOWN), que determina se o contador deve incrementar ou decrementar seu valor. As portas que compõem o *latch* (*lpm_latch*) são as de entrada e saída de dados (DATA e Q) e a porta GATE, que habilita o componente para recepção dos dados.

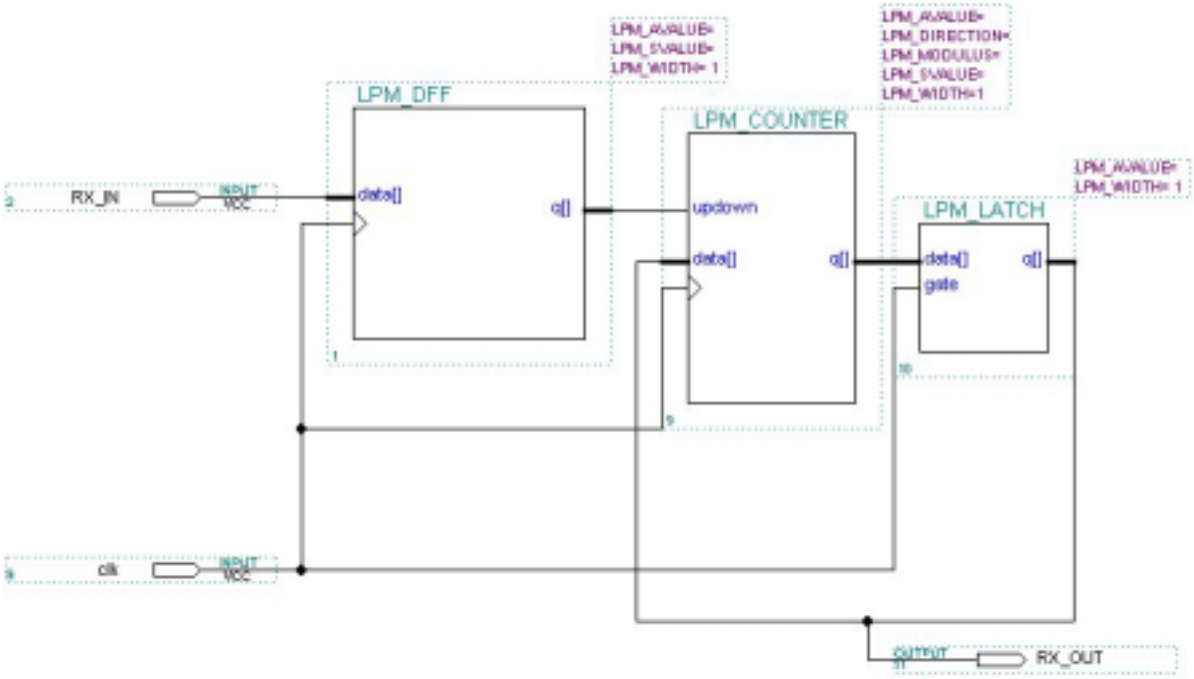

FIGURA 5.9 – Estrutura do RXDIGIFL.

Outro componente da estrutura do MUXINT é multiplexador, que nesta implementação é representado pela macrofunção *a_21mux*. Este consiste de um multiplexador de dois sinais recebidos pelas portas definidas como A e B. Outra porta especificada na função é a S, que faz a seleção dos sinais recebidos. A saída é determinada pela porta Y.

5.1.4 Entidade BDLC

A entidade BDLC foi criada utilizando o editor gráfico do MAX+PLUS II, a fim de prover a integração dos módulos implementados. Sua estrutura pode ser observada na figura (fig. 5.10) a seguir.

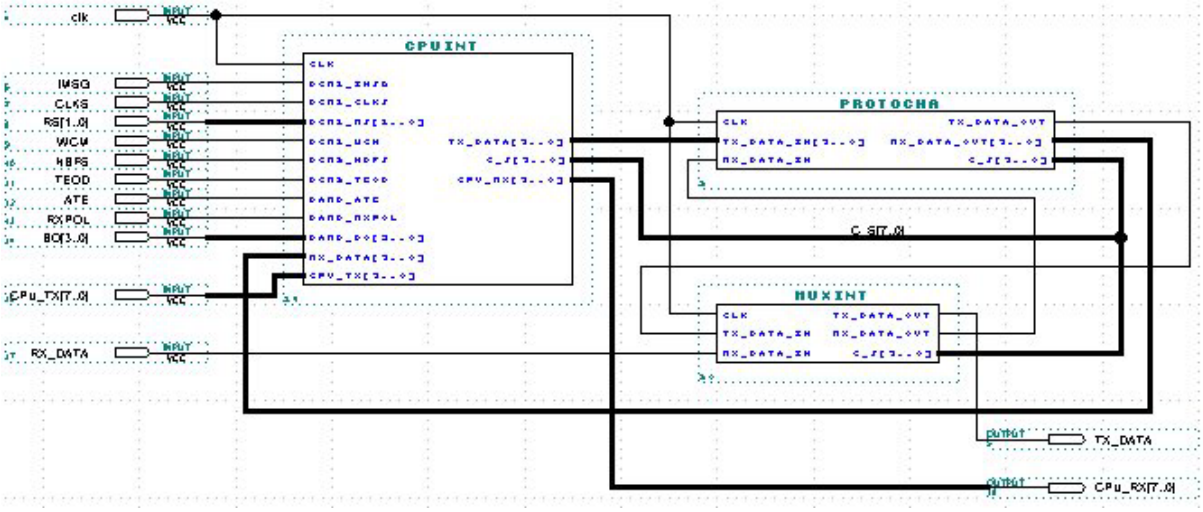

FIGURA 5.10 – Estrutura da entidade BDLC.

Posteriormente, após um processo de compilação, constatou-se que foram atingidos os objetivos em relação ao modelo construído, pois o diagrama hierárquico (fig. 5.11) gerado pelo compilador do MAX+PLUS II condiz com o modelo hierárquico proposto para a construção do BDLC.

