CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE NÓS SENSORES EM FPGA E GERENCIAMENTO DE ENERGIA DINÂMICA DO CIRCUITO

Alexandre Ingles da Silva

Marília, 2014

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE NÓS SENSORES EM FPGA E GERENCIAMENTO DE ENERGIA DINÂMICA DO CIRCUITO

Monografia apresentada ao Centro Universitário Eurípides de Marília como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Dr. Fábio Dacêncio Pereira

Marília, 2014

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Alexandre Ingles da Silva

Desenvolvimento de Nós Sensores Com Gerenciamento de Energia Dinâmica em Tecnologia FPGA

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 8,0 (Oito)

Orientador: Fábio Dacêncio Pereira

1°. Examinador: Rodolfo Barros Chiaramonte

2°. Examinador: Paulo Rogerio de Mello Cardoso

Marília, 02 de dezembro de 2014.

RESUMO

Tecnologias FPGAs surgem com características para serem exploradas na construção de circuitos de baixo consumo de energia. Neste projeto foram construídos em FPGA nós sensores que contam com todas as características necessária para o estabelecimento de uma Rede de Sensores sem Fio RSSF. O circuito tem como objetivo o baixo consumo de energia dinâmica do circuito aplicando algumas técnicas de implementação como DPM (*dynamic power management*) minimizando os domínios de clocks da arquitetura lógica. A arquitetura construída tem todas as características necessárias para integrar, sincronizar e para realizar a comunicação de acordo com a aplicação. O impacto do consumo de energia dinâmica na arquitetura construída em FPGA é analisado. Além disso, são analisados consumo e desempenho, também, desempenho de acordo com a frequência em alguns módulos principais. É discutido a aplicação do circuito construído em uma tecnologia eASIC Nextreme-2, que oferece característica de baixo consumo de energia e, comparado com outros grupos de nós sensores.

Palavras-chave: RSSF, Modelo de Comunicação, FPGA, Arquitetura Lógica, Consumo de Energia Dinâmica.

ABSTRACT

FPGA technologies emerge with characteristics to be exploited in the construction of low-power circuits. In this project were built a FPGA sensor node that has all the necessary characteristics for the establishment of a Wireless Sensor Network (WSN). The circuit aims the low-energy dynamics of the circuit applying some implementation techniques such as DPM (dynamic power management) minimizing clocks domains of the logical architecture. The built architecture has all the features needed to integrate, synchronize, and to perform communication according to the application. The impact of dynamic energy consumption in the built architecture in FPGA is analyzed. In addition, consumption and performance of some modules are analyzed. We discuss the implementation of the built architecture in an eASIC Nextreme-2 technology, which offers feature low-power, and it is compared with other groups of sensor nodes.

Keywords: WSN, Model of Communication, Logic Architecture, Dynamic Power Consumption.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIMBOLOS

FPGA – Field-Programmable Gate Arrays

ASIC – Application Specific Integrated Circuits

DPM – Dynamic Power Manager

RSSF – Rede de Sensores Sem Fio

NOCs – Network-on-Chips

DPCD – Dynamic programmable clock divider

DVFS – Dynamic Voltage and Frequency Scaling

MEMS – Micro Electro-Mechanical Systems

CAD – Computer Aided Design

SRAM – Static Random Access Memory

LUT – lookup-tables

DCM – Dynamic Clock Manager

PLLs – Phase-Locked Loops

DLLs – Delay Locked Loops

VHDL ou VHSIC – Very High Speed Integrated Circuit

RAM – Random Access Memory

CAPITULO 1

1.1 Introdução

Redes de sensores sem fios (RSSFs) estão cada vez mais ganhando popularidade em todo o mundo por suas várias vantagens, tais como em dispositivos embarcados. Estão se espalhando por todas as áreas de aplicações por oferecer baixo custo, incentivando muitos estudos na área. Há uma série de implementações de redes de sensores sem fio e não há um padrão definitivo (Arampatzis, 2005).

Independentemente do tipo exato de plataforma, sabe-se que aplicações podem ser classificadas em algumas formas gerais: aplicações militares, monitoramento ambiental, aplicações centradas comerciais ou humanas e, aplicações para robótica. Com o desenvolvimento de sistemas embarcados e tecnologias de rede, houve um interesse crescente no fornecimento de *fine-grained* e controle de ambientes usando de dispositivos de baixo consumo. Um nó sensor típico possui vários componentes: rádio transceptor com uma antena que tem a capacidade de enviar ou receber pacotes de dados, um micro controlador que pode processar os dados e programar tarefas relativas, vários tipos de sensores de detecção de dados do ambiente, e bateria para o fornecimento de energia (Yu, 2011).

Para construção de redes de sensores sem fio em circuitos reconfiguráveis, vários tipos de tecnologias que atende às necessidades podem ser exploradas. Atualmente a tecnologia FPGA ganha a atenção fornecendo a capacidade e flexibilidade para construção circuitos lógicos, permitindo incluir funções ou importá-las de outros componentes. Há crescimento substancial no desempenho e consumo reduzido de energia tanto para FPGA quanto para tecnologias ASIC, em comparação com algoritmos iterativos não restauráveis (Suresh, 2013).

Embora melhorias significativas já tenham sido feitas, possibilidades de reduzir ainda mais o consumo de energia em FPGAs permanecem. Programação em circuito reconfiguráveis para reduzir o consumo de energia da FPGA é proposta a partir de uma nova família de FPGA *Routing Switch* que são programáveis para operar em três modos diferentes: alta velocidade, baixo consumo de energia, ou *sleep* (Anderson, 2009).

Circuits-on-FPGAs como NOCs (*Network-on-Chip*) podem ser construídos usando a lógica digital da FPGA e sua interface de uso geral I / O, que é necessário para conectar os dispositivos físicos na interface externa do FPGA.

O uso das FPGAs permite reconfiguração dinâmica parcial que tem muitas vantagens quando se trabalha com nós sensores sem fio. A possibilidade de adaptar a funcionalidade do sistema, carregando novas funções em tempo de execução de uma forma muito rápida, quando se trabalha com ambientes mutáveis, abre novas oportunidades, no entanto, a implementação destas técnicas pode não ser suficiente para alcançar valores aceitáveis de consumo de energia. Mesmo que o FPGA esteja trabalhando de uma forma muito eficiente, os componentes restantes consumiram energia mesmo quando não são necessários (Valverde, 2012).

Sendo assim, foi projetada e construída uma arquitetura lógica de nós sensores que visa gerenciar os dispositivos físicos conectados à interface externa da FPGA pelos seus correspondentes componentes lógicos. O circuito funciona como um *core* genérico construído para nós sensores e pode ser adaptado para diferentes aplicações de monitorização de ambiente.

A arquitetura construída visa reduzir o consumo de energia dinâmica do circuito através do uso de um modulo *Dynamic programmable clock divider* (DPCD) que gerencia a distribuição de frequência para os componentes lógicos e, consequentemente gerencia os dispositivos físicos conectados a interface externa da FPGA após estabelecer a sincronização de rede. Também foi analisado em vários aspectos, o consumo total de energia dinâmica do circuito, em que o consumo de energia dinâmica é uma parte do consumo de energia da FPGA (eASIC, 2014).

O módulo divisor de clock que através de sua entrada do clock gera diferentes frequências para outros módulos é usado como parte da técnica para minimizar o consumo de energia dinâmica do circuito, essas frequências são distribuídas para os componentes lógicos que compõem arquitetura lógica.

A análise do consumo de energia do circuito é baseada na energia dinâmica, considerando-se que os dispositivos FPGA diferem entre si: através de famílias de dispositivos e fornecedores de recursos e roteamento da arquitetura em cada FPGA variam muito (Yiannacouras, 2006).

Embora os módulos sejam separados, eles são logicamente ligados entre si e um módulo de sincronização é utilizado para definir o tempo, este tempo é usado para a integração e a sincronização entre as arquiteturas lógicas.

O circuito foi construído visando apenas o consumo de energia dinâmica quando operando em *full mode*. *Low-power* ou *sleep* não são analisados pela razão de que a arquitetura Xilinx Spartan-6 LX16 FPGA utilizada não fornece recursos como DVFS (escala de frequência da tensão dinâmica). Uma avaliação do algoritmo construído é feito para a arquitetura eASIC Nextreme-2 e, com base no consumo de energia dinâmica e estática da arquitetura construída é feita uma analise de consumo nesta tecnologia eASIC.

1.2 Principais tecnologias e métodos utilizados para construção dos nós sensores

Modelo de comunicação:

- Para a implementação dos nós sensores foi construído um modelo de comunicação para que os nós sensores pudessem estabelecer comunicação, sincronização e finalmente uma rede de sensores sem fio (RSSF) de forma autônoma.

Objetivo do modelo de comunicação:

- O modelo de comunicação tem como objetivo o gerenciamento de consumo de energia pelos nós sensores, apenas consumindo energia quando realizando a comunicação.

Tecnologias físicas utilizadas para construção dos nós sensores:

- Os nós sensores foram construídos utilizando da linguagem VHDL em um dispositivo Xilinx Spartan-6 LX16 FPGA. Alguns outros dispositivos como, XBee-PRO 900 RF para transmissão e recepção de pacotes e LV-MaxSonar® - EZ1™ para leitura de dados do ambiente foram utilizados para validação do protótipo implementado.

CAPÍTULO 2

2.1 Redes de Sensores sem Fio

As redes de sensores sem fio (RSSF) são classificadas como a terceira onda da computação, pois possuem a capacidade de monitorar, processar, transmitir informações (Dulman, 2006).

Redes de sensores sem fio é o termo utilizado para classificar dispositivos dotados da capacidade de processamento e comunicação que se auto-organizam em redes do tipo ad hoc estáticas. Apesar dos nós serem estáticos a topologia de rede das RSSFs é dinâmica. Nós sensores são dispositivos dotados de capacidade de sensoriamento, armazenamento, processamento, comunicação e fonte de energia própria (Bhattacharya, 2003), (Dulman, 2006).

O avanço que tem ocorrido na área de microprocessadores, novos materiais de sensoriamento, micro sistemas eletromecânicos (MEMS – *Micro Electro- Mechanical Systems*) e comunicação sem fio tem estimulado o desenvolvimento e uso de sensores "inteligentes" em áreas ligadas a processos físicos, químicos, biológicos, dentre outros. É usual ter num único chip vários sensores, que são controlados pela logica do circuito integrado, com uma interface de comunicação sem fio. Normalmente o termo sensor "inteligente" e construído ao chip que contem um ou mais sensores com capacidade de processamento de sinais e comunicação de dados. A tendência é produzir esses sensores em larga escala, barateando o seu custo, e investir ainda mais no desenvolvimento tecnológico desses dispositivos, levando a novas melhorias e capacidades (Loureiro, 2003).

Normalmente essas redes são compostas por milhares de nós sensores distribuídos em uma determinada área e, possui restrição de energia exigindo mecanismo que ofereça suporte de autoconfiguração, tratamento a falhas e perda de nós. As RSSFs modernas são totalmente autônomas, configurando-se para realizar a comunicação contornando situações adversas sem intervenção alguma de humanos. Para conseguir todas estas características, protocolos devem ser analisados antes de serem empregados em determinados ambientes, afirmando que o nó sensor possa corresponder a todas as funcionalidades descritas pelo protocolo.

O poder computacional de um nodo sensor é pequeno se for considerado um nodo isoladamente. O poder de processamento destes dispositivos encontra-se na topologia dinâmica da rede e nos algoritmos distribuídos. Diferente dos algoritmos distribuídos tradicionalmente implementados, as RSSF trabalham de forma colaborativa (Loureiro, 2003), (Dulman, 2006).

2.1.1 Aplicações

Diversos RSSFs vêm sendo desenvolvido com o surgimento de novas tecnologias ou o barateamento de dispositivos já existentes. Beneficiado por inovações tecnológicas diferentes campos de aplicação se candidatam ou já utilizam das RSSFs para resolver problemas ou se beneficiar das vantagens oferecidas por esta tecnologia. A seguir listamos algumas das principais aplicações de uma RSSFs (Loureiro, 2003).

- *Controle*: Para prover algum mecanismo de controle, seja em um ambiente industrial ou não. Por exemplo, sensores sem fio podem ser embutidos em "peças" numa linha de montagem para fazer testes no processo de manufatura.
- *Ambiente*: Para monitorar variáveis ambientais em locais internos como prédios e residências, e locais externos como florestas, desertos, oceanos, vulcões, etc.
- *Trafego*: Para monitorar trafego de veículos em rodovias, malhas viárias urbanas, etc.
- *Segurança*: Para prover segurança em centros comerciais, estacionamentos, etc.
- *Medicina/Biologia*: Para monitorar o funcionamento de órgãos como o coração, detectar a presença de substancias que indicam a presença ou surgimento de um problema biológico, seja no corpo humano ou animal.
- *Militar*: Para detectar movimentos inimigos, explosões, a presença de material perigoso como gás venenoso ou radiação, etc. Neste tipo de aplicação, os requisitos de segurança são fundamentais. O alcance das transmissões dos sensores é geralmente reduzido para evitar escutas clandestinas. Os dados são criptografados e submetidos a processos de assinatura digital. As dimensões são extremamente reduzidas e podem utilizar nodos sensores moveis como os transportados por robôs.

De forma genérica, RSSFs podem ser usadas em segurança e monitoramento, controle, atuação e manutenção de sistemas complexos, e monitoramento de ambientes internos e externos (Loureiro, 2003).

2.1.2 Fatores Críticos em uma RSSFs

As RSSFs consistem de centenas ou milhares de nodos trabalhando de forma coletiva, diferente das redes de computadores tradicionais. As RSSFs possuem sérias restrições de energia, processamento e memória. Estas restrições obrigam o desenvolvimento de um conjunto de protocolos e algoritmos específicos para o gerenciamento destes dispositivos, maximizando o tempo de vida da rede.

2.1.2.1 Energia

A energia é considerada o principal fator restritivo de um nodo sensor, devido ao nodo ser alimentado por meio de bateria e geralmente as redes de sensores serem aplicadas em ambientes de difícil acesso (Loureiro, 2003), (Dulman, 2006).

As três principais funções consumidoras de energia em um nodo sensor são: o processamento local, a comunicação entre os nodos e o sensoriamento do ambiente (Dulman, 2006).

A transmissão, recepção e busca de nodos, realizada pelo módulo de comunicação de um nodo sensor geralmente é o responsável pelo maior dispêndio de energia. O consumo de energia, também depende do dispositivo de comunicação utilizado a rádio frequência (Dulman, 2006).

