DEPARTAMENTO DE ENGENHARIA
ELECTROTÉCNICA E DE COMPUTADORES

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

NUNO ALEXANDRE VIEGAS LEAL

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

NOVA-HDL-ASSIST: EDIÇÃO E CONFIGURAÇÃO DE CÓDIGO VHDL

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Novembro, 2021

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA
ELECTROTÉCNICA E DE COMPUTADORES

NOVA-HDL-ASSIST: EDIÇÃO E CONFIGURAÇÃO DE CÓDIGO VHDL

NUNO ALEXANDRE VIEGAS LEAL

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

Orientador: Filipe de Carvalho Moutinho
Professor Auxiliar, NOVA School of Science and Technology - Universidade NOVA de Lisboa

Júri:

Presidente: Rui Manuel Leitão Tavares
Professor Auxiliar, NOVA School of Science and Technology - Universidade NOVA de Lisboa

Arguente: Rogério Alexandre Botelho Campos Rebelo
Investigador, NOVA School of Science and Technology - Universidade NOVA de Lisboa

Vogal: Filipe de Carvalho Moutinho
Professor Auxiliar, NOVA School of Science and Technology - Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Novembro, 2021

NOVA-HDL-Assist: Edição e Configuração de Código VHDL

Agradecimentos

Ao Professor Filipe Moutinho, pelo seu tempo e disponibilidade em esclarecer, sem falta, as dezenas de dúvidas que foram surgindo. Agradeço ainda, não só, à sua experiência e *soft skills* que guiaram esta dissertação ao sucesso, mas também ao seu compromisso em orientar futuras versões desta ferramenta, podendo um dia ser bastante relevante na sua missão.

Ao Professor Luís Gomes e à Professora Anikó Costa por toda a ajuda que proporcionaram a mim e ao Professor Filipe Moutinho.

Ao Gustavo Murta, aluno da Unidade Curricular Programa de Introdução à Investigação Cientifica (PIIC), por ter ajudado na investigação e por ter testado o *GHDL* e *GTKWave*.

Ao Eduardo Monteiro, aluno de PIIC, pelo desenvolvimento de um conjunto de módulos configuráveis, permitindo validar objectivos inicialmente estipulados para a ferramenta.

Aos meus amigos e colegas de curso, pelo companheirismo inigualável.

À minha família, mas sobretudo aos meus pais, que me proporcionaram todas as condições económicas e afectivas para estudar aquilo que sempre quis.

Resumo

As FPGA (*Field Programmable Gate Array*) e as linguagens de descrição de *hardware*, como VHDL (VHSIC-HDL, *Very High Speed Integrated Circuit Hardware Description Language*), têm uma grande importância no mundo tecnológico e industrial. Contudo, as dificuldades inerentes a estas linguagens têm questionado a comunidade sobre quais as melhores formas de a promover e ensinar, e quais as melhores ferramentas a desenvolver para facilitar a implementação e ensino de circuitos em VHDL.
Face às dificuldades de aprendizagem e uso de linguagens de descrição de *hardware*, como VHDL, este trabalho tem como objectivo o desenvolvimento de uma ferramenta que facilite a aprendizagem e a descrição de *hardware* através de VHDL.
Para tal é proposta uma ferramenta *Web*, denominada NOVA-HDL-Assist, que permite a configuração de módulos e excertos de código VHDL. Esperando-se que o utilizador consiga aprender e implementar os seus projectos mais eficientemente, no menor tempo possível e com um menor número de erros de sintaxe. A ferramenta em questão está disponível em `http://gres.uninova.pt/nova-hdl-assist/`.
O público-alvo desta ferramenta não são apenas alunos ou entusiastas de Engenharia Electrotécnica ou Electrónica que desejem ter um primeiro contacto com linguagens de descrição de *Hardware*, mas também técnicos e engenheiros de outras áreas que desejem realizar projectos em VHDL com autonomia própria, seja no âmbito profissional ou pessoal.

Palavras-chave: VHDL, HDL, Aplicação Web, FPGA, NOVA-HDL-Assist, Edição, Configuração, Aprendizagem

Abstract

FPGAs (Field Programmable Gate Array) and hardware description languages, such as VHDL (VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language), are of great importance in the technological and industrial world. However, the difficulties inherent in this languages have questioned the community about the best ways to promote and teach it, and what are the best tools to develop, to facilitate the implementation and learning of circuits in VHDL.
In view of the difficulties of using hardware description languages, such as VHDL, this work aims to develop a tool that facilitates the description of hardware through VHDL. For this purpose, a Web tool, called NOVA-HDL-Assist, is proposed, which allows the configuration of modules and VHDL code samples. It is hoped that the user can learn, implement their projects more efficiently, in the shortest possible time and with the fewest syntax errors. The tool is available at http://gres.uninova.pt/nova-hdl-assist/.
The target audience of this tool will not only be students or enthusiasts of Electrotechnical or Electronic Engineering who wish to have a first contact with hardware description languages, but also technicians and engineers from other areas who wish to carry out projects in VHDL autonomously, whether in the professional or personal sphere.

Keywords: VHDL, HDL, Application Web, FPGA, NOVA-HDL-Assist, Edition, Configuration, Learning

Índice

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Siglas

ADC	Analog-to-Digital Converter
AJAX	Asynchronous Javascript and XML
ASIC	Application-Specific Integrated Circuit
CAD	Computer Aided Design
CCM	Configurable Computing Machine
CD	Compact Disk
CLB	Configurable Logic Block
DAC	Digital-to-Analog Converter
DDR	Double Data Rate
DoD	Department of Defense
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
JHDL	Just-Another Hardware Description Language
LSB	Least Significant Bit
LSI	Large-Scale Integration
ML	Micro Learning
MSB	Most Significant Bit
MSI	Medium-Scale Integration

PIPO Parallel-in to Parallel-out
PISO Parallel-in to Serial-out
PLD Programmable logic device
PSM Programable Multiplexer

RTL Register Transfer Level

SIPO Serial-in to Parallel-out
SISO Serial-in to Serial-out
SLSI Super-Large-Scale Integration
SoC System-on-a-chip
SSI Small-Scale Integration

ULSI Ultra-Large-Scale Integration

VHDL VHSIC Hardware Description Language
VHSIC Very High Speed Integrated Circuits
VLSI Very-Large-Scale Integration
VS Visual Studios

W3C World Wide Web Consortium

XML Extensible Markup Language
XSLT Extensible Stylesheet Language

Símbolos

N	Número de bits
π	Valor numérico do Pi

1

Introdução

1.1 Enquadramento

Hoje em dia a maioria dos sistemas embutidos contém um ou vários micro-processadores, circuitos integrados, sensores, entre outros. A complexidade dos sistemas tem aumentado e o que era antigamente maioritariamente feito em circuitos analógicos, hoje é também em digital. Para além disso, a pressão do mercado e competitividade obriga a que os produtos sejam tecnicamente superiores e que o ritmo de desenvolvimento dos mesmos esteja em constante aceleração. De modo a satisfazer estas necessidades são criadas camadas de abstracção, satisfazendo assim os engenheiros e técnicos na necessidade de uma maior eficiência. Inicialmente era necessário desenhar circuitos através do elemento mais simples e de baixo nível que são os transístores, estando este conhecimento reservado apenas para uma minoria. Com o desenvolvimento da indústria e do conhecimento foram criadas camadas de abstracção sobre camadas de abstracção, ou seja, por exemplo, passando dos transístores para as portas lógicas, de portas lógicas para módulos, de portas lógicas e módulos para linguagens de descrição de *hardware* como VHDL (VHSIC-HDL, *Very High Speed Integrated Circuit Hardware Description Language*) e Verilog. Linguagens estas usadas com o auxilio de ferramentas de desenvolvimento e teste digitais. Outrora estes circuitos eram implementados manualmente, recorrendo, por exemplo, a *breadboards* [1].

1.2 Motivação

As linguagens de descrição de *hardware*, em comparação com as linguagens de programação, são difíceis e com uma curva de aprendizagem longa. Estas são também um desafio para quem as utiliza esporadicamente, seja para realizar um projecto no âmbito pessoal ou profissional. São prova disso alguns artigos, como [2], que ilustram que após a introdução de alguns métodos, de forma a melhorar a capacidade de aprendizagem e o gosto por linguagens de descrição de *hardware*, a percentagem de alunos que considera a VHDL complicado continua alta. Ao longo dos anos tem havido um crescente interesse na forma

como se deve ensinar as linguagens de descrição de *hardware* e quais as ferramentas a ser desenvolvidas e aplicadas no ensino. A comunidade tem-se debruçado sobre quais são os melhores métodos e ambientes de desenvolvimento (IDE), uma vez que é aceite pelo geral da comunidade que para além da dificuldade inerente a estas linguagens, os ambientes de desenvolvimento mais usados são complexos. Isto é visto como um entrave na configuração e desenvolvimento de circuitos, por exemplo para FPGA, para indivíduos que não dominem esta área, tal como engenheiros informáticos, mecânicos ou estudantes que tenham o seu primeiro contacto com FPGA e linguagens de descrição de *hardware*. Projectos que poderiam ter sido desenvolvidos no âmbito pessoal ou profissional ficam esquecidos, e presumivelmente irão recorrer a outras plataformas mais amigas de utilizadores inexperientes, mesmo não sendo a melhor opção para os projectos que desejem desenvolver. É relevante, por isso, falar brevemente sobre o Arduino. A linguagem de programação para Arduino é uma simplificação de C/C++ [3]. Esta simplificação permitiu a introdução de uma camada de abstracção que tornou o Arduino uma ferramenta útil não só para engenheiros electrotécnicos, mas também para um grande leque de outras áreas, como engenharia mecânica, aeronáutica e informática. Para além disso, deu também a possibilidade de que simples entusiastas pudessem facilmente aprender e realizar os seus projectos autonomamente [4] [5] [6] [7].
Também é importante referir que nas linguagens de descrição de *hardware*, ao contrário de linguagens como o C ou C++, é necessário ter em consideração o circuito digital, uma vez que o que se descreve em VHDL ou Verilog irá reproduzir-se num circuito que, se sem o devido cuidado, será um circuito defeituoso e pouco satisfatório. A adicionar a este problema surge também o facto de linguagens como a VHDL, dadas as suas regras e palavras-chave reservadas, terem uma sintaxe extensa e por vezes complicada, levando a más práticas e um tempo de pesquisa, em livros ou recursos *online*, elevado.