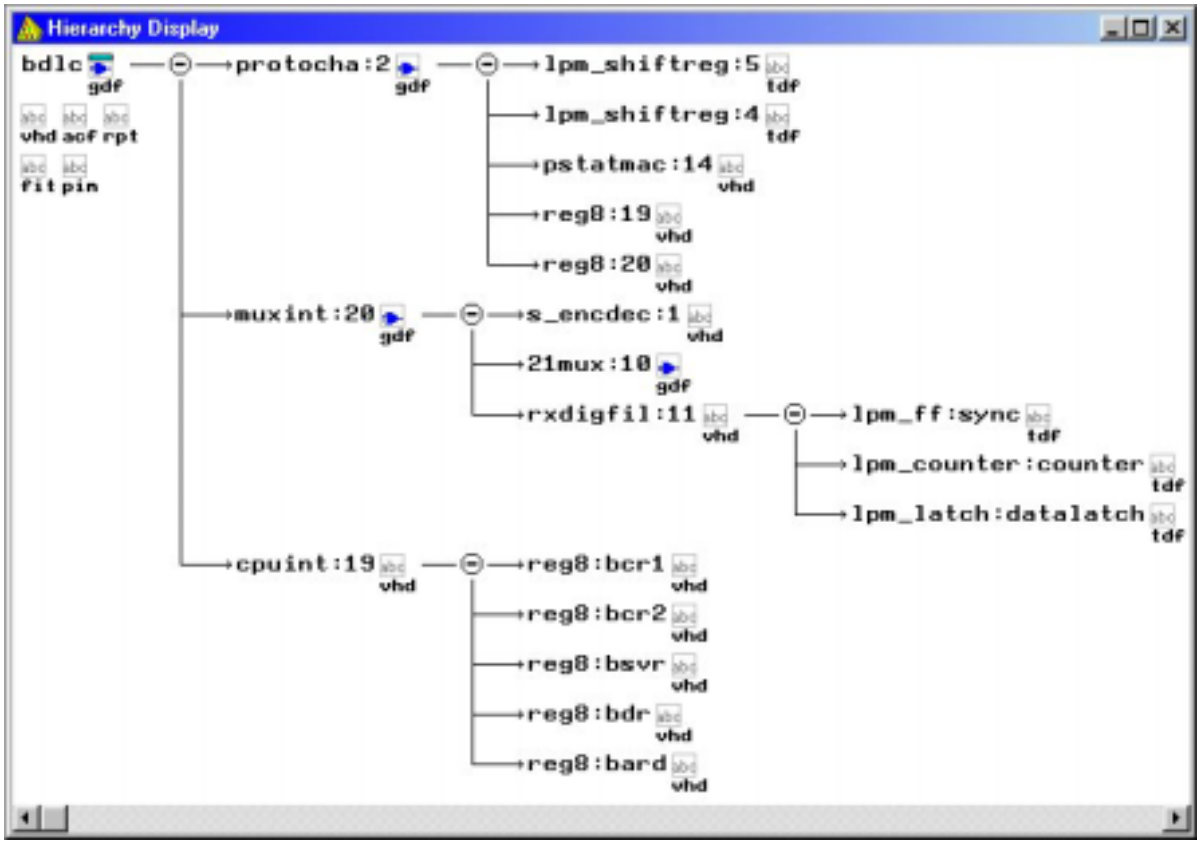

FIGURA 5.11 – Hierarquia do BDLC.

5.2 Especificações funcionais

A segunda etapa do ciclo consiste das especificações funcionais dos componentes, ou seja, a descrição do comportamento de cada entidade agregada ao sistema, utilizando os recursos do VHDL. Neste momento se dá o tratamento necessário a cada sinal que é transmitido a uma entidade.

As descrições funcionais não acompanham a totalidade dos componentes (entidades) mencionadas anteriormente. Algumas destas entidades tem apenas a finalidade de fazer a conexão de componentes que já possuem um comportamento descrito, por exemplo as entidades MUXINT e RXDIGFIL que apenas agregam componentes a sua estrutura. As demais entidades levam sua descrição funcional no corpo da arquitetura (*architecture*) na forma de processos.

5.2.1 Entidade REG8

A entidade REG8, que representa um registrador de 8 bits, foi implementada de maneira a disponibilizar a entrada de dados quando a porta CLK recebe um sinal igual a *um*. Para tanto, a o comando *wait* foi utilizado associado à cláusula condicional *until* (wait until='1') e, na seqüência, o sinal de saída Q recebe a entrada D (q <= d).

5.2.2 Entidade CPUINT

Nesta implementação, a entidade que possui os processos de controle geral do BDLC é a CPUINT. Tais processos definem a inicialização do BDLC e o controle de transmissão e recepção de mensagens.

5.2.2.1 Processo de inicialização do BDLC

Para o processo de inicialização são necessários apenas acessos aos registradores de controle do BDLC, podendo ser descrito, segundo [BDL 97], pelo fluxograma (fig. 5.12) mostrado a seguir.

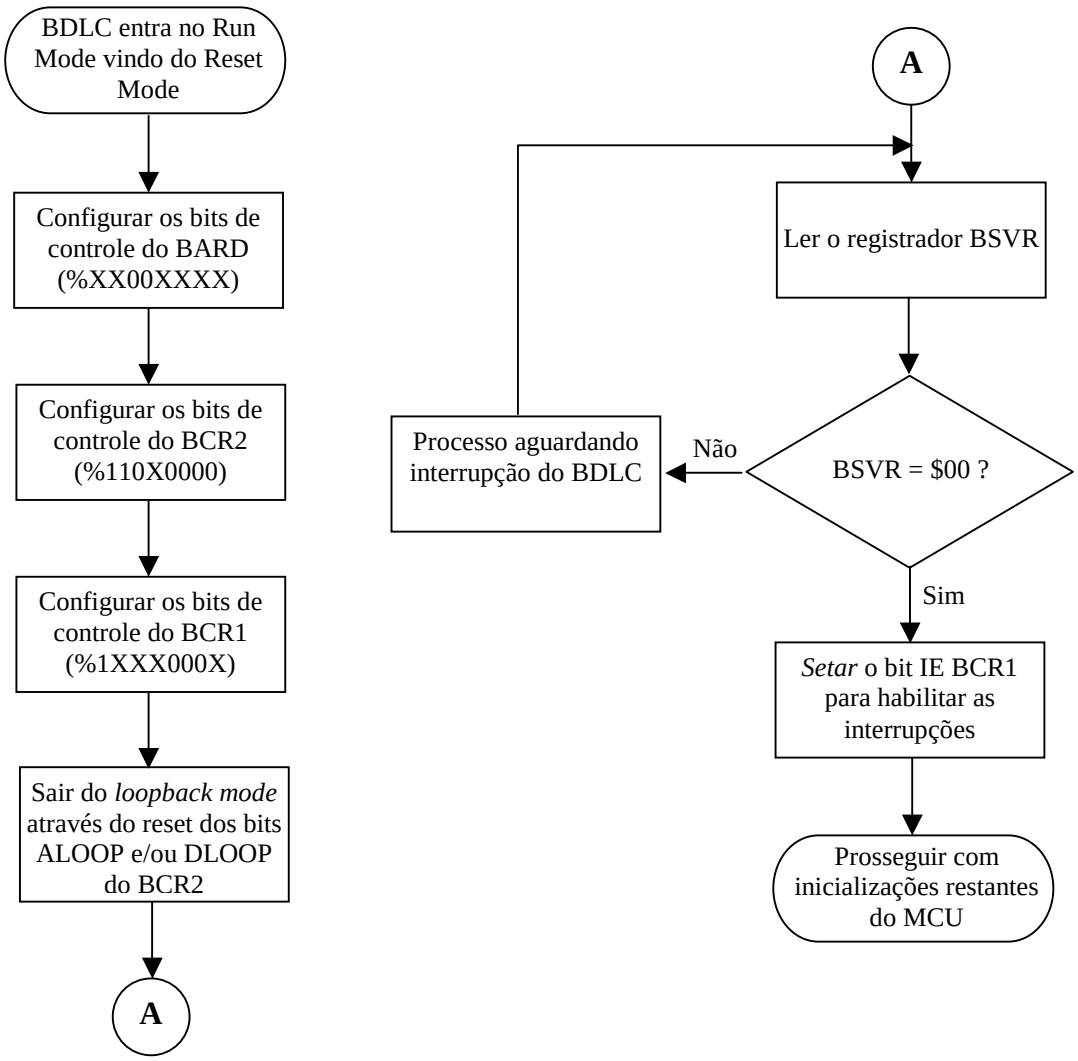

FIGURA 5.12 – Fluxograma da inicialização básica do BDLC.

