Universidade de Brasília
Faculdade do Gama

HiLDA: Ferramenta de Geração Automática de Código para Co-Projeto HW/SW

Lucas Xavier de Moura

TRABALHO DE CONCLUSÃO DE CURSO

ENGENHARIA ELETRÔNICA

Brasília

2023

Universidade de Brasília
Faculdade do Gama

HiLDA: Ferramenta de Geração Automática de Código para Co-Projeto HW/SW

Lucas Xavier de Moura

Trabalho de conclusão de curso submetido como requisito parcial para obtenção do grau de Engenheiro Eletrônico.

Orientador: Prof. Dr. Daniel Mauricio Muñoz Arboleda

Brasília
2023

Xavier de Moura, Lucas.
X769h HiLDA: Ferramenta de Geração Automática de Código para Co-Projeto HW/SW / Lucas Xavier de Moura; orientador Daniel Mauricio Muñoz Arboleda. -- Brasília, 2023.
139 p.

Trabalho de conclusão de curso (Engenharia Eletrônica) -- Universidade de Brasília, 2023.

1. RTL. 2. FSM. 3. Geradores de código. 4. VHDL. I. Mauricio Muñoz Arboleda, Daniel , orient. II. Título

Universidade de Brasília
Faculdade do Gama

HiLDA: Ferramenta de Geração Automática de Código para Co-Projeto HW/SW

Lucas Xavier de Moura

Trabalho de conclusão de curso submetido como requisito parcial para obtenção do grau de Engenheiro Eletrônico.

Trabalho aprovado. Brasília, 5 de outubro de 2023:

Prof. Dr. Daniel M. Muñoz Arboleda, UnB/FGA
Orientador

Prof. Dr. Gilmar Silva Beserra, UnB/FGA
Examinador interno

Prof. Dr. Guillermo Alvarez Bestard, UnB/FGA
Examinador interno

Brasília
2023

Resumo

A complexidade do desenvolvimento de circuitos digitais aumentou nas últimas décadas visto que atualmente uma porção dos projetos necessita da integração entre hardware e software. Para resolver este problema, diversas ferramentas de modelagem e síntese de alto nível tem sido criadas, facilitando a geração de código HDL a partir de uma descrição em alto nível de abstração. Este trabalho mostra o desenvolvimento da ferramenta *HiLDA*, que possui quatro ferramentas integradas, *vFSMgen*, *vRTLgen*, *vTBgen* e *AXIcreator*, um gerador de máquinas de estados e de estruturas RTL (*Register Transfer Level*), um simples criador de *testbenches* e um gerador de interfaces AXI, respectivamente, que geram códigos em VHDL (*Very high speed integrated circuits Hardware Description Language*) a partir de diagramas de blocos na ferramenta *Draw.io*. A ferramenta é capaz de verificar sintaticamente os códigos gerados, bem como simulá-los, utilizando a ferramenta gHDL. O desenvolvimento da ferramenta foi possível a partir da criação de parsers de diagramas gerados pelo *Draw.io* e geradores de código utilizando Python, bem como uma interface gráfica desenvolvida a partir da aplicação Godot para integrar todas essas ferramentas. Os resultados obtidos são promissores visto que as máquinas de estados geradas pela ferramenta *vFSMgen* conseguem facilmente atingir a síntese e a implementação em hardware. Similarmente, a ferramenta *vRTLgen* também consegue produzir circuitos digitais de mediana complexidade sintetizáveis e implementáveis em dispositivos FPGA (*Field Programmable gate Arrays*). Por fim, a ferramenta *AXIcreator* também se mostrou efetiva em encapsular códigos em AXI4-Full e AXI4-Lite, sendo que estes foram validados na placa Avnet Zedboard.

Palavras-chave: RTL. FSM. Geradores de código. VHDL.

Abstract

The complexity of digital circuits development increased in recent decades considering that currently a portion of projects requires the integration of both hardware and software.In order to address this issue, numerous high-level modeling and synthesis tools have been developed to facilitate the generation of HDL (Hardware Description Language) code from a high-level abstract description.Thus, this work presents the development of HiLDA tool,which has two integrated tools, *vFSMgen* and *vRTLgen*, a generator of state machines and RTL(*Register Transfer Level*) structures respectively, which generates codes in VHDL(*Very high speed integrated circuits Hardware Description Language*) from diagrams and is able to syntactically check the codes generated using the gHDL tool.The development of the tool was made feasible through the creation of parsers for diagrams generated by Draw.io and code generators using Python, as well as a graphical interface developed utilizing the Godot application to integrate all these tools.The obtained results are promising, as the state machines generated by the vFSMgen tool can achieve synthesis and hardware implementation on Field Programmable Gate Arrays (FPGAs). Similarly, *vRTLgen* can also produce synthesized and implementable codes in hardware of relatively complex circuits.Finally, the AXIcreator tool has proven to be effective in encapsulating code into AXI4-Full and AXI4-Lite interfaces validated on the Avnet Zedboard.

Keywords: RTL. FSM. Code generators. VHDL.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

BRAM	*Block Random Access Memorys* 18
CLB	*Configurable Logic Block* 18
DNN	*Deep Neural Network* 22
DSP	*Digital Signal Processing* 18
FF	Flip-Flop 15
FPGA	*Field-programmable gate array* 13
FSM	Máquinas de Estados Finitos 13
HDL	Linguagem de Descrição de Hardware 13
HLS	*High-Level Synthesis* 18
IP	Intellectual Property 23
LUT	*Look-Up Table* 17
ML	*Machine Learning* 22
MLP	*Multilayer Perceptron* 58
PSO	*Particle Swarm Optimization* 58
RBFNN	*Radial Basis Function Neural Network* 22
RTL	Nível de Transferência entre Registradores 14
SoC	*System on Chip* 17
UI	Interface de Usuário 23
UNB	Universidade de Brasília 21

Sumário

1 Introdução

Sistemas embarcados são utilizados em diversas indústrias e aplicações do dia a dia para executar tarefas específicas. São utilizados, por exemplo, em carros inteligentes, eletrodomésticos com função IOT (*Internet of things*), aplicações de reconhecimento de imagem na indústria alimentícia, controle industrial, entre outras (BERKOWITZ, 2023).

Recentemente, a indústria está demandando uma grande quantidade de engenheiros eletrônicos. Os grandes criadores de chips ao redor do mundo não conseguem expandir sua produção pela falta de engenheiros. Um dos principais causadores deste problema está na convicção que a área de software paga melhor que a área de hardware. Sendo que nas últimas décadas a procura por cursos na área software se tornaram muito mais populares que os de hardware (MARTIN, 2022).

O uso de geradores de códigos para a implementação de hardware em *Field-programmable gate arrays* FPGAs facilitam e aceleram o desenvolvimento de um circuito. Essas ferramentas possibilitam a automatização de tarefas que seguem padrões regulares e são, muitas vezes, repetitivas, de tal forma que erros são adicionados ao código a partir da desatenção e pequenos equívocos dos programadores ao redigi-lo. Sistemas de machine learning, por exemplo, dependem substancialmente de geradores de códigos (VASILACHE et al., 2023).

Outro aspecto que pode ser explorado é o caráter pedagógico que algumas dessas ferramentas possuem ao permitir que o design seja o foco do usuário, visto que aprender uma nova linguagem, neste caso uma Linguagem de Descrição de Hardware (HDL) , tem uma curva de aprendizagem demorada. A programação a partir de plataformas gráficas é mais rápida que a programação textual (BALID; ABDULWAHED, 2013).

Percebe-se que uma ferramenta para a confecção de circuitos que consiga atender usuários leigos e avançados, e que integre e centralize o uso de geradores de códigos para aproveitar estruturas regulares permitiria que tanto fosse possível ajudar na capacitação de pessoas quanto acelerar o processo de criação de circuitos complexos. Sendo que uma ferramenta com essas características possui alto potencial de uso acadêmico e industrial.

1.1 Objetivos

1.1.1 Objetivo geral

Desenvolver uma aplicação com a capacidade de gerar códigos em VHDL a partir de diagramas com uma interface gráfica de usuário intuitiva, com suporte para geradores de Máquinas de Estados Finitos (FSMs), desenvolvimento de projetos em nível de transferência

entre registradores (RTL) e co-projeto hardware/software.

1.1.2 Objetivo específicos

Com este trabalho espera-se alcançar os seguintes objetivos:

- Criação de uma biblioteca de elementos padrão no Draw.io;
- Desenvolvimento de um gerador de códigos de FSMs a partir de diagramas feitos no programa Draw.io;
- Desenvolvimento de um gerador de códigos de arquiteturas RTL a partir de diagramas feitos no programa Draw.io;
- Desenvolvimento de um gerador de codigo para encapsulamento das arquiteturas usando barramentos on-chip tipo AXI;
- Desenvolvimento de uma interface gráfica para integrar todas as ferramentas.

1.2 Contribuições do Trabalho

Este trabalho originou quatro ferramentas originais e diferentes. Primeiro, um gerador de código VHDL de máquinas de estados finitas a partir de diagramas XML que funciona para a maioria dos casos simples apresentados pelos livros-texto. Segundo, um gerador de código VHDL de estruturas RTL a partir de diagramas esquemáticos que podem conter tanto componentes em VHDL quanto geradores de código em Python. Terceiro, um criador *testbenchs* simples. Quarto, uma ferramenta para encapsulamento de circuitos em VHDL utilizando interfaces AXI4-Lite e AXI4-Full, validada em hardware. Por fim, uma interface gráfica que integra as duas primeiras ferramentas anteriores com a aplicação de código livre chamada gHDL, que é, um analisador, compilador e simulador de códigos VHDL.

2 Fundamentação Teórica

Para o desenvolvimento da aplicação é necessário conhecimento em projetos de circuitos digitais, bem como conhecimento em co-projetos Hardware/Software. Então, nesta seção são apresentados conceitos de FSMs, projetos RTL, bem como o funcionamento de FPGAs e SoCs, e finalizando com diversos trabalhos presentes na bibliografia atual de geradores de códigos.

2.1 Máquinas de Estados Finitos

As FSMs são ferramentas poderosas que podem ser utilizadas em várias aplicações, dentre elas, a FSM pode ser utilizada para realizar o controle de circuitos RTL.

Máquina de estados finitos constituem um termo muito amplo utilizado tanto no desenvolvimento de software e hardware, como no controle de personagens em jogos e no controle de circuitos RTL. No entanto, neste trabalho será abordado o que é chamado de *clocked synchronous state machine* ou máquinas de estados síncronas, sendo que esses circuitos utilizam de Flip-Flops (FFs) sincronizados por sinal de clock para armazenar informações (WAKERLY, 2006).

Uma FSM deve possuir um conjunto de estados, um conjunto de entradas e saídas, um estado inicial, transições, bem como a lógica de transição ou condição de transição e os valores de saída de cada estado (VAHID, 2007). A figura 2.1 mostra um exemplo de FSM, sendo *E0*, *E1*, e *E2* os estados, com *E*0 como o estado inicial, *a* e *x* são a entrada e a saída, respectivamente. As transições são representadas pelas setas. Dessa forma, o diagrama de estados da figura 2.1 representa uma FSM que ao receber a entrada *a* em nível lógico alto gera uma saída de nível lógico alto por dois ciclos de *clock*.

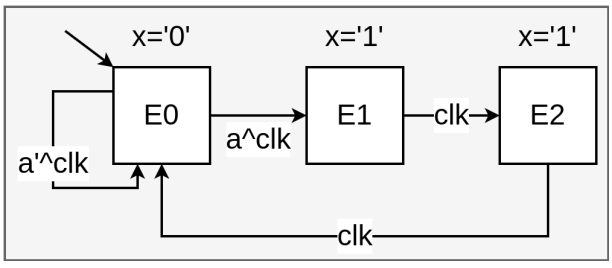

Figura 2.1 – Exemplo de uma FSM qualquer do tipo Moore.

2.1.1 Estrutura e tipos de FSMs

A estrutura de uma FSM é composta pelo registrador de estado, a lógica de transição de estados e a lógica de saída. O registrador de estados é composta por FFs, geralmente do

tipo D, enquanto que tanto a lógica de saída quanto a lógica de transição são implementados a partir de circuitos combinacionais, sendo que a ligação entre esses componentes varia conforme o tipo da FSM.

Existem dois tipos de FSMs, dos tipos *Moore* e *Mealy*, sendo que a lógica de transição de estados nos dois tipos depende do estado atual e das entradas, de modo que muda somente a lógica de saída . Na FSM do tipo *Moore* a saída depende somente do estado atual enquanto que no tipo *Mealy* depende do estado atual e também das entradas da FSM. A Figura 2.2 mostra a estrutura geral das FSMs.

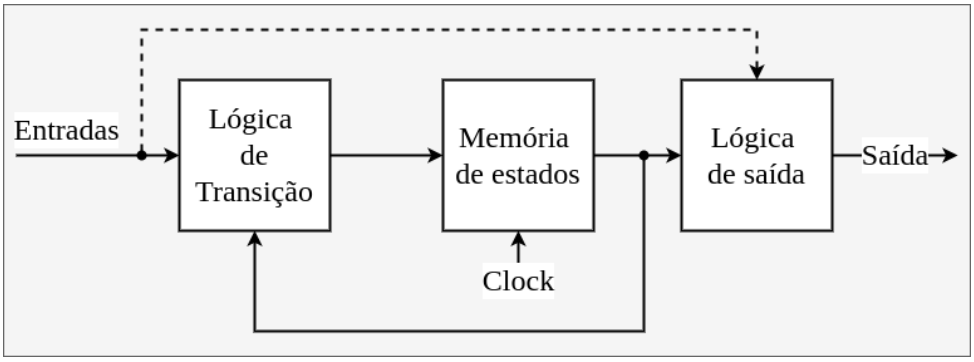

Figura 2.2 – Estrutura de FSMs do tipo *Moore* e *Mealy* adaptado de (WAKERLY, 2006).

2.2 Projeto RTL

O projeto RTL (*Register Transfer Level*) começa com a identificação do comportamento de um circuito e a partir deste comportamento o projetista estabelece as transferências e operações realizadas com as entradas, saídas e registradores, bem como o controle dessas transferências e operações necessárias para obter um circuito lógico digital (VAHID, 2007)

Um projeto RTL geralmente possui dois circuitos principais, ambos sincronizados por um mesmo clock, que precisam ser elaborados, como mostra a Figura 2.3. Primeiramente o *bloco de controle* que consiste em uma FSM e, em segundo lugar, o *bloco operacional*, que contém todas as operações e registradores necessários para o funcionamento do circuito.

O bloco de controle, como mencionado anteriormente, é basicamente uma FSM que recebe sinais binários de entrada, inclusive do bloco operacional, e controla os registradores e operações a partir de sinais de habilitação, *reset* e carregamento. Enquanto que o bloco operacional recebe os estímulos do bloco de controle em conjunto com as entradas de dados para realizar operações aritméticas, lógicas, relacionais, de deslocamento, entre outras. A Figura 2.4 mostra um exemplo de projeto RTL de um barramento simples que possui um bloco de controle que espera um endereço para realizar o envio de dados e um bloco operacional que salva os valores de 'Q' em um registrador e, ao receber um endereço válido, envia um sinal para o bloco de controle para enviar o dado.

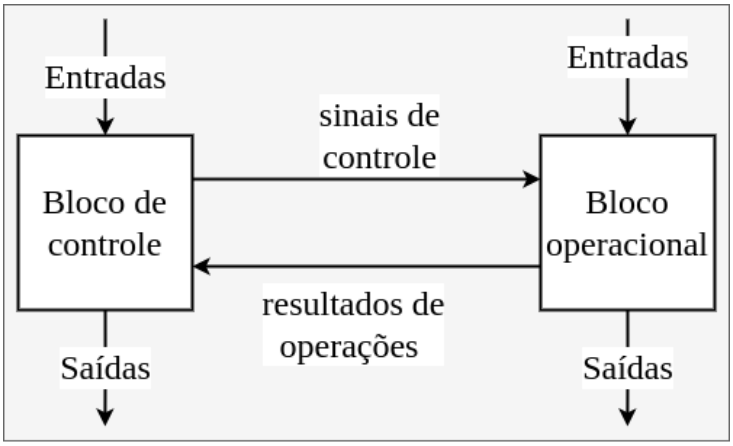

Figura 2.3 – Principais blocos de um projeto RTL.

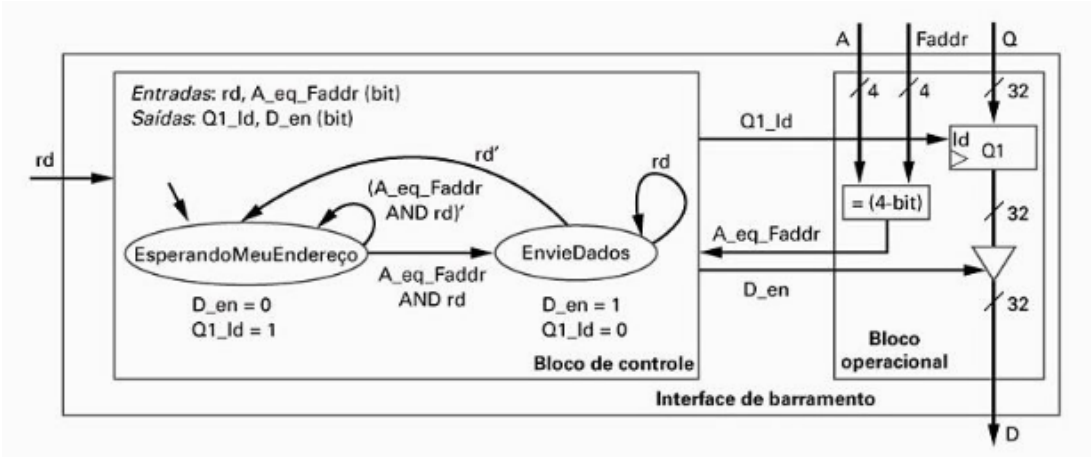

Figura 2.4 – Bloco operacional e de controle de uma interface de barramento simples (VAHID, 2007).

Normalmente a implementação de projetos RTL é custosa e complexa porque cada circuito necessita de componentes específicos. Assim, a partir do uso de FPGAs é possível rapidamente implementar e testar estes circuitos.

2.3 FPGAs e Sistemas em Chip (SoC)

Um FPGA contém um agrupamento de células lógicas genéricas em conjunto com interruptores programáveis para rotear as conexões entre células, assim a partir da combinação entre funções programadas em cada célula e o roteamento feito a partir dos interruptores é possível criar diversos circuitos digitais (CHU, 2008). Além de que essas células são reprogramáveis, assim, o dispositivo pode, em alguns casos, trocar de função após e até mesmo durante a sua utilização (AMD, 2023a).

As células lógicas, geramente, possuem uma *Look-Up Table* (LUT), um conjunto de flip-flops tipo D, multiplexadores e unidades de carry, sendo possível implementar circuitos combinacionais e sequenciais a partir delas. Algumas famílias de FPGAs agrupam duas células lógicas para formar um *slice* e quatro *slices* para formar um *Configurable Logic Block*

(CLB) (CHU, 2008). As arquiteturas mais recentes, como mostra a Figura 2.5, possuem blocos com diversas funções como CLBs, *Block Random Access Memorys* (BRAMs) e *Digital Signal Processing* (DSP) bem como blocos para gerenciamento de *clock*.

A demanda por sistemas embarcados para uso específico e parametrizáveis levaram a criação de SoCs que integram FPGAs, processadores de software e vários outros periféricos.

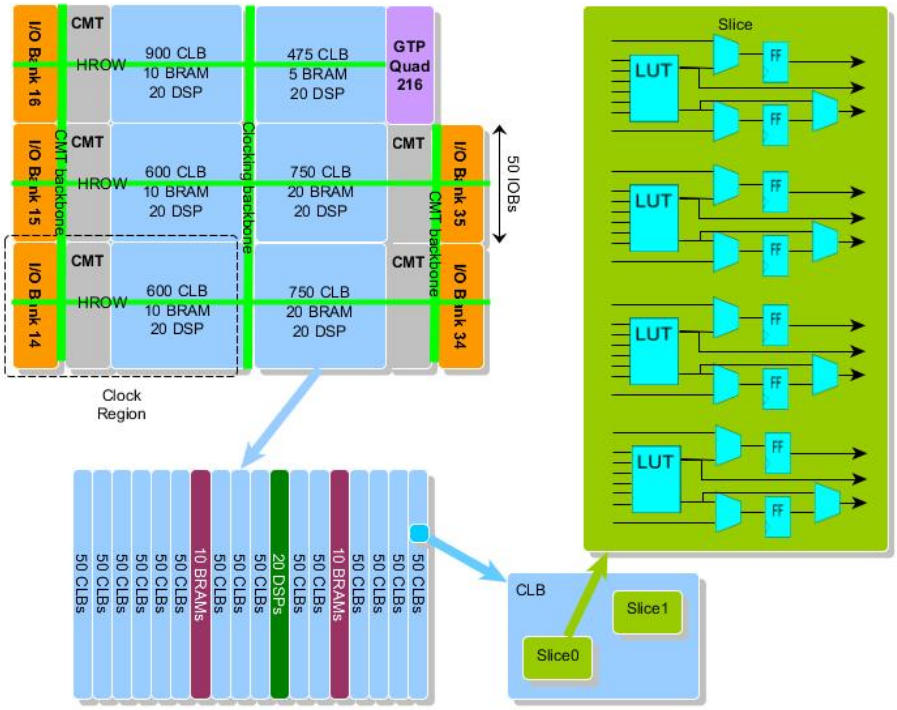

Figura 2.5 – Layout do FPGA Artix XC7A35T (EB, 2015).

A arquitetura *Zynq-7000*, mostrada na Figura 2.6, apresenta uma arquitetura poderosas que integra um processador ARM com um FPGA Artix-7. A arquitetura Zynq-7000 possui diversas interfaces de comunicação AXI e periféricos e recursos, tais como: suporte a memória DDR3, PCIe Gen 2, USB 2.0, SD/SDIO, interfaces de comunicação SPI, UART, I2C, e conversores analógico digital.

Os SoCs geraram uma demanda para a criação de projetos que integram hardware e software e, por causa dessa integração, estes projetos tornam-se altamente complexos, por esse motivo técnicas de *High-Level Synthesis* (HLS) se tornaram atrativas para executá-los.

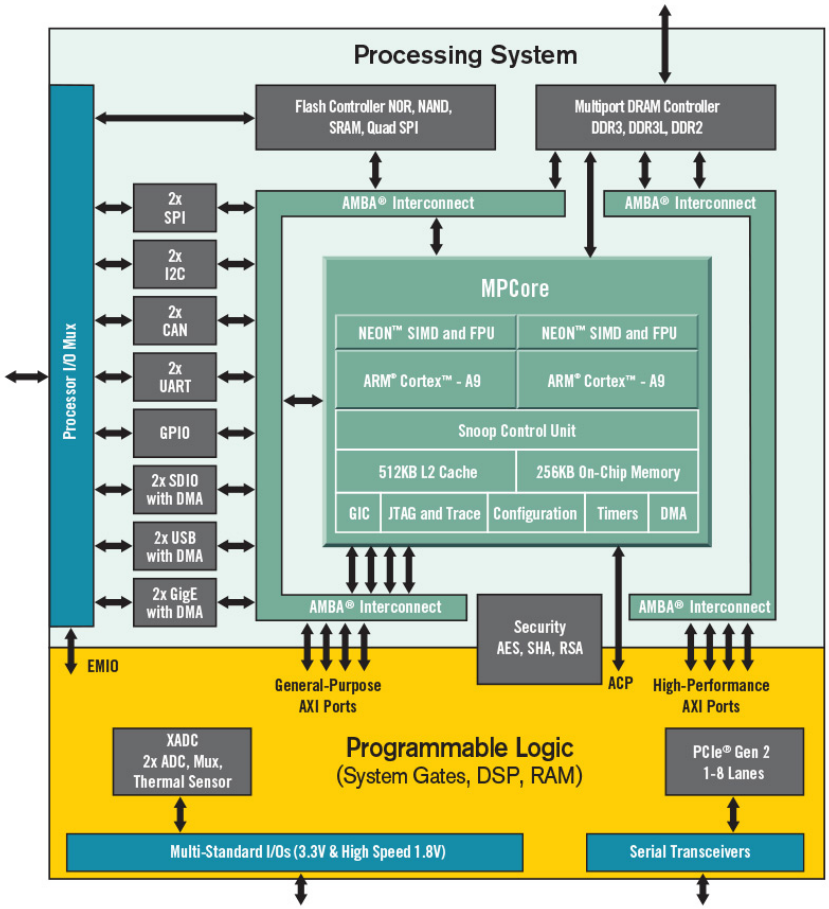

Figura 2.6 – Arquitetura *Zynq-7000* (AMD, 2023b).

2.4 Síntese de Alto Nível (HLS)

HLS são ferramentas que utilizam de uma descrição em alto nível de um sistema para gerar uma estrutura RTL Qquivalente. Um fluxo de projeto utilizando este tipo de ferramenta segue os seguintes passos: (a) criação do sistema em linguagem de alto nível; (b) verificação da funcionalidade do sistema; (9)c) utilização da ferramenta para gerar o RTL correspondente; (d) verificação do RTL gerado e a experimentação com diferentes arquiteturas a partir de um mesmo sistema (XILINX, 2023).

As ferramentas HLS geram a arquitetura RTL a partir de algoritmos genéricos e levam em consideração certos objetivos do projetista como a área utilizada, *throughput*, potência, dispositivo SoC, entre outros (PROST-BOUCLE; MULLER; ROUSSEAU, 2013). Assim, a HLS permite a reutilização de uma mesma arquitetura em diversas plataformas e configurações diferentes.

As tarefas realizadas por uma ferramenta HLS de forma geral são (COUSSY et al., 2009):

- Compilar a especificação: transformar a descrição do sistema em uma representação

formal;

- Alocação de recursos: define tipos e número de recursos necessários para satisfazer os objetivos do projetista;
- Agendamento (*schedule*) de operações: define a ordem em que as operações são realizadas, bem como devem ser realizadas em sequências ou paralelamente;
- Vinculação (*binding*): variáveis são vinculadas às unidades de armazenamento, operações vinculadas às unidades funcionais e ligações a multiplexadores e barramentos;
- Geração da arquitetura RTL: Implementação da arquitetura a partir de componentes RTL.

O HLS apresenta uma maneira automatizada e rápida para gerar arquiteturas RTL a partir de descrições de alto nível. O *Vitis HLS*, por exemplo, é uma aplicação capaz não somente de gerar a estrutura RTL como também integrá-la com processadores de software presentes nos SoCs, o que costuma ser feito usando o protocolo AXI4 utilizado pela Xilinx. O *Vitis HLS*, por exemplo, consegue utilizar o protocolo AXI para realizar o acesso a diferentes tipos de dados como está indicado na Figura 2.7. Assim ferramentas de HLS são muito importantes na implementação de co-projetos hardware/software pela capacidade de alcançar uma solução satisfatória em um tempo relativamente pequeno.

C-argument type	Paradigm	Interface protocol (I/O/Inout)
Scalar(pass by value)	Register	AXI4-Lite (`s_axilite`)
Array	Memory	AXI4 Memory Mapped (`m_axi`)
Pointer to array	Memory	`m_axi`
Pointer to scalar	Register	`s_axilite`
Reference	Register	`s_axilite`
hls::stream	Stream	AXI4-Stream (`axis`)

Figura 2.7 – Tipos de paradigma e protocolos AXI utilizados no *Vitis HLS* (XILINX, 2023).

2.5 Interfaces AXI4

AXI4 é um protocolo adotado pela AMD para realizar transferência de informações entre propriedades intelectuais diferentes. Sendo que existem três tipos de interfaces diferentes : AXI4, para requisitos de mapeamento de memória de alta performance, AXI4-Lite, similar ao AXI4, porém utilizado em baixas taxas de dados, e AXI4-Stream, utilizado quando é necessária alta velocidade de transferência de dados (AMD, 2017).

2.5.1 AXI4 e AXI4-Lite

As interfaces AXI4 e AXI4-Lite são protocolos mapeados em memória em que as transações ocorrem a partir do envio de um endereço de memória alvo, sendo diferente do procolo AXI4-Stream que é centrado no fluxo de dados com um canal unidirecional (AMD, 2017).

A AXI4 e AXI4-Lite são bem semelhantes com a capacidade de movimentar dados do dispositivo principal para o secundário e realizar o caminho contrário simultaneamente. Porém o AXI4-Lite somente consegue realizar uma transferência de dados por transação enquanto que a interface AXI4 consegue realizar até 256 transferências de dados (AMD, 2017).

As duas interfaces possuem cinco canais: canal de endereço de leitura, canal de endereço de leitura, canal de dados de leitura, canal de dados de escrita, canal de resposta de escrita. Estes são os canais que permitem a simultaneidade dessas interfaces, sendo que o AXI4 permite o uso de diferentes *clocks* entre diferentes sistemas interligados por AXI (AMD, 2017).

Logo, as interfaces AXI4 são um importante componente para interligar diferentes circuitos e sistemas processados.

2.6 Estado da Arte

Ferramentas geradoras de códigos são variadas em natureza e aplicações. A *Aldec*, por exemplo, possui a ferramenta *Active-HDL* que é capaz de receber entradas tanto de texto quanto gráficas, esquemáticos e diagramas de estados, para gerar códigos em HDL (WILSON, s.d.). Similarmente a ferramenta *Forsyde-SystemC* utiliza modelos de computaçã (MoCs) e técnicas de descrição formal de sistemas ciber-físicos, semelhantes ao estilo de fluxo de dados (*data-flow*), para gerar códigos em HDL a partir de descrição em SystemC (FORSYDE, s.d.). Por outro lado, a ferramenta *Vitis HLS* utiliza-se de códigos em linguagens de alto nível C ou C++ para gerar estruturas RTL (XILINX, 2023).

Na Universidade de Brasília foi desenvolvida uma ferramenta de uso geral focada em sistemas dinamicamente reconfiguráveis, chamada *RTR-Lib*. O RTR-Lib é uma aplicação baseada em *Matlab-Simulink* em conjunto com um repositório com vários componentes pré-caracterizados para gerar códigos VHDL, sendo que esta aplicação se destaca por possuir suporte ao protocolo de comunicação AXI4-Lite usando dispositivos *Zynq-7000* da Xilinx (IVO; MUÑOZ, D. M., 2019).

Além de alguns geradores de aplicação geral, existem também vários geradores de aplicação específica. Uma área de aplicação é em geradores de controladores *fuzzy*, o mais conhecido da área é o *Xfuzzy* que utiliza uma plataforma gráfica para auxiliar na síntese

de lógica *fuzzy* em hardware e software (LOPEZ et al., 1998). Outro caso interessante é o *FLOA*, desenvolvida na Universidade de Brasília (UNB), que se utiliza de uma interface gráfica criada no *Matlab* para obter parâmetros de controladores *fuzzy Takagi-Sugeno* e, assim, gerar códigos VHDL que implementam este controlador, sendo que esta aplicação se destaca pela complexidade da interface gráfica utilizada e a capacidade de parametrização dos controladores (JESUS, 2017).

Uma aplicação que potencialmente pode se beneficiar da implementação de geradores de códigos são algoritmos de otimização bioinspirados porque apresentam alto potencial de paralelismo, visto que em sua concepção são utilizadas partículas, *swarm units*, que podem executar seus cálculos paralelamente (ARBOLEDA et al., 2009). Sendo que as estruturas desses algoritmos podem ser implementados tanto parcialmente em FPGAs quanto completamente para acelerar os cálculos (MUÑOZ, D. M.; LLANOS et al., 2010).

Uma grande porção dos geradores de código desenvolvidos na UnB são relacionados à *Machine Learning* (ML) devido à estrutura regular que estas possuem e a necessidade de acelerar as operações que envolvem ML. A ferramenta *vRBFgen* desenvolvida no *Matlab* que a partir da topologia e a largura de bits do ponto flutuante de uma rede *Radial Basis Function Neural Network* (RBFNN), gera automaticamente o código VHDL correspondente (AYALA et al., 2017). Outro trabalho, que trata de iterfaces miocinéticas para controle de próteses de mão robóticas (*magnetic tracking*) inclui um gerador de regressores lineares chamado *pLinRgen* que gera o código VHDL da estrutura linear do problema proposto utilizando uma combinação de arquiteturas *pipeline* e FSMs, podendo ser configurada com diversos parâmetros (MENDEZ et al., 2022).

Finalizando, outra aplicação semelhante é a *DNNBuilder* que recebe os arquivos de definição de uma *Deep Neural Network* (DNN) em conjunto com seus pesos para gerar aceleradores em hardware para a respectiva rede (ZHANG et al., 2018).

Portanto, a geração de códigos de HDLs é bastante explorada. Sendo que a maioria dos geradores são para aplicações específicas, mas há também uma boa quantidade de geradores de uso geral. No entanto, a maioria desses geradores são ou para uso da indústria ou para uso acadêmico, assim, percebe-se uma carência de geradores para o uso didático. Por fim, a tabela 2.1 compara diversos geradores em relação aos seus recursos.

Tabela 2.1 – Comparação entre geradores de códigos.

Autor (ano)	Nome da Ferramenta	Descrição gráfica	Geração de FSM	Geração RTL	Geração de testbench	Comunicação com interface AXI	Observações
Xilinx(2019)	Vitis HLS	Não	Sim	Sim	Não	Sim	Ferramenta de uso industrial e proprietária.
Aldec(1997)	Active-HDL	Sim	Sim	Sim	Não	Não	Ferramenta de uso industrial e proprietária.
ForSyDe(2011)	ForSyDe-SystemC	Sim	Não	Não*	Não	Não	Apesar de não gerar RTL, gera estrutura de fluxo de dados.
Ivo(2019)	RTRLib	Sim	Não	Sim	Não	Sim	Ferramenta de uso acadêmico com foco em reconfiguração dinâmica.
López(1998)	Xfuzzy	Sim	Não	Sim	Não	Não	*Ferramenta com foco em lógica fuzzy.*
Jesus(2017)	FLOA	Sim	Não	Sim	Não	Não	*Ferramenta com foco em controladores fuzzy.*
Este trabalho	Hilda	Sim	Sim	Sim	Sim	Sim	Ferramenta para uso didático.

3 Metodologia

Este capítulo apresenta primeiramente as ferramentas computacionais usadas no desenvolvimento do trabalho e os kits de desenvolvimento FPGA para realização de testes físicos. Em seguida se apresentam os métodos e técnicas usadas no desenvolvimento de cada ferramenta da aplicação proposta, sendo que foi empregado um cuidado especial para tentar respeitar a indentação e criar um código legível.

3.1 Ferramentas utilizadas neste trabalho

Este trabalho se apoia especialmente diversas ferramentas de software livre como o Draw.io, Godot e o GTKWave. Mas utiliza, também, algumas plataformas e placas proprietárias que fazem parte do fluxo de projeto da eletrônica digital.

3.1.1 Draw.io

Draw.io é uma ferramenta gratuita e *open-source* para a criação de diagramas de posse e desenvolvida pela JGraph Ltd e Draw.io AG. Ela é usada como base deste trabalho porque possui uma grande variedade de elementos utilizáveis, além de permitir exportar seus diagramas em formato .XML.

3.1.2 Godot

Godot é uma *game engine* gratuita e *open-source* que pode ser utilizada para a criação de Interfaces de Usuário (UI). Godot possui diversos elementos utilizados em UI como janelas de escolha de arquivos, botões, campos de entrada de texto, entre outros. Uma das principais vantagens do *Godot* é que ele possui a capacidade de gerar executáveis de seus programas para vários sistemas operacionais como o Linux, Windows e mobile, permitindo criar aplicações multiplataforma.

3.1.3 Vivado

Vivado é uma aplicação da *AMD-Xilinx* capaz de, a partir de *Hardware Description Languages* (HDLs), implementar de circuitos digitais combinacionais e sequenciais, desenvolver co-projetos hardware/software, e programar SoCs FPGAs. O Vivado apresenta também uma série de blocos de propriedade intelectual (IPs) da *Xilinx* que podem facilmente ser utilizados. Todas as placas de desenvolvimento utilizadas neste trabalho usam SoCs FPGAs da Xilinx, portanto, é necessário utilizar o Vivado.

3.1.4 Placa de Desenvolvimento Basys3

A Basys3 é uma placa de desenvovlvimento FPGA da Digilent que é normalmente utilizada para fins educacionais devido aos recursos didáticos tais como chaves, botões, leds, displays de sete segmentos, porta VGA, entre outros. Possui um FPGA Artix-7 da Xilinx com 33280 CLBs, 20800 LUTs, 41600 FFs, 1800Kbits em blocos de RAM e 90 DSPs. A Figura 3.8 mostra a imagem de uma placa Basys3 e seus diversos recursos.

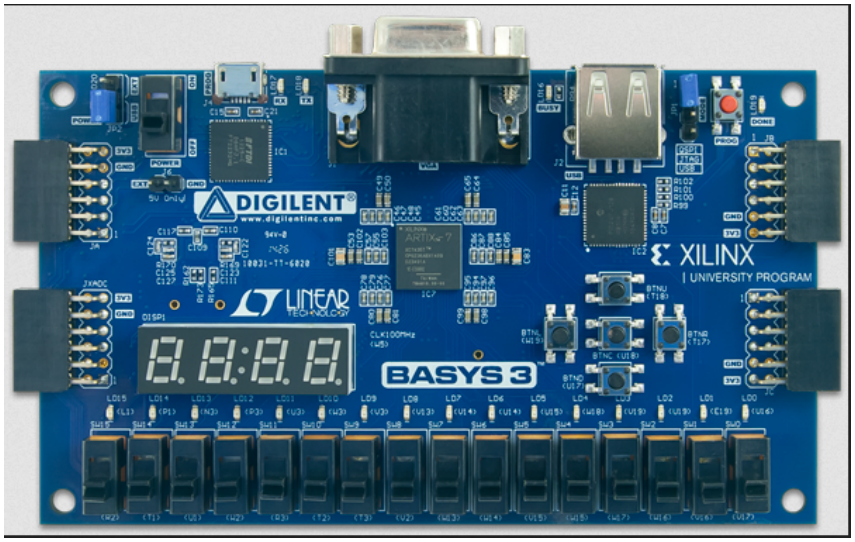

Figura 3.8 – Exemplo de uma placa *Basys3* (DIGILENT, 2023).

3.1.5 Placa de Desenvolvimento Zedboard

O kit de desenvolvimento Zedboard da Avnet, como mostra a Figura 3.9, é um SoC com muitos recursos que será utilizado para o desenvolvimento do que tange à integração de software e hardware deste projeto. Alguns de seus recursos são listados abaixo:

- Chip Zynq 7020 com dois processadores ARM Cortex A9 e FPGA Artix 7;
- 512 MB DDR3;
- 8-bit VGA;
- 8 interruptores, 7 botões, display LCD;
- Conversor XADC de oito canais e 100 MSPS;
- interfaces USB 2.0, HDMI e de entrada e saída de audio;
- USB-UART;
- Porta Ethernet;
- Suporte a PetaLinux BSP.

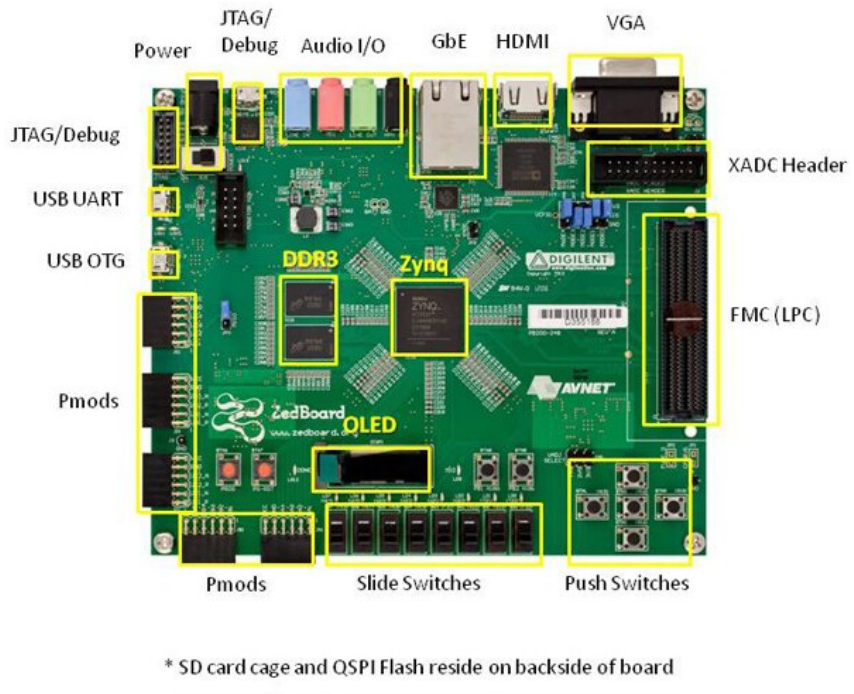

Figura 3.9 – Placa Zedboard da Avnet (AVNET, 2023).

3.1.6 GTKWave

GTKWave é um visualizador de ondas utilizado para depurar simulações em VHDL e Verilog, que possui suporte a diversos formatos de onda. Neste projeto, o GTKWawe foi utilizado para visualizar as ondas resultantes das simulações utilizando o gHDL que salva os formatos de onda em um arquivo GHW.

3.2 Biblioteca de Elementos no Draw.io

Para começar o desenvolvimento das ferramentas primeiramente é necessário convencionar como serão feitos os diagramas que irão descrever máquinas de estados finitos e projetos RTL. Assim, abaixo são descritos como foram selecionados os diversos elementos utilizados nas ferramentas vFSMgen e vRTLgen.