Em muitas aplicações, os sensores serão colocados em áreas remotas, o que não permitira facilmente o acesso a esses elementos para manutenção. Neste cenário, o tempo de vida de um sensor depende da quantidade de energia disponível. Aplicações, protocolos, e algoritmos para RSSFs não podem ser escolhidos considerando apenas sua “elegância” e capacidade, mas definitivamente a quantidade de energia consumida. Assim, o projeto de qualquer solução para esse tipo de rede deve levar em consideração o consumo, o modelo de energia e o mapa de energia da rede (Loureiro, 2003).

2.1.2.2 Tolerância a Falha

A tolerância a falhas é requisito primordial para projetos de aplicações e protocolos em RSSF, pois são frequentes.

As falhas podem ser classificadas em dois grandes grupos, as falhas silenciosas e as falhas bizantinas (Macedo, 2006).

As falhas silenciosas são caracterizadas pela não exteriorização dos resultados, a não apresentação da saída ocorre enquanto o componente ou o sistema está falho. Nas falhas bizantinas o sistema continua gerar as saídas, mas estas são inconsistentes, resultando em respostas corretas e incorretas.

As falhas silenciosas podem ser causadas por elementos internos ou externos a rede, fenômenos atmosféricos, fontes móveis de interferência, desastres naturais, quebra acidental, bloqueio do processador (*deadlock*), maliciosas (segurança da rede) e esgotamento de bateria.

2.2 Tecnologia FPGA

A computação reconfigurável baseia-se em dispositivos lógicos reprogramáveis que podem atingir um desempenho elevado e, ao mesmo tempo, fornecer a flexibilidade da programação em nível de portas lógicas. Um dispositivo de hardware típico utilizado em computação reconfigurável são as FPGAs (*Field-Programmable Gate Arrays*). A velocidade e os recursos disponíveis em FPGAs recentes são comparáveis com os dos ASICs, enquanto se beneficiam de muita da flexibilidade inerente às implementações em software.

2.2.1 Origem

As FPGAs foram já a bastante tempo introduzidas, mais exatamente pelo co-fundador da Xilinx em 1985 e, atualmente constitui de uma grande variedades de componentes disponíveis como produtos comerciais.

2.2.2 Arquitetura FPGA

As FPGAs são os dispositivos lógicos programáveis de maior capacidade disponível hoje em dia. O número de portas de sistema em FPGAs comerciais tem crescido bastante rapidamente desde a sua introdução nos meados dos anos 80, e atinge atualmente uma boa parte em dispositivos da Xilinx (Skliarova, 2003).

A arquitetura típica de uma FPGA está representada na Fig. 1. Uma FPGA inclui um *array* de blocos lógicos interligados por recursos de encaminhamento e cercado por um conjunto de blocos de entrada/saída, sendo todos estes componentes programáveis pelo utilizador. Os blocos lógicos contêm elementos combinatórios e sequenciais possibilitando a implementação de funções lógicas bem como de circuitos sequenciais (Skliarova, 2003).

Os recursos de encaminhamento incluem segmentos de pistas de ligação pré-fabricadas e interruptores programáveis. Um circuito lógico é implementado em FPGA ao distribuir a lógica entre os blocos individuais e interligá-los posteriormente com os interruptores programáveis. É de notar que os atrasos resultantes são fortemente influenciados pela distribuição da lógica e pela estrutura de encaminhamento. Portanto, o desempenho de circuitos mapeados em FPGA depende bastante da eficácia das ferramentas CAD (*Computer Aided Design*) utilizadas para a implementação (Skliarova, 2003).

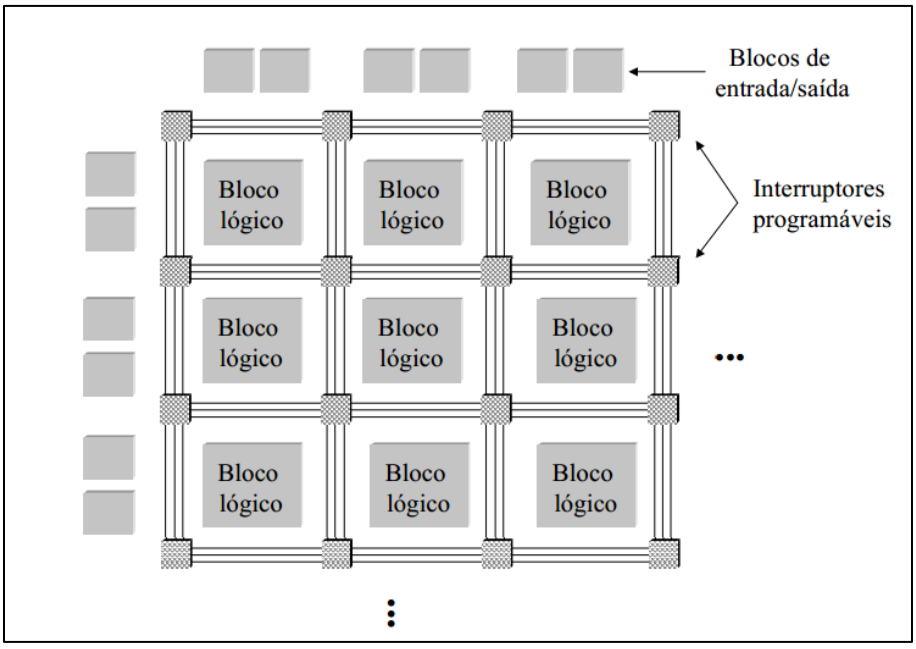

Figura 1 – Arquitetura típica de FPGA (Skliarova, 2003).

Uma das maiores vantagens das FPGAs é a sua arquitetura flexível que serve muito bem para uma ampla gama de aplicações. Estas aplicações incluem implementação de controladores de dispositivos, circuitos de codificação, lógica arbitrária, prototipagem e emulação de sistemas, etc. As FPGAs recentes passaram a incorporar várias estruturas heterogéneas tais como blocos de memória, o que possibilita a implementação de sistemas completos num único encapsulamento (Skliarova, 2003).

2.2.3 Classificação das FPGAs

FPGA modernas consistem de recursos de lógica programável incorporados em um mar de interconexões programáveis. Os recursos lógicos programáveis podem ser configurados para executar qualquer função lógica, ao passo que as interconexões fornecem a flexibilidade para conectar qualquer sinal no projeto de qualquer recurso lógico (Grover, 2012).

A tecnologia de programação para os recursos lógicos e de interconexão podem ser *Static Random Access Memory* (SRAM), memória FLASH, ou antifuse (Han, 2007), (Brown,1994), (Greene, 1993).

FPGA baseadas em SRAM oferece no circuito reconfigurabilidade à custa de ser volátil, enquanto antifuse são dispositivos de gravação única. FPGA baseado em Flash fornecem uma alternativa intermediária proporcionando reconfigurabilidade, bem como não volatilidade. A maioria das tecnologias de programação popular em *state-of-the-art* FPGAs é SRAM (Grover, 2012).

2.2.4 Dissipação de Energia em FPGAs

Como em todos os circuitos integrados, FPGA dissipam energia estática e dinâmica. Energia estática é consumida devido ao *leakage* do transistor e é dissipada no *leakages* a partir da fonte de alimentação para o *ground* através de transistores que estão no *off-state* devido a *sub-threshold leakage* (a partir da fonte para o dreno), *gate-induced drain leakage*, e *gate direct-tunneling leakage*. A energia dinâmica é consumida pela alternância dos nós em função da

tensão, frequência e capacitância e é dissipada quando capacitâncias são carregadas e descarregadas durante o funcionamento do circuito (Skliarova, 2003).

FPGA consomem muito mais energia do que ASICs porque tem um grande número de transistores por função lógica, a fim de programar o dispositivo como dito acima.

No entanto, programação é a essência desta tecnologia e esse *overhead* deve ser assumido. Os circuitos extras que oferece flexibilidade para um FPGA afeta tanto a energia estática e dinâmica dissipada pelo FPGA. A FPGA contém um grande número de bits de configuração, tanto dentro de cada elemento lógico e no roteamento utilizado para conectar os elementos lógicos. Cada um destes bits de configuração dissipa energia estática ao passo que um ASIC não contém qualquer bit programação e, assim, consome significativamente menos energia estática do que um FPGA. Além disso, uma vez que as *lookup-tables* (LUT) consomem significativamente mais transistores do que a lógica correspondente em um ASIC, a energia estática dissipada na lógica é maior em uma FPGA do que em uma ASIC (Skliarova, 2003).

2.2.5 Reduzindo Energia Dinâmica com Esquema de Clock

Com o aumento do tamanho e da complexidade das (FPGAs), distribuição de clock tornou-se uma questão importante. Tornou-se cada vez mais difícil manter altas taxas de clock com a complexidade e tamanho dos circuitos implementados constantemente. Gerenciadores de clock estão sendo incorporadas em FPGAs para resolver problemas de distribuição de clocks, e para melhorar a sua flexibilidade e funcionalidade (Xilinx, 1998), (Altera, 1999), (Lucent,1999).

Gerenciadores de clock podem ser utilizados em muitas aplicações das FPGA. Usando os recursos de multiplicação e divisão de frequência, os *designers* são capazes de manipular as taxas de clock *on e off-chip*, e usá-lo para o alinhamento do clock entre os subsistemas que utilizam diferentes taxas de clock (Brynjolfson, 2000).

Dynamic clock management (DCM) pode ter um forte impacto na redução de consumo de energia em FPGAs. A maioria dos clocks das FPGAs modernas têm gerenciadores para solucionar problemas de altas velocidades de clock. Incluem PLLs e / ou *Delay Locked Loops* (DLLs), juntamente com divisores e interface de circuitos do clock. Energia dinâmica é

consumida alternando os nós em função da tensão, frequência e capacitância são dissipadas quando capacitâncias são carregadas e descarregadas durante a operação do circuito, em seguida, consumidos durante eventos de comutação no núcleo, ou I / O da FPGA (Grover, 2012).

Dynamic programmable clock divider (DPCD) pode ser implementado com o objetivo de multiplicar as frequências de uma arquitetura lógica e distribuí-la a outros módulos lógicos através dos sinais lógicos, reduzindo a quantidade de energia dinâmica consumida pelo circuito.

Para isso um simples divisor de clock pode ser implementado. A taxa de clock inicial deve ser utilizada como entrada, reduzindo assim a frequência em várias fases; estas fases caracterizadas por uma frequência pode ser distribuído para os módulos lógicos no circuito. Enquanto a frequência de operação do circuito é diminuída, mas, se cumprir os requisitos computacionais, uma quantidade significativa de energia consumida pelo circuito pode ser salvo, com uma boa oportunidade de implementar circuitos que visa o baixo consumo de energia.

Com um DPCD, que atinge todos os módulos da arquitetura, é possível propor uma oportunidade de economizar energia, executando a aplicação em baixa frequência durante o modo ocioso. Como em qualquer caso de circuitos integrados FPGAs pode dissipar a energia por alguns motivos.

Existem várias técnicas que podem reduzir o consumo de energia dinâmica do circuito, como a redução da energia dinâmica no esquema do clock; reduzir *Logic and Signal Power*; Redução de energia I / O. Cada um deve ser considerado para reduzir o consumo de energia dinâmica do circuito e, assim, atingir um consumo aceitável para aplicações que requerem um baixo consumo de energia.

2.2.6 Expectativas para as FPGAs

Em FPGA, interruptores programáveis controlados por memória de configuração ocupam uma grande área e adiciona uma quantidade significativa de capacitância parasita e resistência aos recursos de lógica e roteamento. Como resultado, os FPGAs são aproximadamente 3 vezes mais lentas, 20 vezes maiores, e 12 vezes menos eficiente de energia em comparação com os ASIC (Kuon,2006).

A sobrecarga de área, combinada com os custos de pesquisa e desenvolvimento, aumenta o custo por unidade de FPGAs, o que os torna menos adequado para aplicações de alto volume. Além disso, a velocidade e sobrecarga de potência impede o uso de FPGAs para aplicações de alta velocidade ou *low power* (Grover, 2012).

Em mais de 20 anos desde a introdução do FPGA, pesquisa e desenvolvimento tem produzido melhorias na velocidade do FPGA e eficiência na área, diminuindo a distância entre FPGAs e ASICs e tornando as FPGAs à plataforma de escolha para a implementação de circuitos digitais. FPGAs são promessas significativas como uma rápida substituição do mercado ASIC em muitas aplicações. Um número significativo de estudos incluem foco mais rápido e mais eficientes recursos da área de roteamento programáveis (Grover, 2012).

2.3 VHDL

Para descrição de hardware existe vários tipos de tecnologias, a que utilizamos foi o VHDL. O nome VHDL é um acrônimo de VHSIC *Hardware Description Language*. Já o termo VHSIC é o acrônimo de *Very High Speed Integrated Circuit*. Assim pode-se traduzir livremente o nome VHDL como "linguagem de descrição de hardware para circuitos integrados de velocidade muito alta" (Edson, 2007).

A linguagem VHDL foi originalmente desenvolvida por empresas contratadas pelo governo americano e agora é um padrão requerido por todos os ASICs (*Application Specific Integrated Circuits* – circuitos integrados específicos da aplicação) projetados para o exército americano. Ele foi padronizado pelo IEEE em 1987 (Padrão 1076-1987 ou VHDL 87) e foi atualizado em 1993 (Padrão 1076-1993 ou VHDL 93). Os trabalhos do IEEE continuam (Edson, 2007).

A linguagem VDHL implementada em um dispositivo reprogramável (exemplo: FPGA – *Field Programmable Gate Array*) descreve o que um sistema faz e como é feito. Ela permite, através de simulação, verificar o comportamento do sistema assim como descrever o hardware em diferentes níveis de abstração como algoritmo comportamental e transferência entre registradores (RTL).

O ciclo de projeto construído em VHDL segue a seguinte ordem (Introdução ao VHDL, 2014).

- *Especificação*: determinar requisitos e funcionalidade do projeto.
- *Codificação*: descrever em VHDL todo o projeto, segundo padrões de sintaxe.
- *Simulação de código fonte*: simular o código em ferramenta confiável a fim de verificar preliminarmente cumprimento da especificação.
- *Síntese*: compilação de um código VHDL para uma descrição abstrata.
- *Otimização*: seleção da melhor solução de implementação para uma dada tecnologia.
- *Fitting*: lógica sintetizada e otimizada mapeada nos recursos oferecidos pela tecnologia.
- *Simulação do modelo*: resultados mais apurados de comportamento e *timing*.
- *Geração*: configuração das lógicas programáveis ou de fabricação de ASICs.

