INSTITUTO FEDERAL DE SANTA CATARINA

RENAN RODOLFO DA SILVA

Integração de ramais analógicos com FPGA utilizando processador *softcore*

São José - SC

dezembro/2022

INTEGRAÇÃO DE RAMAIS ANALÓGICOS COM FPGA UTILIZANDO PROCESSADOR *SOFTCORE*

Monografia apresentada ao Curso de Engenharia de Telecomunicações do campus São José do Instituto Federal de Santa Catarina para a obtenção do diploma de Engenheiro de Telecomunicações.

Orientador: Prof. Roberto de Matos, Dr.

São José - SC

dezembro/2022

RENAN RODOLFO DA SILVA

INTEGRAÇÃO DE RAMAIS ANALÓGICOS COM FPGA UTILIZANDO PROCESSADOR *SOFTCORE*

Este trabalho foi julgado adequado para obtenção do título de Engenheiro de Telecomunicações, pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, e aprovado na sua forma final pela comissão avaliadora abaixo indicada.

São José - SC, 16 de dezembro de 2022:

Prof. Roberto de Matos, Dr.
Orientador
Instituto Federal de Santa Catarina

Prof. Fábio Alexandre de Souza, Dr.
Instituto Federal de Santa Catarina

Prof. Marcos Moecke, Dr.
Instituto Federal de Santa Catarina

A minha esposa e minha filha

AGRADECIMENTOS

Agradeço primeiramente a minha família, que sempre me apoiaram e foram compreensivos.

A minha esposa Naiara e minha filha Lara, que ao longo de toda a graduação, foram pacientes e parceiras.

Agradeço aos meus colegas e amigos, que de alguma forma fizeram parte dessa jornada.

Por fim, agradeço ao corpo docente do IFSC, em especial ao meu orientador, Roberto, pelos ensinamentos, disposição e comprometimento.

Esquecer é como uma ferida. A ferida pode cicatrizar, mas já deixou uma cicatriz.
Monkey D. Luffy, One Piece

RESUMO

A capacidade atual dos componentes lógicos programáveis, mais especificamente as *Field-Programmable Gate Array* (FPGA), vem possibilitando a criação de plataformas sob medida para uma determinada aplicação de forma rápida e otimizada. Isso permite que o *software* e *hardware* sejam desenvolvidos juntos e evoluam inclusive depois do produto lançado. Para permitir essas experimentações no campo de desenvolvimento de produtos de telecomunicações, este trabalho propõe uma infraestrutura de integração de um *kit* de desenvolvimento de FPGA com uma placa de ramal, responsável por conectar telefones analógicos em centrais telefônicas comerciais, para criar uma plataforma mínima para experimentações envolvendo canais digitais de áudio e processadores sintetizados. Para a proposta, foi utilizado a placa de ramal da central telefônica Impacta 16, modelo 4990083, e o *kit* de desenvolvimento FPGA, modelo DE2-115, utilizando o processador *softcore* Nios II. A metodologia utilizada permitiu que fossem levantadas as informações necessárias para validar a solução proposta. Um conjunto de testes foram realizados para validar a integração e as implementações realizadas. Os resultados evidenciaram que toda a infraestrutura de *hardware* e *software* para realizar a integração do ramal analógico com FPGA foram executados.

Palavras-chave: FPGA. Placa de ramal. *Softcore*.

ABSTRACT

The current capacity of programmable logic components, more specifically the Field-Programmable Gate Array (FPGA), has enabled the creation of tailored systems for a given application in a fast and optimized way, which allows software and hardware to be developed together and evolve even after the product is released. In this way, this work proposes an implementation of a minimal platform for experiments with digital audio channels and softcore processors to allow the development of telecommunications-related products. The proposed platform is composed of an FPGA development kit and a telephone exchange extension card, which is a board that connects an analog branch line to a PABX. In particular, this work uses the FPGA development kit, model DE2-115, along with the Nios II softcore processor, and the Intelbras Impacta 16 board, model 4990083. The experimental results show that the developed hardware and software allowed the integration and communication between boards.

Keywords: FPGA. Expansion card. Softcore.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

A inovação guia as empresas na busca de novos produtos e mercados, principalmente nos setores de tecnologia como telecomunicações. As ferramentas e os métodos de desenvolvimento na área de *hardware* e *software* embarcado vem, há algum tempo se adequando para acompanhar esse ritmo. Antes, a criação de uma plataforma de *hardware* sob medida com processador, memória e periféricos específicos, para uma determinada aplicação, poderia levar meses para ser finalizada. Além disso, o processo é naturalmente propenso a erros, aumentando ainda mais o tempo para que um produto chegasse ao mercado depois da sua concepção, o chamado *time to market*.

Apesar da tecnologia de componentes lógicos programáveis não ser nova, o aumento da densidade lógica desses componentes, o aumento de blocos fixos especializados e a evolução das ferramentas têm sido determinantes para a aceleração do desenvolvimento do *hardware* e *software* embarcado. Atualmente, é possível criar uma plataforma de *hardware* em curto tempo com a vantagem de poder atualizá-la durante toda a vida útil do produto. Sendo uma característica muito interessante para testar uma ideia inovadora com um produto viável mínimo, visando a evolução do *hardware* mesmo depois do lançamento.

Dentre os componentes lógicos programáveis, a classe que se destaca pela quantidade de elementos lógicos é a FPGA. Segundo González et al. (2012), uma FPGA contém milhões de conexões e células lógicas que podem ser configuradas de forma a criar qualquer sistema digital síncrono. Dessa forma, essa classe de componente possibilita a criação de sistemas computacionais genéricos, incluindo *Central Processing Unit* (CPU), memória, entradas e saídas, além de sistemas específicos e otimizados para uma determinada aplicação.

Em sistemas computacionais genéricos sintetizados em FPGA, o núcleo principal, ou seja, a CPU, é chamado de *softcore*. Em geral, esses componentes são descritos em Linguagem de Descrição de *Hardware* (HDL) e, segundo Tong e Anderson (2006), possuem um alto nível de abstração e maior flexibilidade para customizações, permitindo adicionar novas instruções ou mesmo mudar pequenos detalhes da arquitetura da CPU. Além disso, alguns *softcores*, principalmente os de código aberto, buscam ser independentes de plataforma, podendo ser sintetizados em componentes de diversos fabricantes. A desvantagem é que o desempenho dos *softcores* depende da tecnologia da FPGA utilizada. Para lidar com esse problema, processadores tradicionais *hardcore* vêm sendo integrados na mesma pastilha da FPGA, com a desvantagem da perda de flexibilidade interna da CPU. Independente do uso de um *softcore* ou *hardcore*, a grande vantagem de utilizar uma FPGA é permitir conectar periféricos, co-processadores e blocos customizados à CPU.

Considerando as vantagens do uso da FPGA e de *softcores*, este trabalho tem por

objetivo desenvolver a infraestrutura de *hardware*, *software* e lógica programável para integrar uma placa de ramal, utilizada em centrais telefônicas comerciais como interface analógica para telefones, com um *kit* de desenvolvimento de FPGAs.

1.1 Objetivo geral

O objetivo deste trabalho é integrar uma placa de ramal da central telefônica Impacta 16, modelo 4990083, com um *kit* de desenvolvimento FPGA, modelo DE2-115, para criar uma plataforma mínima para experimentações envolvendo canais digitais de áudio e processadores sintetizados (*softcores*).

1.1.1 Objetivos específicos

Foram definidos alguns objetivos específicos a partir do objetivo geral:

1. Integrar fisicamente a placa de ramal com o *kit* FPGA, projetando e confeccionando uma placa adaptadora;
2. Projetar e implementar blocos em VHDL independente de plataforma para a interface de controle e dados da placa de ramal;
3. Configurar a plataforma com *softcore* para gerenciar o sistema, integrando os blocos em VHDL criados;
4. Implementar o *software* para controle da placa de ramal;
5. Experimentar com os canais de áudio digital gerados pela placa de ramal.

1.2 Organização do texto

O restante do documento está divido da seguinte forma: no Capítulo 2 são abordados os principais fundamentos teóricos julgados necessários para o desenvolvimento do projeto. No Capítulo 3 é apresentado o desenvolvimento do projeto. Os testes e resultados são apresentados no Capítulo 4. Finalmente, no Capítulo 5, as conclusões e trabalhos futuros são apresentados.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os tópicos considerados relevantes para o desenvolvimento deste trabalho. Aborda a origem dos dispositivos lógicos programáveis e algumas de suas evoluções. Relata a definição de *hardware*, *software co-design* e o conceito de *softcore* e seus fluxos de projeto. Discute, também, o conceito de *codec*, codificação digital e de placa de ramal.

2.1 Dispositivos lógicos programáveis

Programmable Logic Device (PLD) é um Circuito Integrado (CI) que fornece liberdade ao usuário para configurá-lo, fazendo com que seja possível implementar um circuito lógico ou fazer alguma alteração no projeto caso seja necessário. Segundo Costa (2009), em comparação com algumas outras tecnologias de circuitos integrados digitais, os PLDs apresentam um ciclo de projeto menor, custos reduzidos e possibilitam a implementação da lógica programável.

Por exemplo, em relação aos circuitos discretos, os quais, em geral, possuem apenas um tipo de porta lógica e exigem várias conexões físicas para construção do circuito, os PLDs permitem aos projetistas inserir uma grande quantidade de funções em um mesmo CI. Devido às vantagens do uso dos PLDs e os investimentos na área, várias evoluções ocorreram nos últimos anos. Segundo Cofer e Harding (2006), os PLDs são divididos em três grupos principais:

- *Simple Programmable Logic Devices* (SPLD);
- *Complex Programmable Logic Device* (CPLD);
- *Field-Programmable Gate Array* (FPGA).

2.1.1 SPLD

Os SPLDs surgiram em torno de 1970, e englobam dois principais tipos de dispositivos lógicos programáveis, sendo o *Programmable Logic Array* (PLA) e o *Programmable Array Logic* (PAL). O primeiro consiste em dois níveis de portas lógicas: um plano de portas *AND* seguido por um plano de portas *OR*, ambos programáveis (COSTA, 2009). Cada saída do plano *AND* de um PLA pode corresponder a qualquer produto das entradas. Da mesma forma, cada saída do plano *OR* pode ser configurada para produzir a soma dos produtos de qualquer saída do plano *AND* (COSTA, 2009). Essas características tornam

esse tipo de SPLD muito versátil em razão da alta capacidade de entradas permitidas, mas, em contrapartida, possui um custo elevado e baixa velocidade.

Devido à dificuldade em se produzir um dispositivo PLA em consequência dos dois níveis lógicos programáveis, surgiram os dispositivos PAL, em que se tem apenas a porta *AND* programável, fazendo com que o custo seja menor e alcance uma maior velocidade. A versatilidade dos dispositivos PAL é significativamente melhorada pelo uso de macrocélulas, onde o sinal de saída gerado pode ser combinacional ou registrado, dependendo da lógica utilizada (TINDER, 2000).

As arquiteturas PAL e PLA usam *Complementary Metal Oxide Semiconducto* (CMOS) e podem ser projetadas com o uso de *Erasable Programmable Read-Only Memory* (EPROM) ou com *Electrically Erasable Programmable Read-Only Memory* (EEPROM), permitindo assim, que sejam reprogramáveis.

2.1.2 CPLD

Os CPLDs eram compostos basicamente por vários SPLDs integrados, chegando a centenas de portas lógicas (KARRIS, 2007), mas, atualmente, podem chegar a dezena de milhares de portas lógicas, devido à evolução das arquiteturas. A complexidade da produção é proporcional à quantidade de portas lógicas utilizadas no CI. Na estrutura do CPLD existem diversos blocos lógicos, definidos pela Altera como *Logic Array Block* (LAB), que possuem interconexões programáveis geralmente denominadas *Programmable Interconnect Array* (PIA) (FLOYD, 2009). Cada um dos blocos equivale a uma SPLD, possui entradas e saídas que podem ser conectadas a outros blocos lógicos usando PIA ou nos pinos do componente para interface com os sistemas externos de Entrada e Saída (I/O).

As macrocélulas, segundo (FLOYD, 2009), consistem geralmente em uma porta *OR* e alguma lógica de saída associada, pode ser configurada para lógica combinacional, lógica registrada (com memorização) ou uma combinação de ambos que permite criar lógicas sequenciais das mais diversas. Elas estão presentes na arquitetura CPLD e variam com o número de blocos lógicos (ou blocos funcionais) sendo responsáveis pela lógica combinacional ou sequencial no *chip* (WEBER et al., 2016). Essas macrocélulas também possibilitam uma flexibilidade adicional, onde cada arquitetura pode ter a sua saída ajustada de uma forma específica, por exemplo, em termos de produto. Desta forma, um CPLD pode ser aplicado como uma máquina de estado ou decodificador de sinais, substituindo centenas de circuitos discretos que implementariam a mesma função (COSTA, 2009).

CPLDs podem ser projetadas de uma forma variada dependendo da sua finalidade. É possível variar a sua densidade, tecnologia de processo, consumo de potência, tensão de alimentação e velocidade (FLOYD, 2009). Por esse motivo, é necessário fazer uma análise

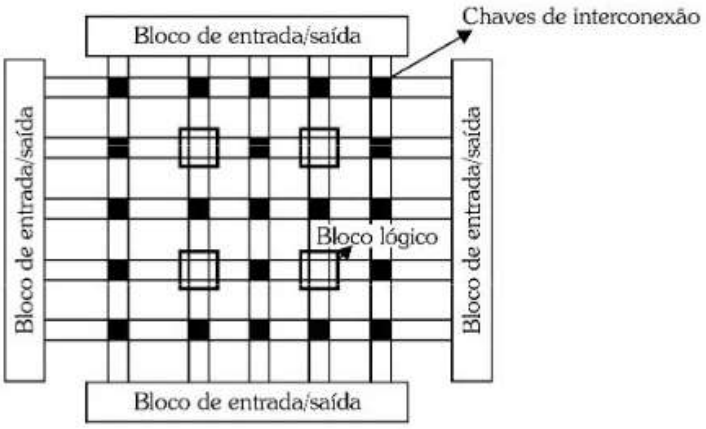
intensa durante o projeto para verificar se o dispositivo alvo conseguirá suportar uma expansão, caso necessário.

2.1.3 FPGA

Segundo Wilson (2015), ao invés de uma matriz fixa de portas, a FPGA usa o conceito de LAB, podendo além de criar rotas de comunicação nos dispositivos, configurar cada bloco lógico otimamente. As LABs possuem *LookUp Table* (LUT) que servem como uma memória programável volátil sendo utilizadas para implementar qualquer função booleana com N entradas (COFER; HARDING, 2006).

Diferente dos CPLDs que possuem suas configurações armazenadas em memórias não voláteis, as FPGAs geralmente utilizam memórias voláteis, dessa forma, a FPGA necessita de uma memória externa não volátil (ex.: *Flash*) para armazenar a programação do componente. Uma FPGA pode se basear em Tecnologia Programável uma Vez (OTP) ou em Memória de Acesso Aleatório (SRAM). Para aplicações de prototipagem rápida, a configuração recomendada deve ser do tipo reprogramável em vez de OTP pelo fato de possibilitar alterações na lógica caso seja necessário, embora OTP possa ter aplicações significativas em produtos estáveis e bem testados (COFER; HARDING, 2006), por exemplo, em aplicações espaciais.

Figura 1 – Estrutura simplificada de uma FPGA



Fonte: (COSTA, 2009)

Como mencionado, a FPGA é constituída de muitas LABs e também por blocos de I/O e chaves de interconexões. A Figura 1 ilustra essa estrutura de forma simplificada. Os blocos lógicos formam uma matriz bidimensional, e as chaves de interconexão são organizadas como canais de roteamento horizontal e vertical entre as linhas e colunas dos blocos lógicos (COSTA, 2009). Isto permite aos projetistas conectar as LABs conforme a necessidade do projeto.

Dessa forma, a escolha da melhor arquitetura a ser usada em um projeto, depende muito da aplicação a ser desenvolvida e também do objetivo que se deseja alcançar. Em alguns casos, a velocidade pode ser prioridade, em outros, o número de elementos necessários ou a potência consumida. Também existem projetos em que o foco é a quantidade de pinos de conexão externa.

2.1.4 Fluxo de projeto de *hardware* digital

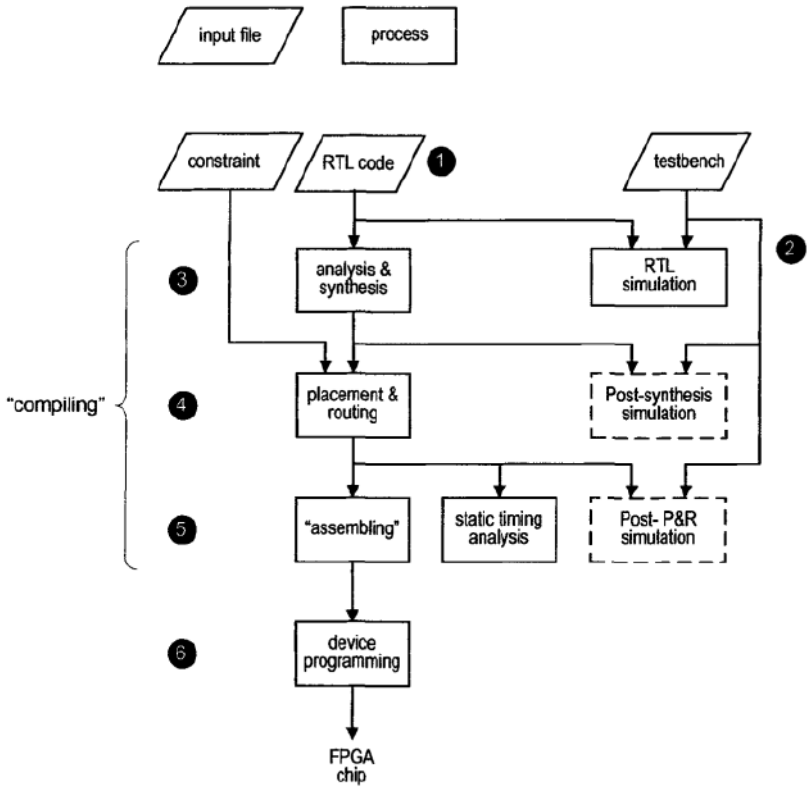
O fluxo de desenvolvimento de *hardware* digital proposto por Chu (2011), consiste nas etapas que podem ser vistas na Figura 2. Inicialmente, é necessário estabelecer qual a finalidade do projeto, fazer o levantamento das restrições, por exemplo, consumo e desempenho. O primeiro passo do desenvolvimento (1) é descrever o circuito que se deseja implementar, que pode ser feito utilizando uma HDL. Em paralelo com a descrição do circuito, os arquivos de testes (*testbench*) guiam o processo de simulação (2). A etapa 3 consiste em executar a síntese para otimizar o projeto e executar o mapeamento de tecnologia para implementar a lógica usando recursos do dispositivo, como elementos lógicos e blocos de memória, ou seja, é feito a inferência de, por exemplo, *flip-flops*, *latches* e máquinas de estado de linguagens "comportamentais" para as células padrão do dispositivo alvo (Intel Corporation, 2005). Essa etapa gera um arquivo padronizado no nível de células lógicas (*cell-level netlist*) com o circuito otimizado.

A próxima etapa é o *placement and routing*, que usa os arquivos gerados na etapa anterior para alinhar os requisitos de lógica e tempo do projeto com os recursos físicos disponíveis do dispositivo. O *software* mapeia as funções lógicas nas células com a melhor localização para roteamento e temporização, seleciona caminhos de interconexão e atribui os sinais de entrada e saída aos pinos apropriados (Intel Corporation, 2005). Isso é normalmente feito de forma direta e automática pelo *software* de desenvolvimento fornecido pela fabricante da FPGA, mas é possível realizar atribuições de pinos e outras configurações manualmente. Para finalizar, é gerado um arquivo de configuração (5) do projeto que será usado para programar o *hardware* (6).

2.2 *Hardware* e *Software co-design*

O *co-design* de *hardware-software* foi um conceito que começou na década de 1990. Seu conceito central era o desenvolvimento simultâneo de componentes de *hardware* e *software* de sistemas eletrônicos complexos (Cadence PCB solutions, 2019). Até então, o *hardware* era projetado de uma forma independente do *software*, salvo algumas exceções. O *co-design* de *hardware* e *software* visa o projeto de Sistema em Chip (SoC) que envolve a integração de microprocessadores de uso geral, estruturas *Digital Signal Processor* (DSP), FPGA, núcleos *Application-Specific Integrated Circuit* (ASIC), periféricos de bloco de

Figura 2 – Fluxo de desenvolvimento de uma FPGA



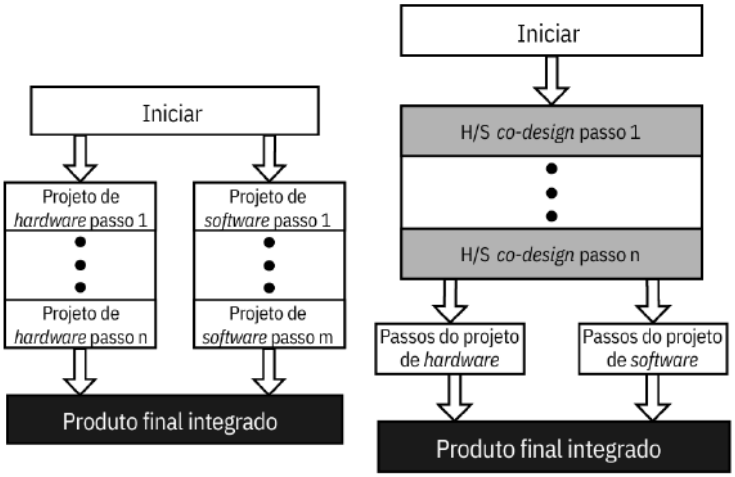
Fonte: (CHU, 2011)

memória e barramentos de interconexão em um chip (DARWISH; BAYOUMI, 2005).

Segundo Wolf (1994), *hardware* e *software* devem ser projetados em conjunto para garantir que a implementação não apenas funcione corretamente, mas também atenda às metas de desempenho, custo e confiabilidade. Na Figura 3a é possível observar o fluxo de desenvolvimento de um projeto tradicional, onde o *hardware* e *software* estão sendo projetados independentemente até a integração final do projeto. Essa forma pode trazer riscos ao produto final, já que não foram realizados testes ou simulações no processo de desenvolvimento para garantir que não haja nenhum conflito entre eles.

Na Figura 3b é apresentado o fluxo de desenvolvimento utilizando o conceito de *co-design*, onde o *hardware* e *software* são projetados de forma simultânea, fazendo todos os testes necessários para garantir a integração final e assim, economizando tempo e aumentando a eficiência. O *software* e *hardware* podem ser testados e simulados em um ambiente que atinge quase o mesmo desempenho que a implementação real (ANEMAET; AS, 2003).

Figura 3 – Fluxos de desenvolvimento



(a) Fluxo tradicional (b) Fluxo *co-design*

Fonte: Traduzido de (DARWISH; BAYOUMI, 2005)

Um dos motivos do aumento de uso *co-design* de *hardware-software* segundo Micheli e GUPTA (1997), deve-se a introdução da Ferramenta de Projeto Assistido por Computador (CAD), aumentando assim, a qualidade potencial, encurtando o tempo de desenvolvimento e desempenhando um papel estratégico fundamental no desenvolvimento do projeto, além do uso de processadores *softcore*. A implementação do projeto pode ter diferentes modelagens e pode ser necessário assumir estratégias distintas consoante o particionamento de *hardware* e *software*, e segundo Micheli e GUPTA (1997), pode ser decidido pelo projetista, com sucessivos refinamentos e modificações do modelo inicial, ou determinado por uma ferramenta CAD, até encontrar quais partes do modelo podem ser melhor implementadas em *hardware* e aquelas melhor implementadas em *software*.

2.2.1 *Softcore*

O termo *softcore* é empregado para processadores normalmente descritos em HDL e sintetizados em FPGAs, devido à alta densidade de lógica se comparado com CPLDs. Um processador *softcore* pode ser configurado e ajustado adicionando ou removendo recursos sistema por sistema para atender às metas de desempenho ou custo (CHU, 2011). Essa flexibilidade permite aos projetistas definir se o projeto terá foco na redução de custo, quantidade de periféricos de I/O ou no tamanho. Também possibilita que a FPGA possa ser trocada sem muitas complicações, uma vez que os processadores *softcores*, na sua maioria, são independentes de tecnologia e podem ser sintetizados em diversos modelos de dispositivos lógicos, sendo assim, são mais imunes a se tornarem obsoletos (TONG; ANDERSON; KHALID, 2006).

De acordo com Borisov, Valentina e Kukenska (2007), *softcores* de FPGA, possuem

conjuntos de instruções, unidade lógica aritmética, arquivos de registro e outros recursos especificamente adaptados para o uso eficiente de recursos de FPGA. Isso porque os fornecedores de processadores introduziram *softcores* adaptados para esse uso ao passar dos anos. Como mencionado, a flexibilidade é aumentada, mas em alguns casos pode comprometer a velocidade de execução, sendo necessário adicionar recursos externos de *hardware* dedicados.

Processador *hardcore*, por outro lado, não permite aos projetistas personalizar o funcionamento interno da CPU ou fazer qualquer alteração na sua arquitetura. Não pode ser reconfigurado em tempo de execução e normalmente não pode ser retirado do projeto quando não for mais útil, mas podem ter uma eficiência de área e um processamento mais rápido que os *softcores*, em geral.

De acordo com Borisov, Valentina e Kukenska (2007), as ferramentas de desenvolvimento de *softcore* são responsáveis pela parametrização do *softcore* e periféricos associados, pela implementação de barramentos, mapas de memória, estruturas de interrupção e periféricos de processador necessários.

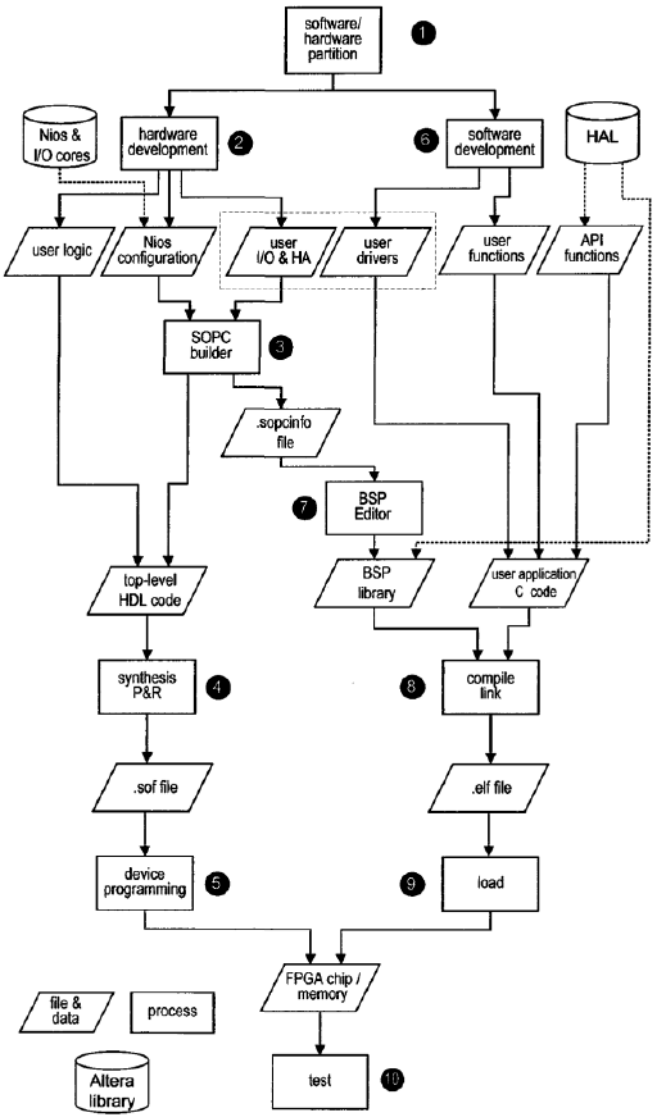
Um exemplo deste processador é o processador *softcore* Nios II baseado na arquitetura *Reduced Instruction Set Computer* (RISC). A Intel disponibiliza a ferramenta de desenvolvimento CAD e também um sistema para fazer a programação em *chip*, chamado *Platform Designer*, que era anteriormente conhecido como SOPC *Builder*. Isso permite aos projetistas configurar a plataforma utilizando o Nios e os periféricos. Outro *software* disponibilizado por essa fabricante, é o Quartus Prime, usado para sintetizar, programar e depurar seus projetos e construir sistemas embutidos nas FPGAs da Intel (TONG; ANDERSON; KHALID, 2006).