Inicialmente devem ser escritos os valores desejados nos registradores de controle. Como sinais de entrada para os registradores, foram denominados *ent*_bcr1, *ent_bcr2*, *ent_bard*, *ent_bdr* e *ent_bsvr* do tipo *std_logic_vector (7 downto 0)*, pois a entidade REG8 recebe sinais de 8 bits. Da mesma forma, os sinais de saída dos registradores são denominados q_bcr1, q_*bcr2*, q_*bard*, q_*bdr* e q_*bsvr*. A escrita dos valores nos registradores se dá por estes sinais.

Para facilitar o entendimento de atribuições de valores aos registradores, foram criados *alias* para representar os bits de cada um destes. Os *alias* são relacionados aos sinais de entrada dos registradores e, neste caso, nomeiam um ou mais bits do sinal. A tabela (tab. 5.1) a seguir mostra os sinais relacionados com seus respectivos *alias*.

TABELA 5.1 – Identificação dos bits dos registradores por *alias*.

Sinais de entrada	Alias
ent_bcr1	IMSG : std_logic is ent_bcr1(7);
	CLKS : std_logic is ent_bcr1(6);
	RS : std_logic_vector is ent_bcr1(5 downto 4);
	IE : std_logic is ent_bcr1(1);
	WCM : std_logic is ent_bcr1(0);
ent_bcr2	ALOOP : std_logic is ent_bcr2(7);
	DLOOP : std_logic is ent_bcr2(6);
	RX4XE : std_logic is ent_bcr2(5);
	NBFS : std_logic is ent_bcr2(4);
	TEOD : std_logic is ent_bcr2(3);
	TSIFR : std_logic is ent_bcr2(2);
	TMIFR1 : std_logic is ent_bcr2(1);
	TMIFR0 : std_logic is ent_bcr2(0);
ent_bard	ATE : std_logic is ent_bard(7);
	RXPOL : std_logic is ent_bard(6);
	BO : std_logic_vector is ent_bard(3 downto 0);

O primeiro registrador a ser configurado é BARD, onde os dois primeiros bits resultam das entradas *bard_ate* e *bard_rxpolI* e os quatro últimos da entrada *bard_bo*, sendo que todas pertencem a entidade CPUINT. O acesso do registrador é através do sinal *ent_bard*, logo, este deve portar os bits *bard_ate*, *bard_rxpol* e *bard_bo*. Isto ocorre por atribuições ao sinal *ent_bard* indiretamente, pois utiliza-se os *alias* criados para referenciar a parte do vetor em que os dados devem ser armazenados. A tabela (tab. 5.2) abaixo mostra os valores atribuídos ao sinal *ent_bard*.

TABELA 5.2 – Inicialização do BARD.

Alias do sinal *ent_bard*	Bits de configuração
ATE	Bard_ate
RXPOL	bard_rxpol
ent_bard (5 downto 4)	“00”
BO	bard_bo

O processo de inicialização prossegue com a configuração do BCR2, onde há apenas um bit (NBFS) que é configurado pelo usuário, sendo este sinal proveniente da porta *bcr2_nbfs* definida na entidade CPUINT. Para o BCR2 o sinal de entrada é o *ent_bcr2* e o processo de configuração utiliza as mesmas estratégias adotadas para inicialização do BARD, isto é, faz-se sucessivas atribuições ao sinal de entrada a fim de formar uma palavra de 8 bits. Os valores que compõem o sinal *ent_bcr2* são indicados na tabela abaixo (tab. 5.3).

TABELA 5.3 – Inicialização do BCR2.

Alias do sinal *ent_bcr2*	Bits de configuração
ALOOP	‘1’
DLOOP	‘1’
RX4XE	‘0’
NBFS	bcr2_nbfs

TEOD	‘0’
TSIFR	‘0’
TMIFR1	‘0’
TMIFR0	‘0’

Na seqüência, o próximo registrador a ser inicializado é o BCR1. Este possui quatro bits de fontes externas sendo dois, portas bcr1_clks e bcr1_wcm, do tipo *std_logic* e os outros dois, porta bcr1_rs, um vetor de bits (*std_logic_vector(1 downto 0)*). Da mesma forma que os demais registradores, o sinal de entrada do BCR1 (*ent_bcr1*) é formado por uma seqüência de atribuições, cujos valores são mostrados na tabela abaixo (tab. 5.4).

TABELA 5.4 – Inicialização do BCR1

Alias do sinal *ent_bcr1*	Bits de configuração
IMSG	‘1’
CLKS	bcr1_clks
RS	bcr1_rs
ent_bcr1 (3 downto 2)	“00”
IE	‘0’
WCM	bcr1_wcm

O passo seguinte é a saída do *loopback mode* através do reset do bit ALOOP e/ou do bit DLOOP. Estes bits pertencem ao BCR2, portanto o sinal *ent_bcr2* deve ser alterado, caso o ALOOP seja *resetado* o *ent_bcr2(7)* recebe o valor zero (‘0’) e na ocasião do DLOOP ser *resetado* o *ent_bcr2(6)* recebe zero (‘0’).

O BDR não sofre processo de inicialização, pois serve apenas para armazenar dados a serem transmitidos ou dados recebidos, ou seja, é utilizado sob demanda. O BSVR sofre constantes atualizações que determinam seu estado. Inicialmente sua configuração indica que não há nenhuma interrupção pendente (código $00), isto é, o sinal de entrada *ent_bsvr* é inicializado com zeros (*ent_bsvr <= “00000000”;*).

Baseando-se no fluxograma de inicialização, o passo seguinte seria ler o BSVR e em seguida testar o valor lido, verificando se o conteúdo indica o código $00. Este processo é simplificado utilizando o laço condicional WHILE, onde *q_bsvr* é testado, que é o sinal de saída do BSVR (while q_bsvr <> “00000000” LOOP). Caso o teste seja verdadeiro, segue-se para o tratamento das pendências, permanecendo no corpo do laço até que todas as interrupções tenham sido atendidas. Caso contrário prossegue-se para o último passo da inicialização dos registradores, onde o bit IE, que pertence ao BCR1, é *setado* (*IE <= ‘1’*) habilitando as interrupções.