3.2.1 Elementos utilizados na ferramenta vFSMgen

O gerador de FSMs utiliza pouquíssimos elementos. Sendo necessário somente um elemento para os estados, um bloco para designar os valores das saídas de cada estado, no caso das FSMs do tipo Moore, e um elemento para designar o valor de cada variável de entrada. Sendo também necessário um elemento para indicar as transições dos estados, bem como as saídas no caso das FSMs do tipo Mealy.

Para os estados foram escolhidos os elementos *Retângulo*, incluindo o *Quadrado*, e

Círculo, todos presentes no *Draw.io*. Estes dois objetos foram escolhidos porque essas são as figuras mais utilizadas, no geral, neste tipo de diagrama. A Figura 3.10 mostra um exemplo de diagrama de estados utilizando o *Quadrado*.

Para indicar as saídas de cada estado é utilizado o elemento *Lista*. Cada lista é identificada pelo seu nome, que deve ser igual ao nome do estado ao qual ela está se referindo. Cada um de seus elementos indica um valor de saída da FSM e eles são atribuídos utilizando quase a mesma sintaxe de atribuição do VHDL (‘nome’<=‘valor’) como mostra a Figura 3.10, sendo que não é necessário terminar a atribuição com o ponto e vírgula.

As *Listas* são muito versáteis, podendo exercer várias funções ao atrelar o nome da lista à uma função específica. Uma dessas funções é identificar os valores de saída. Na Figura 3.10 há uma lista com o nome *Variables* que tem a função de indicar os nomes das entradas das FSMs. Outra função é designar o nome das saídas do tipo *Mealy*, esta, no caso, é atrelada ao nome *Variables_Mealy*. Estas duas funções serão aprofundadas posteriormente.

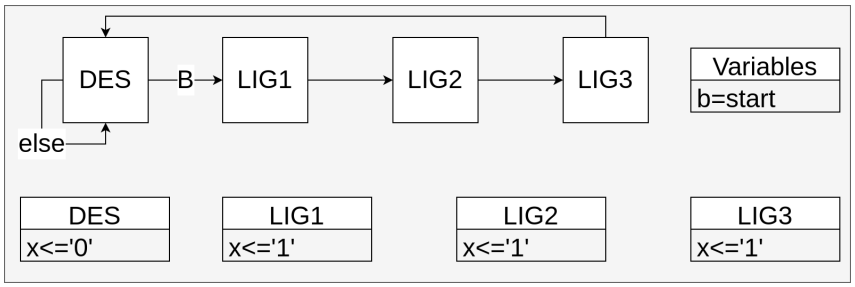

Figura 3.10 – Exemplo de uma FSM simples do tipo Moore.

A *Linha* indica a transição de um estado para outro. Nos rótulos das *Linhas*, as variáveis que condicionam as transições são indicadas por uma única letra, se a letra for minúscula, a transição ocorrerá quando o valor da variável for ‘0’ e quando a letra for maiúscula ocorrerá quando o valor for ‘1’, e uma linha sem rótulo indica uma transição não condicionada. Os estados que podem transicionar para mais de um estado necessariamente precisam de uma transição do tipo *else*, obrigando, assim, o projetista a contemplar todas as possibilidades. Na figura 3.10, por exemplo, a transição de *DES* para *LIG1* ocorre quando o valor de ‘B’ é igual a ‘1’, nos outros casos a FSM continua no estado DES.

As transições entre estados, muitas vezes, são compostas por operações lógicas mais complexas oriundas da combinação de operadores ‘e’ e ‘ou’. No diagrama o símbolo ‘&’ representa a lógica ‘e’ e o símbolo ‘|’ representa a lógica ‘ou’, sendo também possível utilizar parênteses. A Figura 3.11 exemplifica um diagrama com transições de maior complexidade.

Até então foram mostrados somente exemplos de FSMs do tipo *Moore*, mas em algumas situações são necessárias as FSMs do tipo *Mealy* e nesse caso o valor das saídas são definidas nas transições entre estados. Esse tipo de saída é implementado no rótulo das *Linhas*, sendo que elas são separadas da lógica de transição utilizando uma barra (‘/’),

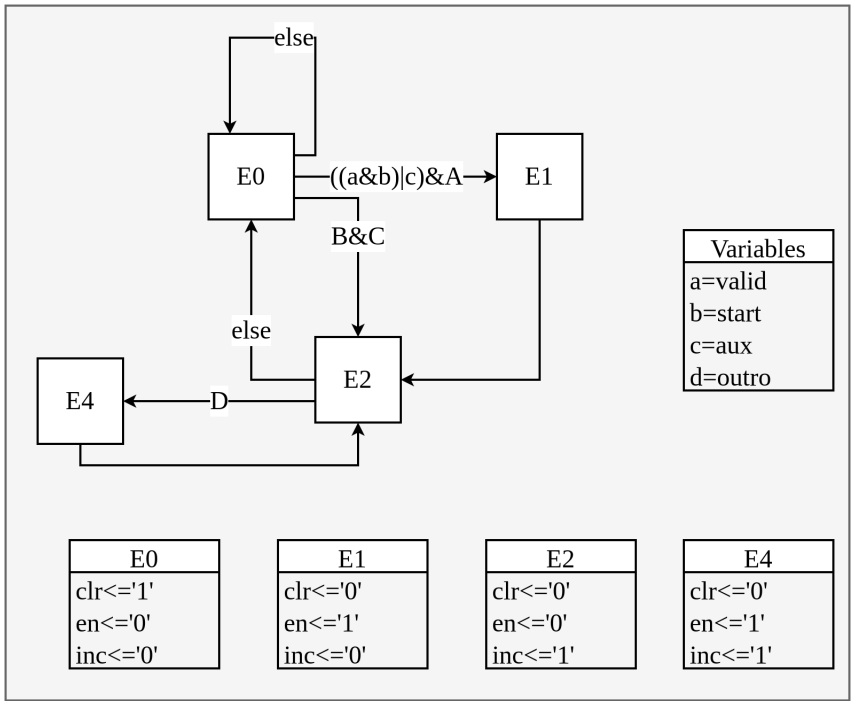

Figura 3.11 – Exemplo de uma FSM do tipo Moore com transições mais complexas.

assim, de formal geral, o rótulo é representado por 'lógica de transição '/'saídas mealy', como mostra a Figura 3.12.

A partir da Figura 3.12 também é possível notar como as *Listas* são utilizadas para definir os nomes das variáveis que aparecem nas transições. Tanto os nomes das variáveis utilizadas na lógica de transição pela Lista denominada *Variables* quanto das variáveis utilizadas nas saídas das FSMs do tipo *Mealy*, designada por *Variables_mealy*. O valor que está na esquerda é trocado pelo valor que está na direita ao gerar o código, assim, as variáveis que aparecem nos rótulos das *Linhas* são somente auxiliares.

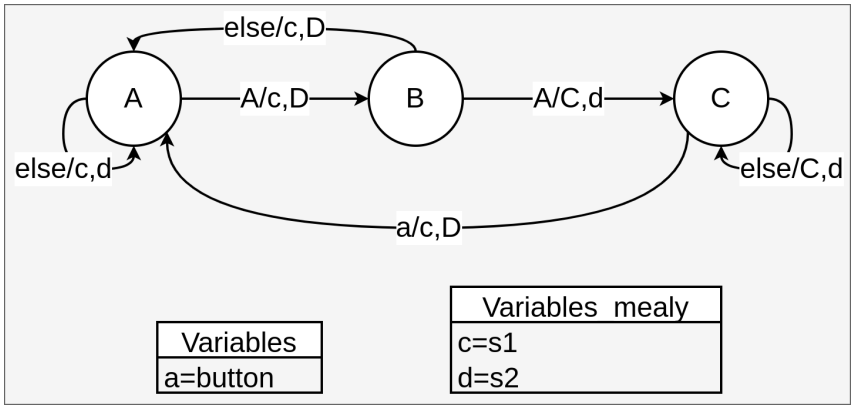

Figura 3.12 – Exemplo de uma FSM do tipo Mealy.

3.2.2 Elementos utilizados na ferramenta vRTLgen

O gerador RTL apresenta uma quantidade maior de elementos em função de sua complexidade. Então, para reproduzir um diagrama de um projeto RTL é preciso de elementos que representem:

- Entradas, saídas e constantes;
- Instanciação de componentes provenientes de códigos VHDL;
- Geradores de códigos;
- Listas de constantes e tipos;
- Ligações entre diferentes componentes e geradores;

3.2.2.1 Entradas e saídas

A descrição de um circuito em VHDL começa pela entidade, nesta são definidas as portas de entrada e saída dos circuitos, logo, é necessário definir elementos no diagrama para representar essas portas. Tal elemento deve ter a capacidade de incorporar as características de uma porta, ou seja, um nome, modo e um tipo.

Este trabalho abrange somente os modos *in* e *out*, sendo que foram escolhidos um elemento do Draw.io para cada um dos modos. Para o modo *in* foi escolhido o elemento DAC e o elemento Delta para o modo *out*, no entanto, o elemento gráfico somente indica o modo, o nome e o tipo são indicados pelo texto do elemento separados por uma barra. A Figura 3.13 apresenta um circuito com duas entradas destacadas em vermelho, uma saída destacada em roxo e em amarelo um gerador de código simples, que gera uma porta lógica 'e' entre as duas entradas.

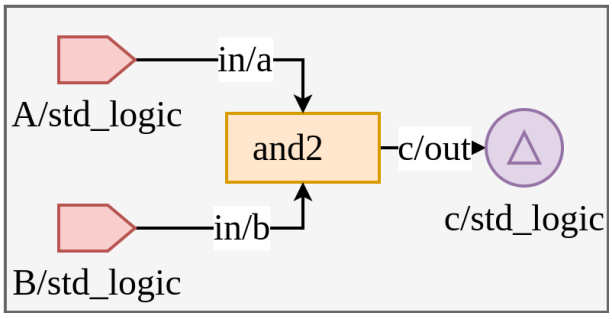

Figura 3.13 – Exemplo de um esquemático simples.

3.2.2.2 Geradores de códigos

Os geradores de código utilizados nesse trabalho são recursos que precisam ser parametrizáveis, assim, além do nome do gerador escolhido, é preciso que os geradores possam receber uma quantidade variável de parâmetros.

Para os geradores de código foi escolhido o bloco *Retângulo*, como mostra o retângulo laranja da Figura 3.13. O texto do bloco contém, primeiramente, o nome do gerador em Python escolhido, que nesse caso gera uma função ‘e’ de duas entradas, e em seguida, seus diversos argumentos divididos por barras, quando há argumentos.

3.2.2.3 Componentes VHDL

A ferramenta *vRTLgen* precisa ter a capacidade de instanciar componentes descritos a partir códigos VHDL, assim, o elemento escolhido foi o *Dual In-Line IC* pelo formato de circuito integrado do elemento. O nome do componente é o nome da instância de componente. A Figura 3.14 mostra um exemplo de um circuito com um componente instanciado destacado em azul e além dos elementos anteriormente discutidos, a Figura 3.14 também apresenta o elemento *Generic Component*, em verde, que representa uma constante na entrada de um componente ou gerador de código.

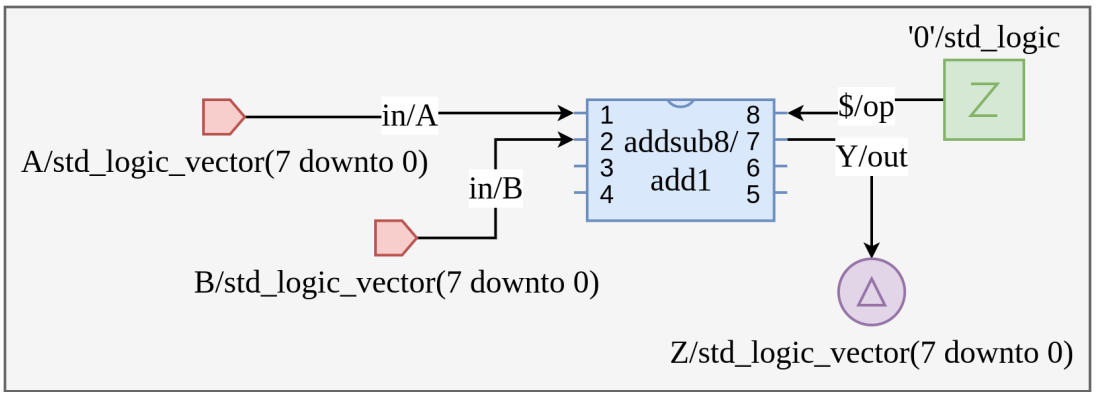

Figura 3.14 – Exemplo de um esquemático utilizando a instanciação de componentes.

3.2.2.4 Listas de constantes e tipos

Em códigos VHDL é muito importante a capacidade de criar constantes para fins específicos e também tipos personalizados que se adequem à aplicação que está sendo desenvolvida. Ambos os casos podem ser implementados a partir de Listas, pois elas podem assumir diferentes funções como mostrado na seção 3.2.1. A função de cada lista é definida por seu nome, para tipos é usado *Types_list* e para constantes é usado o nome *Constant_list* (vide Figura 3.15).

O funcionamento das duas listas é bem simples, a lista de constantes recebe o nome, tipo e valor, separados por sinais de igual. A lista de tipos recebe o nome do tipo e os valores que ele pode receber separados por um sinal de igual.

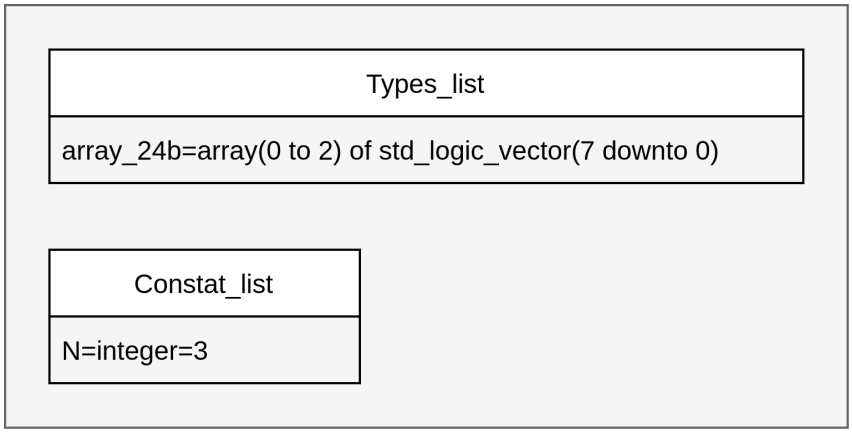

Figura 3.15 – Listas de constantes e tipos.

3.2.2.5 Conexões entre diferentes elementos

O Draw.io é uma ferramenta de uso geral para a criação de diagramas, então foi necessário criar uma forma de indicar as ligações entre os componentes de maneira que seja possível distinguir a origem e o destino de cada dado, ou seja, distinguir o fluxo de dados. É importante pontuar que as ligações funcionam da mesma forma para instâncias e geradores.

Como visto nos exemplos anteriores, é utilizada a *Linha* para conectar todos os elementos, sendo que o fluxo dos dados é apresentado em seu rótulo e devem ser representadas conexões entre:

- Entradas e componentes;
- Componentes e saídas;
- Constantes e componentes;
- Componentes entre si.

O fluxo dos dados é descrito no rótulo da *Linha* sempre usando o formato "'fonte'/'destino'". Quando um sinal tem origem em uma entrada a fonte deve ser necessariamente especificada como 'in' e quando o sinal se destina a uma saída o destino deve ser necessariamente 'out', como ilustra a Figura 3.13. Outro caso ocorre quando a entrada de um componente é uma constante, nesta situação a fonte deve necessariamente ser '$' como mostra a Figura 3.14. Por fim, há a conexão entre componentes, a fonte deve receber o nome da porta do componente de origem e o destino deve receber a porta do componente de destino como demonstrado na Figura 3.16, onde a saída de um somador é a entrada de outro.

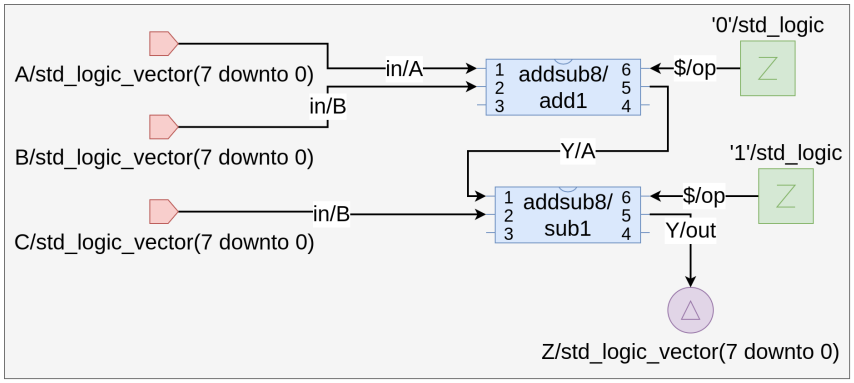

Figura 3.16 – Exemplo de dois somadores/subtratores de 8 bits interligados.

3.3 Desenvolvimento do vFSMgen

A vFSMgen é uma ferramenta pensada para realizar tanto a geração de FSMs do tipo Mealy quanto Moore, assim foi necessário criar um código capaz de criar os dois tipos de FSM. Esta seção apresenta o desenvolvimento da ferramenta em detalhes.

3.3.1 Desenvolvimento do *parser*

É possível exportar os diagramas do Draw.io em um formato .xml ou .drawio. Os dois arquivos possuem, basicamente, a mesma estrutura, mas a partir de arquivos .xml é possível exportar cada diagrama separadamente. Assim, um *parser* pode ser desenvolvido tanto para diagramas individuais quanto para um conjunto de diagramas, extraindo todas as informações dos diagramas. Dado que a ferramenta HiLDa utiliza arquivos .xml para os diagramas, é necessário entender sua estrutura.

Os arquivos .xml contêm informações que são utilizadas para determinar:

- Um identificador próprio de cada elemento;
- Elementos presentes dos diagramas;
- Função de cada elemento;
- Conexões entre elementos.

Os diagramas são compostos por etiquetas ou *tags*, em estruturas hierárquicas, que podem conter textos e atributos. A partir destes textos e atributos o *parser* obtém as informações necessárias. As etiquetas de cada elemento dos diagramas possuem o nome *mxCell* em que são presentes os seguintes atributos:

- **Id**: Apresenta uma identificação única para cada elemento;

- **Value**: Indica o texto contido no elemento. Este elemento pode indicar tanto o nome de um estado quanto representar a função de uma lista;
- **Style**: Define qual elemento está sendo utilizado pela propriedade “shape". No código 3.1, linha 8, há a definição de um componente do tipo *circuito integrado* ;
- **Source** e **target**: Em uma linha indica qual a fonte da linha e o seu destino final;
- **Parent**: Indica se o elemento pertence a outro elemento e especifica o Id do elemento pai.

Código 3.1 – Exemplo de código .xml exportado do Draw.io.

```
1  <mxGraphModel dx="800" dy="531" grid="1" gridSize="10" guides="1"
       tooltips="1" connect="1" arrows="1" fold="1" page="1"
       pageScale="1" pageWidth="1169" pageHeight="827" math="0"
       shadow="0">
2    <root>
3      <mxCell id="0" />
4      <mxCell id="1" parent="0" />
5
6      <mxCell id="VNDJfdysLbIDVn8irZk3-1"
7          value="IC"
8          style="shadow=0;dashed=0;align=center;html=1;strokeWidth=1;
9                 shape=mxgraph.electrical.logic_gates.dual_inline_ic;
10                labelNames=a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
11                whiteSpace=wrap;"
12         vertex="1"
13         parent="1">
14
15       <mxGeometry x="340" y="110" width="100" height="200"
             as="geometry" />
16     </mxCell>
17   </root>
18 </mxGraphModel>
```

O *parser* utiliza a biblioteca xml presente no *Python* para abrir os arquivos xml e ler seu conteúdo. Em seguida, cada elemento do gráfico é analisado a partir do atributo *style* e, dependendo do valor, o elemento é adicionado a um dicionário, também conhecido como "vetor associativo", relativo ao seu elemento. A associação de cada valor e sua respectiva lista é ilustrada na Tabela 3.2.

A Figura 3.17 mostra como o *parser* inicialmente separa cada elemento do diagrama dependendo do valor encontrado em *style*. O valor "whiteSpace=wrap" indica um estado da FSM, e seu valor, que indica o nome do estado, é guardado em um dicionário com o seu *Id* como chave. Desta forma cada estado recebe um valor único que aponta somente para ele e independe do seu nome. Neste mesmo caso, em outro dicionário é estocado em uma lista

Tabela 3.2 – Identificadores de cada elemento do diagrama de FSMs.

Texto em Style	Elemento
whiteSpace=wrap	Estado
endArrow	Transição
swimlane	Listas de entradas e saídas
text	Valores de saída da FSM e nomes de variáveis

vazia que conterá as transições pertencentes aquele estado, sendo que a chave também é o *Id* do elemento.

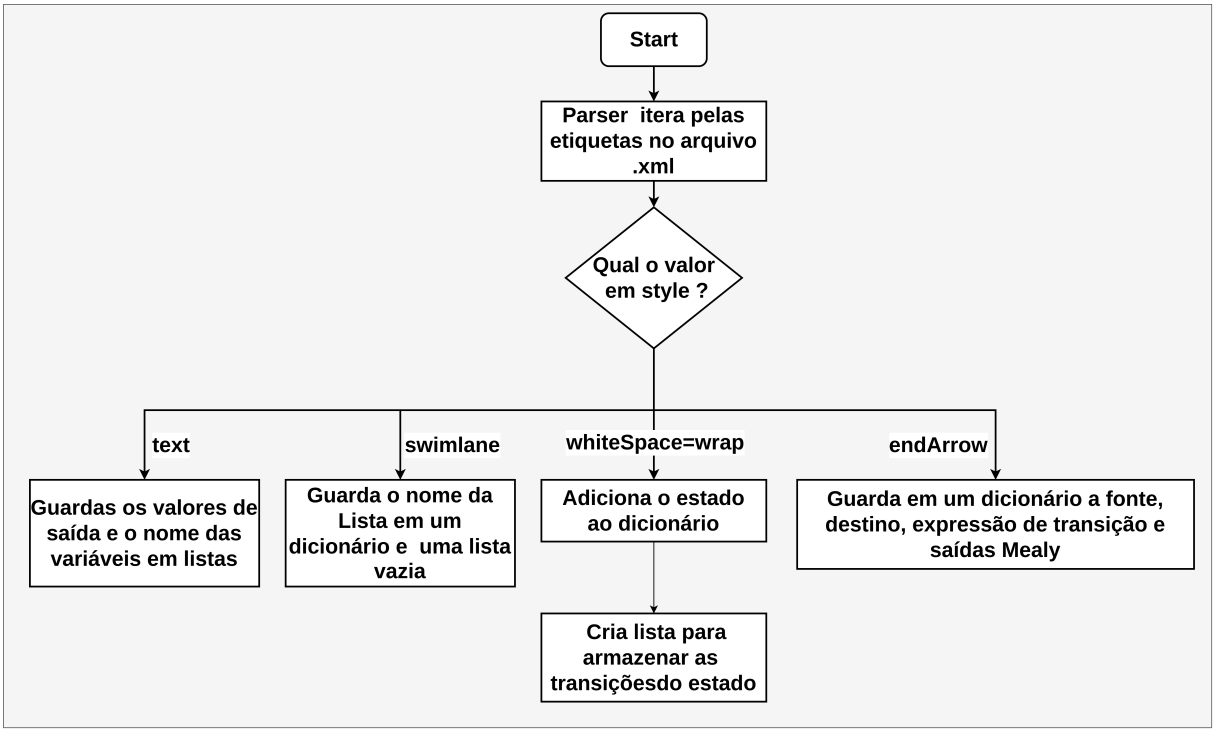

Figura 3.17 – Separação de cada elemento.

Para o valor "endArrow", que indica uma transição, é necessário guardar o *Id* do estado em que a transição se origina, o *Id* do estado alvo, a expressão de transição e os valores de saída para FSMs do tipo Mealy. Os *Ids* dos estados são encontrados nas etiquetas "source"e "target", enquanto que a expressão de transição e os valores de saída estão na etiqueta "value"separados por uma barra, caso haja saída *Mealy*. Neste caso, a chave é o *Id* único da Linha que aparece no gráfico.

O valor "swimlane"representa as *Listas* de nomes variáveis e saídas de estados para FSMs do tipo *Moore*.Sendo que a função de cada *Lista* é representada por seu nome, que é

utilizado para criar uma lista onde são estocados os nomes das variáveis e as saídas *Moore*.

O último elemento de valor "text"representa os textos contidos nas Listas e indicam os nomes das variáveis e saídas das FSMs tipo Moore. Neste caso, as listas vazias criadas anteriormente para cada elemento identificado por "swimlane"são finalmente preenchidas. Isso somente é possível porque cada um destes elementos irá possuir um valor na etiqueta "parent"que aponta para a cada lista, facilitando, assim a distribuição de cada elemento. Por fim, cada valor de entrada e saída é adicionado às listas independentes que serão utilizadas na criação da entidade do código VHDL.

Após separar todos os elementos, o programa distribui as transições para cada uma das listas de transições de cada estado. A lista recebe especificamente as transições que originam no estado especificado pelo ID da lista, ou seja, todas as transições que saem daquele estado. E, por fim, as saídas *Mealy* são separadas em um dicionário próprio para facilitar o processo de geração de código. Assim, o programa conterá os dados apresentados na Figura 3.18.

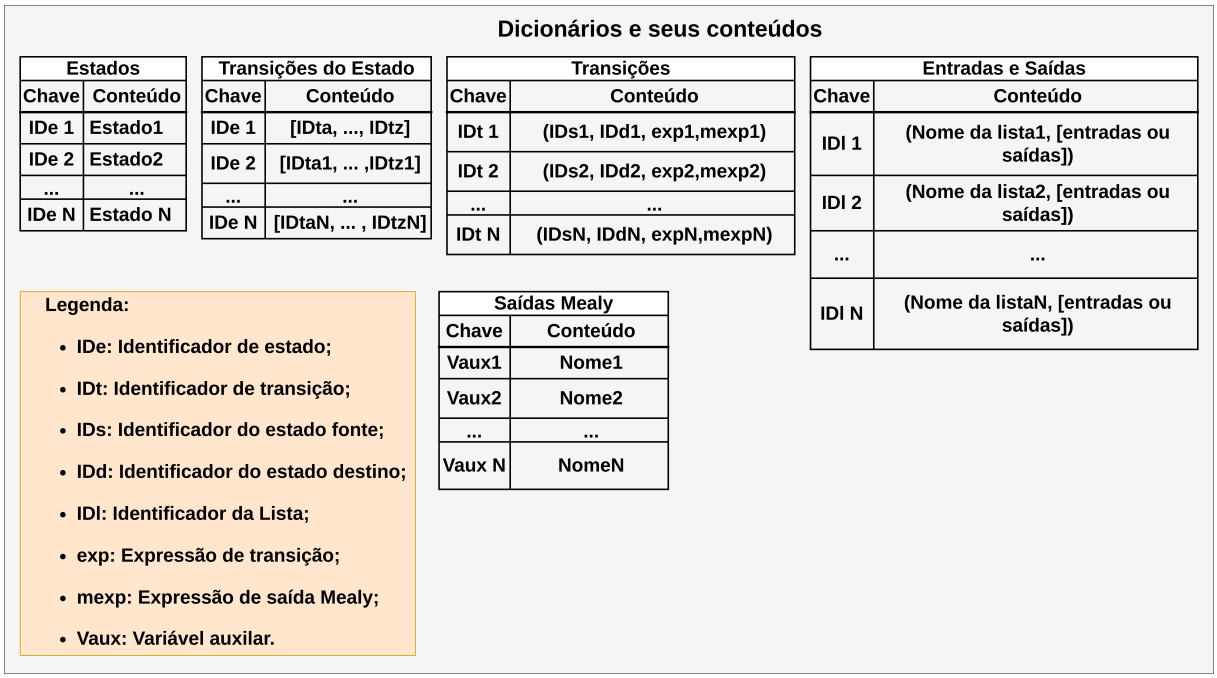

Figura 3.18 – Informações obtidas a partir do *parser*.

3.3.2 Geração da estrutura das FSMs

As FSMs possuem estruturas bem simples, sendo que elas possuem, na implementação deste trabalho, três estados muito bem definidos e neste caso o maior desafio para implementá-las é criar um *parser* que contemple tanto as FSMs do tipo Mealy quanto Moore, ao mesmo tempo.

O primeiro passo para montar o arquivo VHDL da FSM é a declaração de bibliotecas. Todas as FSMs do escopo deste trabalho utilizam somente a biblioteca "STD_LOGIC_1164"que

é adicionada no começo de toda FSM gerada.

Em seguida é montada a entidade. O nome da entidade é obtido a partir dos argumentos do programa, sendo o primeiro argumento o nome do arquivo .xml e o segundo o nome da FSM. Logo após, o programa adiciona a porta de *clock* e *reset* e todos os valores de entrada contidos na Lista *Variables* que estão estocados no dicionário de listas de entradas e saídas, como também adiciona os valores de output presentes na lista de outputs. O resumo do processo é mostrado na Figura 3.19.

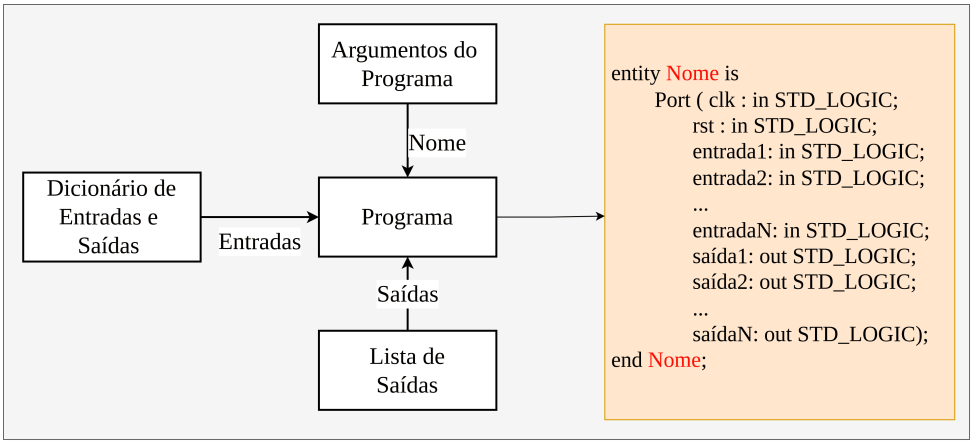

Figura 3.19 – Simplificação do processo de montagem da entidade.

Para montar a arquitetura, primeiramente são criados os tipos dos estados, adicionando os nomes dos estados obtidos. Em seguida são criados os sinais de estado atual e do próximo estado, sendo eles do tipo criado anteriormente.

Após a definição do tipo e a criação dos sinais começa a arquitetura, a qual é desenvolvida com três processos. O primeiro processo cuida do registro dos estados, este possui na lista de sensitividade somente o *clock* e o *reset* e quando o *reset* recebe o valor ‘1’ a máquina de estados reseta, e, nos casos que o *reset* não está ativado, a cada ciclo de *clock*, o estado atual recebe o próximo estado.

O segundo processo cuida da transição de estados e recebe as entradas da máquina de estados em sua lista de sensitividade. O programa cria uma estrutura "CASE-WHEN"e itera por todos os estados no dicionário de estados, adicionando-os ao "CASE-WHEN", sendo que para cada estado são adicionadas as transições presentes no dicionário de transições de estado. Para isso, é criada uma estrutura "IF-ELSE"para cada estado com mais de uma transição, inclusive utilizando "ELSIF"para estados com 3 ou mais transições, e os argumentos destes são as expressões de transição presentes no diagrama, porém traduzidas para VHDL. Por fim, é adicionada a condição "When others"que transiciona por padrão para o estado inicial evitando, assim, a criação de *latches*.

O terceiro e último processo é a lógica de saída do circuito que possui a lista de sensitividade igual ao processo anterior. Novamente é criada uma estrutura "CASE-WHEN"e para cada estado são adicionadas as saídas. Primeiramente são adicionadas as saídas Moore

presentes no dicionário de entradas e saídas, conforme apresentado na Figura 3.18. Isso é feito a partir do nome da lista que deve ser o mesmo nome do estado. Em seguida são adicionadas as saídas *Mealy*, as quais são colocadas dentro de estruturas "IF-ELSE", pois dependem das entradas, que são iguais aquelas presentes na lógica combinacional de transição de estados. Os valores de cada saída são obtidos a partir do dicionário de saídas *Mealy*, sendo que uma letra minúscula indica uma saída '0' e uma letra maiúscula indica uma saída '1'. Por fim, é adicionada a condição "When others"que apresenta todas as saídas zeradas evitando *latches*. Por último, a arquitetura é devidamente finalizada e o programa é escrito em um arquivo VHDL com nome igual ao nome da entidade.

A Figura 3.20 apresenta um resumo do fluxo de funcionamento do programa para gerar a arquitetura de hardware de uma FSM. É importante salientar que durante a execução do programa o código é progressivamente montado em uma *string* e no final essa *string* é colocada no arquivo VHDL.

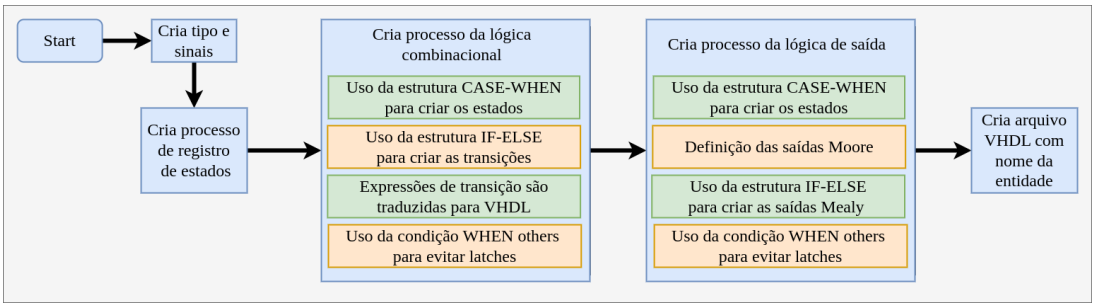

Figura 3.20 – Simplificação do processo de montagem da arquitetura.

3.3.3 Exemplo de FSM

A Figura 3.21 mostra a uma FSM simples com componentes tanto de Moore quanto de Mealy. Após exportar o diagrama e rodar o comando "python3 main.py <nome do arquivo>.xml <nome da FSM>", deve aparecer a mensagem "*File ready! Please check the reset state as it is randomly chosen*", que indica que a geração do arquivo foi bem sucedida e avisa que um estado aleatório foi escolhido para ser o primeiro estado.

Percebe-se, a partir do código A.1, que foram geradas três entradas e três saídas, sendo duas entradas o *reset* e o *clock*, e a terceira a entrada *button* que está indicada no diagrama pela letra A. Todos os estados A, B, e C aparecem na definição do tipo. As transições de estados estão de acordo com o diagrama, bem como as saídas da FSM.

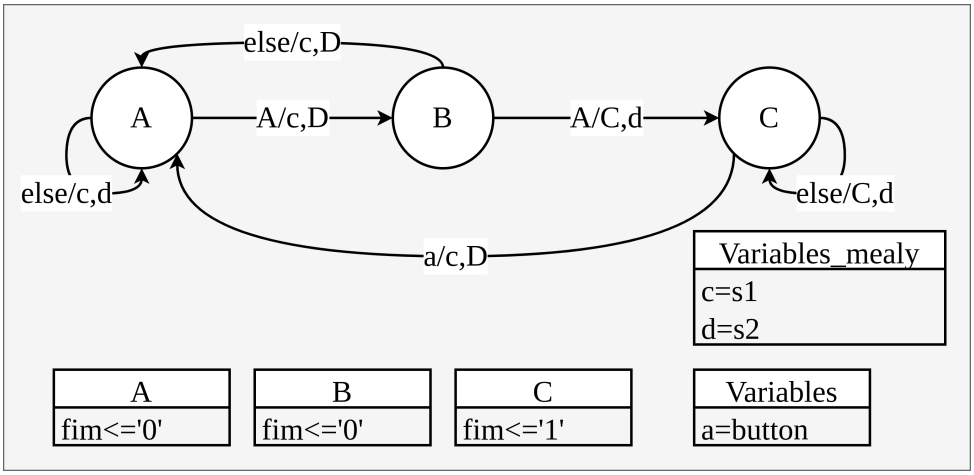

Figura 3.21 – Exemplo de uma FSM simples.

3.4 Desenvolvimento do vRTLgen

3.4.1 Bibliotecas de componentes e geradores

O gerador de projeto RTL depende de códigos VHDL e de geradores de códigos preexistentes. Os arquivos VHDL devem ser salvos em uma pasta chamada "component_files", enquanto que os geradores de códigos devem estar em uma pasta chamada "generator_files". Ao gerar o arquivo será criada uma pasta com o nome do diagrama escolhido contendo os arquivos VHDL necessários, sendo importante pontuar que *packages* devem ser adicionados manualmente no Vivado.

O gerador de códigos possui uma estrutura fixa como mostra o Código 3.2. A estrutura é composta pela inicialização, linha 2, onde são criados dois dicionários com o nome das entradas e o tipo de cada entrada. Em seguida há uma rotina de inicialização, linha 6, que possui a função principal de mudar os valores dos dicionários de entrada e saída a partir dos parâmetros passados pelo usuário, ou seja, é possível parametrizar entradas e saídas. Por fim, na linha 11, há a rotina de geração de códigos, que também recebe parâmetros do usuário para parametrizar a geração, sendo que o código deve ser estocado em uma *string* e no fim da função este deve ser retornado.

Código 3.2 – Estrutura do gerador de código.

```
1  class GenericGenerator():
2      def __init__(self):
3          self.input_ports={'a1':'tipo1', 'a2':'tipo2','a3':'tipo3'}
4          self.output_ports={'b1':'tipo3', 'b2':'tipo4','b3':'tipo5'}
5
6      def initialization(self,parameters=[]):
7          # Rotina de inicialização adicionada pelo usuário.
8          ...
9          pass
10
11     def generic_func(self,parameters=[], inputs={},outputs={}):
```

```
12          entity = '' # String que conterá o código do gerador
13          ...
14          return entity # A string deve ser retornada
```

O código 3.3 é um exemplo de gerador de código que cria uma porta *E* de duas entradas. O resultado será uma linha no código VHDL com o seguinte conteúdo "signal3<=signal1 and signal2;". Importante pontuar que esse é um exemplo genérico, na prática os sinais seriam indicados pelas ligações no diagrama.

Código 3.3 – Gerador que cria uma porta "E"de duas entradas.

```
1  class And2Class:
2      def __init__(self):
3          self.input_ports = {'a':'std_logic', 'b':'std_logic'}
4          self.output_ports = {'c':'std_logic'}
5      def initialization(self,parameters=[]):
6          pass
7      def
          and2(self,parameters=[],inputs={'a':'signal1','b':'signal2'}
8  ,outputs={'c':'signal3'}):
9          entity = '\n\n'
10         entity += outputs['c']+'<='+inputs['a']+" and
              "+inputs['b']+';\n\n'
11         return entity
```

Por fim, cada gerador de código deve ser adicionado ao arquivo em Python chamado "General_generator_file.py". A estrutura deste arquivo é mostrada no código 3.4. O código do gerador deve ser importado no começo do arquivo e sua classe declarada na inicialização da classe principal. Em seguida, deve-se criar uma função com o nome do arquivo que será chamada pelo código principal, sendo que essa função tem o objetivo de informar as portas utilizadas no gerador para o programa principal inicializar o gerador e gerar o código VHDL.

Código 3.4 – Estrutura do arquivo que integra todos os geradores.

```
1  from generator_files.<nome do arquivo> import <nome da classe>
2
3  class GeneralGenerator:
4      def __init__(self):
5          self.<nome da classe> = <nome da classe>()
6
7      def <nome do arquivo>(self,parameters=[],inputs={},outputs={}
8                            ,showconfig=0):
9          if showconfig==1:
10             return (self.<nome da classe>.input_ports, self.<nome
                   da classe>.output_ports)
11         elif showconfig == 2:
12             self.<nome da classe>.initialization(parameters)
13         else:
14             return self.<nome da classe>.<nome da
                   função>(parameters, inputs, outputs)
```

3.4.2 *Parser* para o *vRTLgen*

O *parser* do *vRTLgen* é ligeiramente mais complexo que o *parser* do *vFSMgen* por causa da quantidade de elementos e situações que devem ser levados em consideração como discutido na seção 3.2.2.

O *vRTLgen* é capaz de gerar códigos a partir de diversos diagramas contidos em um mesmo arquivo .xml ou .drawio. Esses diagramas são gerados de forma sequencial e adicionados à biblioteca de arquivos VHDL, do primeiro à esquerda ao último à direita, assim, um diagrama pode utilizar circuitos gerados anteriormente.

O programa recebe o nome do arquivo .xml ou .drawio, itera por todo o arquivo e identifica os diversos diagramas, sendo que a cada diagrama o programa principal é acionado, gera-se o código VHDL, e o próximo diagrama é identificado, repetindo-se o mesmo procedimento. Basicamente, A Figura 3.22 mostra a execução desta rotina.