2.2.2 Fluxo de projeto com *softcore*

Fluxo de projeto de um sistema que recorre a um processador *softcore*, pode ser exemplificado com o fluxo de desenvolvimento do processador Nios II proposto por Chu (2011) e apresentado na Figura 4. Como mencionado, quando se utiliza este tipo de processador, o projetista tem a liberdade de adicionar, alterar, remodelar, particionar a plataforma conforme a necessidade do projeto. A parte do desenvolvimento de *hardware* que está sendo representada no lado esquerdo da Figura 4 possui o mesmo fluxo mostrado na subseção 2.1.4 com a diferença que o código HDL referente a plataforma que contém o Nios foi gerado automaticamente pelo *software Platform Design*.

No desenvolvimento de *software*, a Intel disponibiliza uma biblioteca que consiste em drivers de dispositivo de I/O, sendo rotinas de baixo nível para acessar periféricos de I/O e uma coleção de funções de alto nível disponíveis na Interface de Programação de Aplicações (API) (CHU, 2011). Os desenvolvedores de *software* podem usar essa API para

Figura 4 – Fluxo de desenvolvimento do processador softcore Nios II



Fonte: (CHU, 2011)

facilitar a integração com o *hardware*. Outra etapa nesse processo é realizada pelo editor *Board Support Package* (BSP) que verifica os drivers necessários para a implementação do sistema. Próxima etapa compila e vincula as rotinas de *software* e a biblioteca BSP e cria o arquivo de imagem final do *software* (ou seja, gera o arquivo .elf) para carregar a imagem do *software* na plataforma (CHU, 2011). Por fim, o sistema pode ser testado.

2.3 *Codec de áudio*

Um *codec* de áudio é responsável pela codificação e decodificação dos sinais de áudio, podendo ser implementado em *software* ou *hardware*. Um *codec* implementado por *software* funciona com o uso de algoritmos para realizar a compressão e expansão de um arquivo, onde cada *codec* pode usar um algoritmo diferente. *Codec* implementado

por *hardware* é usado para fazer a conversão de sinais de áudio utilizando dispositivos eletrônicos, onde o sistema que converte o sinal analógico de tempo contínuo em um sinal digital de tempo discreto é chamado de conversor A/D, e o processo de converter o sinal digital em analógico é conhecido como conversor D/A. O *codec* é normalmente utilizado para comprimir o tamanho de um arquivo, podendo ser baseado em perda de informação (diminuição da qualidade) ou sem perder a qualidade. O arquivo resultante pode ser transmitido por algum meio, onde após recebido, pode ser restaurado para o padrão original, convertido para outro formato ou ser simplesmente armazenado em uma Unidade de Disco Rígido (HDD), Unidade de Estado Sólido (SSD), memórias Flash ou na nuvem, por exemplo. E em alguns casos, o arquivo pode simplesmente ser reproduzido.

Codec também pode ser utilizado para compactar som e música de alta qualidade para armazenamento mais eficiente, *streaming* de áudio pela Internet e transmissão de voz por Rede Pública de Telefonia Comutada (RPTC), redes celulares e *Voice Over Internet Protocol* (VoIP) (HICSONMEZ; SENCAR; AVCIBAS, 2013).

2.3.1 Codificação digital

O processo para gerar um áudio digital ocorre pela conversão de um áudio analógico para digital através de um Conversor Analógico para Digital (A/D) ou *codec*. Os dados de áudio digital, para *Pulse-Code Modulation* (PCM), por exemplo, consistem em uma sequência de valores binários que representam o número de níveis do quantizador para cada amostra de áudio (PAN, 1993). Segundo Hicsonmez, Sencar e Avcibas (2013), existem mais de cem *codecs* de áudio usados para codificação e decodificação de áudio digital.

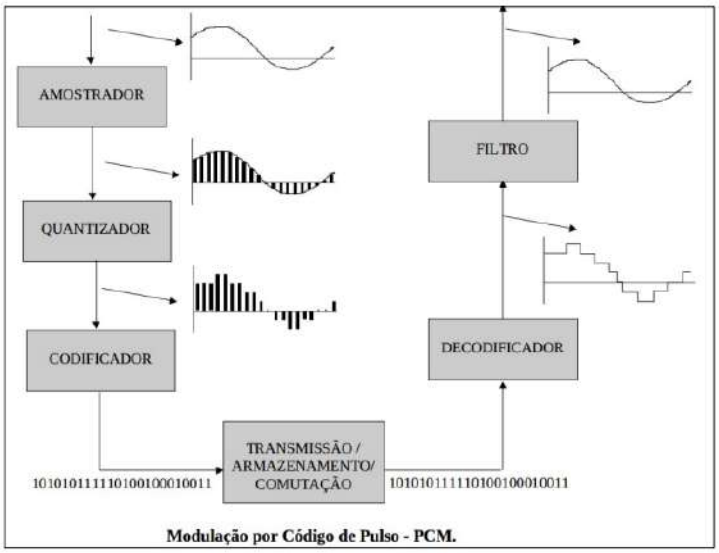
O processo para se codificar um sinal ou digitalizar um sinal analógico consiste em fazer a amostragem no tempo desse sinal, onde se retira amostras (amplitudes) de acordo com a frequência desejada. Após amostrados, esses sinais são quantizados em amplitudes de 0 e 1 (1 *bit*), por exemplo, e por fim, são codificados utilizando a codificação desejada. A representação digital de dados de áudio oferece muitas vantagens: alta imunidade a ruídos, estabilidade e reprodutibilidade e permite a implementação eficiente de muitas funções de processamento (por exemplo, mistura, filtragem e equalização) (PAN, 1993).

2.3.2 PCM

PCM é um dos vários tipos de codificadores existentes e um dos primeiros *codecs* definidos pela especificação G.711 formulada pela União Internacional de Telecomunicações (ITU-T) (MARTINS et al., 2010). PCM é um codificador utilizado para tornar um sinal discreto no tempo por meio do processo de amostragem. Esse processo utiliza uma frequência que deve ser duas vezes a frequência mais alta a ser codificada (Teorema de

Nyquist[1]) para garantir que o sinal consiga ser reconstruído no recebimento sem perdas. Por exemplo, para transmitir a voz humana que possui uma frequência de 3400 Hz é necessária uma taxa de amostragem de ao menos 6800 Hz. Conforme a ITU-T, a taxa de amostragem padrão para garantir banda de guarda (filtro), é de 8000 Hz.

Figura 5 – Codificação PCM



Fonte: (MOECKE, 2006)

Após o processo de amostragem, o sinal é quantizado linearmente em 13 *bits*, onde é realizado um mapeamento do sinal amostrado de acordo com suas amplitudes para gerar um novo símbolo discreto a ser transmitido. Após essa etapa, o sinal é codificado em uma sequência binária. Após a codificação, é realizada a compressão digital do sinal utilizando a lei A ou lei μ, reduzindo o sinal para 8 *bits*, chegando em 256 níveis de quantização ($2^8 = 256$). O sinal codificado é então, transmitido por um canal de transmissão unidirecional com taxa de transmissão de 64 Kbit/s (8x8000). A Figura 5 ilustra o processo de codificação e o processo de decodificação, onde após decodificado, é aplicado um filtro passa-baixa para ser possível recuperar o sinal original.

2.3.3 G.711

A norma G.711 possui duas leis de compressão, sendo a lei A e a lei μ. O uso da compressão faz com que a largura de banda ou recurso necessário para transmissão de um sinal seja reduzido significativamente. Essas leis foram propostas para se alocar *bits* de maneira não uniforme para representar sinais digitais, pois a audição humana não é igualmente sensível às baixas e às altas amplitudes (MARTINS et al., 2010).

[1] <https://www.elettroamici.org/pt/teorema-di-nyquist-shannon/>

A compressão seguida de expansão refere-se ao processo de primeiro compactar um sinal analógico na origem e, em seguida, expandir este sinal de volta ao seu tamanho original quando alcança seu destino (Cisco Systems, 2006). Isso é o que ocorre com essas leis, os *codecs* comprimem dados PCM lineares em dados logarítmicos e a quantidade de compressão pode aumentar conforme o tamanho das amostras.

A Lei A possui um alcance dinâmico maior do que a Lei μ, fazendo com que a segunda tenha um desempenho sinal/distorção melhor para baixos níveis de sinais. Outra diferença é que a lei μ codifica dados de fala de 14 *bits* em amostras de 8 *bits*, enquanto a lei A codifica dados de fala de 13 *bits* em amostras de 8 *bits* (AOKI, 2012).

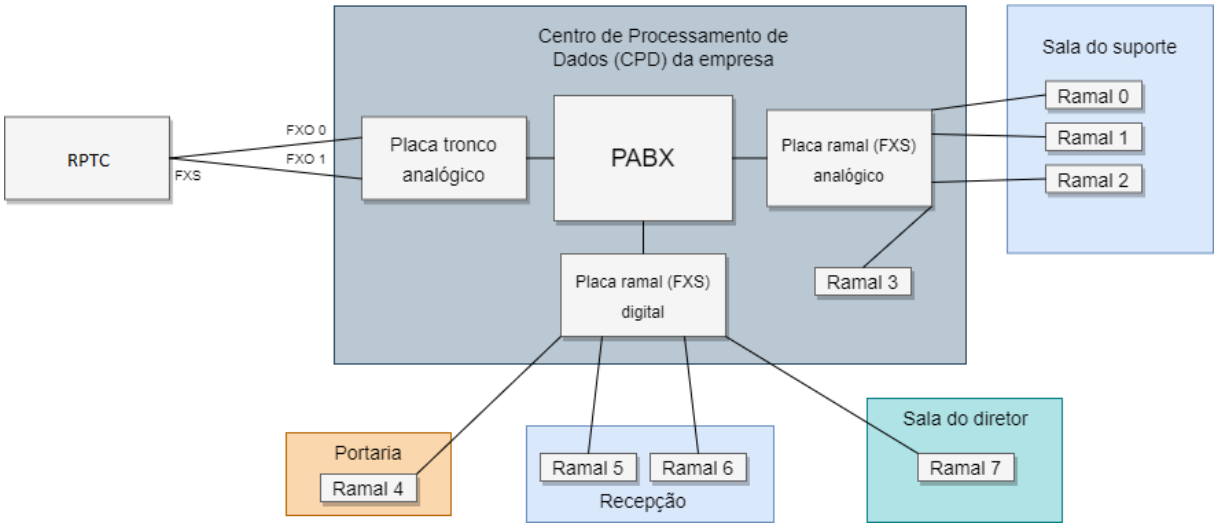
2.4 Placa de ramal

No mercado de telecomunicações, um produto que é muito utilizado para facilitar a comunicação entre as pessoas em um mesmo prédio, condomínio, fábrica ou empresa, é a central *Private Automatic Branch Exchange* (PABX) (HORN; MOIA, 2017). Essa central possibilita diversos tipos de atividade, desde serviços de *telemarketing*, suporte ao cliente, fornecer comunicação entre casas em um condomínio fechado ou até mesmo para comunicação entre cômodos em uma casa de grande porte. Por meio dela, é possível efetuar ou receber chamadas internas (ou externas) utilizando um ponto telefônico (ramal) instalado no ambiente, apartamento ou departamento (HORN; MOIA, 2017). Apesar da existência de aparelhos mais sofisticados (digitais e VoIP) os aparelhos analógicos ainda são os mais utilizados nesses cenários devido ao custo e facilidade de instalação.

As centrais PABXs digitais e analógicas necessitam de uma interface *Foreign eXchange Subscriber* (FXS) para poder conectar e interligar aparelhos analógicos e digitais. Essa interface é conhecida como placa de ramal ou placa FXS, ela fornece alimentação para os aparelhos telefônicos. Um ramal conectado a essa interface pode utilizar as facilidades que a central telefônica oferece, por exemplo: transferência de chamadas, captura de chamadas, estacionamento de chamadas e identificação de chamadas. Cada placa de ramal possui um *codec* que é o responsável por fazer a codificação e decodificação do sinal de áudio, conforme mencionado na seção 2.3. Um exemplo é a placa de ramal analógico da Impacta 16 da Intelbras. Essa placa inclui todas as funcionalidades para interface com até quatro ramais analógicos, entregando canais de voz digitalizados.

Os fabricantes modularizam as centrais para aumentar a versatilidade do produto. Na Figura 6 é possível observar um diagrama de blocos de uma central modular, onde a empresa possui duas linhas analógicas contratadas com a RPTC, possui quatro ramais analógicos conectados na placa de ramal analógico e quatro ramais digitais conectados na placa digital distribuídos em cinco setores diferentes da empresa, onde todos os ramais podem ter acesso à linha externa e podem fazer chamadas internas entre os ramais.

Figura 6 – Diagrama de blocos genérico de uma central modular



Fonte: Própria

2.5 Interface SPI

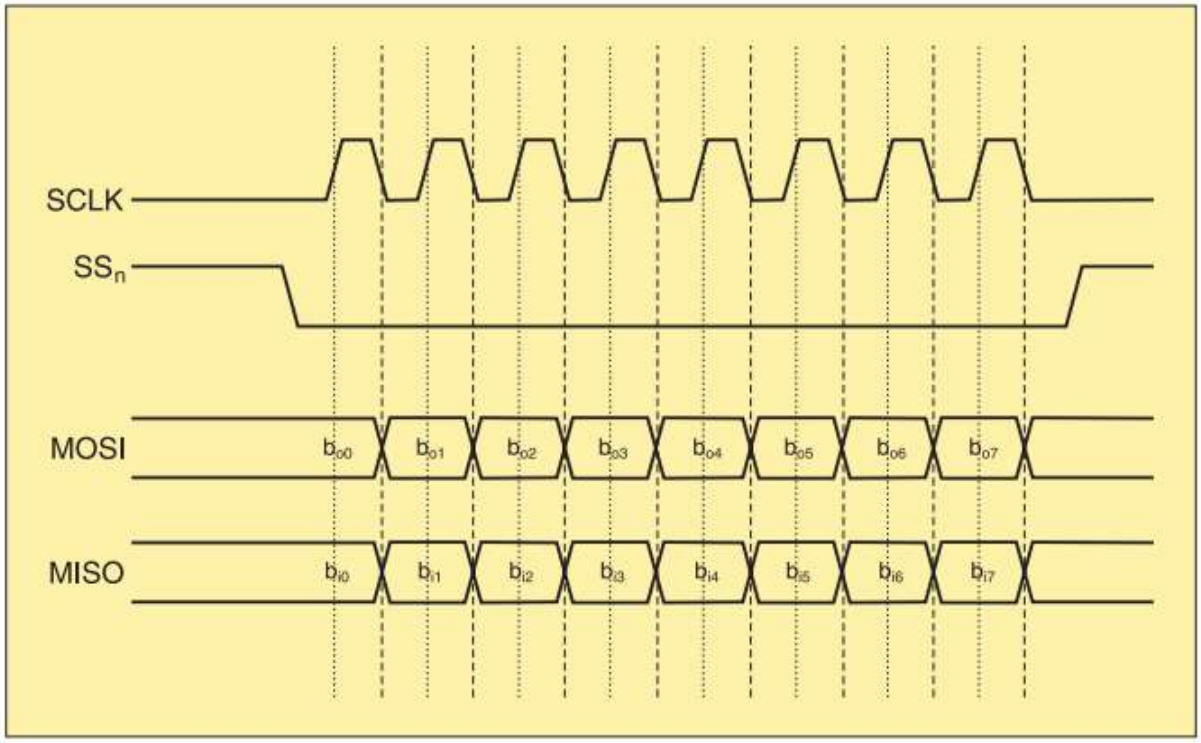
A *Serial Peripheral Interface* (SPI) é um protocolo desenvolvido pela Motorola visando estabelecer uma comunicação entre um microcontrolador (*host*) a dispositivos periféricos, por um *link* serial, síncrono e *full duplex* (SHINGARE; PATIL, 2013). O procolo utiliza o modelo *master-slave*, onde o *master* pode ser conectado a diversos *slaves*.

O barramento utilizado para estabelecer a comunicação entre o *master* e o *slave* possui quatro linhas de sinal, sendo eles:

- *Serial Clock Signal* (SCLK): Sinal de *clock* gerado pelo *master* e utilizado para sincronizar a comunicação entre ele e os *slaves*.
- *Slave Select* (SS): Sinal para fazer a seleção do *slave* a ser utilizado. Também é chamado de Chip Select (CS).
- *Master Out Slave In* (MOSI): Sinal para enviar dados do *master* para o *slave*.
- *Master In Slave Out* (MISO): Sinal para enviar dados do *slave* para o *master*.

Conforme Oudjida et al. (2009), o protocolo SPI não define uma taxa de transmissão máxima, podendo variar segundo a implementação. Os dados transmitidos também são variáveis, é possível definir o tamanho e a quantidade das palavras a serem transferidas. O protocolo SPI também não possui nenhum mecanismo para confirmar o recebimento de dados e não oferece nenhum controle de fluxo (LEENS, 2009).

Figura 7 – Linhas de sinal do barramento SPI



Fonte: (LEENS, 2009)

Na Figura 7 é possível observar a comunicação serial, síncrona e *full duplex* de um dado de 8 *bits* na interface SPI, onde no mesmo pulso do sinal SCLK, o *master* está escrevendo um *bit* no MOSI e lendo um *bit* do MISO, após selecionado o *slave* pelo sinal SS.

3 DESENVOLVIMENTO

Neste capítulo é apresentada a metodologia que dividiu e organizou as atividades a fim de alcançar o objetivo proposto. As etapas do desenvolvimento consistem basicamente no seguinte: i) estudo da placa de ramal; ii) criação do modelo de integração com o *kit* de FPGA; iii) projeto e confecção da placa adaptadora; iv) estudo dos componentes de lógica programável; v) desenvolvimento da plataforma de teste baseado em um cenário simplificado; vi) testes e resultados.

3.1 Metodologia

Para organizar o desenvolvimento, o trabalho foi segmentado nas seguintes etapas:

1. Estudo da placa de ramal: O primeiro passo para concluir a proposta apresentada é entender o funcionamento da placa de ramal da Impacta 16. Dessa forma, essa etapa consistiu em estudar detalhadamente o *datasheet* do componente principal e o esquema elétrico da placa de ramal. Além disso, foram analisadas todas as informações referente ao seu funcionamento, comportamento, configuração e operação, tanto de *hardware* quanto de *software*. Como resultado, foi criado um diagrama de blocos lógico da placa de ramal, o qual foi utilizado como referência durante todas as etapas do desenvolvimento.

2. Modelo de integração: Nessa etapa, foi analisado e definido o modelo de integração que seria adotado entre a placa de ramal, o *kit* de desenvolvimento da FPGA e a fonte de alimentação, sendo consolidada a configuração do *hardware* em relação aos pinos lógicos e de alimentação. Como resultado, foi elaborado um diagrama lógico de integração da placa adaptadora.

3. Placa adaptadora: Após o estudo da placa de ramal e a definição do modelo de integração, nessa etapa, foi desenvolvido o esquemático e o projeto do circuito impresso da placa adaptadora. Também foi executado todo o processo de prototipação e montagem da placa para validá-la. A manufatura industrial e montagem da versão final foram executadas posteriormente.

4. Estudo dos componentes de lógica programável: Etapa cujo propósito foi estudar vários componentes sintetizados para criar a estrutura lógica de controle e comunicação dentro da FPGA para integração com a placa de ramal. Para isso, foi necessário estudar componentes do próprio ambiente de desenvolvimento da Intel, como o *softcore* Nios II, FIFOs, PLL e controlador SPI. Outro componente estudado foi o

controlador TDM de código aberto, o qual foi utilizado para interface dos canais digitais.

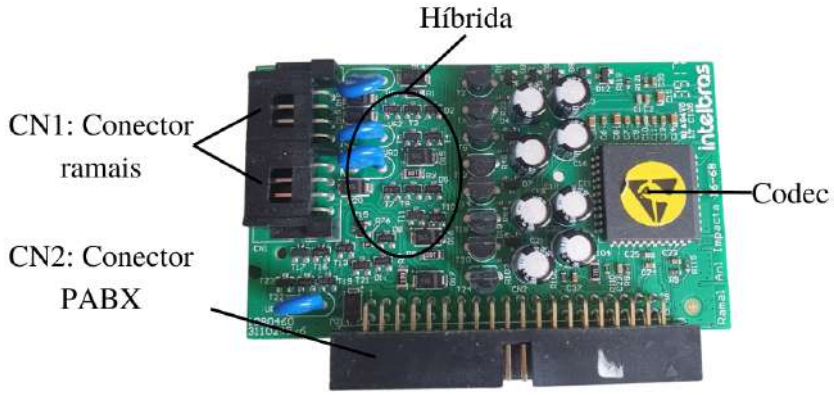
5. Criação da plataforma de desenvolvimento: Nesta etapa, utilizando como base os estudos realizados, foi desenvolvida a plataforma para um cenário simplificado. Além dos blocos prontos, foi necessário implementar novos componentes para fazer o controle da FIFO e do controlador TDM, assim como, adaptar o bloco SPI para funcionar em modo *half duplex*, padrão exigido pela placa de ramal para a interface de controle e configuração.

6. Testes e resultados: Por fim, nessa etapa, para validação das implementações, foram criados os seguintes cenários: i) comunicação entre a placa de ramal e o *kit* FPGA; ii) FIFOs de recepção e transmissão; iii) controlador TDM; iv) alinhamento dos *time slots*; v) Comportamento do processador *softcore*.

3.2 Placa de ramal Impacta 16

A placa de ramal da Impacta 16 da Intelbras (Modelo **4990083**), apresentada na Figura 8, tem alta disponibilidade no IFSC Câmpus São José e no mercado nacional. Além disso, exceto pela fonte, ela integra todos os componentes necessários para funcionamento com ramais analógicos. A placa possui dois conectores para quatro ramais e um conector de 40 pinos, usado para conectar na base da central PABX.

Figura 8 – Placa de ramal da Impacta 16



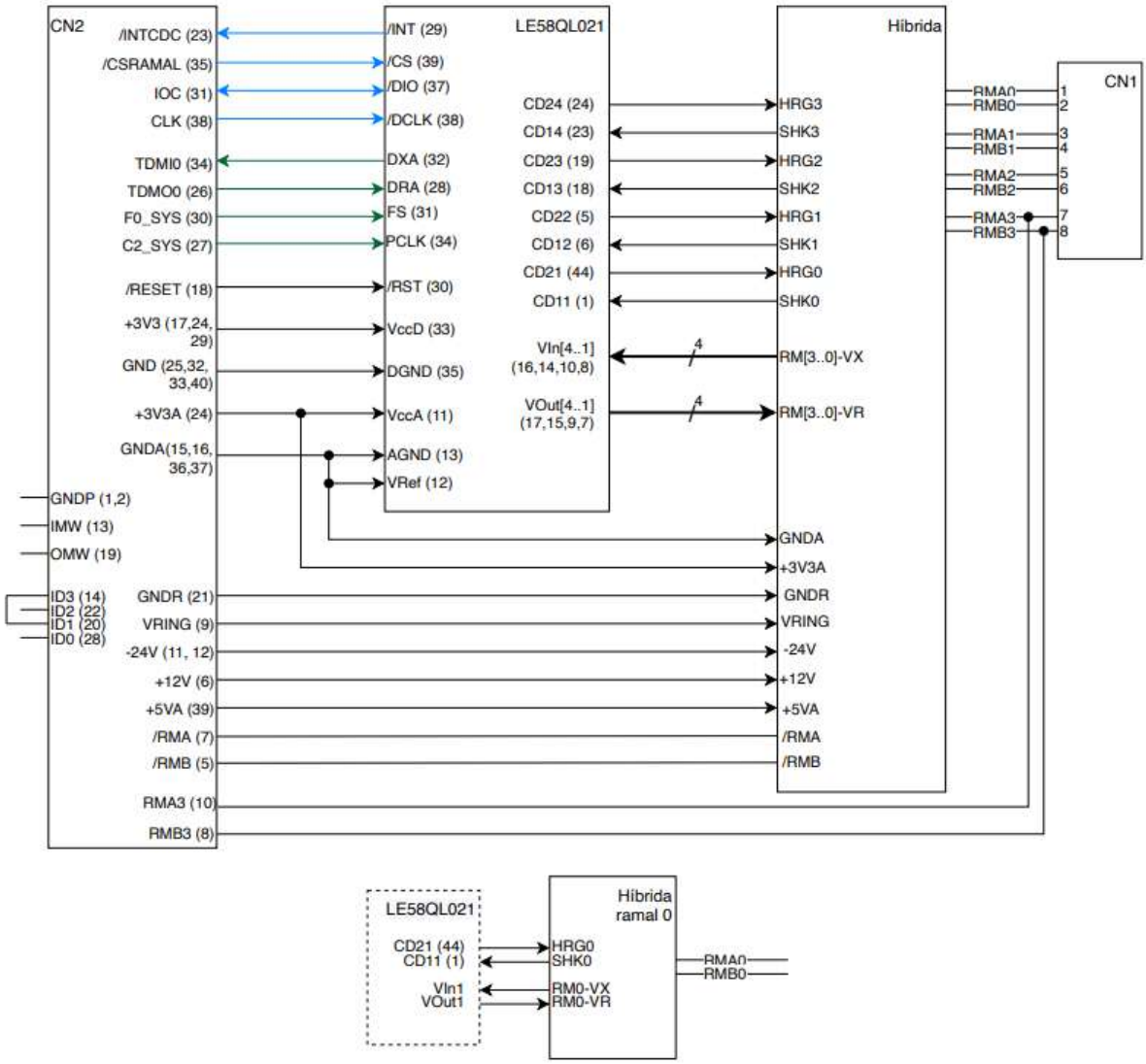
Fonte: Própria

Para fazer a conversão do sinal analógico para digital e vice-versa, a placa utiliza o *codec* LE58QL021BVC da Legerity, o qual possui quatro Circuitos de Interface de Linha de Assinante (SLIC) integrados. Cada SLIC é responsável por monitorar o *status* do ramal conectado e é ligado a um bloco analógico chamado de híbrida, o qual possui as seguintes funcionalidades: i) separação dos sinais de Transmissão (TX) e Recepção (RX) do SLIC

que serão enviados para a CPU; ii) casamento de impedância com o terminal telefônico; iii) sensor de gancho (sinal SHK); iv) controle de *ring* dos ramais (sinal HRG).

Para representar a placa de ramal, foi elaborado o diagrama de blocos apresentado na Figura 9, que teve como base, o estudo do esquema elétrico fornecido pela fabricante da placa. Em azul (as 4 primeiras conexões entre o *codec* e o CN2), são as portas utilizadas pela MPI e as conexões em verde (as 4 conexões após os azuis) são as portas utilizadas pelo *Timer Slot Assigner* (TSA). As próximas seções detalham as informações do *codec* e das portas utilizadas.

Figura 9 – Diagrama de blocos da placa de ramal Impacta 16

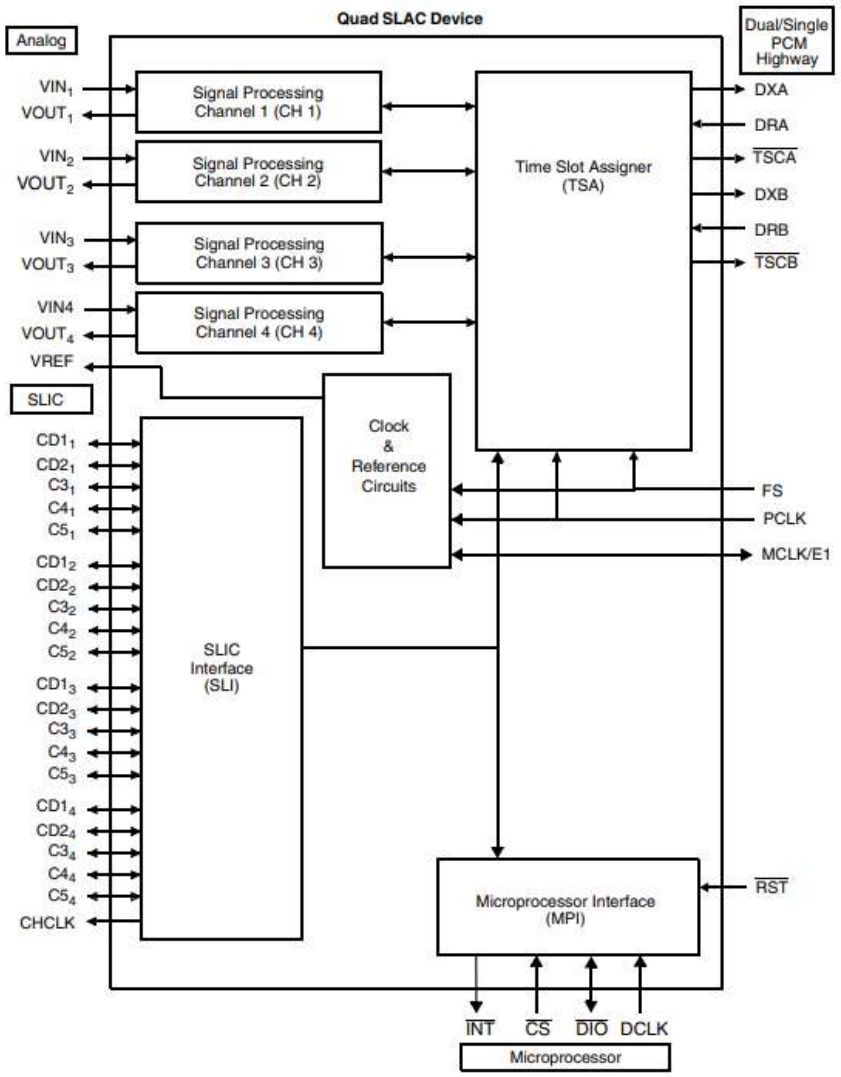


Fonte: Própria

3.2.1 *Codec* LE58QL021BVC

Definido como Circuito de Processamento de Áudio de Linha de Assinante de Baixa Tensão Quádrupla (QLSLAC), é responsável pelos SLICs e funções de codificação e decodificação dos sinais analógicos para PCM, operando em até quatro canais independentes que podem ser programados pelo usuário conforme a necessidade. O QLSLAC, também chamado de *codec*, é formado pelos blocos MPI, interface PCM, SLIC, *clock* e os blocos de processamento de sinal por canal, conforme pode ser visto na Figura 10.