5.2.2.2 Processo de transmissão de mensagens

Para se transmitir mensagens com o BLDC, escreve-se o primeiro byte da mensagem no BDR, iniciando a transmissão. Quando o BSVR indica *status* $10 (registrador de dados TX vazio – TDRE), ocorre a escrita do próximo byte de dados. Uma vez que todos os dados forem carregados no BDR, cabe ao usuário *setar* o bit TEOD do BCR2 para indicar que não há mais dados a serem transmitidos, para então o BDLC completar a transmissão da mensagem. O fluxograma a seguir (fig. 5.13), conforme [BDL 97], demonstra os passos para transmissão de uma mensagem.

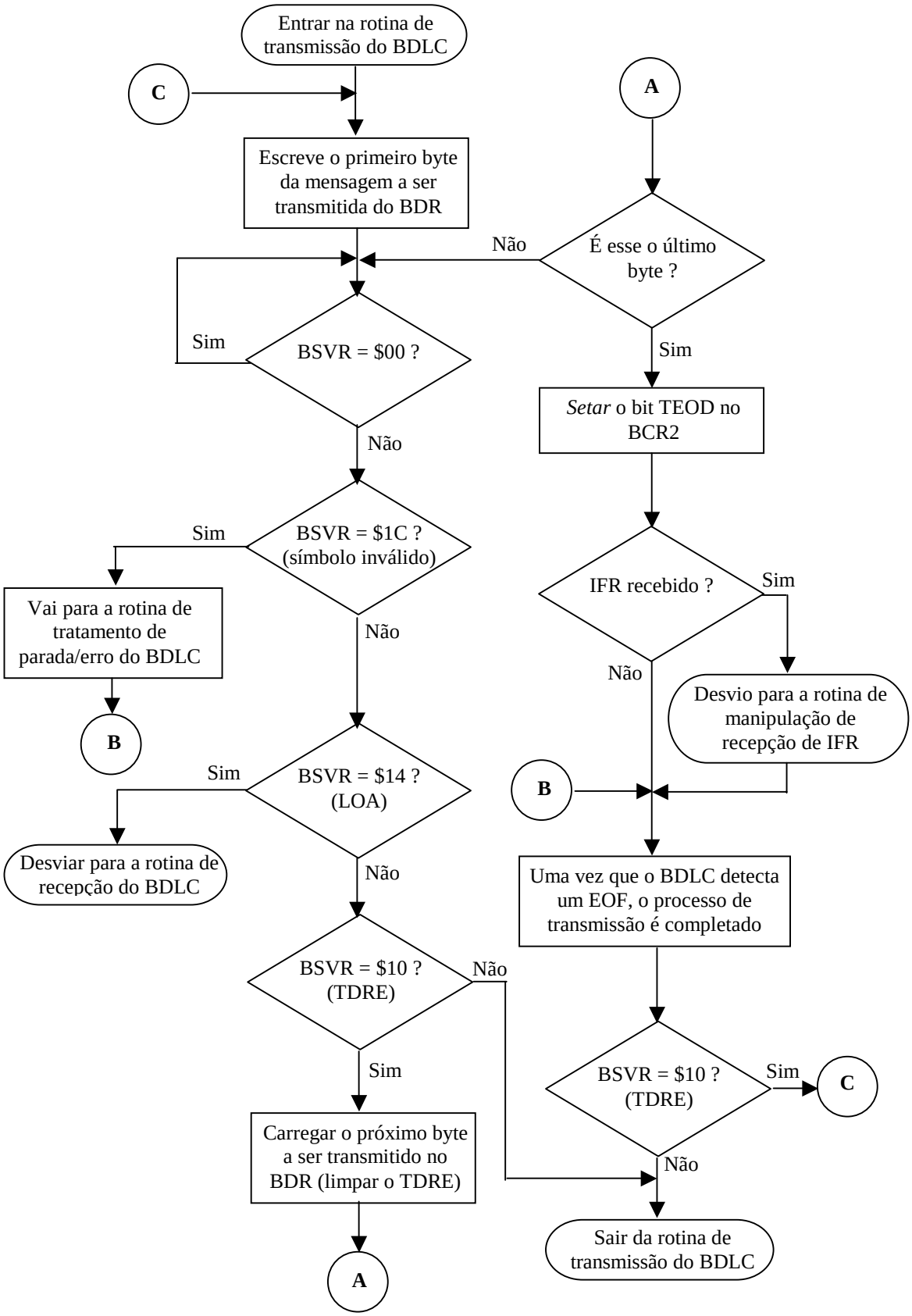

FIGURA 5.13 – Fluxograma de transmissão do BDLC.

A escrita do primeiro byte a ser transmitido no BDR é feita por uma atribuição direta da entrada CPU_TX ao sinal *ent_bdr* (*ent_bdr <= CPU_TX*). Em seguida é

necessário aguardar por uma interrupção, isto é, o BSVR não pode conter o código $00. Para tanto, utiliza-se o comando WAIT associado a cláusula UNTIL, cuja a condição é *BSVR /= $00* (*wait until d_bsvr /= “00000000”;*).

Para tornar possível a escrita de mais de um byte no BDR é necessário que se tenha um laço condicional monitorando este processo, pois o BSVR pode assumir estados críticos que devem receber um tratamento especial. A parte condicional do laço é composta pela expressão *d_bsvr /= "00011100" and q_bsvr /= "00010100" and q_bcr2(3) /= ‘1’*. As duas primeiras comparações indicam, respectivamente, os estados $1C e $14 do BSVR. A terceira comparação revela se o bit TEOD foi *setado*. O corpo deste laço contém um segmento IF-THEN-ELSE, que testa o BSVR para verificar se este é igual a $10 (registrador TX vazio – *d_bsvr = “00010000”*). Caso a condição seja verdadeira, o próximo byte de dados é carregado no BDR (*ent_bdr <= CPU_TX*) e o BSVR recebe o código $00, do contrário, força-se a saída do laço utilizando o comando EXIT, o que resulta na saída da rotina de transmissão do BDLC.

Se a saída do laço for resultante da insatisfação de sua expressão condição, alguns procedimentos que devem ser tomados, pois há certos estados do BSVR que merecem tratamento especial. Por este motivo, após este laço há uma seqüência de testes sobre BSVR. Se o valor dele corresponder ao código $1C (símbolo inválido), passa-se o controle para a rotina de tratamento de erro. Caso valor do BSVR corresponda a $14 (perda de decisão), ocorre o desvio para rotina de recepção. Ou ainda, se o registrador indicar $08 (IFR recebido), trata-se a recepção de IFR. No caso de ocorrer a falha destes testes o BDLC aguarda a detecção de um símbolo de EOF, o qual indica o fim da transmissão.

Baseando-se nas idéias expostas acima, pode-se chegar ao algoritmo demonstrado abaixo (fig. 5.14).

```
ent_bdr <= CPU_TX;
Wait until q_bsvr /= “00000000”;
Transmite_msg:
While q_bsvr /= "00011100" and q_bsvr /= "00010100”
                            and q_bcr2(3) /= ‘1’ loop
   Exit transmite_msg when q_bsvr /= “00010000”;
   Ent_bdr <= CPU_TX;
End loop transmite_msg;
If q_bsvr = “00011100” then          -- $1C
   TRATA_ERRO;
Elsif q_bsvr = “00010100” then       -- $14
   DESVIA_PARA_RECEPCAO;
Elsif q_bsvr = “00001000” then       -- $08
   TRATA_IFR;
Else
   Wait until q_bsvr = “00000100”;   -- EOF
```

FIGURA 5.14 – Algoritmo de transmissão de dados.