1.3 Objectivos

Face a estes impasses, e à semelhança de [7] o objectivo deste trabalho é desenvolver uma ferramenta que facilite a aprendizagem de VHDL. No entanto, enquanto que em [7] foi proposta uma ferramenta que permite compilar e executar o código VHDL no PC (*Personal Computer*), neste trabalho o objectivo é desenvolver uma ferramenta que simplifique a descrição de *hardware* em VHDL. A finalidade é tornar a edição de VHDL mais simples e intuitiva, de modo a que alunos, técnicos e engenheiros que não dominem linguagens de descrição de *hardware* possam desenvolver os seus projectos de forma simples e rápida. Para tal, a ferramenta irá disponibilizar módulos e excertos de código, que poderão ser configurados, bem como permitir a geração automática de código. Desta forma pretende-se que o utilizador consiga mais facilmente obter exemplos de código, e que os consiga configurar. Assim, poderá ser mais simples a aprendizagem de VHDL e o número de erros de sintaxe e de más práticas de descrição de *hardware* tenderá a diminuir. Nesse sentido as contribuições apresentadas neste documento poderão ser uma mais-valia

na redução do tempo de desenvolvimento de projectos, no ensino e mitigação de erros de sintaxe em VHDL.

1.4 Estrutura do Documento

Este documento é constituído por cinco capítulos. O segundo capítulo descreve o estado da arte sobre o problema em questão com uma descrição de abordagens, conceitos teóricos e de trabalhos de outros investigadores que lidaram com desafios semelhantes. O terceiro capítulo descreve a ferramenta desenvolvida e as suas características. O quarto capítulo é relativo aos testes executados na ferramenta e resultados de um inquérito sobre a utilidade da ferramenta. Por fim, o quinto capítulo refere-se às conclusões e trabalhos futuros.

2

Estado da Arte

Este capitulo tem como objectivo fazer uma apresentação à linguagem de descrição de *hardware* utilizada pela ferramenta desenvolvida. Sendo feitas referências à linguagem em si, aspectos interessantes e características da mesma que vão de encontro às funcionalidades da ferramenta.
Este capítulo é composto por cinco subsecções que pretendem apresentar, não só a linguagem de dexcrição de *hardware* usada, mas também as dificuldades na sua aprendizagem, quais os métodos relativos à sua aprendizagem, algumas considerações sobre as FPGA (*Field-Programmable Gate Array*) e as ferramentas comerciais existentes.

2.1 Descrição da VHDL

A VHDL (*VHSIC-HDL*, *Very High Speed Integrated Circuit Hardware Description Language*) foi desenvolvida na década de 80 pelo Departamento de Defesa dos Estados Unidos (*Department Of Defense*) e pelo IEEE (*Institute of Electrical and Electronics Engineers*) e é uma linguagem bastante poderosa usada para descrever *hardware*, através da especificação do seu comportamento. Ao usar HDL (*Hardware Description Language*), os projectistas irão fazer uma representação da estrutura do circuito ou do comportamento do mesmo, que ao escreverem linhas de código irão alterar o sistema ao nível das portas lógicas, módulos e conexões [8]. O uso de VHDL é uma mais valia no sentido que suporta quatro níveis de abstração, sendo eles, citando [8] e [9]:

- Nível funcional (*functional*), em que o algoritmo é descrito;
- Nível comportamental (*behavioral*). Um modelo *behavioral* descreve as conexões funcionais entre diferentes módulos, permitindo que modelos para simulação possam ser rapidamente construídos;
- RTL (*Register Transfer Level*). Neste nível é possível descrever comportamentos de diferentes formas, como *dataflows*, máquinas de estado síncronas e assíncronas, entre outros;
- O nível lógico que se baseia na álgebra booleana.

Importa desde já referir que o código VHDL pode ser usado, não só para especificar circuitos, mas também para especificar simulações. Contudo nem todo o código pode ser sintetizado e nem todo o código VHDL pode ser simulado.

2.1.1 Bibliotecas e *packages*

Uma biblioteca em VHDL permite que um projectista guarde *packages* e *entities* que são recorrentemente necessários em novos projectos em VHDL. Estes ficheiros previamente analisados estão prontos para serem usados num novo projecto. Um exemplo dessas bibliotecas é a *library IEEE* com o *IEEE.STD_LOGIC_1164.ALL* que permite definir operações para modelos que necessitam de suportar sinais de baixas impedâncias, altas impedâncias, e valores desconhecidos. Esta biblioteca inclui também versões dos operadores lógicos *and*, *nand*, *or*, *nor*, *xor*, *xnor* e *not*. Adicionalmente, permite também o uso de funções para a conversão de valores de diferentes tipos. Para finalizar, há que notar que esta biblioteca disponibiliza duas funções bastante úteis para qualquer projectista, são elas o *rising_edge(data)* e o *falling_edge(data)* que retornam o valor lógico verdadeiro no caso de haver uma transição positiva ou negativa do sinal em *data*, respectivamente. Na situação inversa, é retornado o valor lógico falso [10].

2.1.2 Estrutura do VHDL

Estruturalmente o código VHDL, segundo [11], é constituído por,

- Uma secção destinada a comentários relacionados com o programa, como a data, autor, breve descrição do ficheiro;
- A *package* que é uma biblioteca opcional onde é possível partilhar definições;
- A *entity* que contém os *inputs* e *outputs* do sistema que pretendemos modelar. Estas portas podem ser *in*, *out*, *inout* e *buffer*. Os três primeiros intuitivamente se descobre o seu objetivo, já o *buffer* é considerado uma saída e por isso é bastante similar ao *out* com a diferença que o seu valor pode ser lido dentro da *arquitecture*;
- Na *architecture* deverá ser descrito o comportamento do sistema. Pode-se intuitivamente pensar no *architecture* como um bloco, onde são consideradas as portas declaradas na *entity*, e onde estará o nosso circuito lógico. A *architecture* é constituída por uma parte compreendida entre as palavras-chave *architecture* e *begin* onde deverão ser declarados os objectos pertencentes somente à *architecture* em questão. A secção compreendida entre as palavras-chave *begin* e *end* pode ser denominada de corpo do *architecture* e é onde estará todo o *design* ou, se preferível, circuito lógico.

É importante referir que uma *entity* poderá ter mais do que uma *arquitecture* [11]. Em termos de abstração a VHDL pode ser definido como, *behavioral*, *structural* e *dataflow* [12].

A modelação em *behavioral* é usada tanto para circuitos combinatórios como sequenciais, enquanto que a modelação em *structural* e *dataflow* é usada apenas para circuitos combinatórios. O termo *behavioral* é usado para descrever circuitos complexos. Este tipo de modelação de circuitos, em VHDL, é feito no bloco correspondente ao *architecture*. Cada módulo poderá conter várias declarações de *process*. O modelo *structural* refere-se à agregação de componentes com o objectivo de definir um circuito. Os componentes poderão ir desde módulos até ao uso da lógica mais simples, como portas *AND* e *XOR*. O modelo *dataflow* serve para descrever circuitos combinatórios [13]. Na figura 2.1 está representado um registo de dois *bits*. Usando a terminologia da VHDL, dever-se-á chamar ao módulo *Registo de 2bits*, uma *entity*, bem como declarar as suas portas de *input* e *output*, como a figura 2.1 o demonstra.

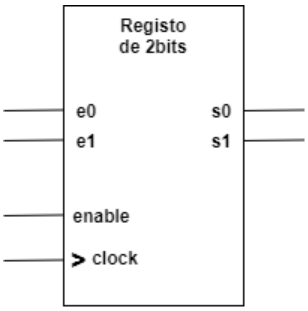

Figura 2.1: Exemplo de um registo de dois *bits*. O registo tem quatro portas de entrada, sendo elas o *e0*, *e1*, *enable* e *clock*. De saída tem as portas *s0* e *s1*.

Portanto, a descrição do módulo em VHDL numa *entity* será,

```
1 entity registodoisbits is
2     port ( e0, e1, enable, clock: in bit;
3            s0, s1: out bit );
4 end entity registodoisbits;
```

Listagem 2.1: Registo de dois *bits*.

Como dito anteriormente e reforçando novamente, este exemplo corresponde à declaração de uma *entity* e, uma vez o código sintetizado, mostra a relação entre o código VHDL e a electrónica digital.

2.1.3 Objetos de dados

Os objetos de dados em VHDL guardam valores. Em VHDL existem três tipos de objetos de dados, sendo eles *variables*,*constants*, e *signals*. De um modo sintético, o *signal* e *variables* podem ser considerados como fios que conectam os diferentes componentes e portas lógicas. O tipo *variable* armazena o seu valor actual e é semelhante a uma variável de outra linguagem de programação, como C, pelo que, para as bibliotecas *STD_LOGIC* e *STD_LOGIC_VECTOR* contém os seguintes tipos pré-definidos [11]:

- *Integer*, que inclui os inteiros ou reais;
- *Boolean*, possui dois valores, *true* e *false*;
- *Character*;
- *Time* (valores possíveis compreendidos entre femtossegundo até horas);
- *Strings*;
- *Bit* (apenas aceita '0' e '1');
- *Bit_vector* (*array* de *bits*).

Os *signals* são declarados da mesma forma que as *variables*. Por outro lado, as *constants*, como o nome indica, mantêm o valor associado. O uso deste tipo de objeto de dados é vantajoso quando se pretende melhorar a leitura do código e reduzir o aparecimento de possíveis erros [11]. Atente-se a alguns exemplos,

- Para especificar, por posição, o conteúdo de um *array* por posição e índices, respectivamente,

```
byteArray := ('0','0','0','1','1','0','1','0');
wordArray:=(0=>'1', 3=>'0', 9=>'1',others=>'1');
```

- Para especificar o conteúdo de um *array STD_LOGIC*,

```
byteArray := "01010101";
wordArray := "0110000010110000";
```

- Para especificar apenas uma parcela de um *array*,

```
byteArray(3 to 5) := "001";
wordArray(9 downto 0) := "1111111010";
```

- Para concatenar *arrays*, pode-se usar o operador &. Note-se o simples exemplo, '3'&'2'&'9' é equivalente a '329'. Já se quisermos fazer um *shift-left* podemos fazer através de, $byteArray(6 \quad down \quad to \quad 0)\&byteArray(7)$. A operação anterior consiste em colocar o *bit* mais significativo na posição 0 do *array* através da concatenação.