Figura 10 – Diagrama de blocos do *codec* LE58QL021BVC



Fonte: (Legerity, 2006)

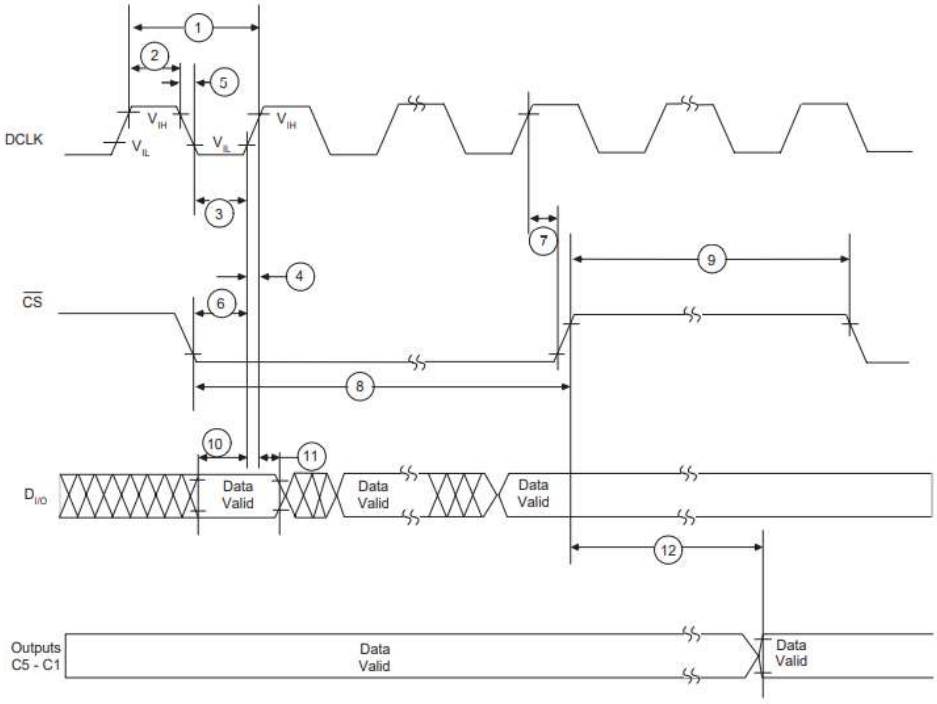
3.2.1.1 Interface do Microprocessador

A Interface do Microprocessador (MPI) é o bloco responsável pela comunicação via barramento serial síncrono, com o sistema de controle externo. Esse bloco distribui as configurações recebidas para os outros blocos do *codec* e fornece informações do estado

para o sistema externo. Além disso, este bloco é responsável pelo circuito de *reset* do *codec* (Legerity, 2006). A interface com a MPI consiste em quatro pinos: i) o *serial data input/output* (DIO), onde os dados de controle são escritos e lidos com o *bit* mais significativo primeiro; ii) o *data clock* (DCLK), que determina a taxa de transmissão, com frequência máxima de 8,912 MHz; iii) *chip select* (CS), sendo um pino de entrada que habilita o *codec* para receber ou enviar dados; iv) a interrupção (INT), que possui a função de notificar a CPU se algum telefone está fora do gancho.

O *codec* implementa uma comunicação serial síncrona com características similares a um escravo *serial peripheral interface* (SPI), exceto pelo DIO que é um pino bidirecional e multiplexa os dados de entrada e saída. Nas Figura 11 e Figura 12 é possível observar os comportamentos do barramento no modo de entrada e saída, respectivamente. Em ambos os modos, é necessário habilitar o *codec*, zerando o CS. O envio ou recepção dos *bits* é sincronizado pelo sinal DCLK. Os tempos máximos e mínimos para o adequado funcionamento do barramento são apresentados nas imagens (anotações de 1 até 19) e detalhadas em Legerity (2006).

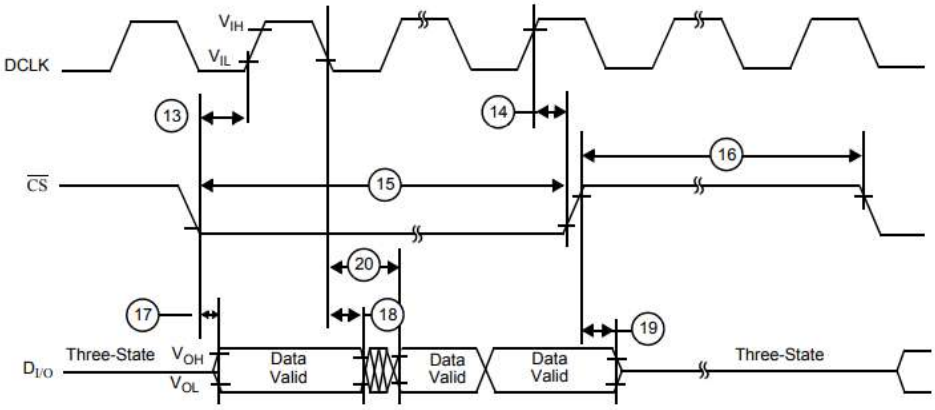
Figura 11 – Modo entrada de dados do MPI



Fonte: (Legerity, 2006))

No Quadro 1, é apresentado alguns dos comandos MPI, em formato hexadecimal, disponíveis para configurar o *codec*. Por exemplo, para configurar o *master clock* a ser utilizado pela placa, é necessário configurar o registrador enviando o comando de escrita `0x46` e após enviar o valor `0x1A` que representa o *clock* de 8,192 Mhz. Para fazer a leitura e verificar se o *clock* foi configurado, pode-se usar o comando `0x47`, que deverá retornar o valor `0x1A`. Todas as configurações e descrições detalhadas de cada comando estão

Figura 12 – Modo saída de dados do MPI



Fonte: (Legerity, 2006))

disponíveis em Legerity (2006). Ao longo deste documento, para facilitar a leitura, será adotado a nomenclatura `(c:hex, v:hex)` para representar o comando utilizado e o valor enviado ou recebido do *codec*. Para o primeiro exemplo apresentado (*master clock*), a utilização da nomenclatura ficaria da seguinte forma: `(c:0x46, v:0x1A)`.

Quadro 1 – Alguns comandos MPI disponíveis

Comandos (Hex)	**Descrição**
`04h`	Reset do *hardware*
`0Eh`	Ativar o *codec*
`44/45h`	Configurar o *time slot*
`46/47h`	Configurar o registrador
`4A/4Bh`	Ativação de canal e registro do modo de operação
`4Dh`	Ler o registro de dados em tempo real
`4Fh`	Ler o registro de dados em tempo real e limpar a interrupção
`40/41h`	Configurar o *time slot* de transmissão
`42/43h`	Configurar o *time slot* de recepção
`50/51h`	Configurar impedância e ganhos analógicos
`52/53h`	Configurar o registrador de I/O do dispositivo SLIC
`54/55h`	Configurar a direção de I/O do dispositivo SLIC e *bits status*
`60/61h`	Configurar funções operacionais
`80/81h`	Configurar coeficientes de filtro GX
`82/83h`	Configurar coeficientes de filtro FR
`6C/6Dh`	Configurar máscara de interrupção dos registradores
`C8/C9h`	Configurar registrador de tempo de *debounce*

Fonte: Adaptado de (Legerity, 2006)

3.2.1.2 Interface SLIC (SLI)

Os pinos CD[1-2], C3, C4 e C5 do *codec*, que fazem interface com a SLIC, segundo Legerity (2006), são pinos programáveis de entrada ou saída, projetados para monitorar ou

controlar o estado da SLIC ou outro dispositivo qualquer, por exemplo, *relays* de controle, LEDs ou outra função que exija um sinal compatível com TTL para controle.

Os CD[1-2] são utilizados normalmente para controlar as campainhas e sensores de gancho dos ramais, uma vez que há comandos exclusivos (c:0x4D/c:0x4F) para obter as informações desses dois pinos dos quatro canais de uma única vez, diminuindo o tempo de acesso e economizando processamento. Além disso, é possível habilitar um pré-processamento desses pinos para evitar repique (CD1) ou ativar filtros (CD2).

A diferença dos pinos C3, C4 e C5 em relação às portas CD[1-2], é que não há pré-processamento e esses pinos são acessados canal por canal, ou seja, é preciso ler ou escrever nos pinos C3, C4 e C5 do canal 0, depois do canal 1 e assim por diante.

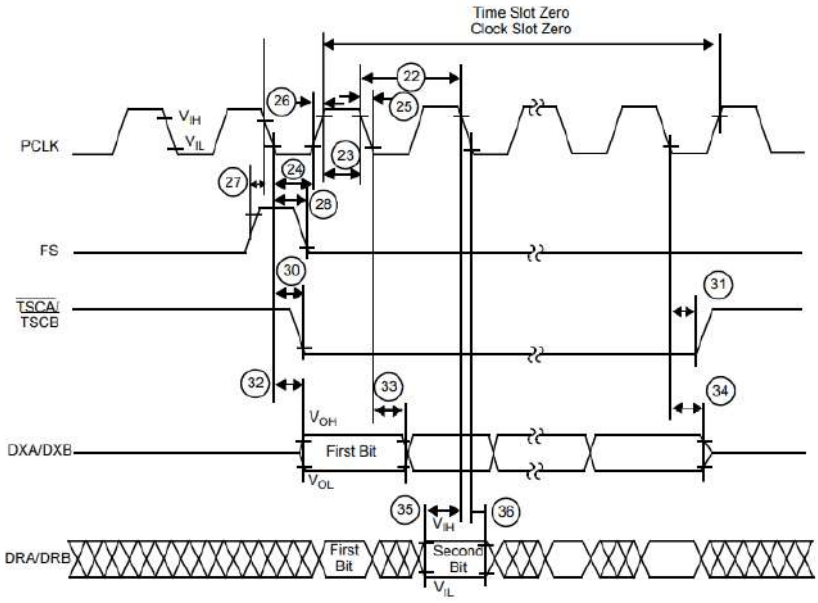
3.2.1.3 Interface PCM

Essa interface é responsável por transmitir o áudio digital, no formato PCM, dos quatro ramais multiplexados em um único barramento. Pode ser programado pelo usuário nos pinos de saída DXA e DXB ou nos pinos de entrada DRA e DRB. Os intervalos de transmissão do canal são controlados pelos pinos TSCA e TSCB, onde a frequência é definida pelo usuário entre 128 kHz a 8,192 MHz. Os dados transmitidos ou recebidos no barramento PCM podem ser no formato de código compactado de 8 *bits*, com um *byte* de sinalização opcional de 8 *bits* na direção de transmissão, ou código linear de *16 bits*. O primeiro formato (8 *bits*) ocupa um *slot* de tempo a cada intervalo (*Frame Sync*), enquanto o segundo formato (16 *bits*) ocupa dois *slots* (Legerity, 2006).

Os sinais analógicos de entrada (VIN) são digitalizados e então convertidos para PCM utilizando a lei A ou a lei μ, após isso, é transmitido para o barramento PCM por meio do TSA, que permite a transmissão dos dados em até 128 canais de 8 *bits* (128 x 64kbps = 8 Mbps) (Legerity, 2006). A interface PCM permite ao projetista fazer ajustes nos coeficientes dos filtros digitais, descritos em Legerity (2006), que podem ser utilizados para alterar o ganho ou a taxa de amostragem do sinal de entrada, por exemplo. A taxa dessa transmissão é definida pelo PCM *Clock* (PCLK).

Na Figura 13 é possível observar uma transmissão do DXA/DXB que se inicia a partir da borda negativa do PCLK, o *Frame Sync* (FS) (27) identifica o *time slot 0* e o *clock slot 0* do *frame* PCM. TSCA/TSCB após o *delay* do PCM vai para 0 (30), ativando assim o intervalo de transmissão. Após transmitir o primeiro *bit* (33), é configurado a interface PCM para receber o *bit* nas portas DRA/DRB (35) na próxima descida do PCLK. Após transmitir todos os bits, TSCA/TSCB volta para 1.

Figura 13 – Barramento PCM configurado na borda de descida



Fonte: (Legerity, 2006))

3.2.2 Análise do funcionamento da placa de ramal

Com as informações teóricas levantadas acerca do funcionamento, comportamento, configuração e operação do *codec* LE58QL021BVC, foi realizado, com o auxílio de um analisador lógico da Saleae[1], modelo *Logic Pro* 16, a análise da comunicação SPI da placa de ramal com a central PABX Impacta 16. Esse analisador possui 16 entradas digitais e analógicas, uma frequência máxima de amostragem de 100 MHz, e os dados são transferidos via *Universal Serial Bus* (USB) para o computador.

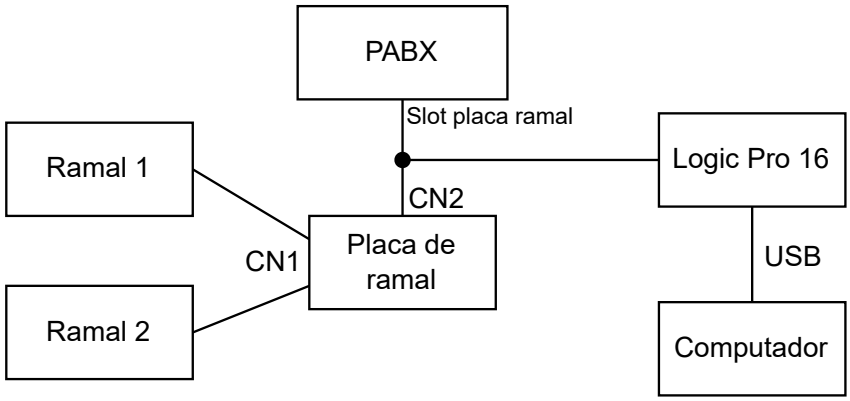
O analisador foi utilizado para capturar a transmissão de dados da placa de ramal relacionada com os pinos da MPI e do TSA para complementar o estudo realizado sobre o *codec* na subseção 3.2.1. Após a captura, foi utilizado o *software Logic* 2, que a Saleae disponibiliza para realizar a exibição e análise da captura realizada. Essa análise teve como objetivo verificar os comandos MPI utilizados pelo *codec* ao:

- Inicializar;
- Retirar um ramal do gancho;
- Realizar uma chamada;
- Atender uma chamada.

[1] <https://www.saleae.com>

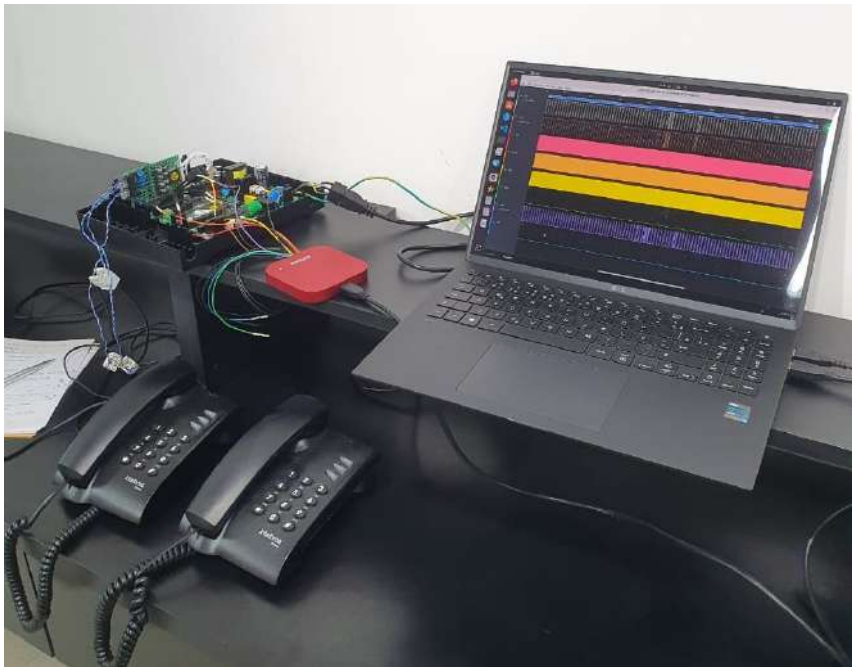
O *setup* do sistema utilizado para fazer as capturas é apresentado na Figura 14 e o cenário montado utilizando esse *setup* pode ser visto na Figura 15. A seguir, será discutido as principais operações identificadas nas capturas.

Figura 14 – Diagrama de blocos do *setup* do sistema



Fonte: Própria

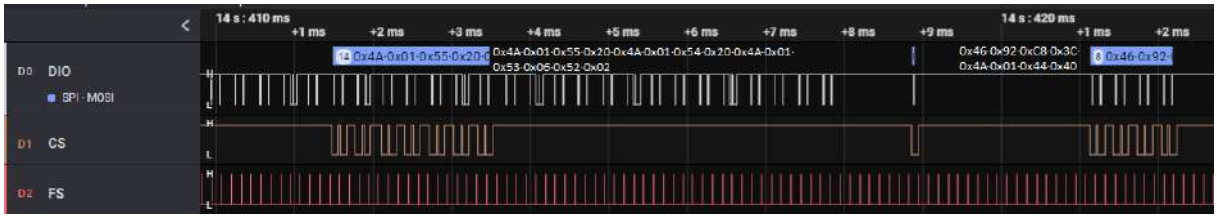
Figura 15 – Cenário montado para análise do funcionamento da placa de ramal



Fonte: Própria

A primeira análise feita foi referente à inicialização do *codec*, apresentada na Figura 16. Nessa observação, foi averiguado que o *codec* é configurado pelo `DIO` da seguinte forma: modo de interrupção com saída *open drain*; `PCLK` de 2,048 MHz como sendo o *master clock* (c:0x46, v:0x92); tempo de *debounce* de 15ms para leitura do pino de gancho (c:0xC8, v:0x3C); pinos da interface SLIC são definidos como entradas e ativos (c:0x54, v:0x20); transmite dados na borda positiva do `PCLK` (c:0x44, v:0x40).

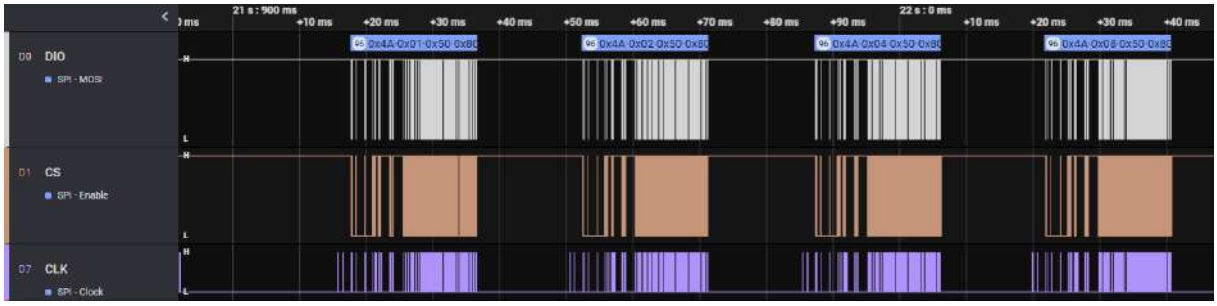
Figura 16 – Inicialização do *codec*



Fonte: Própria

Após o *codec* fazer o alinhamento do *master clock* configurado, com a sincronização de quadros, foi verificado que é feita a inicialização dos seus quatro canais. Para reduzir o custo de mapear os endereços lógicos desses canais e ter a ciência de qual ramal está habilitado em um único comando, o *codec* utiliza um registrador de canais de 4 *bits*, em que cada *bit* representa um canal. Na Figura 17, é possível observar ser feita a inicialização do canal 1 (c:0x4A, v:0x01), em seguida do canal 2 (c:0x4A, v:0x02), canal 3 (c:0x4A, v:0x04) e do canal 4 (c:0x4A, v:0x08). As demais configurações realizadas nessa etapa foram configurações de filtros, coeficientes e ganhos dos canais.

Figura 17 – Inicialização dos quatro canais do *codec*

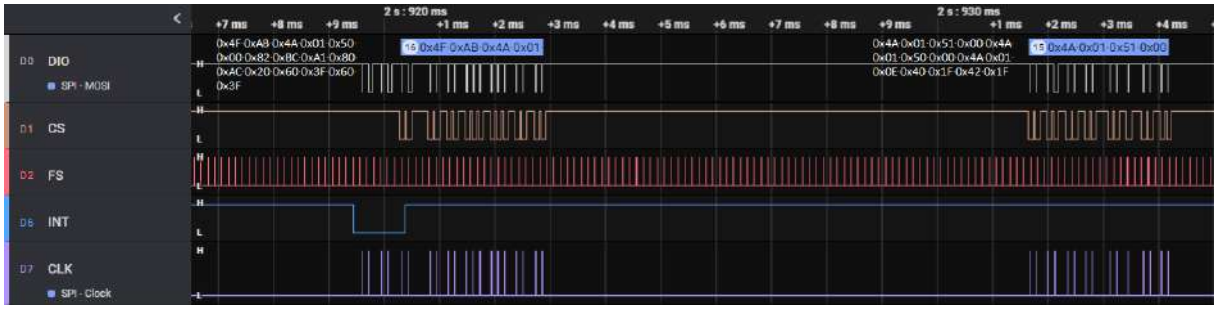


Fonte: Própria

Com a central inicializada, foi retirado um ramal (canal 1) do gancho para verificar o comportamento do *codec*. De início, foi possível identificar que o ramal estava fora do gancho com a descida do sinal de interrupção `INT`, como pode ser observado na Figura 18. Essa identificação também é possível pela leitura do registrador de tempo real (c:0x4F, v:0xAB). Após identificar essa interrupção, foi verificado que o *codec* configura ganhos analógicos (c:0x50, v:0x00), coeficientes de filtros (comando 0x82 e comando 0x80), estabelece a compressão dos dados com codificação lei A (c:0x60, v:0x3F), ativa o canal (comando 0x0E) e define o *time slot* 31 para transmissão (c:0x40, v:0x1F) e recepção (c:0x42, v:0x1F) para o canal 1.

Posteriormente, foi realizado uma chamada do ramal 1 para o ramal 2, e a captura dessa chamada pode ser vista na Figura 19. Nessa parte, o *codec* configura para o canal 2, ganhos analógicos (c:0x50, v:0x80), ativa o canal (comando 0x0E), define o *time slot*

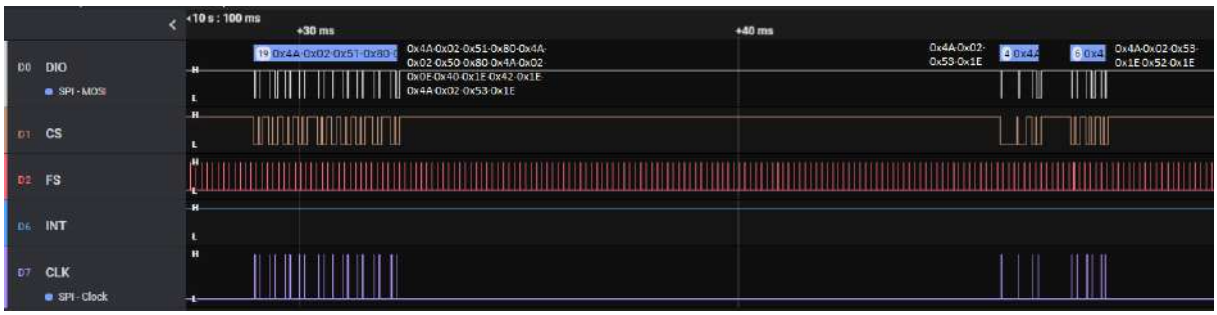
Figura 18 – Ramal 1 fora do gancho



Fonte: Própria

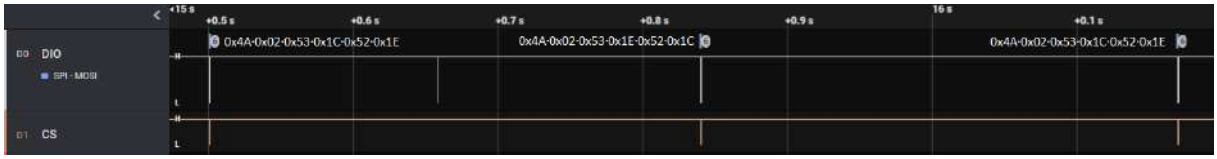
30 para transmissão (c:0x40, v:0x1E) e recepção (c:0x42, v:0x1F). Após um período, foi observado o *ring* no canal 2, e na captura, foi possível constatar que quando ocorre o *ring*, o *codec* altera a direção da SLIC para entrada (c:0x52, v:0x1C) e quando o *ring* está desativado, altera a direção da SLIC para saída (c:0x52, v:0x1E). A captura do *ring* pode ser vista na Figura 20.

Figura 19 – Chamada sendo realizada para o ramal 2



Fonte: Própria

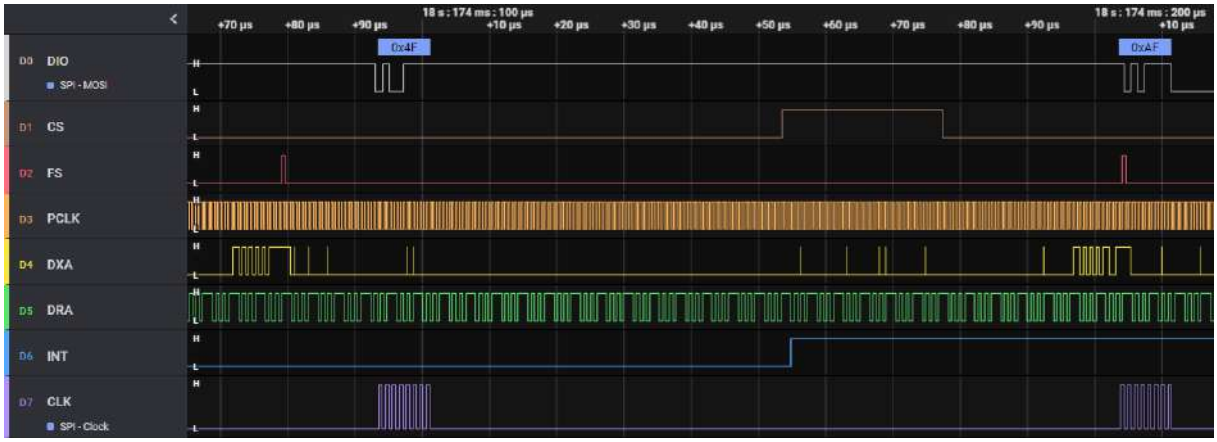
Figura 20 – Habilitando o *ring* no ramal 2



Fonte: Própria

Por fim, ao retirar o ramal 2 do gancho e fazer o atendimento, verificado pela interrupção e pelo comando de leitura do registrador de tempo real (c:0x4F, v:0xAF), foi confirmado que a transmissão do sinal PCM (DXA) está sendo realizada nos *time slots* 30 e 31, conforme configurado. Na recepção (DRA), não foi possível chegar em uma conclusão do seu comportamento, pois parece estar recebendo dados ou ruídos a todo momento. Na Figura 21 é possível visualizar o atendimento.