5.2.2.3 Processo de recepção de mensagens

O processo de recepção concentra-se principalmente nos bytes de dados recebidos. Estes bytes de dados são armazenados no BDR após a indicação de um estado de que o registrador de dados RX está cheio (RDRF). A indicação é feita através do BSVR, cujo código é igual a $0C. Quando um símbolo de EOF é detectado o BSVR assume o valor indicado pelo código $04 (EOF recebido), constatando-se assim que a mensagem está

completa. O fluxograma a seguir (fig. 5.15), conforme [BDL 97], demostra o processo de recepção de uma mensagem.

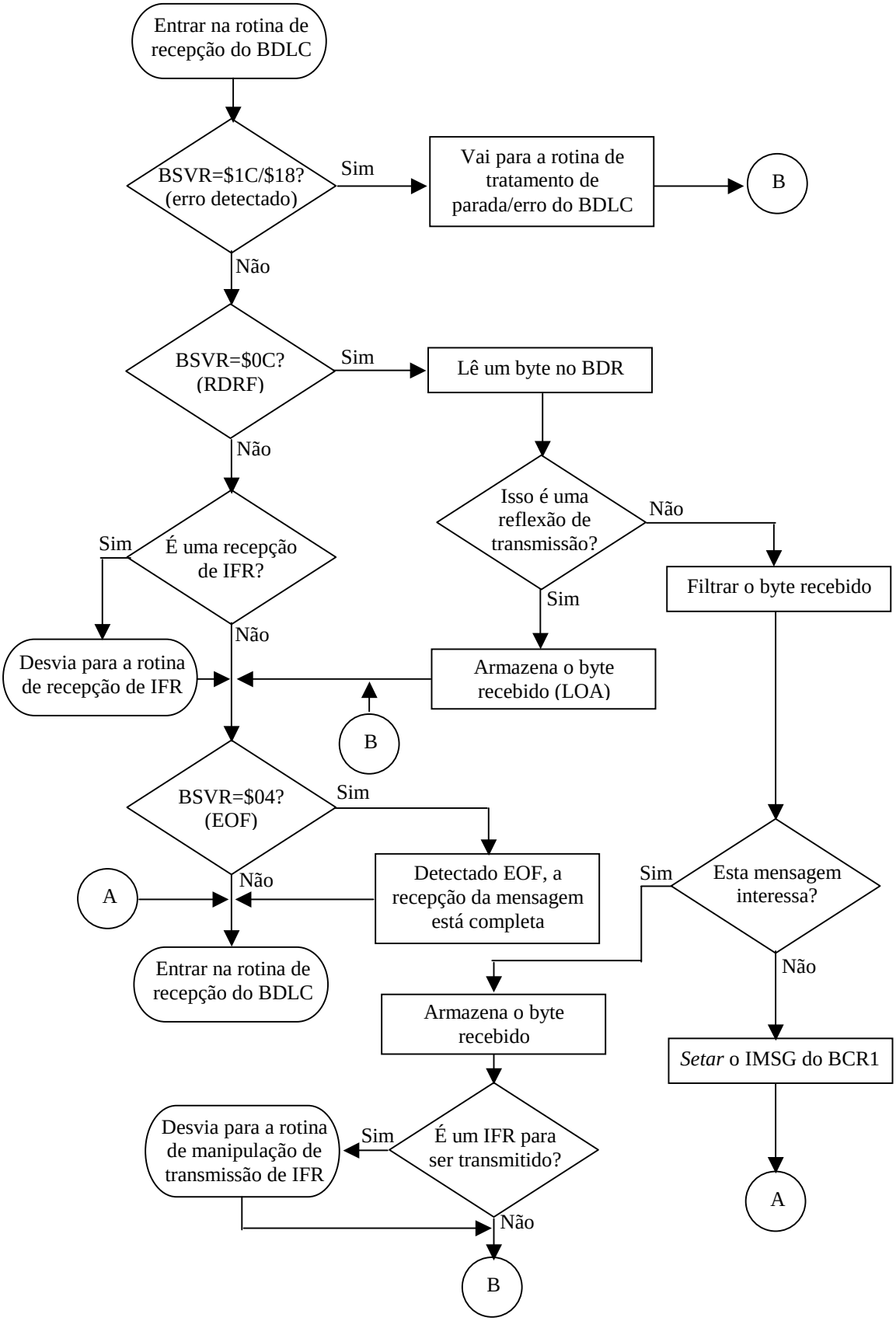

FIGURA 5.15 – Fluxograma de recepção do BDLC

Para implementar o processo de recepção utiliza-se uma seqüência de comandos IF-THEN-ELSE aninhados da maneira mostrada no fluxograma de recepção. Não há a necessidade de laços neste processo, pois a rotina analisa os sinais recebidos sob demanda.

Há várias possibilidades de trajetos no fluxograma, sendo que estes são definidos por uma seqüência de testes.

Uma primeira possibilidade de trajeto seria a detecção de um símbolo inválido ou erro de CRC dos dados recebidos, sendo que para realização deste teste a condição deve ser a seguinte: *q_bsvr = "00011100" or q_bsvr = "00011000"* ($1C | $18). A seguir ocorre a chamada de tratamento de erro, e após a execução desta, realiza-se outro teste para verificar se um símbolo de EOF foi recebido (*q_bsvr = "00000100"*). Sendo satisfeito, significa que a mensagem esta completa e então *ent_bsvr <= "00000000"* para indicar que não há mais pendências.

Outra possibilidade seria a não detecção de erro no BSVR, mas sim o código de RDRF. Os próximos passos são determinados pela execução de outro teste, onde verifica-se o BSVR está no estado de perda de decisão (*q_bsvr = "00010100"*). Caso este teste seja verdadeiro, a saída CPU_RX recebe o conteúdo do BDR e segue-se para a verificação de ocorrência de EOF. Se o estado de perda de decisão não for confirmado, testa-se o bit IMSG do BCR1 (*q_bcr1(1) = '1'*) para verificar se o usuário se interessa pela mensagem. Se for válido, CPU_RX recebe o conteúdo do BDR e , a posteriori, verifica se o byte recebido é um IFR a ser transmitido (*q_bcr2(3) = '1' and q_bcr2(2 downto 0) /= "000"*). Se constatado um IFR, procede-se para rotina de manipulação de IFR. Na seqüência do trajeto, segue-se para verificação de EOF.

Uma terceira possibilidade seria ocasionada pela falha dos dois primeiros testes, resultando em outro acesso ao BVSR para verificar se um IFR foi recebido (*q_bsvr = "00001000"*). Caso tenha sido detectado um IFR, chama-se a rotina de manipulação de IFR. Na seqüência, verifica-se a ocorrência de um símbolo de EOF.