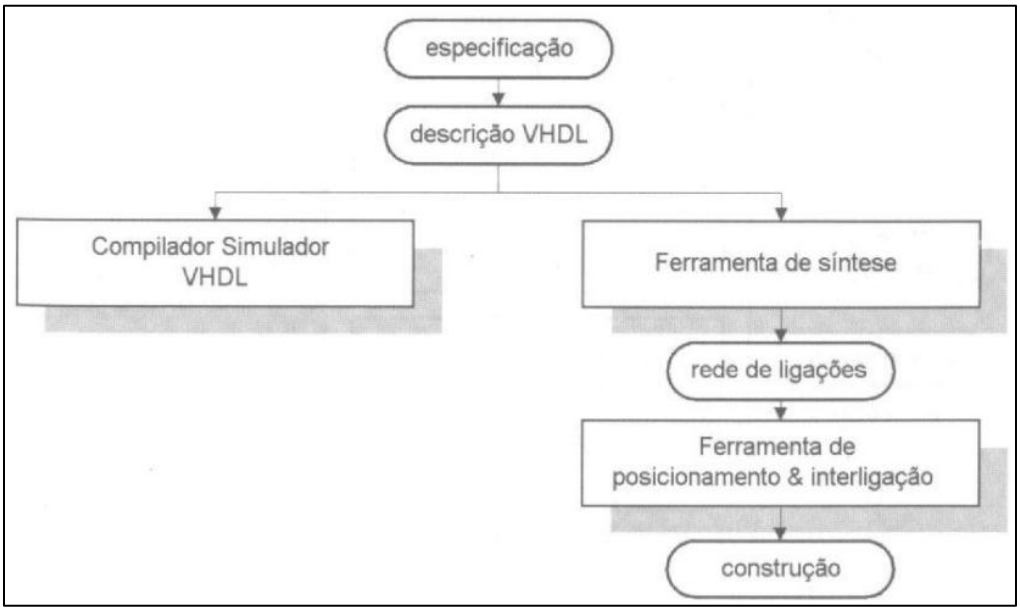

Figura 2 - Etapas de um projeto construído na linguagem VHDL (Introdução ao VHDL, 2014).

O próximo capítulo trata dos dispositivos físicos utilizados para implementação e validação da arquitetura lógica construída na linguagem VHDL. Estes dispositivos oferecem

todas as funcionalidades para sua utilização e características adequadas para o funcionamento do protótipo.

CAPÍTULO 3

Para construir a arquitetura lógica proposta foi feito o uso de módulos físicos, que darão suporte para o desenvolvimento e a construção dos circuitos lógicos. Cada módulo físico é representado por um componente lógico que através das conexões dos sinais lógicos formaram uma arquitetura lógica denominada nós sensores e, a interligação entre estes nós sensores estabelecera uma RSSF. Os módulos físicos utilizados para formar o nó sensor são sensores, rádios transceptor e uma plataforma Nexys3 baseada na Xilinx Spartan-6 LX16 FPGA que serão especificados nesta sessão. Estes dispositivos são conectados na interface externa da *board* Nexys3 e são manipulados pelo seu correspondente componente lógico. A arquitetura lógica é toda construída e validada na linguagem VHDL. A seguir tem os detalhes e todas as características dos dispositivos físicos utilizados.

3.1 Dispositivos Físicos

Nesta seção serão apresentados e detalhados todos os dispositivos físicos utilizados para a construção da arquitetura lógica. Esses dispositivos permitirão que a arquitetura lógica possa ter acesso ao mundo externo estabelecendo assim comunicação com outros dispositivos.

3.1.1 Dispositivo FPGA

O Nexys3 é uma plataforma de desenvolvimento de circuito digital completo, pronto para uso com base no dispositivo Xilinx Spartan-6 LX16 FPGA. O Spartan-6 é otimizado para lógica de alto desempenho, e oferece alta capacidade, alto desempenho e recursos. Oferece 2.278 *slices* cada um contendo quatro LUTs de 6 entradas e oito *flip-flops* de 576Kbits de bloco de RAM rápido para utilização. Através da interface externa da *board* Nexys3 como conectores pmod são usados para conectar os dispositivo como sensores e rádio transceptor, um USB-UART é usado para comunicação serial com um PC e Adept USB Port são usados para *power on* o FPGA. Nexys3 é compatível com todas as ferramentas CAD da Xilinx, incluindo *ChipScope*, EDK, e o *WebPack free* que será utilizado para construção do circuito (Xilinx, 2014).

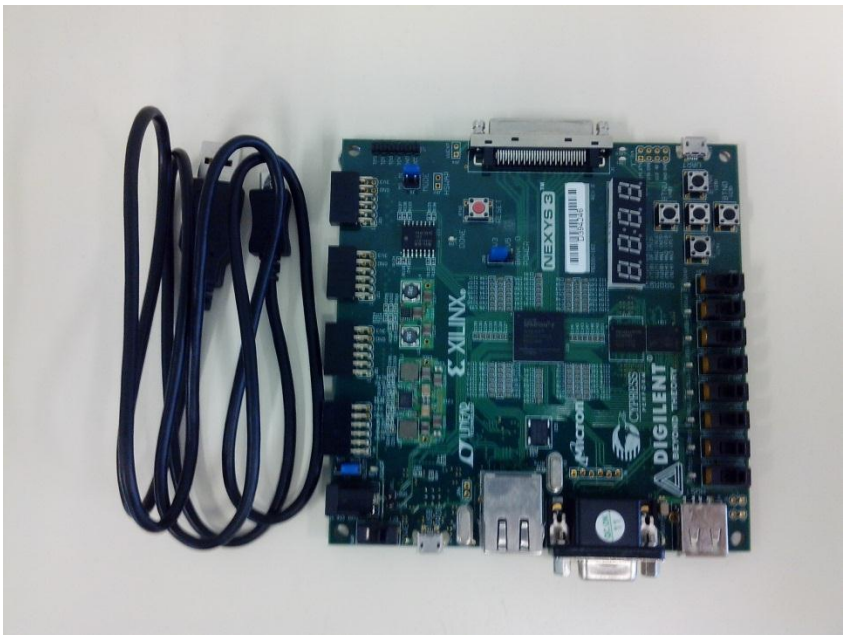

Figura 3 - Dispositivo Xilinx Spartan-6 LX16 FPGA e USB

3.1.1.1 Sistema Adept

Sistema de alta velocidade do Adept Digilent USB2 pode ser utilizada para programar os dispositivos FPGA e PCM, executar testes automatizados na *board*, adicionar dispositivos de I/O virtuais baseados em PC (como botões, *switches* e LEDs) para os projetos FPGA e troca baseada em registros e dados com o arquivo baseado em FPGA. Adept automaticamente reconhece a *board* Nexys3 e apresenta uma interface gráfica com abas para cada uma dessas aplicações. Adept também inclui APIs públicas / DLLs para que possa escrever aplicações para troca de dados com a placa Nexys3 em até 38Mbytes / seg. (Xilinx, 2014).

3.1.2 Radio Transceptor

Módulos XBee-PRO 900 *embedded* RF combina com rápidas redes ponto-a-multiponto com as vantagens de alcance da tecnologia RF de 900 MHz. Construído sobre uma plataforma RF 156 Kbps, esse módulo é ideal para aplicações que requerem baixa latência e maior taxa de transferência de dados. Com *line-of-sight* (LOS) distâncias de até seis milhas com uma antena de

alto ganho, módulo XBee-PRO 900 *embedded* RF também são ideais para aplicações onde os dispositivos são distribuídos a grandes distâncias (Digilent, 2014).

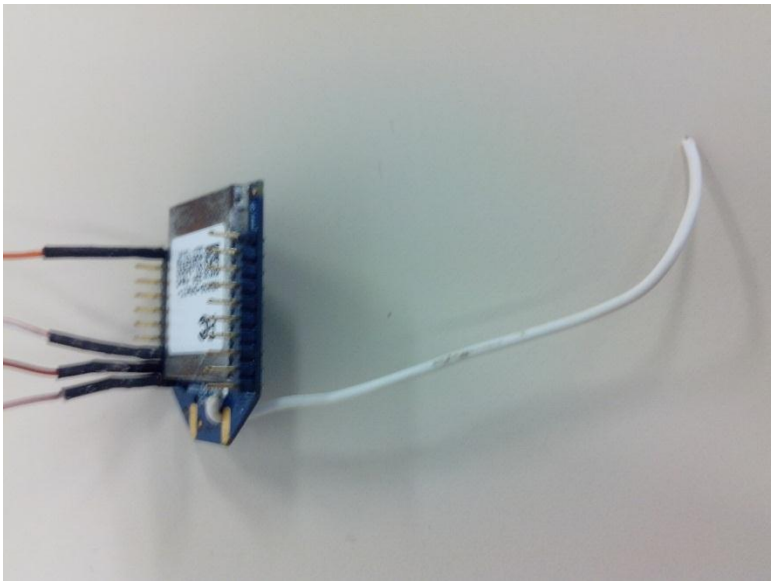

Figura 4 - Módulo XBee-PRO 900 *embedded* RF

3.1.3 Sensor

Com 2.5V - 5.5V de alimentação o LV-MaxSonar® - EZ1 ™ fornece detecção de muito perto à longo alcance e que vão, em um pacote incrivelmente pequeno. O LV-MaxSonar® - EZ1 ™ detecta objetos a partir de 0 polegadas a 254 polegadas (6,45 metros)-e fornece informações de alcance que variam de 6 polegadas para 254 polegadas com resolução de 1 polegada. Os formatos de saída de interface são pulsos da largura da saída, saída analógica de tensão e saída serial digital (Digilent, 2014).

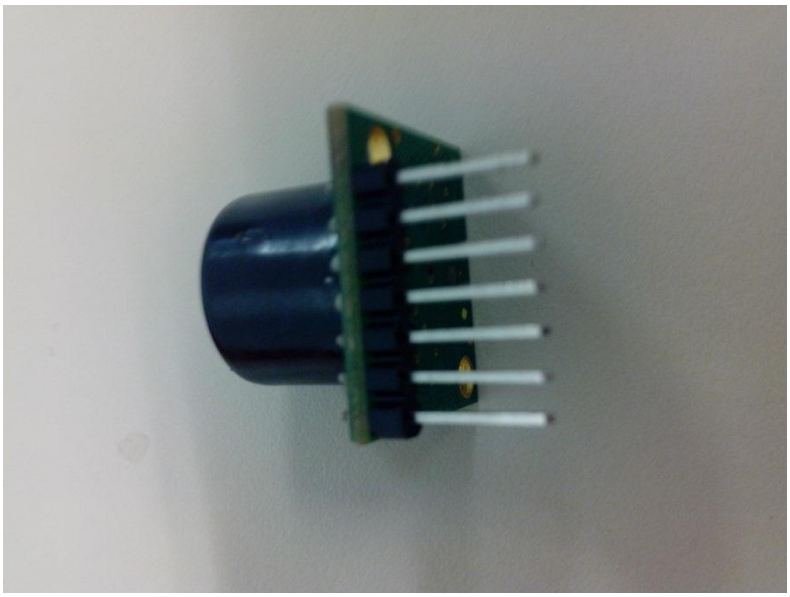

Figura 5 - Módulo LV-MaxSonar® - EZ1

3.1.4 ISE WebPACK Design Software

Software de design ISE® Webpack ™ é totalmente caracterizado por soluções de *design* FPGA *front-to-back* para Linux, Windows XP e Windows 7. ISE Webpack é a solução *downloadable* ideal para FPGA e projeto CPLD oferecendo síntese e simulação HDL, execução, montagem de dispositivos e de programação JTAG. ISE Webpack proporciona um fluxo de *front-to-back* completo que dá acesso instantâneo às funções ISE e funcionalidade, sem nenhum custo (Xilinx, 2014).

O próximo capitulo detalha o modelo de comunicação utilizado para integração dos dispositivos nós sensores. Este modelo de comunicação foi construído nos laboratórios de pesquisa do compsi (UNIVEM).

CAPÍTULO 4

As especificações para comunicação entre as arquiteturas lógicas construídas são baseadas no modelo de comunicação especificado nesta sessão que tem como principal objetivo o equilíbrio entre consumo e disponibilidade de uma rede de sensores sem fio.

Para isso, a estratégia para a redução de consumo tem como base a ativação dos dispositivos que compõe a rede de sensores apenas nos instantes de tempo destinados a comunicação. Sendo que no intervalo de tempo entre troca de pacotes de informações o sistema de comunicação permanecerá em *low power*.

Esta característica exige um modelo de comunicação que estabeleça o sincronismo entre os dispositivos da rede para que nos instantes apropriados estes sejam ativados, estabeleçam a comunicação e possam propagar informações até a Central.

Nesta sessão estão descritas as nomenclaturas utilizadas, as limitações de comunicação e o modelo.

4.1 Janela de Comunicação

A janela de comunicação é formada por todas as operações referentes a um ciclo de comunicação. Após uma janela de comunicação o sistema é configurado para trabalhar em baixo consumo e somente será ativado na próxima janela de comunicação. O intervalo entre janelas de comunicação é chamado de ciclo. A Fig. 6 apresenta a estrutura de uma janela de comunicação. Através desta estrutura um dispositivo pode estabelecer contato com seus dispositivos sucessores e antecessores na rede de sensores.

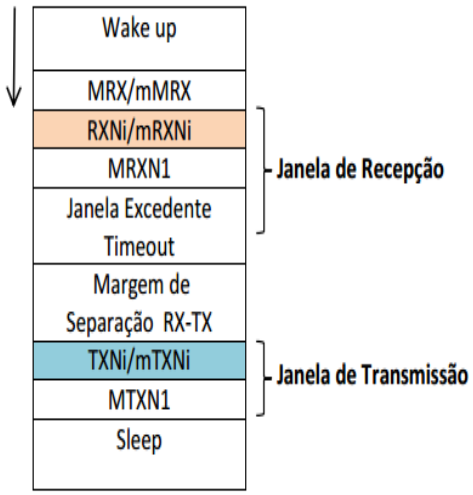

Figura 6 – Janela de Comunicação

O arquitetura lógica apoia-se nesta estrutura para sincronizar e transmitir informações referentes ao estado de cada sensor da rede. Na sequência tem-se a nomenclatura utilizada.

4.2 Nomenclaturas

Janela de Recepção: margem de tempo considerada para a recepção dos pacotes de dados dos dispositivos antecessores na rede de comunicação. Toda recepção de um pacote de dados válido deve gerar automaticamente um pacote de reconhecimento que será transmitido para o dispositivo antecessor.

Janela de Transmissão: Após o processamento do pacote de dados deve-se retransmiti-lo para os dispositivos sucessores na rede. Para todo pacote transmitido deve-se receber um pacote de reconhecimento do dispositivo sucessor.