Figura 21 – Chamada estabelecida entre ramal 1 e o ramal 2



Fonte: Própria

3.3 Modelo de integração

Como já mencionado, o uso de componentes de lógica programável permite ao projetista ter liberdade de alterar a plataforma desenvolvida conforme a necessidade do projeto. Para essa proposta, foi adotado o *kit* de desenvolvimento DE2-115 baseado na família de FPGAs *Cyclone IV E* da Intel e o processador *softcore* Nios II. Esse tipo de processador, como mencionado na subseção 2.2.1, possibilita uma fácil conexão com periféricos, possibilitando a criação de um *System on Programable Chip* (SoPC). Esse *kit* foi escolhido pela alta disponibilidade no *campus* e, porque facilitaria a integração com a placa de ramal, uma vez que possui um conector de expansão de 40 pinos, ligado diretamente na FPGA, com o mesmo formato do conector utilizado na placa de ramal. Além disso, também possui um oscilador de 50 MHz, utilizado como fonte de *clock* pelos componentes do sistema, memória SRAM, *Flash* e interface de programação e *debug* facilitadas via conexão USB *Blaster*.

Para ser possível a integração da placa de ramal com a plataforma de desenvolvimento, foi necessário elaborar um diagrama lógico de integração de uma placa adaptadora para servir como um *Backplane*[2] para conectar a placa de ramal da Impacta 16 e também a placa fonte necessária para alimentar a placa, devido ao fato da plataforma não fornecer as tensões necessárias para o correto funcionamento da placa de ramal.

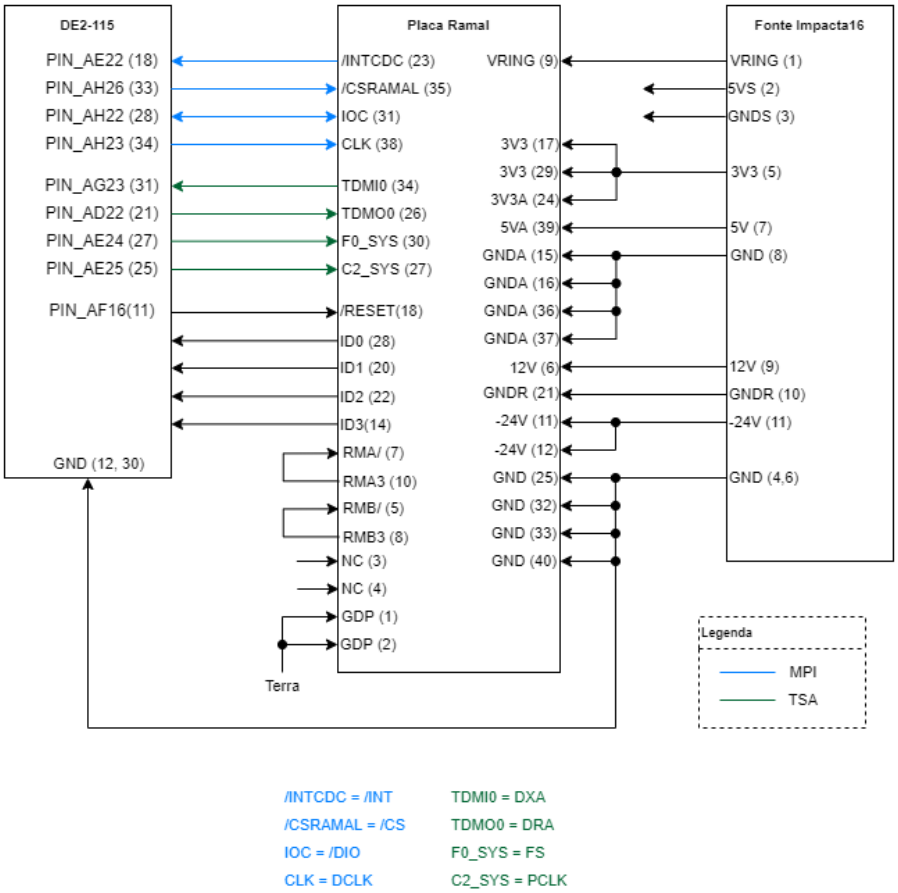
A placa fonte aproveitada é a mesma utilizada na central PABX Impacta 16, por ser uma fonte estável, que fornece todas as tensões necessárias para o funcionamento da placa de ramal. Ela é responsável por fornecer as seguintes tensões: i) 3.3 V, utilizada para o controle de *ring*; ii) 5 V, responsável pelo controle de gancho dos ramais; iii) -24 V, responsável pela alimentação dos aparelhos analógicos e digitais; iv) 100 V, tensão

[2] *Backplane* é uma placa de circuito impresso contendo conexões (slots) para conectar outras placas

necessária para garantir o *ring* em todos os ramais; v) 12 V, utilizada para fazer a alimentação do circuito de transmissão e recepção de áudio. Na Figura 22 é possível visualizar o diagrama de blocos da placa adaptadora.

A maioria dos pinos descritos na placa de ramal desse diagrama já foram comentados ao longo da seção 3.2, faltando citar os pinos RMA e RMB utilizados para fazer o acoplamento de energia quando usados na central Impacta, não sendo necessário para essa proposta. Os pinos ID[0...3] foram reservados para uma eventual necessidade de adicionar algum outro recurso a placa. Os pinos apresentados no bloco DE2-115 são os *General Purpose Input/Output* (GPIO) que foram escolhidos da placa de expansão do *kit* para prover a interface de comunicação com a FPGA.

Figura 22 – Diagrama lógico de integração da placa adaptadora



Fonte: Própria

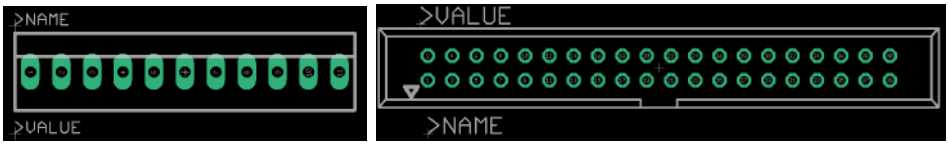
3.4 Placa adaptadora

O diagrama apresentado na Figura 22 foi utilizado como base para desenvolver uma *Placa de Circuito Impresso* (PCB) a ser utilizada na proposta discutida. A PCB foi desenvolvida com o uso da versão gratuita do Eagle, um *software* de Projeto Eletrônico

Programável (EDA), onde foi possível criar um esquemático correspondente ao diagrama de blocos da placa adaptadora para projetar a PCB.

Na produção do esquemático do projeto, foi utilizado o símbolo (*footprint*) 22-23-2111 que é um conector de 11 pinos para ser usado na placa fonte e o 057-040-1 que é um conector de 40 pinos para a interface com a plataforma e a placa de ramal, ambos estão inclusos na biblioteca do Ealge. Na Figura 23a e na Figura 23b são apresentados os símbolos conforme aparecem na biblioteca do *software*. No Apêndice E, é apresentado o esquemático gerado no editor de esquemático Eagle CAD para projetar a PCB.

Figura 23 – *Footprints* utilizados na placa adaptadora

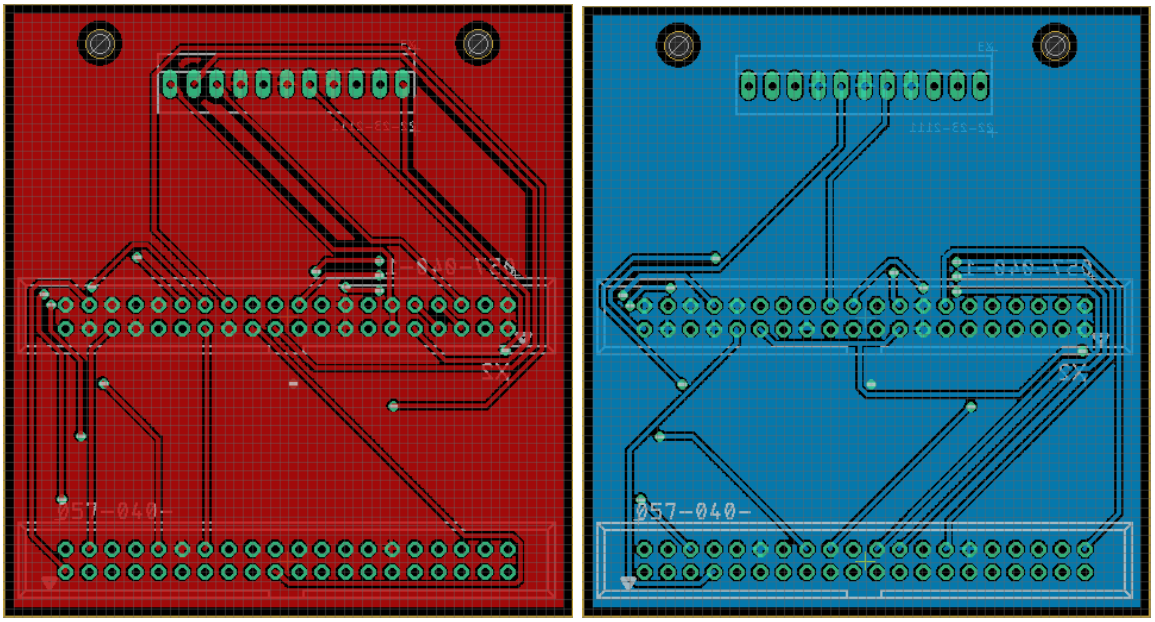


(a) *Footprint* 22-23-2111 (b) *Footprint* 057-040-1

Fonte: Própria

Com o esquemático pronto, foi iniciado o projeto da PCB. O Eagle permite fazer o desenho da placa a partir do esquemático usando o editor de projeto de PCB. Nele é necessário fazer o posicionamento dos componentes, ajustes de trilhas entre as ilhas, malhas, furos e o plano de terra em até duas camadas. Na Figura 24a é apresentada a camada superior da PCB e na Figura 24b a camada inferior da PCB desenvolvida.

Figura 24 – Placa adaptadora em ambiente de desenvolvimento



(a) Camada superior da PCB (b) Camada inferior da PCB

Fonte: Própria

Para realizar a prototipação da placa, foi necessário seguir algumas orientações no desenvolvimento do *layout* da placa para se obter um melhor resultado, dentre elas, foi necessário realizar os testes de verificação de regras elétricas (ERC), e a verificação de regras de projeto (DRC). A ERC, é uma função do *software Eagle* que faz a análise do esquema elétrico para localizar a existência de possíveis erros de conexões. As regras do projeto foram configuradas usando a função *design rules* com as configurações apresentadas abaixo.

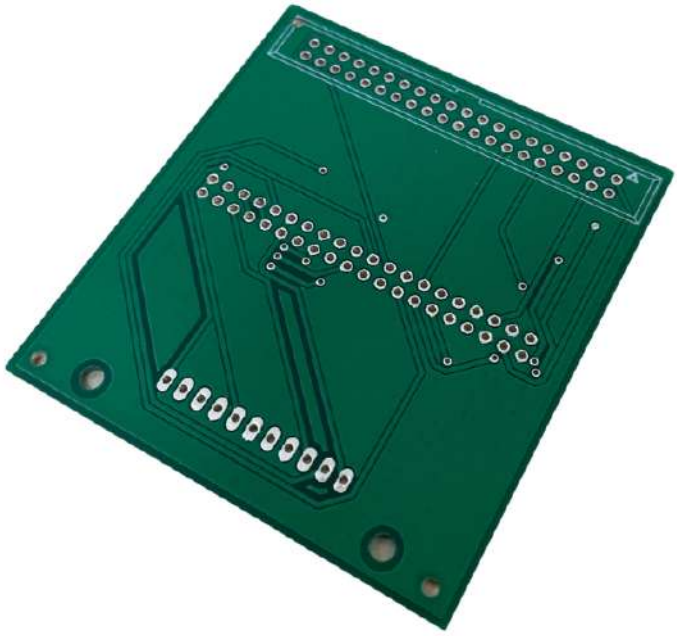
- *Clearance* de 12 mil (distância entre vias, trilhas, malhas e furos);
- Largura das trilhas: 20 mil;
- Vias: 0.6 mm;
- Borda no layer 20 (*Dimensium*);
- Uma camada *Top* e uma camada *Bottom*;
- Plano de terra conectado no GND na camada *Top* e uma no *Bottom*;
- Respeitar tamanho 265 mm x 200 mm;
- Sem espaçamento entre a malha e a borda da placa;
- Margem (120 mils) entre as trilhas e a borda da placa.

Depois da revisão do projeto do circuito impresso, foi realizado a exportação dos arquivos contendo as informações da PCB no formato *Gerber* e enviado ao Departamento Acadêmico de Eletrônica (DAELN) no IFSC Câmpus Florianópolis, onde foi realizada a prototipação da placa em uma fresadora LPKF. Em seguida, foi efetuada a montagem dos conectores e realizado teste de continuidade para validar o circuito desenvolvido. A placa protótipo serviu para validar o projeto e fazer os primeiros testes da placa de ramal conectada ao *kit* da FPGA. Após a validação do protótipo, a fabricação da placa foi contratada junto a uma empresa especializada. Na Figura 25 é possível observar a placa pronta sem a montagem dos conectores.

3.5 Estudo dos componentes de lógica programável

Após o desenvolvimento da placa adaptadora, começou-se o estudo dos blocos que iriam compor o sistema que seria sintetizado na lógica programável. Por exemplo, foi necessário fazer um estudo sobre o componente de filas em *hardware*, para armazenar temporariamente os dados e mantê-los ordenados. Outra necessidade que surgiu, foi encontrar uma forma de fazer a multiplexação dos 32 canais PCM e estabelecer a comunicação

Figura 25 – Placa de circuito impresso fabricada



Fonte: Própria

serial com a placa de ramal. Para isso, foram realizadas pesquisas e estudos para encontrar um controlador capaz de fazer a multiplexação por divisão de tempo (TDM), que atenda as necessidades do projeto. Também foi necessário fazer um estudo dos componentes do ambiente de desenvolvimento da Intel para configurar um sistema baseado em Nios. Em sequência, será apresentado o resultado dos estudos realizados.

3.5.1 *FIFOs*

Para gerenciar e organizar os sinais transmitidos e recebidos via comunicação serial entre a placa de ramal e o processador Nios II, foi adotado o uso de duas First In, First Out (FIFO), uma para recepção e outra para transmissão dos dados. A FIFO armazena os dados transmitidos em uma memória temporária e fornece o controle de fluxo desses dados armazenados, possibilitando que o *software* possa manipulá-los com ordenação.

Para o projeto, foi definido a necessidade de uma FIFO utilizando os blocos internos de memória da FPGA, disponível no catálogo do *software* Quartus, com largura de 8 *bits* e profundidade de 256 palavras. As configurações da FIFO foram realizadas através do *MegaWizard Plug-in Manager*, onde foi configurado a FIFO para operar com dois *clocks*, um para leitura e um para escrita, configurado os sinais de controle a serem utilizados e a forma que os dados serão acessados.

Dentre os sinais configurados, o sinal `wrreq` faz o controle de escrita e o `rdreq` controla a leitura da FIFO. Os sinais `wrclk` e `rdclk` são, respectivamente, o *clock* de escrita e leitura dos dados. O controle para identificar quando a FIFO está cheia se faz pelo

`wrfull` e quando está vazia pelo `rdempty`. Por fim, o `data[7..0]` é a interface de entrada dos dados que serão armazenados na FIFO, e o `q[7..0]` a saída dos dados armazenados.

3.5.2 Controlador TDM

Após a análise do funcionamento da placa de ramal, foi identificado que ela opera com uma frequência de *clock* de 2,048 Mhz e faz a transmissão dos dados usando 8 *bits*. Com isso, outra necessidade que surgiu foi encontrar uma forma de fazer a multiplexação desses 32 canais para transmitir e receber os dados nos canais corretos utilizando a frequência de 2,048 MHz, sabendo que, o FS identifica o início do *time slot* 0. Depois de algumas pesquisas e estudos, foi proposto o uso de um controlador TDM disponível na comunidade *OpenCores*[3] e que também está disponível no GitHub[4], o qual fornece as funcionalidades básicas de Multiplexação por Divisão de Tempo (TDM) a serem usadas com o *codec* da placa de ramal.

Segundo Khatib (2001), alguns dos recursos básicos desse sistema são: multiplexação de 32 *time slots* com taxa de *bits* de 2,048 Mhz, interface de comunicação serial de entrada e saída, leitura e escrita do *slot* TDM do barramento serial, conexão com outras interfaces de comunicação serial, conexão com filas externas e outas facilidades.

Para integrar esse controlador TDM ao projeto proposto, foi necessário entender alguns detalhes do seu funcionamento interno, por exemplo, as máquinas de estados[5] usadas para controlar a leitura e escrita dos *bits* recebidos na comunicação serial, assim como, estudar uma forma de adicionar as filas externas (i.e., FIFO) para garantir que os *bits* recebidos da placa de ramal sejam ordenados corretamente e sem perdas. Após levantar todas essas informações acerca do funcionamento, foi realizado o mapeamento dos sinais utilizados pelo controlador TDM para adaptar ao barramento Avalon[6] da Altera, pois toda a estrutura do projeto original do controlador TDM é compatível apenas com o barramento *WishBone*[7]. No Quadro 2 é apresentado a relação entre os sinais do controlador TDM usados e os sinais a serem usados no projeto.

[3] <https://opencores.org/projects/tdm>
[4] <https://github.com/freecores/tdm>
[5] <https://github.com/freecores/tdm/blob/master/code/tdm_cont/core/tdm_cont.vhd>
[6] <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf>
[7] <https://wishbone-interconnect.readthedocs.io/en/latest/index.html>

Quadro 2 – Adaptação dos sinais usados no Controlador TDM para funcionar no Avalon

Sinal TDM	**Sinal Avalon**	**Direção**	**Descrição**
rst_n	reset	Entrada	Reset do sistema
CLK_I	CLOCK_50	Entrada	Clock do sistema
NoChannels	"11111"	Entrada	Quantidade de time slots usados
DropChannels	"00000"	Entrada	Quantidade de time slots ignorados
C2	clk_2M	Entrada	Clock da interface serial
DSTi	TDMI0	Entrada	Recepção dos dados seriais
DSTo	TDMO0	Saída	Transmissão dos dados seriais
F0_n	fs_wire	Entrada	Sincronização do frame (FS)
F0od_n	F0od_n_wire	Saída	Atraso na sincronização de quadro
RxD	rxd_wire	Saída	Recepção de dados (bytes)
RxValidData	RxValidData	Saída	Dados válidos para receber
FrameErr	FrameErr	Saída	Erro no dado recebido
RxRead	frame_end	Entrada	Inicia a leitura dos dados
RxRdy	RxRdy_view_wrie	Saída	Sinaliza que um byte está completo
TxD	txd_wire	Entrada	Transmissão de dados (bytes)
TxValidData	TxValidData	Entrada	Dados válidos para transmitir
TxWrite	'1'	Entrada	Inicia a escrita dos dados
TxRdy	TxRdy_view_wire	Saída	Sinaliza que um byte completo

Fonte: Própria

3.5.3 *Softcore* Nios II

De modo a levantar informações sobre o funcionamento e configuração do *softcore* Nios II presente no *kit* de desenvolvimento da Intel, foi necessário fazer um estudo nas documentações disponibilizadas pela Intel referente ao *softcore* e aos blocos *Intellectual Property* (IP) disponíveis no catálogo do *software Platform Designer* para identificar os possíveis componentes necessários para fazer a configuração do sistema Nios para a proposta deste trabalho.

Inicialmente, foi verificado que para fazer a configuração do Nios, é necessário utilizar uma interface *JTAG UART* para estabelecer um *link* entre o computador onde o *kit* de desenvolvimento está conectado e o processador, por uma interface chamada *USB Blaster*. Após estabelecido esse *link*, é possível utilizar o módulo chamado *JTAG Debug* para permitir que o computador conectado controle o *softcore* Nios II em tempo real.

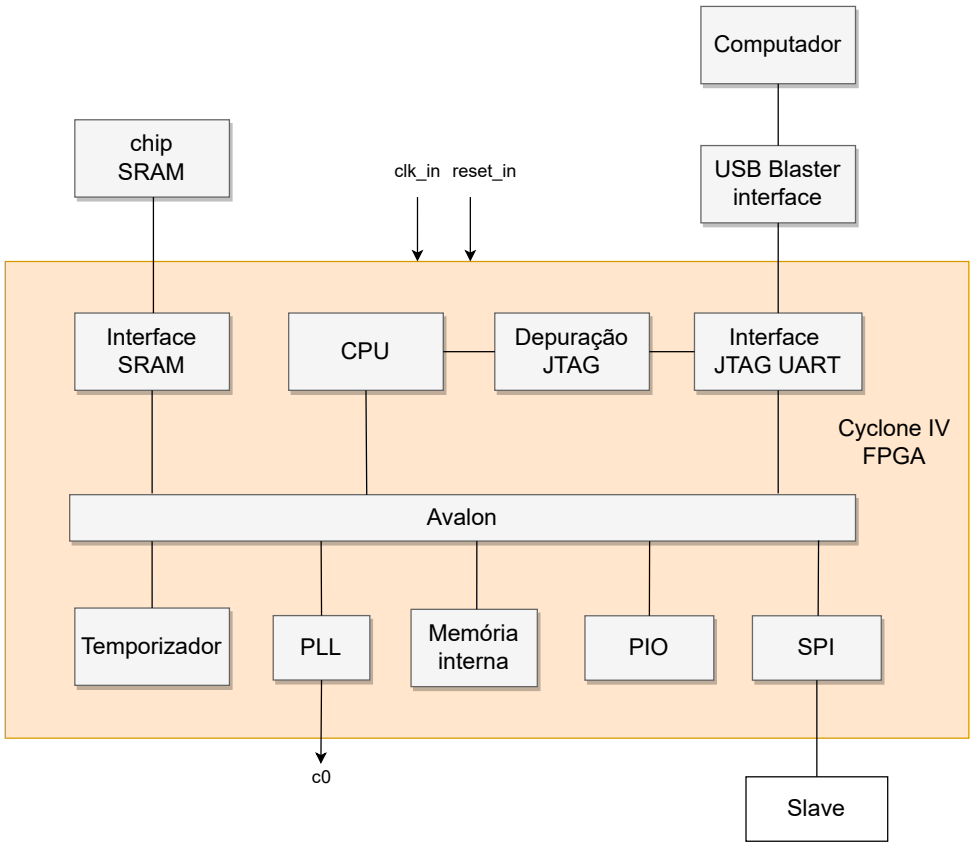
Na versão do *software* Quartus Prime utilizado (20.1), foi verificado haver duas implementações do Nios para ser utilizada no sistema, o **Nios II/e** e o **Nios II/f**. O primeiro é uma versão econômica (*economy*), ocupando menos recursos da FPGA e a segunda, é uma versão rápida (*fast*) com finalidade de oferecer um melhor desempenho ao sistema, fornecendo mais recursos, ao custo de um maior consumo de lógica. Para o projeto, foi definido que a versão **Nios II/e** é suficiente para a implementação proposta, além de ser uma versão gratuita (*fast* necessita de licença). Esses componentes são interligados por

meio do barramento Avalon.

Outro componente necessário para o funcionamento do processador é a memória interna (*onchip-memory*), que são blocos de memória disponibilizados pela FPGA para serem usados no processamento do Nios. Também é possível utilizar um componente de interface SRAM para utilizar um *chip* externo de memória. Foi identificando também, a possibilidade de usar um componente (*timer*) para temporizar as operações realizadas pelo processador.

Para fornecer um novo valor de *clock* para o sistema, sincronizado com o gerado pela FPGA, é possível utilizar um componente de PLL (*altpll*) que recebe um *clock* de entrada e gera na saída um *clock* segundo os parâmetros de multiplicação e divisão utilizados na configuração. Esse *clock* pode ser exportado para ser utilizado em todo o sistema.

Figura 26 – Exemplo de um sistema Nios II



Fonte: Própria

Com a base para a configuração do sistema definido, foi necessário estudar as formas de comunicação que poderiam ser utilizadas para a proposta. Dentre as formas estudadas, foi identificado que o componente *Parallel Input/Output* (PIO) poderia ser utilizado para habilitar/desabilitar alguma função se configurado com largura igual a 1 (pode assumir o valor 0 ou 1). Outra possibilidade averiguada, foi o componente SPI *3 Wire Serial* configurado com o tipo *master*, que pode ser utilizado para receber e enviar dados

através da configuração MISO e MOSI para um *slave* (i.e. placa de ramal). Na Figura 26 é possível visualizar um diagrama de blocos que representa a arquitetura mencionada para gerenciar o sistema Nios II.

3.6 Criação da plataforma de desenvolvimento

Nessa etapa, foi proposto um cenário simplificado, no qual fosse possível validar os objetivos propostos sem aumentar demasiadamente a complexidade dos requisitos para geração da plataforma de *hardware* programável – CPU *softcore* e periféricos – e *software*. Mesmo assim, além dos blocos prontos e da lógica de integração, foi necessário implementar um bloco em VHDL para fazer o controle da FIFO de transmissão e da FIFO de recepção, assim como, inserir uma lógica para fazer o controle da transmissão serial entre a placa de ramal e o processador. Em continuidade, a plataforma de *hardware* programável foi criada e configurada para possibilitar o desenvolvimento do *software* de controle da placa de ramal. Após estabelecido essa plataforma e o *software*, foi possível fazer a integração e testes básicos no sistema. A seguir, será apresentado com mais detalhes as etapas citadas.

3.6.1 Cenário proposto

Nessa etapa, foi estipulado o cenário mínimo para validar o funcionamento da implementação realizada. O cenário definido, consiste em fazer uma função de *hotline* sem DTMF nos ramais. Essa função permite que um ramal consiga acessar outro ramal (chamar) apenas retirando o monofone do gancho. Normalmente, existem duas opções de *hotline*, com retardo, quando é configurado um tempo, e apenas após esse tempo, ocorre o *hotline*, e sem retardo, quando a função é realizada imediatamente ao retirar do gancho. Por facilidade, o cenário proposto envolve realizar o *hotline* sem retardo. Para ser possível chegar no resultado esperado desse cenário, foi necessário implementar duas máquinas de estados, uma para controlar a leitura e escrita das FIFOs, e outra para fazer a comutação dos canais, e inserir uma lógica para estabelecer a comunicação serial entre a placa de ramal e o processador, conforme será apresentado a seguir.

3.6.1.1 Controlador FIFO

O controle de leitura e escrita das FIFOs e do controlador TDM, poderia ser feito de duas maneiras. A primeira consiste em implementar um novo componente em VHDL ao projeto, incluindo toda a lógica para fazer o controle utilizando a FPGA, e a segunda, seria criar a lógica por *software*, utilizando o Nios II.

Utilizar o processador para fazer esse controle, traria uma maior flexibilidade para o sistema, devido às facilidades de um *softcore* (veja subseção 2.2.1), porém, para o cenário proposto, por possuírem domínios de *clock* diferentes, poderia aumentar a complexidade

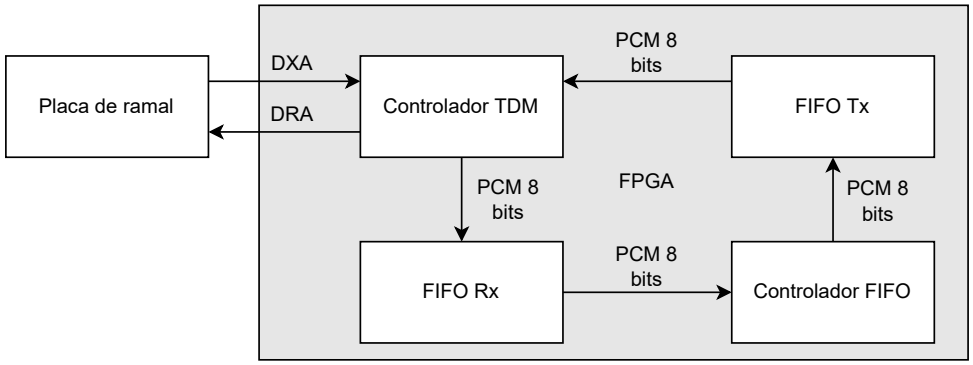
da implementação. Utilizar a FPGA para o controle, teria a vantagem de ser possível executar simulações (*testbenchs*) para validar o comportamento da lógica implementada, que usando o Nios, não seria possível. Pensando nisso, optou-se em fazer o componente em VHDL, definido como Controlador FIFO, para incluir a lógica de controle utilizando a FPGA.

Para auxiliar na implementação desse bloco, foi necessário implementar uma entidade para auxiliar no controle e identificação dos canais usados (Código A.3 do apêndice). Essa nova entidade, foi implementada para ter três sinais auxiliares, sendo eles: `frame_start`, `frame_end` e `frame_num`. Esses sinais, tem a finalidade de, respectivamente, indicar o início de um novo *byte*, o fim do *byte* e o número do canal.