5.2.3 Entidade PSTATMAC

A máquina de estados de protocolo gerência o acesso ao barramento, a formatação de mensagens (através do S_ENCDEC), a detecção de colisões, as decisões, e a verificação/geração de CRC.

Como parte de sua implementação, utiliza uma função para gerar e outra para conferir CRC. A função que calcula o CRC utiliza o polinômio divisor $X^8 + X^4 + X^3 + X^2 + 1$. Cada byte da mensagem é processado pela função, sendo que o byte de CRC é o complemento de um valor *resto* resultante das divisões. A função que verifica o CRC retorna um valor *boolean*, para constatar a validade da mensagem. Para a mensagem ser válida é necessário que o polinômio resultante da aplicação do polinômio divisor sobre todos os bytes de dados, incluindo o byte de CRC recebido, seja igual a $X^7 + X^6 + X^2$. Caso contrário, deve-se notificar o erro de CRC através do barramento C_S (barramento de *status* ligado ao BSVR – *C_S <= "00011000"*).

Além do controle de CRC, cabe a máquina de estados de protocolo verificar se há dados para serem serializados no registrador de deslocamento TX. Isto é constatado com a leitura do barramento C_S, o qual não pode conter o código TDRE (registrador TX vazio). De forma semelhante, ocorre um teste para verificar a condição do registrador de

deslocamento RX, para que os dados vindos do barramento J1850 possam ser armazenados no neste. O registrador recebe os dados se no barramento C_S for detectado RDRF (registrador RX cheio).

5.2.4 Função LPM_SHIFTREG

Os registradores de deslocamento TX e RX, representados pela função parametrizada *lpm_shiftreg*, realizam o deslocamento bits baseados em um parâmetro que indica a direção (LPM_DIRECTION). No caso deste trabalho, este parâmetro assume um valor padrão ("LEFT"), pois não é utilizado explicitamente quando o componente é instanciado. Por tanto, o deslocamento é da esquerda para a direita, onde, por exemplo, uma entrada DATA[7..0] seria serializada disponibilizando primeiramente o bit DATA(7) e posteriormente os demais bits até o extremo direito, isto é, o bit DATA(0).

5.2.5 Entidade S_ENCDEC

A implementação do codificador/decodificador de símbolos é baseada no protocolo de comunicação serial SAE J1850. Segundo [BDL 97], as mensagens são estruturadas conforme a figura (fig. 5.16) abaixo:

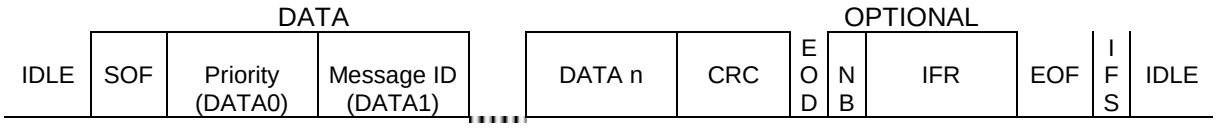

FIGURA 5.16 – Formato das mensagens do barramento J1850 (VPW).

O protocolo limita cada mensagem VPW a um tamanho máximo de 12 bytes, excluindo os símbolos de SOF, EOD, NB e EOF. Os símbolos que ocorrem em uma VPW possuem características próprias, como constata-se pela tabela (tab. 5.4) mostrada abaixo.

TABELA 5.5 – Símbolos VPW.

Símbolo	Definições
0 Lógico	Uma transição ativa para passiva seguida por um período passivo de 64 μs Ou uma transição passiva para ativa seguida por um período ativo de 128 μs
1 Lógico	Uma transição ativa para passiva seguida por um período passivo de 128 μs Ou uma transição passiva para ativa seguida por um período ativo de 64 μs
NB	Mesmas propriedades do 0 e 1 lógico
BREAK	Uma transição passiva para ativa seguida por um período ativo de pelo menos 240 μs
SOF	Uma transição passiva para ativa seguida por um período ativo de 200 μs
EOD	Uma transição ativa para passiva seguida por um período passivo de 200 μs
EOF	Uma transição ativa para passiva seguida por um período passivo de 280 μs
IFS	Um período passivo de 20 μs
IDLE	Um período passivo maior que 300 μs

O processo de recepção tem a finalidade de analisar as propriedades dos sinais recebidos de forma que se possa identificar os símbolos VPW. Para estes fins, tratando-se de VHDL, utilizam-se cláusulas que verificam atributos dos sinais, tais como: '*event*, 'stable e

'last_value. Estas cláusulas retornam valores do tipo *boolean* para que se verifique a condição de um sinal. Por exemplo, *S'event* retornaria *true* se o sinal sofreu qualquer tipo de evento (transição, ativação, etc.), ou ainda, considerando *S'stable(T)* a resposta seria *true* caso o sinal *S* manteve-se estável durante um período *T*, sendo que *T* é uma constante do tipo *time*.

Para implementação do processo de reconhecimento dos símbolos VPW utiliza-se uma série de testes associados aos atributos dos sinais, a fim de satisfazer as propriedades de todos os símbolos possíveis. Caso um sinal recebido não seja reconhecido, entende-se que este é um símbolo inválido e ,portanto, deve-se passar esta informação para o barramento de *status* (C_S <= "000011100"; -- Código $1C).

Vale lembrar que símbolos como EOF e IFR, devem ser anunciados no barramento C_S com seus respectivos códigos, visto que estes servem como controle para outros processos.

O processo de transmissão de mensagens VPW é resultante do fluxograma (fig. 5.17) mostrado a seguir.