Janela Excedente: Utilizada para espera de pacotes perdidos. Deve sempre existir mesmo que o sistema identifique que todos os pacotes foram recebidos com sucesso.

Margem de separação RX-TX: margem de tempo destinada a separação entre as janelas de recepção e transmissão evitando a colisão de dados em modo de funcionamento adequado. Tempo utilizado para processamento dos pacotes de dados recebidos e preparação para a retransmissão para os dispositivos sucessores.

Relógio Base: Todo dispositivo possui seu relógio base. Este tem a função de medir o tempo de um ciclo de forma precisa. Este nunca deverá ser interrompido. Uma vez iniciado ele servirá de referência para o sincronismo das regras de comunicação. Em relação ao consumo, caso o ambiente de implementação seja um micro controlador, é possível alocar um timer interno

dedicado e de baixo consumo. No caso de implementações em circuitos integrados isso poderá ser feito com eficiência.

Início dos tempos: o início dos tempos é o momento especifico que um o relógio base é iniciado. Isso deve acontecer automaticamente na primeira ocorrência de comunicação, ou seja, quando o dispositivo está em modo de integração.

Marcadores: determinam o inicio de uma operação relacionado ao Relógio base. Neste documento identificado com um "m" antes de uma nomenclatura definida.

TTPD– Tempo de Transmissão do Pacote de Dados.

TPPD – Tempo de Processamento do Pacote de Dados.

TTPR– Tempo de Transmissão do Pacote de Reconhecimento.

TPPD – Tempo de Processamento do Pacote de Reconhecimento.

TWU– Tempo de WakeUp.

TVC– Tempo de Verificação do Canal.

MRX / mMRX – Margem Inicial de Recepção. Espaço de tempo entre a abertura da janela de recepção e a chegada do primeiro pacote de dados. mMRX = marca o instante da abertura da Margem Inicial de Recepção em relação ao relógio base.

RXNi/ mRXNi– Tempo de Recepção do Pacote de Dados (PD) e Transmissão do Pacote de Reconhecimento (PR). mRXNi= marca instante que o pacote de dados foi recebido em relação ao relógio base.

MRXNi– Margem de Tempo Extra para a Recepção de um Pacote.

TXNi / mTXNi–Transmissão do Pacote de Dados (PD) e Recepção do Pacote de Reconhecimento (PR). mTXNi- marca instante que o pacote dados foi transmitido em relação ao relógio base.

MTXNi– Margem de tempo para a transmissão para o próximo vizinho.

Sleep– Sistema em modo de baixo consumo.

WakeUp– Desperta módulos de RF e processador que estão em modo de baixo consumo. O sistema pode ser acordado parcialmente despertando somente o processador ou módulo RF.

4.3 Definições de Tempos

As fórmulas de temporizações definidas a seguir dependem das especificações do módulo de comunicação RF e do ambiente que será implantado a rede de sensores.

MRX = TWU + 20% da Janela de Recepção (porcentagem varia conforme o ambiente e especificação do Módulo RF)

RXNi = TPPD + TVC + TTPR

MRXNi = K*RXNi, onde K=10 para teste inicial.

Janela Excedente = somatório (RXNi + MRXNi) + Timeout, onde Timeout deve ser calibrado conforme o ambiente.

Janela de Recepção = somatório (RXNi + MRXNi) + Janela Excedente, onde 20% do tempo da Janela de Recepção calculada deve ser atribuído a MRX.

Margem de separação = K*Janela de Recepção +onde K=10 para testes iniciais.

TXNi = TTPD + TPPD + TTPR + TPPR

MTXNi = intervalo de tempo definido pelo dispositivo sucessor na integração do sistema.

Sleep = tempo em modo de baixo consumo. Este determina o equilíbrio do tempo de entrega de pacotes de dados a central e o consumo. Para testes iniciais podem ser considerados valores de teste entre 30s a 10 min

4.4 Topologia

A topologia lógica abstrai a posicionamento físico dos dispositivos da rede transformando-a uma rede sequencial quando a operação executada é a recepção de dados. Desta forma, mesmo que alguns dos dispositivos da rede estejam próximos não poderão colidir, uma vez que existe uma ordem de execução predefinida pelo modelo de comunicação. A Fig. 7 apresenta a topologia lógica formada após a aplicação do modelo proposto.

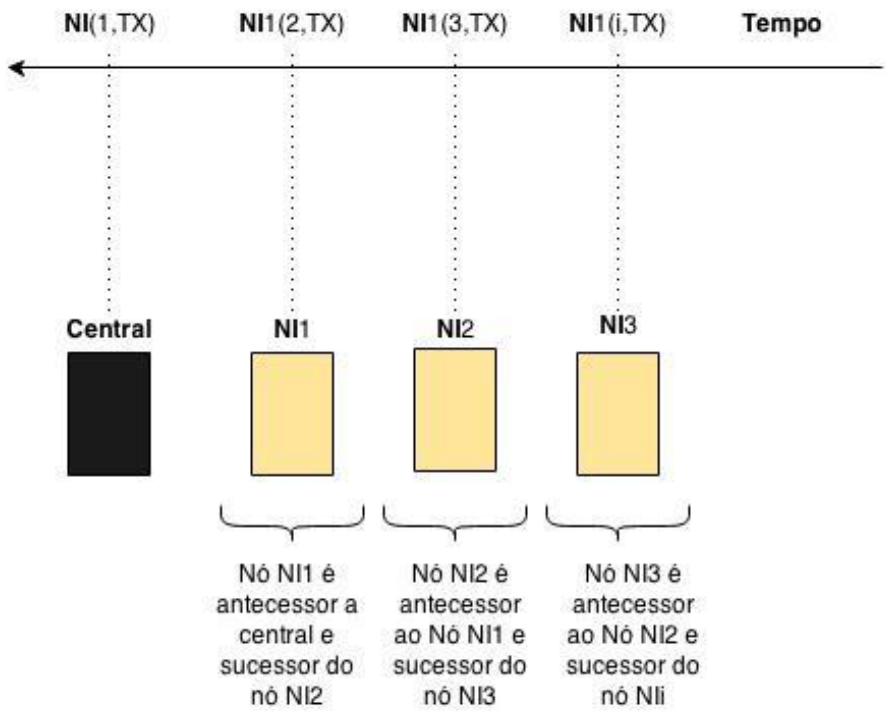

Figura 7 – Topologia Lógica

A janela de transmissão de cada dispositivo estará sincronizada com a janela de recepção de seu dispositivo sucessor, para que estes sequencialmente possam se comunicar como apresentado na Fig. 7.

4.5 Formato dos pacotes

O modelo de comunicação é formado por dois tipos de pacotes: o pacote de dados (PD) e o pacote de reconhecimento (PR). A seguir o formato destes pacotes:

Pacote de dados:

START	TAM_PAC	ID_ORIGEM	ID_DEST	INFO	STOP

Pacote de Reconhecimento:

START	ID_ORIGEM	ID_DEST	CLK	STOP

Os pacotes de informações definidos no protocolo podem ser encapsulados em frames maiores, contendo preâmbulos e controles para a transmissão RF. Sendo que é importante salientar que a adição de novas informações influencia diretamente no consumo de energia.

4.5.1 Pacote de dados

O Pacote de dados (PD) armazena informações referentes aos sensores da rede que indicam seu estado corrente. A seguir detalhes do pacote de dados.

- [START] – (2 Bits) Indica o início do pacote. Os valores lógicos "01" consecutivos indicam o início do pacote.
- [TAM_PAC] – (16 bits) Indica o tamanho do pacote.
- [ID_ORIG] – (8 Bits) identifica o dispositivo origem, ou seja, o dispositivo que montou o pacote.
- [ID_DEST] – (8 Bits) identifica o dispositivo destino, ou seja, o dispositivo que receberá o pacote.
- [INFO] - (256 Bits) indica o estado da rede de sensores.
- [STOP] – (2 Bits) Indica o final do pacote. Os valores lógicos "10" consecutivos indicam o final do pacote.

4.5.2 Pacote de reconhecimento

O Pacote de Reconhecimento (PR) armazena informações referentes ao reconhecimento de um pacote de dados, sincronismo por meio do ajuste de relógio (CLK). A seguir detalhes do Pacote de Reconhecimento (PR).

- [START] – (2 Bits) Indica o início do pacote. Os valores lógicos "01" consecutivos indicam o início do pacote.
- [ID_ORIG] – (8 Bits) identifica o dispositivo origem, ou seja, o dispositivo que montou o pacote.

- [ID_DEST] – (8 Bits) identifica o dispositivo destino, ou seja, o dispositivo que receberá o pacote.
- [CLK] – (16 Bits) indica o relógio do transmissor. O relógio é representado com a precisão de milissegundos.
- [STOP] – (2 Bits) Indica o final do pacote. Os valores lógicos "10" consecutivos indicam o final do pacote.

4.6 Regras de operação em modo normal

Em modo normal de operação o dispositivo basicamente recebe pacotes de dados e emite pacotes de reconhecimento contendo principalmente o ajuste de relógio para seu antecessor. Posteriormente os dados são processados e retransmitidos para os nós sucessores que por sua vez emitem pacotes de reconhecimento nos tempos estabelecidos no processo de integração.

4.6.1 Regras para dispositivos intermediários

A Fig. 3 ilustra todo o processo do modo normal de operação de um dispositivo intermediário no decorrer do tempo. Na sequência regras para operação em modo normal de um dispositivo que já está integrado e comunicando com seu sucessor e antecessor, ou seja, o relógio base e as marcações de ações estão equacionados.

1. Consulta Relógio Base. Espera pela marcação de tempo MRX (Margem da janela de recepção)
2. Abre janela de recepção (modulo RF chaveado para RX).
3. Detecta chegada do ID esperado.
4. Compara marcador mRXNi com Relógio Base, ou seja, compara se o pacote chegou no tempo esperado. Caso esteja incorreto montar pacote de reconhecimento com o ajuste do relógio.

Fórmula: Valor de Ajuste = Relógio Base – mRXNi

5. Verificar canal e, transmitir pacote de reconhecimento para o dispositivo antecessor comunicante (modulo RF chaveado para TX).
6. Espera marcação mTXNi = Relógio Base
7. WakeUp (módulo RF chaveado para TX).
8. Transmite pacote de dados para dispositivo sucessor.
9. Recebe pacote de reconhecimento (PR) e ajusta a marcação mTXNi.
10. Entrar em Sleep completo.
11. Fim de um ciclo. Retornar ao passo 1.

4.6.2 Regras para dispositivos finais

Os dispositivos finais não possuem antecessores, assim basta executar os passos de 6 a 11 anteriormente descritos.

4.7 Limitações do ambiente de comunicação

Para o funcionamento correto da aplicação é necessário que haja algumas configurações de acordo com o ambiente de comunicação e disponibilidades de canais e dispositivos físicos para funcionamento adequado.

4.7.1 Numero de vizinhos de comunicação

Cada dispositivo do sistema, na primeira versão do protótipo, terá um único sucessor e antecessor (no caso de dispositivos intermediário). Uma vez criado e validado o número de vizinho pode ser escalado considerando as metas de consumo de energia.

4.7.2 Disponibilidade do canal de comunicação

O sistema de comunicação RF depende estritamente da disponibilidade e qualidade do canal de comunicação. Desta forma, aparelhos defeituosos, ambiente ruidosos ou qualquer outros fatores correlatos poderão afetar diretamente no funcionamento desejado do modelo de comunicação.

4.7.3 Programação e instalação

Os dispositivos devem ser configurados antes de serem instalados. Cada dispositivo ou nó sensor deve ser definido como final ou intermediário e receberá um ID único.

Na próxima sessão será especificado toda a implementação da arquitetura lógica do nó sensor, analisado o desempenho e consumo de energia dinâmica de alguns módulos principais do circuito.

CAPÍTULO 5

5.1 Arquitetura Lógica

A arquitetura lógica consiste de vários componentes lógicos conectados. Utilizando da ferramenta Xilinx WebPack ISE 13.4 são descritos as funções lógicas dos módulos usando a linguagem VHDL. Para o consumo consciente de energia dinâmica a arquitetura lógica implementa um modelo de operação com a habilidade de realizar um conjunto de tarefas com apenas os módulos para tomada de decisões ativos, para decidir quais componentes realiza o processamento da lógica podendo ser interrompido ou suspenso em uma determinada situação (Nisbet, 2004).

Desta forma, os módulos são construídos de forma a integrarem-se logicamente, suspendendo os módulos que não são necessários. Os dispositivos físicos conectados à interface externa do FPGA também são interrompidos ou suspensos através de seu respectivo módulo lógico.

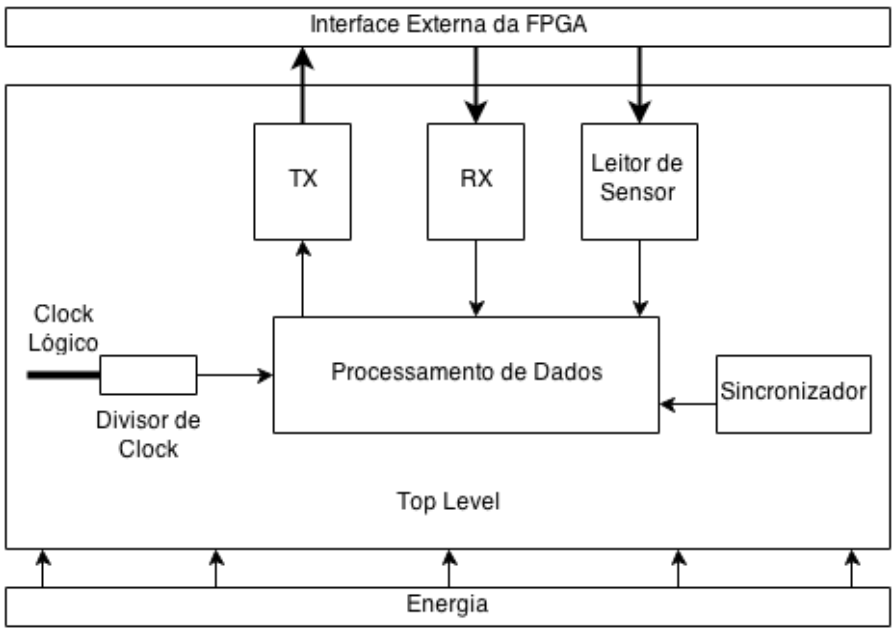

Figura 8 - Arquitetura lógica em blocos

Na Fig. 8 tem se a arquitetura lógica em blocos. Para formar a arquitetura os módulos estão conectados e, seus tempos de execução são gerenciados pelo módulo de sincronização, as frequências de operação são gerenciadas pelo módulo divisor, enquanto que os outros módulos como TX, RX, processamento de dados e um leitor de sensor genérico são projetados para realizar o ciclo de comunicação.