Posteriormente, foi definido o fluxo de dados pretendido para o controlador FIFO. Foi estabelecido que o sinal PCM transmitido serialmente pela placa de ramal pelo pino DXA, deve entrar na porta serial do controlador TDM. Após o controlador TDM receber um sinal PCM completo (8 *bits*), a saída deve ser atribuída a FIFO de recepção para fazer a ordenação dos dados recebidos. A saída da FIFO de recepção seria, então, atribuída para o controlador FIFO, no qual, deve possuir uma lógica para fazer a inversão dos dados transmitidos nos canais. Após passar pela lógica do controlador FIFO, a saída é atribuída para a FIFO de transmissão, onde após o sincronismo, deve atribuir o sinal PCM (8 *bits*) para o controlador TDM, que deve atribuir serialmente os dados para o pino DRA da placa de ramal, estabelecendo assim, o fluxo de transmissão e recepção de dados. A fim de tornar mais simples o entendimento deste fluxo proposto, foi elaborado o diagrama de blocos apresentado na Figura 27.

Para atingir o objetivo desse fluxo, inicialmente, foi elaborada uma máquina de estado finita, em que sua lógica comportamental fora representada em um diagrama *Algorithmic State Machines* (ASM) que pode ser visto na Figura 28. Essa máquina de estado tem por objetivo definir o momento de leitura e de escrita das filas, assim que os dados estiverem prontos para tal.

Figura 27 – Diagrama de blocos do controlador FIFO

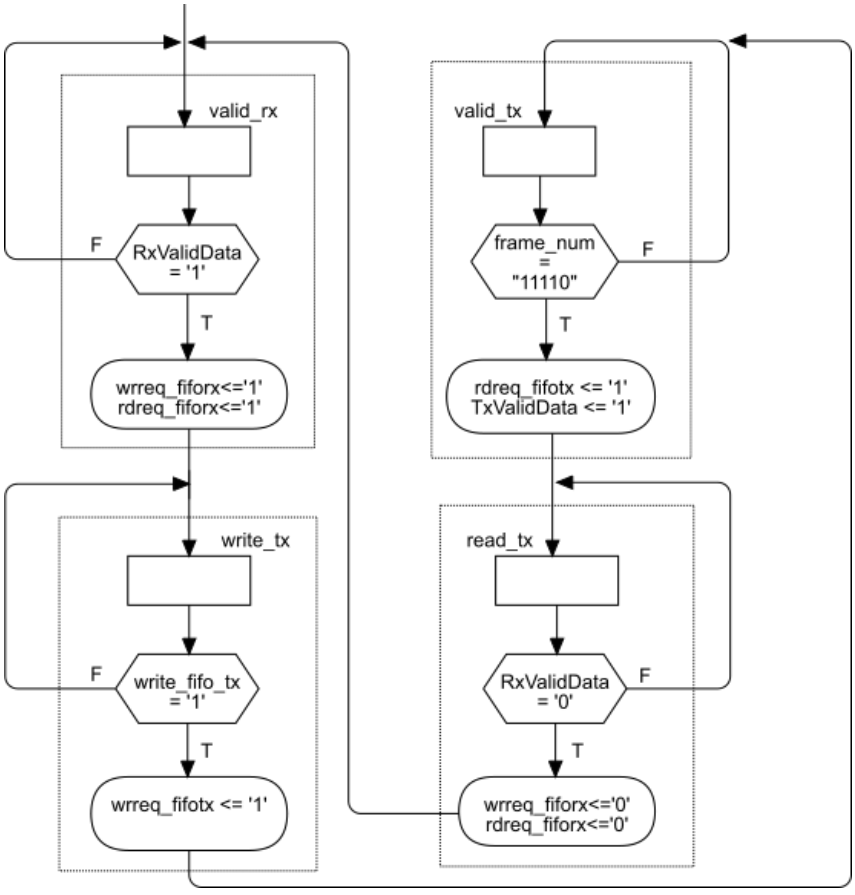


Fonte: Própria

O funcionamento básico dessa máquina de estado, consiste em habilitar a escrita e leitura da FIFO de recepção sempre que o controlador TDM possuir um quadro válido para ser recebido e desabilitado quando não tiver dados. Assim que a FIFO de recepção tiver um quadro válido armazenado, é então, habilitada a escrita na FIFO de transmissão, onde os dados serão armazenados até o próximo `FS`. Para garantir que os dados da FIFO de transmissão fossem iniciados a partir do próximo `FS`, foi necessário utilizar o valor `"11110"`, que representa o canal 30, para o sistema começar a leitura da fila de transmissão um canal antes do `FS`, pois, foi verificado que o controlador TDM gasta um canal para preparar a transmissão.

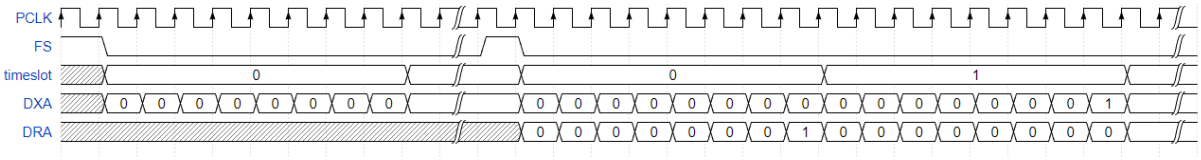
Tendo a máquina de estados para fazer o controle da leitura e escrita dos dados nas FIFOs, foi necessário inserir uma nova lógica no controlador FIFO, para fazer a inversão dos dados nos *time slots*, pois até então, os dados estavam alinhados ordenadamente, como apresentado na subseção 3.5.2. A lógica a ser implementada consiste em fazer com que os dados transmitidos (DXA) pelo ramal 1 no *time slot* 0, fossem recebidos (DRA) no ramal 2 no *time slot* 1, e assim, nos demais canais. O resultado esperado para essa lógica é apresentado na Figura 29.

Figura 28 – Diagrama ASM para representar o controle das FIFOs



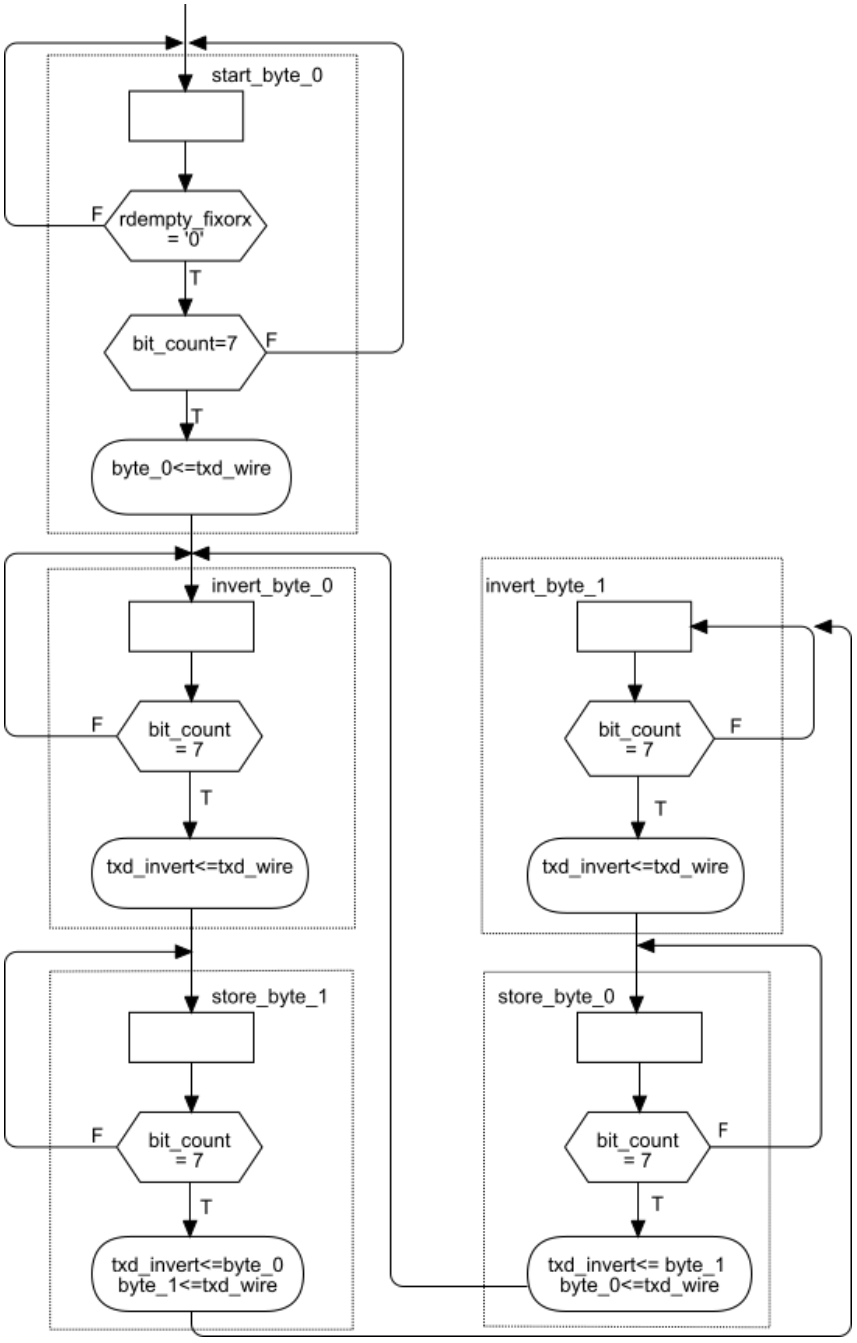
Fonte: Própria

Figura 29 – Resultado esperado dos canais invertidos



Fonte: Própria

Figura 30 – Diagrama ASM para representar a inversão dos canais (*time slots*)



Fonte: Própria

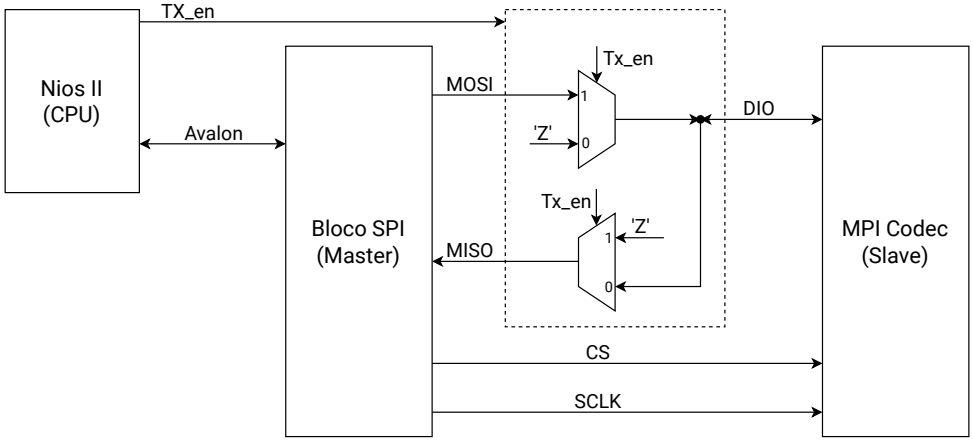
Para atingir o objetivo da inversão, com o fluxo de dados proposto, foi elaborada uma nova máquina de estados finita, em que sua lógica comportamental também foi representada em um diagrama ASM e que pode ser visto na Figura 30. Essa máquina de estado tem a finalidade de armazenar *bytes* e invertê-los em sequência. Na lógica implementada, o sinal `byte_0` e `byte_1` são utilizados para armazenar os dados a serem invertidos e o sinal `txd_invert` é a saída do controlador. De uma forma simplificada, todos os dados transferidos nos canais pares (0, 2, 4, ..., 30) foram armazenados para serem invertidos após o recebimento dos dados nos canais ímpares (1, 3, 5, ..., 31). O sinal de saída `txd_invert` (8 *bits*) é atribuído na entrada da FIFO de transmissão, fazendo com que os dados sejam armazenados com a lógica de inversão. Dessa forma, a partir do segundo `FS`, será possível atingir a inversão desejada.

3.6.1.2 Comunicação SPI

Para configuração e controle da placa de ramal, foi necessário criar uma interface entre o Nios II e o bloco MPI do *codec*. Analisando a interface MPI (subseção 3.2.1.1), percebeu-se que o padrão de comunicação exigido era similar ao SPI, com a diferença que o SPI utiliza comunicação *full duplex* (MOSI e MISO) e o MPI é *half duplex* (DIO). Dessa forma, para facilitar, foi utilizado um bloco SPI da biblioteca da Intel com uma lógica de multiplexação dos sinais MOSI e MISO para o sinal DIO.

Basicamente, a lógica consiste em controlar o momento que o processador envia e recebe dados da placa de ramal com o uso de multiplexadores. Na Figura 31 é apresentado a lógica criada, onde é possível observar, que o processador só vai conseguir enviar dados para a placa de ramal (*Slave*) quando o sinal `Tx_en` estiver em 1. Também é possível observar que o processador só vai conseguir receber dados da placa de ramal, quando o `Tx_en` estiver em 0.

Figura 31 – Lógica para comunicação serial entre placa de ramal e o processador



Fonte: Própria

Usando essa estratégia, depois da integração, foi possível alcançar rapidamente

o objetivo de se comunicar com o *codec*, uma vez que o bloco SPI da biblioteca Intel já possuía a interface com o barramento *Avalon* do Nios II e bibliotecas de *software* disponíveis.

3.6.2 Plataforma de *hardware*

Plaftorm Designer foi o *software* usado para fazer a integração do sistema e gerar a lógica de interconexão para conectar as funções de IP e os subsistemas. Essa interconexão foi realizada por meio do *Avalon Switch Fabric*, onde foram adicionados os componentes necessários para o sistema baseado em Nios II. As funções planejadas para serem utilizadas na configuração do sistema foram:

- *Clock*: Para o controle do sistema Nios, utiliza os 50 MHz fornecidos pelo *kit*;
- Processador Nios II: Responsável pelo processamento do sistema (CPU). Configurado com a opção Nios II/e;
- *On-Chip memory*: A *FPGA* disponibiliza blocos de memória que podem ser configurados para serem usados no processador. Para esse projeto foi necessário configurar a *On-Chip memory* com 122.800 bytes de memória para evitar o transbordo de dados;
- *Jtag Uart*: Responsável por fazer a interface com o computador através do *USB Blaster*;
- 3 Wire Serial (spi_master): Realiza a comunicação serial (SPI) com o DIO da placa de ramal;
- PIO (tx_en): Utilizado para controle da transmissão. Como a placa de ramal possui um pino DIO bidirecional que multiplexa os dados de entrada e saída, foi necessário controlar o momento da transmissão e da recepção;
- PIO (rst_qsys): Atribuído a porta de *reset* do sistema;
- PLL (atlpll): Utilizado para dividir o *clock* do sistema e gerar um *clock* de 2,048 MHz para ser usado em outros componentes e na placa de ramal;
- *Clock bridge*: Para ser possível exportar o *clock* gerado pelo PLL para o sistema.

Para fornecer esse *clock* de 2,048 MHz para o sistema, foi necessário configurar o PLL da plataforma usando um fator de multiplicação de *clock* do sistema de 128 e um fator de divisão de 3125, com um ciclo de trabalho de 50%, pois o *clock* do *kit* era de 50 MHz.

Adicionalmente, como a placa de ramal faz a transmissão dos dados usando 8 *bits*, necessitando de 8 pulsos de *clock* para transmitir um *byte* completo, foi necessário

adicionar outra frequência no PLL da plataforma para fornecer um *clock* de 8 kHz ao sistema. Essa adição tem a finalidade de fazer a sincronização de quadro dos 32 canais usados pela placa de ramal. Um receio que surgiu com a adição desse novo *clock*, foi se o Quartus conseguiria fazer a divisão do *clock* do sistema para chegar no valor desejado, pois, demandaria bastante recurso. Para evitar que isso acontecesse e também para não adicionar outro PLL, foi então, implementado uma entidade que gera um sinal de *strobe* a cada 256 pulsos do *clock* da placa. A porta lógica utilizada no projeto para fazer a sincronização dos quadros é o `FS`, o qual recebe o valor do *strobe*. O código usado para gerar o *strobe* pode ser visto no Código A.2 do apêndice e a configuração realizada no *Platform Designer* pode ser vista no Apêndice C.

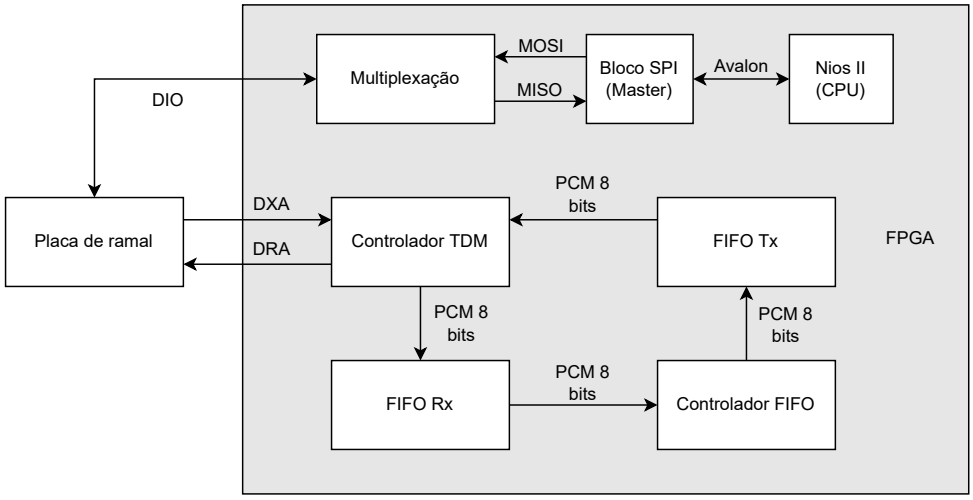
3.6.3 Integração

A fim de consolidar a implementação dos componentes lógicos e do *software*, foi necessário fazer a integração dessas entidades. Para isso, foi desenvolvido um projeto em Linguagem de Descrição de Hardware VHSIC (VHDL) utilizando o *software* Intel Quartus Prime 20.1, o qual permite que a lógica programável seja sintetizada na FPGA presente no *kit* de desenvolvimento. Para configuração do projeto foi utilizado o dispositivo EP4CE115F29C7 da família *Cyclone* IV E. Esse dispositivo possui um total de 114.000 elementos lógicos, 528 pinos de entrada e saída, memória embarcada de 3,888 Kbits e quatro *Phase-Locked Loop* (PLL) de uso geral (Altera Corporation, 2016).

Para a proposta, foi criado um arquivo VHDL definido como a entidade *top_level* do projeto (Código A.1 do apêndice), onde foi instanciado todas as entidades implementadas como componentes, assim como, instanciado o módulo gerado pelo *Platform Designer*. Na Figura 32 é apresentada uma visão em alto nível da integração realizada, e no Apêndice D é possível visualizar o RTL *Viewer* completo do projeto.

Com a integração finalizada, foi realizada a associação dos pinos da FPGA para as portas utilizadas no projeto, conforme mapeado no diagrama apresentado na seção 3.3. A associação no *software* Quartus foi realizada através da função *Pin Planner* e pode ser vista na Figura 33. Posteriormente, foi realizado a compilação, e a depuração do projeto para corrigir alguns alertas que surgiram. Ao fim da depuração, foi iniciado a gravação na FPGA utilizando o arquivo `.sof` gerado após a compilação. Com a lógica gravada na FPGA, o próximo processo foi utilizar o *software* Nios implementado para testar e validar a implementação. A integração física da placa de ramal utilizando a placa adaptadora no *kit* de desenvolvimento, pode ser vista na Figura 34.

Figura 32 – Diagrama de blocos simplificado do sistema proposto.



Fonte: Própria

Figura 33 – Associação de pinos na FPGA

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Current Strength
C2_SYS	Output	PIN_AE25	4	B4_N1	PIN_AE25	3.3-V LVTTL	8mA (default)
CLOCK_50	Input	PIN_Y2	2	B2_N0	PIN_Y2	3.3-V LVTTL	8mA (default)
CS	Output	PIN_AH26	4	B4_N0	PIN_AH26	3.3-V LVTTL	8mA (default)
DCLK	Output	PIN_AH23	4	B4_N1	PIN_AH23	3.3-V LVTTL	8mA (default)
DIO	Bidir	PIN_AH22	4	B4_N1	PIN_AH22	3.3-V LVTTL	8mA (default)
FS	Output	PIN_AE24	4	B4_N0	PIN_AE24	3.3-V LVTTL	8mA (default)
INT	Input	PIN_AE22	4	B4_N0	PIN_AE22	3.3-V LVTTL	8mA (default)
KEY0	Input	PIN_M23	6	B6_N2	PIN_M23	3.3-V LVTTL	8mA (default)
RST	Output	PIN_AF16	4	B4_N2	PIN_AF16	3.3-V LVTTL	8mA (default)
TDMI0	Input	PIN_AG23	4	B4_N1	PIN_AG23	3.3-V LVTTL	8mA (default)
TDMO0	Output	PIN_AD22	4	B4_N0	PIN_AD22	3.3-V LVTTL	8mA (default)

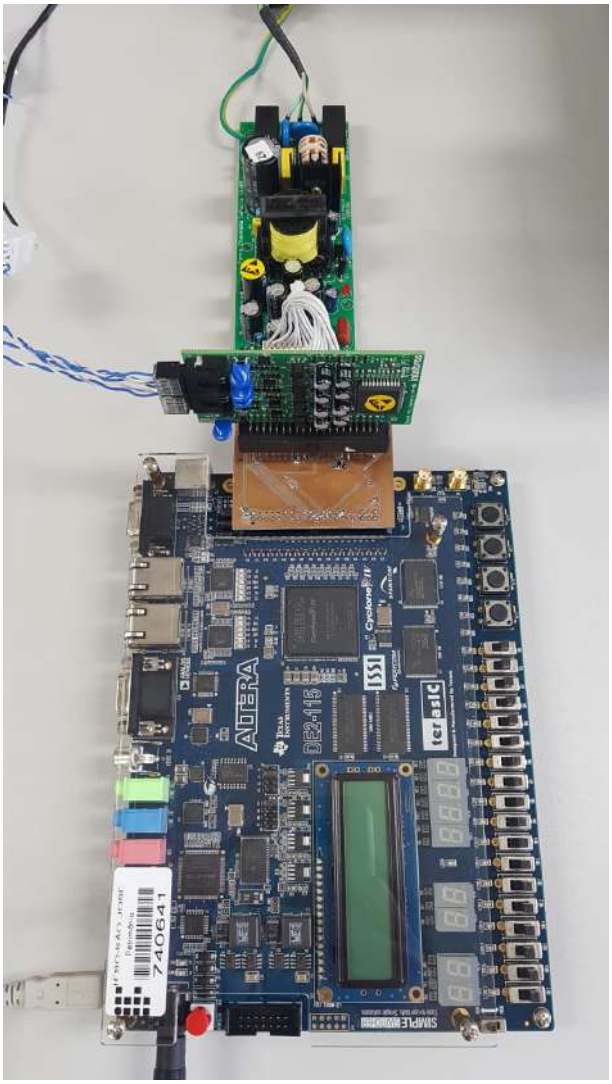
Fonte: Própria

3.6.4 *Software* Nios II

Para a implementação do *software* para controle da placa de ramal, foi utilizado a ferramenta de construção de *software* Nios II para suporte ao Eclipse, disponível no Quartus Prime. Essa ferramenta permite criar uma aplicação Nios II com uso do editor BSP a partir do arquivo `sopcinfo` gerado pelo *Platform Designer*, como exemplificado na subseção 2.2.2. Ao final desse processo, é gerado uma imagem do *software* que possui todas as rotinas e bibliotecas necessárias para o funcionamento na plataforma.

A Altera disponibiliza uma biblioteca chamada altera_avalon_spi.h, que fornece funções para controlar o barramento SPI. Dentre essas funções, foi utilizada a função `alt_avalon_spi_command()` para enviar *bits* para a porta DIO da placa de ramal pela porta MOSI e receber *bits* do DIO na porta MISO da SPI configurada na interconexão. Para utilizar essa função, foi necessário informar o endereço da memória do SPI mestre, a posição do escravo que recebe os dados, o tamanho dos dados transmitidos e as constantes que serão usadas, conforme pode ser observado no Código 3.1.

Figura 34 – Integração das placas com o *kit* de desenvolvimento



Fonte: Própria

Código 3.1 – Função SPI Avalon

```
1 int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
2                             alt_u32 write_length, const alt_u8 * write_data,
3                             alt_u32 read_length, alt_u8 * read_data,
4                             alt_u32 flags);
```

Outra biblioteca utilizada é a altera_avalon_pio_regs, que fornece acesso fácil de I/O a dispositivos externos. Dessa biblioteca, em especial, foi aproveitado a função `IOWR_ALTERA_AVALON_PIO_DATA()` para controle do sentido de transmissão da SPI, através do PIO `tx_en`, e para controlar o *reset* do sistema, pelo PIO `rst_qsys`, usado para inicializar os componentes. Além dessas duas bibliotecas mencionadas, para auxiliar no desenvolvimento do *software*, foi utilizado as bibliotecas apresentadas no Código 3.2

Código 3.2 – Bibliotecas do *software* Nios.

```
#include "alt_types.h"
#include "altera_avalon_spi_regs.h"
#include "altera_avalon_spi.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "io.h"
#include <stdio.h>
#include <unistd.h>
```

Após ter definido a forma de comunicação com a placa de ramal, foi necessário definir quais configurações deveriam ser feitas para inicializar e controlar o *codec* da placa de ramal. Com as análises realizadas na subseção 3.2.2, foi estipulado que as configurações mínimas necessárias para controlar o *codec* seriam:

1. Realizar o *hardware reset*;
2. Definir a frequência a ser utilizada no *master clock*;
3. Configurar o tempo de *debounce* entre as SLICs;
4. Configurar os *time slots* e *clock slots*;
5. Configurar os canais dos registradores a serem utilizados;
6. Configurar a direção dos SLICs
7. Configurar máscara de interrupção;
8. Definir os coeficientes dos filtros;
9. Definir a codificação a ser usada (lei A ou lei μ);
10. Ativar o *codec*.

A partir dessa configuração, foi esperado ser possível inicializar o *codec*, sincronizar o *master clock* com o *frame sync*, enviar comandos para os ramais (i.e., *ring*), receber informações dos ramais (i.e., estado do gancho). O código desenvolvido pode ser visto no Apêndice B ou no *GitHub*[8].

[8] <https://github.com/renaner123/TCC>

4 TESTES E RESULTADOS

Neste capítulo, serão apresentados os testes realizados para validar as entidades implementadas e o cenário proposto, e os resultados. Em suma, os testes executados foram os seguintes:

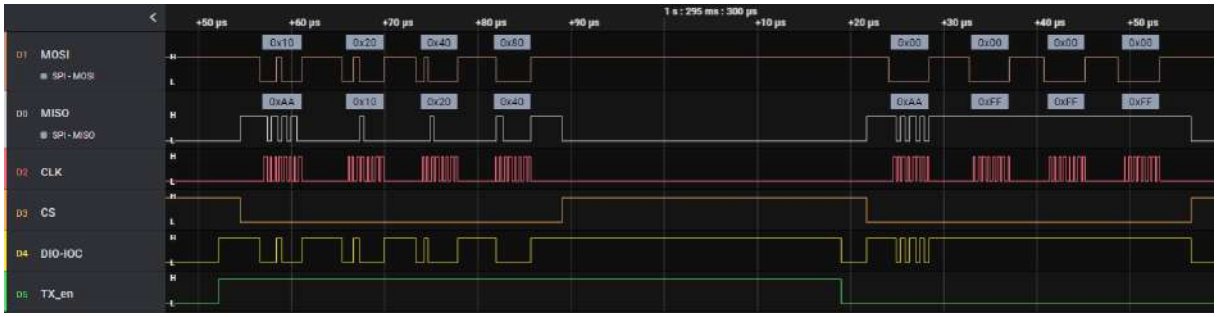
1. Validação da MPI: teste realizado para validar a comunicação entre a placa de ramal e o *kit* de desenvolvimento;
2. Validação das FIFOs: validar o funcionamento das FIFOs de transmissão e recepção;
3. Validação do controlador TDM: para verificar se o controlador TDM faz a multiplexação dos 32 canais corretamente;
4. Validação do controlador FIFO: confirmar se as máquinas de estados implementadas garantem o alinhamento dos canais;
5. Inicialização do *codec*: verificar se foi possível iniciar o *codec* e sincronizá-lo;
6. Validação do *hotline*: após validar todas as entidades, validar a função de *hotline* nos ramais.