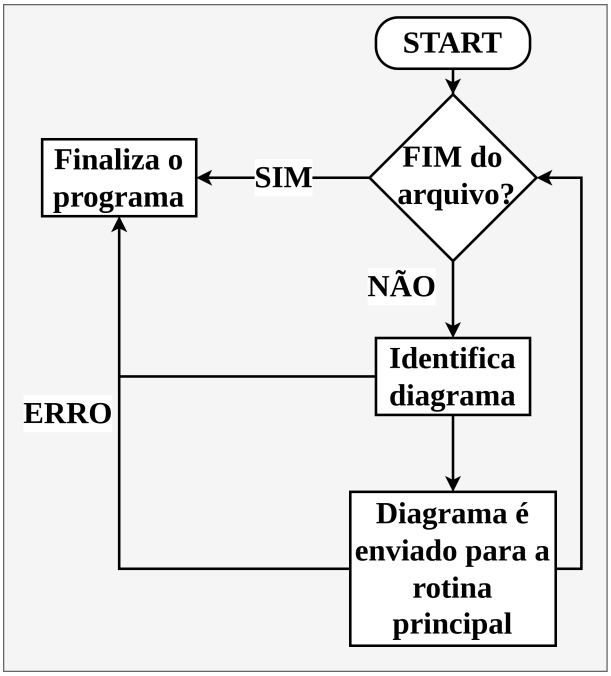

Figura 3.22 – Rotina de geração sequencial de códigos.

Para facilitar o desenvolvimento do gerador foram definidos alguns objetos para guardar os dados de componentes, sinais, entradas e saídas.

Os objetos são mostrados na Figura 3.23. Primeiramente há um objeto que guarda informações sobre entradas, saídas e sinais como:

- **Nome**: Uma string com o nome do sinal, entrada ou saída.
- **Portas**: Porta de origem e destino do sinal.

- **Tipo**: String com o tipo do sinal.
- **Sinal**: Valor numérico indicando se o objeto é um "signal", "in"ou "out".

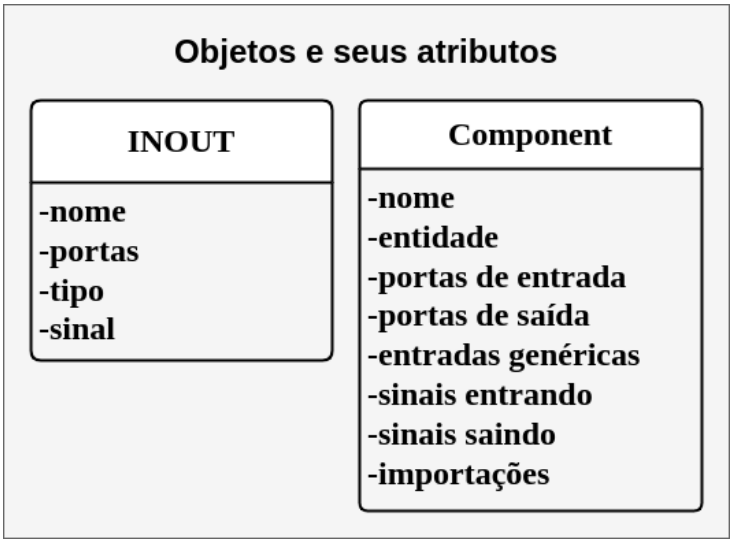

Figura 3.23 – Objetos principais do gerador e seus atributos.

O segundo objeto guarda características importantes dos componentes, que são:

- **Nome**: Uma String com o nome do componente;
- **Entidade**: Uma String com uma cópia da entidade do componente;
- **Portas de entrada**: Lista de portas do tipo "in" do componente.
- **Portas de saída**: Lista de portas do tipo "out" do componente.
- **Entradas Genéricas**: Lista de entradas genéricas do componente.
- **Sinais entrando**: Lista com todos os elementos do tipo "signal" entrando no componente.
- **Sinais saindo**: Lista com todos os elementos do tipo "signal" saindo do componente.
- **Importações**: Lista de bibliotecas utilizadas no arquivo.

O primeiro passo para gerar os arquivos é identificar arquivos VHDL preexistentes e carregá-los para serem utilizados no gerador. Para isso é preciso usar um *parser* para detectar os componentes da entidade de cada arquivo.

Esse *parser* itera por todos os arquivos na pasta "component_files", cada arquivo é lido linha por linha para identificar bibliotecas utilizadas, portas de entrada e portas de saída. É criado um objeto para conter as informações de cada componente, mostrado na Figura 3.24, e esses objetos são salvos em um arquivo *pickle* após passar por todos os arquivos VHDL para uso futuro.

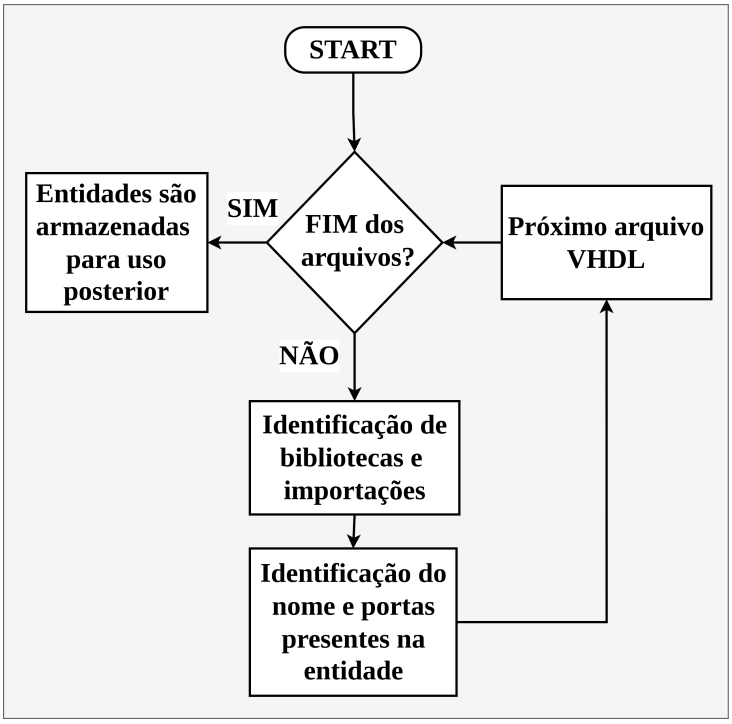

Figura 3.24 – Resumo da caracterização de componentes preexistentes.

3.4.2.1 *Parser* de diagramas individuais

Após a etapa de caracterização dos arquivos VHDL preexistentes, começam as operações do *parser* no diagrama que foi obtido da maneira mostrada na seção 3.4.2. São utilizados os mesmos atributos mencionados na seção 3.3.1, sendo que os elementos que devem ser identificados, como mencionado na seção 3.2.2, são:

- Entradas, saídas e constantes;
- Instanciação de componentes provenientes de códigos VHDL;
- Geradores de códigos;
- Listas de constantes e tipos;
- Ligações entre diferentes componentes e geradores;

Novamente esses elementos são identificados pelo atributo *style* do arquivo .xml. Os identificadores de cada elemento nesse caso são:

- **dual_inline_ic**: Representam os componentes de arquivos VHDL. O nome do componente e o nome da instância são salvos em um dicionário, sendo a chave o ID do elemento.

- **endArrow**: Conexões entre elementos do gráfico. Para cada conexão é criado um objeto, mostrado na Figura 3.23, inicializado com o nome e esse objeto é adicionado em um dicionário com o nome, as portas e o modo da conexão. Cada objeto de conexão é guardado em um dicionário com a chave igual ao ID do elemento.
- **dac**: Entradas ("IN") do circuito. Estas são adicionadas a um dicionário de entradas e saídas, sendo a chave o ID do elemento e seu conteúdo o nome da entrada e o tipo, sendo que também são adicionadas à uma lista de inputs o nome da entrada e o tipo.
- **delta**: Saídas ("OUT") do circuito. Estas são adicionadas a um dicionário de entradas e saídas, sendo a chave o ID do elemento e seu conteúdo o nome da saída e o tipo, sendo que também são adicionadas à uma lista de outputs o nome da saída e o tipo.
- **whiteSpace=wrap**: Geradores de código presentes no diagrama. São adicionados a um dicionário de geradores, sendo a chave o ID e o valor estocado o nome do gerador, os parâmetros passados pelo usuário, e dois dicionários vazios, que serão utilizados para estocar os sinais de entrada e saída do dicionário.
- **childLayout=stackLayout**: Listas de constantes e tipos declarados. Esses elementos são adicionados em um dicionário usando seu ID como chave e possuem uma lista que conterá os tipos ou constantes declaradas.
- **text**: Elementos das listas de constantes ou tipos declarados.

A partir de todos esses identificadores são obtidas as informações na Figura 3.25. Esta figura não apresenta todas as informações, pois há muitas listas auxiliares, mas mostra as características principais obtidas.

Primeiramente há um dicionário com todos os geradores de código utilizados que contém o nome, parâmetros utilizados, dicionário com as entradas e dicionário com as saídas do elemento.

Os componentes possuem dois dicionários associados, o primeiro possui os objetos, mostrados na Figura 3.23, que representam cada arquivo VHDL. O segundo dicionário apresenta o nome do arquivo e o nome da instância, assim pode-se facilmente encontrar o objeto requerido no dicionário de objetos.

As entradas, saídas e sinais possuem 3 dicionários associados. O primeiro possui todas as conexões feitas no gráfico que estão salvas em objetos, ver Figura 3.23, enquanto que os outros dois representam cada bloco de saída e entrada e os blocos de constantes.

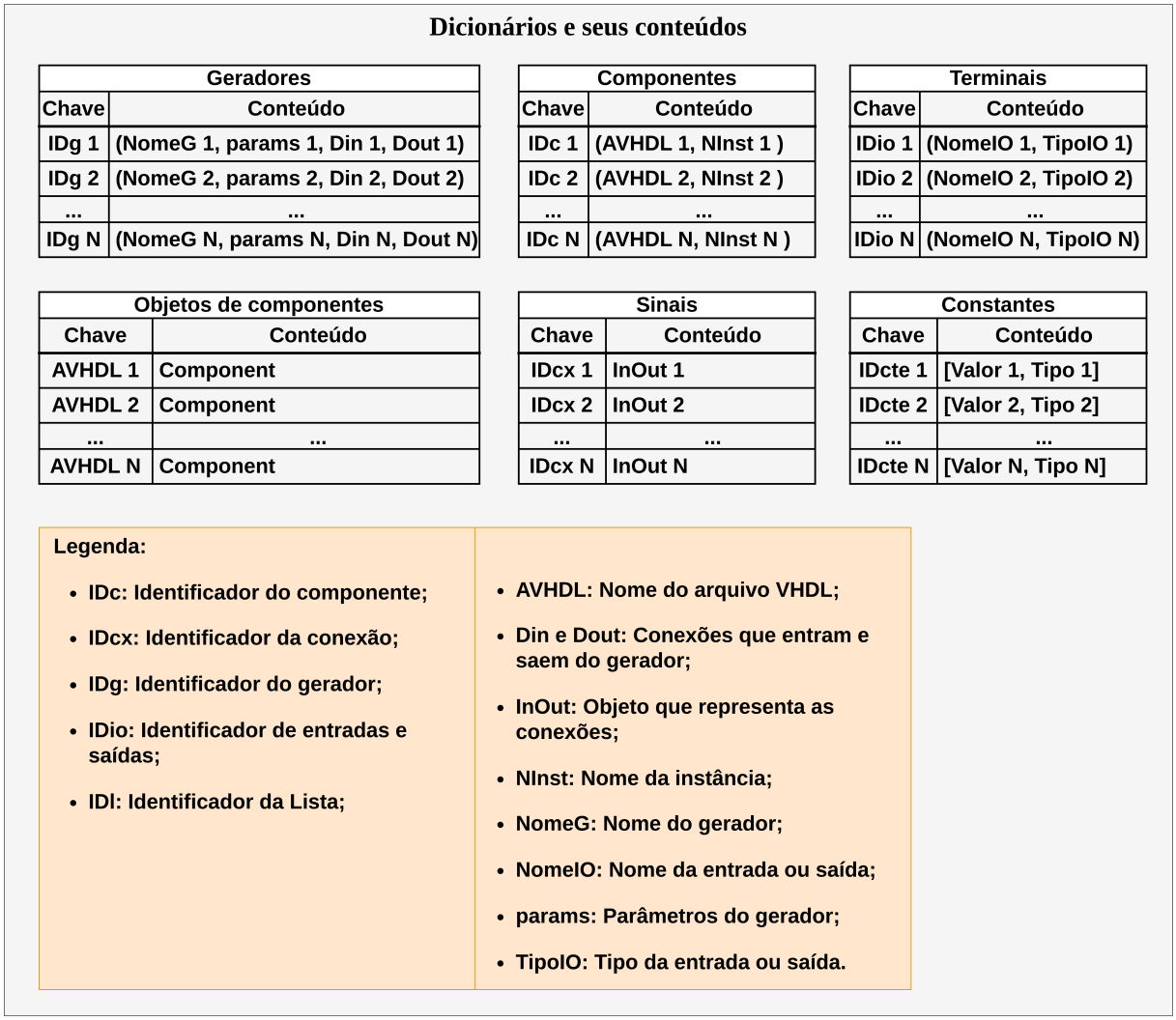

Figura 3.25 – Dicionários com informações obtidas do diagrama.

3.4.3 Geração da estrutura RTL

Projetos RTL possuem uma estrutura muito mais complexa que o projeto de uma FSM, assim, é necessário gerar muito mais porções de código. O principal desafio na geração de uma estrutura RTL é a conexão entre componentes.

O programa começa a montar o VHDL após coletar as informações necessárias do diagrama. O primeiro estágio é a importação de bibliotecas e por padrão a biblioteca da IEEE "STD_LOGIC_1164" e "NUMERIC_STD" são importadas em todos os projetos criados pela ferramenta, sendo que também são importados quaisquer arquivos utilizados pelos componentes presentes no diagrama.

Após as importações começa a entidade nomeada a partir do nome do diagrama e as entradas e saídas são dispostas na entidade. A Figura 3.26 resume o fluxo da criação desta primeira parte do arquivo VHDL.

Após a entidade, começa a arquitetura onde primeiramente são definidos os tipos

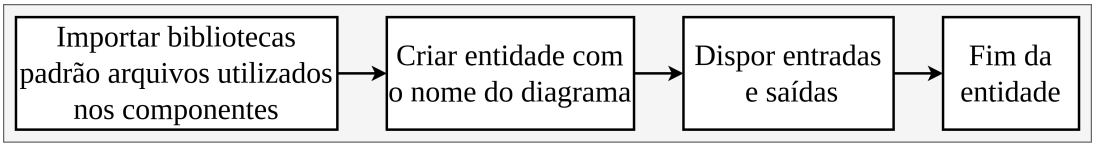

Figura 3.26 – Fluxo de importação de arquivos e criação da entidade.

e constantes, os quais são dispostos em dicionários, como mencionado na seção 3.4.2.1. Assim, o programa itera pelas declarações encontradas nas *Listas* e adiciona cada uma delas. Destaca-se que a ferramenta suporta apenas declarações de tipos compostos, especificamente *arrays*.

Em seguida, os componentes são declarados no arquivo a partir do dicionário mostrado na Figura 3.25. A declaração de um componente possui quase a mesma estrutura da entidade. Assim, pode-se usar a cópia salva no objeto *Component* mostrado na Figura 3.23 e simplesmente substituir a palavra *entity* por *component*. Isso é feito para todos os componentes presentes no diagrama de modo que componentes repetidos são ignorados.

Para terminar as declarações restam somente os sinais e as constantes definidas pelos blocos de constantes. Essas conexões podem sair de um bloco e ir para vários outros, então o primeiro passo é igualar o nome de todas as conexões que saem de uma mesma porta.

Após a uniformização dos nomes das conexões, pode-se declarar os sinais, que consiste basicamente em escrever a palavra-chave *signal*, em seguida o nome do sinal e indicar o seu tipo. Não há suporte para inicialização do sinal. Pela variedade de formas de conexões, é preciso dar suporte a vários casos, de forma que para facilitar o desenvolvimento desses casos, é criado um dicionário auxiliar que possui os *IDs* de conexões que saem de um componente ou gerador em conjunto do elemento de destino dessa conexão como mostra a Figura 3.27. Assim, os sinais são declarados a partir destas listas de cada componente.

Auxiliar para conexões	
Chave	**Conteúdo**
IDcg a	**([IDcx 1, IDed a1], [IDcx 2, IDed a2], ...,[IDcx N, IDed aN])**
IDcg b	**([IDcx 1, IDed b1], [IDcx 2, IDed b2], ...,[IDcx N, IDed bN])**
...	**...**
IDcg z	**([IDcx 1, IDed z1], [IDcx 2, IDed z2], ...,[IDcx N, IDed zN])**

Legenda:

- **IDcg: Identificador do componente ou gerador;**
- **IDed: Identificador do elemento destino;**
- **IDcx: Identificador da conexão.**

Figura 3.27 – Dicionário para na definição dos sinais de conexão.

O primeiro caso são os sinais que saem de um gerador de código para uma saída ou outros componentes. Esse caso é muito importante porque em sua execução os geradores de código passam pela inicialização a partir de seus parâmetros. O gerador é iniciado, o tipo de cada sinal é definido com base na porta em que está conectado e o sinal é declarado.

O segundo caso consiste em sinais provenientes de componentes, ou seja, saem de instâncias. Este funciona de forma similar, obtém-se novamente o tipo do sinal a partir dos dicionários declarados anteriormente e monta-se a declaração do sinal a partir do nome e tipo. Tanto neste caso quanto no anterior podem ocorrer sinais com nomes repetidos, assim, o programa necessita sempre registrar os nomes já declarados, pois declará-los mais de uma vez ocasiona erros de sintaxe.

Os demais casos são as entradas e as constantes. As entradas são conectadas diretamente sem a utilização de sinais. As constantes provenientes de blocos são declaradas como qualquer outra constante.

3.4.3.1 Instanciação de componentes

Os componentes são instanciados após todas as declarações. Para isso os inputs e outputs dos componentes são distribuídos em dicionários para auxiliar a instanciação de modo que cada porta referencia um nome e um tipo como mostra a Figura 3.28. Com as entrada e saídas mapeadas basta iterar pelo nome das portas do componente e adicionar as conexões. Nesse processo são verificados o tipo da entrada e da saída, uma mensagem avisando o usuário é escrita na tela caso eles não sejam do mesmo tipo e em alguns casos o programa tenta converter os tipos, esses casos são mostrados no Código A.2.

Auxiliar para portas	
Chave	**Conteúdo**
Porta a	**(Nome a, Tipo a)**
Porta b	**(Nome b, Tipo b)**
...	**...**
Porta c	**(Nome c, Tipo c)**

Figura 3.28 – Dicionário para auxílio da conexão das portas.

3.4.3.2 Geradores de códigos e finalização do código

Os geradores de códigos são simplesmente as funções adicionadas pelo usuário, mostrado no Código 3.2, que após chamadas pelo programa, retornam uma *string* que é escrita no corpo do arquivo VHDL, especificamente dentro da arquitetura. Assim, basta iterar pelos geradores e chamar a função de cada um com seus parâmetros, entradas e saídas. Ao final da arquitetura são ligados todos sinais que estão ligados às saídas do circuito e a arquitetura é devidamente fechada. Por fim, é criada uma pasta para o diagrama e os arquivos

VHDL de componentes utilizados são colocados nesta pasta em conjunto com o arquivo gerado, o qual é adicionado na biblioteca de arquivos. Essa é uma função que sempre ocorre na geração sequencial, mas pode ser alterada caso seja preciso.

3.4.3.3 Exemplo: *Multiply-Accumulate*

Uma função muito importante na computação é a *multiply-accumulate* que consiste em sucessivas multiplicações acumuladas em um registrador, sendo que essa função é muito importante para a multiplicação entre dois vetores. Como mostra a Figura 3.29, ela é bem simples e consiste apenas em um multiplicador que nesse caso possuem 27 bits de largura em representação aritmética de ponto flutuante.

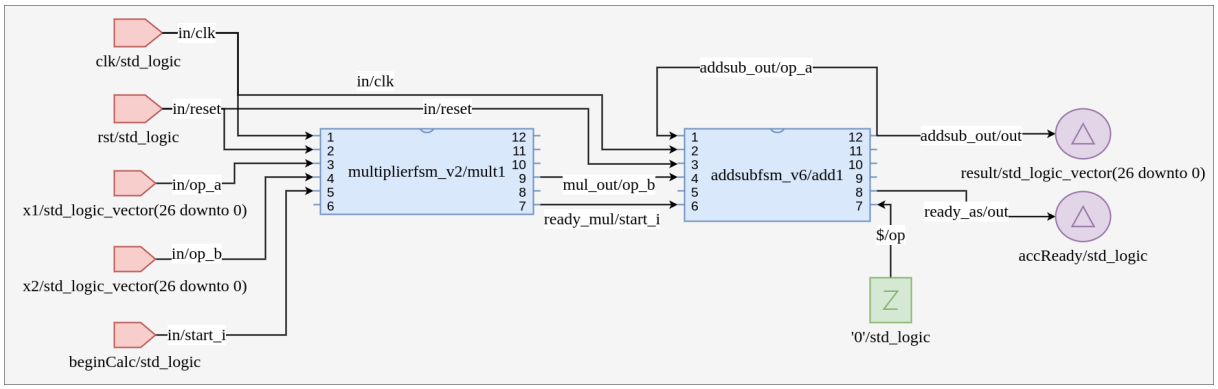

Figura 3.29 – Diagrama no Draw.io da função *multiply-accumulate*.

Exportando-se o diagrama e utilizando-o como entrada da *vRTLgen* a partir do terminal utilizando o comando "python3 <nome do diagrama> 'Y' ", sendo que 'Y' indica que o arquivo VHDL gerado deve ser adicionado à biblioteca. O programa retorna o nome do diagrama indicando que a geração foi bem-sucedida e cria uma pasta com o multiplicador, o somador e o código RTL referente ao diagrama, o qual é mostrado no Apêndice A.3.

3.5 Gerador de Testbench (vTBgen) e simulador

Neste trabalho foi desenvolvida uma interface para a ferramenta *Gerador de Testbench*, também chamada de vTBgen, mostrada na Figura 3.30. Essa interface possui botões para adicionar arquivos VHDL, adicionar estímulos de entrada, adicionar saídas e gerar o código VHDL do *testbench*. Os botões "LOAD"e "SAVE"serão desenvolvidos futuramente para salvar e carregar *testbenches* produzidos.

O gerador de *testbench* pode ser aberto utilizando o botão de nome "create testbench"e é aberta a janela mostrada na Figura 3.30. A partir desta interface é possível criar um *testbench*, sendo que ao adicionar o componente, suas entradas e saídas são mostradas na tela para auxiliar o usuário.

Atualmente a ferramenta permite a instanciação e simulação de apenas um componente no testbench. Adicionalmente, a ferramenta possui suporte para três tipos de entrada e um tipo de saída, como mostra a Figura 3.31. Os elementos presentes na ferramenta e suas funcionalidades são:

- **Clock**: Cria um sinal de *clock* para ser utilizado como entrada. Possui campos para definir um nome, a frequência do clock, sendo possível utilizar alguns prefixos do Sistema Internacional de Unidades como em 100MHz, e um campo de valor inicial.
- **Constant**: Cria uma constante com o valor indicado pelo usuário. Possui campos para definir um nome, o tipo e o valor da constante.
- **Variable Signal**: Cria um sinal que pode receber diversos valores diferentes em diversos instantes de tempo. Possui campos para definir um nome, o tipo, o valor inicial do sinal e, por fim, uma lista de valores com seus respectivos instantes de tempo separados por ponto e vírgula. Essa função utiliza da palavra-chave *after* no VHDL para transicionar os valores.
- **Output Signal**: Cria um sinal para receber a saída de um circuito. Possui campos para definir um nome e o tipo do sinal.

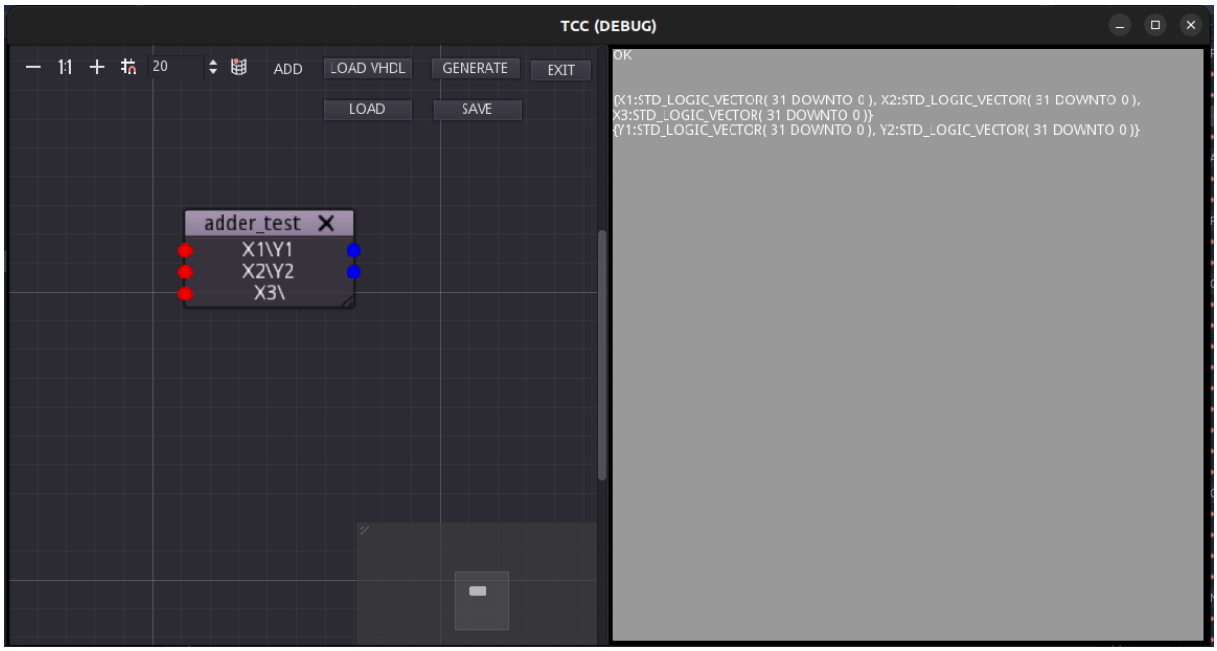

Figura 3.30 – Interface do gerador de testbench.

O *Godot* possui nativamente um criador de diagramas de grafos, então há uma função especifica para obter todas as informações do diagrama, logo, não é preciso criar um *parser*. A partir destas informações a geração do código é realizada por um código adaptado da ferramenta *vRTLgen*.

Para finalizar, uma ferramenta de simulação foi incorporada ao HiLDA integrando a função de simulação do simulador open-source gHDL a uma interface gráfica em conjunto com as bibliotecas da AMD Xilinx chamadas Unisim. Para adicionar as bibliotecas Unisim é necessário indicar o local de instalação das bibliotecas da AMD Xilinx utilizando o botão "CONFIG"mostrado na Figura 3.34. Para realizar as simulações é utilizado um *bash script* que automaticamente importa os arquivos da biblioteca da AMD Xilinx no gHDL, simula o arquivo e abre o resultado utilizando o um visualizador de ondas gratuito GTKWave. Deste modo, a interface gráfica apenas possui a função de escolher o arquivo que será simulado, a entidade que será simulada e configurações como o tempo de simulação e o local de instalação das bibliotecas.

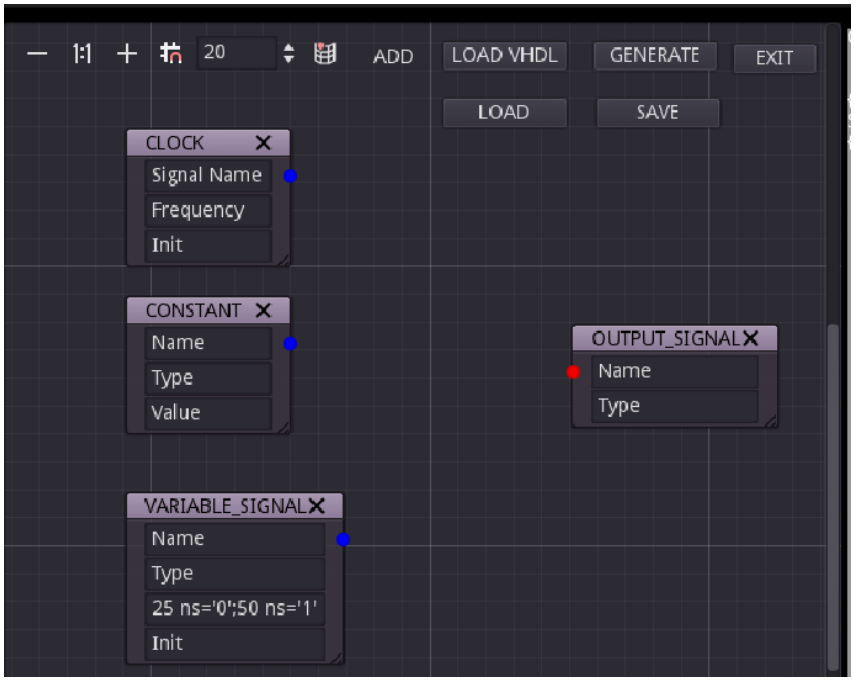

Figura 3.31 – Funções implementadas na vTBgen.

3.6 Desenvolvimento do *AXIcreator*

A ferramenta HiLDA inclui um gerador de encapsulamento AXI com suporte às interfaces AXI4-Lite e AXI4-Full, no modo escravo, para co-projeto HW/SW usando SoCs FPGA da AMD Xilinx. As interfaces AXI possuem muitas particularidades, assim, o Godot foi escolhido para implementá-las, permitindo a customização de cada uma das duas interfaces implementadas que não seria possível ou seria dificultada caso implementada no Draw.io.

3.6.1 Interfaces AXI suportadas e suas representações gráficas

As interfaces AXI4-Lite e AXI4-Full podem ser adicionadas a partir do botão “ADD". Sendo que para a implementação destas foi utilizada a ferramenta de diagramas de grafos

presente nativamente no Godot, assim foi possível criar uma interface de usuário personalizada para cada um dos protocolos baseada nas opções apresentadas no Vivado para a criação destas interfaces.

As interfaces para a implementação do AXI4-Lite tanto na aplicação HiLDA quanto no Vivado é mostrada na Figura 3.32. A interface do AXI4-Lite no *AXIcreator* possui as seguintes características:

- Não há como escolher o nome da interface no *AXIcreator*, sendo que ele é escolhido automaticamente baseado no nome do código VHDL que está sendo encapsulado;
- O tipo de interface é sempre indicado no nome do nó;
- O modo de interface é fixo para o *AXIcreator* e sempre é do modo escravo;
- A largura de dados para as interfaces são de 32 bits, mas o *AXIcreator* consegue se adaptar à interfaces com largura menor que 32 bits;
- O número de registradores pode variar de 4 a 512;
- Saídas e entradas do AXI são criadas individualmente e seguem a nomenclatura do AXI4-Lite.

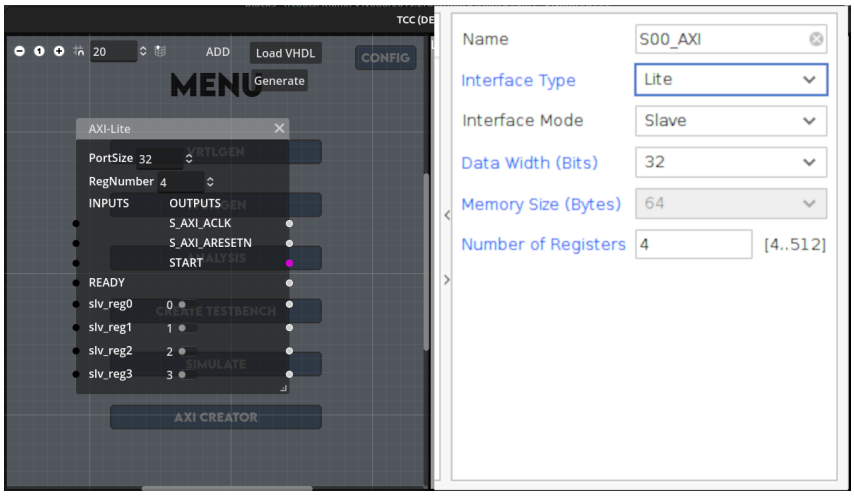

Figura 3.32 – Interfaces para a criação de protocolos AXI4-Lite da plataforma HiLDA à esquerda e do Vivado à direita.

No caso da interface AXI4-Full no *AXIcreator*, mostrado na Figura 3.33, possui as seguintes características:

- O nome da interface no é escolhido automaticamente baseado no nome do código VHDL que está sendo encapsulado.
- A interface AXI4-Full é configurada no modo escravo.

- Os tamanhos de memória são iguais aos apresentados no Vivado apresentando valores entre 64 até 1024 bytes;
- Saídas e entradas do AXI são criadas em blocos de palavras de 32 bits, sendo necessário, também, nomeá-las.

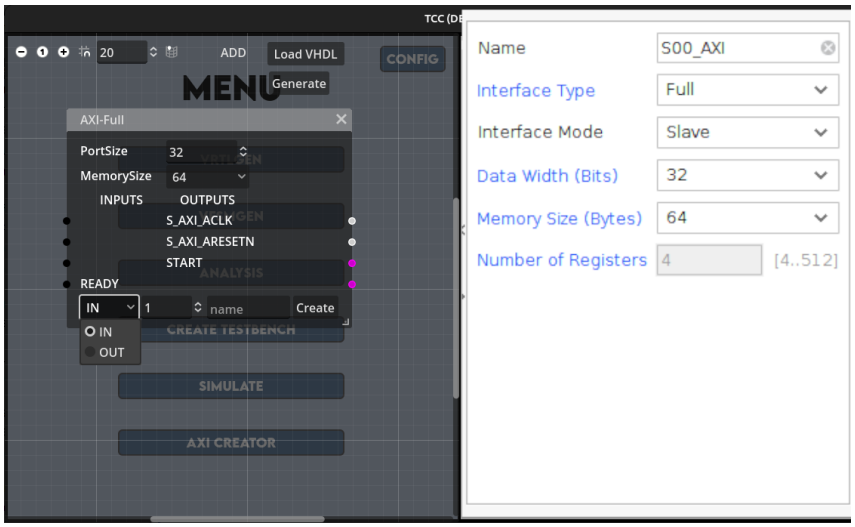

Figura 3.33 – Interfaces para a criação de protocolos AXI4-Full da plataforma HiLDA à esquerda e do Vivado à direita.

3.6.2 Funcionamento e implementação dos protocolos

As duas interfaces quando criadas no *AXIcreator* possuem um funcionamento similar, porém a forma com que ambas são implementadas é alterada para acomodar as particularidades de cada interface.

Percebe-se a partir das Figuras 3.32 e 3.33 que é possível conectar tanto o “clock"quanto o “reset"da interface ao código que será encapsulado. As duas apresentam, também, uma porta chamada “START"e uma porta chamada “READY", sendo que a porta “START"é assertada durante um ciclo de “clock"assim que a última entrada do AXI é escrita na memória da interface, indicando que os dados estão prontos para serem utilizados. Por outro lado, a porta “READY"aciona o processo de escrita das saídas do AXI na memória da interface ao receber um pulso de um ciclo de *clock*.

Concluindo, as interfaces funcionam a partir dos estímulos escritos em “READY"e lidos em “START"para determinar se os dados serão lidos da memória da interface ou escritos nela. O resto das portas são característicos de cada interface e serão detalhados nas seções que explicam suas implementações.

3.6.3 Implementação do AXI4-Lite

Para gerar os arquivos das interfaces foram utilizados arquivos gerados pelo Vivado como base para cada uma das interfaces, porém adaptadas para 32 bits, e trechos de código que devem ser gerados pela aplicação identificados por "<nome do identificador>" no corpo do texto para facilitar a detecção pelo *script*.

Assim, ao pressionar o botão "GENERATE", um *script* em Python lê o arquivo base linha por linha e ao detectar um dos identificadores que deve ser alterado, insere o texto conforme as configurações indicadas na interface de usuário.

Os identificadores para a implementação do AXI4-Lite são:

- **"<libs>"**: Indica que essa porção do código deve ser substituída pelas bibliotecas indicadas pelo usuário;
- **"<entity>"**: Designa o campo que deve ser substituído pelo nome da entidade do arquivo que está sendo encapsulado;
- **"<regn>"**: Esse campo define a largura do endereço da interface AXI denominado "C_S_AXI_ADDR_WIDTH", sendo que depende do número de registradores indicados pelo usuário e seu valor é igual ao número de registradores utilizados;
- "**<optaddr>"**: Esse identificador é relativo ao número de endereços necessários para acessar toda a memória sendo que este também depende do número de registradores e seu valor é $int(ceil(log_2(n\_registradores)-1))$, isto é, a quantidade de bits necessários para acessar todos os registradores;
- **"<signals>"**: Região onde deve ocorrer a declaração de sinais utilizados na interface. Neste caso são declarados os sinais que entram no componente encapsulado quando possuem entradas do tipo array, todas as saídas do componente encapsulado e sinais de "reset", "start", "ready", bem como os sinais utilizados pela interface AXI4-Lite denominados "slv_reg";
- **"<components>"**: Indica onde o componente que será encapsulado será declarado;
- **"<read>"**: Trecho do código que lida com as entradas do AXI que são escritas nos registradores "slv_reg". Nesse caso o *script* em Python cria exatamente a mesma estrutura de qualquer AXI-Lite criado pelo Vivado, porém, adiciona o sinal "START" que emite um pulso de *clock* ao receber a última entrada definida pelo usuário;
- **"<slvregs>"**: Indica a lista de sensitividade do processo que escreve dos registradores "slv_reg" para a saída da interface. Neste caso o *script* somente adiciona os registrados "slv_reg" na lista de sensitividade;

- **“<write>”**: Trecho do código que escreve os valores contidos nos registradores “slv_reg” para a saída da interface. Novamente, foi utilizado como base o arquivo gerado pelo Vivado e consiste em um “CASE-WHEN” que escreve “slv_regx” na saída dependendo do endereço de memória, sendo que x pertence ao intervalo de 0 até o número definido pelo usuário. Os registradores “slv_reg” definidos como saída são substituídos por um sinal criado pelo *script* para escrever as saídas do código encapsulado na interface AXI, sendo que esses sinais possuem zeros concatenados à direita para compensar a largura de bits indicada pelo usuário;
- **“<userlogic>”**: Região onde o componente a ser encapsulado é instanciado e suas entradas e saídas são conectadas. Primeiramente as entradas que recebem mais de um registrador “slv_reg” são atribuídas à um registrador único que é conectado à entrada do circuito encapsulado, um sinal recebe o *reset* do AXI invertido e o componente é instanciado conforme as conexões que o usuário efetuou, sendo que as entradas são automaticamente redimensionadas para a largura de bits indicada pelo usuário.

Após a conclusão da geração de cada um desses campos no arquivo, o código é gerado e fica localizado em uma pasta chamada “AXIrepo” e está pronto para ser utilizado no Vivado. Os arquivos podem ser adicionados a um “Block Design” no Vivado, o qual irá reconhecer automaticamente a interface AXI.

3.6.4 Implementação do AXI4-Full

O gerador do AXI4-Full funciona da mesma maneira, porém o trecho de código em cada identificador muda em alguns casos e permanece igual em outros. Assim, novamente, ao pressionar o botão “GENERATE” os campos com identificadores são preenchidos conforme a configuração escolhida pelo usuário.

Os identificadores para a implementação do AXI4-Full são:

- **“<libs>”**: Porção do código substituída pelas bibliotecas do usuário;
- **“<entity>”**: Designa o campo que deve ser substituído pelo nome da entidade do arquivo que está sendo encapsulado;
- **“<regn>”**: Similarmente ao AXI4-Lite, este indicador determina a largura do barramento de endereço de memória, porém, nesse caso ela é referente à quantidade de bytes escolhidos pelo usuário e não mais uma quantidade de registradores como é no AXI4-Lite, assim possui o valor de $log2(N)$, sendo N a quantidade de bytes;
- **“<opt_mem>”**: Este campo é similar ao anterior, mas este define o tamanho do endereço de memória para acessar as palavras de 32 bits armazenadas na interface e é definido por $log2(N) - 3$, sendo N igual ao tamanho da memória em bytes;

- **“<arrsize>”**: Para realizar a interface entre o código encapsulado e a interface é necessário criar um registrador que possui o mesmo tipo da memória interna da interface que é definido como um *array* do tipo “std_logic_vector(31 downto 0)”, assim, “<arrsize>” define o tamanho deste array que é referente à quantidade de memória determinada pelo usuário e possui o tamanho de $2^{log2(N)-2} - 1$, sendo N igual ao tamanho da memória em bytes;
- **“<signals>”**: Similarmente ao AXI4-Lite, nesta seção do código são definidos todos os sinais para realizar as conexões indicadas pelo usuário. Os sinais criados nesta etapa são “sSTART”, “sREADY”, “sRESET e os sinais de entrada e saída do código encapsulado que possuem o tipo igual ao indicado na porta da entidade.
- **“<components>”**: Indica onde o componente que será encapsulado será declarado;
- **“<sstart>”**: Campo que representa a quantidade de memória total utilizada pelo usuário como entrada no código encapsulado. Assim, cria uma condicional que gera um pulso em “sSTART” de um ciclo de *clock* ao escrever no último endereço de memória.
- **“<sready>”**: Condicional que escreve as saídas do código encapsulado para a memória da interface. Este condicional é acionado ao receber o valor lógico alto no sinal “sready” que deve ser conectado ao código encapsulado.
- **“<userlogic>”**: Representa a área de instanciação do código encapsulado. Nesta seção o sinal de *reset* é invertido, sinais auxiliares recebem os valores lidos na memória, sendo que a ferramenta automaticamente monta *arrays* em sinais que recebem mais de uma entrada e, por fim, estes sinais são conectados ao código instanciado conforme as ligações feitas pelo usuário na interface gráfica.