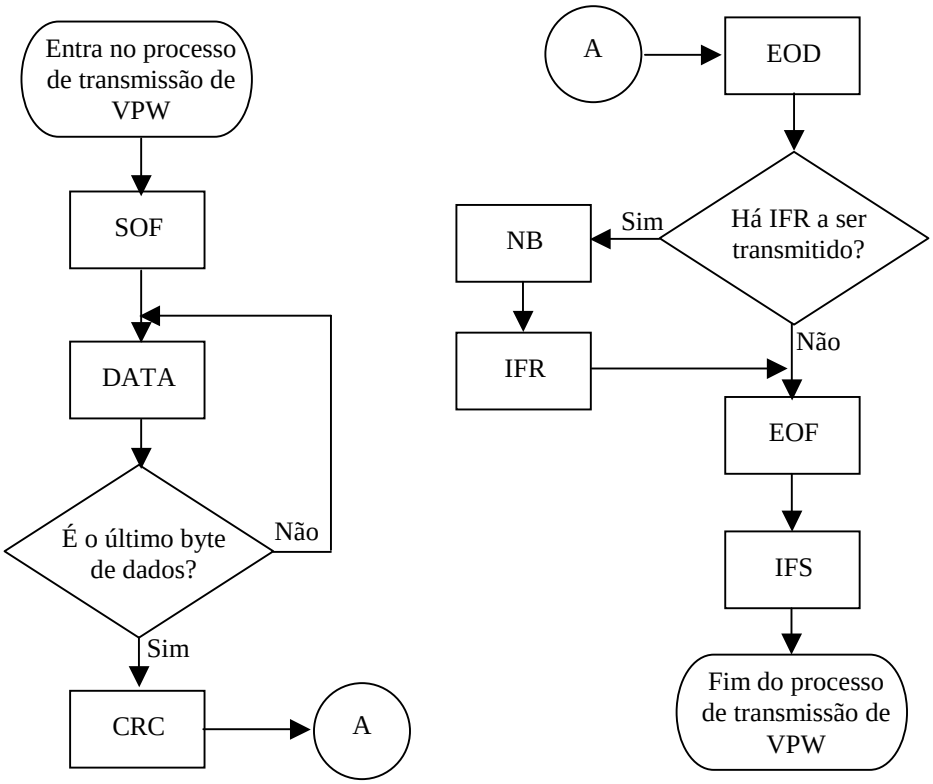

FIGURA 5.17 – Fluxograma para gerar seqüências VPW

Na transmissão de uma mensagem, o primeiro símbolo a ser transmitido é o SOF. Feito isto, o processo começa a transmitir bit a bit os dados da mensagem, até que seja indicado o fim dos bytes de dados. Posteriormente, é transmitido o byte de CRC seguido de um símbolo EOD. Caso sejam utilizados os bits opcionais, o símbolo NB é transmitido seguido dos bits de IFR. Para concluir a transmissão da mensagem transmiti-se um EOF seguido por um IFS.

Para implementar o processo de transmissão utiliza-se basicamente um laço condicionado há uma referência de fim de transmissão de dados de mensagem. Os demais

comandos, que geram os sinais de símbolos, são processados conforme a ordem demonstrada no fluxograma. Para transmitir os símbolos VPW, conforme suas propriedades, são utilizadas seqüências de atribuições a sinais, seguidas pelo comando WAIT associado a cláusula UNTIL cuja condição é a verificação do atributo *'stable*. Por exemplo, para transmitir um EOF a seqüência seria: *S <= '0'; wait until S'stable(280 us)*, sendo que *S* é o sinal que é passado para o barramento J1850.

5.2.6 Entidade RXDIGFIL

O RXDIGFIL não possui especificações funcionais diretas em sua arquitetura, pois esta entidade só referencia outros componentes, que neste caso são funções do MAX+PLUS II. A *lpm_dff* define um *flip-flop* do tipo D que passa um sinal de entrada a saída quando o sinal de CLOCK sofre uma transição. O contador *lpm_counter* é controlado pelo sinal UP/DOWN, que nas configurações adotadas ao receber um sinal indicando o *um lógico*, incrementa o contador, caso contrário, se o sinal for *zero*, o contador decrementa. O ato de decremento ou incremento só é confirmado se o sinal recebido pela porta CLOCK sofre uma transição. A outra função utilizada para compor o filtro digital RX é a *lpm_latch* que implementa um *latch*. O funcionamento básico deste componente depende da porta GATE, que ao receber um sinal igual a *um* habilita a recepção do sinal de entrada, isto é, significa dizer que o sinal de entrada implica na saída. Caso GATE seja *zero*, o sinal da entrada não interfere na saída.

5.2.7 Macrofunção *a_21mux*

O multiplexador *loopback* é implementado com a utilização da macrofunção *a_21mux*, cujo funcionamento é descrito pela tabela (tab. 5.6) abaixo, onde H indica sinal alto, L sinal baixo e X qualquer sinal:

TABELA 5.6 – Funcionamento do multiplexador *a_21mux*

Entradas			**Saída**
S	**A**	**B**	**Y**
L	X	H	H
L	X	L	L
H	H	X	H
H	L	X	L

5.3 Utilização da ferramenta MAX+PLUS II para simulação

Nesta etapa do projeto são realizados processos como síntese (automática) e testes sobre as aplicações implementadas (iterativo). O processo de compilação do MAX+PLUS II realiza síntese automaticamente. O tipo de síntese esta relacionada com o dispositivo escolhido para simulação, sendo este selecionado pelo compilador. Para realização deste trabalho, o dispositivo escolhido pertence a família MAX7000, explanado na seção 5.5 (representação física).

Os testes realizados com a ferramenta verificam a funcionalidade do projeto e fazem um levantamento de dados relativos ao desempenho do circuito (análise temporal).

Quanto à funcionalidade, as implementações realizadas portaram-se como se esperava. Pode ser constatado que componentes simples como REG8, já possuem um tempo de retardo (4,0 nanosegundos) relativo ao deslocamento de dados para saída (fig. 5.18).

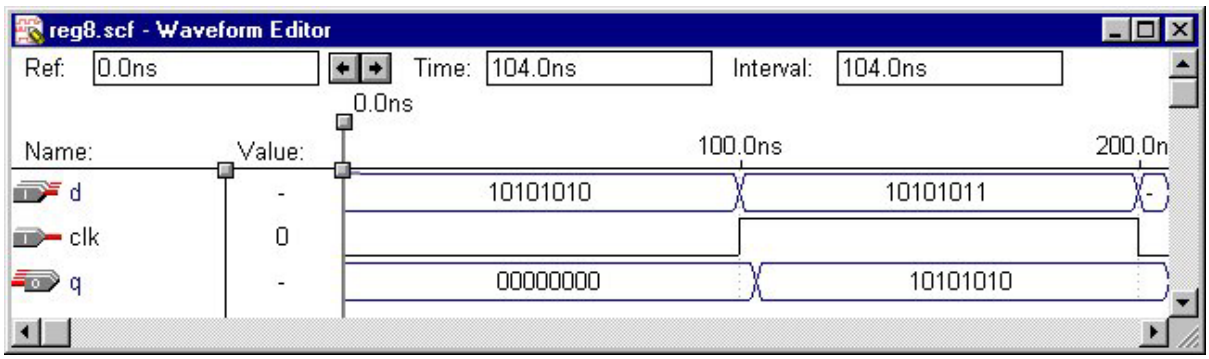

FIGURA 5.18 – Simulação do REG8

O tempo de retardo é proporcional a associação de componentes do circuito. Simulando a entidade RXDIGFIL (fig. 5.19), este fato pode ser constatado, visto que o tempo de deslocamento do sinal da entrada para saída é de *7,6 ns*.