4.1 Validação da MPI

Este teste tem por objetivo validar a comunicação serial que acontece entre a placa de ramal e o processador Nios II por meio da interconexão. Para efetuar essa validação, foi necessário testar a lógica implementada na subseção 3.6.1.2 para multiplexar os sinais MISO e MOSI da interconexão em um sinal bidirecional a ser enviado para o pino `DIO` da placa de ramal. Para auxiliar, foi adicionado uma SPI escrava na interconexão.

O teste consistiu em escrever quatro *bytes* no MOSI e tentar ler um *byte* no MISO. A escrita deve ocorrer somente quando o `tx_en` estiver em 1 e o `CS` em 0, e a leitura somente quando `tx_en` e `CS` estiverem em 0. A interrupção não foi validada nessa etapa, pois necessitava de um ramal conectado para fazer a notificação ao processador. Na Figura 35 é possível observar que o DIO recebeu os valores escritos no MOSI enquanto o `tx_en` estava em alto, e recebeu o valor do MISO corretamente quando `tx_en` estava em baixo, portanto, o resultado corresponde com o esperado.

Figura 35 – Teste de validação da MPI



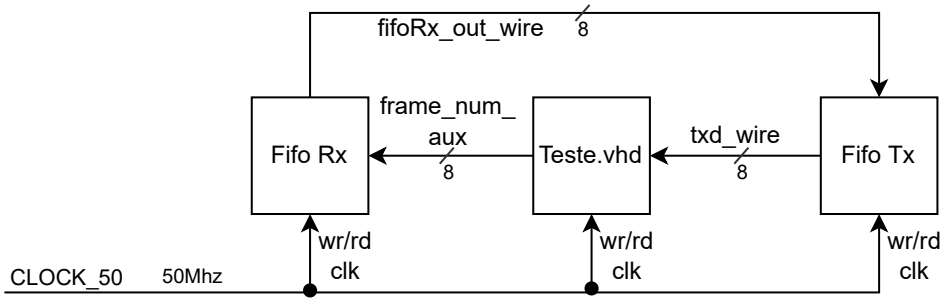
Fonte: Própria

4.2 Validação das FIFOs

A fim de validar o funcionamento de escrita e leitura da FIFO, foi criado um módulo VHDL que instancia duas entidades do componente FIFO, apresentado na subseção 3.5.1, uma para recepção e uma para transmissão, e rotinas de testes que controlam os sinais de entrada e saída. Esse processo de teste é conhecido por *testbench*. Para representar esse *testbench*, foi criado o diagrama de blocos apresentado na Figura 36, no qual indica as entidades e os principais sinais analisados.

Essa rotina de teste consiste em habilitar a escrita na FIFO de recepção (sinal `fifoRx_wrreq_control`), inserido valores aleatórios na entrada da FIFO (`frame_num_aux`) até que ela fique cheia (`fifoRx_wrfull_wire`). Em seguida, é habilitado a leitura (sinal `fifoRx_rdreq_control`) da FIFO para analisar a saída (`fifoRx_out_wire`).

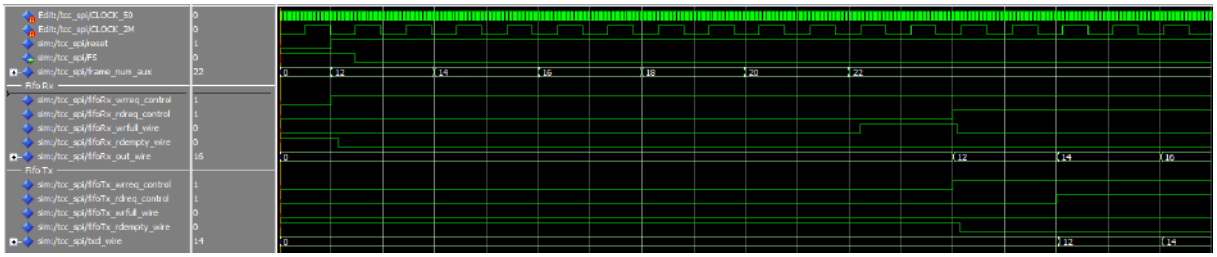
Figura 36 – Diagrama de blocos para validação das FIFOs



Fonte: Própria

Aproveitando a saída da FIFO de recepção, foi iniciado a escrita desses dados na FIFO de transmissão, em seguida, a leitura, para verificar se teria algum problema de sincronização entre os *clocks* de leitura e escrita usados, que para esse teste, foi de 50 MHz. Na Figura 37 é possível observar a simulação do *testbench*, realizada por meio do *software ModelSim*.

Figura 37 – Simulação das FIFOs em *chip* de memória



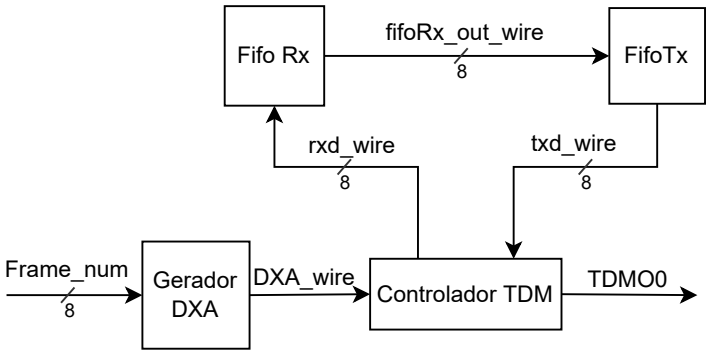
Fonte: Própria

4.3 Validação do controlador TDM

Para ser possível validar o controlador TDM, foi necessário implementar uma entidade (Gerador DXA) para gerar um fluxo PCM com as mesmas características que o sinal gerado pela placa de ramal. Essa entidade registra o *byte* de entrada `frame_num` e o serializa *bit* a *bit* a cada pulso do PCLK, gerando o sinal DXA. Com esse bloco e com a análise feita na subseção 3.5.2, foi possível iniciar a validação da multiplexação do controlador TDM.

O primeiro teste foi atribuir o sinal DXA gerado na entrada da interface serial, chamada de interface *Serial Telecom Bus* (ST-Bus), e verificar se o controlador gera o sinal PCM correspondente. O segundo teste foi validar o sentido contrário, fazendo o controlador receber um sinal PCM, e verificar se a saída da interface *ST-bus* era coerente com o sinal gerado. O diagrama de blocos desse *testbench* é apresentado na Figura 38, no qual indica as entidades e os principais sinais utilizados no *testbench*.

Figura 38 – Diagrama de blocos para validação do controlador TDM

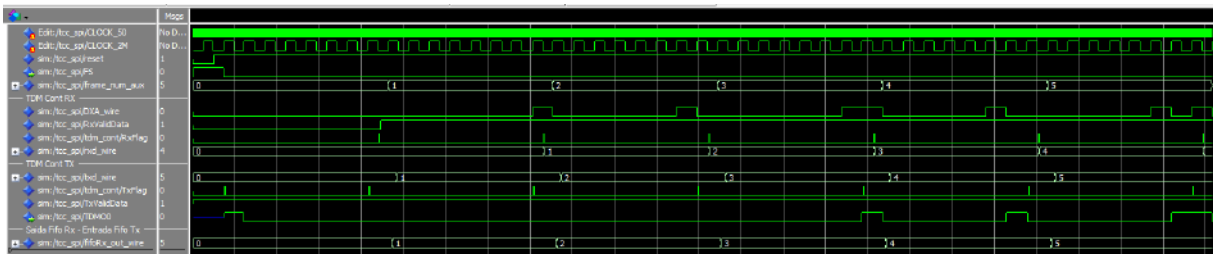


Fonte: Própria

Na Figura 39 é apresentado o resultado da simulação inicial, onde é possível observar que o controlador TDM, após receber um *byte* completo, indicado pelo sinal `RxFlag`, notifica o sistema que possui dados válidos para recepção, indicado pelo sinal `RxValidData`. A partir desse momento, o controlador atribui para saída `rxd_wire` o valor correspondente ao DXA. Nessa mesma simulação, também é possível verificar, que após o

controlador receber um sinal PCM no `txd_wire`, gera os dados correspondentes na saída serial, representada pelo sinal `TDMO0`, com um pequeno atraso.

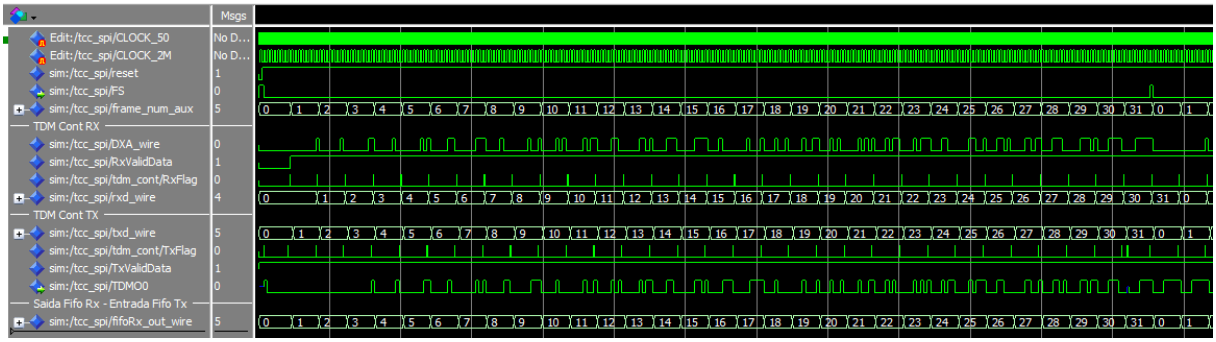
Figura 39 – Simulação das entradas e saídas seriais do controlador TDM



Fonte: Própria

Com base na simulação apresentada e na simulação da Figura 40, foi possível constatar que o controlador TDM atende as necessidades do projeto. Ele faz a multiplexação dos 32 canais PCM integrado com duas FIFOs externas, possibilita a comunicação serial de entrada e saída e possibilita o controle para iniciar a transmissão e recepção dos dados.

Figura 40 – Simulação da multiplexação do controlador TDM



Fonte: Própria

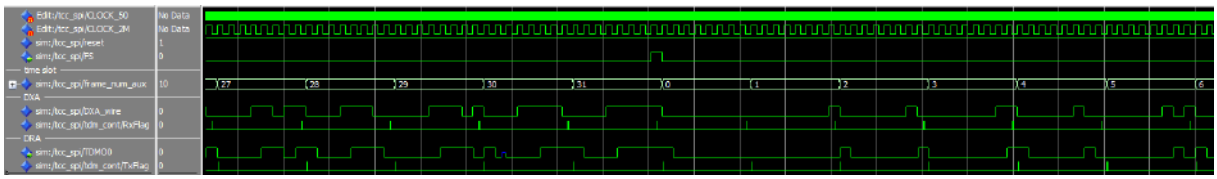
4.4 Validação do controlador FIFO

O objetivo deste teste foi verificar a lógica usada na máquina de estado implementada para controlar a leitura e escrita dos dados, assim como, validar a máquina de estado implementada para fazer a inversão dos dados no cenário proposto. A primeira validação foi verificar se a máquina de estado utilizada para fazer o controle das FIFOs, com o controlador TDM, torna a transmissão e recepção ordenada, conforme foi apresentado na subseção 3.5.2, no resultado esperado para os canais alinhados.

Para essa validação, foi utilizado como base o projeto usado na seção 4.3, com adição da entidade controlador FIFO entre as FIFOs, que tem a finalidade de fazer o controle da leitura e escrita dos dados nas filas. Após criar o *testbench* para validar esse

cenário, foi realizado a simulação apresentada parcialmente na Figura 41, onde pode-se observar o alinhamento da transmissão e recepção dos dados seriais após o segundo `FS`. Tanto o sinal DXA (`DXA_wire`), quanto o sinal DRA (`TDMO0`), estão iniciando com o valor do *time slot* 0. Com isso, foi possível validar, que o resultado da lógica implementada na máquina de estado está compatível com o comportamento do PCM *highway* apresentado na subseção 3.2.1.3.

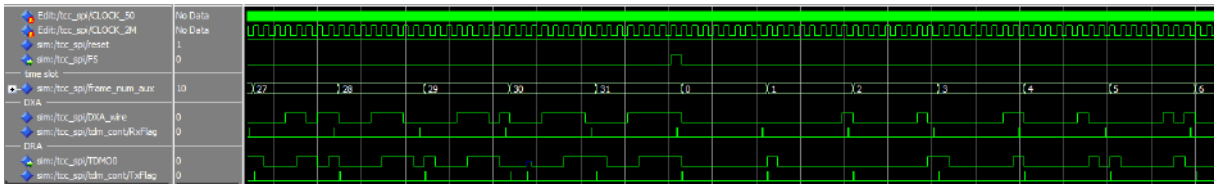
Figura 41 – Simulação dos canais alinhados



Fonte: Própria

Ainda no mesmo teste, foi adicionado a lógica da máquina de estado responsável por fazer a inversão dos dados transmitidos. Apenas relembrando, a inversão consiste em fazer com que os dados transmitidos (DXA) pelo ramal 1 no *time slot* 0, fossem recebidos (DRA) no ramal 2, após um `FS`, no *time slot* 1, e assim, nos demais canais. Após fazer a alteração no projeto, foi realizado uma nova simulação utilizando o mesmo *testbench*. O resultado dessa simulação pode ser visto na Figura 42. Nele foi possível verificar, que a lógica da inversão corresponde com o resultado esperado que foi apresentado na subseção 3.6.1.

Figura 42 – Simulação dos canais invertidos



Fonte: Própria

4.5 Inicialização do *codec*

Após validar as entidades implementadas, foi necessário testar o *software* de controle e configuração do *codec* rodando no processador sintetizado Nios II. Para isso, foi utilizado o código apresentado na subseção 3.6.4 e os arquivos VHDL do projeto, disponíveis no Apêndice A. O teste consistiu em verificar as configurações mínimas estipuladas na subseção 3.6.4. Após realizar o fluxo de desenvolvimento de projeto com *softcore* (veja subseção 2.2.2), foram iniciados os testes com uso da IDE *Eclipse*. A saída do terminal da IDE, após executar o *software*, pode ser vista no Código 4.1.

Código 4.1 – Saída via terminal da IDE da inicialização do *codec*

```
1  Entered Main
2  Enviando comando 04: para fazer um hardware reset
3  Enviando comando 46 e após la para configurar o master clock:
4  Enviando comando c8 e após 3c para configurar o debounce:
5  Enviando comando 4a e após 01 para selecionar canal 1:
6  Enviando comando 44 e após 40 para configurar time slot:
7  Enviando comando 55 para verificar o cfail
8  Valor recebido do codec: 20:
9  Enviando comando 47 para verificar o master clock configurado:
10 Valor recebido do codec: la:
11 Enviando comando 60 e após 00 para configurar os coeficientes
12 Enviando comando 61 para verificar coeficientes configurados
13 Valor recebido do codec: 00:
14 Enviando comando 4a e após 03 para configurar os registradores dos canais
15 Enviando comando 4b para verificar registradores dos canais
16 Valor recebido do codec: 03:
17 Enviando comando 54 e após 02 para configurar a direcao dos slics
18 Enviando comando 55 para verificar o CSIA e a direcao dos slics
19 Valor recebido do codec: 22:
20 Enviando comando 0e para ativar o codec
21 Enviando comando 55 para verificar se o codec ativou
22 Valor recebido codec: 42:
```

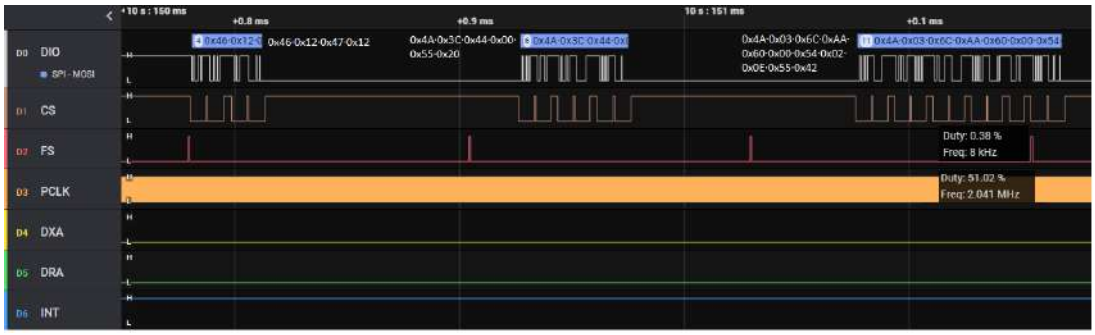
Na saída de *debug*, é possível verificar que os comandos enviados foram aceitos pelo *codec*. Por exemplo, ao enviar o comando para definir o *master clock* `(c:0x46, v:0x1A)` (linha 3) e após usar o comando de leitura do *master clock* `(c:0x47, v:-)` (linha 9), o *codec* retorna o valor configurado (`0x1A`) (linha 10). Ao fim da execução, ao utilizar o comando de leitura da SLIC `(c:0x55, v:-)` (linha 21), o *codec* retornou o valor `0x42` (linha 22), indicando que os canais estão ativos e que o *master clock* está alinhado com o `FS`. Com isso, foi constatado que foi possível fazer a inicialização e configuração do *codec* na placa de ramal.

4.6 Validação do *hotline*

Com os testes de validação dos componentes e *softwares* utilizados e implementados no projeto, foi iniciado a validação do cenário proposto. Para o controle da placa ramal, foi utilizado como base o *software* utilizado na seção 4.5. Inicialmente foi utilizado as configurações mínimas, apresentadas anteriormente, para fazer a inicialização do *codec*. Na Figura 43 é apresentado a análise lógica dessa inicialização. Nela é possível observar que foi utilizado o `PCLK` como *master clock* usando um *clock* de 2,048 MHz (c:0x46, v:0x12), transmite dados na borda positiva do `PCLK` (c:0x44, v:0x00), que as mudanças que acontecerem nos canais devem gerar interrupções (c:0x6C, v:0xAA) e configurado

os coeficientes padrões (c:0x60, v:0x00). Também é possível observar nessa figura, que a lógica utilizada para gerar o sinal de sincronismo (FS), assim como, a saída do PLL, estão conforme o esperado. Nessa mesma etapa, foi realizado a inicialização dos dois primeiros canais do *codec*, sendo o canal 1 e o canal 2, usando os valores padrões dos coeficientes e dos ganhos.

Figura 43 – Inicialização do *codec* no cenário proposto



Fonte: Própria

Após a inicialização do *codec* e dos ramais, foi adicionado no *software* a lógica para monitorar o estado dos ramais (gancho) para realizar uma chamada assim que identificar um ramal fora do gancho. Na configuração foi definido que quando o ramal 2 estiver fora do gancho, é habilitado o *ring* no ramal 1. Para isso, assim que ocorre a interrupção indicando que o ramal 2 está fora do gancho (c:0x4F, v:0xAE), é realizado para o ramal 1 e 2 a configuração dos coeficientes padrões (c:0x60, v:0x3F), ganhos analógicos (c:0x50, v:0x00) e a direção da SLIC (c:0x54, v:0x02). Ainda na configuração, foi definido que o ramal 1 transmite (c:0x40, v:0x00) e recebe (c:0x42, v:0x00) dados no *time slot* 0 e o ramal 2 transmite (c:0x40, v:0x01) e recebe (c:0x42, v:0x01) dados no *time slot* 1. Posteriormente, é iniciado o laço de repetição para habilitar (c:0x52, v:0x1C) e desabilitar (c:0x52, v:0x1E) o *ring* no ramal 1. A análise lógica desse processo pode ser visto na Figura 44.

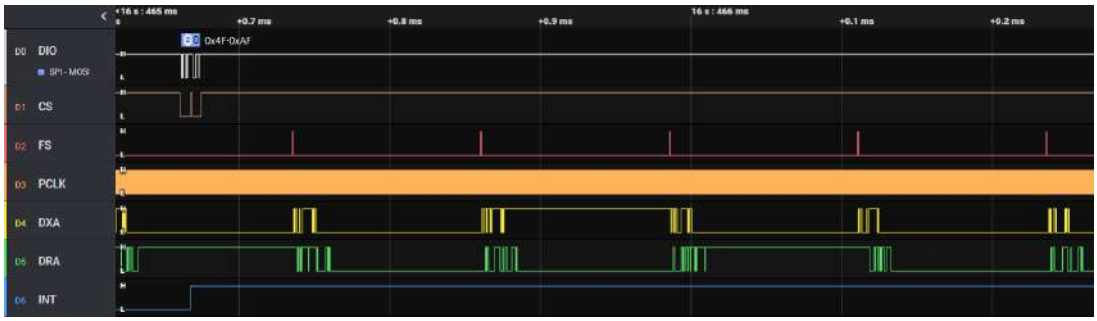
Figura 44 – Chamada sendo realizada no cenário proposto

Fonte: Própria

Assim que o ramal 1 realiza o atendimento, verificado pela interrupção ou pelo comando 0x4F com valor 0xAF, foi constatado que a voz transmitida pelo ramal 1 não chega no ramal 2, foi possível somente ouvir assopros/chiados no próprio ramal que estava tentando transmitir. Na análise lógica apresentada na Figura 45, é possível verificar que os

sinais transmitidos (DXA) no *time slot* 0 que deveriam ser recebidos (DRA) no próximo `FS` no *time slot* 1, não estão da forma esperada.

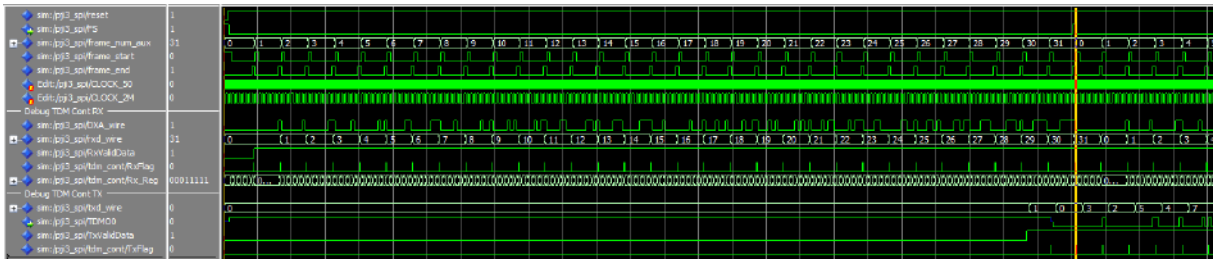
Figura 45 – Chamada estabelecida no cenário proposto



Fonte: Própria

Para garantir que a implementação da máquina de estado, responsável por fazer a sincronização da transmissão e recepção na função de *hotline*, estava correta, foi projetado um novo *testbench* para validar esse comportamento. A simulação deste *testbench* pode ser vista na Figura 46, onde é possível verificar que no primeiro `FS` não é transmitido nada (sinal TDMO0) e assim que ocorre o segundo `FS`, o valor recebido no sinal DXA_wire no *time slot* 0 (no primeiro FS) é recebido no TDMO0 no *time slot* 1 (no segundo FS) e assim por diante.

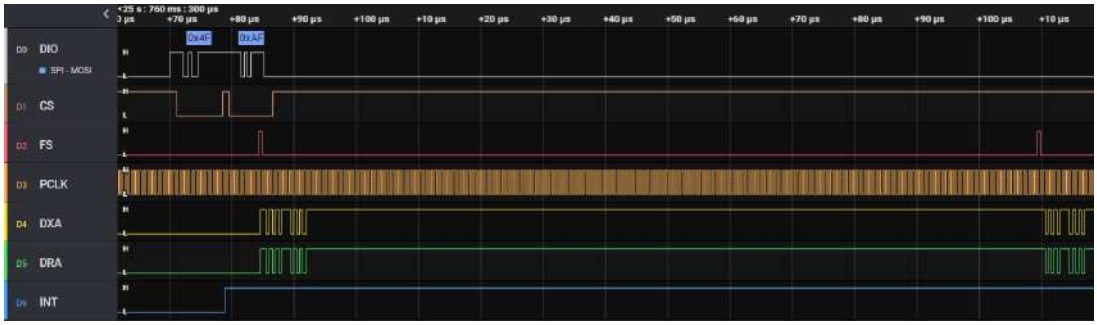
Figura 46 – Simulação da inversão de canais para função *hotline*



Fonte: Própria

Também para garantir o funcionamento do *software* Nios implementado para controle da placa de ramal, foi realizado um novo teste, onde foi retirado todas as entidades implementadas para fazer com que os dados transmitidos pela placa de ramal voltem diretamente para ela sem passar por nenhum componente. Esse processo é chamado de *loopback* e deve fazer com que os dados transmitidos no *time slot* 0 sejam recebidos de imediato no próprio *time slot* 0. Ao realizar esse teste, foi possível verificar que a configuração implementada no *software* para iniciar o *codec* e os canais estão funcionando, pois, foi possível ouvir de forma clara e distinta o assopro do ramal, mas não a voz. Na Figura 47 é possível observar que os sinais transmitidos (DXA) são praticamente os mesmos recebidos (DRA) nos dois primeiros *time slots*.

Figura 47 – *Loopback* para validar a implementação do *software*



Fonte: Própria

Quadro 3 – Coeficientes de filtros da placa de ramal

Coeficiente	**Comando (Hex)**	**Valor (Hex)**
Gx	80	88 e 78
Gr	82	A2 e A0
X	88	CA,E0,2A,A5,AA,AD,3A,8F,32,A5,AA e 3E
R	8A	3A,C0,2B,70,23,BE,2D,36,AB,AD,AB,B6,5C e BF
Z	84	AA,A2,A3,4C,BA,AB,C3,AE,AB,A4,34,D1,66,9F e 01
B1	86	22,73,34,2A,32,A5,D8,FE,87,D8,7F,87,A8 e F0
B2	95	2E e 01

Fonte: Própria

A fim de tentar melhorar o processamento dos sinais digitais transmitidos no *loopback*, foi alterado os valores dos coeficientes de filtro, que estavam no padrão, para os valores de coeficientes analisados na subseção 3.2.2, onde foram identificados os comandos e valores utilizados nas configurações dos coeficientes de filtros, conforme apresentado no Quadro 3. Após fazer essa configuração, foi realizado o teste com o *loopback* novamente e não foi obtido uma melhora na qualidade do áudio transmitido e recebido e nem estabelecido a comunicação PCM.

5 CONCLUSÃO

Esse trabalho teve como objetivo fazer a integração de uma placa de ramais analógicos, usada em centrais telefônicas, a um *kit* de desenvolvimento FPGA. O trabalho envolveu o uso de processadores sintetizados (*softcores*), desenvolvimento de *software* embarcado para o mesmo, uso e criação de blocos de *hardware* e criação de um cenário mínimo para experimentação com os canais digitais.

A metodologia utilizada permitiu que fosse levantado informações referente ao funcionamento, comportamento e configuração, tanto de *hardware* quanto de *software* da placa de ramal da central Impacta 16. Possibilitando a criação de um diagrama para representá-la, assim como, possibilitou elaborar um diagrama para criação de uma placa adaptadora, a qual foi utilizada para fazer a conexão da placa de ramal e da sua fonte de alimentação ao *kit* de desenvolvimento.

Com os diagramas criados, foi iniciado o processo de geração da PCB com uso do *software* Eagle, onde foi desenvolvido o esquemático com base no diagrama da placa adaptadora. Após ter o esquemático validado, foi iniciado o processo de geração da PCB. Antes de enviar o projeto da PCB a manufatura, foram realizados alguns testes e ajustes para obter um melhor resultado na produção. Após a finalização da produção da placa adaptadora, foram feitas análises utilizando um analisador lógico para embasar o desenvolvimento.

Após ter a placa produzida, foi iniciado o desenvolvimento da lógica programável a ser gravada na FPGA. Para atingir o objetivo proposto, foi necessário usar o *software* *Platform Designer*, disponível no *software* Quartus Prime, para gerar a lógica de interconexão para estabelecer a comunicação entre os subsistemas e as funções da biblioteca IP. Nessa etapa, foi necessário encontrar e estudar um controlador TDM pronto de forma a facilitar a integração.

Para fazer a integração da proposta, foi necessário gerar um projeto utilizando o *software* Quartus Prime. Instanciando as entidades implementadas e também o controlador TDM escolhido. Após ter a integração finalizada, foi proposto um cenário para poder validar essa integração. O cenário proposto consistiu em fazer uma função de *hotline* entre dois ramais, ou seja, quando um ramal é retirado do gancho, deve gerar uma chamada para outro ramal, e ao atender, deve ser possível estabelecer a comunicação.

Antes de ser iniciado os testes do cenário proposto, foram feitas validações nas entidades e *softwares* desenvolvidos, para verificar se o comportamento corresponde ao esperado. Nos testes, foram validados o processo de leitura e escrita das FIFOs, a comunicação SPI entre a placa de ramal e o processador, a comunicação serial gerada pelo

controlador TDM, a lógica das máquinas de estado criadas para ordenar e organizar os canais e a inicialização do *codec* da placa de ramal. Todas essas validações obtiveram o resultado esperado, conforme as informações que haviam sido levantadas nas análises lógicas e nos documentos de referência.