Igualmente ao AXI4-Lite, após o processo de geração o arquivo é salvo em uma pasta e fica disponível para uso. Com isso, a descrição do funcionamento das duas interfaces está completo.

3.7 Desenvolvimento da Interface Gráfica

A interface gráfica integra os geradores de códigos *vFSMgen*, *vRTLgen* e o *AXIcreator*, bem como funcionalidades do gHDL como a capacidade de analisar códigos sintaticamente e executar simulações.

3.7.1 Funções da interface gráfica

A aplicação precisa ser capaz de integrar as seguintes funcionalidades a partir da interface gráfica:

- Carregar diagramas de FSMs para gerar e verificar a sintaxe de arquivos VHDL gerados;
- Carregar diagramas de projetos RTL para gerar e verificar a sintaxe de arquivos VHDL gerados;
- Verificar a sintaxe arquivos VHDL individualmente;
- Geração de *testbenches* e simulação a partir do gHDL, utilizando a biblioteca de simulação da AMD Xilinx chamada Unisim.
- Encapsular códigos utilizando protocolos AXI4-Lite ou AXI4-Full.

Para o desenvolvimento da aplicação foi escolhido o software *Godot*, sendo este uma *game engine* com diversos objetos úteis para o projeto, como: botões para diversas funções, janelas para escolha de arquivos, caixas de texto que servem como logs, criador de diagramas de grafos, entre outros (GODOT, 2023).

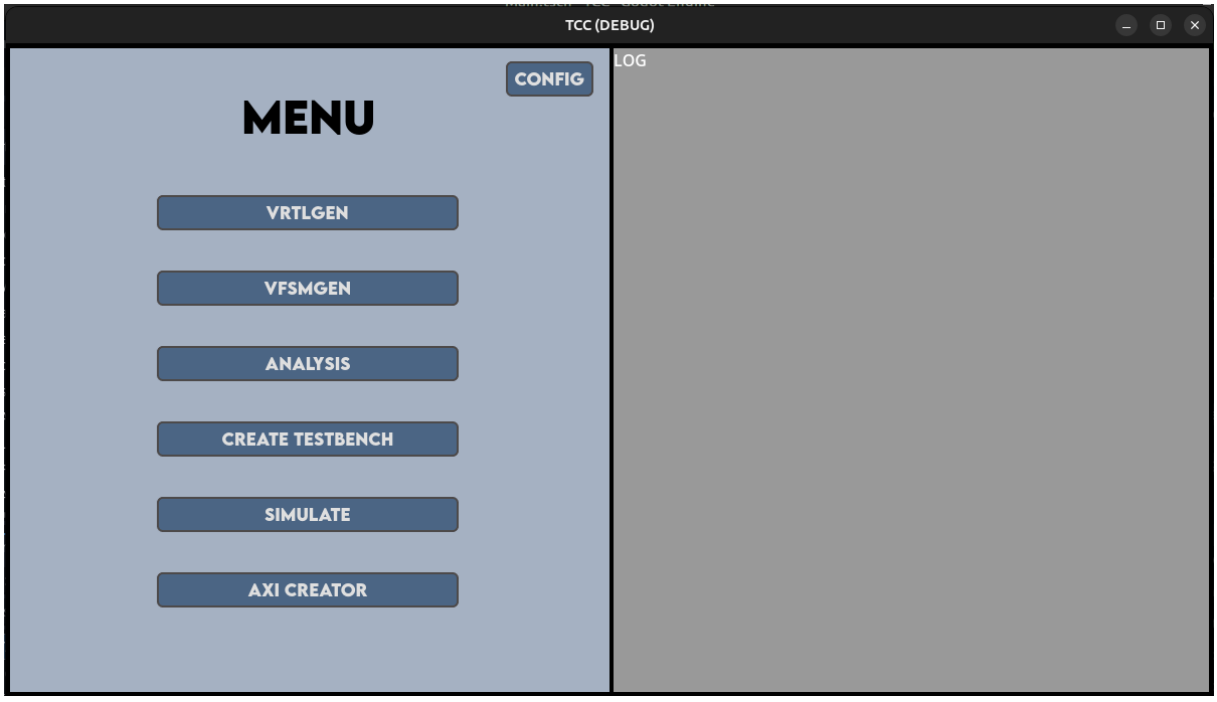

Figura 3.34 – Tela inicial da interface gráfica.

A Figura 3.34 apresenta a tela principal da interface gráfica. Os dois primeiros botões são os geradores *vRTLgen* e o *vFSMgen*, o terceiro botão é o analisador individual de arquivos VHDL, o quarto e o quinto botão são, respectivamente, o gerador de *testbench* e o simulador. O último botão é o gerador de encapsulamento AXI4-Full e AXI4-Lite. Por fim, há o botão denominado “CONFIG"possibilita definir configurações gerais relacionadas à simulação e o gerador de encapsulamento AXI.

3.7.2 Integração do vFSMgen

O funcionamento da *vFSMgen* consiste na execução do *script* em Python, a partir do terminal, e na verificação do arquivo gerado. Primeiramente, uma fonte de entrada de texto aparece para a escolha do nome da FSM, diferentemente da *vRTLgen*, na qual o nome do arquivo e da entidade é o nome do diagrama. Em seguida escolhe-se o diagrama a partir de uma janela de escolha de arquivos, a ferramenta realiza todos os procedimentos restantes a partir desta etapa. A Figura 3.35 resume o fluxo de funcionamento da vFSMgen.

O primeiro procedimento consiste em verificar a execução do código em Python, que caso não tenha sido bem-sucedida, exibe uma mensagem de erro do programa na tela. Do contrário, o programa segue para a verificação do arquivo VHDL gerado utilizando o gHDL via terminal, retornando a mensagem de êxito ou de erro dependendo da execução.

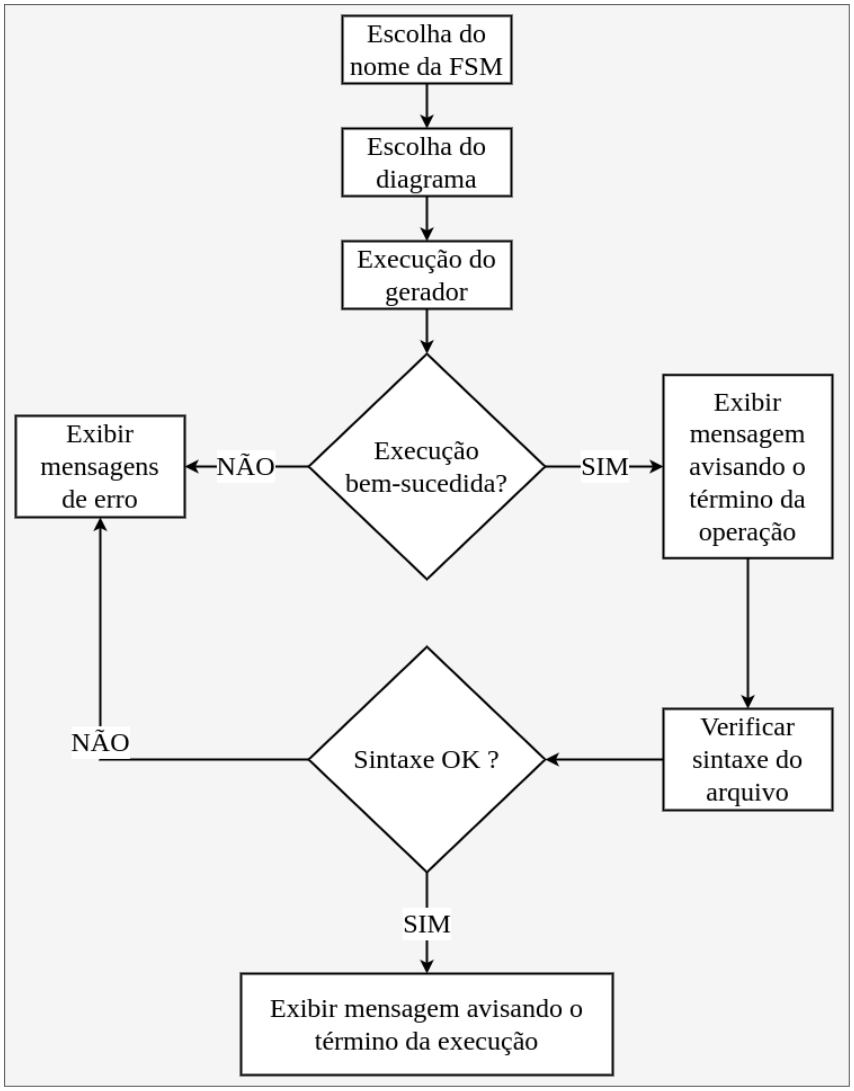

Figura 3.35 – Funcionamento da ferramenta *vFSMgen*.

3.7.3 Integração *vRTLgen*

A *vRTLgen*, similarmente, consiste na execução de outro script em Python e na verificação do código gerado, porém, nesse caso é necessário realizar a verificação de vários arquivos VHDL. Primeiramente escolhe-se o diagrama, executa-se o *script* em Python e uma mensagem de erro é mostrada caso haja algum problema na execução. Em seguida, em caso de êxito, o programa verifica todos os arquivos gerados e emite mensagens de erro ou êxito para cada arquivo. A Figura 3.36 resume o fluxo de funcionamento dessa integração.

Um aspecto interessante é que as mensagens provenientes da análise do gHDL são mostradas na tela, ou seja, caso a execução tenha sido bem-sucedida, mas haja algum problema no diagrama que gerou algum erro de sintaxe, a respectiva mensagem será mostrada na tela. A ferramenta também apresenta mensagens de *warnings*.

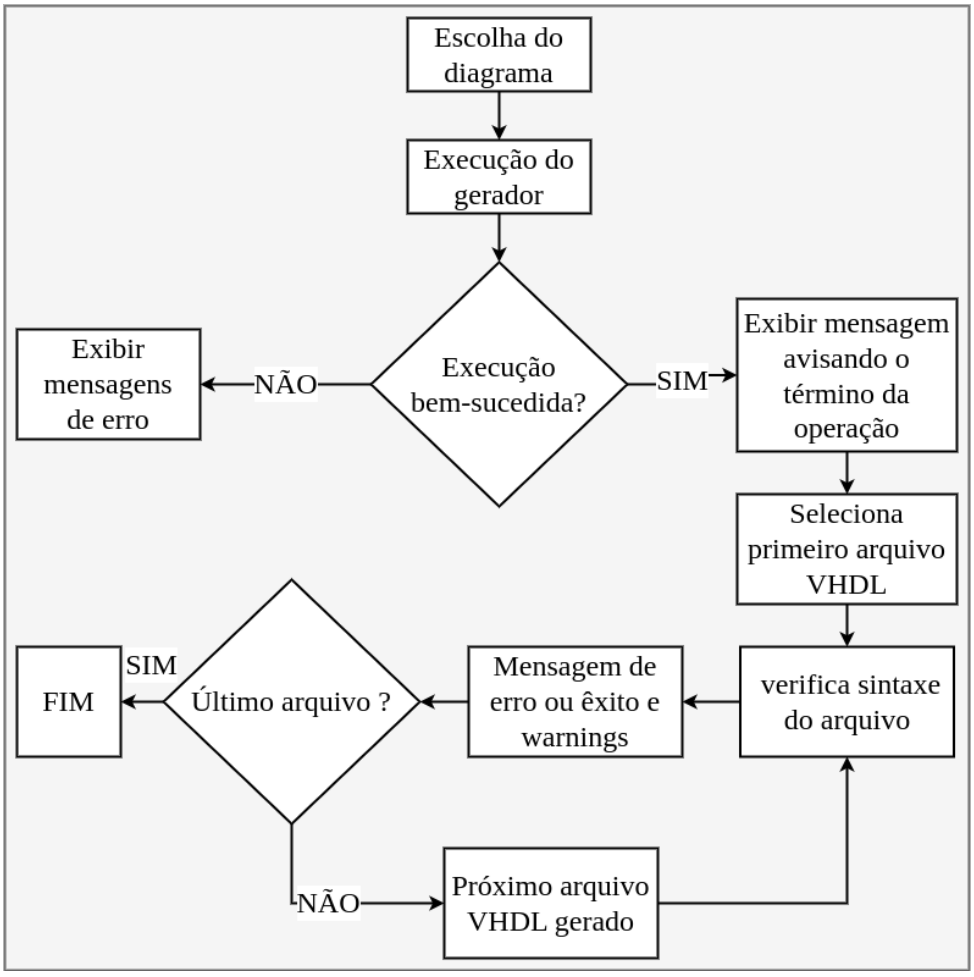

Figura 3.36 – Funcionamento da ferramenta vRTLgen.

3.7.4 Verificação da sintaxe de arquivos VHDL

A função *analysis* analisa e compila os arquivos passados como argumento da função, essa função pode ser executada utilizando o comando "ghdl -a <nome do arquivo>.vhdl".

Essa função é utilizada também para verificar todos os arquivos gerados pelo *vRTLgen* e pelo *vFSMgen*.

O processo de verificação é simples. Ao apertar o botão, abre-se uma janela para a escolha do arquivo, mostrado na Figura 3.37. Após a escolha do arquivo, a função *analysis* do gHDL é executada no terminal pelo Godot utilizando a função "OS.execute()". Caso a execução tenha sido um sucesso, a mensagem "OK"aparece na tela, do contrário uma mensagem de erro é mostrada.

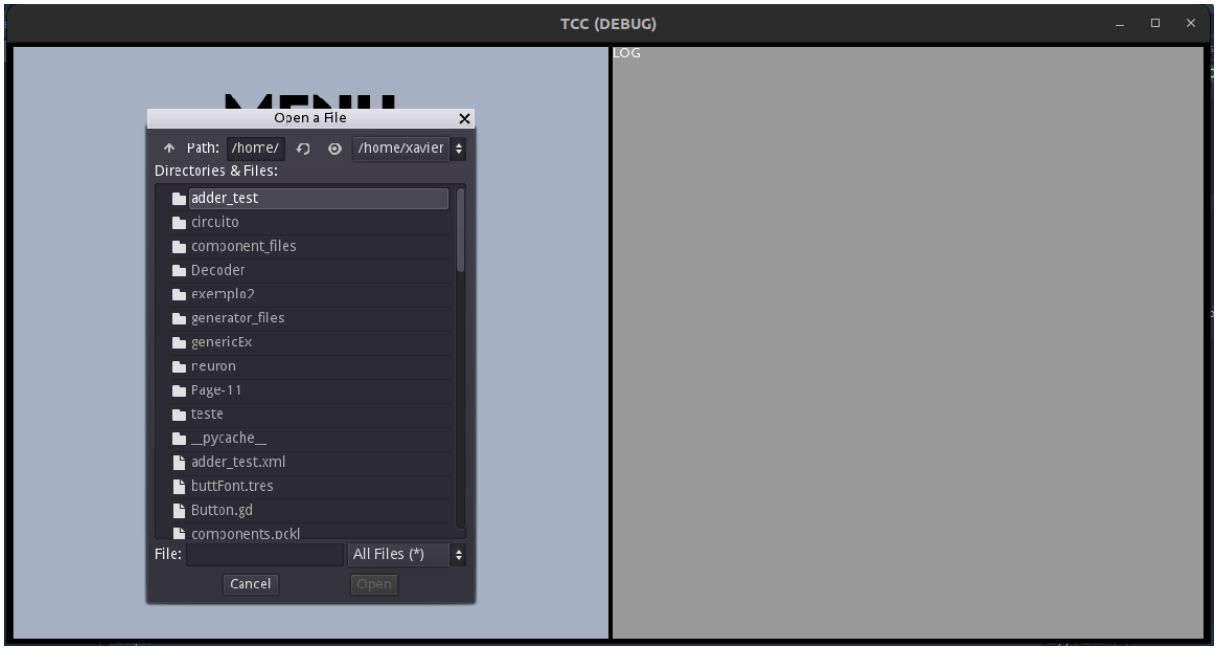

Figura 3.37 – Janela de seleção de arquivos.

4 Resultados

Para a validação das ferramentas criadas neste trabalho são propostos casos de estudo para cada uma das ferramentas principais criadas. Um exemplo de FSM com foco didático para a ferramenta vFSMgen, a elaboração de um perceptron para a vRTLgen e dois exemplos para o *AXIcreator*, sendo eles o encapsulamento de um neurônio tipo perceptron utilizando AXI4-Lite e o encapsulamento de uma rede neural *Multilayer Perceptron* (MLP) utilizando AXI4-Full. O perceptron e a MLP foram baseados na implementação em hardware de uma *Particle Swarm Optimization* (PSO)(MUÑOZ, D. et al., 2014). Por fim, foram elaborados diversos vídeo tutoriais dos casos de estudo que podem ser encontrados nesta *playlist*.

4.1 Caso de Estudo de uma FSM

As FSMs possuem diversas aplicações não somente em hardware, mas também em software. Para demonstrar a capacidade da ferramenta *vFSMgen* foi escolhida a aplicação de uma FSM de controle de um marca-passo atrioventricular como mostrado na figura 4.38. O exemplo é proposto como exercício em um livro guia na área de sistemas digitais (VAHID, 2007).

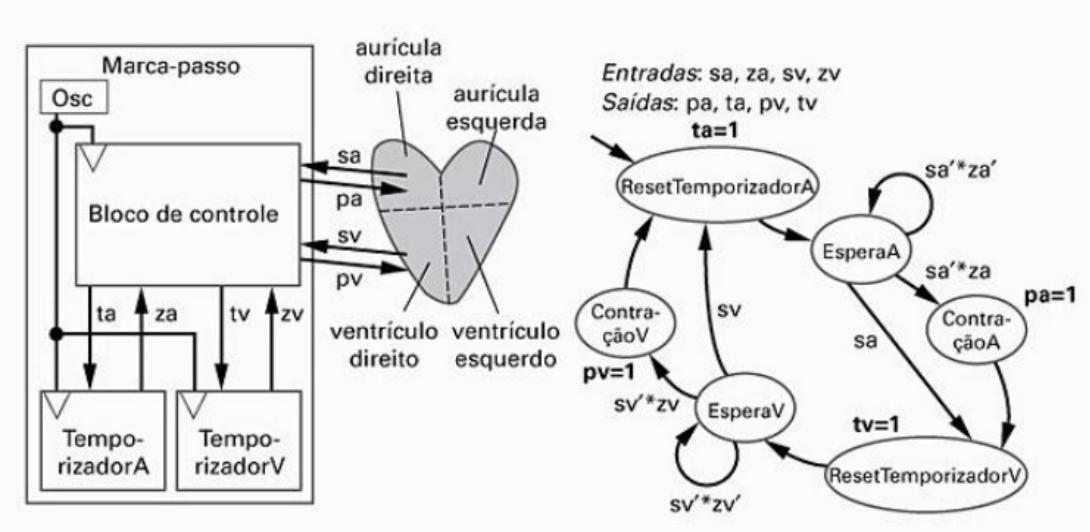

Figura 4.38 – Bloco de controle de um marca-passo atrioventricular (VAHID, 2007).

A Figura 4.39 mostra o diagrama recriado no Draw.io. A lista *Variables* indica como cada entrada está representada no diagrama e o resto das listas representam a saída de cada um dos estados. As transições de cada estado são implementadas de forma igual à original, porém uma delas é substituída por uma transição *else*. Após exportar o diagrama em um arquivo .xml e utilizá-lo no vFSMgen, o código A.4 foi gerado, que após submetido à ferramenta de análise do gHDL, retorna um resultado bem-sucedido (vide Figura 4.40).

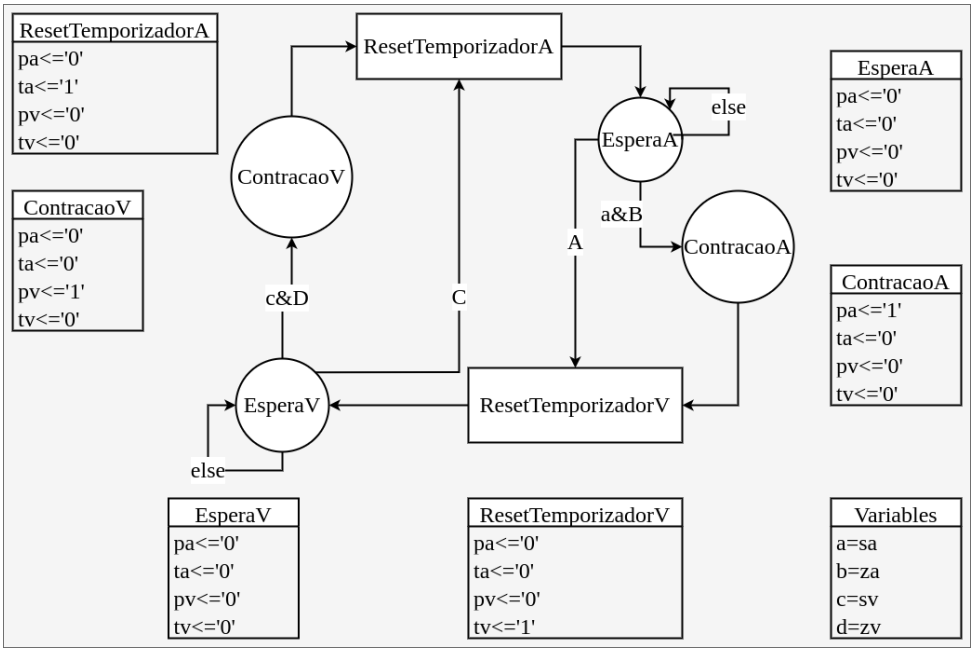

Figura 4.39 – Diagrama da FSM recriado no Draw.io.

Figura 4.40 – Resultado da análise da FSM do exemplo do marca-passo utilizando o gHDL dentro da aplicação.

Com os resultados positivos da análise sintática do circuito, pode-se importar o arquivo VHDL ao Vivado e realizar a síntese lógica. Para testar o comportamento da implementação foi utilizada a ferramenta vTBgen para gerar o código A.5 como *testbench*. Este código testa as transições presentes no diagrama bem como a saída de cada estado. A Figura 4.41 mostra a FSM inicializando no estado *ResetTemporizadorA*, transicionando para *EsperaA* onde espera o sinal *ssa* mudar para '1' no intuito de transicionar para *ResetTemporizadorV*. Por fim, este transiciona diretamente para *EsperaV*. Nesta imagem também é possível verificar que as saídas dos quatro estados mencionados estão de acordo com o gráfico.

A FSM permanece no estado *EsperaV* até o sinal *ssv* mudar para '1' e o estado atual transicionar para *ResetTemporizadorA*. Logo após é mostrado os passos de transição originando de *EsperaA* para *ResetTemporizadorV* passando por *ContracaoA* e a transição de *EsperaV* para *ResetTemporizadorA* passando por *ContracaoV*. Novamente é possível verificar que as saídas de cada estado estão de acordo com o diagrama, como mostra a Figura 4.42. O processo utilizado para gerar estes resultados pode ser conferido neste vídeo

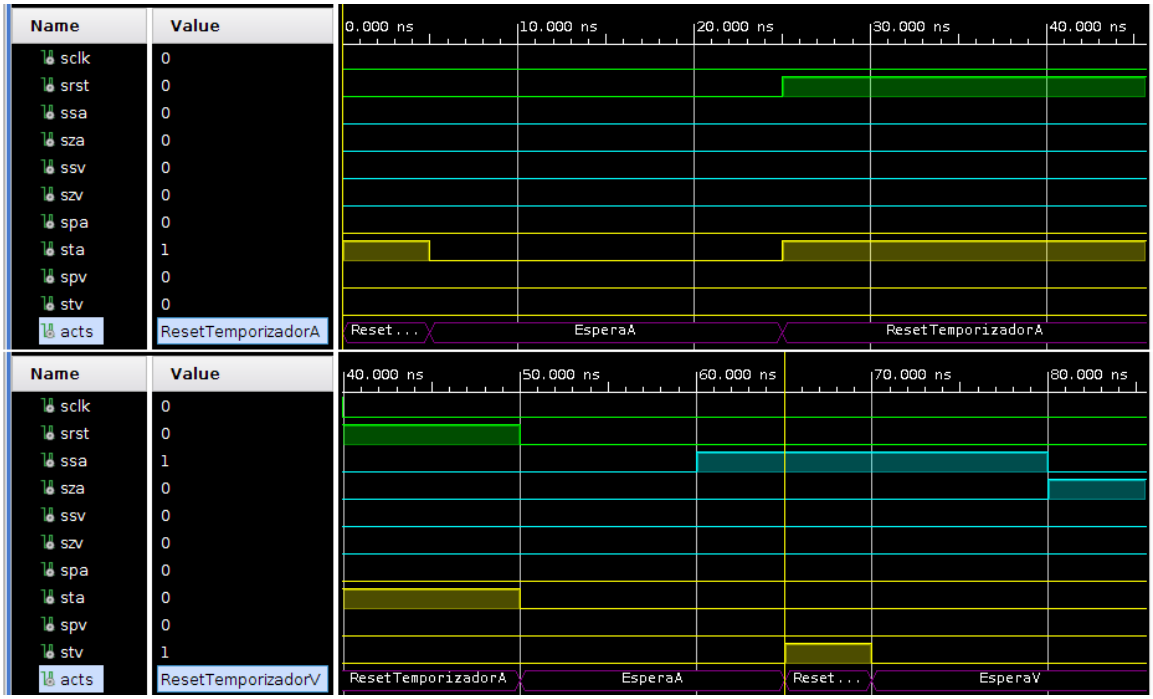

Figura 4.41 – Simulação da FSM do exemplo do marca-passo.

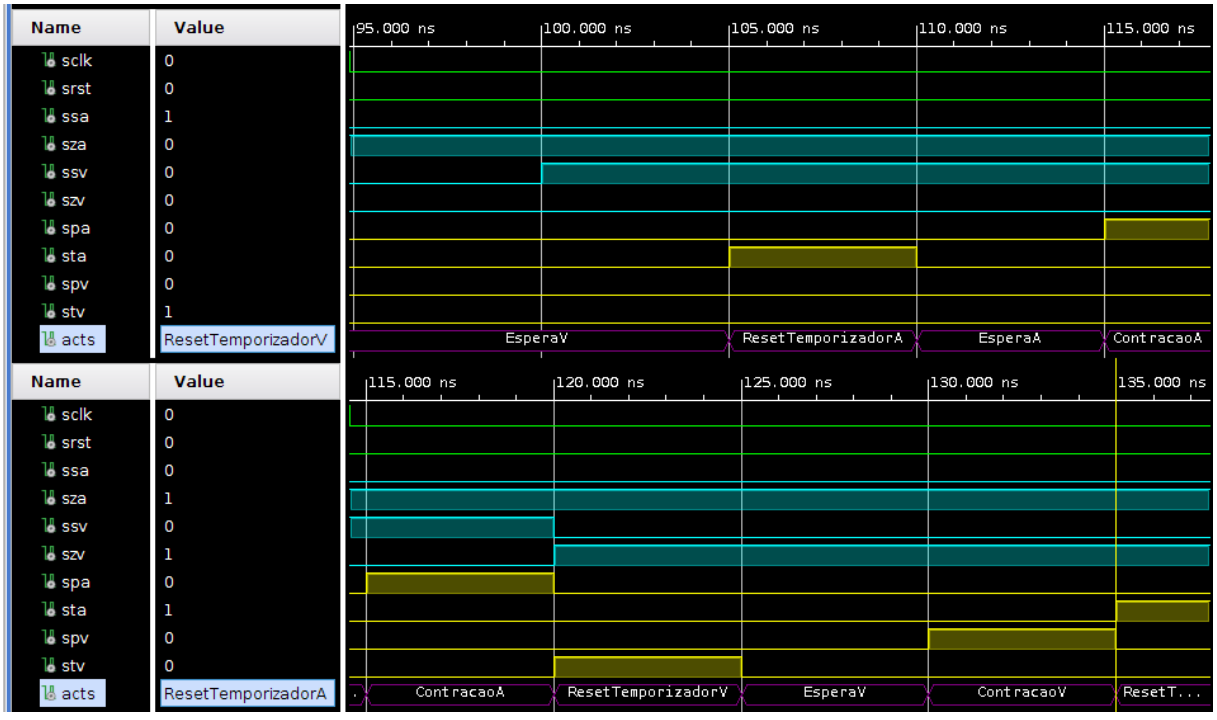

Figura 4.42 – Continuação da simulação da FSM do exemplo do marca-passo.

Uma vez verificado o funcionamento do circuito, foi criado manualmente um arquivo .xdc, mapeando as entradas do circuito em botões da placa *Basys3*, e as saídas em leds. Foi realizada a implementação física utilizando um clock de 100MHz. Após a implementação física foi realizada a caracterização do circuito.

A Tabela 4.3 mostra a utilização de recursos do circuito. Como esperado há uma baixa utilização de LUTs (*Look Up Tables*) e FFs. A Figura 4.43 apresenta o layout do circuito implementado no FPGA. Observa-se que o circuito atendeu os requisitos de temporização para *setup* e *hold*, conforme mostrado na Figura 4.44 que apresenta a análise de timing. O circuito utiliza aproximadamente 75mW, sendo maior parte utilizada pela potência estática do circuito, especificamente o IO, como mostrado na Figura 4.45.

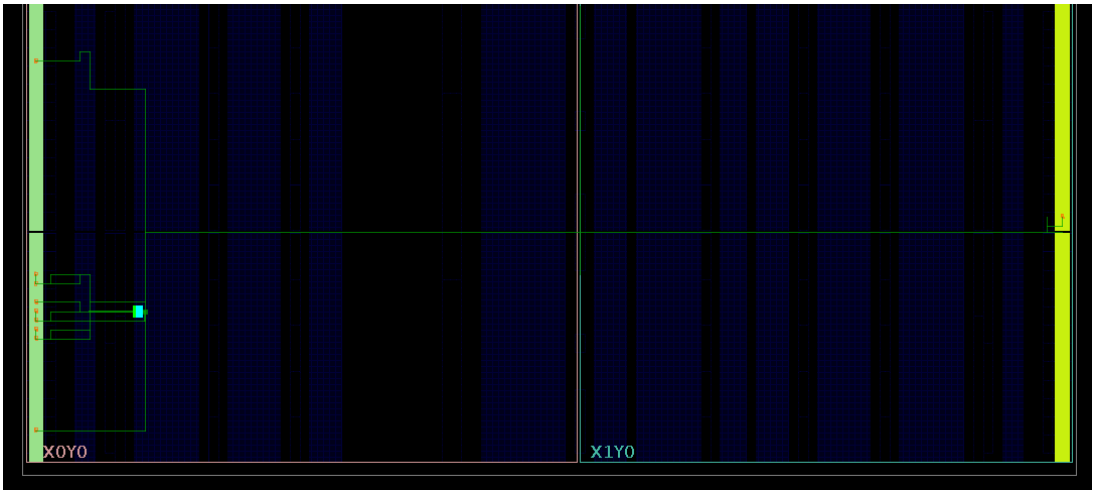

Figura 4.43 – *Place and Route* referente ao circuito da FSM.

Tabela 4.3 – Utilização de recursos pós-implementação.

Resource	Utilization	Available	Utilization %
LUT	4	20800	0.01923077
FF	10	41600	0.024038462
IO	10	106	9.433963
BUFG	1	32	3.125

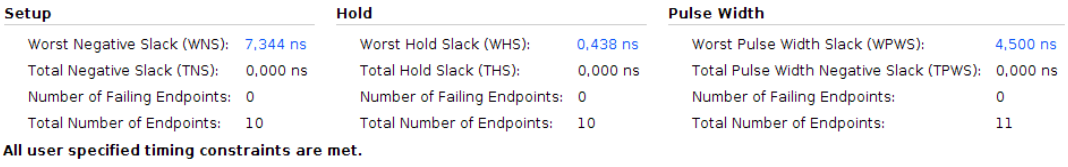

Figura 4.44 – Análise de *timing* referente ao circuito da FSM.

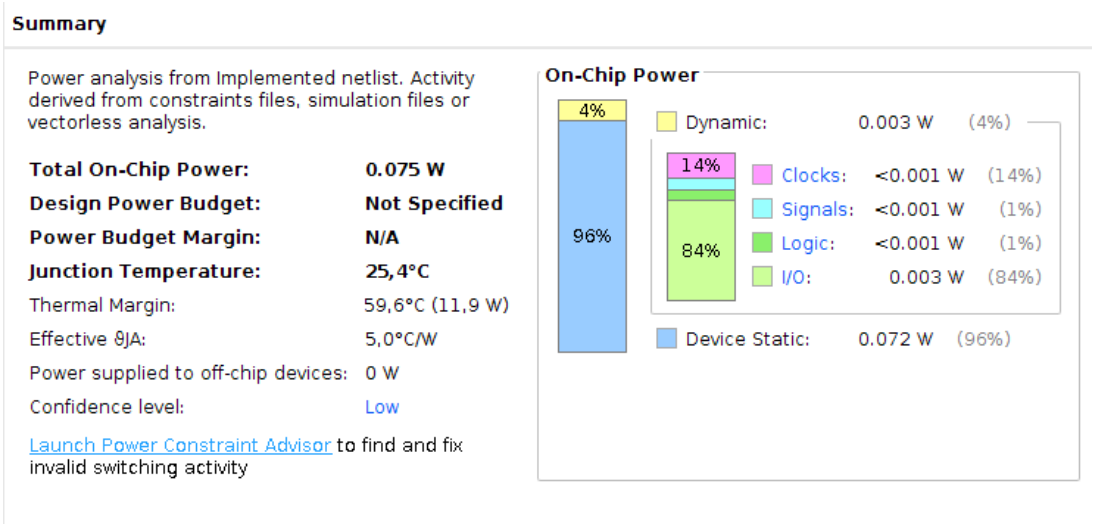

Figura 4.45 – Análise de potência referente ao circuito da FSM.

Assim, percebe-se que com o circuito gerado pela *vFSMgen* em conjunto com o Vivado foi possível realizar todas as etapas de criação de um circuito de uma FSM com sucesso.

4.2 Caso de Estudo de um Projeto RTL

Para demonstrar o funcionamento da ferramenta *vRTLgen* foi escolhida a implementação de um neurônio do tipo *Perceptron* como caso de estudo. A estrutura do *Perceptron* é mostrada na Figura 4.46, a qual consiste em valores de entradas multiplicadas por pesos de conexão, cujos resultados são acumulados e adicionados a um bias. Por fim, o resultado é utilizado como argumento de uma função de ativação. Neste projeto a função de ativação não será implementada e para as operações de multiplicação e adição serão utilizados os IPCores desenvolvidos pelos pesquisadores do grupo LEIA – GRACO (Grupo de Automação e Controle) do Departamento de Engenharia Mecânica (ENM) da Universidade de Brasília (MUÑOZ, D. M.; SÁNCHEZ et al., 2010).

Dessa forma, o objetivo é a implementação da seguinte equação,

$$y = ativ\_func\left(\left[\sum_{n=0}^{N} x_n \cdot w_n\right] + bias\right) \tag{4.1}$$

onde, x representam as entradas, w os pesos de conexão, N o número de entradas, e y a saída do neurônio.

Assim, foi implementado um *Perceptron* de 4 entradas ($N = 4$) e foi usada uma representação aritmética de ponto flutuante de 27 bits e uma saída de 32 bits. Essa configuração pode ser implementada de várias formas diferentes, porém nesse caso foram utilizados 2 multiplicadores de 27 bits que realizam as multiplicações entre as entradas e os pesos de forma paralela, sendo que as entradas e pesos são multiplexadas para que cada multiplicador

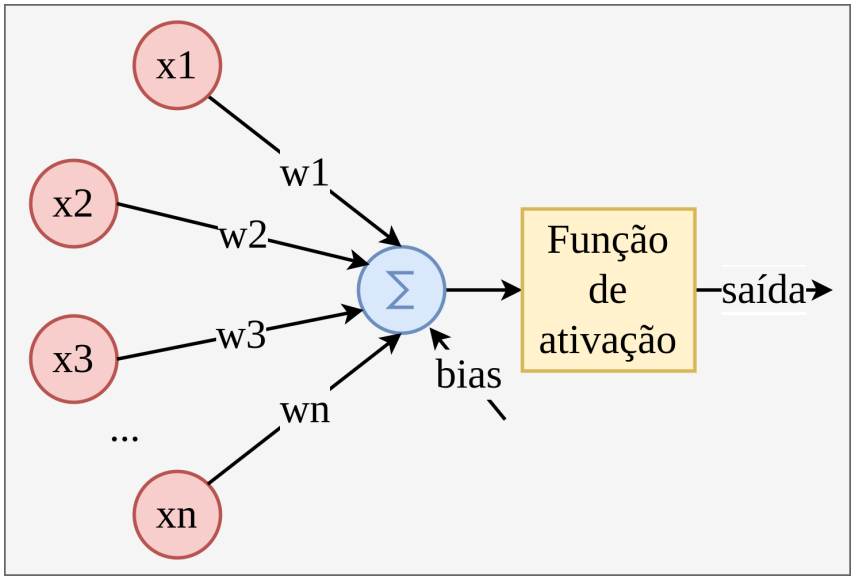

Figura 4.46 – Diagrama de funcionamento de um *Perceptron*.

efetue metade das operações. Cada uma das multiplicações é somada por um somador de 27 bits, acumuladas por um segundo somador, e o resultado do somatório é adicionado ao *bias*. O valor do *bias* foi implementado como uma constante para mostrar a capacidade do gerador de declarar constantes em VHDL. Finalmente, são adicionados 5 bits de valor '0' nos bits menos significativos da palavra para transformá-la em 32 bits.

A Figura 4.47 mostra o diagrama do *Perceptron* finalizado. Os elementos estão destacados com cores diferentes referente a cada tipo. Em vermelho destacam-se as entradas, as saídas em roxo claro, os componentes instanciados em azul, os geradores de código em amarelo e as constantes declaradas em verde e roxo escuro.

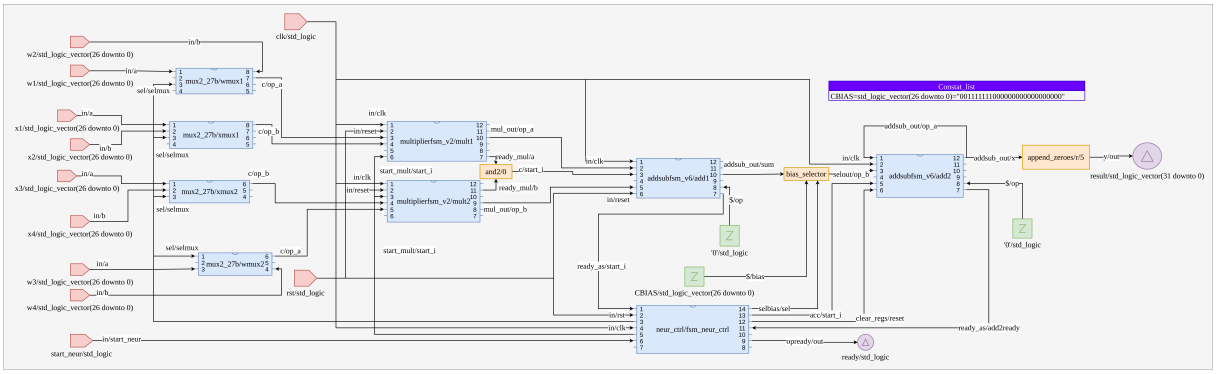

Figura 4.47 – Diagrama de um perceptron criado no Draw.io.

Para o funcionamento do *Perceptron* foi necessário criar uma FSM para controlar os muxes, multiplicações e adições realizadas. Para tal foi utilizada a ferramenta *vFSMgen* e a Figura 4.48 mostra o diagrama de funcionamento do controle da FSM. Este diagrama também foi utilizado como entrada da *vFSMgen*.