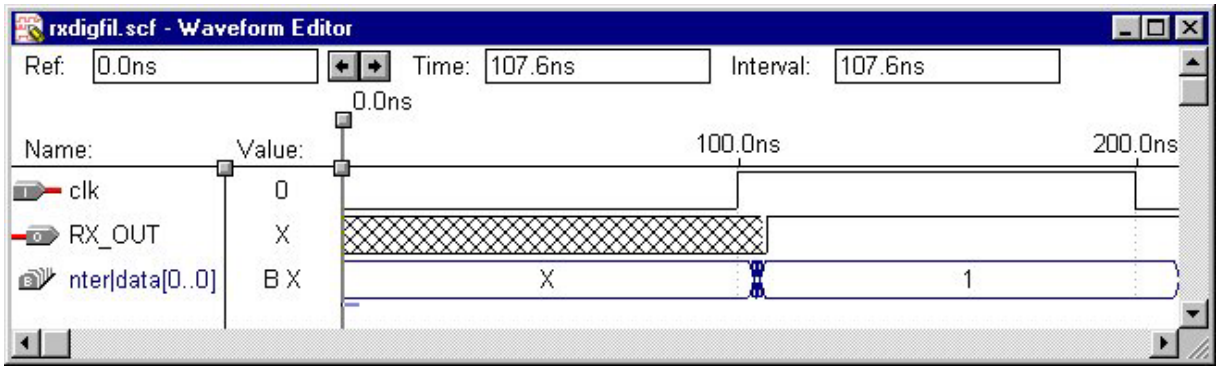

FIGURA 5.19 – Simulação do RXDIGFIL

Para levantar os dados relativos a performance dos componentes utiliza-se o *Timing Analyser* do MAX+PLUS II.

Dados extraídos da entidade CPUINT revelam uma seqüência de atrasos relativos a passagem de dados aos registradores e até que cheguem a seus destinos. Tempo gasto pelo fluxo de dados de uma entrada da CPUINT até a saída de um dos registradores é de *4.5 ns*. Somado a isto, tem-se o tempo relativo aos processos que a entidade trata, onde existe um retardo de *7.5 ns*.

Uma análise da entidade PROTOCHA mostra que o custo médio de tempo é de *10 ns* sendo que deste, *4.0 ns* são relativos ao custo de deslocamento nos registradores REG8 e *6.0 ns* relativos a operações nos registradores de deslocamento e na PSTATMAC. Como registro de performance, o *Timing Analyser* detectou, em um período de relógio de *6.6 ns*, a freqüência de 151.51 MHz.

Analisando a entidade MUXINT verificam-se duas possibilidades de custos de deslocamento. Uma delas seria o deslocamento direto da entrada TX_DATA_IN a saída TX_DATA_OUT passando apenas pelos processos relativos a S_ENCDEC, no qual o custo é de *6.0 ns*. Outra seria relativa ao deslocamento dos sinais que passam pelo RXDIGFIL, cujo custo é *7.6 ns*.

Com a integração dos módulos (entidade BDLC) percebe-se uma significativa queda de desempenho comparada as simulações individuais. O registro de desempenho, em um período de relógio de *13 ns*, foi de uma freqüência de 76.92MHz.

A associação dos componentes do BDLC provocou uma grande queda de desempenho. A medida que novos processos são agregados a estrutura dos componentes, o tempo de resposta tende a aumentar.

5.4 Representação física

A representação física é a última etapa no desenvolvimento de um projeto de descrição de hardware. Para realização desta utilizam-se ferramentas (CAD) que possibilitem uma comunicação com dispositivos programáveis. No caso deste trabalho, a utilização da ferramenta MAX+PLUS II é suficiente.

O Altera MAX+PLUS II possui uma vasta gama de dispositivos programáveis. Com a compilação do protótipo aqui apresentado, obteve-se como resposta a indicação para o uso de um dispositivo EPM7064LC68-7, pertencente a família MAX7000. Esse dispositivo opera sob uma tensão de *5 volts*, possui 68 pinos para entrada e saída e 64 células lógicas.

Para a realização do protótipo foram utilizados, no dispositivo, 30 pinos de entrada e 16 pinos de saída. Além disso, são determinadas 34 células lógicas para o uso, representando 53% da capacidade do dispositivo. Por estes recursos é que torna-se possível a futura utilização em campo do protótipo implementado.

6. Conclusões

Obteve-se com o desenvolvimento do trabalho um estudo da linguagem VHDL voltado aos recursos que esta oferece e que são condizentes com as necessidades de implementação dos componentes do Sistema Distribuído Tolerante a Falhas proposto em [RIB 00], em especial do microcontrolador BDLC. Dentre os recursos que esta linguagem oferece é de suma importância a sua flexibilidade e grande aceitação, além de ser realmente "poderosa" para efetuar descrição de hardware.

Foi demonstrada a ferramenta MAX+PLUS II salientando como sua utilização na implementação de componentes facilita esta tarefa, visto que oferece macrofunções que são definições de componentes específicos e as megafunções que são genéricas e podem ser adaptadas conforme a necessidade do usuário. Assim nota-se a importância desta ferramenta para evitar o trabalho desnecessário e desgastante de implementar todo um componente, quando este é simples e pode ser substituído pelo uso destas funções que o MAX+PLUS II oferece. Além disso o módulo de simulação da ferramenta é muito bom, o que favorece os testes dos componentes que venham a ser implementados futuramente para dar seqüência ao trabalho aqui demonstrado.

A conclusão deste trabalho permite definir os rumos a serem seguidos por quem desejar efetuar a implementação do sistema, especialmente do microcontrolador BDLC, pois foram desenvolvidas algumas implementações básicas a partir das quais é possível definir as metas necessárias ao sucesso, na prática e não só com fundamentação teórica, da implementação dos componentes do microcontrolador. Para tanto é necessário que o desenvolvimento das demais funções do BDLC seja feito com o uso da linguagem VHDL e da ferramenta MAX+PLUX II, visto serem estes os recursos utilizados nesta etapa inicial.

Os módulos implementados foram testados quanto a sua funcionalidade e fornecem uma garantia de que o sistema proposto por [RIB 00] será implementado com sucesso, pois o referido projeto baseia-se em componentes eletrônicos que seguem uma lógica de desenvolvimento semelhante ao que se apresenta neste trabalho.

Bibliografia

[AIR 99] AIRIAU, Roland; BERGÉ, Jean-Michel; OLIVE, Vincent. **Ciruit Synthesis with VHDL**. Terceira edição. Norwell: Kluwer Academic Publishers, 1999. 221p.

[MAX 97] ALTERA CORPORATION. **MAX+PLUS II Getting Started**. San Jose: Altera Corporation, 1997. 353p.

[ASH 90] ASHENDEN, Peter J.. **The VHDL Cookbook** – First Edition. Adelaide: Dept. Computer Science - University of Adelaide, 1990.

[LIP 90] LIPSETT, Roger; SCHAEFER, Carl F.; USSERY, Cary. **VHDL: HARDWARE DESCRIPTION AND DESIGN**. Terceira edição. Norwell: Kluwer Academic Publishers, 1990. 299p.

[BDL 97] Motorola, Inc. **Byte Data Link Controller** – Reference Manual. 1997.

[PER 93] PERRY, Douglas L.. **VHDL – Second Edition**. McGraw-Hill, 1993. 390p.

[RIB 00] RIBEIRO, Leonardo, VARGAS, Fabian, MACARTHY, Marcello. **Especificação de um Sistema Distribuído Tolerante a Falhas Baseado em Microcontroladores Comerciais para Aplicação na Indústria Automobilística**. Pelotas: Curso de Informática – UFPel, 2000. 60p.