Atente-se agora a 2.2, que corresponde à declaração de variáveis dentro de uma *architecture* de uma máquina de estados com três estados.

```
1 type STATE_TYPE is (s0,s1,s2); -- Para definir os estados
2 signal estado_actual, proximo_estado: STATE_TYPE;
```

Listagem 2.2: Exemplo da declaração de variáveis numa máquina de estados com três estados.

2.1.4 Operadores aritméticos, lógicos e relacionais

Os operadores juntamente com uma determinada expressão, executam operações específicas. Os operadores podem-se dividir em aritméticos, lógicos e relacionais [14]. Em 2.1, está presente uma tabela que dá a conhecer alguns operadores.

Operadores		
Operador	Operação	Descrição
+	Adição	Soma os operandos
-	Subtracção	Subtrai os operandos
*	Multiplicação	Multiplica os operandos
/	Divisão	Divide o primeiro operando pelo segundo
and	And	Avalia a condição, sendo verdadeira se ambos os operandos forem verdadeiros
or	Or	Avalia a condição, sendo verdadeira se pelo menos um dos operandos for verdadeiro
xor	Xor	Avalia a condição, sendo verdadeira se pelo menos um dos operandos for verdadeiro, mas não ambos
nand	Nand	Avalia a condição, sendo verdadeira se ambos os operandos não forem verdadeiros
nor	Nor	Avalia a condição, sendo verdadeira se ambos os operandos forem falsos
xnor	Xnor	Avalia a condição, sendo verdadeira se ambos os operandos são, ao mesmo tempo, falsos ou verdadeiros
<	Menor do que	Verdadeiro se o primeiro operando for menor do que o segundo
>	Maior do que	Verdadeiro se o primeiro operando for maior do que o segundo
=	Igual a	Verdadeiro se o primeiro operando for igual segundo
/=	Diferente de	Verdadeiro se o primeiro operando for diferente do segundo
<=	Igual ou menor do que	Verdadeiro se o primeiro operando for igual ou menor do que o segundo
>=	Igual ou maior do que	Verdadeiro se o primeiro operando for igual ou maior do que o segundo

Tabela 2.1: Tabela com alguns dos operadores lógicos, relacionais e aritméticos. Adaptado de [14].

2.1.5 Instruções sequenciais

2.1.5.1 Declaração de *if*

Em bastantes arquitecturas, o comportamento está dependente de diversas condições, pelo que o uso do *if* se torna indispensável [15]. A declaração e implementação são feitas da forma a listagem 2.3 sugere.

```
1 if [condiçao_a_avaliar] then
2     [execuçao]
3 elsif [condiçao_a_avaliar] then
4     [execuçao]
5 else
6     [execuçao]
7 end if;
```

Listagem 2.3: Instrução *If*.

A forma de processamento do *if* é bastante semelhante às linguagens de programação, pelo que não levantará problemas mesmo a iniciantes. A condição após a palavra-chave *if* é avaliada. Caso o valor da condição seja verdadeiro, será executado o código seguinte, caso a condição seja falsa, irá passar para a avaliação da condição seguinte. Em muitos modelos é usada a palavra-chave *else* para realizar uma execução definida pelo projectista quando todas as condições falham [15]. Atente-se na listagem 2.4, que poderá aplicar-se à transmissão de mensagens via rádio de um componente electrónico.

```
1  if temSinalGSM then
2      if recetorNaoOcupado then
3          enviaMensagem <= true;
4      else
5          enviaMensagem <= false;
6          tentarReenvio <= true; -- Volta a tentar passado uns segundos
7      end if;
8  else
9     aguardarPorSinalGSM <= true; -- Para avaliar estado do sinal
10 end if;
```

Listagem 2.4: Exemplo de um possível excerto de descrição em VHDL sobre a transmissão de mensagens via rádio.

Neste exemplo, a condição *temSinalGSM* é avaliada, caso haja sinal *GSM* e o recetor não estiver ocupado a receber informação, o dispositivo transmissor irá enviar a sua mensagem. Caso o recetor esteja ocupado, cancela o envio da mensagem e aguarda um espaço de tempo para o reenvio, através da variável *tentarReenvio*. Por fim, caso não haja sinal *GSM*, o dispositivo irá aguardar que a ligação seja restabelecida [15].

2.1.5.2 Declaração do *when*

O *when* é semelhante ao *if*, a grande diferença assenta que, enquanto o *if* pode avaliar várias condições em simultâneo, o *when* avalia apenas uma. As vantagens e desvantagens de uma em relação à outra são várias, pelo que caberá ao projectista saber qual usar em cada situação para se conseguir um código mais simples, rápido e intuitivo [15]. A sintaxe do *when* está descrita na 2.5.

```
1 [nome_objeto] := [expressao_a_executar] when [condiçao]
2                  else [expressao_a_executar] when [condiçao]
3                  else [expressao_a_executar];
```

Listagem 2.5: Instrução *when*.

Um exemplo em VHDL poderá ser o que está apresentado na listagem 2.6.

```
1 enviaMensagem <= true when recetorNaoOcupado
2                  else false;
```

Listagem 2.6: Exemplo do uso da instrução *when*.

2.1.5.3 Declaração do *case*

O *case* é mais uma instrução que é importante conhecer. A sintaxe está descrita na listagem 2.7.

```
1 case [tipo_de_dados] is
2     when [opçao1] => [execuçao]
3     when [opçao2] => [execuçao]
4 end case;
```

Listagem 2.7: Instrução *case*.

Considere-se agora, um exemplo prático dentro da linha do que tem sido referido. Imagine-se que se pretende modelar uma unidade de controlo de frequências, com um sinal de entrada, *comDisp*, declarado para ser do tipo de *enumeration*, à semelhança do código na listagem 2.8.

```
1 type comDisp is (frequencia1, frequencia2, frequencia3);
```

Listagem 2.8: Exemplo do uso de *enumeration*.

Então, usando o *case* na listagem 2.9, para descrever o comportamento,

```
1 case valDispositivo is
2     when frequencia1 =>
3         dispAEnviar := identificadorDisp1;
4     when frequencia2 =>
5         dispAEnviar := identificadorDisp2;
6     when frequencia3 =>
```

```
7         dispAEnviar := identificadorDisp3;
8 end case;
```

Listagem 2.9: Exemplo do uso da instrução *case*.

Entre as palavras-chave *case* e *is* está a expressão de seleção. Na listagem 2.9 o valor presente na expressão referida anteriormente indica qual das expressões seguidas pelo *when* deverá ser executada. Existe apenas um valor. Neste caso, o *valDispositivo* contém a frequência que deverá ser usada. Se esse valor for *frequencia1*, a instrução a executar será *dispAEnviar* := *identificadorDisp1*, e por adiante. É fácil de se notar que o *case* é bastante similar ao *if*. A diferença reside na forma como as instruções de execução são escolhidas. Enquanto que no *if* as instruções são escolhidas de acordo com o valor lógico da variável a avaliar, no caso do *case* existe um valor que é comparado com outros valores. O *case* é uma ferramenta poderosa na descrição de *hardware* em VHDL, pelo que o leitor deverá dedicar tempo ao seu estudo. Os erros cometidos pelos projectistas costumam ser comuns [15].

2.1.5.4 Instrução *null*

Por vezes, durante a descrição de modelos é necessário indicar que uma condição não terá qualquer execução. Isto aplica-se a, por exemplo a instruções como o *if* e *case*. Importa referir que o uso do *null* em grande parte das situações não é necessário, contudo é uma boa prática usá-lo [15]. O uso desta instrução é bastante simples, atente-se no exemplo da listagem 2.10 adaptado a esta situação,

```
1 case valDispositivo is
2     when frequencia1 =>
3         dispAEnviar := identificadorDisp1;
4     when frequencia2 =>
5         dispAEnviar := identificadorDisp2;
6     when frequencia3 =>
7         null;
8 end case;
```

Listagem 2.10: Exemplo genérico do uso da instrução *case*.

O *null* é também bastante útil caso se queira declarar, por exemplo, um *process*, mas que ainda está por modelar [15].

```
1 recetor: process (frequencia_recebida)
2 begin
3     null; -- Por fazer
4 end process recetor;
```

Listagem 2.11: Exemplo do uso do *null* dentro de um *process*.

2.1.5.5 Instrução *loop*

O *loop* é usado quando se pretende repetir uma instrução, várias vezes em ciclo. O uso desta instrução é bastante útil quando se pretende modelar, por exemplo, um contador [15]. Atente-se no exemplo em 2.12, constituído por um *loop* dentro de um *process*.

```
1 loop
2     wait until utilizadorPrimeBotao;
3     contador := contador + 1;
4 end loop;
```

Listagem 2.12: Exemplo do uso do *loop*.

No exemplo presente em 2.12, as instruções compreendidas entre as palavras-chave *loop* e *end loop* irão ser repetidas eternamente. A primeira instrução, a *wait until utilizadorPrimeBotao*, aguarda que o utilizador prima o botão para adicionar um, pelo que o utilizador terá de premir sempre o botão para se fazer a soma [15].

2.1.5.6 Instrução *exit*

Como visto no exemplo 2.12 da secção 2.1.5.5, o *loop* irá reproduzir-se eternamente. Caso o projectista queira terminar essa reprodução, poderá recorrer à instrução *exit*, através da especificação de uma condição [15]. A sintaxe do *exit* está presente em 2.13.

```
1 exit when [condiçao];
```

Listagem 2.13: Instrução *exit*.

Então, no exemplo 2.12 da secção 2.1.5.5, se se quiser sair do *loop* quando o *contador* atingir o valor inteiro de *15*, atente-se no exemplo 2.14.

```
1 loop
2     wait until utilizadorPrimeBotao;
3     contador := contador + 1;
4     exit when contador = 15;
5 end loop;
```

Listagem 2.14: Exemplo do uso do *loop* com a instrução *exit*.