5.1.1 Módulos Lógicos

Nesta sessão, é descrito de forma detalhada as principais funcionalidades dos módulos implementados na Xilinx Spartan-6 LX16 FPGA, e os detalhes de consumo de lógica de cada um. Mesmo sendo módulos separados alguns são dependentes, sinais lógicos são usados para se comunicarem entre si. Estes módulos são nomeados como leitores de sensores, transmissor (TX), receptor (RX), processamento de dados, divisor de frequência, sincronização e top level, este ultimo, um módulo principal que conectar todos os outros módulos lógicos e permite a conexão através da interface externa do FPGA com os módulos físicos externos, cada um com seu respectivo módulo lógico. Abaixo estão os detalhes da implementação de cada um dos módulos lógicos que formam um core genérico para nó sensor onde cada um exerce sua função para e realização do ciclo de comunicação.

5.1.2 Processamento de dados

Este módulo tem como função processar os dados recebidos e que serão transmitidos, os módulos lógicos podem ser conectados ao modulo de processamento de dados e configurado. Os dados recebidos pelo módulo receptor e dados lidos do módulo leitor de sensor são enviados diretamente para o processamento, após o processamento esses dados são enviados para o módulo transmissor, para transmissão. Dependendo da frequência utilizada o desempenho é afetado, quando se opera com a frequência mínima torna possível o consumo mínimo de energia dinâmica pelo módulo lógico, mas uma maior quantidade de falha na comunicação e perda de desempenho quando comparado com a frequência máxima seu consumo de energia é maior e é identificado um melhor desempenho e maior confiabilidade na comunicação. Os dados são processados de uma forma genérica em que apenas o tamanho do pacote é relevante para

arquitetura. Somente as informações necessárias para o processamento do pacote estão incluídas. Um ciclo de processamento é caracterizado a partir do recebimento do pacote a ser processado até o recebimento de um pacote de confirmação deste pacote transmitido.

Tabela 1 - Consumo Lógico do Módulo de Processamento de Dados

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	306	18224
Numero de Slice LUTs	2861	9112

Na Tab. 1 tem se o numero de registradores e *slices* LUTs utilizados para construir o módulo. Para o processamento dos pacotes de dados um grande número de *Slices* LUTs e outros registradores são utilizados, isso acontece pela necessidade de processar tanto pacotes de dados recebidos a partir do módulo de leitura do sensor quanto do módulo receptor (RX).

5.1.3 Modulo Genérico Leitor de Sensor

Para validar a comunicação da arquitetura um módulo genérico para leitura do sensor foi implementado, este módulo simplesmente lê os dados do sensor de acordo com o definição de transmissão e recepção do sensor utilizado e após a formação do pacote de dados entrega-o para o modulo de processamento de dados.

Tabela 2 - Consumo Lógico do Módulo Leitor de Sensor

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	548	18224
Numero de Slice LUTs	674	9112

Na Tab. 2 tem se detalhes lógicos deste módulo, o numero de *Slices Registers* e *Slices* LUTs utilizados. Por ser um módulo genérico consome bastante lógica do FPGA.

5.1.4 Modulo Transmissor (TX)

O módulo lógico transmissor tem sua saída lógica (tx) conectada à interface externa da FPGA que liga a entrada (RX) do rádio transceptor. O módulo lógico funciona em vários *baud rate*, 1.200 bps a 230.400 bps, e possui um limite de tamanho do pacote a ser transmitido, o pacote a ser transmitido pode conter diferentes tamanhos, o tamanho é identificado na recepção e, em seguida, o transmissor é relatado. A transmissão é feita byte por byte, cada byte é adicionado um bit de *stop* e *start*. Os tempos de execução do modulo transmissor são controlados pelo módulo de sincronização (os tempos são definidos na configuração). O módulo transmissor tem controle do dispositivo físico, radio transceptor, podendo ativa-lo eu desativa-lo quando necessário.

Tabela 3 - Consumo Lógico do Módulo Transmissor (TX)

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	177	18224
Numero de Slice LUTs	298	9112

A Tab. 3 mostra os detalhes lógicos deste módulo. É consumida pouca lógica da FPGA por este módulo, uma vez que apenas realiza a transmissão dos pacotes de dados.

5.1.5 Modulo Receptor (RX)

O módulo receptor tem sua entrada lógica (RX) conectada a interface externa do FPGA que conecta na saída (TX) do rádio transceptor. O módulo funciona com configurações de *baud rate* iguais ao módulo transmissor, 1.200 bps a 230.400 bps. O pacote é formado de byte por byte, onde cada byte é adicionado um bit *start* e *stop*. O receptor identifica o tamanho atual do pacote a ser recebido, tornando-o um receptor flexível. Assim como o módulo transmissor, as execuções do módulo receptor são em tempos pré-definido pelo módulo de sincronização. Os tempos de execuções do módulo receptor são controlados pelo módulo de sincronização, após habilitar o módulo receptor para execução, o próprio módulo receptor ativa o rádio transceptor

para realização da recepção de dados, quando terminado os dois módulos lógico e físico são desativados para economizar energia.

Tabela 4 - Consumo Lógico do Módulo Receptor

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	316	18224
Numero de Slice LUTs	623	9112

A Tab. 4 mostra o consumo lógico do módulo RX. Fazendo uso de registradores para armazenar dados recebidos, uma grande quantidade de lógica da FPGA é utilizada, uma vez que o módulo tem a característica de recepção de pacotes de dados com diferentes tamanhos.

5.1.6 Módulo de sincronização

Este módulo é responsável pelos tempos de execução dos módulos do circuito; sua lógica é baseada em um relógio contador que após os tempo configurado para a primeira janela de comunicação ele é usado como referência para *power-on* e *power-off* os módulos lógicos de acordo com os tempos acertados na sincronização dos nós sucessores e antecessores. Os tempos de *power-off* da inicio no *sleep* e os tempos de *power-on* tem inicio no *Wake-up* dos módulos.

Sleep: Depois de completar o ciclo de comunicação os dispositivos físicos conectados ao FPGA com exceção da FPGA são desativados e o circuito não é executado, somente o módulo de sincronização continua a funcionar.

Wake up: Neste caso em que a FPGA não é desligada, mesmo quando em *sleep* o circuito não perde a sua configuração, os módulos lógicos são acordados pelo módulo de sincronização e consequentemente seus respectivos dispositivos físicos para realização do ciclo de comunicação.

Tabela 5 - Consumo Lógico do Módulo de Sincronização

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	38	18224
Numero de Slice LUTs	89	9112

A Tab. 5 demonstra que este módulo usa pouca lógica do FPGA, uma vez que é apenas baseado em um relógio contador que marca os tempos de execução para outros módulos.

5.1.7 Modulo Divisor de Clock

Com uma entrada lógica de clock de 10MHz com base em um oscilador CMOS de 100 MHz da FPGA o módulo divisor gera frequências denominadas A e B e são distribuídas entre os módulos lógicos que compõe a arquitetura lógica. Esta distribuição é aplicada, a fim de, diminuir o acúmulo de clock usado pelo circuito. Os módulos lógicos que são conectados ao I / O da FPGA para estabelecer conexão com os módulos físicos são configurados de acordo com a frequência de operação do dispositivo físico, enquanto os outros têm a sua frequência que pode variar de 100 kHz a 1 MHz.

Tabela 6 - Consumo Lógico do Módulo Divisor de Clock

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	26	18224
Numero de Slice LUTs	63	9112

A Tab. 6 demonstra que para este módulo, pouca lógica do FPGA é utilizada, por ser apenas um simples divisor de clock, tendo como entrada um clock logico de 10MHz e duas saída denominadas como frequência A e B.

5.1.8 Módulo Top Level

Este módulo conecta todos os módulos que fazem parte da arquitetura, os sinais lógicos e outros tipos de conexão que permite que os módulos se comuniquem uns com os outros. Também através deste módulo o módulo lógico e dispositivo físico é conectado usando as I/O da FPGA. Para simulação da arquitetura um total de oito I / Os foram utilizadas.

Tabela 7 - Consumo Lógico do Top Level

Logica Usada	**Usada**	**Disponível**
Numero de Slice Registers	2	18224
Numero de Slice LUTs	1	9112

A Tab. 7 demonstra que pouca lógica da FPGA é usada para construir este módulo, ele só conecta os módulos para formar a arquitetura.

5.2 Desempenho e Analises de Consumo de Energia

Para conseguir um bom desempenho e aceitável consumo de energia dinâmica do circuito algumas técnicas como módulo divisor de clock para diminuição da distribuição de clock pelo circuito e um rígido controle de consumo de energia no circuito através de um módulo de sincronização são utilizados. As diferentes maneiras de analisar a energia dinâmica da arquitetura lógica são detalhadas nesta sessão.

5.2.1 Sincronização da Rede

A sincronização da rede inicia na primeira janela de comunicação de cada nó, os tempos de sincronização após a primeira janela de comunicação fica definido entre os nós sucessor e antecessor. No período de *sleep* o módulo de sincronização é o único módulo que fica em execução enquanto os outros não são executados. No tempo definido pela sincronização, os módulos iniciam execução e reestabelece a comunicação com seu sucessor.

5.2.2 Frequência de clock

O processador físico do FPGA opera com um oscilador CMOS de 100 MHz, mas é utilizado um clock lógico de 10 MHz para simulação de consumo de energia dinâmica do circuito. Este clock logico de 10MHz é usado como base para gerar frequências denominadas A e B. Estas duas frequências de operação são distribuídos pelo circuito, uma das frequências definidas como frequência A é utilizada para ligar os módulos lógicos que fazem conexão com a interface I / O da FPGA. Estas I/Os tem uma conexão direta com os dispositivos físicos usados para validar a arquitetura, dispositivos como sensores e rádio transceptor fazem uso destas portas configuradas com suas relativas frequências de operação. A frequência definida como, frequência B, está ligada aos módulos que não fazem conexão com a interface I / O da FPGA e podem ser configurados de acordo com a frequência de operação desejada.

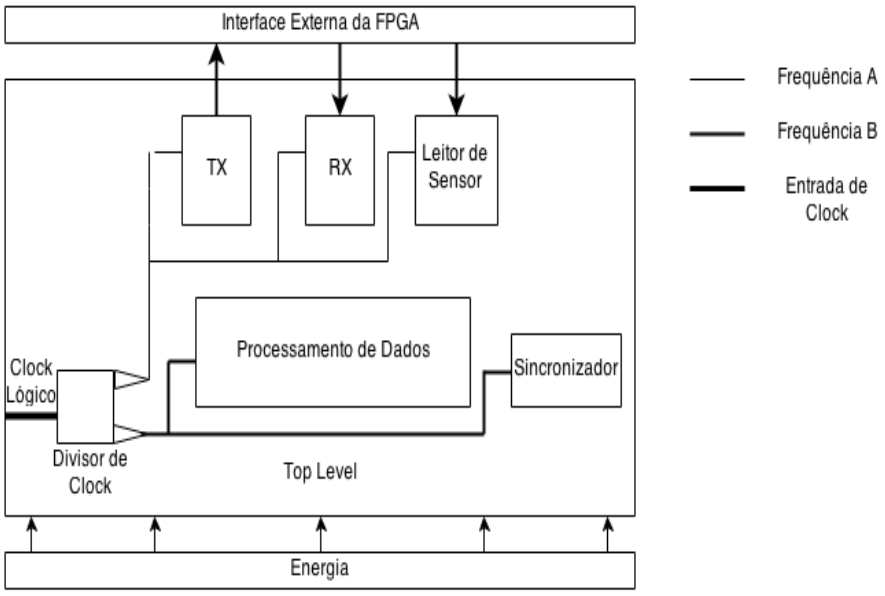

Figura 9 - Distribuição lógica de frequências entre os módulos da arquitetura

A Fig. 9 mostra a arquitetura lógica construída em blocos, cada um destes módulos são conectados logicamente, tem se também a distribuição de frequências pelo circuito. O consumo de energia dinâmica do circuito quando se opera com um domínio de clock para cada módulo lógico foi uma média de 23mW. Após a aplicação da técnica para gerar frequências a partir de um único domínio de clock, o consumo de energia dinâmica do circuito é inferior a 3mW.

Tabela 8 - Consumo de Energia Dinâmica pelo Clock e Frequências

Clock e Frequência	**Média de Consumo**
Clock de 10MHz	0,81mW
Frequência A de 100KHz	0.38mW
Frequência B de 1MHz	0.60mW

Na Tab. 8 é apresentado o consumo de energia dinâmica do clock e das frequências geradas a partir do módulo divisor de clock. Esta técnica permite que qualquer módulo lógico possa ser integrado à arquitetura utilizando uma das duas frequências.

5.2.3 Modos de Analises de Consumo de energia Dinâmica da Arquitetura

O consumo de energia foi medido utilizando a ferramenta de análise Xilinx XPower. Na sequência tem a média de consumo de energia dinâmica do circuito em diferentes modos de operação. Os modos de analise do consumo de energia dinâmica do circuito são divididos em alguns módulos em específicos; tais como, módulos que só funcionam em determinados momentos definidos após a integração da arquitetura lógica, portanto, analise do consumo de energia dinâmica de toda a arquitetura será discutido no item 6.1.

Quando a frequência de operação do módulo de processamento de dados é inferior a 100 kHz, não é possível executar o processamento de dados, então a análise do consumo de energia dinâmica do circuito é realizada com frequências superiores a 100 kHz distribuídas entre os módulos lógicos. Quando apenas o módulo de sincronização está executando a arquitetura lógica consome uma média de energia dinâmica de 0.48mW, neste caso nenhum outro módulo é executado.