O diagrama é, enfim, submetido à *vRTLgen*, como mostra a Figura 4.49, sendo que

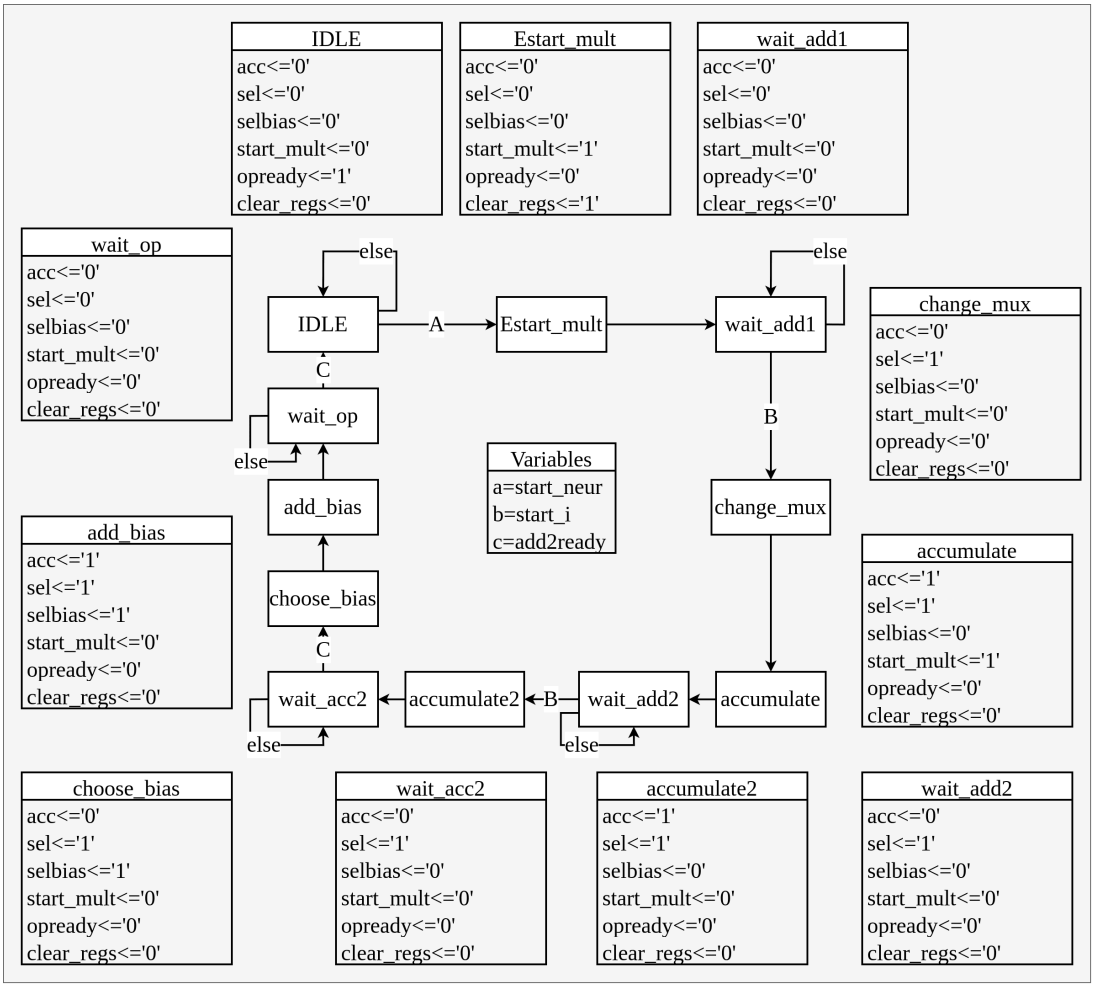

Figura 4.48 – Diagrama da FSM de controle.

à esquerda é mostrada a janela de seleção de diagramas e à direita o resultado tanto dos geradores quanto da análise sintática. O gerador em Python foi bem-sucedido e gerou o código mostrando no Anexo A.7. Entretanto, percebe-se que o gHDL não aponta para um erro na sintaxe dos arquivos, mas aponta para a falta do arquivo *fpupack*, ou seja, indica ao usuário um arquivo que deve ser adicionado ao executar as próximas etapas do fluxo de projeto.

Os códigos gerados foram adicionados ao Vivado e seu esquemático RTL foi obtido, como mostra a Figura 4.50. Percebe-se que o esquemático é muito semelhante ao diagrama utilizado como entrada da *vRTLgen*, sendo possível identificar as estruturas geradas pelos geradores de código como a porta *and* e o multiplexador do *bias*.

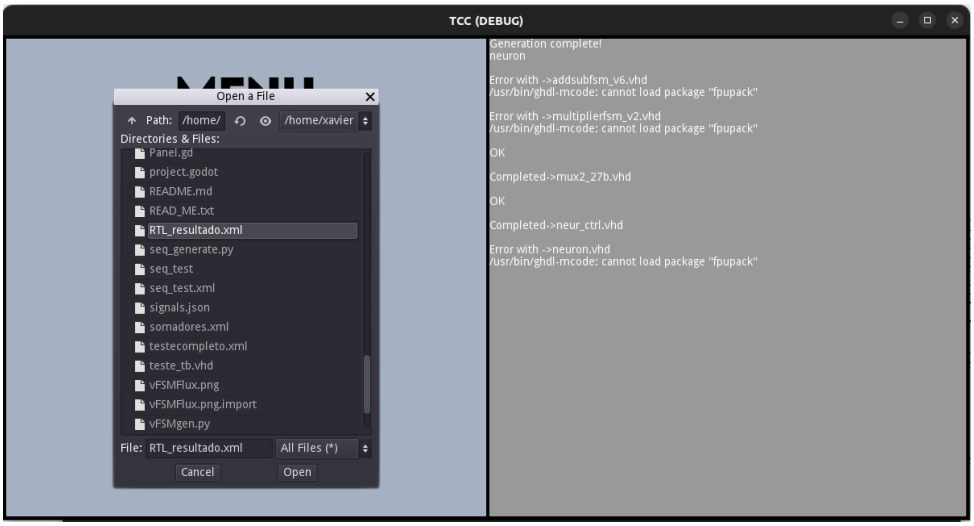

Figura 4.49 – Tela de escolha de diagramas do vRTLgen e resultado.

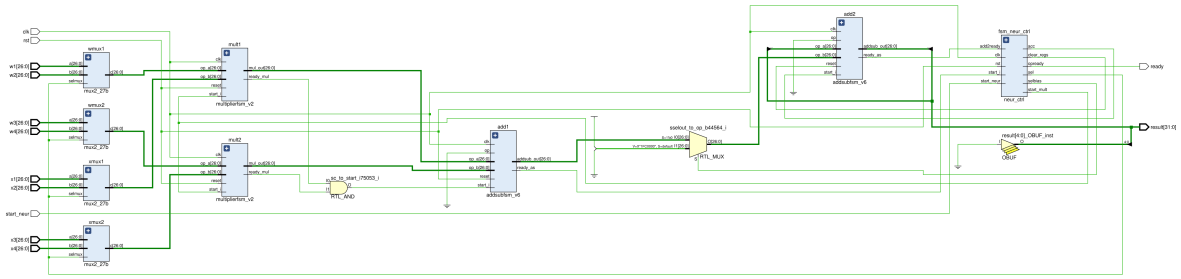

Figura 4.50 – Diagrama esquemático gerado pelo Vivado.

Para testar o funcionamento do *Perceptron* foi utilizada a ferramenta de criação de testbench *vTBgen*. Do lado esquerdo da figura 4.51 mostra-se como foi configurado o *testbench* e do lado direito apresenta-se o código obtido (vide Anexo A.8). São utilizadas constantes nas entradas e pesos, um *clock* de 100MHz, um sinal variável no *reset* e na entrada que inicia os cálculos. Por fim, são conectadas duas saídas, uma indicando o resultado em 32 bits em ponto flutuante, e outra que indica se o resultado está pronto.

A Figura 4.52 mostra o resultado da simulação do circuito obtido. Após a inicialização do circuito, um sinal para iniciar as operações. O valor das entradas foi configurado em "0x2000000", 2.0 em base 10, os pesos em "0x2020000", 3.0 em base 10, e o valor do *bias* em 1.0, como mostra a Figura 4.47. O resultado final foi de 25.0, cuja representação em hexadecimal é "0x41c80000", confirmando a corretude lógica da implementação. Por fim, é gerado mais um pulso para iniciar novamente os cálculos e o mesmo resultado é obtido, comprovando que o circuito está funcionando corretamente.

Visto que a verificação do funcionamento do *Perceptron* foi bem-sucedida, pode-se implementá-lo em uma *Basys3*. Entretanto, não há como mapear todas as entradas e saídas do *Perceptron* na *Basys3* devido à grande quantidade de pinos de IO requetidos, sendo necessário realizar alguns ajustes.

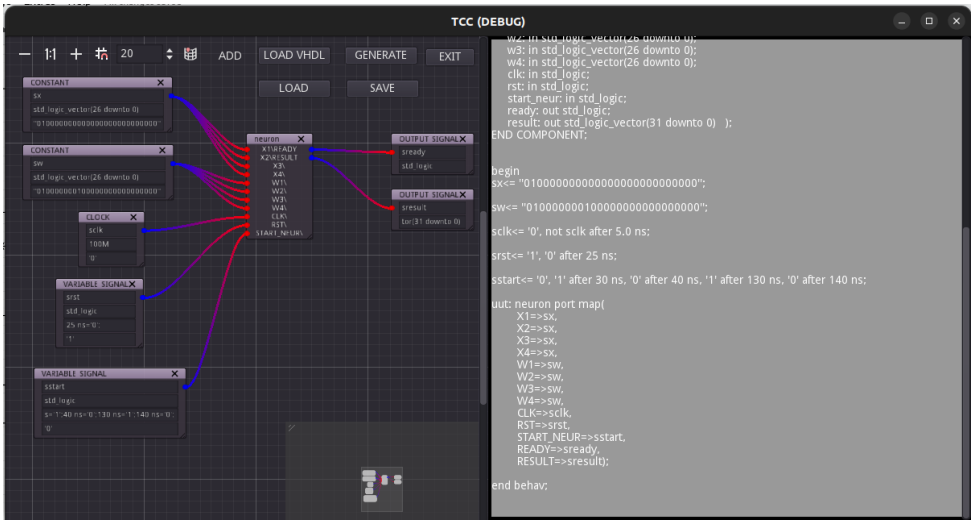

Figura 4.51 – Configuração do *tesbench* para o *Perceptron*.

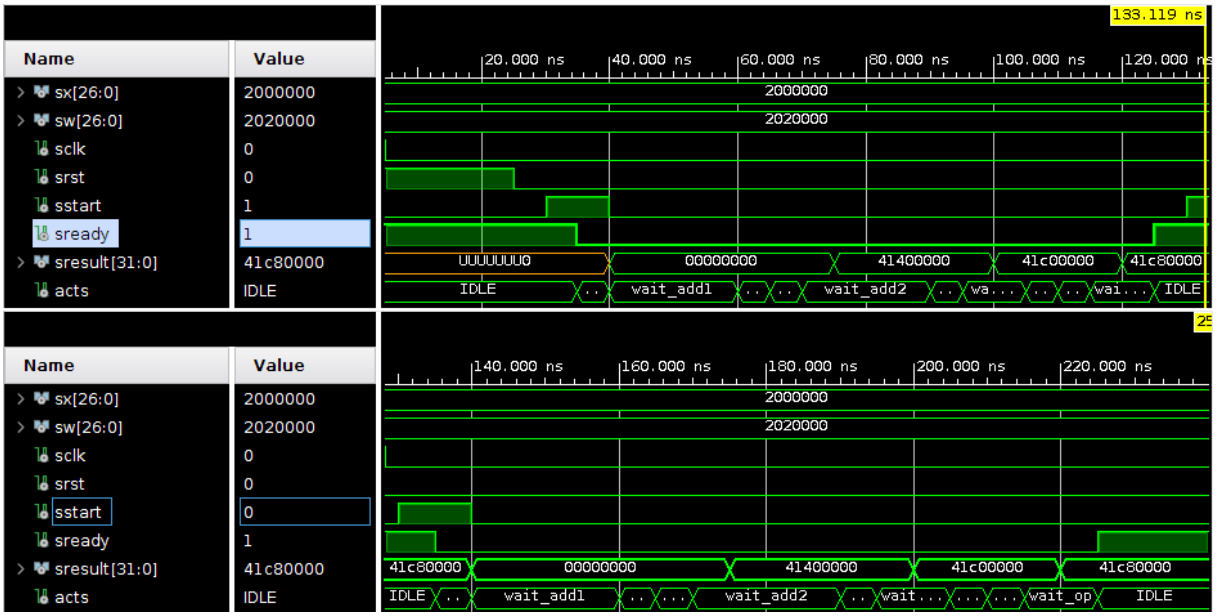

Figura 4.52 – Resultado da simulação do *testbench*.

A ferramenta *vRTLgen* possui a capacidade de reusabilidade de códigos. O código gerado a partir da Figura 4.47 é automaticamente adicionado à biblioteca de códigos da ferramenta e pode ser prontamente reutilizado. A Figura 4.53 mostra o circuito chamado de *topmodule* que é implementado na Basys3, sendo que o funcionamento do circuito se resume a um *Perceptron* que recebe constantes W1, W2, W3 e W4 em conjunto com as entradas X1, X2, X3 e X4 geradas pelo gerador *gen_in* que, por sua vez, recebe uma *seed* e gera dois conjuntos de entradas que podem ser selecionadas por um *switch*. Após os cálculos, a palavra resultante de 32 bits alimenta *choose_out* que divide o resultado em palavras de 16 bits que são mostradas nos leds da Basys3, sendo possível escolher a parte alta ou a parte baixa a partir de um *switch* visto que a o sinal *ready* é mostrado no bit menos significativo da parte baixa.

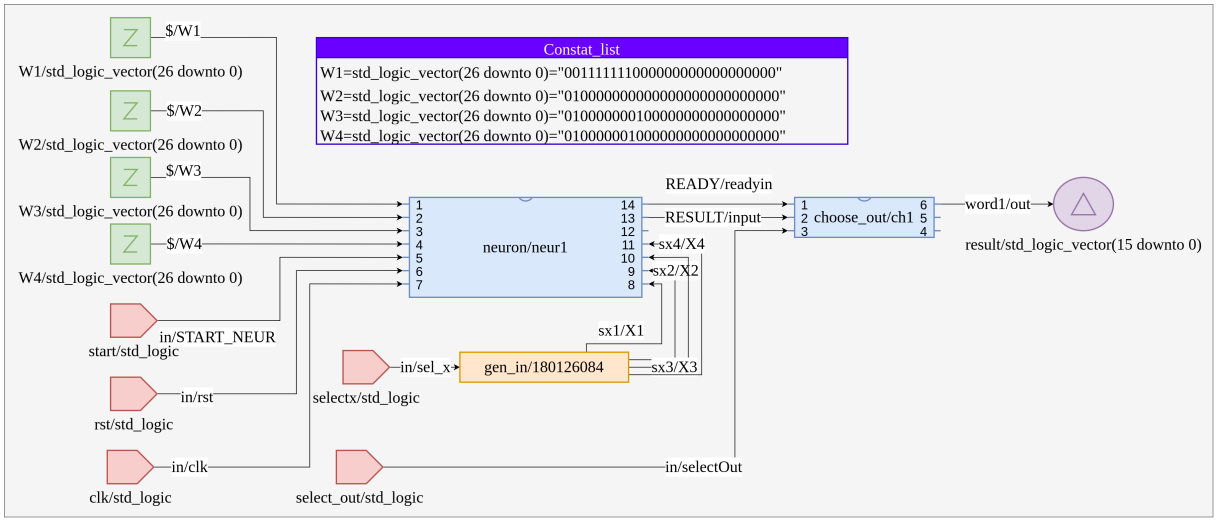

Figura 4.53 – Diagrama no Draw.io dp circuito implementado na Basys3.

O *vRTLgen* gerou o código apresentado no Anexo A.9 quando alimentado com o diagrama da Figura 4.53, e a partir deste código o diagrama RTL da Figura 4.54 foi gerado no Vivado. Percebe-se novamente, que o diagrama é muito semelhante ao diagrama montado no Draw.io, sendo que nesse caso o *gen_in* gerou os multiplexadores mostrados na figura.

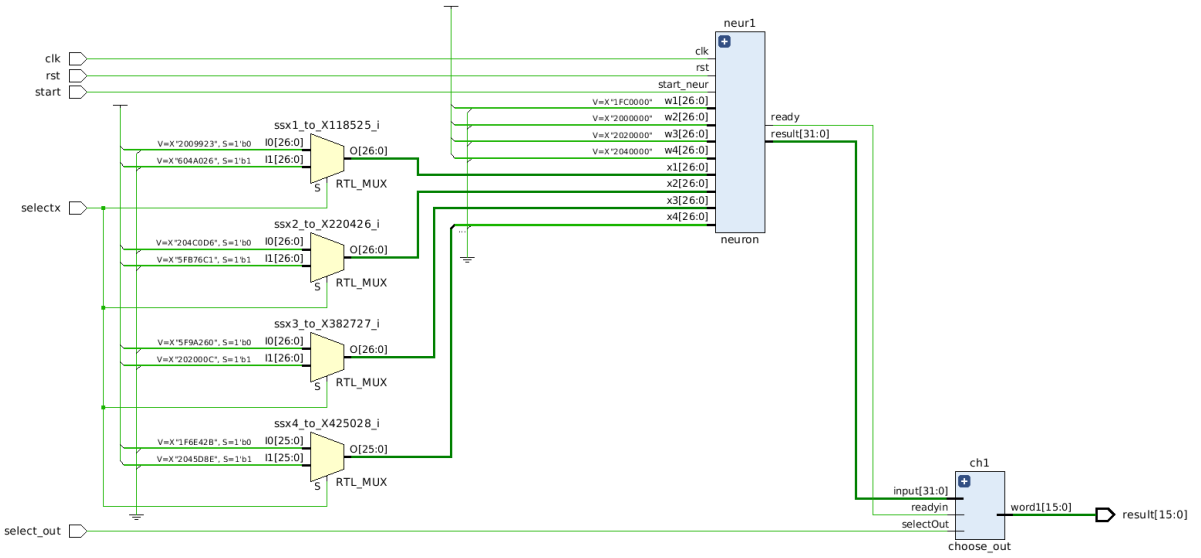

Figura 4.54 – Circuito implementado na Basys3.

Nesta implementação é muito importante saber quais são os valores de entrada do circuito para verificar o funcionamento do circuito. Felizmente, a ferramenta *vRTLgen* é capaz de imprimir qualquer texto que esteja presente em funções *print* dentro dos geradores de código, desta forma o usuário pode criar avisos e notificações personalizadas. Por exemplo, como mostra a Figura 4.55, o gerador *gen_in* imprime na tela o valor de cada variável, bem como o resultado aproximado que deve ser obtido.

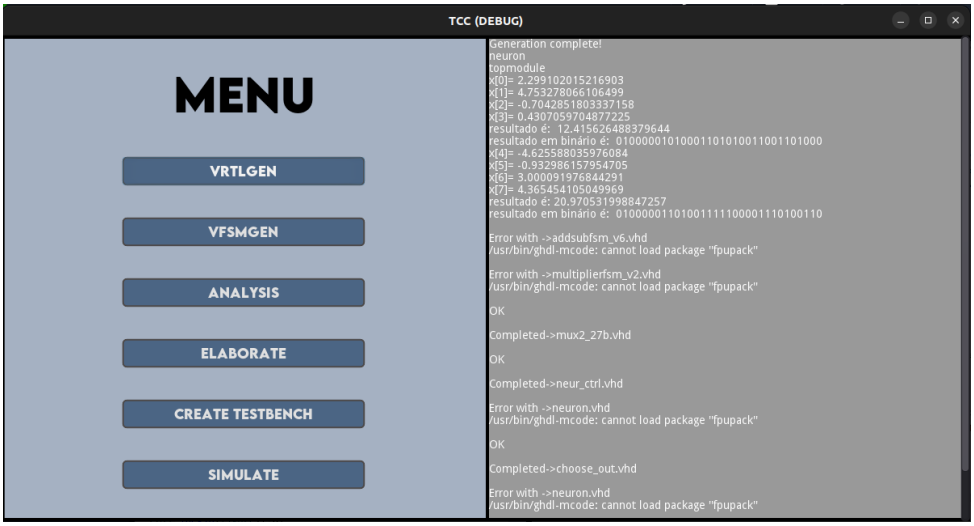

Figura 4.55 – Valores de entrada e resultados notificados pelo gerador de código.

Novamente, o *vTBgen* é utilizado para gerar o *testbench* do circuito, como mostra a Figura 4.56 que gera o código A.10. Este testbench funciona de maneira bem simples, os cálculos do primeiro conjunto de entradas são realizados após o *reset* do circuito e a parte alta e baixa da palavra do resultado são mostradas na saída, logo após trocam-se a entrada e repete-se o procedimento.

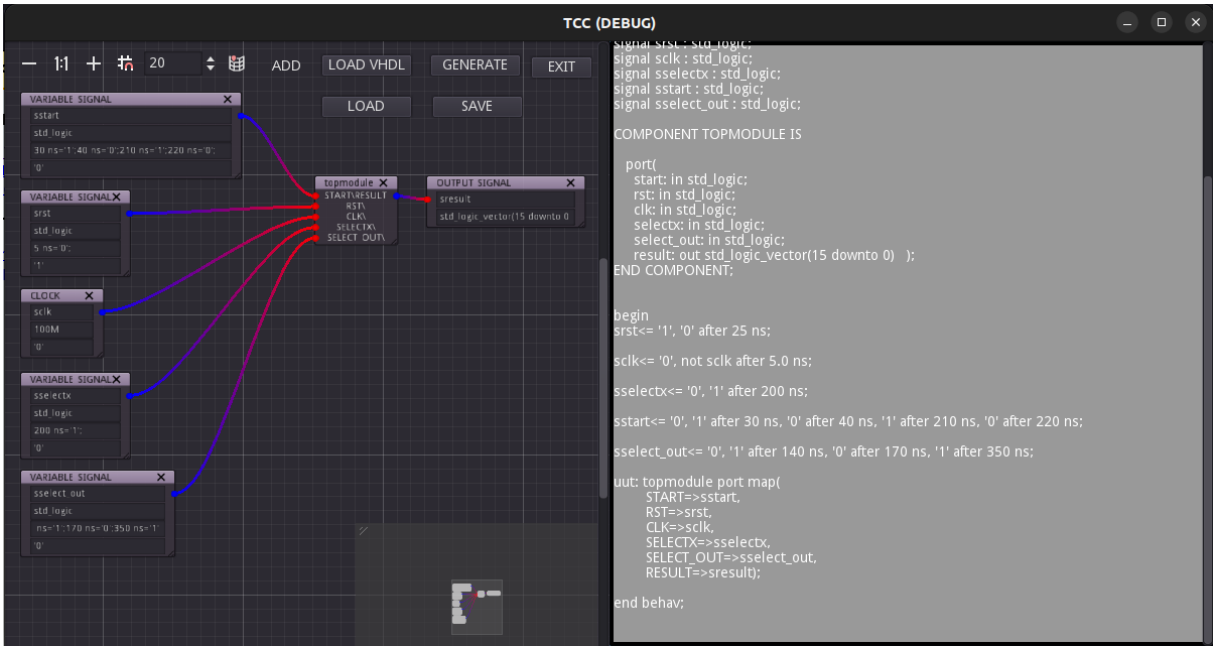

Figura 4.56 – Diagrama do *testbench* do *topmodule*.

A Figura 4.57 mostra a simulação do *testbench*. Após o *reset*, *sstart* é acionado e após alguns nanosegundos a saída 0xa621 aparece na saída. Ao trocar o valor de *sselect_out*, a saída se transforma em 0x4146. Quando as duas saídas (alta e baixa) são combinadas, ignorando o bit menos significativo, e transformados em *float*, representam o valor 12.415557861328125 que é próximo do que é mostrado na Figura 4.55. Logo após, trocam-se os valores de entrada

e recomeçam os cálculos, gerando ao final do processo o valor 0x41a7c3a0 que representa 20.97052001953125 e novamente se confirma a obtenção de valores bem próximos, bem como o funcionamento do circuito. A diferença entre a saída obtida e a saída esperada é devido a que a saída esperada é calculada em software usando representação de 64 bits em ponto flutuante. As etapas para chegar a este resultado pode ser conferido neste vídeo .

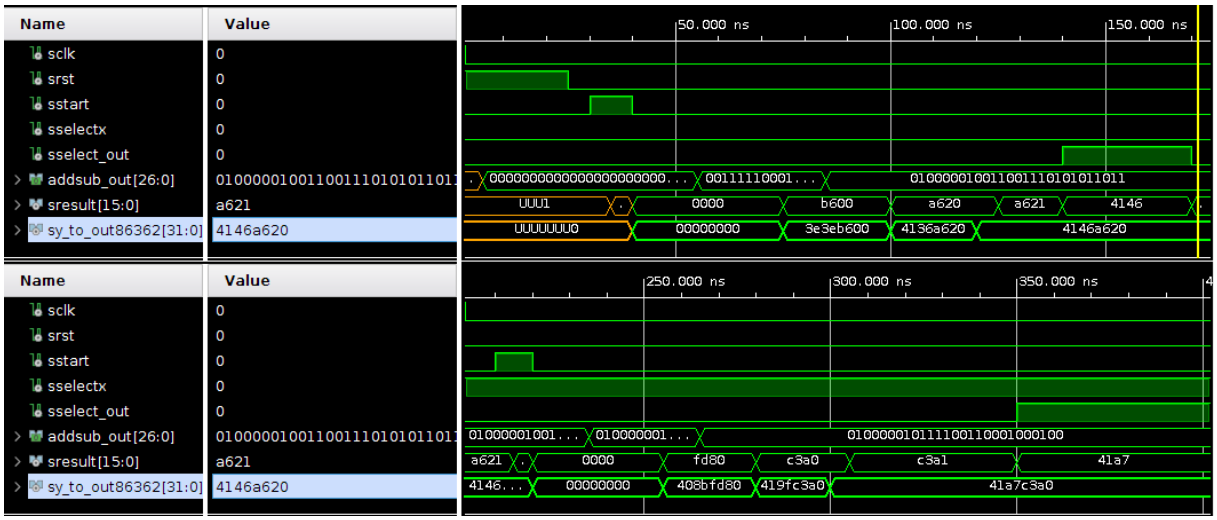

Figura 4.57 – Simulação do *testbench* do *topmodule*.

Com o comportamento do circuito verificado foi possível criar um arquivo de *constraints* que utiliza um *clock* de 50MHz, conecta as entradas em botões e *switches*, e conecta as saídas nos leds da placa Basys3.

Após a implementação do circuito utilizando o *Vivado*, a utilização de recursos é mostrada na Figura 4.58. Observa-se uma alta utilização de recursos, porém isso é esperado pelo uso de somadores e multiplicadores em ponto flutuante.

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)
topmodule	806	171	234	806	2	21	1
neur1 (neuron)	802	171	234	802	2	0	0
add1 (addsubfsm_v6)	205	51	83	205	0	0	0
add2 (addsubfsm_v6_0)	273	51	94	273	0	0	0
fsm_neur_ctrl (neur_ctrl)	102	11	57	102	0	0	0
mult1 (multiplierfsm_v2)	120	29	46	120	1	0	0
mult2 (multiplierfsm_v2_1)	147	29	57	147	1	0	0

Figura 4.58 – Utilização de recursos do circuito.

A área utilizada pelo circuito é mostrada na Figura 4.59 e, como esperado, o *Perceptron* utiliza maior parte da área, sendo que ele está destacado em vermelho.

A utilização do *clock* de 50MHz garante que o circuito obedeça a critérios de *timing*, assim, os tempos de *setup* e *hold* são mostrados na Figura 4.60. Finalmente, a Figura 4.61 mostra uma estimativa do consumo de energia do circuito, sendo 73mW de potência total, dos quais 70mW são de potência estática e 3mW de potência dinâmica.

Novamente, a partir de diagramas oriundos do Draw.io em conjunto com as ferramentas *vRTLgen*, *vFSMgen* e *vTBgen* foi possível realizar todas as etapas de projeto necessárias para desenvolver um circuito em FPGA. Inclusive, foram facilmente reutilizados circuitos de etapas anteriores para gerar e testar novos designs.

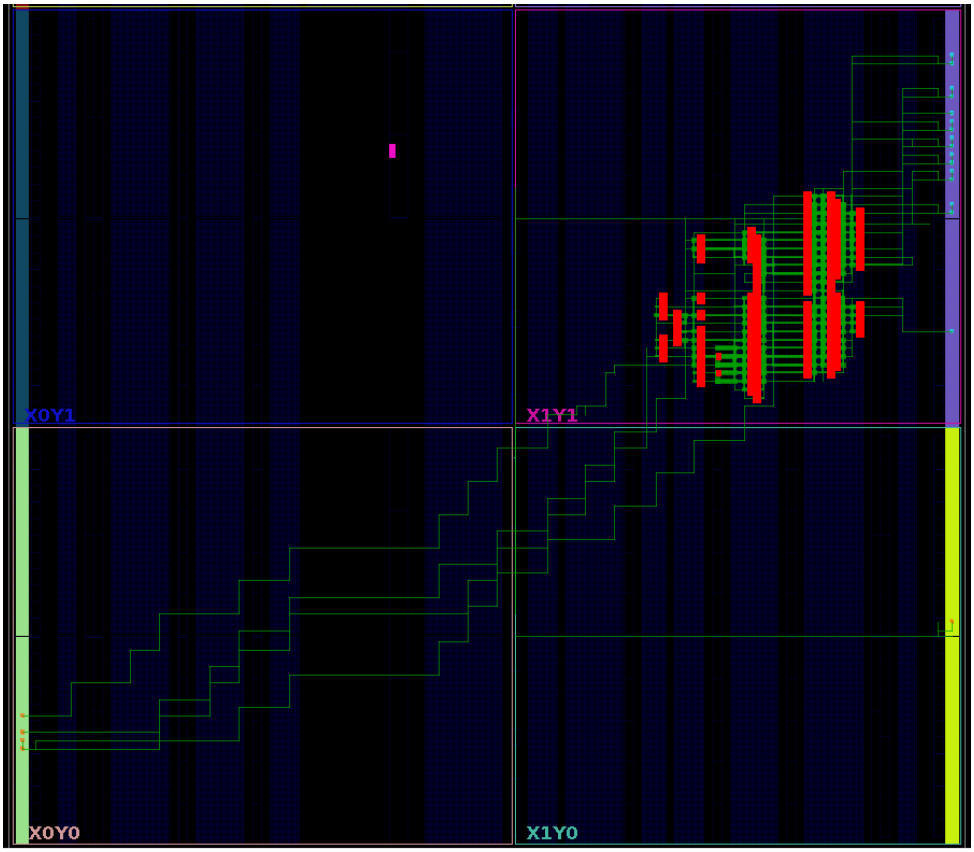

Figura 4.59 – *Place and Route* do *topmodule*.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7,133 ns	Worst Hold Slack (WHS):	0,184 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	325	Total Number of Endpoints:	325	Total Number of Endpoints:	172

All user specified timing constraints are met.

Figura 4.60 – Sumário de análise de *timing* do circuito.

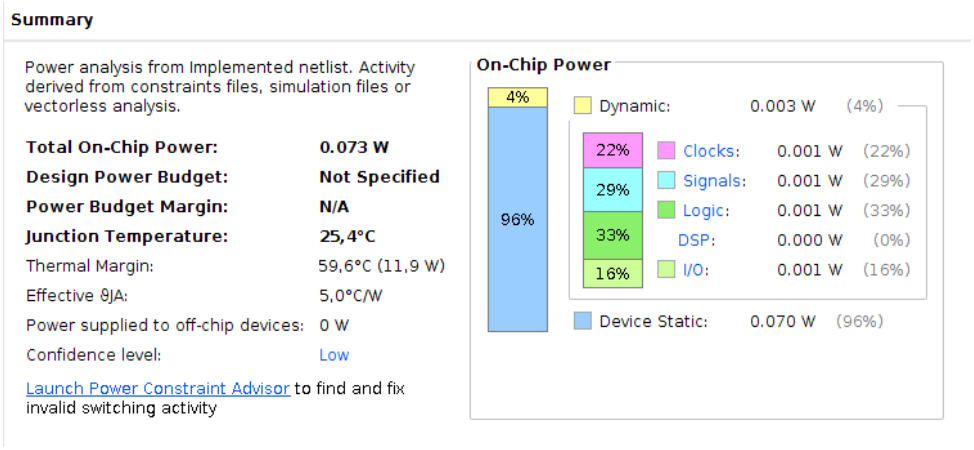

Figura 4.61 – Utilização de potência do circuito.

4.3 Caso de estudo encapsulamento com AXI4-Lite

Para demonstrar o funcionamento da ferramenta *AXIcreator* foi utilizado o mesmo circuito do perceptron da Figura 4.47, porém com a saída de 27 bits ao invés de 32 bits, ou seja, retirou-se a função “append_zeroes” do diagrama. Assim, o circuito resultante possui uma entrada de “clock”, uma entrada de “reset”, uma entrada “start_neur” que começa os cálculos do perceptron, 8 entradas de 27 bits para as entradas e pesos do perceptron, uma saída “ready” e uma saída de 27 bits que representa o resultado.

A configuração para este circuito no *AXIcreator* é mostrada na Figura 4.62. Percebe-se que o tamanho da porta ou *port size* é configurado para 27 bits, são utilizados 9 registradores para entradas e saídas, o “clock” e o “reset” da interface são conectados em suas portas equivalentes do código encapsulado, bem como o “START” e o “READY”. Por fim, os 4 primeiros registradores são conectados nas entradas, os próximos 4 são conectados aos pesos e o último é conectado à saída do código encapsulado. O processo utilizado para encapsular o código pode ser conferido neste vídeo .

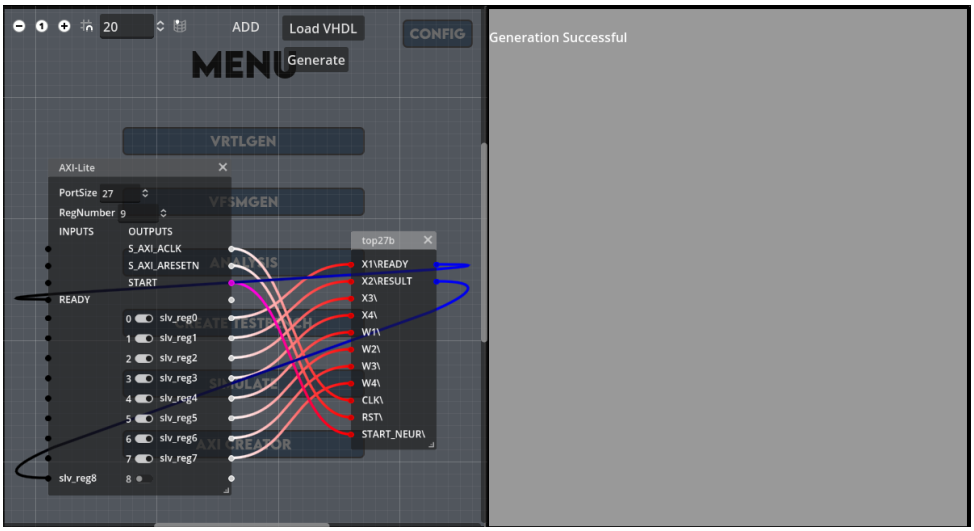

Figura 4.62 – Conexões realizadas no *AXIcreator*.

Os códigos gerados, apresentados nos Anexos A.11 e A.12, foram adicionados ao Vivado em um *Block Design* com um *ZYNQ Processing System* para realizar a conexão com a placa Zedboard e, neste caso, a interface AXI é automaticamente reconhecida ao adicionar o IP do *Perceptron* no *Block Design*, como mostrado na Figura 4.63.

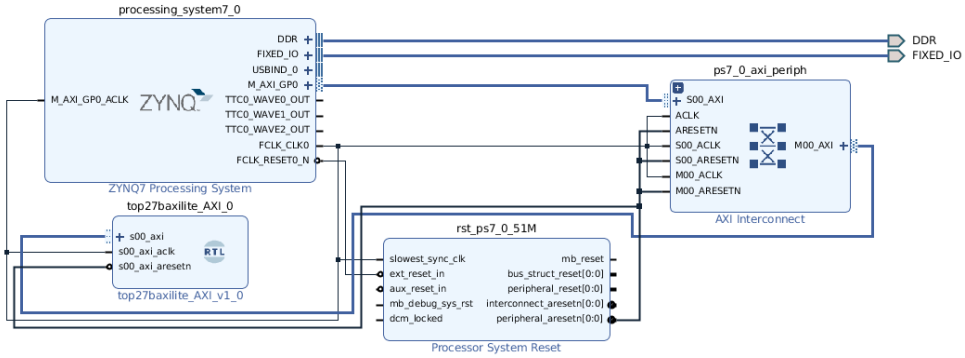

Figura 4.63 – *Block Design* gerado pelo Vivado com a interface AXI4-Lite. A interface criada possui o nome de "top27baxilite_AXI_0".

Este circuito foi implementado com a frequência do "FCLK_CLK0" de 50MHz para garantir que atenda aos critérios de *timing*. Assim, a implementação e a geração de *bitstream* foi bem sucedida, sendo que o circuito atende os critérios de *timing* como mostra a Tabela 4.4, a utilização de recursos do circuito é mostrada na Figura 4.64 com maior parte dos recursos concentrados na interface AXI.

Tabela 4.4 – Valores pós-implementação de timing.

WNS (ns)	TNS (ns)	WHS (ns)	THS (ns)	WPWS (ns)	TPWS (ns)
6.362	0	0.051	0	8.823	0

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)
design_neuronio_Lite_wrapper	1433	937	477	1373	60	2	130	1
design_neuronio_Lite_i (design_neuronio_Lite)	1433	937	477	1373	60	2	0	1
processing_system7_0 (design_neuronio_Lite	0	0	0	0	0	0	0	1
ps7_0_axi_periph (design_neuronio_Lite_ps7_	350	428	140	291	59	0	0	0
rst_ps7_0_51M (design_neuronio_Lite_rst_ps7	16	33	11	15	1	0	0	0
top27baxilite_AXI_0 (design_neuronio_Lite_top	1067	476	331	1067	0	2	0	0

Figura 4.64 – Utilização do circuito pós-implementação.

A Figura 4.65 mostra o *Place and Route* do circuito e a Figura 4.66 mostra o consumo de energia do circuito que é relativamente alto, porém é o esperado em um SoC.

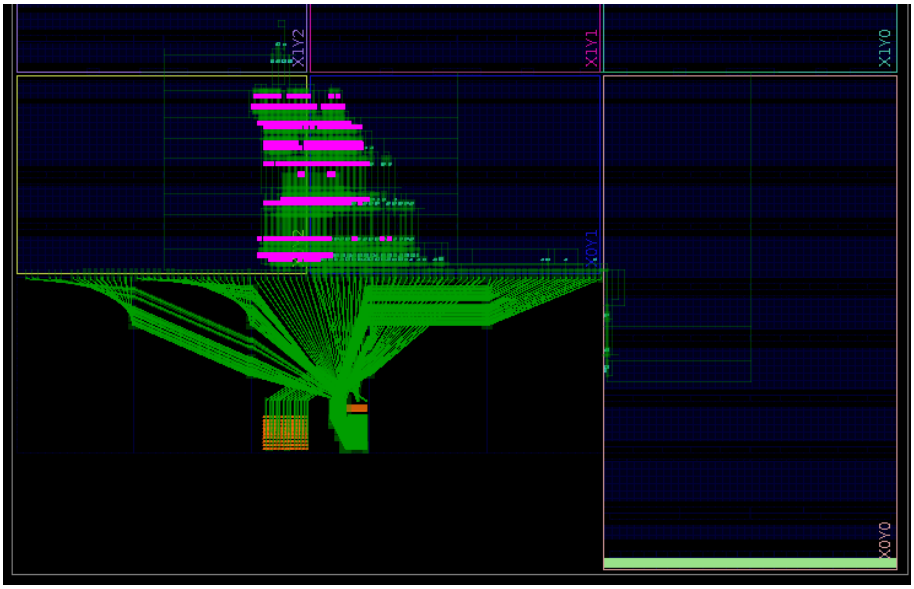

Figura 4.65 – *Place and Route* do circuito com a interface AXI destacada em roxo.

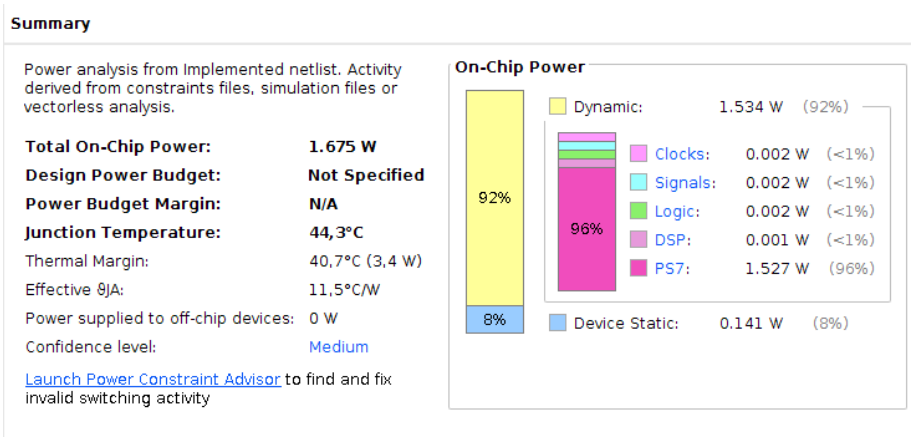

Figura 4.66 – Uso de energia do circuito.

Com a caracterização do circuito concluída, a validação em hardware utilizando a placa Zedboard foi realizada utilizando o código em C mostrado no Anexo A.15, que testa o circuito com 2 conjuntos de valores diferentes mostrados na Tabela 4.5.

Tabela 4.5 – Valores de entrada e saída do perceptron.

Entrada	Conjunto 1 hex(27b)	Valor(float32)	Conjunto 2 hex(27b)	Valor(float32)
X1	1FC0000	1	1FC0000	1
X2	2000000	2	2000000	2
X3	2020000	3	2020000	3
X4	2040000	4	2040000	4
W1	1FE0000	1,5	1FE0000	1,5
W2	1FC0000	1	1FC0000	1
W3	2006666	2,2	2006666	2,2
W4	2020000	3	2030000	3,5
Valor esperado	41B8CCC0	23,1	41c8ccc0	25,1

O resultado da operação é escrito na porta serial, como mostra a Figura 4.67, assim,

verificando que o circuito funciona corretamente. O processo de validação pode ser verificado neste vídeo.