Para além do exemplo 2.14, o *exit* poderá ter outras intenções, consoante a criatividade e objectivos do projectista. Como é o caso de dois ciclos *loop*, um dentro do outro, na qual se pretende terminar o ciclo exterior recorrendo apenas ao ciclo interior [15].

2.1.5.7 Instrução *next*

O *next* é mais uma instrução de controlo de ciclos. O *next* poderá ser comparado a um *if*, no sentido que apenas irá permitir a execução da instrução seguinte, caso a condição seja verdade [15]. A sintaxe do *next*, com condição, está exemplificada na listagem 2.15.

```
1 next when [condiçao];
```

Listagem 2.15: Instrução *next*.

Um exemplo prático está disponível em 2.16.

```
1 loop
2     next when utilizadorPrimeBotao;
3     contador := contador + 1;
4 end loop;
```

Listagem 2.16: Exemplo do uso do *loop* com a instrução *next*.

Neste caso substituímos o *wait until* pelo *next*. Ou seja, irá realizar a instrução seguinte quando o utilizador premir o botão [15].

2.1.5.8 Instruções *while* e *for*

Nas secções anteriores foram abordados os elementos de controlo de ciclos, contudo em projectos mais complexos e longos o uso massificado destes elementos poderá trazer problemas de legibilidade do projecto. Assim, dever-se-á usar as instruções *while* e *for*, juntamente, caso necessário, com as instruções referidas anteriormente. Posto isto, são apresentados dois códigos em VHDL comparando o ciclo *for* com o *while*. O primeiro código, presente nas listagens 2.17 e 2.18, representa a sintaxe, já o segundo, presente nas listagens 2.19 e 2.20, é o exemplo adaptado que se tem vindo a usar ao longo das secções anteriores.

```
1 for [par metro] in [intervalo] loop
2     [execuçao]
3 end loop;
```

Listagem 2.17: Instrução *for*.

```
1 while [condiçao] loop
2     [execuçao]
3 end loop;
```

Listagem 2.18: Instrução *while*.

```
1 for valor in 0 to 15 loop
2     contador <= valor;
3     wait for utilizadorPrimeBotao;
4 end loop;
```

Listagem 2.19: Exemplo genérico do uso da instrução *for*.

```
1 while valor < 15 loop
2     contador := contador + 1;
3     valor <= contador;
4 end loop;
```

Listagem 2.20: Exemplo genérico do uso da instrução *while*.

Como se pode observar, as duas instruções em questão são semelhantes mas não iguais. O *while* é bastante intuitivo, pelo que este documento deverá focar-se somente no *for*. O ciclo *for* especifica quantas vezes as instruções entre as palavras-chave *loop* e *end loop* deverão ser executadas. No exemplo 2.19 foi usada uma variável e um intervalo de valores. A variável especificada guarda o valor que é incrementado a cada ciclo, como é no caso dos ciclos *for* das linguagens de programação, como C++. Importa salientar, que apesar

de no exemplo se ter usado um intervalo de valores discretos, poder-se-á usar qualquer outro tipo de dados, conforme a criatividade ou necessidade do projectista. Em suma, a inexistência de um *exit* e *next* permite que o ciclo *loop* itere enquanto a condição for verdadeira. Se a condição do *loop* falhar à primeira tentativa, não será realizada qualquer iteração [15].

2.1.6 Subprogramas

O *process* define a sequência de um processo de forma independente, representando o comportamento de uma parte do desenho do sistema a modelar. A sua declaração deverá incluir a palavra-chave *process* seguido de parênteses com os sinais, que declarados para a arquitectura no geral, serão partilhados e usados no *process*. De seguida, de forma opcional, e caso seja necessário, entre as palavras-chave *process* e *begin*, é possível declarar as *variables* que serão apenas usadas no *process* em questão. Importa notar que a declaração de *signals* não é permitida. De seguida, entre as palavras-chave *begin* e *end process* deverá ser descrito o comportamento do módulo em questão [16][17]. Posto isto, a manipulação de dados dentro de um, por exemplo, *process*, é feita através de instruções sequenciais, tais como: *if*, *case*, *loop*, *for* e *while*. Chamam-se instruções sequenciais uma vez que a sua execução é feita sequencialmente [16][17]. De modo ao projectista poder implementar ciclos, a VHDL disponibiliza subprogramas chamados de *functions*, *procedures* e *testebench*. Estas ferramentas devem ser usadas de modo a optimizar o código HDL, através da sua reutilização e facilidade de leitura. Durante a escrita de subprogramas é sempre uma boa prática escolher um nome que indique a operação pela qual é responsável [16][17]. Um *procedure* é útil quando se pretende executar o mesmo código múltiplas vezes em locais diferentes. A sua declaração poderá ser feita como a listagem 2.21 o exemplifica.

```
1 procedure nome_identificador [declaraçao das portas de input e output] is
2     [declaraçao de variáveis]
3 begin
4     [descriçao do comportamento do bloco]
5 end nome_identificador
```

Listagem 2.21: Subprograma *procedure*.

As *functions* são bastante semelhantes aos *procedures* com a diferença que retornam um valor após a execução do código. Citando o documento *"Functions, Procedures, and Testbenches"* da *Xilinx University*, as funções deverão ser usadas quando as seguintes condições se verificam:

- Não há necessidade de recorrer a ferramentas de controlo temporal (*delays*, temporizadores, *events*, entre outros);
- É retornada um único valor;
- Existe pelo menos um argumento de entrada;

- Não existem argumentos de *output* ou *inout*.

A declaração de uma *function* deverá ser feita como a listagem 2.22 o demonstra.

```
function nome_identificador [declaraçao das portas de input e output]
return tipo/variável is
    [declaraçao de variáveis]
begin
    [descriçao do comportamento do bloco]
end nome_identificador
```

Listagem 2.22: Subprograma *function*.

O *testebench* é um modelo escrito em HDL cujo propósito é gerar de forma automática, por exemplo, sinais, que irão verificar o funcionamento do modelo desenhado. Esta ferramenta é especialmente útil, uma vez que para além de facilitar o trabalho do projectista, facilita também a identificação de erros que manualmente pudessem não ser notados [18]. É importante conhecer as instruções relativas à dimensão temporal e controlo de tempo, como é o caso do *after*, *wait*, *wait on*, *wait until* e *wait for*. O uso destas instruções são uma mais valia, no sentido de ser possível aproximar a modelação do circuito ao seu comportamento temporal. De modo oposto, o não uso das mesmas pressupõe que o tempo de simulação é nulo, não havendo dimensão temporal, o que deve ser evitado [18].

2.1.7 Máquinas de Estado

As máquinas de estado são também chamadas de circuitos sequenciais síncronos. É de extrema relevância abordar este tema em VHDL devido à simplificação que esta linguagem poderá trazer ao desenho do circuito, cuja electrónica digital permite inferir que os componentes principais serão, por exemplo, *latches* e *flip-flops*. Existem dois tipos de circuitos síncronos sequenciais, circuito de *Mealy* e o de *Moore*. A diferença entre ambos é que o sinal de saída da máquina de *Moore* depende apenas do estado presente, enquanto que na máquina de *Mealy* depende simultaneamente do estado actual, bem como dos sinais de entrada. Um exemplo em VHDL de um *process* dentro de uma *architecture* que ao receber como sinais de entrada os sinais responsáveis pelo estado actual e o *input* do utilizador deverá ser responsável pela alteração de estado da máquina. Note-se que consoante o valor lógico *0* ou *1* do sinal de *input* a máquina deverá mudar de estado [16].

```
process (estado_atual, input)
begin
    proximo_estado<=estado_atual;       -- mantém o mesmo estado caso as
    case estado_atual is                -- condiçoes falhem
        when s0=>
            if input='1' then           -- input a 1, altera para estado 2
                proximo_estado <= s2;
            end if;
        when s1 =>
            if input='0' then           -- input a 0, altera para estado 2
```

```
11                proximo_estado <= s2;
12            end if;
13        when s2 =>
14            if input='0' then           -- input a 0, altera para estado 0
15                proximo_estado <= s0;
16            else                        -- input a 1, altera para estado 1
17                proximo_estado <= s1;
18            end if;
19    end case;
20 end process;
```

Listagem 2.23: Exemplo de uma máquina de estados.

Para a implementação deste tipo de circuitos é aconselhado fazer-se um diagrama de estados, antes de se começar qualquer descrição em VHDL. Uma vez concluído o diagrama de estados, as ideias estarão mais organizadas, pelo que aplicando os conhecimentos de VHDL, a descrição do hardware por parte do projectista será bastante mais intuitiva e rápida [16]. Na figura 2.2 está presente um exemplo de um diagrama de estados e na listagem 2.23 o seu *process*, correspondente à mudança de estados, em código VHDL.

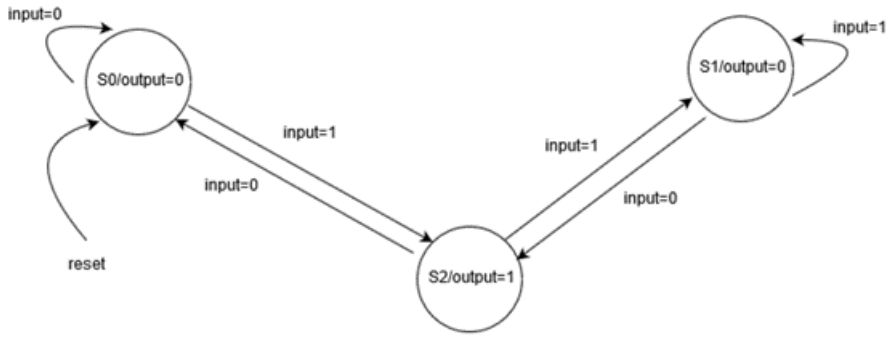

Figura 2.2: Exemplo de um diagrama de estados usado numa máquina de *Moore*.