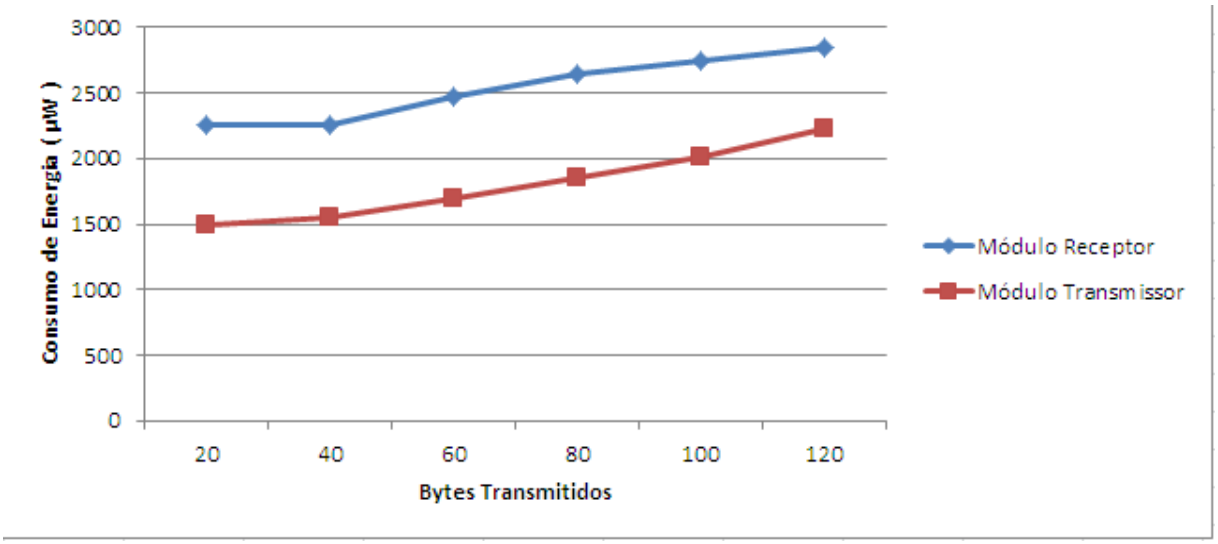

Figura 10. Média do consumo de energia dinamica pelos módulos receptores e transmissores

Na Fig. 10 tem se o consumo de energia dinâmica dos principais módulos, transmissor e receptor, com uma frequência de operação de 1MHz o seu consumo aumenta de acordo com a quantidade de bytes a ser transmitidos e recebidos, o consumo destes dois módulos tem grande impacto na arquitetura, uma vez que, quanto mais bytes a serem transmitidos mais energia dinâmica será usada pela arquitetura.

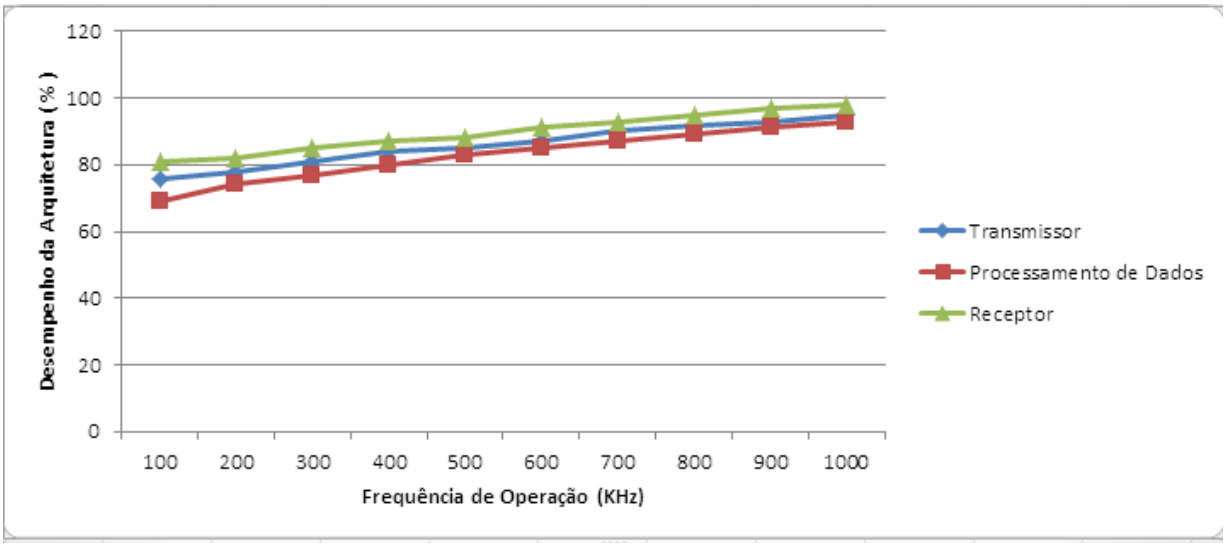

Figura 11. Desempenho de alguns módulos em diferentes frequências de operação

Na Fig. 11, o desempenho de cada módulo chave da arquitetura é apresentado. O módulo de Processamento de Dados tem um desempenho inferior, por ser dependente de outros módulos.

O módulo receptor tem uma grande vantagem quando se opera em qualquer frequência enquanto que o desempenho do transmissor é um pouco menor.

Na próxima sessão será detalhado os resultados conseguidos a partir da comparação com trabalhos correlatos.

CAPÍTULO 6

6.1 Trabalhos correlatos

Outros trabalhos foram realizados para conseguir consumo de energia dinâmica eficiente em FPGA. Nesta sessão apresentaremos alguns destes trabalhos para comparação. Existem algumas técnicas do DPM (*dynamic power management*), tais técnicas tem como objetivo reduzir o impacto do consumo de energia em FPGA (Dargie, 2012).

Os resultados experimentais do consumo de energia dinâmica de nós sensores implementado em *Flash Based*-FPGA pode ser mantida abaixo de 4mW, mesmo com altas taxas de amostragem (Grassi, 2012). Enquanto que nesta implementação usando *Sram Based*-FPGA e aplicando as técnica apresentadas, mesmo com altas taxas de amostragem o consumo de energia dinâmica mantem abaixo 3mW. Isto é possível através da replicação de frequência, com uma única entrada de clock várias frequência são geradas. Assim, nesta pesquisa, a razão de usar o FPGA para desenvolvimento de nós sensores visa reduzir o número de domínios de clock do circuito, uma vez que é o principal responsável pelo consumo de energia dinâmica do circuito quando implementados em FPGA. A técnica utilizada alcançou simultaneamente eficiência por meio dos recursos de FPGA e gerenciamento de energia dinâmica através da implementação do circuito de consumo consciente de energia. Para assegurar que foi alcançado o objetivo com a técnica utilizada, uma comparação entre implementações de nós sensores a partir de diferentes tecnologias FPGAs é realizado. Aplicando a técnica de gerenciamento de clock na Xilinx 6 é possível alcançar um rendimento máximo do sistema com menor frequência do que o trabalho realizado em uma Xilinx 3, neste caso foi analisado os módulos de transmissão e recepção (Liao, 2013).

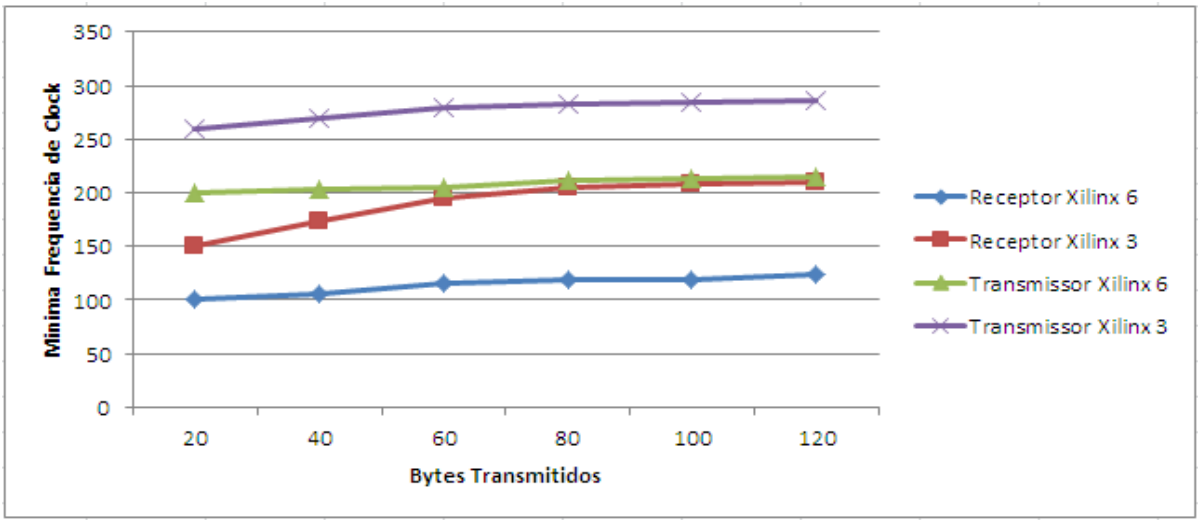

Figura 12 - Comparação entre diferentes arquiteturas FPGAs para alcançar máxima *throughput* na transmissão e recepção

Na Fig. 12 uma comparação entre dois trabalhos é realizada. Conclui que a técnica utilizada DPM (*dynamic power management*) tem um bom desempenho na Xilinx 6 exigindo menor frequência de clock para manter 250 kbit/s de *throughput* do sistema de transferência.

6.2 Discussão e Proposta

Um dos destaques do projeto é que após a aplicação da técnica de gerar frequências com um único domínio de clock foi obtido um ganho de 87% de energia dinâmica em comparação com a implementação da primeira versão do protótipo usando um domínio de clock para cada módulo lógico construído. A visão do trabalho privilegia o consumo de energia dinâmica conseguida após a realização de simulações na tecnologia FPGA, assim propomos a avaliação e comportamento da mesma arquitetura lógica em uma tecnologia ASIC que tem a característica *low-power*, avaliando consumo de energia dinâmica e estática do nó sensor.

Foi comparado três grupos de nós sensores: autônomo, combinações de microcontroladores e FPGAs e co-processadores FPGA para nós experimentais cuja análise do consumo de energia foi dentro do consumo de energia dos nós sensores comerciais e, em alguns casos, foi reduzida várias vezes (Piedra, 2012).

Afirmamos que a arquitetura lógica implementada na Spartan-6 LX16 FPGA consome menos de 3 mW em média de energia dinâmica e um consumo médio de energia estática de 22mW, com esses dados afirmados a arquitetura implementada não pode competir com este grupo de nós sensores que tem seu consumo dentro dos padrões comerciais, porque a nossa FPGA utilizada para a implementação não é considerada uma tecnologia de baixo consumo de energia. Por outro lado, se aplicarmos o mesmo algoritmo em uma tecnologia eASIC Nextreme-2, podemos obter uma redução de até 80% no consumo de energia quando comparado com FPGAs (NEW ASICs, 2011).

Assim, considerando-se um ganho de energia de 80% a partir da arquitetura implementada na Spartan-6 LX16 FPGA a mesma arquitetura lógica em uma eASIC Nextreme-2 terá média de consumo de energia tanto dinâmica como estática de 5mW.

Em Anexo segue os códigos dos módulos lógicos da arquitetura construídos na linguagem VHDL. O primeiro é o código referente ao módulo Transmissor responsável pela transmissão dos pacotes de dados, o segundo é o código do módulo Receptor responsável pela Recepção de pacotes de dados, o terceiro é o código do módulo Leitor de Sensor que faz leitura dos dados enviados pelo sensor, o quarto é o código do módulo de Processamento de Dados, o quinto é o código do modulo responsável pela sincronização das arquiteturas, o sexto é o código do módulo Divisor de Clock e o sétimo é o código do modulo Top que faz conexões de módulos lógicos e conexões dos módulos lógicos com os módulos físicos.

CAPÍTULO 7

7.1 Conclusão e trabalhos futuros

Rede de Sensores sem fio emerge com grande oportunidade para serem exploradas como tecnologias de baixo consumo de energia. Aproveitando-se de tecnologias como o FPGA que está se tornando cada vez mais dispositivos de processamento de baixo consumo, é possível a implementação de grandes aplicações que não só consome pouca energia, mas também gerencia sua própria energia. O trabalho visa à implementação de nós sensores em tecnologias FPGA para simulações e analises de consumo de energia dinâmica da arquitetura lógica. Estas tecnologias têm grandes utilidades em situações onde há restrição de energia, ambiente de difícil acesso, podendo ser aplicados e operados autonomamente, ou seja, sem intervenção humana. A arquitetura lógica construída, que são nós sensores, possuem habilidades necessárias para realizar operações de transmissões, recepções e processamento de dados de forma autônoma, além de a rede ser dirigida pelo modelo de comunicação é possível gerenciar o consumo de energia dinâmica do circuito através de um módulo lógico de sincronização ativando e desativando os módulos físicos que compõem o nó sensor. Apesar de a tecnologia FPGA utilizada não oferecer características *low-power*, foi utilizado como base o consumo de energia estática e dinâmica da arquitetura lógica construída para analise em uma tecnologia eASIC Nextreme-2. As análises se mostraram muito interessantes quando comparado com outros tipos de processador.

O projeto oferece novas oportunidades para trabalhos futuros, como incrementar o modelo de comunicação, algoritmo para regeneração da rede e proposta para a migração da arquitetura lógica construída para um dispositivo ASIC, como por exemplo, uma tecnologia eASIC Nextreme-2 que consome menos energia dinâmica e possui recursos de gerenciamento de energia estática.

Referências

Altera Corporation, Using the ClockLock & ClockBoost Features in APEX Devices, Application Note 115 (ver. 1.0), May 1999.

Anderson, J.H., Najm, F.N., Low-Power Programmable FPGA Routing Circuitry, IEEE transactions on very large scale integration (VLSI) systems, 2009.

Arampatzis, Th., Lygeros, J., Manesis, S., A Survey of Applications of Wireless Sensors and Wireless Sensor Networks, 13th Mediterranean Conference on Control and Automation Limassol, Cyprus, pp. 719-724, 2005.

Aula 3 - Introdução ao VHDL: [Online]. Disponível: http://www2.ufersa.edu.br/portal/view/uploads/setores/147/arquivos/MM/MM_Aula03%20-%2020102.pdf

Berndt A., Hollmeijer, P. Havinga, J. Hurink. Introdução a Redes de Sensores sem Fio: Departamento de Ciência da Computação – Universidade do Estado de Mato Grosso (UNEMAT).

Bhattacharya, S., Kim, H., Prabh, S., Abdelzaher, T., Energy-Conserving Data Placement and Asynchronous Multicast in Wireless Sensor Networks. MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services, p.173-185, USENIX, 2003.

Brown, S., An Overview of Technology, Architecture and CAD Tools for Programmable Logic Devices, in Proc. of IEEE Custom Integrated Circuits Conf., 1994, pp. 69-76,1994.

Brynjolfson I., Zilic Z.,: FPGA Clock Management for Low Power, Department of Electrical and Computer Engineering, McGill Universit, 2000.

Dargie, W.: Dynamic Power Management in Wireless Sensor Networks: State-of-the-Art: I.J. Information Engineering and Electronic Business, 2012.

Dulman, S., Chattterjea, P., Introduction to wireless sensor network, Embeded systems Handbook, p. 31.1 – 31.10, 2006.