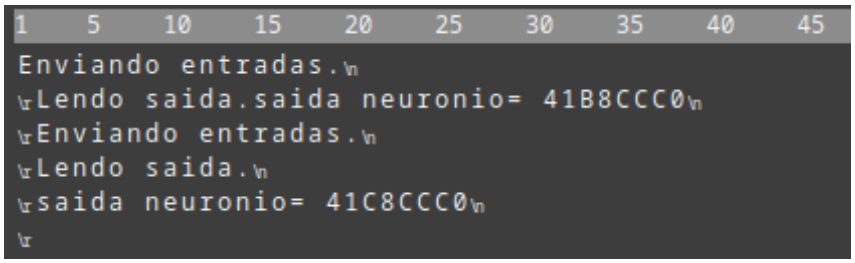

Figura 4.67 – Resultados obtidos com o encapsulamento AXI4-Lite.

4.4 Caso de estudo encapsulamento com AXI4-Full

Para a demonstração do AXI4-Full foi utilizado um *IP Core* de uma rede neural artificial perceptron de múltiplas camadas ou MLP, sendo que este possui 4 perceptrons na camada de entrada, 4 perceptrons na camada escondida e 3 na camada de saída. Assim, são 4 entradas de 27 bits, 3 saídas de 27 bits e seus pesos já estão definidos internamente.

Para esta MLP a configuração escolhida no AXIcretor é mostrada na Figura 4.68. Esta mostra que o *port size* está configurado para 27 bits, utiliza-se uma memória de 64 bytes ou 16 palavras de 32 bits, o *clock* e o *reset* da interface são ligados em suas portas equivalentes do código encapsulado, bem como o "START" e o "READY". Por fim, são conectadas as entradas e saídas, sendo que as entradas são um *array* de tamanho 4 indicado pelo texto "0->3" com o nome de "sx" na ferramenta e as saídas são um *array* de tamanho 3 indicado pelo texto "0->2" com o nome de "ssaida" na ferramenta.

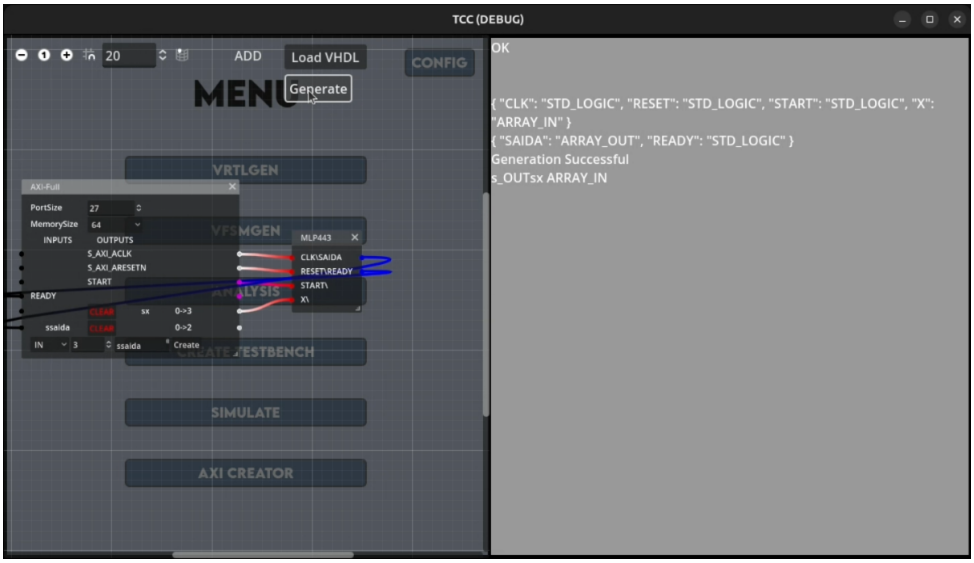

Figura 4.68 – Conexões realizadas no *AXIcreator*.

Os códigos gerados, apresentados nos Anexos A.13 e A.14, foram adicionados ao Vivado em um Block Design com um ZYNQ Processing System para realizar a conexão

com a placa Zedboard e, neste caso, a interface AXI foi automaticamente reconhecida ao adicionar o IP no Block Design, resultando no circuito da Figura 4.69. Percebe-se que quando utiliza-se uma interface AXI4-Full o Vivado utiliza o *AXI SmartConnect* ao invés do *AXI Interconnect*. O processo para chegar a este resultado pode ser conferido neste vídeo.

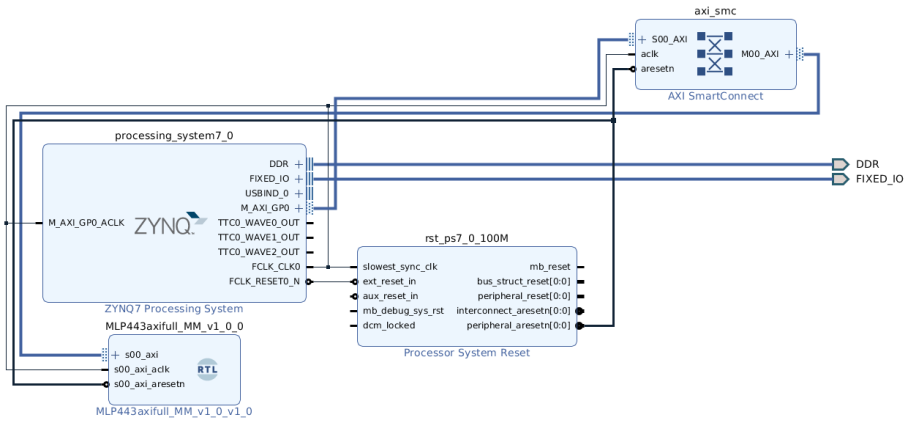

Figura 4.69 – *Block Design* gerado pelo Vivado com a interface AXI4-Full. A interface criada possui o nome de “MLP443axifull_MM_v1_0_0”.

Este circuito foi implementado com a frequência padrão de “FCLK_CLK0” de 100MHz com sucesso e gerado o *bitstream*. Assim, os critérios de *timing* foram atendidos como mostra a Tabela 4.6. A utilização de recursos é mostrada na Figura 4.70, como esperado, uma MLP utiliza uma grande quantidade de recursos.

Tabela 4.6 – Valores pós-implementação de timing.

WNS (ns)	TNS (ns)	WHS (ns)	THS (ns)	WPWS (ns)	TPWS (ns)
0.119	0	0.043		0 3.651	0

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)
design_MLP_wrapper	12819	2842	464	74	3702	12579	240	14	130	1
design_MLP_i (design_MLP)	12819	2842	464	74	3702	12579	240	14	0	1
axi_smc (design_MLP_axi_s	1071	764	1	0	333	832	239	0	0	0
MLP443axifull_MM_v1_0_0	11734	2045	463	74	3394	11734	0	14	0	0
processing_system7_0 (de	0	0	0	0	0	0	0	0	0	1
rst_ps7_0_100M (design_M	16	33	0	0	9	15	1	0	0	0

Figura 4.70 – Utilização de recursos do circuito.

A Figura 4.71 mostra o *Place and Route* do circuito com uma utilização de área relativamente alta e a Figura 4.72 mostra o consumo de energia do circuito que é relativamente alto, porém é o esperado em um SoC, ainda mais com um circuito relativamente grande.

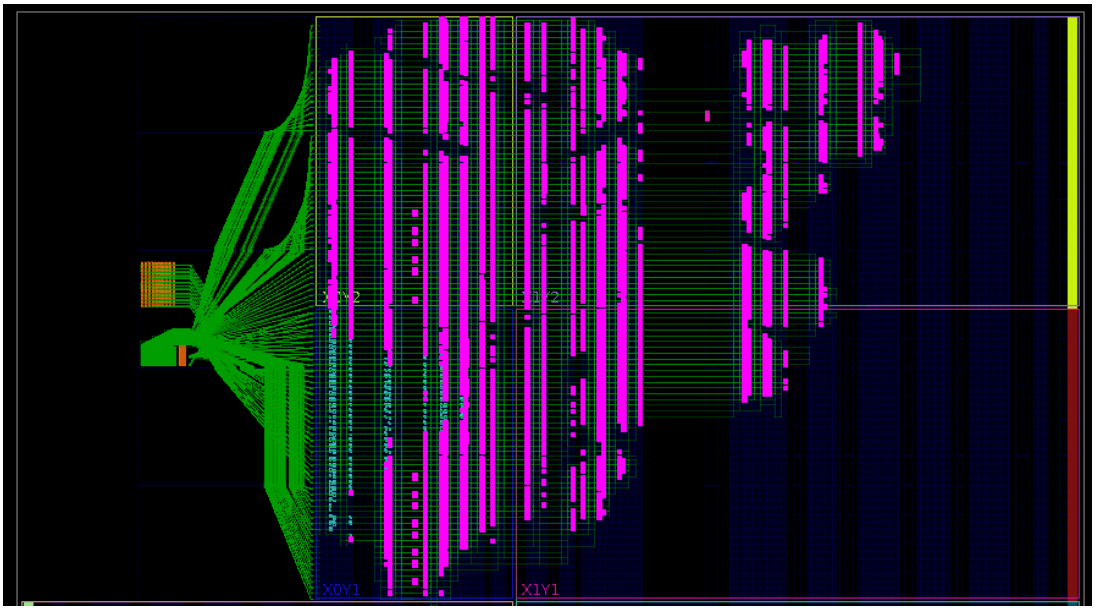

Figura 4.71 – *Place and Route* do circuito com a interface AXI destacada em rosa.

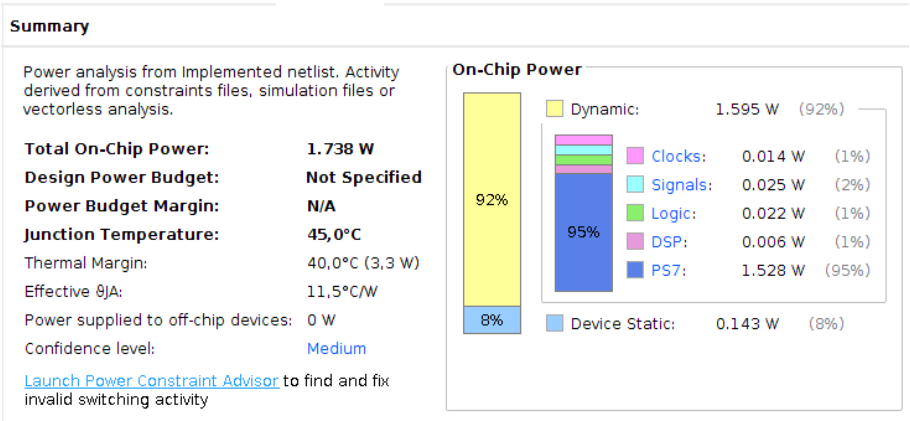

Figura 4.72 – Energia utilizada pelo circuito.

Com a caracterização do circuito concluída a validação em hardware utilizando a placa Zedboard foi realizada utilizando o código doo Anexo A.16 que testa a MLP com os valores de entrada ‘1.0’, ‘2.0’, ‘3.0’ e ‘4.0’. Para estes valores, segundo a simulação do circuito, deve-se encontrar os resultados em hexadecimal de “390EFEA0”, “3DCC49E0”, “3F7EBA60”. Assim, o funcionamento é validado pela saída serial que imprime os valores mostrados na Figura 4.73. O processo de validação pode ser conferido neste vídeo.

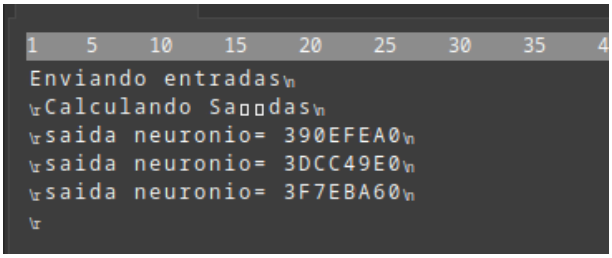

Figura 4.73 – Resultados da MLP impressos na porta serial.

Assim, finalizam-se as demonstrações referentes ao protocolo AXI. Percebe-se que é possível facilmente encapsular um código VHDL, sendo necessário somente que ele possua portas de “START”, “READY” e todas as saídas e entradas de mesmo tamanho. Conclui-se também que com a aplicação HiLDA é possível criar um design complexo, testá-lo e integrá-lo a um SoC de uma forma relativamente simples.

5 Conclusões

O Draw.io se mostrou como uma aplicação promissora para ser utilizada como base de geradores de códigos. A diversidade de elementos e a capacidade de facilmente exportar diagramas elimina a necessidade de criar uma aplicação específica para criá-los, pelo menos ao nível de prototipação, sendo que para geradores voltados para o meio profissional é desejável uma aplicação específica.

A ferramenta *vFSMgen* foi executada com sucesso dentro das limitações propostas. Até o momento já é possível criar diagramas sintetizáveis e implementáveis de FSMs do tipo *Moore* e *Mealy*. Por fim, adições desejáveis a essa ferramenta são entradas e saídas de qualquer tipo e não somente binárias, assim, é possível criar FSMs mais complexas.

A ferramenta *vRTLgen* no momento é funcional, assim como mostrado nos resultados, já é possível criar circuitos relativamente complexos sintetizáveis, implementáveis em FPGAs e, também, validados em hardware utilizando a placa Zedboard. No entanto, a ferramenta não possui suporte para entradas genéricas e o seu principal ponto fraco são diagramas com muitos elementos, ao utilizá-la é recomendado que o circuito seja dividido em pedaços menores para facilitar a criação das conexões.

A ferramenta *AXIcreator* possui atualmente a capacidade de encapsular códigos utilizando tanto a interface AXI4-Lite quanto a AXI4-Full. Como demonstrado anteriormente, a ferramenta é funcional e cria códigos que funcionam em hardware. No entanto, esta ferramenta tem algumas limitações, como suportar apenas o modo escravo e limitar o uso a apenas uma interface AXI4-Full ou AXI4-Lite. Adicionalmente, a forma de encapsular um componente esta condicionada ao uso de uma entrada de *start* e uma saída *ready*.

A interface gráfica responsável pela integração de ferramentas funciona de forma satisfatória, sendo que no momento é possível acessar as ferramentas *vFSMgen*, *vRTLgen*, *vTBgen*, *AXIcreator*, a funcionalidade de análise do gHDL, bem como exercer simulações utilizando o gHDL.

Com isso todos os objetivos do trabalho foram alcançados satisfatoriamente. A ferramenta HiLDA tem grande potencial de aplicação em ambientes educacionais e de pesquisa, além de grandes possibilidades de expansão de suas funcionalidades. Por exemplo, facilmente podem ser adicionados códigos VHDL e novos geradores de códigos à vRTLgen. Novas topologias para encapsulamento utilizando AXI4 também podem ser criadas. Por fim, pode-se aprimorar a integração da ferramenta HiLDA com o Vivado utilizando-o apenas a partir de *scripts*.

Referências

AMD. **AXI Vivado Reference AXI Reference Guide**. AMD, jul. 2017. Disponível em: <https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide>. Citado nas pp. 20, 21.

AMD. **Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)**. AMD, mai. 2023a. Disponível em: <https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration/Navigating-Content-by-Design-Process>. Citado na p. 17.

AMD. **Zynq 7000 SOC**. 2023b. Disponível em: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productAdvantages>. Citado na p. 19.

ARBOLEDA, D. M. M.; LLANOS, C. H.; COELHO, L. d.; AYALA-RINCÓN, M. Hardware Architecture for Particle Swarm Optimization Using Floating-Point Arithmetic. In: 2009 Ninth International Conference on Intelligent Systems Design and Applications. 2009. P. 243–248. DOI: 10.1109/ISDA.2009.107. Citado na p. 22.

AVNET. **Zedboard | Avnet Boards**. AVNET, 2023. Disponível em: <https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/>. Citado na p. 26.

AYALA, H. V. H.; MUÑOZ, D. M.; LLANOS, C. H.; SANTOS COELHO, L. dos. Efficient hardware implementation of radial basis function neural network with customized-precision floating-point operations. **Control Engineering Practice**, v. 60, p. 124–132, 2017. ISSN 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2016.12.004. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0967066116302921>. Citado na p. 22.

BALID, W.; ABDULWAHED, M. A novel FPGA educational paradigm using the next generation programming languages case of an embedded FPGA system course. In: 2013 IEEE Global Engineering Education Conference (EDUCON). 2013. P. 23–31. DOI: 10.1109/EduCon.2013.6530082. Citado na p. 13.

BERKOWITZ, E. **Why should you use an industrial embedded system for production automation?** Fev. 2023. Disponível em: <https://www.cprime.com/resources/blog/why-should-you-use-an-industrial-embedded-system-for-production-automation/>. Citado na p. 13.

CHU, P. P. **FPGA prototyping by VHDL examples: Xilinx spartan-3 version**. Wiley-Interscience, 2008. Citado nas pp. 17, 18.

COUSSY, P.; GAJSKI, D. D.; MEREDITH, M.; TAKACH, A. An Introduction to High-Level Synthesis. **IEEE Design & Test of Computers**, v. 26, n. 4, p. 8–17, 2009. DOI: 10.1109/MDT.2009.69. Citado na p. 19.

DIGILENT. **Basys 3 artix-7 FPGA trainer board: Recommended for introductory users**. Digilent, 2023. Disponível em: <https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/>. Citado na p. 25.

EB. **Xilinx Artix Arty**. EB, nov. 2015. Disponível em: <https://eb.dy.fi/author/eb/page/5/>. Citado na p. 18.

FORSYDE. **The SystemC implementation of ForSyDe**. ForSyDe. Disponível em: <https://forsyde.github.io/ForSyDe-SystemC/ct-tutorial>. Citado na p. 21.

GODOT. **List of features**. Godot, 2023. Disponível em: <https://docs.godotengine.org/en/3.5/about/list_of_features.html>. Citado na p. 55.

IVO, R. M.; MUÑOZ, D. M. RTRLib: A High-Level Modeling Tool for the Implementation of Dynamically Partial Reconfigurable System-on-Chips. In: 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig). 2019. P. 1–5. DOI: 10.1109/ReConFig48160.2019.8994779. Citado na p. 21.

JESUS, T. R. de. **Arquiteturas de hardware dedicadas de controladores nebulosos para auxílio à locomoção de deficientes visuais**. 2017. Dissertação (Mestrado em Sistemas Mecatrônicos) – Universidade de Brasília, Brasília. Citado na p. 22.

LOPEZ, D.; JIMENEZ, C.; BATURONE, I.; BARRIGA, A.; SANCHEZ-SOLANO, S. Xfuzzy: a design environment for fuzzy systems. In: 1998 IEEE International Conference on Fuzzy Systems Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36228). 1998. v. 2, 1060–1065 vol.2. DOI: 10.1109/FUZZY.1998.686265. Citado na p. 22.

MARTIN, D. **US chip industry has another shortage: Electronics engineers**. The Register, jul. 2022. Disponível em: <https://www.theregister.com/2022/07/08/semiconductor_engineer_shortage/>. Citado na p. 13.

MENDEZ, S. P.; GHERARDINI, M.; SANTOS, G. V. d. P.; MUÑOZ, D. M.; AYALA, H. V. H.; CIPRIANI, C. Data-Driven Real-Time Magnetic Tracking Applied to Myokinetic Interfaces. **IEEE Transactions on Biomedical Circuits and Systems**, v. 16, n. 2, p. 266–274, 2022. DOI: 10.1109/TBCAS.2022.3161133. Citado na p. 22.

MUÑOZ, D. M.; SÁNCHEZ, D.; LLANOS, C.; AYALA-RINCÓN, M. Tradeoff of FPGA Design of a Floating-point Library for Arithmetic Operators. **International Journal of Integrated Circuits and Systems**, v. 5, n. 1, 2010. Citado na p. 63.

MUÑOZ, D.; LLANOS, C.; COELHO, L.; AYALA-RINCÓN, M. Hardware opposition-based PSO applied to mobile robot controllers. **Engineering Applications of Artificial Intelligence**, v. 28, 2014. Citado na p. 59.

MUÑOZ, D. M.; LLANOS, C. H.; COELHO, L. d. S.; AYALA-RINCÓN, M. Comparison between two FPGA implementations of the Particle Swarm Optimization algorithm for high-performance embedded applications. In: 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA). 2010. P. 1637–1645. DOI: 10.1109/BICTA.2010.5645256. Citado na p. 22.

PROST-BOUCLE, A.; MULLER, O.; ROUSSEAU, F. A Fast and Autonomous HLS Methodology for Hardware Accelerator Generation under Resource Constraints. In: 2013 Euromicro Conference on Digital System Design. 2013. P. 201–208. DOI: 10.1109/DSD.2013.30. Citado na p. 19.

VAHID, F. **Digital Design**. J. Wiley & Sons, 2007. Citado nas pp. 15–17, 59.

VASILACHE, N.; ZINENKO, O.; BIK, A. J. C.; RAVISHANKAR, M.; RAOUX, T.; BELYAEV, A.; SPRINGER, M.; GYSI, T.; CABALLERO, D.; HERHUT, S.; LAURENZO, S.; COHEN, A. Structured Operations: Modular Design of a Code Generators for Tensor Compilers. In: MENDIS, C.; RAUCHWERGER, L. (Ed.). **Languages and Compilers for Parallel Computing**. Cham: Springer Nature Switzerland, 2023. P. 141–156. ISBN 978-3-031-31445-2. Citado na p. 13.

WAKERLY, J. F. **Digital Design: Principles and practices**. 4. ed.: Pearson Prentice Hall, 2006. Citado nas pp. 15, 16.

WILSON, A. **Active-HDL**. FirstEDA. Disponível em: <https://firsteda.com/products/aldec/aldec-active-hdl/>. Citado na p. 21.

XILINX. **Vitis High-Level Synthesis User Guide (UG1399)**. Mai. 2023. Disponível em: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Benefits-of-High-Level-Synthesis>. Citado nas pp. 19–21.

ZHANG, X.; WANG, J.; ZHU, C.; LIN, Y.; XIONG, J.; HWU, W.-m.; CHEN, D. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2018. P. 1–8. DOI: 10.1145/3240765.3240801. Citado na p. 22.

Anexos

Anexo A – Códigos Gerados pela Aplicação HiLDA e de Validação em Hardware

A.1 Exemplo de FSM simples

Código A.1 – Código gerado a partir do exemplo da figura 3.21.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity exemplo is
5          Port ( clk : in STD_LOGIC;
6                 rst : in STD_LOGIC;
7                 button: in STD_LOGIC;
8                 fim: out STD_LOGIC;
9                 s1: out STD_LOGIC;
10                s2: out STD_LOGIC);
11 end exemplo;
12
13 architecture Behavioral of exemplo is
14
15 type states is (A,B,C);
16 signal acts,nxts:states:=A;
17
18 begin
19 stateReg:process(clk,rst)
20 begin
21     if rst = '1' then
22         acts <= A;
23     elsif rising_edge(clk) then
24         acts <= nxts;
25     end if;
26 end process;
27
28 combLogic:process(acts,button)
29 begin
30
31 case acts is
32    when A=>
33        if button='1' then
34           nxts<= B;
35        else
36           nxts<=A;
37        end if;
```

```
38      when B=>
39          if button='1' then
40              nxts<= C;
41          else
42              nxts<=A;
43          end if;
44      when C=>
45          if button='0' then
46              nxts<= A;
47          else
48              nxts<=C;
49          end if;
50      when others=>
51          nxts<= A;
52  end case;
53  end process;
54
55  outputLogic:process(acts,button)
56  begin
57
58  case acts is
59      when A=>
60          fim<='0';
61
62          if button='1' then
63                s1<='0';
64                s2<='1';
65          else
66                s1<='0';
67                s2<='0';
68          end if;
69      when B=>
70          fim<='0';
71
72          if button='1' then
73                s1<='1';
74                s2<='0';
75          else
76                s1<='0';
77                s2<='1';
78          end if;
79      when C=>
80          fim<='1';
81
82          if button='0' then
83                s1<='0';
84                s2<='1';
85          else
86                s1<='1';
87                s2<='0';
88          end if;
89      when others=>
```

```
90          fim<='0';
91          s1<='0';
92          s2<='0';
93  end case;
94  end process;
95  end Behavioral;
```

A.2 Casos suportados para conversão automática de tipos de sinais

Código A.2 – Código com casos suportados para conversão de tipo de sinal.

```
1  def convert_signal(type0, type1, val):
2      type0 = type0.upper().split("(")[0]
3      type1 = type1.upper().split("(")[0]
4
5      vals = (type0, type1)
6
7      if type0.upper() == type1.upper():
8          return val
9      else:
10         print("Signal type does not match in", val)
11     match vals:
12         case ("STD_LOGIC_VECTOR", "UNSIGNED"):
13             print(val, "-> has been transformed from ", type0,
                   "to", type1)
14             return f"unsigned({val})"
15
16         case ("STD_LOGIC_VECTOR", "SIGNED"):
17             print(val, "-> has been transformed from ", type0,
                   "to", type1)
18             return f"signed({val})"
19
20         case ("STD_LOGIC_VECTOR", "INTEGER"):
21             print(val, "-> has been transformed from ", type0,
                   "to", type1)
22             return f"to_integer(unsigned({val}))"
23
24         case ("UNSIGNED", "STD_LOGIC_VECTOR"):
25             print(val, "-> has been transformed from ", type0,
                   "to", type1)
26             return f"std_logic_vector({val})"
27
28         case ("UNSIGNED", "SIGNED"):
29             print(val, "-> has been transformed from ", type0,
                   "to", type1)
30             return f"signed({val})"
31
32         case ("UNSIGNED", "INTEGER"):
```

```
33              print(val, "-> has been transformed from ", type0,
                    "to", type1)
34              return f"to_integer({val})"
35
36          case ("SIGNED", "STD_LOGIC_VECTOR"):
37              print(val, "-> has been transformed from ", type0,
                    "to", type1)
38              return f"std_logic_vector({val})"
39
40          case ("SIGNED", "INTEGER"):
41              print(val, "-> has been transformed from ", type0,
                    "to", type1)
42              return f"to_integer({val})"
43
44          case ("SIGNED", "UNSIGNED"):
45              print(val, "-> has been transformed from ", type0,
                    "to", type1)
46              return f"unsigned({val})"
47
48          case ("INTEGER", "UNSIGNED"):
49              print(val, "-> has been transformed from ", type0,
                    "to", type1)
50              return f"to_unsigned({val},1)"
51
52          case ("INTEGER", "SIGNED"):
53              print(val, "-> has been transformed from ", type0,
                    "to", type1)
54              return f"to_signed({val},1)"
55
56          case ("INTEGER", "STD_LOGIC_VECTOR"):
57              print(val, "-> has been transformed from ", type0,
                    "to", type1)
58              return f"std_logic_vector(to_unsigned({val},1))"
59          case _:
60              print(f"{val}--->conversion not supported " + type0 +
                    " and " + type1)
61              exit()
```

A.3 Código do *multiply-accumulate*

Código A.3 – Código gerado a partir do exemplo da figura 3.29.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  USE WORK.FPUPACK.ALL;
6
7  entity exRTL is
8
```

```
9     port(
10      clk: in std_logic;
11      rst: in std_logic;
12      x1: in std_logic_vector(26 downto 0);
13      x2: in std_logic_vector(26 downto 0);
14      beginCalc: in std_logic;
15      result: out std_logic_vector(26 downto 0);
16      accReady: out std_logic   );
17  end exRTL;
18
19  architecture Behavioral of exRTL is
20
21  component  addsubfsm_v6 is
22    port (reset      :  in std_logic;
23       clk         :  in std_logic;
24       op      :  in std_logic;
25       op_a      :  in std_logic_vector(FP_WIDTH-1 downto 0);
26       op_b      :  in std_logic_vector(FP_WIDTH-1 downto 0);
27       start_i     :  in std_logic;
28       addsub_out : out std_logic_vector(FP_WIDTH-1 downto 0);
29       ready_as    : out std_logic);
30  end component;
31
32  component  multiplierfsm_v2 is
33      port (reset     :  in std_logic;
34            clk      :  in std_logic;
35         op_a     :  in std_logic_vector(FP_WIDTH-1 downto 0);
36            op_b      :  in std_logic_vector(FP_WIDTH-1 downto 0);
37        start_i   :  in std_logic;
38            mul_out   : out std_logic_vector(FP_WIDTH-1 downto 0);
39          ready_mul: out std_logic);
40  end component;
41
42  signal saddsub_out_to_out6970 :STD_LOGIC_VECTOR( FP_WIDTH-1 DOWNTO
      0 );
43  signal saddsub_out_to_out6970 :STD_LOGIC;
44  signal smul_out_to_op_b5684 :STD_LOGIC_VECTOR( FP_WIDTH-1 DOWNTO 0
      );
45  signal sready_mul_to_start_i9575 :STD_LOGIC;
46  constant sconst_to_op19318 : std_logic:='0';
47
48  begin
49
50  add1: addsubfsm_v6
51    port map(
52       RESET=>rst,
53       CLK=>clk,
54       OP=>sconst_to_op19318,
55       OP_A=>saddsub_out_to_out6970,
56       OP_B=>smul_out_to_op_b5684,
57       START_I=>sready_mul_to_start_i9575,
58       ADDSUB_OUT=>saddsub_out_to_out6970,
```

```
59      READY_AS=>saddsub_out_to_out6970);
60
61 mult1: multiplierfsm_v2
62   port map(
63      RESET=>rst,
64      CLK=>clk,
65      OP_A=>x1,
66      OP_B=>x2,
67      START_I=>beginCalc,
68      MUL_OUT=>smul_out_to_op_b5684,
69      READY_MUL=>sready_mul_to_start_i9575);
70
71
72
73
74
75 result<=saddsub_out_to_out6970;
76 accReady<=saddsub_out_to_out6970;
77
78
79 end behavioral;
```

A.4 Códigos utilizados na implementação do controlador de marca-passo

Código A.4 – Código do bloco de controle de um marca-passo gerado pelo vFSMgen.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity marcapasso is
5          Port ( clk : in STD_LOGIC;
6                 rst : in STD_LOGIC;
7                 sa: in STD_LOGIC;
8                 za: in STD_LOGIC;
9                 sv: in STD_LOGIC;
10                zv: in STD_LOGIC;
11                pa: out STD_LOGIC;
12                pv: out STD_LOGIC;
13                ta: out STD_LOGIC;
14                tv: out STD_LOGIC);
15 end marcapasso;
16
17 architecture Behavioral of marcapasso is
18
19 type states is
     (ResetTemporizadorA,EsperaA,ContracaoA,ResetTemporizadorV,EsperaV,Contrac
20 signal acts,nxts:states:=ResetTemporizadorA;
21
```

```
22 begin
23 stateReg:process(clk,rst)
24 begin
25     if rst = '1' then
26         acts <= ResetTemporizadorA;
27     elsif rising_edge(clk) then
28         acts <= nxts;
29     end if;
30 end process;
31
32 combLogic:process(acts,sa,sv,za,zv)
33 begin
34
35 case acts is
36    when ResetTemporizadorA=>
37        nxts<=EsperaA;
38    when EsperaA=>
39        if sa='0' and za='1' then
40           nxts<= ContracaoA;
41        elsif sa='1' then
42           nxts<= ResetTemporizadorV;
43        else
44           nxts<=EsperaA;
45        end if;
46    when ContracaoA=>
47        nxts<=ResetTemporizadorV;
48    when ResetTemporizadorV=>
49        nxts<=EsperaV;
50    when EsperaV=>
51        if sv='0' and zv='1' then
52           nxts<= ContracaoV;
53        elsif sv='1' then
54           nxts<= ResetTemporizadorA;
55        else
56           nxts<=EsperaV;
57        end if;
58    when ContracaoV=>
59        nxts<=ResetTemporizadorA;
60    when others=>
61        nxts<= ResetTemporizadorA;
62 end case;
63 end process;
64
65 outputLogic:process(acts,sa,sv,za,zv)
66 begin
67
68 case acts is
69    when ResetTemporizadorA=>
70        pa<='0';
71        ta<='1';
72        pv<='0';
73        tv<='0';
```

```
74      when EsperaA=>
75          pa<='0';
76          ta<='0';
77          pv<='0';
78          tv<='0';
79      when ContracaoA=>
80          pa<='1';
81          ta<='0';
82          pv<='0';
83          tv<='0';
84      when ResetTemporizadorV=>
85          pa<='0';
86          ta<='0';
87          pv<='0';
88          tv<='1';
89      when EsperaV=>
90          pa<='0';
91          ta<='0';
92          pv<='0';
93          tv<='0';
94      when ContracaoV=>
95          pa<='0';
96          ta<='0';
97          pv<='1';
98          tv<='0';
99      when others=>
100         pa<='0';
101         pv<='0';
102         ta<='0';
103         tv<='0';
104 end case;
105 end process;
106 end Behavioral;
```

Código A.5 – Código do *testbench* do controlador marca-passo.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity marcapasso_tb is
6  end marcapasso_tb;
7
8  architecture behav of marcapasso_tb is
9
10 signal sclk : std_logic;
11 signal srst : STD_LOGIC;
12 signal spa : STD_LOGIC;
13 signal sta : STD_LOGIC;
14 signal spv : STD_LOGIC;
15 signal stv : STD_LOGIC;
16 signal ssa : STD_LOGIC;
```

```
17 signal sza : STD_LOGIC;
18 signal ssv : STD_LOGIC;
19 signal szv : STD_LOGIC;
20
21 COMPONENT MARCAPASSO IS
22         Port ( clk : in STD_LOGIC;
23                rst : in STD_LOGIC;
24                sa: in STD_LOGIC;
25                za: in STD_LOGIC;
26                sv: in STD_LOGIC;
27                zv: in STD_LOGIC;
28                pa: out STD_LOGIC;
29                pv: out STD_LOGIC;
30                ta: out STD_LOGIC;
31                tv: out STD_LOGIC);
32 END COMPONENT;
33
34
35 begin
36 sclk<= '0', not sclk after 5.0 ns;
37
38 srst<= '0', '1' after 25 ns, '0' after 50 ns;
39
40 ssa<= '0', '1' after 60 ns, '0' after 80 ns;
41
42 sza<= '0', '1' after 80 ns;
43
44 ssv<= '0', '1' after 100 ns, '0' after 120 ns;
45
46 szv<= '0', '1' after 120 ns;
47
48 uut: marcapasso port map(
49     CLK=>sclk,
50     RST=>srst,
51     SA=>ssa,
52     ZA=>sza,
53     SV=>ssv,
54     ZV=>szv,
55     PA=>spa,
56     PV=>spv,
57     TA=>sta,
58     TV=>stv);
59
60 end behav;
```

A.5 Códigos utilizados na implementação do *Perceptron*

Código A.6 – Código da FSM de controle do perceptron.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity neur_ctrl is
        Port ( clk : in STD_LOGIC;
               rst : in STD_LOGIC;
               start_neur: in STD_LOGIC;
               start_i: in STD_LOGIC;
               add2ready: in STD_LOGIC;
               acc: out STD_LOGIC;
               clear_regs: out STD_LOGIC;
               opready: out STD_LOGIC;
               sel: out STD_LOGIC;
               selbias: out STD_LOGIC;
               start_mult: out STD_LOGIC);
end neur_ctrl;

architecture Behavioral of neur_ctrl is

type states is
   (IDLE,Estart_mult,wait_add1,accumulate,wait_add2,accumulate2,choose_bias,
signal acts,nxts:states:=IDLE;

begin
stateReg:process(clk,rst)
begin
    if rst = '1' then
        acts <= IDLE;
    elsif rising_edge(clk) then
        acts <= nxts;
    end if;
end process;

combLogic:process(acts,add2ready,start_i,start_neur)
begin

case acts is
   when IDLE=>
       if start_neur='1' then
          nxts<= Estart_mult;
       else
          nxts<=IDLE;
       end if;
   when Estart_mult=>
       nxts<=wait_add1;
   when wait_add1=>
       if start_i='1' then
          nxts<= accumulate;
       else
          nxts<=wait_add1;
```

```
50          end if;
51      when accumulate=>
52          nxts<=wait_add2;
53      when wait_add2=>
54          if start_i='1' then
55              nxts<= accumulate2;
56          else
57              nxts<=wait_add2;
58          end if;
59      when accumulate2=>
60          nxts<=choose_bias;
61      when choose_bias=>
62          if add2ready='1' then
63              nxts<= add_bias;
64          else
65              nxts<=choose_bias;
66          end if;
67      when add_bias=>
68          nxts<=wait_op;
69      when wait_op=>
70          if add2ready='1' then
71              nxts<= IDLE;
72          else
73              nxts<=wait_op;
74          end if;
75      when others=>
76          nxts<= IDLE;
77  end case;
78  end process;
79
80  outputLogic:process(acts,add2ready,start_i,start_neur)
81  begin
82
83  case acts is
84      when IDLE=>
85          acc<='0';
86          sel<='0';
87          selbias<='0';
88          start_mult<='0';
89          opready<='1';
90          clear_regs<='0';
91      when Estart_mult=>
92          acc<='0';
93          sel<='0';
94          selbias<='0';
95          start_mult<='1';
96          opready<='0';
97          clear_regs<='1';
98      when wait_add1=>
99          acc<='0';
100         sel<='1';
101         selbias<='0';
```

```
102         start_mult<='0';
103         opready<='0';
104         clear_regs<='0';
105     when accumulate=>
106         acc<='1';
107         sel<='1';
108         selbias<='0';
109         start_mult<='1';
110         opready<='0';
111         clear_regs<='0';
112     when wait_add2=>
113         acc<='0';
114         sel<='1';
115         selbias<='0';
116         start_mult<='0';
117         opready<='0';
118         clear_regs<='0';
119     when accumulate2=>
120         acc<='1';
121         sel<='1';
122         selbias<='0';
123         start_mult<='0';
124         opready<='0';
125         clear_regs<='0';
126     when choose_bias=>
127         acc<='0';
128         sel<='1';
129         selbias<='1';
130         start_mult<='0';
131         opready<='0';
132         clear_regs<='0';
133     when add_bias=>
134         acc<='1';
135         sel<='1';
136         selbias<='1';
137         start_mult<='0';
138         opready<='0';
139         clear_regs<='0';
140     when wait_op=>
141         acc<='0';
142         sel<='0';
143         selbias<='0';
144         start_mult<='0';
145         opready<='0';
146         clear_regs<='0';
147     when others=>
148         acc<='0';
149         clear_regs<='0';
150         opready<='0';
151         sel<='0';
152         selbias<='0';
153         start_mult<='0';
```

```
154 end case;
155 end process;
156 end Behavioral;
```

Código A.7 – Código do perceptron.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  USE WORK.FPUPACK.ALL;
6
7  entity neuron is
8
9     port(
10      x1: in std_logic_vector(26 downto 0);
11      x2: in std_logic_vector(26 downto 0);
12      x3: in std_logic_vector(26 downto 0);
13      x4: in std_logic_vector(26 downto 0);
14      w1: in std_logic_vector(26 downto 0);
15      w2: in std_logic_vector(26 downto 0);
16      w3: in std_logic_vector(26 downto 0);
17      w4: in std_logic_vector(26 downto 0);
18      clk: in std_logic;
19      rst: in std_logic;
20      start_neur: in std_logic;
21      ready: out std_logic;
22      result: out std_logic_vector(31 downto 0)   );
23 end neuron;
24
25 architecture Behavioral of neuron is
26
27 constant CBIAS : std_logic_vector(26 downto
      0):="00111111100000000000000000";
28 component  mux2_27b is
29     Port ( a : in STD_LOGIC_VECTOR (26 downto 0);
30            b : in STD_LOGIC_VECTOR (26 downto 0);
31            selmux : in STD_LOGIC;
32            c : out STD_LOGIC_VECTOR (26 downto 0));
33 end component;
34
35 component  multiplierfsm_v2 is
36     port (reset    :  in std_logic;
37          clk      :  in std_logic;
38        op_a     :  in std_logic_vector(FP_WIDTH-1 downto 0);
39           op_b      :  in std_logic_vector(FP_WIDTH-1 downto 0);
40       start_i   :  in std_logic;
41           mul_out   : out std_logic_vector(FP_WIDTH-1 downto 0);
42         ready_mul: out std_logic);
43 end component;
44
45 component  addsubfsm_v6 is
```

```
46    port (reset     :  in std_logic;
47       clk         :  in std_logic;
48       op      :  in std_logic;
49       op_a      :  in std_logic_vector(FP_WIDTH-1 downto 0);
50       op_b      :  in std_logic_vector(FP_WIDTH-1 downto 0);
51       start_i     :  in std_logic;
52       addsub_out : out std_logic_vector(FP_WIDTH-1 downto 0);
53       ready_as    : out std_logic);
54  end component;
55
56  component  neur_ctrl is
57          Port ( clk : in STD_LOGIC;
58                 rst : in STD_LOGIC;
59                 start_neur: in STD_LOGIC;
60                 start_i: in STD_LOGIC;
61                 add2ready: in STD_LOGIC;
62                 acc: out STD_LOGIC;
63                 clear_regs: out STD_LOGIC;
64                 opready: out STD_LOGIC;
65                 sel: out STD_LOGIC;
66                 selbias: out STD_LOGIC;
67                 start_mult: out STD_LOGIC);
68  end component;
69
70  signal sc_to_op_b10000 :STD_LOGIC_VECTOR( 26 DOWNTO 0 );
71  signal sc_to_op_b68810 :STD_LOGIC_VECTOR( 26 DOWNTO 0 );
72  signal ssel_to_selmux37912 :STD_LOGIC;
73  signal sstart_mult_to_start_i46645 :STD_LOGIC;
74  signal sselbias_to_sel81370 :STD_LOGIC;
75  signal sacc_to_start_i71971 :STD_LOGIC;
76  signal sopready_to_out80672 :STD_LOGIC;
77  signal sclear_regs_to_reset41673 :STD_LOGIC;
78  signal sc_to_op_a48716 :STD_LOGIC_VECTOR( 26 DOWNTO 0 );
79  signal sc_to_op_a46918 :STD_LOGIC_VECTOR( 26 DOWNTO 0 );
80  signal sready_mul_to_a19428 :STD_LOGIC;
81  signal smul_out_to_op_a27229 :STD_LOGIC_VECTOR( FP_WIDTH-1 DOWNTO
       0 );
82  signal smul_out_to_op_b41131 :STD_LOGIC_VECTOR( FP_WIDTH-1 DOWNTO
       0 );
83  signal sready_mul_to_b15032 :STD_LOGIC;
84  signal sready_as_to_start_i99047 :STD_LOGIC;
85  signal saddsub_out_to_sum28848 :STD_LOGIC_VECTOR( FP_WIDTH-1
       DOWNTO 0 );
86  signal sready_as_to_add2ready88650 :STD_LOGIC;
87  signal saddsub_out_to_x18551 :STD_LOGIC_VECTOR( FP_WIDTH-1 DOWNTO
       0 );
88  signal sc_to_start_i19253 :std_logic;
89  constant sconst_to_op18055 : std_logic:='0';
90  constant sconst_to_op39357 : std_logic:='0';
91  signal sy_to_out17562 :std_logic_vector(31 downto 0);
92  signal sselout_to_op_b72864 :std_logic_vector(26 downto 0);
93  constant sconst_to_bias32066 : std_logic_vector(26 downto
```

```
    0):=CBIAS;
94
95  begin
96
97  xmux1: mux2_27b
98    port map(
99       A=>x1,
100      B=>x2,
101      SELMUX=>ssel_to_selmux37912,
102      C=>sc_to_op_b10000);
103
104 xmux2: mux2_27b
105   port map(
106      A=>x3,
107      B=>x4,
108      SELMUX=>ssel_to_selmux37912,
109      C=>sc_to_op_b68810);
110
111 wmux1: mux2_27b
112   port map(
113      A=>w1,
114      B=>w2,
115      SELMUX=>ssel_to_selmux37912,
116      C=>sc_to_op_a48716);
117
118 wmux2: mux2_27b
119   port map(
120      A=>w3,
121      B=>w4,
122      SELMUX=>ssel_to_selmux37912,
123      C=>sc_to_op_a46918);
124
125 mult1: multiplierfsm_v2
126   port map(
127      RESET=>rst,
128      CLK=>clk,
129      OP_A=>sc_to_op_a48716,
130      OP_B=>sc_to_op_b10000,
131      START_I=>sstart_mult_to_start_i46645,
132      MUL_OUT=>smul_out_to_op_a27229,
133      READY_MUL=>sready_mul_to_a19428);
134
135 mult2: multiplierfsm_v2
136   port map(
137      RESET=>rst,
138      CLK=>clk,
139      OP_A=>sc_to_op_a46918,
140      OP_B=>sc_to_op_b68810,
141      START_I=>sstart_mult_to_start_i46645,
142      MUL_OUT=>smul_out_to_op_b41131,
143      READY_MUL=>sready_mul_to_b15032);
144
```

```
145 add1: addsubfsm_v6
146   port map(
147       RESET=>rst,
148       CLK=>clk,
149       OP=>sconst_to_op18055,
150       OP_A=>smul_out_to_op_a27229,
151       OP_B=>smul_out_to_op_b41131,
152       START_I=>sc_to_start_i19253,
153       ADDSUB_OUT=>saddsub_out_to_sum28848,
154       READY_AS=>sready_as_to_start_i99047);
155
156 add2: addsubfsm_v6
157   port map(
158       RESET=>sclear_regs_to_reset41673,
159       CLK=>clk,
160       OP=>sconst_to_op39357,
161       OP_A=>saddsub_out_to_x18551,
162       OP_B=>sselout_to_op_b72864,
163       START_I=>sacc_to_start_i71971,
164       ADDSUB_OUT=>saddsub_out_to_x18551,
165       READY_AS=>sready_as_to_add2ready88650);
166
167 fsm_neur_ctrl: neur_ctrl
168   port map(
169       CLK=>clk,
170       RST=>rst,
171       START_NEUR=>start_neur,
172       START_I=>sready_as_to_start_i99047,
173       ADD2READY=>sready_as_to_add2ready88650,
174       ACC=>sacc_to_start_i71971,
175       CLEAR_REGS=>sclear_regs_to_reset41673,
176       OPREADY=>sopready_to_out80672,
177       SEL=>ssel_to_selmux37912,
178       SELBIAS=>sselbias_to_sel81370,
179       START_MULT=>sstart_mult_to_start_i46645);
180
181
182
183
184
185 sc_to_start_i19253<=sready_mul_to_a19428 and sready_mul_to_b15032;
186
187
188
189 sy_to_out17562<=saddsub_out_to_x18551&"00000";
190
191 with sselbias_to_sel81370 select sselout_to_op_b72864<=
192     saddsub_out_to_sum28848 when '0',
193     sconst_to_bias32066 when others;
194
195 result<=sy_to_out17562;
196 ready<=sopready_to_out80672;
```

```
197
198
199 end behavioral;
```

Código A.8 – Código do *testebench* do perceptron.