Com o *software* e as entidades validadas, foi então iniciado o teste do *hotline*. No teste, foi possível estabelecer a função de *hotline* entre os ramais, mas, não foi possível estabelecer a comunicação PCM entre os ramais. Para validar se o problema era da lógica implementada ou algo relacionado ao *hardware*, foi realizado um teste de *loopback*, onde a saída serial da placa de ramal foi conectada diretamente na entrada da placa de ramal, retirando toda a lógica desenvolvida. Dessa forma, também não foi possível estabelecer a comunicação.

Tendo como base os resultados discutidos, conclui-se que toda a infraestrutura de *hardware* e *software* para cumprir os objetivos propostos foram executados. Entretanto, devido a problemas relacionados, possivelmente, por questões de configuração do *codec*, não foi possível verificar por completo o funcionamento do cenário proposto, apesar de as simulações estarem de acordo com o esperado. Sendo assim, o refinamento do cenário e novos testes ficarão como sugestão para trabalhos futuros.

5.1 Trabalhos futuros

Esse trabalho abre várias possibilidades de continuidade, sendo elas:

- Utilizar FIFO *Avalon* entre o controlador TDM e o Nios II para permitir uma efetiva interface entre os dois domínios de *clock*;
- Analisar em mais detalhes as configurações do *codec* e da placa de ramal para identificar o motivo pelo qual o sinal PCM não está sendo transmitido de um ramal para outro;
- Fazer o processamento de sinais PCM nos canais da plataforma;
- Fazer a transcodificação dos sinais PCM para o codec G.723, por exemplo;
- Melhorar o *software* para controle da placa de ramal;
- Estudar a compatibilidade de outros controladores TDM.

REFERÊNCIAS

Altera Corporation. *1. Cyclone IV FPGA Device Family Overview*. 2016. Disponível em: <https://br.mouser.com/datasheet/2/612/cyiv-51001-1299459.pdf>. Citado na página 56.

ANEMAET, P.; AS, T. Microprocessor soft-cores: An evaluation of design methods and concepts on fpgas. *part of the Computer Architecture (Special Topics) course ET4078, Department of Computer Engineering*, Citeseer, 2003. Disponível em: <https://pretopia.net/files/paper_softcores.pdf>. Citado na página 22.

AOKI, N. Lossless steganography for speech communications. *Recent Advances in Steganography*, p. 75, 2012. Disponível em: <https://www.intechopen.com/chapters/40671>. Citado na página 28.

BORISOV, P.; VALENTINA, M.; KUKENSKA, S. Implementation of soft-core processors in fpgas. *INTERNATIONAL SCIENTIFIC CONFERENCE*, 2007. Disponível em: <http://kst.tugab.bg/staff/vally/6.pdf>. Citado 2 vezes nas páginas 23 e 24.

Cadence PCB solutions. *What is Hardware Software Co-design and How Can it Benefit You or Your Business?* 2019. Disponível em: <https://resources.pcb.cadence.com/blog/2019-what-is-hardware-software-co-design-and-how-can-it-benefit-you-or-your-business>. Citado na página 21.

CHU, P. P. Embedded sopc design with nios ii processor and vhdl examples. *Embedded SoPC Design with Nios II Processor and VHDL Examples*, John Wiley and Sons, 9 2011. Disponível em: <https://www.engineeringvillage.com/share/document.url?mid=cpx_4c1dc5c7164c88cae3fM40001017816339&database=cpx&view=detailed>. Citado 5 vezes nas páginas 21, 22, 23, 24 e 25.

Cisco Systems. *Técnicas de codificação de forma de onda*. 2006. Disponível em: <https://www.cisco.com/c/pt_br/support/docs/voice/h323/8123-waveform-coding.html>. Citado na página 28.

COFER, R.; HARDING, B. F. *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Elsevier, 2006. Disponível em: <https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=166447&lang=pt-br&site=ehost-live>. Citado 2 vezes nas páginas 18 e 20.

COSTA, C. d. *Projetos de circuitos digitais com FPGA*. [S.l.: s.n.], 2009. Citado 3 vezes nas páginas 18, 19 e 20.

DARWISH, T.; BAYOUMI, M. 5 - trends in low-power vlsi design. In: CHEN, W.-K. (Ed.). *The Electrical Engineering Handbook*. Burlington: Academic Press, 2005. p. 263–280. ISBN 978-0-12-170960-0. Disponível em: <https://www.sciencedirect.com/topics/engineering/hardware-software-codesign>. Citado 2 vezes nas páginas 22 e 23.

FLOYD, T. *Sistemas digitais: fundamentos e aplicações*. [S.l.]: Bookman Editora, 2009. Citado na página 19.

GONZáLEZ, D. et al. A low cost matching motion estimation sensor based on the nios ii microprocessor. *Sensors (Switzerland)*, v. 12, p. 13126–13149, 10 2012. ISSN 14248220. Disponível em: <https://www.mdpi.com/1424-8220/12/10/13126/htm>. Citado na página 16.

HICSONMEZ, S.; SENCAR, H. T.; AVCIBAS, I. Audio codec identification from coded and transcoded audios. *Digital Signal Processing*, Elsevier, v. 23, n. 5, p. 1720–1730, 2013. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S1051200413000845>. Citado na página 26.

HORN, R. A.; MOIA, J. Artigo disponibilizado on-line revista ilha digital atualizaÇÃo tecnolÓgica e resultados de testes de compatibilidade eletromagnÉtica para homologaÇÃo de uma central de comunicaÇÃo pabx. *Revista Ilha Digital*, v. 0, p. 1–8, 2017. Disponível em: <https://repositorio.ifsc.edu.br/bitstream/handle/123456789/1988/Artigo_Ricardo_Horn.pdf?sequence=2&isAllowed=y>. Citado na página 28.

Intel Corporation. *Welcome to the Quartus II Software*. 2005. Disponível em: <https://www.intel.com>. Citado na página 21.

KARRIS, S. T. *Digital circuit analysis and design with Simulink modeling and introduction to CPLDs and FPGAs*. [S.l.]: Orchard Publications, 2007. Citado na página 19.

KHATIB, J. *TDM Controller core*. 2001. Disponível em: <https://github.com/freecores/tdm/blob/master/docs/tdm_project.pdf>. Citado na página 47.

LEENS, F. *An introduction to I2C and SPI protocols*. 2009. 8-13 p. Disponível em: <https://ieeexplore.ieee.org/document/4762946>. Citado 2 vezes nas páginas 29 e 30.

Legerity. *Le58QL02/021/031 Quad Low Voltage Subscriber Line Audio-Processing Circuit VE580 Series*. 2006. Disponível em: <https://datasheetspdf.com/pdf-file/830493/Legerity/Le58QL02/1>. Citado 5 vezes nas páginas 34, 35, 36, 37 e 38.

MARTINS, A. E. A. et al. *Codificadores de Voz*. 2010. Disponível em: <https://www.gta.ufrj.br/grad/10_1/codec/Tecnicas_de_Codificacao.html>. Citado 2 vezes nas páginas 26 e 27.

MICHELI, G. D.; GUPTA, R. K. Hardware/software co-design. *Proceedings of the IEEE*, Institute of Electrical and Electronics Engineers Inc., v. 85, p. 349–365, 1997. ISSN 00189219. Disponível em: <https://ieeexplore.ieee.org/abstract/document/558708>. Citado na página 23.

MOECKE, M. *PCM - Modulação por Código de Pulso*. 2006. Disponível em: <http://tele.sj.ifsc.edu.br/~fabiosouza/Tecnologo/Telefonia%201/Apostila%20Modulacao%20PCM%20v2006.pdf>. Citado na página 27.

OUDJIDA, A. et al. *FPGA implementation of I2C & SPI protocols: A comparative study*. 2009. 507-510 p. Disponível em: <https://ieeexplore.ieee.org/abstract/document/5410881>. Citado na página 29.

PAN, D. Y. Digital audio compression. *Digital Technical Journal*, v. 5, n. 2, p. 28–40, 1993. Disponível em: <https://www.ee.columbia.edu/~dpwe/papers/Pan93-acomp.pdf>. Citado na página 26.

SHINGARE, T. D.; PATIL, R. *SPI implementation on FPGA*. Citeseer, 2013. 7–9 p. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=660e2932d0ddbf2a188c442ad81e4551c6bd7ff9>. Citado na página 29.

TINDER, R. F. *Engineering Digital Design: Revised Second Edition*. Elsevier, 2000. Disponível em: <https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=92149&lang=pt-br&site=ehost-live>. Citado na página 19.

TONG, J. G.; ANDERSON, I. D.; KHALID, M. A. Soft-core processors for embedded systems. In: IEEE. *2006 International Conference on Microelectronics*. 2006. p. 170–173. Disponível em: <https://ieeexplore.ieee.org/document/4243676>. Citado 2 vezes nas páginas 23 e 24.

TONG, J. G.; ANDERSON, I. D. L. Soft-core processors for embedded systems. *Research Center for Integrated MicrosystemsDepartment of Electrical and Computer EngineeringUniversity of WindsorSoftSoft–Core ProcessorsCore ProcessorsFor Embedded SystemsFor Embedded Systems*, p. 1–38, 2006. Disponível em: <https://www.vlsi.uwindsor.ca/presentations/2007/2-Soft%20Soft-Core%20Processors%20Core%20Processors_khalid.pdf>. Citado na página 16.

WEBER, A. F. et al. Arquitetura fpgas e cplds da altera. 2016. Disponível em: <https://wiki.sj.ifsc.edu.br/images/2/2a/DLP29006-AE1-Tema1-2016-1.pdf>. Citado na página 19.

WILSON, P. *Design recipes for FPGAs: using Verilog and VHDL*. [S.l.]: Newnes, 2015. Citado na página 20.

WOLF, W. H. Hardware–software co-design of embedded systems. *Proceedings of the IEEE*, v. 82, p. 967–989, 1994. ISSN 15582256. Disponível em: <https://ieeexplore.ieee.org/abstract/document/293155>. Citado na página 22.

Apêndices

APÊNDICE A – CÓDIGOS DO PROJETO QUARTUS

As listagens a seguir apresentam os códigos VHDL (entidades) utilizados nessa proposta. O Código A.1 apresenta a instanciação de todas as entidades como componentes, utilizadas para serem gravados na FPGA. Código A.2 representa a entidade implementada para gerar o sinal para sincronizar os quadros (gerar o *strobe*). No Código A.3, apresenta o código desenvolvido para fazer o controle e identificação dos canais usados, e o Código A.4 apresenta a entidade que foi implementada a máquina de estado para controlar a leitura e escrita dos dados e a inversão dos dados transmitidos e recebidos.

Código A.1 – Entidade *top level*

```
1  --Author: Renan Rodolfo da Silva
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.std_logic_arith.all;
6
7  library work;
8
9  entity pji3_spi is
10   port (
11     CLOCK_50    : in std_logic;
12     KEY0        : in std_logic;      --reset
13     DIO         : inout std_logic;   --IOC
14     DCLK        : out std_logic;     --CLK
15     FS          : out std_logic;     --F0_SYS
16     C2_SYS      : out std_logic;     -- C2_SYS/PCLK
17     CS          : out std_logic;     --CSRAMAL
18     INT         : in std_logic;      --INTCDC
19     TDMI0       : in std_logic;      --DXA
20     TDMO0       : out std_logic      --DRA
21   );
22
23 end pji3_spi;
24
```

```
25 architecture system of pji3_spi is
26
27   component DE2_115_SOPC_bridge_pll is
28     port (
29       clk_clk       : in std_logic := 'X';  -- clk
30       clk_2m_clk    : out std_logic;      -- clk
31       reset_reset_n     : in std_logic := 'X';  -- reset_n
32       spi_master_external_MISO  : in std_logic := 'X';  -- MISO
33       spi_master_external_MOSI  : out std_logic;      -- MOSI
34       spi_master_external_SCLK  : out std_logic;      -- SCLK
35       spi_master_external_SS_n  : out std_logic;      -- SS_n
36       tx_en_export    : out std_logic;      -- export
37       rst_qsys_export     : out std_logic       -- export
38     );
39   end component DE2_115_SOPC_bridge_pll;
40
41   --usado para gerar o sinal de strobe
42   component frame_sync is
43     port (
44       pclk    : in std_logic;
45       reset   : in std_logic;
46       strobe  : out std_logic
47     );
48   end component frame_sync;
49
50   --pcm_ctrl e pcm_tx_tb sao usados para gerar a saida DXA
51   component pcm_ctrl is
52     port (
53       reset, Pclk, fs       : in std_logic;
54       frame_start, frame_end  : out std_logic;
55       frame_num           : out std_logic_vector(4 downto 0)
56     );
57   end component;
58
59   component pcm_tx_tb is
60     port (
61       frame_start, frame_end, fs, Pclk, reset : in std_logic;
62       frame_num       : in std_logic_vector(7 downto 0);
63       DXA           : out std_logic
```

```
64    );
65  end component;
66
67  component fiforx is
68    port (
69      data    : in std_logic_vector (7 downto 0);
70      rdclk   : in std_logic;
71      rdreq   : in std_logic;   -- 1 lê o bit da fifo, 0 não lê
72      wrclk   : in std_logic;
73      wrreq   : in std_logic;     -- 1 escreve na fifo, 0 não escreve
74      q       : out std_logic_vector (7 downto 0);
75      rdempty   : out std_logic;    -- 0 tem dados na fifo
76      wrfull  : out std_logic     -- 1 fifo está cheia
77    );
78  end component fiforx;
79
80  component fifotx is
81    port (
82      data    : in std_logic_vector (7 downto 0);
83      rdclk   : in std_logic;
84      rdreq   : in std_logic;   -- 1 lê o bit da fifo, 0 não lê
85      wrclk   : in std_logic;
86      wrreq   : in std_logic;   -- 1 escreve na fifo, 0 não escreve
87      q       : out std_logic_vector (7 downto 0);
88      rdempty   : out std_logic;-- 0 tem dados na fifo
89      wrfull  : out std_logic   -- 1 fifo está cheia
90    );
91  end component fifotx;
92
93  component fifo_controller is
94    port (
95      pclk        : in std_logic;
96      FS, reset     : in std_logic;
97      --fifo rx
98      rdreq_fiforx  : out std_logic;  -- 1 lê o bit da fifo, 0 não lê
99      TxValidData   : out std_logic;
100     frame_num     : in std_logic_vector(4 downto 0);
101     RxValidData   : in std_logic;
102     wrreq_fiforx  : out std_logic;  -- 1 escreve na fifo, 0 não
```

```
    escreve
103       rdempty_fixorx : in std_logic;  -- 0 tem dados na fifo
104       wrfull_fiforx   : in std_logic;   -- 1 fifo está cheia
105       --fifo tx
106       rdreq_fifotx  : out std_logic;  -- 1 lê o bit da fifo, 0 não lê
107       wrreq_fifotx  : out std_logic;  -- 1 escreve na fifo, 0 não
    escreve
108       rdempty_fixotx : in std_logic;  -- 0 tem dados na fifo
109       txd_wire    : in std_logic_vector(7 downto 0);
110       txd_invert    : out std_logic_vector(7 downto 0);
111       tx_write    : out std_logic;
112       wrfull_fifotx   : in std_logic  -- 1 fifo está cheia
113     );
114   end component fifo_controller;
115   component tdm_cont_ent is
116     port (
117       rst_n   : in std_logic; -- System asynchronous reset
118       C2    : in std_logic; -- ST-Bus clock
119       DSTi    : in std_logic; -- ST-Bus input Data
120       DSTo    : out std_logic; -- ST-Bus output Data
121       F0_n    : in std_logic; -- St-Bus Framing pulse
122       F0od_n  : out std_logic; -- ST-Bus Delayed Framing pulse
123       CLK_I   : in std_logic; -- System clock
124
125       --Backend interface
126       NoChannels    : in std_logic_vector(4 downto 0); -- no of Time
    slots
127       DropChannels  : in std_logic_vector(4 downto 0); -- No of
    channels to be dropped
128
129       RxD         : out std_logic_vector(7 downto 0); -- Parellel Rx
    output
130       RxValidData   : out std_logic; -- Valid Data
131       FramErr       : out std_logic; -- Frame Error due to
132       -- buffer overflow
133       RxRead      : in std_logic; -- Read Byte
134       RxRdy       : out std_logic; -- Byte ready
135       TxErr       : out std_logic;
136       -- Tx Error in transmission due to buffer underflow
```

```
137       TxD          : in std_logic_vector(7 downto 0); -- Parellal Tx
      Input
138       TxValidData    : in std_logic; -- Tx Valid Data
139       TxWrite        : in std_logic; -- Write byte
140       TxRdy        : out std_logic; -- Byte Ready
141
142       -- Serial Interfaces
143       EnableSerialIF : in std_logic; -- Enable Serial Interface
144
145       Tx_en0       : out std_logic; -- Tx enable channel 0
146       Tx_en1       : out std_logic; -- Tx enable channel 1
147       Tx_en2       : out std_logic; -- Tx enable channel 2
148
149       Rx_en0       : out std_logic; -- Rx enable channel 0
150       Rx_en1       : out std_logic; -- Rx enable channel 1
151       Rx_en2       : out std_logic; -- Rx enable channel 2
152       SerDo        : out std_logic; -- serial Data out
153       SerDi        : in std_logic   -- Serial Data in
154
155     );
156   end component tdm_cont_ent;
157
158   signal MISO_m          : std_logic;
159   signal MOSI_m          : std_logic;
160   signal SCLK            : std_logic;
161   signal SS_n            : std_logic;
162   signal TX_en         : std_logic;
163   signal rst_qsys        : std_logic;
164   signal fs_wire         : std_logic;
165   signal clk_2M          : std_logic;
166
167   signal SerDo_wire        : std_logic;
168   signal SerDi_wire        : std_logic;
169   signal DSTo_wire       : std_logic;
170   signal DSTi_wire       : std_logic;
171   -- sinais tdm cont
172   signal FOod_n_wire     : std_logic;
173   signal reset         : std_logic;
174   signal DXA_wire        : std_logic;
```

```
175   signal RxRdy_view_wire  : std_logic;
176   signal TxRdy_view_wire  : std_logic;
177   signal TDMI0_wire       : std_logic;
178   signal TDMO0_wire       : std_logic;
179   signal rxd_wire       : std_logic_vector(7 downto 0);
180   signal txd_wire       : std_logic_vector(7 downto 0);
181   -- Sinais pra gerar o DXA
182   signal frame_start, frame_end   : std_logic;
183   signal frame_num          : std_logic_vector(4 downto 0);
184   signal frame_num_aux        : std_logic_vector(7 downto 0);
185   -- sinais fixo rx
186   signal fifoRx_rdempty_wire    : std_logic;
187   signal fifoRx_wrfull_wire     : std_logic;
188   signal fifoRx_rdreq_control   : std_logic;
189   signal fifoRx_wrreq_control   : std_logic;
190   signal fifoRx_out_wire      : std_logic_vector (7 downto 0);
191   signal txd_invert           : std_logic_vector (7 downto 0);
192   -- sinais fixo tx
193   signal fifoTx_rdempty_wire    : std_logic;
194   signal fifoTx_wrfull_wire     : std_logic;
195   signal fifoTx_rdreq_control   : std_logic;
196   signal fifoTx_wrreq_control   : std_logic;
197
198   signal TxValidData        : std_logic;
199   signal RxValidData        : std_logic;
200   signal tx_write           : std_logic;
201
202 begin
203   u0 : component DE2_115_SOPC_bridge_pll
204     port map(
205       clk_clk => CLOCK_50,          -- clk.clk
206       clk_2m_clk => clk_2M,           -- clk_2m.clk
207       reset_reset_n => KEY0,          -- reset.reset_n
208       spi_master_external_MISO => MISO_m, -- spi_master_external.MISO
209       spi_master_external_MOSI => MOSI_m, -- .MOSI
210       spi_master_external_SCLK => SCLK,   -- .SCLK
211       spi_master_external_SS_n => SS_n,   -- .SS_n
212       tx_en_export => TX_en,          -- tx_en.export
213       rst_qsys_export => rst_qsys
```

```
214
215     );
216
217     fifo_catalog_rx : component fiforx port map(
218       data => rxd_wire,
219       rdclk => clk_2M,
220       rdreq => fifoRx_rdreq_control,
221       wrclk => clk_2M,
222       wrreq => fifoRx_wrreq_control,
223       q => fifoRx_out_wire,
224       rdempty => fifoRx_rdempty_wire,
225       wrfull => fifoRx_wrfull_wire
226     );
227
228     fifo_catalog_tx : component fifotx port map(
229       data => txd_invert,
230       rdclk => clk_2M,
231       rdreq => fifoTx_rdreq_control,
232       wrclk => clk_2M,
233       wrreq => fifoTx_wrreq_control,
234       q => txd_wire,
235       rdempty => fifoTx_rdempty_wire,
236       wrfull => fifoTx_wrfull_wire
237     );
238
239     fifo_controller_top : component fifo_controller
240       port map(
241         pclk => clk_2M,
242         FS => fs_wire,
243         reset => reset,
244         --fifo rx
245         rdreq_fiforx => fifoRx_rdreq_control,
246         wrreq_fiforx => fifoRx_wrreq_control,
247         TxValidData => TxValidData,
248         frame_num => frame_num,
249         RxValidData => RxValidData,
250         rdempty_fixorx => fifoRx_rdempty_wire,
251         wrfull_fiforx => fifoRx_wrfull_wire,
252         --fifo tx
```

```
253         rdreq_fifotx => fifoTx_rdreq_control,
254         wrreq_fifotx => fifoTx_wrreq_control,
255         rdempty_fixotx => fifoTx_rdempty_wire,
256         tx_write => tx_write,
257         txd_wire => fifoRx_out_wire,
258         txd_invert => txd_invert,
259         wrfull_fifotx => fifoTx_wrfull_wire
260       );
261
262       frame_strobe : component frame_sync port map(
263         pclk => clk_2M,
264         reset => reset,
265         strobe => fs_wire
266       );
267
268       pcm_ctrl_inst : pcm_ctrl
269       port map(
270         reset => reset,
271         Pclk => clk_2M,
272         fs => fs_wire,
273         frame_start => frame_start,
274         frame_end => frame_end,
275         frame_num => frame_num
276       );
277       frame_num_aux <= "000" & frame_num;
278
279       pcm_tx_tb_inst : pcm_tx_tb
280       port map(
281         frame_start => frame_start,
282         frame_end => frame_end,
283         fs => fs_wire,
284         Pclk => clk_2M,
285         reset => reset,
286         frame_num => frame_num_aux,
287         DXA => DXA_wire
288       );
289
290       --Para entrar com o a saida do DXA deve alterar o DSTi_wire para
    DXA_wire na porta DSTi
```

```
291       tdm_cont : tdm_cont_ent port map(
292         rst_n => reset,        -- System reset
293         CLK_I => CLOCK_50,     -- System clock
294
295         -- ST-Bus
296         C2 => clk_2M,          -- ST-Bus Clock
297         DSTi => TDMI0,         -- in ST-Bus input Data
298         DSTo => TDMO0,         -- out ST-Bus output Data
299         F0_n => fs_wire,       -- IN ST-Bus framing pulse
300         F0od_n => F0od_n_wire,  -- out ST-Bus delayed framing pulse
301
302         -- BackEnd
303         RxD => rxd_wire,       -- out Parellel Rx output
304         TxD => txd_wire,       -- in Parellal Tx input
305
306         -- Controle
307         NoChannels => "11111",  -- in - Números time slot -> 111
308         DropChannels => "00000", -- in - Time slot to be dropped ->
    000
309
310         -- Backend
311         RxValidData => RxValidData,   -- out valid data strobe  -> 1
312         --FramErr         => FramErr,    -- out wb              -> 0
313
314         -- Backend
315         RxRead => frame_end, -- in - read byte, fsm rx_Buffer{idle:0,
    read:0, write:RxRdy, waitwrite:1}
316         RxRdy => RxRdy_view_wire, -- out  - valid data exist - fsm
    tdm_count {idle:0, write:1, others:1}
317
318         TxValidData => TxValidData, --in - Valid Data, fsm tx_buffer {
    idle:0, read:1, waitread:1, write:0}
319         TxWrite => '1', --in - Write Byte, fsm tdm_count {idle:0, read
    :0, waitread:1, write:0}
320         TxRdy => TxRdy_view_wire, --out- Ready to get data fsm
    tdm_count {"11":0,"00":0,"01":1,"others":0}
321
322         -- Signal
323         EnableSerialIF => '0', --  (EnableSerialIF = '1') THEN DSTo <=
```

```
    SerDi; ELSE DSTo <= Tx_reg(7);
324         SerDi => SerDi_wire, -- Serial Data in
325         SerDo => SerDo_wire
326
327       );
328
329       reset <= '0' when rst_qsys = '1' else '1';
330
331       DIO <= MOSI_m when TX_en = '1' else 'Z';
332       MISO_m <= DIO when TX_en = '0' else 'Z';
333
334       DCLK <= SCLK;
335       CS <= SS_n;
336       C2_SYS <= clk_2M;
337       FS <= fs_wire;
338
339     end system;
```

Código A.2 – Entidade para sincronização de quadros

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity frame_sync is
6    port (
7      pclk     : in    std_logic;
8      reset    : in    std_logic;
9      strobe   : out   std_logic
10   );
11 end entity frame_sync;
12
13 architecture arch of frame_sync is
14
15   signal count_next, count_reg : unsigned(7 downto 0);
16
17 begin
18
19   process (pclk, reset) is
20   begin
```

```
21     if (reset = '0') then
22     count_reg <= (others => '0');
23     elsif (pclk'event and pclk = '1') then
24     count_reg <= count_next;
25     end if;
26   end process;
27
28   count_next <= count_reg + 1;
29   strobe <= '1' when count_reg = 0 else
30         '0';
31
32 end architecture arch;
```

Código A.3 – Entidade para identificar os canais

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  --bloco do circuito pcm
6  entity pcm_ctrl is
7    port (
8      reset, Pclk, fs: in std_logic;
9      frame_start, frame_end : out std_logic;
10     frame_num : out std_logic_vector(4 downto 0)
11   );
12 end entity;
13
14 architecture ifsc_v1 of pcm_ctrl is
15   signal r_reg8, r_next8: unsigned(2 downto 0);
16    signal r_reg32, r_next32, enable: unsigned(4 downto 0);
17    signal frame_start_std, frame_end_std: std_logic;
18 begin
19
20   -- register
21   process (Pclk, reset)
22   variable aux_end : integer range 0 to 8;
23   begin
24     if (reset = '0') then
25       r_reg8 <= (others => '0');
```

```
26      r_reg32 <= "00000";
27    elsif (Pclk'event and Pclk = '1') then
28      r_reg8 <= r_next8;
29      r_reg32 <= r_next32;
30    end if;
31  end process;
32
33   r_next8 <= r_reg8 + 1 when fs = '0' else (others => '0');
34
35  process (r_reg8,frame_start_std,frame_end_std,enable,r_reg32)
36  begin
37    if (r_reg8="000") then
38      frame_start_std <= '1';
39    else
40      frame_start_std <= '0';
41    end if;
42
43    if (r_reg8="111") then
44      enable <= r_reg32 +1;
45      frame_end_std <='1';
46
47    else
48      enable <= r_reg32 +1;
49      frame_end_std <='0';
50    end if;
51
52    if (frame_end_std = '1') then
53      enable <= r_reg32 + 1;
54    else
55      enable <= r_reg32;
56    end if;
57
58    frame_start <= frame_start_std;
59    frame_end <= frame_end_std;
60  end process;
61
62  r_next32 <= enable when fs = '0' else (others => '0');
63
64  frame_num <= std_logic_vector(r_reg32);
```

```
65
66 end architecture;
```

Código A.4 – Entidade para controlar as FIFOs

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4
5  entity fifo_controller is
6    port (
7      pclk : in std_logic;
8      FS, reset : in std_logic;
9      --fifo rx
10     rdreq_fiforx : out std_logic; -- 1 lê o bit da fifo, 0 não lê
11     wrreq_fiforx : out std_logic; -- 1 escreve na fifo, 0 não escreve
12     TxValidData : out std_logic;
13     RxValidData : in std_logic;
14     frame_num : in std_logic_vector(4 downto 0);
15     txd_invert : out std_logic_vector(7 downto 0);
16     txd_wire : in std_logic_vector(7 downto 0);
17     rdempty_fixorx : in std_logic; -- 0 tem dados na fifo
18     wrfull_fiforx : in std_logic; -- 1 fifo está cheia
19     --fifo tx
20     rdreq_fifotx : out std_logic; -- 1 lê o bit da fifo, 0 não lê
21     wrreq_fifotx : out std_logic; -- 1 escreve na fifo, 0 não escreve
22     rdempty_fixotx : in std_logic; -- 0 tem dados na fifo
23     tx_write : out std_logic;
24     wrfull_fifotx : in std_logic -- 1 fifo está cheia
25   );
26 end;
27
28 architecture fifo_controller of fifo_controller is
29   type States_type is (valid_rx, write_tx, read_tx, valid_tx);
30   signal state : States_type;
31
32   type Inverter_type is (start_byte_0, invert_byte_0, store_byte_1,
       invert_byte_1, store_byte_0);
33   signal state_invert : Inverter_type;
34   signal write_fifo_tx : std_logic;
```