2.2 Dificuldades na Aprendizagem da VHDL

A VHDL, em geral, é uma linguagem de descrição de *hardware* considerada bastante difícil de aprender. A curva de aprendizagem é mais longa em comparação com outras linguagens, como o C++ ou Java. Em [2] leccionou-se um conjunto de métodos de ensino de VHDL e no final realizou-se um inquérito aos alunos, tendo 83% considerado a VHDL difícil de aprender, contrastando ao mesmo tempo com 70% dos alunos a referirem que gostaram de escrever código em VHDL. Para a realização deste inquérito foram realizadas dez aulas laboratoriais e um trabalho final. As aulas laboratoriais abordaram temas base como:

- Implementação de circuitos combinatórios;

- Aplicação de equações Booleanas, mintermos, maxtermos e mapas de *Karnaugh*;
- Uso de *multiplexers* e *demultiplexers*, implementação de circuitos aritméticos de números de 8-*bits*, e implementação de *shift registers* de 16-*bits*;
- Implementação de máquinas de estado.

Para o trabalho final, era dado aos alunos a opção de escolha de um projecto entre quatro. Todos eles com aplicação directa de circuitos digitais. Na realidade, apesar da dificuldade de aprendizagem da VHDL, é possível enumerar algumas razões que levam a uma curva de aprendizagem mais longa. Antes de tudo, é necessário ter em conta que as ferramentas e plataformas usadas nesta área são só por si complexas, pelo que é necessário um tempo significativo até que o utilizador as domine. Alguns exemplos são os ambientes de desenvolvimento disponibilizados pela *Xilinx*, *Altera/Intel*, e placas de circuitos integrados como as FPGA e componentes como conversores analógico-digitais e memórias DDR (*Double Data Rate*). Para além disso, é da opinião de vários autores que os livros sobre VHDL focam-se em demasia na sintaxe e formalismo da linguagem, secundarizando a relação entre os circuitos digitais e a VHDL [19]. Esta secundarização causa deficiências de aprendizagem, no sentido em que se torna difícil mapear e relacionar a que circuito e implementação de *hardware* corresponde cada excerto de código escrito em VHDL. Este nível de abstracção é preocupante para pessoas pouco experientes, uma vez que apesar da sintaxe ser semelhante às linguagens de programação, o mesmo não se pode dizer em relação ao comportamento do sistema implementado [4].

2.3 Métodos Relativos à Apredizagem de VHDL

Desde a primeira vez que se falou da VHDL, em 1980, que a comunidade se tem preocupado e interrogado sobre quais as melhores práticas para o ensino desta linguagem. Nesta secção são apresentados alguns métodos tradicionais de ensino, bem como alguns métodos que os autores propõem.

2.3.1 Métodos Tradicionais

Os métodos tradicionais são bem conhecidos por toda a comunidade e também por pessoas que não sejam da área, mas que estejam familiarizadas com ela. Estes métodos assentam no uso de um ambiente de desenvolvimento que permita a síntese de VHDL e a sua implementação numa placa de circuito integrado, como uma FPGA. Com estas ferramentas e depois de algumas palestras relativas ao VHDL e circuitos digitais, os alunos são convidados a especificar os mais variados problemas, dependendo dos programas de ensino de cada instituição [6]. Segundo [20], alguns passos tradicionalmente usados para a implementação em VHDL são:

- Definir requisitos e especificações;

- Desenvolver um modelo do sistema que atinja as especificações pretendidas;
- Especificar o modelo em VHDL;
- Simular o modelo especificado em VHDL de modo a determinar se se comporta como especificado inicialmente;
- Sintetizar, implementar e configurar a placa de circuito integrado;
- Verificar através de testes se a placa de circuito integrado se comporta como desejado.

2.3.2 Métodos Alternativos

Dada a dificuldade da linguagem, alguns autores, para além dos métodos tradicionais, propõe também algumas ferramentas que auxiliem a aprendizagem. São várias as propostas documentadas em artigos. Nesta secção serão abordadas apenas algumas.
Em primeiro lugar, segundo [4], a VHDL deverá ser ensinado através de exemplos, por exemplo, poderão ser dados exemplos de circuitos lógicos e qual a sua correspondência ao código VHDL. Para isto foi desenvolvida uma ferramenta chamada *Interactive Learning Toolbox*, ou apenas *Toolbox*. O objectivo desta ferramenta é permitir que os alunos compreendam mais facilmente de que forma um determinado circuito digital corresponde a um modelo VHDL, aliviando assim a deficiência existente nos livros da área. A *Toolbox* resume-se a dois tipos de mapeamento, o *forward mapping* e *backward mapping*, que traduzido, corresponde a mapeamento direto e mapeamento inverso, respetivamente. O mapeamente directo consiste na introdução de conceitos básicos da VHDL, como a sintaxe, estrutura, exemplos, entre outros, cujo objectivo é ilustrar a utilização do código VHDL juntamente com o circuito. O mapeamento inverso, será o oposto do descrito anteriormente. Ou seja, pegando num circuito são ilustradas as possibilidades de código VHDL correspondentes, permitindo assim uma melhor compreensão de como a VHDL afecta a implementação em *hardware*.
Na mesma linha de pensamento, os autores do artigo [5] criaram uma ferramenta *web* com três partes fundamentais. Ensinar aos alunos os conceitos teóricos fundamentais, desenhar e simular o circuito digital, e por fim verificar o comportamento num dispositivo lógico programável (PLD). Esta ferramenta foi desenvolvida em HTML permitindo que seja acedida por todos os alunos a qualquer hora e em qualquer lugar. O acesso remoto é uma mais valia no sentido em que os alunos poderão ter uma aprendizagem personalizada e mais flexível, contrastando com as formas tradicionais. Esta ferramenta é constituída por um tutorial composto por uma parte teórica e prática. A parte teórica é responsável por ensinar a sintaxe da VHDL e introduzir conceitos básicos através de exemplos. A parte prática apresenta um conjunto de exercícios para ilustrar a forma como os circuitos digitais são desenhados através de CAD (*Computer Aided Design*). Existe também um simulador que permite que os estudantes automaticamente criem circuitos a partir do

código VHDL, simulando o comportamento do circuito modelado e implementando-o em *chips* reais.
Os autores de [21] apresentam o *Micro Learning*, ou abreviadamente ML. O objetivo do ML é ensinar os alunos a trabalhar com sub-blocos. Os autores fazem referência a três níveis, V1, V2 e V3, sendo cada nível uma versão melhorada e mais complexa da anterior. O artigo propõe a realização de sub-blocos aumentando progressivamente a dificuldade, onde os três níveis referidos se encaixam. Esses sub-blocos são:

- *Multiplexer*, subdividido na realização de um *multiplexer* de 4, 8 e 16 *bits* em VHDL;
- Instruções aritméticas de 4, 8 e 16 *bits* em VHDL;
- Instruções lógicas de 4, 8 e 16 *bits* em VHDL;
- *Shifting* de 4, 8 e 16 *bits* em VHDL;
- Unidade Aritmética e Lógica (ALU) de 4, 8 e 16 *bits* em VHDL.

Nestes cinco pontos são abordados, sistemas numéricos, circuitos e operações aritméticas, álgebra de boole, mapas de karnaugh, portas lógicas, *shift registers*, meio somador e somador completo, meio subtractor e subtractor completo, codificador e descodificador, e *multiplexer*. De modo a comprovar a efectividade deste método, os autores do artigo num primeiro semestre não usaram ML, e num segundo semestre usaram ML. Em ambos os casos foi pedido aos alunos que preenchessem um inquérito onde se demonstrou que usando ML se obtém melhores resultados em termos de proficiência e efectividade, permitindo a realização de projectos mais complexos num espaço de tempo, que com métodos tradicionais não seria possível.
Os autores de [7] referem que a utilização de métodos visuais, como animações, facilita a aprendizagem face aos métodos tradicionais. O estudo do *hardware* envolve um grande número de conceitos abstractos que tipicamente são difíceis de visualizar para os menos experientes. Exemplo disso é a propagação de sinais digitais dentro de arquitecturas computacionais ou outras unidades funcionais. Estes autores criticam ainda o facto de se usarem ambientes de desenvolvimento comerciais que são demasiado complexos e desnecessários num ambiente que se quer introdutório e onde frequentemente apenas é necessário simular projectos simples em HDL, que muitas vezes contêm apenas um ficheiro. Para solucionar os problemas descritos, os autores de [7] sugerem algumas ferramentas como:

- Uma plataforma *Web* para visualização e animação de processos que consistem em portas lógicas. Inclui esquemas de circuitos combinatórios usando portas lógicas, *multiplexers*, *decoders* e máquinas de estado;
- GHDL, uma aplicação leve e *open-source* que permite que os utilizadores compilem e executem código VHDL diretamente usando apenas a linha de comandos do *Windows*, *Linux* ou *macOS*;

- GTKWave, para a visualização das formas de onda.

Outros autores [22], privilegiando uma atitude de desenvolvimento baseada em modelos, fazem referência a um ambiente de desenvolvimento *Web* chamado IOPT-Tools, disponível em `http://gres.uninova.pt/IOPT-Tools`. Composto por um conjunto de ferramentas, permite modelar sistemas de controlo digital através de redes de Petri. Algumas características interessantes das IOPT-Tools são:

- Edição gráfica interactiva de modelos de controladores. Neste caso houve o uso de tecnologias W3C (*World Wide Web Consortium*), como AJAX (*Synchronous Javascript and* XML (*Extensible Markup Language*)) e XLST (*Extensible Stylesheet Language Transformations*);
- Geração automática de código VHDL, C, etc., que permite, não só uma simplificação da implementação do controlador final ou da lógica inerente aos módulos, eliminando a necessidade de escrever código de baixo nível, mas também minimizar a complexidade e reduzir as dificuldades e a falta de conhecimento nesta área [23];
- Simulação e *model-checking* contribuem para a redução do tempo e custo de desenvolvimento, permitindo também a detecção de erros de modelação em fases precoces do projecto.

Já as IOPT-Flow, referidas pelos autores de [24], à semelhança das IOPT-Tools são também um ambiente de desenvolvimento baseado em ferramentas *Web*. Estas foram propostas para o desenvolvimento de sistemas embutidos e ciberfísicos, com maior ênfase no processamento de dados, quando comparadas com as IOPT-Tools. Como afirmado em [24], os modelos combinam redes de Petri com *dataflows*. Os autores sugerem que esta combinação é uma mais valia no desenho de sistemas que ao mesmo tempo necessitem de máquinas de estado e de processar dados.