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5
6  entity neuron_tb is
7  end neuron_tb;
8
9  architecture behav of neuron_tb is
10
11 signal sx : std_logic_vector(26 downto 0);
12 signal sw : std_logic_vector(26 downto 0);
13 signal sclk : std_logic;
14 signal srst : std_logic;
15 signal sstart : std_logic;
16 signal sready : std_logic;
17 signal sresult : std_logic_vector(31 downto 0);
18
19 COMPONENT NEURON IS
20
21    port(
22      x1: in std_logic_vector(26 downto 0);
23      x2: in std_logic_vector(26 downto 0);
24      x3: in std_logic_vector(26 downto 0);
25      x4: in std_logic_vector(26 downto 0);
26      w1: in std_logic_vector(26 downto 0);
27      w2: in std_logic_vector(26 downto 0);
28      w3: in std_logic_vector(26 downto 0);
29      w4: in std_logic_vector(26 downto 0);
30      clk: in std_logic;
31      rst: in std_logic;
32      start_neur: in std_logic;
33      ready: out std_logic;
34      result: out std_logic_vector(31 downto 0)   );
35 END COMPONENT;
36
37
38 begin
39 sx<= "01000000000000000000000000000";
40
41 sw<= "01000000010000000000000000000";
42
43 sclk<= '0', not sclk after 5.0 ns;
44
45 srst<= '1', '0' after 25 ns;
46
```

```
47 sstart <= '0', '1' after 30 ns, '0' after 40 ns, '1' after 130 ns,
       '0' after 140 ns;
48
49 uut: neuron port map(
50      X1=>sx,
51      X2=>sx,
52      X3=>sx,
53      X4=>sx,
54      W1=>sw,
55      W2=>sw,
56      W3=>sw,
57      W4=>sw,
58      CLK=>sclk,
59      RST=>srst,
60      START_NEUR=>sstart,
61      READY=>sready,
62      RESULT=>sresult);
63
64 end behav;
```

Código A.9 – Código do *topmodule* do *Perceptron*.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  USE WORK.FPUPACK.ALL;
6
7  entity topmodule is
8
9     port(
10      start: in std_logic;
11      rst: in std_logic;
12      clk: in std_logic;
13      selectx: in std_logic;
14      select_out: in std_logic;
15      result: out std_logic_vector(15 downto 0)   );
16 end topmodule;
17
18 architecture Behavioral of topmodule is
19
20 constant W1 : std_logic_vector(26 downto
       0):="00111111100000000000000000";
21 constant W2 : std_logic_vector(26 downto
       0):="01000000000000000000000000";
22 constant W3 : std_logic_vector(26 downto
       0):="01000000010000000000000000";
23 constant W4 : std_logic_vector(26 downto
       0):="01000000100000000000000000";
24 component  neuron is
25
26    port(
```

```
27      x1: in std_logic_vector(26 downto 0);
28      x2: in std_logic_vector(26 downto 0);
29      x3: in std_logic_vector(26 downto 0);
30      x4: in std_logic_vector(26 downto 0);
31      w1: in std_logic_vector(26 downto 0);
32      w2: in std_logic_vector(26 downto 0);
33      w3: in std_logic_vector(26 downto 0);
34      w4: in std_logic_vector(26 downto 0);
35      clk: in std_logic;
36      rst: in std_logic;
37      start_neur: in std_logic;
38      ready: out std_logic;
39      result: out std_logic_vector(31 downto 0)   );
40  end component;
41
42  component  choose_out is
43      Port ( input : in STD_LOGIC_VECTOR (31 downto 0);
44             readyin : in STD_LOGIC;
45             selectOut : in STD_LOGIC;
46             word1: out STD_LOGIC_VECTOR (15 downto 0));
47  end component;
48
49  signal sword1_to_out8860 :STD_LOGIC_VECTOR( 15 DOWNTO 0 );
50  signal sRESULT_to_input3951 :STD_LOGIC_VECTOR( 31 DOWNTO 0 );
51  signal sREADY_to_readyin31230 :STD_LOGIC;
52  constant sconst_to_W13388 : std_logic_vector(26 downto 0):=W1;
53  constant sconst_to_W254110 : std_logic_vector(26 downto 0):=W2;
54  constant sconst_to_W342712 : std_logic_vector(26 downto 0):=W3;
55  constant sconst_to_W498614 : std_logic_vector(26 downto 0):=W4;
56  signal ssx1_to_X118525 :std_logic_vector(26 downto 0);
57  signal ssx2_to_X220426 :std_logic_vector(26 downto 0);
58  signal ssx3_to_X382727 :std_logic_vector(26 downto 0);
59  signal ssx4_to_X425028 :std_logic_vector(26 downto 0);
60
61  begin
62
63  neur1: neuron
64    port map(
65      X1=>ssx1_to_X118525,
66      X2=>ssx2_to_X220426,
67      X3=>ssx3_to_X382727,
68      X4=>ssx4_to_X425028,
69      W1=>sconst_to_W13388,
70      W2=>sconst_to_W254110,
71      W3=>sconst_to_W342712,
72      W4=>sconst_to_W498614,
73      CLK=>clk,
74      RST=>rst,
75      START_NEUR=>start,
76      READY=>sREADY_to_readyin31230,
77      RESULT=>sRESULT_to_input3951);
78
```

```
79  ch1: choose_out
80    port map(
81       INPUT=>sRESULT_to_input3951,
82       READYIN=>sREADY_to_readyin31230,
83       SELECTOUT=>select_out,
84       WORD1=>sword1_to_out8860);
85
86
87
88
89
90  randomIn:process(selectx)
91  begin
92      case selectx is
93          when '0' =>
94              ssx1_to_X118525<="01000000000100110010010011";
95              ssx2_to_X220426<="01000000100110000011010110";
96              ssx3_to_X382727<="10111111001101000100110000";
97              ssx4_to_X425028<="00111110110111001000010111";
98          when '1' =>
99              ssx1_to_X118525<="11000000100101000000100110";
100             ssx2_to_X220426<="10111111011011101100000001";
101             ssx3_to_X382727<="01000000010000000000001100";
102             ssx4_to_X425028<="01000000100010111011000111";
103          when others=>
104             ssx1_to_X118525<="11000000100101000000100110";
105             ssx2_to_X220426<="10111111011011101100000001";
106             ssx3_to_X382727<="01000000010000000000001100";
107             ssx4_to_X425028<="01000000100010111011000111";
108     end case;
109 end process;
110
111
112
113 result<=sword1_to_out8860;
114
115
116 end behavioral;
```

Código A.10 – Código do *testbench* do *topmodule* do *Perceptron*.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity topmodule_tb is
6  end topmodule_tb;
7
8  architecture behav of topmodule_tb is
9
10 signal sresult : std_logic_vector(15 downto 0);
11 signal srst : std_logic;
```

```
12 signal sclk : std_logic;
13 signal sselectx : std_logic;
14 signal sstart : std_logic;
15 signal sselect_out : std_logic;
16
17 COMPONENT TOPMODULE IS
18
19    port(
20      start: in std_logic;
21      rst: in std_logic;
22      clk: in std_logic;
23      selectx: in std_logic;
24      select_out: in std_logic;
25      result: out std_logic_vector(15 downto 0)   );
26 END COMPONENT;
27
28
29 begin
30 srst<= '1', '0' after 25 ns;
31
32 sclk<= '0', not sclk after 5.0 ns;
33
34 sselectx<= '0', '1' after 200 ns;
35
36 sstart<= '0', '1' after 30 ns, '0' after 40 ns, '1' after 210 ns,
      '0' after 220 ns;
37
38 sselect_out<= '0', '1' after 140 ns, '0' after 170 ns, '1' after
      350 ns;
39
40 uut: topmodule port map(
41     START=>sstart,
42     RST=>srst,
43     CLK=>sclk,
44     SELECTX=>sselectx,
45     SELECT_OUT=>sselect_out,
46     RESULT=>sresult);
47
48 end behav;
```

A.6 Códigos utilizados na implementação das interfaces AXI

Código A.11 – Código mais extrerno do AXI4-Lite encapsulando o perceptron.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
```

```
5  entity top27baxilite_AXI is
6    generic (
7      -- Users to add parameters here
8
9      -- User parameters ends
10     -- Do not modify the parameters beyond this line
11
12
13     -- Parameters of Axi Slave Bus Interface S00_AXI
14     C_S00_AXI_DATA_WIDTH  : integer := 32;
15     C_S00_AXI_ADDR_WIDTH  : integer := 9
16   );
17   port (
18     -- Users to add ports here
19
20     -- User ports ends
21     -- Do not modify the ports beyond this line
22
23
24     -- Ports of Axi Slave Bus Interface S00_AXI
25     s00_axi_aclk  : in std_logic;
26     s00_axi_aresetn : in std_logic;
27     s00_axi_awaddr  : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1
          downto 0);
28     s00_axi_awprot  : in std_logic_vector(2 downto 0);
29     s00_axi_awvalid : in std_logic;
30     s00_axi_awready : out std_logic;
31     s00_axi_wdata : in std_logic_vector(C_S00_AXI_DATA_WIDTH-1
          downto 0);
32     s00_axi_wstrb : in std_logic_vector((C_S00_AXI_DATA_WIDTH/8)-1
          downto 0);
33     s00_axi_wvalid  : in std_logic;
34     s00_axi_wready  : out std_logic;
35     s00_axi_bresp : out std_logic_vector(1 downto 0);
36     s00_axi_bvalid  : out std_logic;
37     s00_axi_bready  : in std_logic;
38     s00_axi_araddr  : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1
          downto 0);
39     s00_axi_arprot  : in std_logic_vector(2 downto 0);
40     s00_axi_arvalid : in std_logic;
41     s00_axi_arready : out std_logic;
42     s00_axi_rdata : out std_logic_vector(C_S00_AXI_DATA_WIDTH-1
          downto 0);
43     s00_axi_rresp : out std_logic_vector(1 downto 0);
44     s00_axi_rvalid  : out std_logic;
45     s00_axi_rready  : in std_logic
46   );
47 end top27baxilite_AXI;
48
49 architecture arch_imp of top27baxilite_AXI is
50
51   -- component declaration
```

```
52
53   component top27baxilite is
54     generic (
55     C_S_AXI_DATA_WIDTH  : integer := 32;
56     C_S_AXI_ADDR_WIDTH  : integer := 9
57     );
58     port (
59     S_AXI_ACLK  : in std_logic;
60     S_AXI_ARESETN : in std_logic;
61     S_AXI_AWADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
          downto 0);
62     S_AXI_AWPROT  : in std_logic_vector(2 downto 0);
63     S_AXI_AWVALID : in std_logic;
64     S_AXI_AWREADY : out std_logic;
65     S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
          0);
66     S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1
          downto 0);
67     S_AXI_WVALID  : in std_logic;
68     S_AXI_WREADY  : out std_logic;
69     S_AXI_BRESP : out std_logic_vector(1 downto 0);
70     S_AXI_BVALID  : out std_logic;
71     S_AXI_BREADY  : in std_logic;
72     S_AXI_ARADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
          downto 0);
73     S_AXI_ARPROT  : in std_logic_vector(2 downto 0);
74     S_AXI_ARVALID : in std_logic;
75     S_AXI_ARREADY : out std_logic;
76     S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
          0);
77     S_AXI_RRESP : out std_logic_vector(1 downto 0);
78     S_AXI_RVALID  : out std_logic;
79     S_AXI_RREADY  : in std_logic
80     );
81   end component top27baxilite;
82 begin
83
84 -- Instantiation of Axi Bus Interface S00_AXI
85
86   top27baxilite_inst : top27baxilite
87   generic map (
88     C_S_AXI_DATA_WIDTH  => C_S00_AXI_DATA_WIDTH,
89     C_S_AXI_ADDR_WIDTH  => C_S00_AXI_ADDR_WIDTH
90   )
91   port map (
92     S_AXI_ACLK  => s00_axi_aclk,
93     S_AXI_ARESETN => s00_axi_aresetn,
94     S_AXI_AWADDR  => s00_axi_awaddr,
95     S_AXI_AWPROT  => s00_axi_awprot,
96     S_AXI_AWVALID => s00_axi_awvalid,
97     S_AXI_AWREADY => s00_axi_awready,
98     S_AXI_WDATA => s00_axi_wdata,
```

```
99      S_AXI_WSTRB => s00_axi_wstrb,
100     S_AXI_WVALID  => s00_axi_wvalid,
101     S_AXI_WREADY  => s00_axi_wready,
102     S_AXI_BRESP => s00_axi_bresp,
103     S_AXI_BVALID  => s00_axi_bvalid,
104     S_AXI_BREADY  => s00_axi_bready,
105     S_AXI_ARADDR  => s00_axi_araddr,
106     S_AXI_ARPROT  => s00_axi_arprot,
107     S_AXI_ARVALID => s00_axi_arvalid,
108     S_AXI_ARREADY => s00_axi_arready,
109     S_AXI_RDATA => s00_axi_rdata,
110     S_AXI_RRESP => s00_axi_rresp,
111     S_AXI_RVALID  => s00_axi_rvalid,
112     S_AXI_RREADY  => s00_axi_rready
113   );
114   -- Add user logic here
115
116   -- User logic ends
117
118 end arch_imp;
```

Código A.12 – Código mais interno do AXI4-Lite encapsulando o perceptron.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.fpupack.all;
7  use work.rnapack.all;
8  entity top27baxilite is
9    generic (
10     -- Users to add parameters here
11
12     -- User parameters ends
13     -- Do not modify the parameters beyond this line
14
15     -- Width of S_AXI data bus
16     C_S_AXI_DATA_WIDTH  : integer := 32;
17     -- Width of S_AXI address bus
18     C_S_AXI_ADDR_WIDTH  : integer := 9
19   );
20   port (
21     -- Users to add ports here
22
23     -- User ports ends
24     -- Do not modify the ports beyond this line
25
26     -- Global Clock Signal
27     S_AXI_ACLK  : in std_logic;
28     -- Global Reset Signal. This Signal is Active LOW
29     S_AXI_ARESETN : in std_logic;
```

```
30      -- Write address (issued by master, acceped by Slave)
31      S_AXI_AWADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
           downto 0);
32      -- Write channel Protection type. This signal indicates the
33          -- privilege and security level of the transaction, and
               whether
34          -- the transaction is a data access or an instruction
               access.
35      S_AXI_AWPROT  : in std_logic_vector(2 downto 0);
36      -- Write address valid. This signal indicates that the master
           signaling
37          -- valid write address and control information.
38      S_AXI_AWVALID : in std_logic;
39      -- Write address ready. This signal indicates that the slave
           is ready
40          -- to accept an address and associated control signals.
41      S_AXI_AWREADY : out std_logic;
42      -- Write data (issued by master, acceped by Slave)
43      S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
           0);
44      -- Write strobes. This signal indicates which byte lanes hold
45          -- valid data. There is one write strobe bit for each eight
46          -- bits of the write data bus.
47      S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1
           downto 0);
48      -- Write valid. This signal indicates that valid write
49          -- data and strobes are available.
50      S_AXI_WVALID  : in std_logic;
51      -- Write ready. This signal indicates that the slave
52          -- can accept the write data.
53      S_AXI_WREADY  : out std_logic;
54      -- Write response. This signal indicates the status
55          -- of the write transaction.
56      S_AXI_BRESP : out std_logic_vector(1 downto 0);
57      -- Write response valid. This signal indicates that the channel
58          -- is signaling a valid write response.
59      S_AXI_BVALID  : out std_logic;
60      -- Response ready. This signal indicates that the master
61          -- can accept a write response.
62      S_AXI_BREADY  : in std_logic;
63      -- Read address (issued by master, acceped by Slave)
64      S_AXI_ARADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
           downto 0);
65      -- Protection type. This signal indicates the privilege
66          -- and security level of the transaction, and whether the
67          -- transaction is a data access or an instruction access.
68      S_AXI_ARPROT  : in std_logic_vector(2 downto 0);
69      -- Read address valid. This signal indicates that the channel
70          -- is signaling valid read address and control information.
71      S_AXI_ARVALID : in std_logic;
72      -- Read address ready. This signal indicates that the slave is
73          -- ready to accept an address and associated control
```

```
            signals.
74      S_AXI_ARREADY : out std_logic;
75      -- Read data (issued by slave)
76      S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
           0);
77      -- Read response. This signal indicates the status of the
78          -- read transfer.
79      S_AXI_RRESP : out std_logic_vector(1 downto 0);
80      -- Read valid. This signal indicates that the channel is
81          -- signaling the required read data.
82      S_AXI_RVALID  : out std_logic;
83      -- Read ready. This signal indicates that the master can
84          -- accept the read data and response information.
85      S_AXI_RREADY  : in std_logic
86    );
87  end top27baxilite;
88
89  architecture arch_imp of top27baxilite is
90
91    -- AXI4LITE signals
92    signal axi_awaddr : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto
         0);
93    signal axi_awready  : std_logic;
94    signal axi_wready : std_logic;
95    signal axi_bresp  : std_logic_vector(1 downto 0);
96    signal axi_bvalid : std_logic;
97    signal axi_araddr : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto
         0);
98    signal axi_arready  : std_logic;
99    signal axi_rdata  : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
         0);
100   signal axi_rresp  : std_logic_vector(1 downto 0);
101   signal axi_rvalid : std_logic;
102
103   -- Example-specific design signals
104   -- local parameter for addressing 32 bit / 64 bit
         C_S_AXI_DATA_WIDTH
105   -- ADDR_LSB is used for addressing 32/64 bit registers/memories
106   -- ADDR_LSB = 2 for 32 bits (n downto 2)
107   -- ADDR_LSB = 3 for 64 bits (n downto 3)
108   constant ADDR_LSB  : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
109   constant OPT_MEM_ADDR_BITS : integer := 3;
110   ------------------------------------------------
111   ---- Signals for user logic register space example
112   ------------------------------------------------
113   ---- Number of Slave Registers 7
114         signal START :std_logic:='0';
115         signal sready :std_logic:='0';
116         signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
             downto 0);
117         signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
             downto 0);
```

```
118         signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
119         signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
120         signal slv_reg4 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
121         signal slv_reg5 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
122         signal slv_reg6 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
123         signal slv_reg7 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
124         signal slv_reg8 :std_logic_vector(C_S_AXI_DATA_WIDTH-1
               downto 0);
125         signal s_RESULT_SLV_REG8
               :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 5);
126   signal slv_reg_rden : std_logic;
127   signal sreset : std_logic:='0';
128   signal slv_reg_wren : std_logic;
129   signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1
        downto 0);
130   signal byte_index : integer;
131   signal aw_en  : std_logic;
132
133       component top27b is
134
135    port(
136      x1: in std_logic_vector(26 downto 0);
137      x2: in std_logic_vector(26 downto 0);
138      x3: in std_logic_vector(26 downto 0);
139      x4: in std_logic_vector(26 downto 0);
140      w1: in std_logic_vector(26 downto 0);
141      w2: in std_logic_vector(26 downto 0);
142      w3: in std_logic_vector(26 downto 0);
143      w4: in std_logic_vector(26 downto 0);
144      clk: in std_logic;
145      rst: in std_logic;
146      start_neur: in std_logic;
147      ready: out std_logic;
148      result: out std_logic_vector(26 downto 0)   );
149 end component;
150 begin
151   -- I/O Connections assignments
152
153   S_AXI_AWREADY <= axi_awready;
154   S_AXI_WREADY  <= axi_wready;
155   S_AXI_BRESP <= axi_bresp;
156   S_AXI_BVALID  <= axi_bvalid;
157   S_AXI_ARREADY <= axi_arready;
158   S_AXI_RDATA <= axi_rdata;
159   S_AXI_RRESP <= axi_rresp;
160   S_AXI_RVALID  <= axi_rvalid;
```

```
161   -- Implement axi_awready generation
162   -- axi_awready is asserted for one S_AXI_ACLK clock cycle when
        both
163   -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
164   -- de-asserted when reset is low.
165
166   process (S_AXI_ACLK)
167   begin
168     if rising_edge(S_AXI_ACLK) then
169       if S_AXI_ARESETN = '0' then
170         axi_awready <= '0';
171         aw_en <= '1';
172       else
173         if (axi_awready = '0' and S_AXI_AWVALID = '1' and
              S_AXI_WVALID = '1' and aw_en = '1') then
174           -- slave is ready to accept write address when
175           -- there is a valid write address and write data
176           -- on the write address and data bus. This design
177           -- expects no outstanding transactions.
178              axi_awready <= '1';
179              aw_en <= '0';
180           elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
181              aw_en <= '1';
182              axi_awready <= '0';
183         else
184           axi_awready <= '0';
185         end if;
186       end if;
187     end if;
188   end process;
189
190   -- Implement axi_awaddr latching
191   -- This process is used to latch the address when both
192   -- S_AXI_AWVALID and S_AXI_WVALID are valid.
193
194   process (S_AXI_ACLK)
195   begin
196     if rising_edge(S_AXI_ACLK) then
197       if S_AXI_ARESETN = '0' then
198         axi_awaddr <= (others => '0');
199       else
200         if (axi_awready = '0' and S_AXI_AWVALID = '1' and
              S_AXI_WVALID = '1' and aw_en = '1') then
201           -- Write Address latching
202           axi_awaddr <= S_AXI_AWADDR;
203         end if;
204       end if;
205     end if;
206   end process;
207
208   -- Implement axi_wready generation
209   -- axi_wready is asserted for one S_AXI_ACLK clock cycle when
```

```
    both
210 -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
211 -- de-asserted when reset is low.
212
213 process (S_AXI_ACLK)
214 begin
215   if rising_edge(S_AXI_ACLK) then
216     if S_AXI_ARESETN = '0' then
217       axi_wready <= '0';
218     else
219       if (axi_wready = '0' and S_AXI_WVALID = '1' and
              S_AXI_AWVALID = '1' and aw_en = '1') then
220           -- slave is ready to accept write data when
221           -- there is a valid write address and write data
222           -- on the write address and data bus. This design
223           -- expects no outstanding transactions.
224           axi_wready <= '1';
225       else
226         axi_wready <= '0';
227       end if;
228     end if;
229   end if;
230 end process;
231
232 -- Implement memory mapped register select and write logic
    generation
233 -- The write data is accepted and written to memory mapped
    registers when
234 -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are
    asserted. Write strobes are used to
235 -- select byte enables of slave registers while writing.
236 -- These registers are cleared when reset (active low) is
    applied.
237 -- Slave register write enable is asserted when valid address
    and data are available
238 -- and the slave is ready to accept the write address and write
    data.
239 slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and
    S_AXI_AWVALID ;
240
241 process (S_AXI_ACLK)
242 variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
243 begin
244       if rising_edge(S_AXI_ACLK) then
245         if S_AXI_ARESETN = '0' then
246           slv_reg0 <=(others=>'0');
247           slv_reg1 <=(others=>'0');
248           slv_reg2 <=(others=>'0');
249           slv_reg3 <=(others=>'0');
250           slv_reg4 <=(others=>'0');
251           slv_reg5 <=(others=>'0');
252           slv_reg6 <=(others=>'0');
```

```
253               slv_reg7 <=(others=>'0');
254               slv_reg8 <=(others=>'0');
255               START<='0';
256             else
257               START<='0';
258               loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS
                     downto ADDR_LSB);
259               if (slv_reg_wren = '1') then
260                 case loc_addr is
261                   when b"0000"=>
262                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
263                       if ( S_AXI_WSTRB(byte_index) = '1' ) then
264                         slv_reg0(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
265                       end if;
266                     end loop;
267                   when b"0001"=>
268                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
269                       if ( S_AXI_WSTRB(byte_index) = '1' ) then
270                         slv_reg1(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
271                       end if;
272                     end loop;
273                   when b"0010"=>
274                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
275                       if ( S_AXI_WSTRB(byte_index) = '1' ) then
276                         slv_reg2(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
277                       end if;
278                     end loop;
279                   when b"0011"=>
280                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
281                       if ( S_AXI_WSTRB(byte_index) = '1' ) then
282                         slv_reg3(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
283                       end if;
284                     end loop;
285                   when b"0100"=>
286                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
287                       if ( S_AXI_WSTRB(byte_index) = '1' ) then
288                         slv_reg4(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
```

```
289                     end if;
290                   end loop;
291                 when b"0101"=>
292                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
293                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
294                       slv_reg5(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
295                     end if;
296                   end loop;
297                 when b"0110"=>
298                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
299                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
300                       slv_reg6(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
301                     end if;
302                   end loop;
303                 when b"0111"=>
304                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
305                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
306                       slv_reg7(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
307                       START<='1';
308                     end if;
309                   end loop;
310                 when b"1000"=>
311                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
                         loop
312                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
313                       slv_reg8(byte_index*8+7 downto byte_index*8)
                             <= S_AXI_WDATA(byte_index*8+7 downto
                             byte_index*8);
314                     end if;
315                   end loop;
316                 when others =>
317                   START<='0';
318                   slv_reg0<=slv_reg0;
319                   slv_reg1<=slv_reg1;
320                   slv_reg2<=slv_reg2;
321                   slv_reg3<=slv_reg3;
322                   slv_reg4<=slv_reg4;
323                   slv_reg5<=slv_reg5;
324                   slv_reg6<=slv_reg6;
325                   slv_reg7<=slv_reg7;
326                   slv_reg8<=slv_reg8;
327               end case;
328             end if;
```

```
329             end if;
330           end if;
331   end process;
332
333   -- Implement write response logic generation
334   -- The write response and response valid signals are asserted by
      the slave
335   -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID
      are asserted.
336   -- This marks the acceptance of address and indicates the status
      of
337   -- write transaction.
338
339   process (S_AXI_ACLK)
340   begin
341     if rising_edge(S_AXI_ACLK) then
342       if S_AXI_ARESETN = '0' then
343         axi_bvalid  <= '0';
344         axi_bresp   <= "00"; --need to work more on the responses
345       else
346         if (axi_awready = '1' and S_AXI_AWVALID = '1' and
              axi_wready = '1' and S_AXI_WVALID = '1' and axi_bvalid
              = '0'  ) then
347           axi_bvalid <= '1';
348           axi_bresp  <= "00";
349         elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
              --check if bready is asserted while bvalid is high)
350           axi_bvalid <= '0';                                 --
                (there is a possibility that bready is always
                asserted high)
351         end if;
352       end if;
353     end if;
354   end process;
355
356   -- Implement axi_arready generation
357   -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
358   -- S_AXI_ARVALID is asserted. axi_awready is
359   -- de-asserted when reset (active low) is asserted.
360   -- The read address is also latched when S_AXI_ARVALID is
361   -- asserted. axi_araddr is reset to zero on reset assertion.
362
363   process (S_AXI_ACLK)
364   begin
365     if rising_edge(S_AXI_ACLK) then
366       if S_AXI_ARESETN = '0' then
367         axi_arready <= '0';
368         axi_araddr  <= (others => '1');
369       else
370         if (axi_arready = '0' and S_AXI_ARVALID = '1') then
371           -- indicates that the slave has acceped the valid read
              address
```

```
372           axi_arready <= '1';
373           -- Read Address latching
374           axi_araddr  <= S_AXI_ARADDR;
375         else
376           axi_arready <= '0';
377         end if;
378       end if;
379     end if;
380   end process;
381
382   -- Implement axi_arvalid generation
383   -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when
      both
384   -- S_AXI_ARVALID and axi_arready are asserted. The slave
      registers
385   -- data are available on the axi_rdata bus at this instance. The
386   -- assertion of axi_rvalid marks the validity of read data on the
387   -- bus and axi_rresp indicates the status of read
      transaction.axi_rvalid
388   -- is deasserted on reset (active low). axi_rresp and axi_rdata
      are
389   -- cleared to zero on reset (active low).
390   process (S_AXI_ACLK)
391   begin
392     if rising_edge(S_AXI_ACLK) then
393       if S_AXI_ARESETN = '0' then
394         axi_rvalid <= '0';
395         axi_rresp  <= "00";
396       else
397         if (axi_arready = '1' and S_AXI_ARVALID = '1' and
              axi_rvalid = '0') then
398           -- Valid read data is available at the read data bus
399           axi_rvalid <= '1';
400           axi_rresp  <= "00"; -- 'OKAY' response
401         elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
402           -- Read data is accepted by the master
403           axi_rvalid <= '0';
404         end if;
405       end if;
406     end if;
407   end process;
408
409   -- Implement memory mapped register select and read logic
      generation
410   -- Slave register read enable is asserted when valid address is
      available
411   -- and the slave is ready to accept the read address.
412   slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not
      axi_rvalid) ;
413
414   process
      (slv_reg0,slv_reg1,slv_reg2,slv_reg3,slv_reg4,slv_reg5,slv_reg6,slv_reg
```

```
    S_AXI_ARESETN, slv_reg_rden,sready)
  variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
  begin
      -- Address decoding for reading registers
      loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto
         ADDR_LSB);
            case loc_addr is
              when b"0000" =>
                reg_data_out <= slv_reg0;
              when b"0001" =>
                reg_data_out <= slv_reg1;
              when b"0010" =>
                reg_data_out <= slv_reg2;
              when b"0011" =>
                reg_data_out <= slv_reg3;
              when b"0100" =>
                reg_data_out <= slv_reg4;
              when b"0101" =>
                reg_data_out <= slv_reg5;
              when b"0110" =>
                reg_data_out <= slv_reg6;
              when b"0111" =>
                reg_data_out <= slv_reg7;
              when b"1000" =>
                reg_data_out <=s_RESULT_SLV_REG8&"00000";
              when others=>
                reg_data_out <= (others=>'0');
            end case;
      --reg
  end process;

  -- Output register or memory read data
  process( S_AXI_ACLK ) is
  begin
    if (rising_edge (S_AXI_ACLK)) then
      if ( S_AXI_ARESETN = '0' ) then
        axi_rdata  <= (others => '0');
      else
        if (slv_reg_rden = '1') then
          -- When there is a valid read address (S_AXI_ARVALID)
             with
          -- acceptance of read address by the slave (axi_arready),
          -- output the read dada
          -- Read address mux
            axi_rdata <= reg_data_out;     -- register read data
        end if;
      end if;
    end if;
  end process;
  sreset<=not S_AXI_ARESETN;
    uut: top27b port map(
        X1=>slv_reg0(31 downto 5),
```