```
35   signal byte_0 : std_logic_vector(7 downto 0);
36   signal byte_1 : std_logic_vector(7 downto 0);
37 begin
38
39   process (pclk, reset, FS) is
40     variable bit_count : integer range 0 to 7;
41   begin
42     if (reset = '0') then
43       state <= valid_rx;
44       rdreq_fifotx <= '0';
45       wrreq_fifotx <= '0';
46       rdreq_fiforx <= '0';
47       wrreq_fiforx <= '0';
48       TxValidData <= '0';
49       tx_write <= '0';
50       bit_count := 0;
51
52     elsif (pclk'event and pclk = '1') then
53
54       -- máquina de estado para controlador leitura e escrita das
    fifos
55       case State is
56         when valid_rx =>
57           if (RxValidData = '1') then
58             wrreq_fiforx <= '1';
59             rdreq_fiforx <= '1';
60             state <= write_tx;
61           end if;
62
63         when write_tx =>
64
65           if (write_fifo_tx = '1') then
66             wrreq_fifotx <= '1';
67             state <= valid_tx;
68           end if;
69
70         when valid_tx =>
71
72           if (frame_num = "11110") then
```

```
73              TxValidData <= '1';
74              rdreq_fifotx <= '1';
75          state <= read_tx;
76            end if;
77
78          when read_tx =>
79
80            if (RxValidData = '0') then
81              wrreq_fiforx <= '0';
82              rdreq_fiforx <= '0';
83              state <= valid_rx;
84            end if;
85        end case;
86
87        -- máquina de estado para inverter os dados dos time slots
88        case state_invert is
89
90          when start_byte_0 =>
91            if (rdempty_fixorx = '0') then
92              if (bit_count = 7) then
93                bit_count := 0;
94                byte_0 <= txd_wire;
95                state_invert <= invert_byte_0;
96              else
97                bit_count := bit_count + 1;
98              end if;
99            end if;
100
101         when invert_byte_0 =>
102           if (bit_count = 6) then
103             write_fifo_tx <= '1';
104           end if;
105           if (bit_count = 7) then
106             bit_count := 0;
107             txd_invert <= txd_wire;
108             state_invert <= store_byte_1;
109           else
110             bit_count := bit_count + 1;
111           end if;
```

```
112
113         when store_byte_1 =>
114           if (bit_count = 7) then
115             bit_count := 0;
116             txd_invert <= byte_0;
117             byte_1 <= txd_wire;
118             state_invert <= invert_byte_1;
119           else
120             bit_count := bit_count + 1;
121           end if;
122
123         when invert_byte_1 =>
124           if (bit_count = 7) then
125             bit_count := 0;
126             txd_invert <= txd_wire;
127             state_invert <= store_byte_0;
128           else
129             bit_count := bit_count + 1;
130           end if;
131
132         when store_byte_0 =>
133           if (bit_count = 7) then
134             bit_count := 0;
135             txd_invert <= byte_1;
136             byte_0 <= txd_wire;
137             state_invert <= invert_byte_0;
138           else
139             bit_count := bit_count + 1;
140           end if;
141       end case;
142     end if;
143   end process;
144
145 end;
```

APÊNDICE B – *SOFTWARE* PARA CONFIGURAR O *CODEC* USANDO O NIOS II

As listagens a seguir apresentam o *header* da classe proposta para configurar a placa ramal e a implementação dessa classe, desenvolvidas com a linguagem de programação *C++*. O Código B.1 apresenta a classe da placa ramal e o Código B.2 uma implementação da classe placa ramal.

Código B.1 – *Header* da classe placa ramal

```
1  #include "alt_types.h"
2  #include "altera_avalon_spi_regs.h"
3  #include "altera_avalon_spi.h"
4  #include "system.h"
5  #include "altera_avalon_pio_regs.h"
6  #include "io.h"
7  #include <stdio.h>
8  #include <unistd.h>// usleep
9
10 //Comandos utilizados para configurar a codec
11 #define WRITE_CHIP_CONFIGURATION_REGISTER 0X46              //          |01000110
12 #define READ_CHIP_CONFIGURATION_REGISTER 0X47               //          |01000111
13 #define WRITE_DEBOUNCE_TIME_REGISTER 0xC8                   //          |11001000
14 #define WRITE_CHANNEL_ENABLE 0x4A                           //          |01001010
15 #define WRITE_SET_TIMESLOT 0X44                             //          |01000100
16 #define WRITE_INTERRUPT_MASK_REGISTER 0X6C                  //          |01101100
17 #define WRITE_REAL_TIME_DATA_REGISTER 0X4D                  //          |01001101
18 #define WRITE_REAL_TIME_DATA_REGISTER_CLEAR_INTERRUPT 0X4F  //          |01001111
19 #define WRITE_OPERATING_FUNCTIONS 0X60                      //          |01100000
20 #define WRITE_SLIC_DIRECTION_STATUS_BITS 0X54               //          |01010100
21 #define READ_SLIC_DIRECTION_STATUS_BITS 0X55                //          |01010101
22 #define ACTIVATE_CODEC 0x0E                                 //          |00001110
23 #define WRITE_IO_REGISTER 0X52                              //          |01010010
24 #define READ_IO_REGISTER 0X53                               //          |01010011
25 #define RING_CHANNEL 0X1C                                   //          |00011100
26 #define STOP_RING_CHANNEL 0X1E                              //          |00011110
27 #define HARDWARE_RESET 0X04
28 #define TRANSMIT_TIME_SLOT 0x40
29 #define RECEIVE_TIME_SLOT 0x42
30 #define HARDWARE_RESET 0X04
31 #define OPERATING_CONDITIONS 0x70
```

```
32 #define AISN_ANALOG_GAINS 0x50
33 #define GR_FILTER_COEFFICIENTS 0x82
34 #define GX_FILTER_COEFFICIENTS 0x80
35 #define R_FILTER_COEFFICIENTS 0x8A
36 #define Z_FILTER_COEFFICIENTS 0x84
37 #define B1_FILTER_COEFFICIENTS 0x86
38 #define B2_FILTER_COEFFICIENTS 0x96
39 #define TRANSMIT_TIME_SLOT 0x40
40 #define RECEIVE_TIME_SLOT 0x42
41
42 //Valores padroes
43 #define MASTER_CLOCK 0X12     //2.048 Mhz - PCLK            //         |00010010
44 #define DEBOUNCE_TIME 0X3C                                  //         |00111100
45 #define TIME_SLOT 0X00        //FS sincronizado com borda negativa do PCLK
46 #define INTERRUPT_MASK 0XAA                                 //         |10101010
47 #define TIME_REAL_DATA 0XAA                                 //         |10101010
48 #define DEFAULT_COEFFICIENT 0X00
49 #define DIRECTION_SLIC_CD1_CD2 0X02 //CD1 INPUT - CD2 OUTPUT          |00000010
50 #define TAMANHOBUFFER 8
51 #define CHANNEL_1 0x01
52 #define CHANNEL_2 0x02
53
54 class Placaramal{
55   public:
56     // construtores
57     Placaramal();
58
59     ~Placaramal() {};
60
61     void periodical();
62     void init();
63     void init_channel1();
64     void init_channel2();
65     void take_hook(int canal);
66     void put_hook();
67     void veirify_attendance();
68     void dial_tone();
69     void attendance();
70     void clean_buffer();
71     void set_master_clock(alt_u8 clock);
72     void set_coefficients(alt_u8 operation);
73     void channel_enable(alt_u8 channell);
74     void set_slic_direction(alt_u8 direction);
75     void set_slic_direction();
76     void activate_codec();
77     void operating_conditions();
78     void set_debounce_time(alt_u8 debounce);
```

```
79      void set_time_slot(alt_u8 time_slot);
80      void set_time_real_data();
81      void set_interrupt_mask(alt_u8 mask);
82      void transmit_time_slot(alt_u8 timeslot);
83      void receive_time_slot(alt_u8 timeslot);
84      void channel_ring();
85      void hardware_reset();
86      void write_codec(alt_u8 comando_codec, alt_u8 valor_comando);
87      void write_codec(alt_u8 comando_codec);
88      void monitore_channel();
89      alt_u8 read_codec(alt_u8 comando_codec);
90      bool is_cfail(alt_u8 analisar_byte);
91
92    private:
93      // atributos da classe: cada objeto desta classe
94      alt_u8 tx_buf[8];
95      alt_u8 rx_buf[8];
96
97          alt_u8 coefficient_filter_gr_channel1[2] ;
98          alt_u8 coefficient_filter_gx_channel1[2] ;
99          alt_u8 coefficient_filter_r_channel1[14] ;
100         alt_u8 coefficient_filter_z_channel1[15] ;
101         alt_u8 coefficient_filter_b1_channel1[14];
102         alt_u8 coefficient_filter_b2_channel1[2] ;
103
104         alt_u8 coefficient_filter_gr_channel2[2] ;
105         alt_u8 coefficient_filter_gx_channel2[2] ;
106         alt_u8 coefficient_filter_r_channel2[14] ;
107         alt_u8 coefficient_filter_z_channel2[15] ;
108         alt_u8 coefficient_filter_b1_channel2[14];
109         alt_u8 coefficient_filter_b2_channel2[2] ;
110 };
```

Código B.2 – Implementação da classe placa ramal

```
1  #include "placaramal.h"
2
3  Placaramal::Placaramal() {
4
5  }
6      void Placaramal::init(){
7          this->write_codec(HARDWARE_RESET);
8          usleep(4000000);
9          this->set_master_clock(MASTER_CLOCK);
10         //#usleep(1000000);
11         printf("Valor recebido do codec: %02x: \n",this->read_codec(
       READ_CHIP_CONFIGURATION_REGISTER));
12         this->set_debounce_time(DEBOUNCE_TIME);
```

```
13          this->set_time_slot(TIME_SLOT);
14          printf("Valor recebido do codec: %02x: \n",this->read_codec(
    READ_SLIC_DIRECTION_STATUS_BITS));
15          this->channel_enable(CHANNEL_1);
16          this->set_interrupt_mask(INTERRUPT_MASK);
17          this->set_coefficients(0x00);
18          this->set_slic_direction(DIRECTION_SLIC_CD1_CD2);
19          this->activate_codec();
20          printf("Valor recebido do codec: %02x: \n",this->read_codec(
    READ_SLIC_DIRECTION_STATUS_BITS));
21          this->write_codec(WRITE_CHANNEL_ENABLE,CHANNEL_1);
22          this->write_codec(WRITE_IO_REGISTER,0x1E);
23          this->write_codec(WRITE_CHANNEL_ENABLE,0x08);
24          this->write_codec(WRITE_IO_REGISTER,0x1E);
25      }
26
27      void Placaramal::set_master_clock(alt_u8 clock){
28          this->write_codec(WRITE_CHIP_CONFIGURATION_REGISTER, clock);
29      }
30
31      void Placaramal::set_debounce_time(alt_u8 debounce){
32          this->write_codec(WRITE_DEBOUNCE_TIME_REGISTER, debounce);
33      }
34
35      void Placaramal::set_time_slot(alt_u8 time_slot){
36          this->write_codec(WRITE_SET_TIMESLOT,time_slot);
37      }
38
39      void Placaramal::set_time_real_data(){
40          this->write_codec(WRITE_REAL_TIME_DATA_REGISTER,TIME_REAL_DATA);
41      }
42
43      void Placaramal::set_coefficients(alt_u8 operation){
44          this->write_codec(WRITE_OPERATING_FUNCTIONS, operation);
45      }
46
47      void Placaramal::channel_enable(alt_u8 channell){
48          this->write_codec(WRITE_CHANNEL_ENABLE,channell);
49      }
50
51      void Placaramal::set_slic_direction(){
52          this->write_codec(WRITE_SLIC_DIRECTION_STATUS_BITS,DIRECTION_SLIC_CD1_CD2);
53      }
54
55      void Placaramal::set_slic_direction(alt_u8 direction){
56          this->write_codec(WRITE_SLIC_DIRECTION_STATUS_BITS, direction);
57      }
```

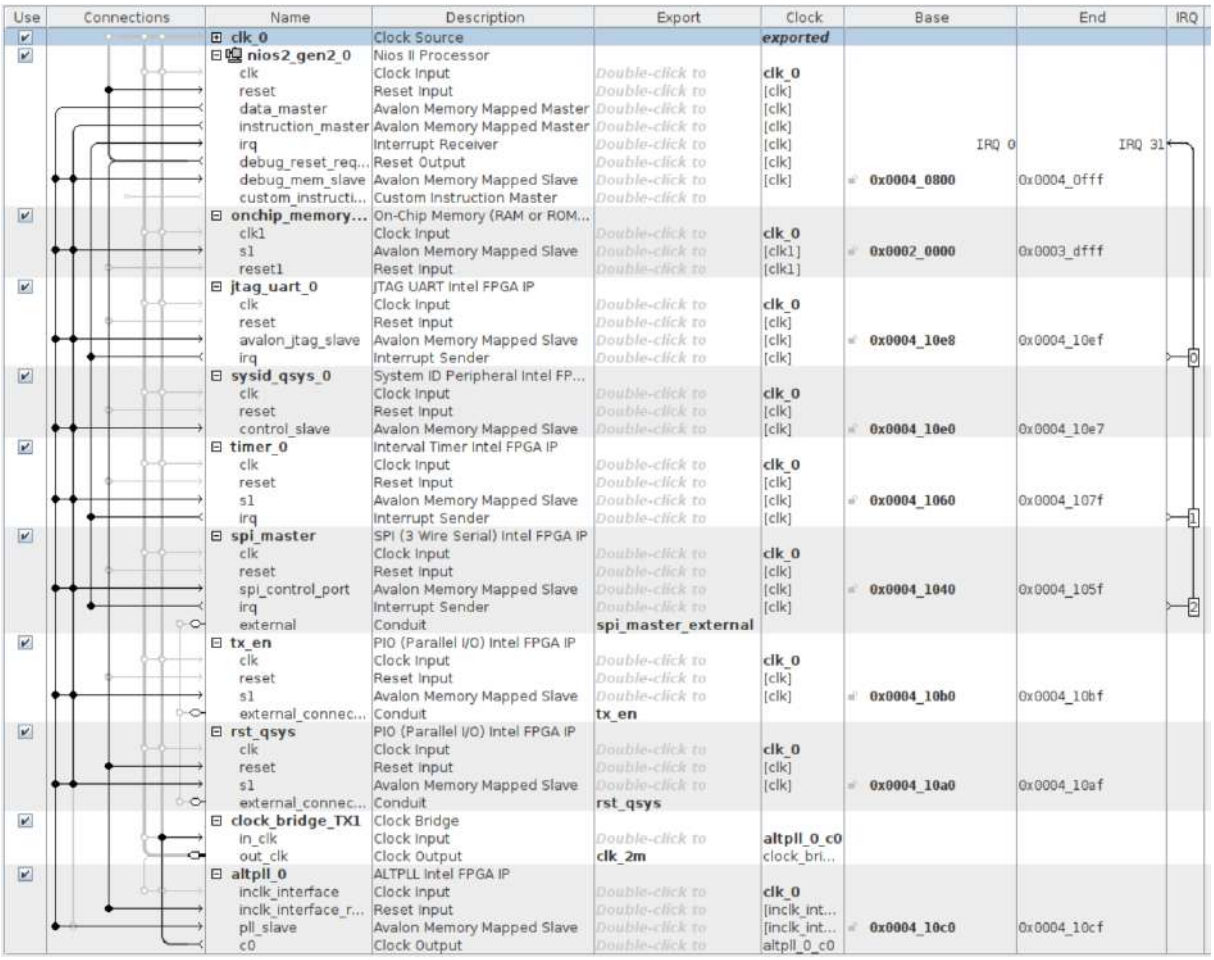
```
58
59      void Placaramal::activate_codec(){
60          this->write_codec(ACTIVATE_CODEC);
61      }
62
63      void Placaramal::set_interrupt_mask(alt_u8 mask){
64          this->write_codec(WRITE_INTERRUPT_MASK_REGISTER, mask);
65      }
66
67      void Placaramal::operating_conditions(){
68          this->write_codec(OPERATING_CONDITIONS,0x00);
69      }
70
71      bool Placaramal::is_cfail(alt_u8 analisar_byte){
72          //printf("%02x\n",analisar_byte);
73          return false;
74      }
75
76      void Placaramal::init_channel1(){
77          this->write_codec(WRITE_CHANNEL_ENABLE,CHANNEL_1);
78          this->write_codec(AISN_ANALOG_GAINS,0x00);
79          this->set_coefficients(0x3F);
80          this->set_coefficients(0x3F);
81          this->write_codec(AISN_ANALOG_GAINS,0x00);
82          this->activate_codec();
83          this->transmit_time_slot(0x00);
84          this->receive_time_slot(0x00);
85      }
86
87      void Placaramal::init_channel2(){
88          this->write_codec(WRITE_CHANNEL_ENABLE,CHANNEL_2);
89          this->write_codec(AISN_ANALOG_GAINS,0x80);
90          this->write_codec(WRITE_CHANNEL_ENABLE,CHANNEL_2);
91          this->set_coefficients(0x3F);
92          this->set_coefficients(0x3F);
93          this->write_codec(AISN_ANALOG_GAINS,0x00);
94          this->activate_codec();
95          this->transmit_time_slot(0x01);
96          this->receive_time_slot(0x01);
97      }
98
99      void Placaramal::channel_ring(){
100     int count = 0;
101
102     IOWR_ALTERA_AVALON_PIO_DATA(RST_QSYS_BASE, 1);
103     usleep(500000);
104     IOWR_ALTERA_AVALON_PIO_DATA(RST_QSYS_BASE, 0);
```

```
105
106     this->init_channel1();
107     this->init_channel2();
108
109         while (count<2) {
110             this->write_codec(WRITE_CHANNEL_ENABLE,CHANNEL_1);
111             this->write_codec(WRITE_IO_REGISTER,RING_CHANNEL);
112             usleep(1000000);
113             this->write_codec(WRITE_IO_REGISTER,STOP_RING_CHANNEL);
114             usleep(4000000);
115             count ++;
116         }
117         this->write_codec(WRITE_CHANNEL_ENABLE,CHANNEL_2);
118         this->write_codec(AISN_ANALOG_GAINS,0x00);
119         this->set_coefficients(0x3F);
120         this->write_codec(WRITE_IO_REGISTER,0x1F);
121     }
122
123     alt_u8 Placaramal::read_codec(alt_u8 comando_codec){
124         this->clean_buffer();
125         this->tx_buf[0]=comando_codec;
126
127         IOWR_ALTERA_AVALON_PIO_DATA(TX_EN_BASE, 1);
128         alt_avalon_spi_command(SPI_MASTER_BASE, 0, 1, &this->tx_buf[0], 0, &this->
    rx_buf[0], 0);
129
130         IOWR_ALTERA_AVALON_PIO_DATA(TX_EN_BASE, 0);
131         alt_avalon_spi_command(SPI_MASTER_BASE, 0, 0, &this->tx_buf[0], 1, &this->
    rx_buf[0], 0);
132         return this->rx_buf[0];
133     }
134
135     void Placaramal::write_codec(alt_u8 comando_codec, alt_u8 valor_comando){
136         this->clean_buffer();
137         this->tx_buf[0]=comando_codec;
138         this->tx_buf[1]=valor_comando;
139
140         IOWR_ALTERA_AVALON_PIO_DATA(TX_EN_BASE, 1);
141         alt_avalon_spi_command(SPI_MASTER_BASE, 0, 1, &this->tx_buf[0], 0, &this->
    rx_buf[0], 0);
142         alt_avalon_spi_command(SPI_MASTER_BASE, 0, 1, &this->tx_buf[1], 0, &this->
    rx_buf[1], 0);
143     }
144
145     void Placaramal::write_codec(alt_u8 comando_codec){
146         this->clean_buffer();
147         this->tx_buf[0]=comando_codec;
```

```
148
149         IOWR_ALTERA_AVALON_PIO_DATA(TX_EN_BASE, 1);
150         alt_avalon_spi_command(SPI_MASTER_BASE, 0, 1, &this->tx_buf[0], 0, &this->
        rx_buf[0], 0);
151     }
152
153     void Placaramal::transmit_time_slot(alt_u8 timeslot){
154         this->write_codec(TRANSMIT_TIME_SLOT, timeslot);
155     }
156
157     void Placaramal::receive_time_slot(alt_u8 timeslot){
158         this->write_codec(RECEIVE_TIME_SLOT, timeslot);
159     }
160     void Placaramal::monitore_channel(){
161       while(true){
162       usleep(3000000);
163       alt_u8 retorno = this->read_codec(
        WRITE_REAL_TIME_DATA_REGISTER_CLEAR_INTERRUPT);
164       if(retorno==0xAE){
165         channel_ring();
166       }
167       printf("Valor recebido do codec: %02x: \n", retorno);
168       }
169     }
170
171     void Placaramal::clean_buffer(){
172         for(int i=0;i<TAMANHOBUFFER;i++){
173             this->tx_buf[i] = 0;
174             this->rx_buf[i]= 0;
175         }
176 }
```

APÊNDICE C – CONFIGURAÇÃO NO *PLATFORM DESIGNER*

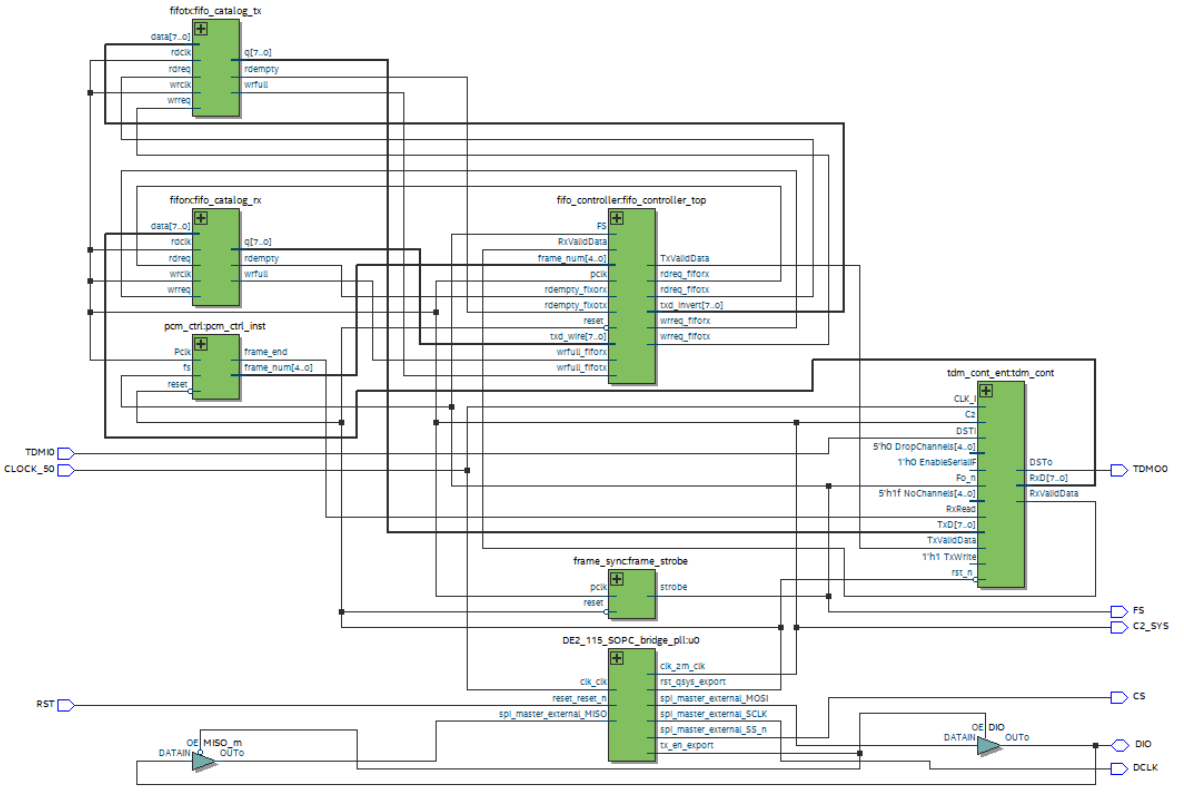
Figura 48 – interconexões realizadas na ferramenta *Platform Designer*



Fonte: Própria

APÊNDICE D – RTL *VIEWER* DO PROJETO QUARTUS

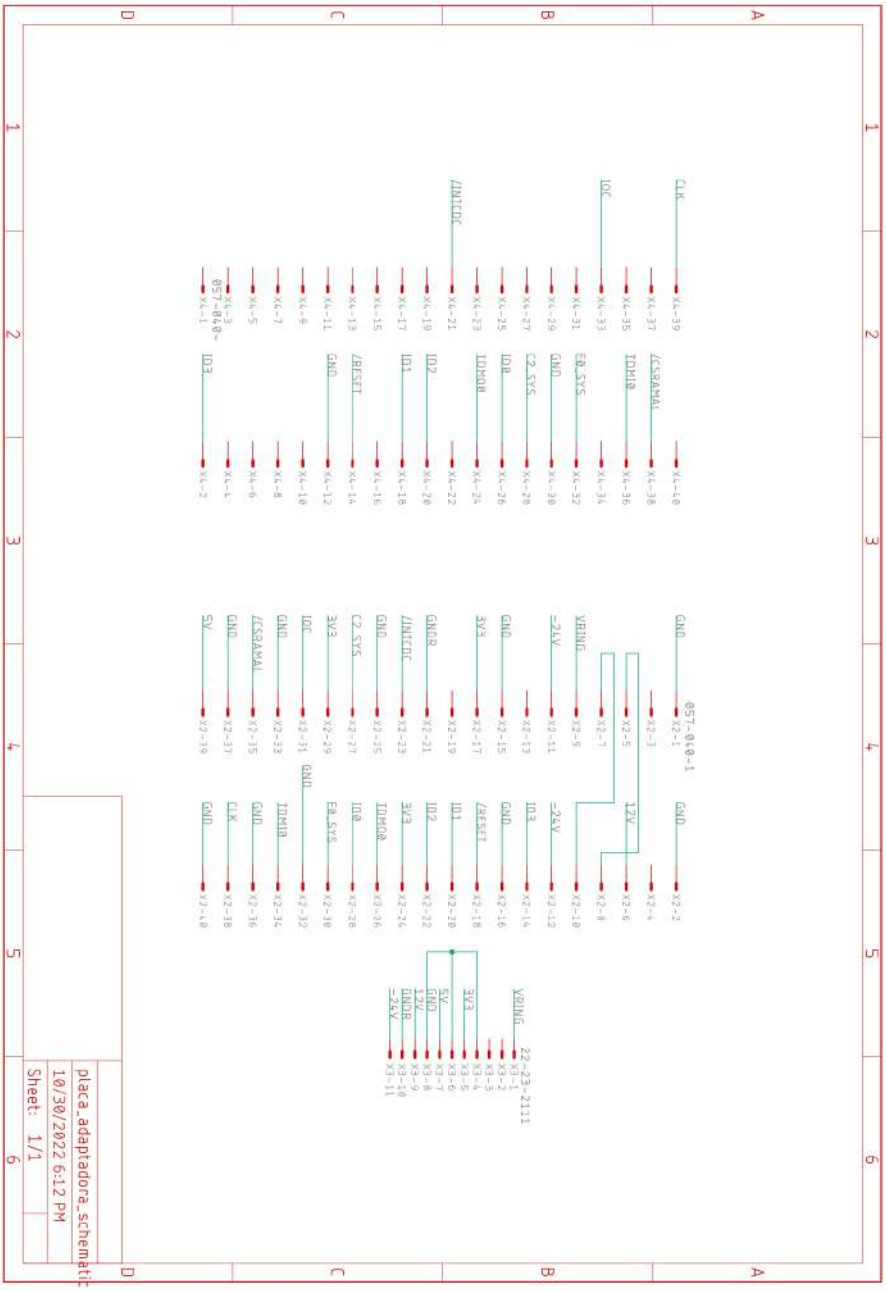
Figura 49 – RTL *viewer* completo do projeto Quartus



Fonte: Própria

APÊNDICE E – ESQUEMÁTICO COMPLETO DA PLACA ADAPTADORA

Figura 50 – Esquemático completo da placa adaptadora



Fonte: Própria