2.4 FPGA

Como [25] constata, a indústria de circuitos integrados esteve sempre em constante evolução. O advento dos circuitos integrados revolucionou a indústria de produtos electrónicos, levando ao aparecimento de telemóveis, auscultadores, CD (*Compact Disc*), entre outros. Neste sentido, a pressão do mercado forçou a evolução dos circuitos integrados para uma maior capacidade de produção, menor preço e maior capacidade de processamento, memória, etc. Isto levou a que os circuitos integrados, inicialmente limitados a um pequeno número de transístores, passassem a ser constituídos por milhares e milhares de transístores.
Os primeiros circuitos integrados compostos por menos de 100 transístores num *chip* denominado circuito integrado de pequena escala, ou em inglês, *Small-Scale Circuits* (SSI), surgiram no inicio da década de 1960.

De seguida, o surgimento dos primeiros circuitos integrados de média escala (MSI) em que cada *chip* era composto por várias centenas de transístores, deu-se no final da década de 1960.
Os circuitos de larga escala (LSI), que foram os primeiros circuitos integrados compostos por centenas de milhares de transístores, começaram a ser desenvolvidos a meio da década de 1970.
Esta evolução foi contínua e a capacidade de agregar lógica digital num único *chip* aumentou exponencialmente, pelo que não faltou até que surgissem os circuitos integrados de muito larga escala (VLSI), ultra larga escala (ULSI) e super larga escala (SLSI). Em 2004 um *chip* era composto por 100 000 000 transístores.
Esta quantidade enorme de lógica digital num único *chip* é um problema para o desenho de circuitos digitais, principalmente através dos métodos tradicionais, como o desenho de *flip-flops* e as suas conexões. Daí o surgimento das linguagens de descrição de *hardware*, ou abreviadamente, HDL.

Tipo de Circuito Integrado vs Número de Transístores	
Tipo de Circuito Integrado	Número Mínimo de Transístores segundo [25]
Pequena Escala	10
Média Escala	100
Larga Escala	1000
Muito Larga Escala	10 000
Ultra Larga Escala	100 000
Super Larga Escala	1 000 000

Tabela 2.2: Tabela com o tipo de Circuito Integrado e o Número de Transístores correspondente segundo [25].

2.4.1 Introdução aos ASIC e FPGA

O ASIC (*Application-Specific Integrated Circuit*), como o próprio nome indica, são circuitos cujo objectivo é a realização de tarefas específicas em situações bem definidas. Durante os anos 80 o uso de ASIC generalizou-se no mercado de circuitos integrados e na indústria de semicondutores. Note-se que o circuito implementado num ASIC fica permanentemente desenhado no silicone [26].
O FPGA (*Field Programmable Gate Array*), tal como o ASIC, é constituído por milhares ou milhões de portas lógicas nas quais as conexões entre si são programáveis. A grande vantagem do FPGA face ao ASIC é que o FPGA é reprogramável, sendo bastante útil como protótipo de um ASIC para desenvolvimento, ou caso seja necessário chegar ao mercado rapidamente. O ASIC por outro lado, ao ter um custo de produção por unidade reduzido é preferível ser usado em larga escala. Importa também referir que uma das grandes vantagens do ASIC é o baixo consumo de energia.
A flexibilidade em termos de se poder reprogramar uma FPGA deve-se ao facto de estas

serem constituídas por blocos lógicos configuráveis, ou abreviadamente, CLB (*Configurable Logic Blocks*), blocos I/O (*input/output*) configuráveis e interconexões programáveis. Os CLB contêm a lógica das FPGA, sendo por isso responsáveis pelas operações lógicas a si atribuídas. As interconexões entre os CLB e os blocos I/O são feitos através de canais horizontais, verticais ou PSM (*Programable Multiplexers*). O número de CLB numa FPGA define a sua complexidade, assim como a funcionalidade dos CLB e dos PSM são descritos através de uma linguagem de descrição de *hardware*, como a VHDL. O código de descrição de *hardware* uma vez implementado, os CLB e PSM irão alterar-se no *chip* e conectar-se entre si através das interconexões programáveis, como referido anteriormente [26].

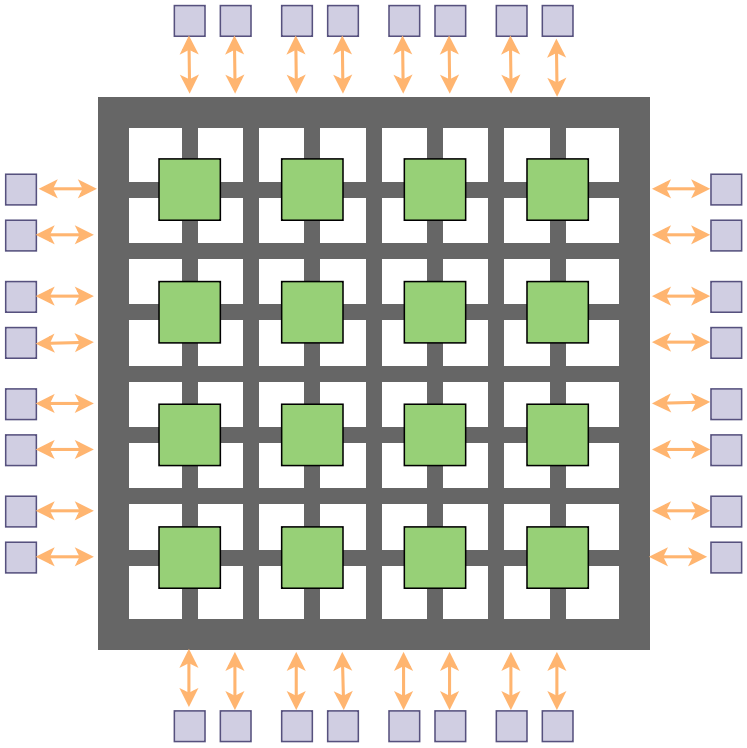

Figura 2.3: Arquitectura da FPGA. Os CLB estão representados a verde, os Blocos I/O a azul e nas restantes cores as conexões. Adaptado de [26].

2.4.2 Uso de HDL em FPGA

A revolução no desenho e concepção de sistemas digitais, ocorrido há algumas décadas, permitiu que os FPGA sejam compostas por milhões de portas lógicas e milhares de *flip-flops*. Isto significa que, dada a complexidade actual dos circuitos, não é possível usar os métodos tradicionais de desenho de circuitos digitais que contém milhares de portas lógicas. Este paradigma levou à introdução de HDL para descrever sistemas digitais. A vantagem das linguagens de descrição de *hardware* é que se irá descrever o comportamento do circuito em vez de escrever as tradicionais equações lógicas Booleanas. Adicionalmente ainda existem ferramentas CAD que servem para simular o que foi descrito em VHDL e

Verilog, e sintetizar a descrição num circuito [25].
A VHDL e Verilog são sem dúvida as linguagens de descrição de *hardware* mais conhecidas e usadas. Contudo existem outras, como é o caso do JHDL (*Just-Another Hardware Description Language*).
O JHDL, como [27] refere, é uma linguagem de descrição de *hardware* que se foca principalmente em modelar circuitos, abordando a orientação a objectos presente noutras linguagens, como o Java. Os recursos de uma FPGA são geridos de uma forma semelhante à forma como as linguagens orientadas a objectos gerem a memória. No JHDL os circuitos são considerados objectos distintos e o circuito é configurado numa *Configurable Computing Machine* (CCM). Os construtores ao serem invocados solicitam uma instância do circuito na plataforma reconfigurável como um objecto alocado na memória, tal e qual como numa linguagem orientada a objetos. Os autores de [27] salientam que o facto de ser possível alternar entre a simulação de *software* ou execução de *hardware* na CCM, permite um ambiente de execução e simulação, sendo uma característica que torna o JHDL com o CCM algo útil e vantajoso.

2.4.3 Exemplos de FPGA Disponíveis no Mercado

Nesta secção pretende-se dar a conhecer algumas das FPGA disponíveis no mercado. Com

FPGA da *Xilinx*			
Modelo	Indústria Automóvel	Indústria de Defesa	Indústria Aeroespacial
XA Artix-7	X		
XA Spartan-7	X		
XQ Artix-7		X	
XQ Spartan-6		X	
Virtex-5QV			X
Virtex-4QV			X

Tabela 2.3: Tabela com algumas FPGA da *Xilinx* e área a que melhor se adequam.

base nas FPGA apresentadas na tabela 2.3 e do site da *xilinx* (`https://www.xilinx.com/products/silicon-devices/fpga.html`) é possível retirar algumas características interessantes, como:

- A *Spartan-6* suporta um grande número de protocolos I/O. É adequada para uma variedade de aplicações avançadas como o *infotainment* e automação industrial;
- A *Virtex UltraScale* fornece um bom desempenho e integração a 20 nm, sendo usada indústria. É adequada para aplicações de rede e emulação de teste em grande escala;
- A *Artix-7* oferece uma boa relação desempenho-por-watt, processamento de sinais digitais, etc. Esta família é adequada para uma variedade de aplicabilidades que necessitem de um baixo consumo de energia como, *Software-Defined Radio*, câmaras

presentes em máquinas e centros de processamento da rede que necessitem de fazer a conexão entre o *backbone* e as sub-redes periféricas.

Um problema complexo em que as FPGA são bastante úteis é no processamento de sinais, onde por vezes são necessários componentes extras como conversores analógico-digital e digital-analógico. A utilidade das FPGA nesta área deve-se ao facto de nas comunicações os processamentos terem de ser rápidos e síncronos com o resto do sistema.

2.5 Ferramentas Comerciais Existentes

No mercado existem vários *softwares* comerciais que auxiliam a descrição de *hardware*. Nesta secção são apresentados alguns exemplos desses *softwares*.
A figura 2.4 corresponde à ferramenta *Vivado* da *Xilinx*. Esta ferramenta juntamente com a *Quartus Prime* da *Intel*, é uma das mais conhecidas.
O *Vivado* e o *Quartus Prime* são semelhantes. Ambos realizam tarefas relacionadas com a síntese e análise de projectos em HDL e implementação em dispositivos lógicos programáveis. Dado que são ferramentas comerciais e têm como publico-alvo profissionais experientes, estas ferramentas são complexas e requerem algum treino [7].