Dulman, S., Havinga, P., Architectures for wireless sensor networks. Embeded systems Handbook, p. 33.1 – 33.22, 2006.

eASIC Nextreme-2 45nm NEW ASICs, 2011. [Online]. Available: http://www.easic.com/Spatr7ve/website-wp1/wp-content/uploads/2011/ 02/eAS IC-Nextreme-2-Product-Brief.pdf

Edson T. Midorikawa: Uma Introdução às Linguagens de Descrição de Hardware: EPUSP - PCS 2304 - Projeto Lógico Digital, 2007.

Grassi, P. R., Sciuto, D.: Energy-Aware FPGA-Based Architecture for Wireless Sensor Networks: 15th Euromicro Conference on Digital System Design, pp.866-873, 2012.

Greene, J., Hamdy, E., Beal, S., Antifuse Field Programmable Gate Arrays, Proc. IEEE, vol. 81, no. pp. 1042-1056, 1993.

Grover, N., Soni, M., Reduction of Power Consumption in FPGAs - An Overview, Information Engineering and Electronic Business, 2012.

Han, K., Chan, N., Kim, S., Flash-based Field Programmable Gate Array Technology With Deep Trench Isolation, in Proc. of IEEE Custom Integrated Circuits Conf., pp. 89-91, 2007.

ISE WebPACK Design Software, Xilinx, 2014. [Online]. Available: http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm

Kuon, I., Rose, J., Measuring the Gap Between FPGAs and ASICs, ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 21-30, 2006.

Kuon, I.; Tessier, R.; Rose, J. FPGA Architecture: Survey and Challenges. Found. Trends Electron. Des. Autom. Vol.2, No.2, pp. 135–253, 2008.

Liao, J., Singh, B., FPGA based wireless sensor node with customizable event-driven architecture: EURASIP Journal on Embedded Systems, 2013.

Loureiro A., Nogueira J., Ruiz L.., Mini R., Nakamura E., Figueiredo C.: Redes de Sensores Sem Fio: Departamento de Ciencia da Computacão Universidade Federal de Minas Gerais, 2003.

Lucent Technologies, ORCA Series 3C and 3T FPGAs, Data Sheet, June 1999.

LV-MaxSonar® - EZ1™, Digilent, 2014. [Online]. Available: http://maxbotix.com/documents/MB1010_Datasheet.pdf

Macedo, D., Nogueira, J., Loureiro, A., Um protocolo de roteamento para redes de sensores sem fio adaptável por regras de aplicação, 2006.

Modulo FPGA, National Instruments, 2011. [Online]. Disponível: http://zone.ni.com/reference/en-XX/help/371599G-01/lvfpgaconcepts/fpga_basic_chip_terms

Nisbet, A.; Dobson, S.: A systems architecture for sensor networks based on hardware/software co-design, 1st IFIP Workshop on Autonomic Communications, 2004.

Piedra, A.; Braeken, An.; Touhafi, A.: Sensor Systems Based on FPGAs and Their Applications: A Survey, Sensors 2012.

Reduce FPGA Power Consumption, eASIC corporation, 2014. [Online]. Available: http://www.easic.com

Skliarova, I., Ferrari, A., Introdução à computação reconfigurável, Revista do Detua, vol. 2, Nº 6, 2003.

Spartan 6 LX16 Nexys3, Xilinx, 2014. [Online]. Available: http://www.xilinx.com

Suresh, S.; Beldianu, S.,F.; Ziavras, S.,G.: FPGA and ASIC Square for high Performance and Power Efficiency: 24th IEEE International Conferenceon Applicatio-specific Systems, Architectures and Processors, pp. 269-272, 2013.

Valverde , J., Otero, A., Lopez, M., Portilla, J., de la Torre, E., Riesgo, T.: Using SRAM Based FPGAs for Power Aware High Performance Wireless Sensor Networks: Sensors 2012.

Xilinx Inc., Using the Virtex Delay-Locked Loop (XAPP132 Version 1.31), Advanced Application Note, October, 21, 1998.

XBee-PRO 900, Digilent, 2014. [Online]. Available: https://www.sparkfun.com/datasheets/Wireless/Zigbee/XBee-900-Datasheet.pdf

Yiannacouras, P., Steffan, J.G., Rose, J., Exploration and Customization of FPGA-Based Soft Processors: Computer-Aided Design of Integrated Circuits and Systems, 2006 IEEE.

Yu, F., Jain, R.: A Survey of Wireless Sensor Network Simulation Tools: I.J. Information Engineering and Electronic Business, 2011.

Anexo 1.

Item A – Modulo Transmissor

```
entity TRANSMISSOR _TX is

            port ( clock : in STD_LOGIC;
                   valor : in STD_LOGIC_VECTOR(127 downto0);
                   saida : out STD_LOGIC;
                   cont_trans : in STD_LOGIC;
                   rst : in STD_LOGIC;
                   cee : in STD_LOGIC;
                   tam_pac_tx : in integer range 0 downto 160;
                   busy : out STD_LOGIC
                 );
end TRANSMISSOR_TX;

architecture Behavioral of TRANSMISSOR _TX is

begin
process(clock,rst)
variable bytes_saida : integer range 0 to 9 := 0;
variable tam_aux : integer range 0 to 160 := 0;
variable count_vector : STD_LOGIC_VECTOR(15 downto 0);
variable stop : integer range 0 to 160 := 0;
variable bufer : STD_LOGIC_VECTOR(127 downto 0);
variable cont : integer range 0 to 20834 := 0; -- 4800 bps
variable busy_ocup : STD_LOGIC := '0';

begin
    if rst = '1' then -- resetar
                          busy <= '1';
                          saida <= '1';
                          bytes_saida := 0;
                          busy_ocup := '0';
                          cont := 0;
                          stop := 0;
                          count_vector := (others=>'0');

          elsif clock'event and clock = '1' then
            if cont_trans = '1' then
                          if cee = '1' and busy_ocup = '0' then
                                        bufer := valor;
                                        tam_aux := tam_pac_tx;
                                        busy_ocup := '1';
                                        cont := 0;
                                        busy <= '1';
```

```
                        end if;

                        if busy_ocup = '1' then
                                        if cont = 20834 then
                                          cont := 0;

                                         if stop = tam_aux then
                                                        saida <= '1';
                                                        busy <= '0';
                                                        bytes_saida := 0;
                                                        stop := 0;
                                                        busy_ocup := '0';
                count_vector := (others=>'0');

                                                elsif bytes_saida = 9 then
                                                        saida <= '1';     --stop
                                                        bytes_saida := 0;
                                                        busy_ocup := '1';

                                                elsif bytes_saida = 0 then
                                                        saida   <= '0';     --start
                                                        bytes_saida := 1;
                                                else
                                                        saida <= bufer(stop);
                                                        bytes_saida := bytes_saida + 1;
                                                        stop := stop + 1;
                                                        count_vector := count_vector + 1;
                                                end if;

                                        else
                                                cont := cont + 1;
                                        end if;
                        end if;
                        end if;
        end if;
end process;
end Behavioral;
```

Item B – Modulo Receptor

```
entity RECEPTOR_RX is
        Port ( clock : in  STD_LOGIC;
                rst : in  STD_LOGIC;
                pino : in  STD_LOGIC;
                dataout : out  STD_LOGIC_VECTOR(127 downto 0);
                ready : out  STD_LOGIC;
                tam_pac_janela : out integer range 0 to 160;
                rx : in  STD_LOGIC
       );
end RECEPTOR _RX;

architecture Behavioral of RECEPTOR _RX is

begin
process(rst,clock)
variable to_read: std_logic := '0';
variable buf_rx : std_logic_vector(127 downto 0);
variable very : std_logic_vector(15 downto 0);
variable aux_tam : std_logic_vector(15 downto 0) := (others=>'1');
variable tam_package : std_logic_vector(15 downto 0);
variable bit_rx : integer range 0 to 9 := 0;
variable count_bit : integer range 0 to 160 := 0;
variable cont : integer range 0 to 20834 := 0; -- 4800 bps
begin
    if rst = '1' then
            ready <= '0';
            dataout <= (others=>'0');
            tam_package := (others=>'0');
    else
    if clock'event and clock = '1' then
          if pino = '0' then
                                          bit_rx := 0;
                                          ready <= '0';
                                          to_read := '0';
                                          count_bit := 0;
                                          tam_package := (others=>'0');
                                          cont := 0;
         else
          if pino = '1' then
                if to_read = '0' and rx ='0' then --bit start
                        to_read := '1';
                        cont := 10417;
                        bit_rx := 0;
                end if;
```

```
                if to_read = '1' then
                        ready <= '0';
                                if tam_package = aux_tam then
                                if rx = '1' then
                                count_bit := count_bit - 1;
                                ready <= '1';
                                tam_pac_janela <= count_bit;
                                dataout(count_bit downto 0) <= buf_rx(count_bit downto 0);
                                to_read := '0';
                                count_bit := 0;
                                tam_package := (others=>'0');
                                else
                                        to_read := '0';
                                end if;

                                elsif bit_rx = 0 then
                                        if rx = '0' then --bit start
                                                bit_rx := 1;
                                        else
                                                bit_rx  := 0;
                                        to_read := '0';
                                end if;
                        elsif bit_rx = 9 then
                          bit_rx  := 0;
                                if rx = '1' then --bit stop
                                        to_read := '0';
                                 else
                                        to_read := '0';
                                end if;
                        else
                                buf_rx(count_bit) := rx;
                                count_bit := count_bit + 1;
                                bit_rx := bit_rx + 1;
                                        tam_package := tam_package + 1;
                                 if count_bit = 16 then
                                        aux_tam := buf_rx(15 downto 0);
                                 end if;
                        end if;
                end if;
                end if;
          end if;
         end if;
     end if;
     end process;
end Behavioral;
```

Item C – Modulo Leitor do Sensor

```
entity LEITOR_SENSOR is
        Port ( clock : in  STD_LOGIC;
                rst : in  STD_LOGIC;
                pino : in  STD_LOGIC;
                dataout : out  STD_LOGIC_VECTOR(127 downto 0);
                ready : out  STD_LOGIC;
                tam_pac_sensor : out integer range 0 to 160;
                rx : in  STD_LOGIC);
        end LEITOR_SENSOR;

architecture Behavioral of LEITOR_SENSOR is

begin
process(rst,clock)
variable to_read: std_logic := '0';
variable buf_rx : std_logic_vector(127 downto 0);
variable very : std_logic_vector(15 downto 0) := "0000000000100000";
variable bit_rx : integer range 0 to 9 := 0;
variable count_vector : integer range 32 to 160 := 32;
variable count_package : integer range 32 to 160 := 32;
variable cont : integer range 0 to 20834 := 0; -- 4800 bps
begin
     if rst = '1' then
          ready <= '0';
          dataout <= (others=>'0');
          very := (others=>'0');
     else
          if clock'event and clock = '1' then
           if pino = '0' then
                                to_read := '0';
                                        count_package := 32;
                                        count_vector := 32;
                                        very := "0000000000100000";
                                        bit_rx := 0;
           else
            if pino = '1' then
                    if to_read = '0' and rx ='0' then -- bit start
                            to_read := '1';
                            cont := 10417;
                            bit_rx := 0;
                    end if;

                    if to_read = '1' then
                            ready <= '0';
```

```
                    cont := 0;
            if very = "0000000000110000" then
                if rx = '1' then
                        buf_rx(15 downto 0) := very;
                        buf_rx(31 downto 16) := "0011001000110000";
                        count_vector := count_vector - 1;
                        ready <= '1';
                        dataout(count_vector downto 0) <= buf_rx(count_vectordownto 0);
                        tam_pac_sensor <= count_vector;
                        to_read := '0';
                        count_package := 32;
                        count_vector := 32;
                        very := "0000000000100000";
                    else
                                to_read := '0';
                    end if;
                    elsif bit_rx = 0 then
                                if rx = '0' then --bit start
                                        bit_rx := 1;
                                else
                                        bit_rx  := 0;
                                        to_read := '0';
                    end if;
                    elsif bit_rx = 9 then
                            bit_rx  := 0;
                                if rx = '1' then --bit stop
                                        to_read := '0';
                                else
                                        to_read := '0';
                                end if;
                    else
                                buf_rx(count_package) := rx;
                                count_vector := count_vector + 1;
                                bit_rx := bit_rx + 1;
                                very := very + 1;
                                count_package := count_package + 1;
                    end if;
                end if;
             end if;
            end if;
           end if;
      end if;
      end process;
end Behavioral;
```