```
464         X2=>slv_reg1(31 downto 5),
465         X3=>slv_reg2(31 downto 5),
466         X4=>slv_reg3(31 downto 5),
467         W1=>slv_reg4(31 downto 5),
468         W2=>slv_reg5(31 downto 5),
469         W3=>slv_reg6(31 downto 5),
470         W4=>slv_reg7(31 downto 5),
471         CLK=>S_AXI_ACLK,
472         RST=>sreset,
473         START_NEUR=>START,
474         READY=> sready,
475         RESULT=>s_RESULT_SLV_REG8);
476
477 end arch_imp;
```

Código A.13 – Código mais extrerno do AXI4-Full encapsulando a MLP.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity MLP443axifull_MM_v1_0 is
6    generic (
7      -- Users to add parameters here
8
9      -- User parameters ends
10     -- Do not modify the parameters beyond this line
11
12
13     -- Parameters of Axi Slave Bus Interface S00_AXI
14     C_S00_AXI_ID_WIDTH   : integer := 1;
15     C_S00_AXI_DATA_WIDTH   : integer := 32;
16     C_S00_AXI_ADDR_WIDTH   : integer := 6;
17     C_S00_AXI_AWUSER_WIDTH   : integer := 0;
18     C_S00_AXI_ARUSER_WIDTH   : integer := 0;
19     C_S00_AXI_WUSER_WIDTH : integer := 0;
20     C_S00_AXI_RUSER_WIDTH : integer := 0;
21     C_S00_AXI_BUSER_WIDTH : integer := 0
22   );
23   port (
24     -- Users to add ports here
25
26     -- User ports ends
27     -- Do not modify the ports beyond this line
28
29
30     -- Ports of Axi Slave Bus Interface S00_AXI
31     s00_axi_aclk   : in std_logic;
32     s00_axi_aresetn : in std_logic;
33     s00_axi_awid   : in std_logic_vector(C_S00_AXI_ID_WIDTH-1
         downto 0);
```

```
34      s00_axi_awaddr  : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1
           downto 0);
35      s00_axi_awlen : in std_logic_vector(7 downto 0);
36      s00_axi_awsize  : in std_logic_vector(2 downto 0);
37      s00_axi_awburst : in std_logic_vector(1 downto 0);
38      s00_axi_awlock  : in std_logic;
39      s00_axi_awcache : in std_logic_vector(3 downto 0);
40      s00_axi_awprot  : in std_logic_vector(2 downto 0);
41      s00_axi_awqos : in std_logic_vector(3 downto 0);
42      s00_axi_awregion  : in std_logic_vector(3 downto 0);
43      s00_axi_awuser  : in std_logic_vector(C_S00_AXI_AWUSER_WIDTH-1
           downto 0);
44      s00_axi_awvalid : in std_logic;
45      s00_axi_awready : out std_logic;
46      s00_axi_wdata : in std_logic_vector(C_S00_AXI_DATA_WIDTH-1
           downto 0);
47      s00_axi_wstrb : in std_logic_vector((C_S00_AXI_DATA_WIDTH/8)-1
           downto 0);
48      s00_axi_wlast : in std_logic;
49      s00_axi_wuser : in std_logic_vector(C_S00_AXI_WUSER_WIDTH-1
           downto 0);
50      s00_axi_wvalid  : in std_logic;
51      s00_axi_wready  : out std_logic;
52      s00_axi_bid : out std_logic_vector(C_S00_AXI_ID_WIDTH-1 downto
           0);
53      s00_axi_bresp : out std_logic_vector(1 downto 0);
54      s00_axi_buser : out std_logic_vector(C_S00_AXI_BUSER_WIDTH-1
           downto 0);
55      s00_axi_bvalid  : out std_logic;
56      s00_axi_bready  : in std_logic;
57      s00_axi_arid  : in std_logic_vector(C_S00_AXI_ID_WIDTH-1
           downto 0);
58      s00_axi_araddr  : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1
           downto 0);
59      s00_axi_arlen : in std_logic_vector(7 downto 0);
60      s00_axi_arsize  : in std_logic_vector(2 downto 0);
61      s00_axi_arburst : in std_logic_vector(1 downto 0);
62      s00_axi_arlock  : in std_logic;
63      s00_axi_arcache : in std_logic_vector(3 downto 0);
64      s00_axi_arprot  : in std_logic_vector(2 downto 0);
65      s00_axi_arqos : in std_logic_vector(3 downto 0);
66      s00_axi_arregion  : in std_logic_vector(3 downto 0);
67      s00_axi_aruser  : in std_logic_vector(C_S00_AXI_ARUSER_WIDTH-1
           downto 0);
68      s00_axi_arvalid : in std_logic;
69      s00_axi_arready : out std_logic;
70      s00_axi_rid : out std_logic_vector(C_S00_AXI_ID_WIDTH-1 downto
           0);
71      s00_axi_rdata : out std_logic_vector(C_S00_AXI_DATA_WIDTH-1
           downto 0);
72      s00_axi_rresp : out std_logic_vector(1 downto 0);
73      s00_axi_rlast : out std_logic;
```

```
74      s00_axi_ruser : out std_logic_vector(C_S00_AXI_RUSER_WIDTH-1
            downto 0);
75      s00_axi_rvalid  : out std_logic;
76      s00_axi_rready  : in std_logic
77    );
78  end MLP443axifull_MM_v1_0;
79
80  architecture arch_imp of MLP443axifull_MM_v1_0 is
81
82    -- component declaration
83    component MLP443axifull_MM_v1_0_S00_AXI is
84      generic (
85      C_S_AXI_ID_WIDTH  : integer := 1;
86      C_S_AXI_DATA_WIDTH  : integer := 32;
87      C_S_AXI_ADDR_WIDTH  : integer := 6;
88      C_S_AXI_AWUSER_WIDTH  : integer := 0;
89      C_S_AXI_ARUSER_WIDTH  : integer := 0;
90      C_S_AXI_WUSER_WIDTH : integer := 0;
91      C_S_AXI_RUSER_WIDTH : integer := 0;
92      C_S_AXI_BUSER_WIDTH : integer := 0
93      );
94      port (
95      S_AXI_ACLK  : in std_logic;
96      S_AXI_ARESETN : in std_logic;
97      S_AXI_AWID  : in std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
98      S_AXI_AWADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
            downto 0);
99      S_AXI_AWLEN : in std_logic_vector(7 downto 0);
100     S_AXI_AWSIZE  : in std_logic_vector(2 downto 0);
101     S_AXI_AWBURST : in std_logic_vector(1 downto 0);
102     S_AXI_AWLOCK  : in std_logic;
103     S_AXI_AWCACHE : in std_logic_vector(3 downto 0);
104     S_AXI_AWPROT  : in std_logic_vector(2 downto 0);
105     S_AXI_AWQOS : in std_logic_vector(3 downto 0);
106     S_AXI_AWREGION  : in std_logic_vector(3 downto 0);
107     S_AXI_AWUSER  : in std_logic_vector(C_S_AXI_AWUSER_WIDTH-1
            downto 0);
108     S_AXI_AWVALID : in std_logic;
109     S_AXI_AWREADY : out std_logic;
110     S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
            0);
111     S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1
            downto 0);
112     S_AXI_WLAST : in std_logic;
113     S_AXI_WUSER : in std_logic_vector(C_S_AXI_WUSER_WIDTH-1 downto
            0);
114     S_AXI_WVALID  : in std_logic;
115     S_AXI_WREADY  : out std_logic;
116     S_AXI_BID : out std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
117     S_AXI_BRESP : out std_logic_vector(1 downto 0);
118     S_AXI_BUSER : out std_logic_vector(C_S_AXI_BUSER_WIDTH-1
            downto 0);
```

```
119     S_AXI_BVALID  : out std_logic;
120     S_AXI_BREADY  : in std_logic;
121     S_AXI_ARID    : in std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
122     S_AXI_ARADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
          downto 0);
123     S_AXI_ARLEN   : in std_logic_vector(7 downto 0);
124     S_AXI_ARSIZE  : in std_logic_vector(2 downto 0);
125     S_AXI_ARBURST : in std_logic_vector(1 downto 0);
126     S_AXI_ARLOCK  : in std_logic;
127     S_AXI_ARCACHE : in std_logic_vector(3 downto 0);
128     S_AXI_ARPROT  : in std_logic_vector(2 downto 0);
129     S_AXI_ARQOS   : in std_logic_vector(3 downto 0);
130     S_AXI_ARREGION  : in std_logic_vector(3 downto 0);
131     S_AXI_ARUSER  : in std_logic_vector(C_S_AXI_ARUSER_WIDTH-1
          downto 0);
132     S_AXI_ARVALID : in std_logic;
133     S_AXI_ARREADY : out std_logic;
134     S_AXI_RID     : out std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
135     S_AXI_RDATA   : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
          0);
136     S_AXI_RRESP   : out std_logic_vector(1 downto 0);
137     S_AXI_RLAST   : out std_logic;
138     S_AXI_RUSER   : out std_logic_vector(C_S_AXI_RUSER_WIDTH-1
          downto 0);
139     S_AXI_RVALID  : out std_logic;
140     S_AXI_RREADY  : in std_logic
141     );
142   end component MLP443axifull_MM_v1_0_S00_AXI;
143
144 begin
145
146 -- Instantiation of Axi Bus Interface S00_AXI
147 MLP443axifull_MM_v1_0_S00_AXI_inst : MLP443axifull_MM_v1_0_S00_AXI
148   generic map (
149     C_S_AXI_ID_WIDTH  => C_S00_AXI_ID_WIDTH,
150     C_S_AXI_DATA_WIDTH  => C_S00_AXI_DATA_WIDTH,
151     C_S_AXI_ADDR_WIDTH  => C_S00_AXI_ADDR_WIDTH,
152     C_S_AXI_AWUSER_WIDTH  => C_S00_AXI_AWUSER_WIDTH,
153     C_S_AXI_ARUSER_WIDTH  => C_S00_AXI_ARUSER_WIDTH,
154     C_S_AXI_WUSER_WIDTH => C_S00_AXI_WUSER_WIDTH,
155     C_S_AXI_RUSER_WIDTH => C_S00_AXI_RUSER_WIDTH,
156     C_S_AXI_BUSER_WIDTH => C_S00_AXI_BUSER_WIDTH
157   )
158   port map (
159     S_AXI_ACLK  => s00_axi_aclk,
160     S_AXI_ARESETN => s00_axi_aresetn,
161     S_AXI_AWID  => s00_axi_awid,
162     S_AXI_AWADDR  => s00_axi_awaddr,
163     S_AXI_AWLEN => s00_axi_awlen,
164     S_AXI_AWSIZE  => s00_axi_awsize,
165     S_AXI_AWBURST => s00_axi_awburst,
166     S_AXI_AWLOCK  => s00_axi_awlock,
```

```
167     S_AXI_AWCACHE => s00_axi_awcache,
168     S_AXI_AWPROT  => s00_axi_awprot,
169     S_AXI_AWQOS => s00_axi_awqos,
170     S_AXI_AWREGION  => s00_axi_awregion,
171     S_AXI_AWUSER  => s00_axi_awuser,
172     S_AXI_AWVALID => s00_axi_awvalid,
173     S_AXI_AWREADY => s00_axi_awready,
174     S_AXI_WDATA => s00_axi_wdata,
175     S_AXI_WSTRB => s00_axi_wstrb,
176     S_AXI_WLAST => s00_axi_wlast,
177     S_AXI_WUSER => s00_axi_wuser,
178     S_AXI_WVALID  => s00_axi_wvalid,
179     S_AXI_WREADY  => s00_axi_wready,
180     S_AXI_BID => s00_axi_bid,
181     S_AXI_BRESP => s00_axi_bresp,
182     S_AXI_BUSER => s00_axi_buser,
183     S_AXI_BVALID  => s00_axi_bvalid,
184     S_AXI_BREADY  => s00_axi_bready,
185     S_AXI_ARID  => s00_axi_arid,
186     S_AXI_ARADDR  => s00_axi_araddr,
187     S_AXI_ARLEN => s00_axi_arlen,
188     S_AXI_ARSIZE  => s00_axi_arsize,
189     S_AXI_ARBURST => s00_axi_arburst,
190     S_AXI_ARLOCK  => s00_axi_arlock,
191     S_AXI_ARCACHE => s00_axi_arcache,
192     S_AXI_ARPROT  => s00_axi_arprot,
193     S_AXI_ARQOS => s00_axi_arqos,
194     S_AXI_ARREGION  => s00_axi_arregion,
195     S_AXI_ARUSER  => s00_axi_aruser,
196     S_AXI_ARVALID => s00_axi_arvalid,
197     S_AXI_ARREADY => s00_axi_arready,
198     S_AXI_RID => s00_axi_rid,
199     S_AXI_RDATA => s00_axi_rdata,
200     S_AXI_RRESP => s00_axi_rresp,
201     S_AXI_RLAST => s00_axi_rlast,
202     S_AXI_RUSER => s00_axi_ruser,
203     S_AXI_RVALID  => s00_axi_rvalid,
204     S_AXI_RREADY  => s00_axi_rready
205   );
206
207   -- Add user logic here
208
209   -- User logic ends
210
211 end arch_imp;
```

Código A.14 – Código mais interno do AXI4-Full encapsulando a MLP.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 library work;
```

```
5  use work.fpupack.all;
6  use work.rnapack.all;
7
8  entity MLP443axifull_MM_v1_0_S00_AXI is
9    generic (
10     -- Users to add parameters here
11
12     -- User parameters ends
13     -- Do not modify the parameters beyond this line
14
15     -- Width of ID for for write address, write data, read address
          and read data
16     C_S_AXI_ID_WIDTH  : integer := 1;
17     -- Width of S_AXI data bus
18     C_S_AXI_DATA_WIDTH  : integer := 32;
19     -- Width of S_AXI address bus
20     C_S_AXI_ADDR_WIDTH  : integer := 6;
21     -- Width of optional user defined signal in write address
          channel
22     C_S_AXI_AWUSER_WIDTH  : integer := 0;
23     -- Width of optional user defined signal in read address
          channel
24     C_S_AXI_ARUSER_WIDTH  : integer := 0;
25     -- Width of optional user defined signal in write data channel
26     C_S_AXI_WUSER_WIDTH : integer := 0;
27     -- Width of optional user defined signal in read data channel
28     C_S_AXI_RUSER_WIDTH : integer := 0;
29     -- Width of optional user defined signal in write response
          channel
30     C_S_AXI_BUSER_WIDTH : integer := 0
31   );
32   port (
33     -- Users to add ports here
34
35     -- User ports ends
36     -- Do not modify the ports beyond this line
37
38     -- Global Clock Signal
39     S_AXI_ACLK   : in std_logic;
40     -- Global Reset Signal. This Signal is Active LOW
41     S_AXI_ARESETN : in std_logic;
42     -- Write Address ID
43     S_AXI_AWID   : in std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
44     -- Write address
45     S_AXI_AWADDR   : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
          downto 0);
46     -- Burst length. The burst length gives the exact number of
          transfers in a burst
47     S_AXI_AWLEN : in std_logic_vector(7 downto 0);
48     -- Burst size. This signal indicates the size of each transfer
          in the burst
49     S_AXI_AWSIZE   : in std_logic_vector(2 downto 0);
```

```
50      -- Burst type. The burst type and the size information,
51      -- determine how the address for each transfer within the
           burst is calculated.
52      S_AXI_AWBURST : in std_logic_vector(1 downto 0);
53      -- Lock type. Provides additional information about the
54      -- atomic characteristics of the transfer.
55      S_AXI_AWLOCK  : in std_logic;
56      -- Memory type. This signal indicates how transactions
57      -- are required to progress through a system.
58      S_AXI_AWCACHE : in std_logic_vector(3 downto 0);
59      -- Protection type. This signal indicates the privilege
60      -- and security level of the transaction, and whether
61      -- the transaction is a data access or an instruction access.
62      S_AXI_AWPROT  : in std_logic_vector(2 downto 0);
63      -- Quality of Service, QoS identifier sent for each
64      -- write transaction.
65      S_AXI_AWQOS : in std_logic_vector(3 downto 0);
66      -- Region identifier. Permits a single physical interface
67      -- on a slave to be used for multiple logical interfaces.
68      S_AXI_AWREGION  : in std_logic_vector(3 downto 0);
69      -- Optional User-defined signal in the write address channel.
70      S_AXI_AWUSER  : in std_logic_vector(C_S_AXI_AWUSER_WIDTH-1
           downto 0);
71      -- Write address valid. This signal indicates that
72      -- the channel is signaling valid write address and
73      -- control information.
74      S_AXI_AWVALID : in std_logic;
75      -- Write address ready. This signal indicates that
76      -- the slave is ready to accept an address and associated
77      -- control signals.
78      S_AXI_AWREADY : out std_logic;
79      -- Write Data
80      S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
           0);
81      -- Write strobes. This signal indicates which byte
82      -- lanes hold valid data. There is one write strobe
83      -- bit for each eight bits of the write data bus.
84      S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1
           downto 0);
85      -- Write last. This signal indicates the last transfer
86      -- in a write burst.
87      S_AXI_WLAST : in std_logic;
88      -- Optional User-defined signal in the write data channel.
89      S_AXI_WUSER : in std_logic_vector(C_S_AXI_WUSER_WIDTH-1 downto
           0);
90      -- Write valid. This signal indicates that valid write
91      -- data and strobes are available.
92      S_AXI_WVALID  : in std_logic;
93      -- Write ready. This signal indicates that the slave
94      -- can accept the write data.
95      S_AXI_WREADY  : out std_logic;
96      -- Response ID tag. This signal is the ID tag of the
```

```
97      -- write response.
98      S_AXI_BID : out std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
99      -- Write response. This signal indicates the status
100     -- of the write transaction.
101     S_AXI_BRESP : out std_logic_vector(1 downto 0);
102     -- Optional User-defined signal in the write response channel.
103     S_AXI_BUSER : out std_logic_vector(C_S_AXI_BUSER_WIDTH-1
           downto 0);
104     -- Write response valid. This signal indicates that the
105     -- channel is signaling a valid write response.
106     S_AXI_BVALID  : out std_logic;
107     -- Response ready. This signal indicates that the master
108     -- can accept a write response.
109     S_AXI_BREADY  : in std_logic;
110     -- Read address ID. This signal is the identification
111     -- tag for the read address group of signals.
112     S_AXI_ARID  : in std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
113     -- Read address. This signal indicates the initial
114     -- address of a read burst transaction.
115     S_AXI_ARADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
           downto 0);
116     -- Burst length. The burst length gives the exact number of
           transfers in a burst
117     S_AXI_ARLEN : in std_logic_vector(7 downto 0);
118     -- Burst size. This signal indicates the size of each transfer
           in the burst
119     S_AXI_ARSIZE  : in std_logic_vector(2 downto 0);
120     -- Burst type. The burst type and the size information,
121     -- determine how the address for each transfer within the
           burst is calculated.
122     S_AXI_ARBURST : in std_logic_vector(1 downto 0);
123     -- Lock type. Provides additional information about the
124     -- atomic characteristics of the transfer.
125     S_AXI_ARLOCK  : in std_logic;
126     -- Memory type. This signal indicates how transactions
127     -- are required to progress through a system.
128     S_AXI_ARCACHE : in std_logic_vector(3 downto 0);
129     -- Protection type. This signal indicates the privilege
130     -- and security level of the transaction, and whether
131     -- the transaction is a data access or an instruction access.
132     S_AXI_ARPROT  : in std_logic_vector(2 downto 0);
133     -- Quality of Service, QoS identifier sent for each
134     -- read transaction.
135     S_AXI_ARQOS : in std_logic_vector(3 downto 0);
136     -- Region identifier. Permits a single physical interface
137     -- on a slave to be used for multiple logical interfaces.
138     S_AXI_ARREGION  : in std_logic_vector(3 downto 0);
139     -- Optional User-defined signal in the read address channel.
140     S_AXI_ARUSER  : in std_logic_vector(C_S_AXI_ARUSER_WIDTH-1
           downto 0);
141     -- Write address valid. This signal indicates that
142     -- the channel is signaling valid read address and
```

```
143     -- control information.
144     S_AXI_ARVALID : in std_logic;
145     -- Read address ready. This signal indicates that
146     -- the slave is ready to accept an address and associated
147     -- control signals.
148     S_AXI_ARREADY : out std_logic;
149     -- Read ID tag. This signal is the identification tag
150     -- for the read data group of signals generated by the slave.
151     S_AXI_RID : out std_logic_vector(C_S_AXI_ID_WIDTH-1 downto 0);
152     -- Read Data
153     S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
            0);
154     -- Read response. This signal indicates the status of
155     -- the read transfer.
156     S_AXI_RRESP : out std_logic_vector(1 downto 0);
157     -- Read last. This signal indicates the last transfer
158     -- in a read burst.
159     S_AXI_RLAST : out std_logic;
160     -- Optional User-defined signal in the read address channel.
161     S_AXI_RUSER : out std_logic_vector(C_S_AXI_RUSER_WIDTH-1
            downto 0);
162     -- Read valid. This signal indicates that the channel
163     -- is signaling the required read data.
164     S_AXI_RVALID  : out std_logic;
165     -- Read ready. This signal indicates that the master can
166     -- accept the read data and response information.
167     S_AXI_RREADY  : in std_logic
168   );
169 end MLP443axifull_MM_v1_0_S00_AXI;
170
171 architecture arch_imp of MLP443axifull_MM_v1_0_S00_AXI is
172
173   -- AXI4FULL signals
174   signal axi_awaddr : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto
          0);
175   signal axi_awready  : std_logic;
176   signal axi_wready : std_logic;
177   signal axi_bresp  : std_logic_vector(1 downto 0);
178   signal axi_buser  : std_logic_vector(C_S_AXI_BUSER_WIDTH-1
          downto 0);
179   signal axi_bvalid : std_logic;
180   signal axi_araddr : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto
          0);
181   signal axi_arready  : std_logic;
182   signal axi_rdata  : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
          0);
183   signal axi_rresp  : std_logic_vector(1 downto 0);
184   signal axi_rlast  : std_logic;
185   signal axi_ruser  : std_logic_vector(C_S_AXI_RUSER_WIDTH-1
          downto 0);
186   signal axi_rvalid : std_logic;
187   -- aw_wrap_en determines wrap boundary and enables wrapping
```

```
188   signal  aw_wrap_en : std_logic;
189   -- ar_wrap_en determines wrap boundary and enables wrapping
190   signal  ar_wrap_en : std_logic;
191   -- aw_wrap_size is the size of the write transfer, the
192   -- write address wraps to a lower address if upper address
193   -- limit is reached
194   signal aw_wrap_size : integer;
195   -- ar_wrap_size is the size of the read transfer, the
196   -- read address wraps to a lower address if upper address
197   -- limit is reached
198   signal ar_wrap_size : integer;
199   -- The axi_awv_awr_flag flag marks the presence of write address
        valid
200   signal axi_awv_awr_flag    : std_logic;
201   --The axi_arv_arr_flag flag marks the presence of read address
        valid
202   signal axi_arv_arr_flag    : std_logic;
203   -- The axi_awlen_cntr internal write address counter to keep
        track of beats in a burst transaction
204   signal axi_awlen_cntr      : std_logic_vector(7 downto 0);
205   --The axi_arlen_cntr internal read address counter to keep track
        of beats in a burst transaction
206   signal axi_arlen_cntr      : std_logic_vector(7 downto 0);
207   signal axi_arburst      : std_logic_vector(2-1 downto 0);
208   signal axi_awburst      : std_logic_vector(2-1 downto 0);
209   signal axi_arlen      : std_logic_vector(8-1 downto 0);
210   signal axi_awlen      : std_logic_vector(8-1 downto 0);
211   --local parameter for addressing 32 bit / 64 bit
        C_S_AXI_DATA_WIDTH
212   --ADDR_LSB is used for addressing 32/64 bit registers/memories
213   --ADDR_LSB = 2 for 32 bits (n downto 2)
214   --ADDR_LSB = 3 for 42 bits (n downto 3)
215
216   constant ADDR_LSB  : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
217   constant OPT_MEM_ADDR_BITS : integer :=  3;
218   constant USER_NUM_MEM: integer := 1;
219   constant low : std_logic_vector (C_S_AXI_ADDR_WIDTH - 1 downto
        0) := (others=>'0');
220   ------------------------------------------------
221   ---- Signals for user logic memory space example
222   ------------------------------------------------
223   signal mem_address : std_logic_vector(OPT_MEM_ADDR_BITS downto
        0);
224   signal mem_select : std_logic_vector(USER_NUM_MEM-1 downto 0);
225   type word_array is array (0 to USER_NUM_MEM-1) of
        std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
226   signal mem_data_out : word_array;
227
228   signal i : integer;
229   signal j : integer;
230   signal mem_byte_index : integer;
231   type BYTE_RAM_TYPE is array (0 to 15) of std_logic_vector(7
```

```
      downto 0);
232   type WORD_RAM_TYPE is array (0 to 15) of std_logic_vector(31
      downto 0);
233   signal top_input_array : WORD_RAM_TYPE;
234         signal sSTART :std_logic:='0';
235         signal sREADY :std_logic:='0';
236         signal sreset :std_logic:='0';
237         signal s_OUTsx:ARRAY_IN;
238         signal s_INssaida:ARRAY_OUT;
239
240   component MLP443 is
241     Port ( clk : in STD_LOGIC;
242            reset : in STD_LOGIC;
243            start : in STD_LOGIC;
244            x : in array_in;
245            saida : out array_out;
246            ready : out STD_LOGIC);
247 end component;
248   -- signals for user logic
249
250
251 begin
252   -- I/O Connections assignments
253
254   S_AXI_AWREADY <= axi_awready;
255   S_AXI_WREADY  <= axi_wready;
256   S_AXI_BRESP <= axi_bresp;
257   S_AXI_BUSER <= axi_buser;
258   S_AXI_BVALID  <= axi_bvalid;
259   S_AXI_ARREADY <= axi_arready;
260   S_AXI_RDATA <= axi_rdata;
261   S_AXI_RRESP <= axi_rresp;
262   S_AXI_RLAST <= axi_rlast;
263   S_AXI_RUSER <= axi_ruser;
264   S_AXI_RVALID  <= axi_rvalid;
265   S_AXI_BID <= S_AXI_AWID;
266   S_AXI_RID <= S_AXI_ARID;
267   aw_wrap_size <= ((C_S_AXI_DATA_WIDTH)/8 *
      to_integer(unsigned(axi_awlen)));
268   ar_wrap_size <= ((C_S_AXI_DATA_WIDTH)/8 *
      to_integer(unsigned(axi_arlen)));
269   aw_wrap_en <= '1' when (((axi_awaddr AND
      std_logic_vector(to_unsigned(aw_wrap_size,C_S_AXI_ADDR_WIDTH)))
      XOR
      std_logic_vector(to_unsigned(aw_wrap_size,C_S_AXI_ADDR_WIDTH)))
      = low) else '0';
270   ar_wrap_en <= '1' when (((axi_araddr AND
      std_logic_vector(to_unsigned(ar_wrap_size,C_S_AXI_ADDR_WIDTH)))
      XOR
      std_logic_vector(to_unsigned(ar_wrap_size,C_S_AXI_ADDR_WIDTH)))
      = low) else '0';
271
```

```
272   -- Implement axi_awready generation
273
274   -- axi_awready is asserted for one S_AXI_ACLK clock cycle when
         both
275   -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
276   -- de-asserted when reset is low.
277
278   process (S_AXI_ACLK)
279   begin
280     if rising_edge(S_AXI_ACLK) then
281       if S_AXI_ARESETN = '0' then
282         axi_awready <= '0';
283         axi_awv_awr_flag <= '0';
284       else
285         if (axi_awready = '0' and S_AXI_AWVALID = '1' and
               axi_awv_awr_flag = '0' and axi_arv_arr_flag = '0') then
286           -- slave is ready to accept an address and
287           -- associated control signals
288           axi_awv_awr_flag  <= '1'; -- used for generation of
                 bresp() and bvalid
289           axi_awready <= '1';
290         elsif (S_AXI_WLAST = '1' and axi_wready = '1') then
291         -- preparing to accept next address after current write
               burst tx completion
292           axi_awv_awr_flag  <= '0';
293         else
294           axi_awready <= '0';
295         end if;
296       end if;
297     end if;
298   end process;
299   -- Implement axi_awaddr latching
300
301   -- This process is used to latch the address when both
302   -- S_AXI_AWVALID and S_AXI_WVALID are valid.
303
304   process (S_AXI_ACLK)
305   begin
306     if rising_edge(S_AXI_ACLK) then
307       if S_AXI_ARESETN = '0' then
308         axi_awaddr <= (others => '0');
309         axi_awburst <= (others => '0');
310         axi_awlen <= (others => '0');
311         axi_awlen_cntr <= (others => '0');
312       else
313         if (axi_awready = '0' and S_AXI_AWVALID = '1' and
               axi_awv_awr_flag = '0') then
314         -- address latching
315           axi_awaddr <= S_AXI_AWADDR(C_S_AXI_ADDR_WIDTH - 1 downto
                 0);  ---- start address of transfer
316           axi_awlen_cntr <= (others => '0');
317           axi_awburst <= S_AXI_AWBURST;
```

```
318           axi_awlen <= S_AXI_AWLEN;
319         elsif((axi_awlen_cntr <= axi_awlen) and axi_wready = '1'
               and S_AXI_WVALID = '1') then
320           axi_awlen_cntr <= std_logic_vector
                 (unsigned(axi_awlen_cntr) + 1);
321
322           case (axi_awburst) is
323             when "00" => -- fixed burst
324               -- The write address for all the beats in the
                     transaction are fixed
325               axi_awaddr     <= axi_awaddr;       ----for awsize =
                     4 bytes (010)
326             when "01" => --incremental burst
327               -- The write address for all the beats in the
                     transaction are increments by awsize
328               axi_awaddr(C_S_AXI_ADDR_WIDTH - 1 downto ADDR_LSB)
                     <= std_logic_vector
                     (unsigned(axi_awaddr(C_S_AXI_ADDR_WIDTH - 1
                     downto ADDR_LSB)) + 1);--awaddr aligned to 4 byte
                     boundary
329               axi_awaddr(ADDR_LSB-1 downto 0)  <= (others => '0');
                      ----for awsize = 4 bytes (010)
330             when "10" => --Wrapping burst
331               -- The write address wraps when the address reaches
                     wrap boundary
332               if (aw_wrap_en = '1') then
333                 axi_awaddr <= std_logic_vector
                       (unsigned(axi_awaddr) -
                       (to_unsigned(aw_wrap_size,C_S_AXI_ADDR_WIDTH)));
334               else
335                 axi_awaddr(C_S_AXI_ADDR_WIDTH - 1 downto ADDR_LSB)
                       <= std_logic_vector
                       (unsigned(axi_awaddr(C_S_AXI_ADDR_WIDTH - 1
                       downto ADDR_LSB)) + 1);--awaddr aligned to 4
                       byte boundary
336                 axi_awaddr(ADDR_LSB-1 downto 0)  <= (others =>
                       '0');  ----for awsize = 4 bytes (010)
337               end if;
338             when others => --reserved (incremental burst for
                    example)
339               axi_awaddr(C_S_AXI_ADDR_WIDTH - 1 downto ADDR_LSB)
                     <= std_logic_vector
                     (unsigned(axi_awaddr(C_S_AXI_ADDR_WIDTH - 1
                     downto ADDR_LSB)) + 1);--for awsize = 4 bytes
                     (010)
340               axi_awaddr(ADDR_LSB-1 downto 0)  <= (others => '0');
341           end case;
342         end if;
343       end if;
344     end if;
345   end process;
346   -- Implement axi_wready generation
```

```
347
348   -- axi_wready is asserted for one S_AXI_ACLK clock cycle when
          both
349   -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
350   -- de-asserted when reset is low.
351
352   process (S_AXI_ACLK)
353   begin
354     if rising_edge(S_AXI_ACLK) then
355       if S_AXI_ARESETN = '0' then
356         axi_wready <= '0';
357       else
358         if (axi_wready = '0' and S_AXI_WVALID = '1' and
                axi_awv_awr_flag = '1') then
359           axi_wready <= '1';
360           -- elsif (axi_awv_awr_flag = '0') then
361         elsif (S_AXI_WLAST = '1' and axi_wready = '1') then
362
363           axi_wready <= '0';
364         end if;
365       end if;
366     end if;
367   end process;
368   -- Implement write response logic generation
369
370   -- The write response and response valid signals are asserted by
          the slave
371   -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID
          are asserted.
372   -- This marks the acceptance of address and indicates the status
          of
373   -- write transaction.
374
375   process (S_AXI_ACLK)
376   begin
377     if rising_edge(S_AXI_ACLK) then
378       if S_AXI_ARESETN = '0' then
379         axi_bvalid  <= '0';
380         axi_bresp   <= "00"; --need to work more on the responses
381         axi_buser <= (others => '0');
382       else
383         if (axi_awv_awr_flag = '1' and axi_wready = '1' and
                S_AXI_WVALID = '1' and axi_bvalid = '0' and S_AXI_WLAST
                = '1' ) then
384           axi_bvalid <= '1';
385           axi_bresp  <= "00";
386         elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
387         --check if bready is asserted while bvalid is high)
388           axi_bvalid <= '0';
389         end if;
390       end if;
391     end if;
```

```
392   end process;
393   -- Implement axi_arready generation
394
395   -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
396   -- S_AXI_ARVALID is asserted. axi_awready is
397   -- de-asserted when reset (active low) is asserted.
398   -- The read address is also latched when S_AXI_ARVALID is
399   -- asserted. axi_araddr is reset to zero on reset assertion.
400
401   process (S_AXI_ACLK)
402   begin
403     if rising_edge(S_AXI_ACLK) then
404       if S_AXI_ARESETN = '0' then
405         axi_arready <= '0';
406         axi_arv_arr_flag <= '0';
407       else
408         if (axi_arready = '0' and S_AXI_ARVALID = '1' and
                axi_awv_awr_flag = '0' and axi_arv_arr_flag = '0') then
409           axi_arready <= '1';
410           axi_arv_arr_flag <= '1';
411         elsif (axi_rvalid = '1' and S_AXI_RREADY = '1' and
                (axi_arlen_cntr = axi_arlen)) then
412         -- preparing to accept next address after current read
                completion
413           axi_arv_arr_flag <= '0';
414         else
415           axi_arready <= '0';
416         end if;
417       end if;
418     end if;
419   end process;
420   -- Implement axi_araddr latching
421
422   --This process is used to latch the address when both
423   --S_AXI_ARVALID and S_AXI_RVALID are valid.
424   process (S_AXI_ACLK)
425   begin
426     if rising_edge(S_AXI_ACLK) then
427       if S_AXI_ARESETN = '0' then
428         axi_araddr <= (others => '0');
429         axi_arburst <= (others => '0');
430         axi_arlen <= (others => '0');
431         axi_arlen_cntr <= (others => '0');
432         axi_rlast <= '0';
433         axi_ruser <= (others => '0');
434       else
435         if (axi_arready = '0' and S_AXI_ARVALID = '1' and
                axi_arv_arr_flag = '0') then
436           -- address latching
437           axi_araddr <= S_AXI_ARADDR(C_S_AXI_ADDR_WIDTH - 1 downto
                0); ---- start address of transfer
438           axi_arlen_cntr <= (others => '0');
```

```
439             axi_rlast <= '0';
440             axi_arburst <= S_AXI_ARBURST;
441             axi_arlen <= S_AXI_ARLEN;
442           elsif((axi_arlen_cntr <= axi_arlen) and axi_rvalid = '1'
                 and S_AXI_RREADY = '1') then
443             axi_arlen_cntr <= std_logic_vector
                   (unsigned(axi_arlen_cntr) + 1);
444             axi_rlast <= '0';
445
446             case (axi_arburst) is
447               when "00" =>  -- fixed burst
448                 -- The read address for all the beats in the
                       transaction are fixed
449                 axi_araddr     <= axi_araddr;      ----for arsize =
                       4 bytes (010)
450               when "01" =>  --incremental burst
451                 -- The read address for all the beats in the
                       transaction are increments by awsize
452                 axi_araddr(C_S_AXI_ADDR_WIDTH - 1 downto ADDR_LSB)
                       <= std_logic_vector
                       (unsigned(axi_araddr(C_S_AXI_ADDR_WIDTH - 1
                       downto ADDR_LSB)) + 1); --araddr aligned to 4
                       byte boundary
453                 axi_araddr(ADDR_LSB-1 downto 0)  <= (others => '0');
                        ----for awsize = 4 bytes (010)
454               when "10" =>  --Wrapping burst
455                 -- The read address wraps when the address reaches
                       wrap boundary
456                 if (ar_wrap_en = '1') then
457                   axi_araddr <= std_logic_vector
                         (unsigned(axi_araddr) -
                         (to_unsigned(ar_wrap_size,C_S_AXI_ADDR_WIDTH)));
458                 else
459                   axi_araddr(C_S_AXI_ADDR_WIDTH - 1 downto ADDR_LSB)
                         <= std_logic_vector
                         (unsigned(axi_araddr(C_S_AXI_ADDR_WIDTH - 1
                         downto ADDR_LSB)) + 1); --araddr aligned to 4
                         byte boundary
460                   axi_araddr(ADDR_LSB-1 downto 0)  <= (others =>
                         '0');  ----for awsize = 4 bytes (010)
461                 end if;
462               when others => --reserved (incremental burst for
                     example)
463                 axi_araddr(C_S_AXI_ADDR_WIDTH - 1 downto ADDR_LSB)
                       <= std_logic_vector
                       (unsigned(axi_araddr(C_S_AXI_ADDR_WIDTH - 1
                       downto ADDR_LSB)) + 1);--for arsize = 4 bytes
                       (010)
464           axi_araddr(ADDR_LSB-1 downto 0)  <= (others => '0');
465             end case;
466           elsif((axi_arlen_cntr = axi_arlen) and axi_rlast = '0' and
                 axi_arv_arr_flag = '1') then
```

```
467           axi_rlast <= '1';
468         elsif (S_AXI_RREADY = '1') then
469           axi_rlast <= '0';
470         end if;
471       end if;
472     end if;
473   end  process;
474   -- Implement axi_arvalid generation
475
476   -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when
      both
477   -- S_AXI_ARVALID and axi_arready are asserted. The slave
      registers
478   -- data are available on the axi_rdata bus at this instance. The
479   -- assertion of axi_rvalid marks the validity of read data on the
480   -- bus and axi_rresp indicates the status of read
      transaction.axi_rvalid
481   -- is deasserted on reset (active low). axi_rresp and axi_rdata
      are
482   -- cleared to zero on reset (active low).
483
484   process (S_AXI_ACLK)
485   begin
486     if rising_edge(S_AXI_ACLK) then
487       if S_AXI_ARESETN = '0' then
488         axi_rvalid <= '0';
489         axi_rresp  <= "00";
490       else
491         if (axi_arv_arr_flag = '1' and axi_rvalid = '0') then
492           axi_rvalid <= '1';
493           axi_rresp  <= "00"; -- 'OKAY' response
494         elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
495           axi_rvalid <= '0';
496         end  if;
497       end if;
498     end if;
499   end  process;
500   -- ------------------------------------------
501   -- -- Example code to access user logic memory region
502   -- ------------------------------------------
503
504   gen_mem_sel: if (USER_NUM_MEM >= 1) generate
505   begin
506     mem_select  <= "1";
507     mem_address <= axi_araddr(ADDR_LSB+OPT_MEM_ADDR_BITS downto
        ADDR_LSB) when axi_arv_arr_flag = '1' else
508                    axi_awaddr(ADDR_LSB+OPT_MEM_ADDR_BITS downto
                       ADDR_LSB) when axi_awv_awr_flag = '1' else
509                    (others => '0');
510   end generate gen_mem_sel;
511
512   -- implement Block RAM(s)
```

```
513   BRAM_GEN : for i in 0 to USER_NUM_MEM-1 generate
514     signal mem_rden : std_logic;
515     signal mem_wren : std_logic;
516   begin
517     mem_wren <= axi_wready and S_AXI_WVALID ;
518     mem_rden <= axi_arv_arr_flag ;
519
520    BYTE_BRAM_GEN : for mem_byte_index in 0 to
          (C_S_AXI_DATA_WIDTH/32-1) generate
521      signal byte_ram : BYTE_RAM_TYPE;
522      signal word_ram : WORD_RAM_TYPE;
523      signal data_in  : std_logic_vector(C_S_AXI_DATA_WIDTH-1
           downto 0);
524      signal data_out : std_logic_vector(C_S_AXI_DATA_WIDTH-1
           downto 0);
525    begin
526      --assigning 8 bit data
527      data_in  <= S_AXI_WDATA;
528      data_out <= word_ram(to_integer(unsigned(mem_address)));
529      BYTE_RAM_PROC : process( S_AXI_ACLK, sready) is
530      begin
531        if ( rising_edge (S_AXI_ACLK) ) then
532          sstart <= '0';
533          if ( mem_wren = '1' ) then
534            word_ram(to_integer(unsigned(mem_address))) <= data_in;
535            top_input_array(to_integer(unsigned(mem_address))) <=
                 data_in;
536            if mem_address= "0011" then
537              sstart <= '1';
538            end if;
539          end if;
540          if sready='1' then
541                      word_ram(4)<=s_INssaida(0)&"00000";
542                      word_ram(5)<=s_INssaida(1)&"00000";
543                      word_ram(6)<=s_INssaida(2)&"00000";
544            end if;
545        end if;
546
547      end process BYTE_RAM_PROC;
548      process( S_AXI_ACLK ) is
549        begin
550          if ( rising_edge (S_AXI_ACLK) ) then
551            if ( mem_rden = '1') then
552              mem_data_out(i) <= data_out;
553            end if;
554          end if;
555      end process;
556
557    end generate BYTE_BRAM_GEN;
558
559   end generate BRAM_GEN;
560   --Output register or memory read data
```

```
561
562   process(mem_data_out, axi_rvalid ) is
563   begin
564     if (axi_rvalid = '1') then
565       -- When there is a valid read address (S_AXI_ARVALID) with
566       -- acceptance of read address by the slave (axi_arready),
567       -- output the read dada
568       axi_rdata <= mem_data_out(0);  -- memory range 0 read data
569     else
570       axi_rdata <= (others => '0');
571     end if;
572   end process;
573
574
575         sreset<= not S_AXI_ARESETN;
576          s_OUTsx<=(top_input_array(0)(31 downto
                5),top_input_array(1)(31 downto
                5),top_input_array(2)(31 downto
                5),top_input_array(3)(31 downto 5));
577         uut: MLP443 port map(
578             CLK=>S_AXI_ACLK,
579             RESET=>sreset,
580             START=>sstart,
581             READY=>sready,
582             X=>s_OUTsx,
583             SAIDA=>s_INssaida);
584
585 end arch_imp;
```

Código A.15 – Código utilizado para validar o AXI4-Lite encapsulando o perceptron.

```
1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "xil_types.h"
5  #include "xstatus.h"
6  #include "xbasic_types.h"
7  #include "xil_io.h"
8  #include "xparameters.h"
9
10
11 union
12 {
13     Xuint8 u8[4];
14     Xuint32 u32;
15     Xfloat32 f32;
16 }unT;
17
18 int main()
19 {
20
21
```

```
22  u32 baseaddr;
23  int res_int, Whole, Thousands;
24  float res_float;
25
26    init_platform();
27
28    baseaddr = XPAR_TOP27BAXILITE_AXI_0_BASEADDR;
29
30    xil_printf("Enviando entradas.\n\r");
31
32    unT.f32 = 1.0;
33    Xil_Out32(baseaddr+0,unT.u32);
34
35    unT.f32 = 2.0;
36    Xil_Out32(baseaddr+4,unT.u32);
37
38    unT.f32 = 3.0;
39    Xil_Out32(baseaddr+8,unT.u32);
40
41    unT.f32 = 4.0;
42    Xil_Out32(baseaddr+12,unT.u32);
43
44
45    unT.f32 = 1.5;
46    Xil_Out32(baseaddr+16,unT.u32);
47
48    unT.f32 = 1.0;
49    Xil_Out32(baseaddr+20,unT.u32);
50
51    unT.f32 = 2.2;
52    Xil_Out32(baseaddr+24,unT.u32);
53
54    unT.f32 = 3.0;
55    Xil_Out32(baseaddr+28,unT.u32);
56
57    xil_printf("Lendo saida.");
58
59    unT.u32=Xil_In32(baseaddr+32);
60    res_int=unT.u32;
61            res_float = unT.f32;
62            Whole = res_float;
63            Thousands = (res_float - Whole)*100000;
64            xil_printf("saida neuronio= %x\n\r",res_int);
65
66    xil_printf("Enviando entradas.\n\r");
67
68        unT.f32 = 1.0;
69        Xil_Out32(baseaddr+0,unT.u32);
70
71        unT.f32 = 2.0;
72        Xil_Out32(baseaddr+4,unT.u32);
73
```

```
74          unT.f32 = 3.0;
75          Xil_Out32(baseaddr+8,unT.u32);
76
77          unT.f32 = 4.0;
78          Xil_Out32(baseaddr+12,unT.u32);
79
80
81          unT.f32 = 1.5;
82          Xil_Out32(baseaddr+16,unT.u32);
83
84          unT.f32 = 1.0;
85          Xil_Out32(baseaddr+20,unT.u32);
86
87          unT.f32 = 2.2;
88          Xil_Out32(baseaddr+24,unT.u32);
89
90          unT.f32 = 3.5;
91          Xil_Out32(baseaddr+28,unT.u32);
92
93          xil_printf("Lendo saida.\n\r");
94
95          unT.u32=Xil_In32(baseaddr+32);
96          res_int=unT.u32;
97                  res_float = unT.f32;
98                  Whole = res_float;
99                  Thousands = (res_float - Whole)*100000;
100                 xil_printf("saida neuronio= %x\n\r",res_int);
101     cleanup_platform();
102     return 0;
103 }
```

Código A.16 – Código utilizado para validar o AXI4-Full encapsulando a MLP.

```
1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "xil_types.h"
5  #include "xstatus.h"
6  #include "xbasic_types.h"
7  #include "xil_io.h"
8  #include "xparameters.h"
9
10 union
11 {
12     Xuint8 u8[4];
13     Xuint32 u32;
14     Xfloat32 f32;
15 }unT;
16
17 int main()
18 {
19   u32 baseaddr;
```

```
20    int res_int, Whole, Thousands;
21    float res_float;
22      //Xil_DCacheEnable();
23      //Xil_ICacheEnable();
24      init_platform();
25
26      baseaddr = XPAR_MLP443AXIFULL_MM_V1_0_0_BASEADDR;
27
28      xil_printf("Enviando entradas\n\r");
29      //Entradas
30      unT.f32 = 1.0;
31      Xil_Out32(baseaddr+0,unT.u32);
32      unT.f32 = 2.0;
33      Xil_Out32(baseaddr+4,unT.u32);
34      unT.f32 = 3.0;
35      Xil_Out32(baseaddr+8,unT.u32);
36      unT.f32 = 4.0;
37      Xil_Out32(baseaddr+12,unT.u32);
38
39      xil_printf("Calculando Saídas\n\r");
40
41
42      unT.u32=Xil_In32(baseaddr+16);
43      res_int=unT.u32;
44      xil_printf("saida neuronio= %x\n\r",res_int);
45
46      //saidas
47      unT.u32=Xil_In32(baseaddr+20);
48      res_int=unT.u32;
49      xil_printf("saida neuronio= %x\n\r",res_int);
50
51      unT.u32=Xil_In32(baseaddr+24);
52          res_int=unT.u32;
53          xil_printf("saida neuronio= %x\n\r",res_int);
54
55      cleanup_platform();
56      return 0;
57  }
```