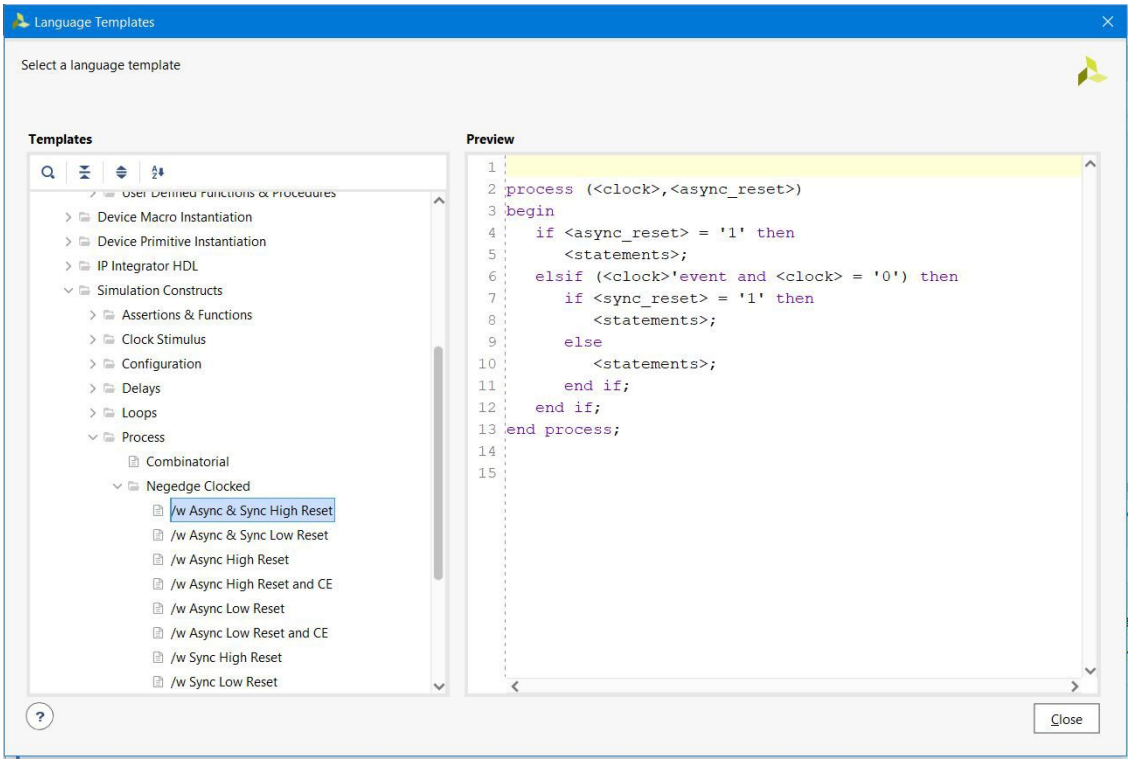

Figura 2.4: Exemplo de um módulo disponibilizado no Vivado pela *Xilinx*.

Nas figuras relativas ao *Vivado* e *Quartus Prime* mostra-se os *templates* disponibilizados pelos *softwares*. Estes *templates* são úteis, umas vez que facilitam e reduzem o tempo de pesquisa aquando na descrição de *hardware*.

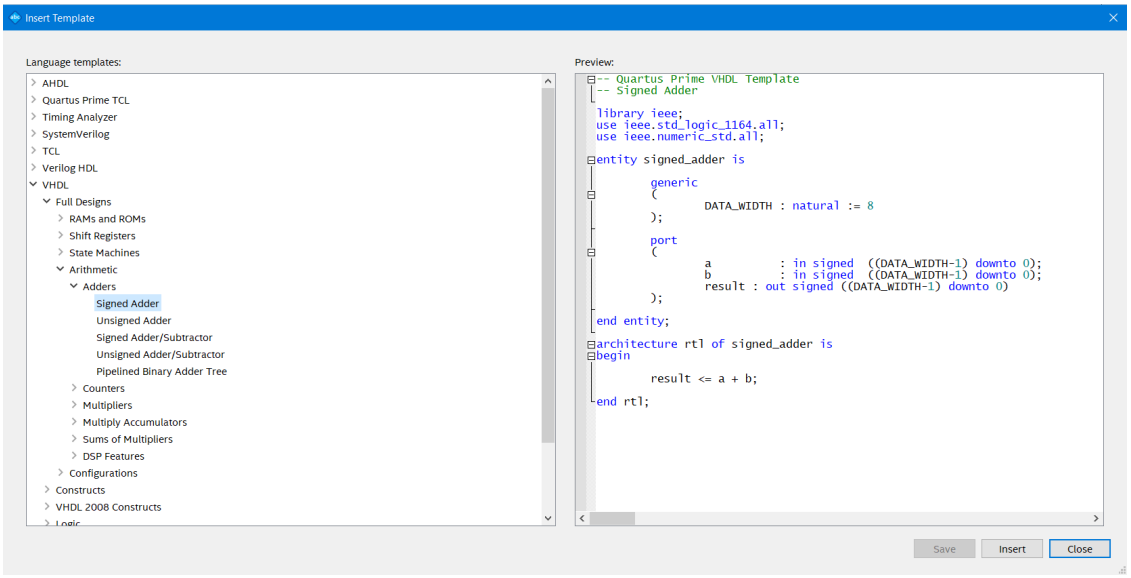

Figura 2.5: Exemplo de um módulo disponibilizado no Quartus pela *Intel*.

De notar que na figura 2.6, relativa ao *Matlab*, estão representados circuitos digitais que podem ser traduzidos para VHDL automaticamente.
A ferramenta *Simulink*, do *Matlab*, é um instrumento versátil para investigadores e engenheiros. Esta inclui um ambiente gráfico interactivo e um conjunto de bibliotecas disponíveis, que podem ser personalizadas, permitindo que o utilizador modele, simule e analise os sinais mais importantes para o seu projecto [28]. Modelos *Simulink*, que correspondem a circuitos digitais, podem ser convertidos para HDL, utilizando o HDL Coder.

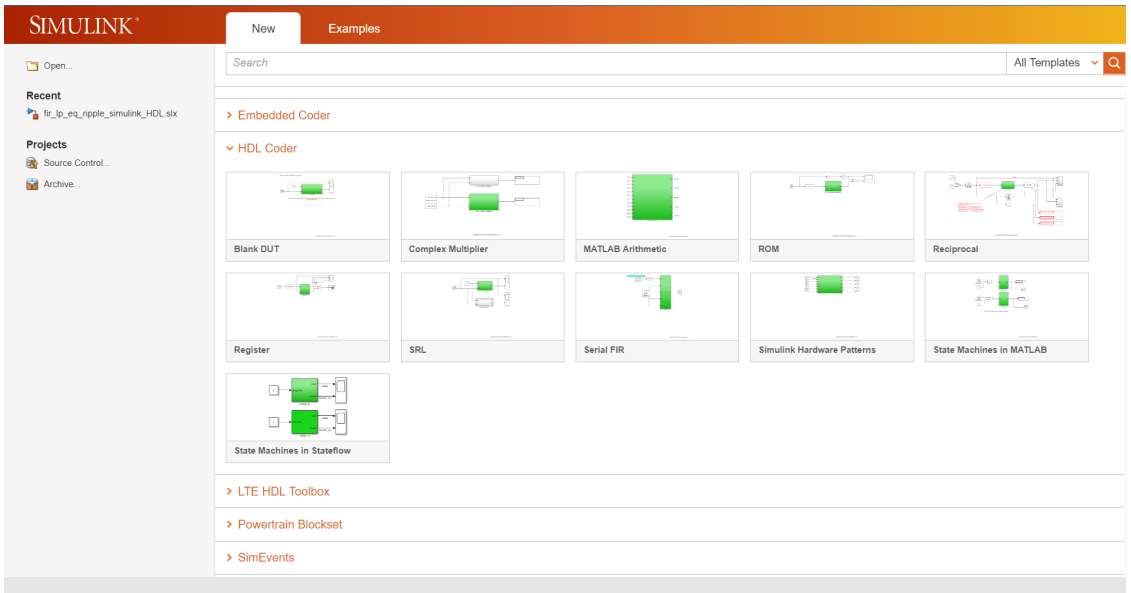

Figura 2.6: Exemplo de alguns módulos disponibilizados pelo *HDL Coder* no *Simulink*, *Matlab*.

O Active-HDL, da *Aldec*, representado na figura 2.7, é um ambiente de desenvolvimento, com uma interface gráfica intuitiva, totalmente integrado para o desenho de circuitos integrados, na qual os circuitos podem ser descritos em código C/C++ ou em HDL (VHDL e Verilog) [28].

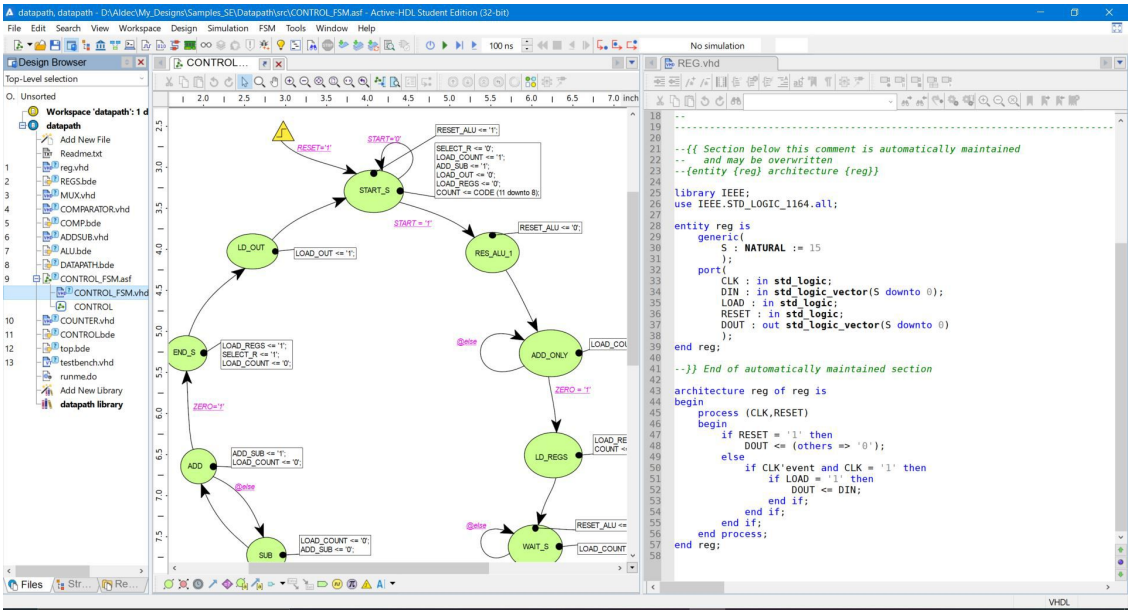

Figura 2.7: Exemplo de um dos módulo disponibilizado pelo *Active-HDL* da *Aldec*. No lado esquerdo está representado um diagrama de estados e no lado direito o respectivo código VHDL auto-gerado.