Item D – Modulo Processamento de Dados

```
entity PROCESSAMENTO DE DADOS is
        port(clock : in STD_LOGIC;
                    per : out STD_LOGIC;
                    busy : in STD_LOGIC;
                    button : in STD_LOGIC;
                    teste_TX: out STD_LOGIC;
                    redy : in STD_LOGIC;
                    entrada : in STD_LOGIC_vECTOR(127 downto 0);
                    entrada_pac_sensor : in
                    STD_LOGIC_vECTOR(127 downto 0);
                    sinal_pacote_sensor : in STD_LOGIC;
                    tam_pac_janela : in integer range 0 to 160;
                    tam_pac_sensor : in integer range 0 to 160;
                    tam_pac_tx : out integer range 0 to 160;
                   dataout : out STD_LOGIC_vECTOR(127 downto 0));
end PROCESSAMENTO DE DADOS;

architecture Behavioral of PROCESSAMENTO DE DADOS is
signal var :STD_LOGIC_VECTor(127 downto 0);
signal var_aux :STD_LOGIC_VECTor(31 downto 0);
signal count :STD_LOGIC_VECTor(15 downto 0);
signal teste :STD_LOGIC_VECTor(1 downto 0) := "00";
signal teste_redy :STD_LOGIC_VECTor(1 downto 0) := "00";
signal teste1 :STD_LOGIC_VECTor(1 downto 0) := "00";
signal teste_sensor :STD_LOGIC_VECTor(1 downto 0) := "00";
signal teste_janela :STD_LOGIC_VECTor(1 downto 0) := "00";
signal ESTADO : STD_LOGIC_VECTor(2 downto 0) := "000";
constant INICIO : STD_LOGIC_VECTor(2 downto 0) := "000";
constant PACOTE_SENSOR : STD_LOGIC_VECTor(2 downto 0) := "001";
constant INTERMEDIARIO : STD_LOGIC_VECTor(2 downto 0) := "010";
constant PACOTE_JANELA : STD_LOGIC_VECTor(2 downto 0) := "011";
constant FINAL : STD_LOGIC_VECTor(2 downto 0) := "100";
constant OUTROS : STD_LOGIC_VECTor(2 downto 0) := "101";
begin
process(clock)
variable cont : integer range 0 to 2105 := 0;
variable cont1 : integer range 0 to 150000 := 0;
variable d : integer range 0 to 10 := 0;
variable tam_aux2 : integer range 0 to 160 := 0;
variable tam_aux : integer range 0 to 160 := 0;
variable soma : integer range 0 to 160 := 0;
variable ajuste : integer range 0 to 160 := 0;
variable var_aux : integer range 0 to 160 := 0;
begin
```

```
if clock'event and clock = '1' then
 cont := 0;
 case ESTADO is

        when INICIO =>
                        per <= '0'; -- desativar pacote
                                        if teste = "01" and teste1 = "01" then
                                        -- aguardar pacote de reconhecimento
                                          if count < "1110101001100000" then
                                            count <= count + 1;
                                          if redy = '1' and teste_redy = "01" then
                                            teste_TX <= '1';
                                            var <= entrada;
                                            teste_redy <= "00";
                                                count <= (others=>'0');
                                                ESTADO <= OUTROS;
                                              elsif redy = '0' then
                                                 teste_redy <= "01";
                                                 ESTADO <= INICIO;
                                          end if;
                                          else
                                            ESTADO <= PACOTE_SENSOR;
                                                teste <= "00";
                                            teste1 <= "00";
                                                count <= (others=>'0');
                                          end if;
                                        else
                                    ESTADO <= PACOTE_SENSOR;
                                end if;

      when PACOTE_SENSOR =>
                        if teste = "01" and teste1 = "01" then
                          ESTADO <= PACOTE_JANELA;
      elsif sinal_pacote_sensor = '1' and teste_sensor = "01" then
                          tam_aux := tam_pac_sensor;
                          var(47 downto 15) <= entrada_pac_sensor(47 downto 15);
                          ESTADO <= PACOTE_SENSOR;
                          teste_sensor <= "00";
                          teste <= "01";
                        elsif sinal_pacote_sensor = '0' then
                        teste_sensor <= "01";
                        ESTADO <= PACOTE_SENSOR;
      elsif redy = '1' and teste_janela = "01" then --se dados prontos
                          tam_aux2 := tam_pac_janela;
                          soma := tam_aux + tam_aux2 + 1;
                          teste_janela <= "00";
```

```
                    var(soma downto 48) <= entrada(tam_aux2 downto 0); -- passar dados
recebidos
                        teste1 <= "01";

                        elsif redy = '0' then
                            teste_janela <= "01";
                            ESTADO <= PACOTE_SENSOR;
                        else
                             ESTADO <= PACOTE_SENSOR;
                        end if;

                when PACOTE_JANELA =>
                  var(15 downto 0) <= entrada_pac_sensor(15 downto 0) + entrada(15 downto 0);
                  ESTADO <= INTERMEDIARIO;

                when INTERMEDIARIO =>
                                tam_pac_tx <= soma + 1;
                                ESTADO <= FINAL;

                when OUTROS =>
                                                    tam_pac_tx <= 32;
                                                    teste <= "00";
                                                    teste1 <= "00";
                                                    ESTADO <= FINAL;

                when FINAL =>
                        per <= '1'; --avisar pacote pronto
                        dataout <= var; --passar dados p/ saida
                        teste_TX <= '0';
                        ESTADO <= INICIO;
                        soma := 0;
                        ajuste := 0;
                        tam_aux := 0;
                        tam_aux2 := 0;

            when others => null;
    end case;
    end if;
end process;
end Behavioral;
```

Item E – Modulo Sincronizador

```
entity SINCRONIZADOR is
        port(clock : in STD_LOGIC;
                    leave : out STD_LOGIC;
                    leave_001 : out STD_LOGIC;
                    teste_TX: in STD_LOGIC;
                    control : in STD_LOGIC;
                    control_ready : in STD_LOGIC
                );
end SINCRONIZADOR;

architecture Behavioral of SINCRONIZADOR is
signal cont: STD_LOGIC_VECTOR(15 downto 0);
signal cont3: STD_LOGIC_VECTOR(28 downto 0);
signal cont1: STD_LOGIC_VECTOR(16 downto 0);
begin
process(clock)
variable cont2 : integer range 0 to 10418 := 0;
variable count_janela : integer range 0 to 200000010 := 0;
variable estado : integer range 0 to 3 := 0;
variable estado_tempo : integer range 0 to 5 := 0;
variable var : std_logic := '0';
begin

  if clock'event and clock = '1' then

        case estado is

            when 0 =>
            if teste_TX = '1' and var = '0' then --esperando transmissor trabalhar pela primeira vez p/ sincronizar pacote
                estado := 1; -- proximo estado
                    var := '1'; --avisa que ja enviou e recebeu
                    cont <= cont + 1;
                else
                estado := 0;
                    leave <= '1';
                    leave_001 <= '1';
                    cont <= "0100111000100000"; --atribuir valor mais ou menos calculado do tempo de recebimento e transmissão do pacote
                end if;

            when 1 =>
        if cont <= "1110101001100000" and var = '0' then  -- executa o bloco com no maximo 3 segundos
```

```
                cont <= cont + 1;
                        case estado_tempo is
                  when 0 =>

                                if teste_TX = '1' then
                                  estado_tempo := 1;
                                elsif count_janela <= 200000000 then
                                   leave_001 <= '1';
                                          estado_tempo := 0;
                                          leave <= '1';
                                 elsif count_janela > 200000000 then
                                   leave_001 <= '0';
                                   estado_tempo := 1;
                                   leave <= '0';
                                   count_janela := 0;
                                 else
                                   count_janela := count_janela + 1;
                                 end if;
                           when 1 =>
                                   var := '1';
                                   leave <= '0';
                           when others => null;
        end case;

                elsif cont1 = "11000011010100000" then
                cont <=(others=>'0');
                cont1 <=(others=>'0');
                var := '0';
                estado := 1;
                estado_tempo := 0;

                elsif cont > "1110101001100000" then -- dormir 10 segundos após os 3
                 cont1 <= cont1 + 1;
                 leave <= '0';
                 leave_001 <= '0';
                else
                  cont <= cont + 1;
                  leave <= '0';
      leave_001 <= '0';
     end if;

           when others => null;
    end case;
          end if;
end process;
end Behavioral;
```

Item F – Modulo Divisor

```
entity Divisor is
    Port ( clk_100MHz : in  STD_LOGIC;
                        clk_1Hz : out  STD_LOGIC;
                        clk_ManHz : out  STD_LOGIC);
end Divisor;

architecture Behavioral of Divisor is
signal cont: integer range 0 to 100000001;
signal cont_man: integer range 0 to 100000001;
signal cont_10417: integer range 0 to 100000001;
begin
process (clk_100MHz)
begin
        if clk_100Mhz'event and clk_100MHz='1' then
                cont <= cont + 1;
                        cont_man <= cont_man + 1;
                        cont_10417 <= cont_10417 + 1;
                  if cont = 20834 then
                        clk_1Hz <= '1';
                  else
                        clk_1Hz <= '0';
                  end if;
                  if cont > 20834 then
                       cont <= 0;
                  end if;
                  if cont_man = 1000 then
                        clk_ManHz <= '1';
                  else
                        clk_ManHz <= '0';
                  end if;
                  if cont_man > 1000 then
                       cont_man <= 0;
                  end if;
        end if;
    end process;
end Behavioral;
```

Item G – Modulo Top

```
entity TOP is
                                        port ( clock : in STD_LOGIC;
                                                tx : out STD_LOGIC;
                                                rx : in STD_LOGIC;
                                                rx_sensor : in STD_LOGIC;
                                                tx_sensor : out STD_LOGIC;
                                                rx_exb : in STD_LOGIC;
                                                tx_exb : out STD_LOGIC;
                                                gps_enable : out STD_LOGIC
                                            );
end TOP;
architecture Behavioral of TOP is

signal dataout : STD_LOGIC_VECTOR(127 downto 0);
signal dataout2 : STD_LOGIC_VECTOR(127 downto 0);
signal busy : STD_LOGIC;
signal teste_OK : STD_LOGIC;
signal clk_1Hz : STD_LOGIC;
signal cont_trans : STD_LOGIC;
signal clk_ManHz : STD_LOGIC;
signal clk_10417Hz :  STD_LOGIC;
signal teste_TX : STD_LOGIC;
signal dataout3 : STD_LOGIC_VECTOR(127 downto 0);
signal dataout4 : STD_LOGIC_VECTOR(127 downto 0);
signal busyTX2 : STD_LOGIC;
signal tam_pac_tx : integer range 0 to 160;
signal tam_pac_janela : integer range 0 to 160;
signal tam_pac_sensor : integer range 0 to 160;
signal ce : STD_LOGIC;
signal ceTX2 : STD_LOGIC;
signal rst : STD_LOGIC;
signal ready : STD_LOGIC;
signal ready_1 : STD_LOGIC;
signal tx_DS : STD_LOGIC;
signal tx_DR : STD_LOGIC;
signal per : STD_LOGIC;
signal desliga : STD_LOGIC;
signal desligar_pacote_receiver : STD_LOGIC;
signal desativar_pacote : STD_LOGIC;
signal desativar_SerialRX : STD_LOGIC;
signal abilitar_desabilitarPR : STD_LOGIC;

component Divisor is
        Port ( clk_100MHz : in  STD_LOGIC;
```

```
                clk_1Hz : out  STD_LOGIC;
                clk_ManHz : out  STD_LOGIC
        );
end component;

component Leitor_Sensor is
            Port( clock : in  STD_LOGIC;
                  rst : in  STD_LOGIC;
                  pino : in  STD_LOGIC;
                  dataout : out  STD_LOGIC_VECTOR(127 downto 0);
                  ready : out  STD_LOGIC;
                  tam_pac_sensor : out integer range 0 to 160;
                  rx : in  STD_LOGIC
                );
end component;


component Sincronizador is
                                    port( clock : in STD_LOGIC;
                                          leave : out STD_LOGIC;
                                          leave_001 : out STD_LOGIC;
                                          teste_TX: in STD_LOGIC;
                                          control : in STD_LOGIC;
                                          control_ready : in STD_LOGIC

            );
end component;

component PROCESSAMENTO_DADOS is
            port( clock : in STD_LOGIC;
                                 button : in STD_LOGIC;
                                teste_TX: out STD_LOGIC;
                                entrada_pac_sensor : in STD_LOGIC_vECTOR(127 downto 0);
                                redy : in STD_LOGIC;
                                per : out STD_LOGIC;
                                busy : in STD_LOGIC;
                                entrada : in STD_LOGIC_vECTOR(127 downto 0);
                                sinal_pacote_sensor : in STD_LOGIC;
                                tam_pac_janela : in integer range 0 to 160;
                                tam_pac_sensor : in integer range 0 to 160;
                                tam_pac_tx : out integer range 0 to 160;
                                dataout : out STD_LOGIC_vECTOR(127 downto 0)
                        );
end component;

component TRANSMISSOR_TX is
```

```
                                        port( clock : in STD_LOGIC;
                                              valor : in STD_LOGIC_VECTOR(127 downto 0);
                                              saida : out STD_LOGIC;
                                              rst : in STD_LOGIC;
                                              cee : in STD_LOGIC;
                                              cont_trans : in STD_LOGIC;
                                              tam_pac_tx : in integer range 0 to 160;
                                              busy : out STD_LOGIC
                                        );
end component;

component RECEPTOR_RX is
            Port( clock : in  STD_LOGIC;
                  rst : in  STD_LOGIC;
                  dataout : out  STD_LOGIC_VECTOR(127 downto 0);
                  ready : out  STD_LOGIC;
                  pino : in  STD_LOGIC;
                  tam_pac_janela : out integer range 0 to 160;
                  rx : in  STD_LOGIC
                );
end component;


begin

COMP_DIVISOR : Divisor port map( clk_100MHz => clock,
                  clk_1Hz => clk_1Hz,
                  clk_ManHz => clk_ManHz
                );

COMP_LEITOR_SENSOR: leitor_sensor Port map( clock => clk_1Hz,
                        rst => rst,
                        dataout => dataout,
                        ready => ready,
                        pino => desativar_SerialRX,
                        tam_pac_sensor => tam_pac_sensor,
                        rx => rx_sensor
                      );

COMP_SINCRONIZADOR: SINCRONIZADOR  port map ( clock => clk_ManHz,
                                              leave => abilitar_desabilitarPR,

                                              leave_001 => desliga,

                                              teste_TX => teste_TX,
                                              control => ready_1,
```

```
                                                    control_ready => busyTX2
                                                    );

COMP_PROCESSAMENTO_DADOS: PROCESSAMENTO_DADOS port map (
                                                    clock => clk_ManHz,
                                                    redy => ready_1,

                                                    teste_TX => teste_TX,
                                                    per => desligar_pacote_receiver,
                                                    busy => busyTX2,
                                                    dataout => dataout3,
                                                    entrada => dataout4,

                                                    entrada_pac_sensor => dataout,

                                                    sinal_pacote_sensor => ready,

                                                    tam_pac_janela => tam_pac_janela,
                                                    tam_pac_sensor => tam_pac_sensor,
                                                    tam_pac_tx => tam_pac_tx,
                                                    button => abilitar_desabilitarPR
                                                                );

COMP_TRANSMISSOR: TRANSMISSOR_TX port map( clock => clock,
                                                                valor => dataout3,
                                                                 saida => tx_DR,
                                                                 rst => rst,
                                                                 cee => ceTX2,

                                                                 cont_trans => cont_trans,

                                                                 tam_pac_tx => tam_pac_tx,
                                                                 busy => busyTX2
                                                                );

COMP_RECEPTOR: RECEPTOR_RX Port map( clock => clk_1Hz,
                                rst => rst,
                                dataout => dataout4,
                                ready => ready_1,
                                pino => desliga,
                               tam_pac_janela => tam_pac_janela,
                                rx => rx_exb
                        );


process(busyTX2,per)
```

```
begin
  if busyTX2 = '1' then
                tx <= tx_DR;
                tx_exb <= tx_DR;
  end if;
end process;

rst <= '0';
gps_enable <= '1';
tx_sensor <= '1';

process(desligar_pacote_receiver, busy)
variable ativa : STD_LOGIC := '1';
begin
        if busyTX2 = '1' then
                ativa := '0';
        elsif busyTX2 <= '0' then
          cont_trans <= '0';
        elsif desligar_pacote_receiver = '0' then
                ceTX2 <= '0';
                ativa := '1';
        elsif desligar_pacote_receiver = '1' then
                cont_trans <= '1';
                ceTX2 <= ativa;
        end if;
end process;
end Behavioral;
```