O Active-HDL, à semelhança das IOPT-Tools e IOPT-Flow, permite a tradução de diagramas de estado em HDL. O Active-HDL permite também a tradução de código HDL em diagramas de blocos e vice-versa.
O Cadence Stratus HLS, que como o *datasheet* da ferramenta refere, permite que rapidamente se desenhe e se verifique implementações em RTL (*Register Transfer Level*) através de modelos descritos em C++, SystemC ou C, automatizando o desenho e verificação [29].
A Plataforma Xilinx PYNQ oferece uma interface de programação baseada em Python. Esta plataforma permite que os projectistas, através da linguagem de programação Python, explorem as capacidades de lógica reconfigurável das placas. O uso de Python, graças à sua natureza e às bibliotecas prontas para serem usadas, é uma vantagem face às HDL e outras formas menos abstractas de desenvolvimento de circuitos reconfiguráveis [30].

3

Descrição da Ferramenta

Ao longo deste capítulo é apresentada a ferramenta proposta, e são discutidas as suas funcionalidades, características e os modelos e módulos em VHDL presentes na ferramenta. Na primeira secção com o auxilio de um diagrama de casos de uso e de um diagrama de sequência, são explicadas as funcionalidades e características gráficas da ferramenta. Na segunda secção são apresentados todos os campos de configuração que podem ser encontrados nos diferentes módulos/modelos presentes na ferramenta. Na terceira secção é feita uma referência à base de dados. Nas secções seguintes são explicadas as tecnologias usadas e é feita uma breve descrição sobre as mesmas e quais os pontos mais importantes da sua implementação. Por fim, na última secção são expostos e explicados os módulos e exemplos de descrições em VHDL oferecidos na ferramenta.

3.1 Funcionalidades

Na ferramenta proposta, denominada *NOVA-HDL-Assist*, são oferecidos um conjunto de módulos, descritos na secção 3.10, bastante comuns em electrónica digital. Estes módulos são gravados na base de dados. O utilizador ao inicializar a página *Web* irá seleccionar um dos módulos. A solicitação dos módulos será processada em tempo real e obtida através de uma comunicação com o servidor, onde está a base de dados. Os módulos serão posteriormente carregados para uma área de texto, como se pode observar na figura 3.1.
O modo de funcionamento da ferramenta está descrito, esquematicamente, na figura 3.2. No lado do servidor está contida a base de dados com os excertos e módulos de VHDL. A inserção de novos excertos ou módulos em VHDL deverá ser feita directamente na base de dados. No lado do cliente está presente a interface gráfica e o processamento da informação submetida pelo utilizador e recebida da base de dados. A informação é processada pelo código em *JavaScript* e de seguida é actualizada na página *Web*.
A interface gráfica da NOVA-HDL-Assist, em traços gerais, com o módulo contador síncrono seleccionado, está representado na figura 3.1.

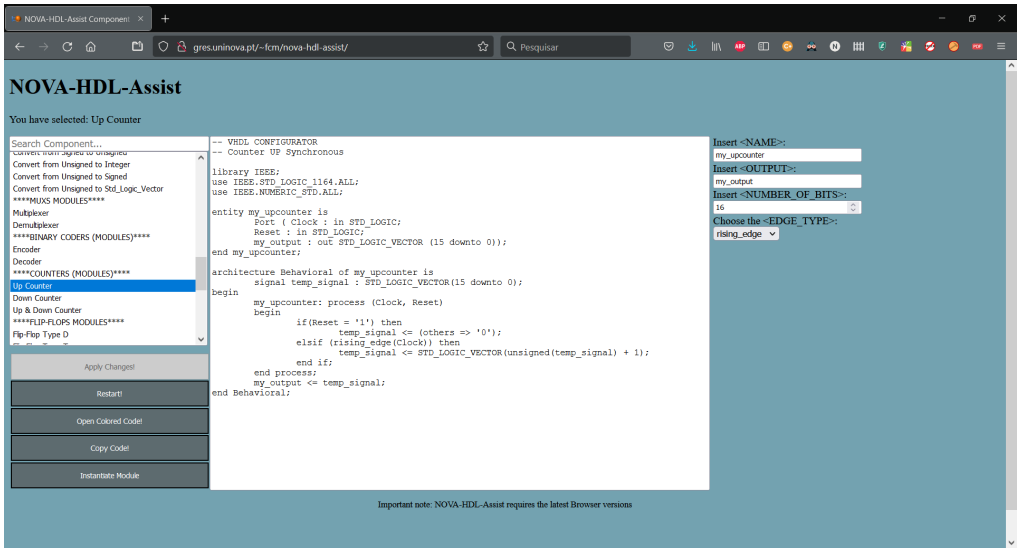

Figura 3.1: Ferramenta em desenvolvimento com o módulo de contador síncrono seleccionado.

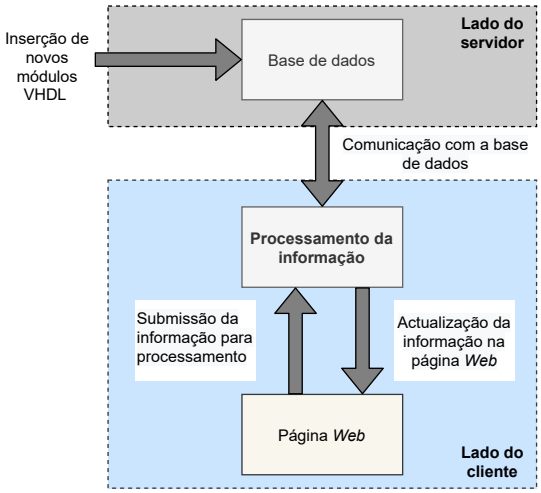

Figura 3.2: Arquitectura da ferramenta [31].

O diagrama de caso uso, presente na figura 3.3, que descreve a interacção entre a NOVA-HDL-Assist e o projectista, tem algumas características interessantes que devem ser analisadas. Imediatamente ao abrir a NOVA-HDL-Assist, o utilizador poderá escolher as opções:

- *Seleccionar Módulo*, que irá carregar o módulo para a área de texto. Esta opção levará a que o utilizador instancie o módulo e/ou configure o módulo. Se o utilizador optar pelo segundo caminho, terá de preencher os elementos auto-gerados pela ferramenta, como as caixas de texto e painéis de selecção;

- *Configurar Módulo*, em que os valores introduzidos pelo utilizador nos campos de configuração e o botão *Apply Changes!* irão causar as alterações desejadas, pelo

utilizador, ao módulo VHDL;

- *Reiniciar página Web*, que tem como finalidade reiniciar a página caso algum *bug* ou erro por resolver impossibilite o seu uso temporariamente;
- *Abrir código a cores*, que irá colorir o código do módulo de acordo com um tema pré-definido e segundo as regras de sintaxe e apresentação da VHDL;
- *Copiar código*, que como o nome diz, irá copiar o código para a área de transferências;
- *Instanciar Módulo*, que irá abrir uma nova página *Web* com o módulo configurado pelo utilizador correctamente instanciado.

Importa também referir que, o «*extend*» refere-se a um comportamento ou característica partilhável entre vários *casos de uso*.

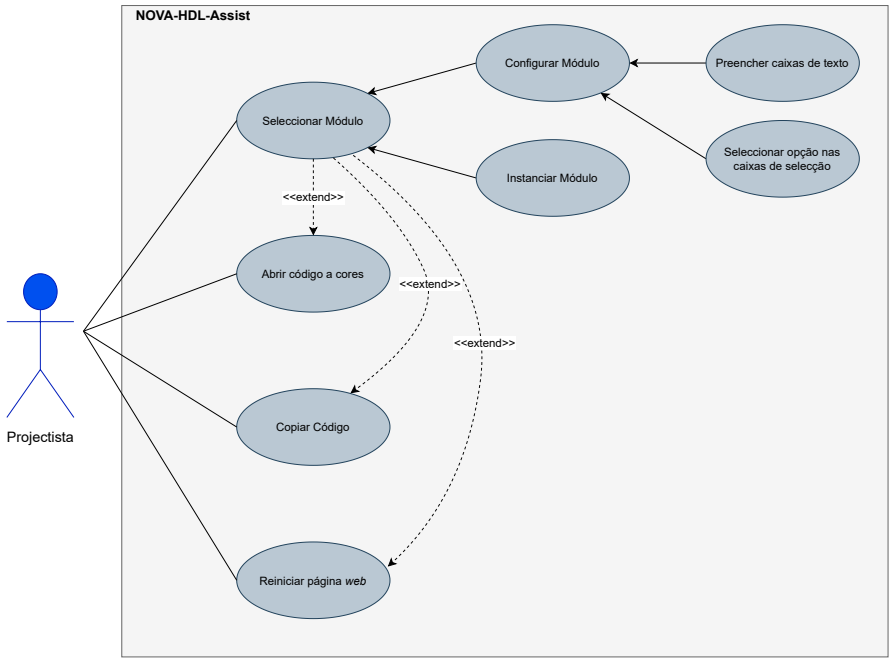

Figura 3.3: Diagrama casos de uso da NOVA-HDL-Assist.

Na figura 3.4 está presente um diagrama de sequência que representa o funcionamento da ferramenta e as interacções entre os seus elementos, para o cenário de configuração de um módulo/modelo de código e instanciação de um módulo.
Como é referido anteriormente e na figura 3.4, a comunicação com a base de dados e a obtenção dos ficheiros necessários é feita através de um pedido *HTML GET* para o *GitHub*.

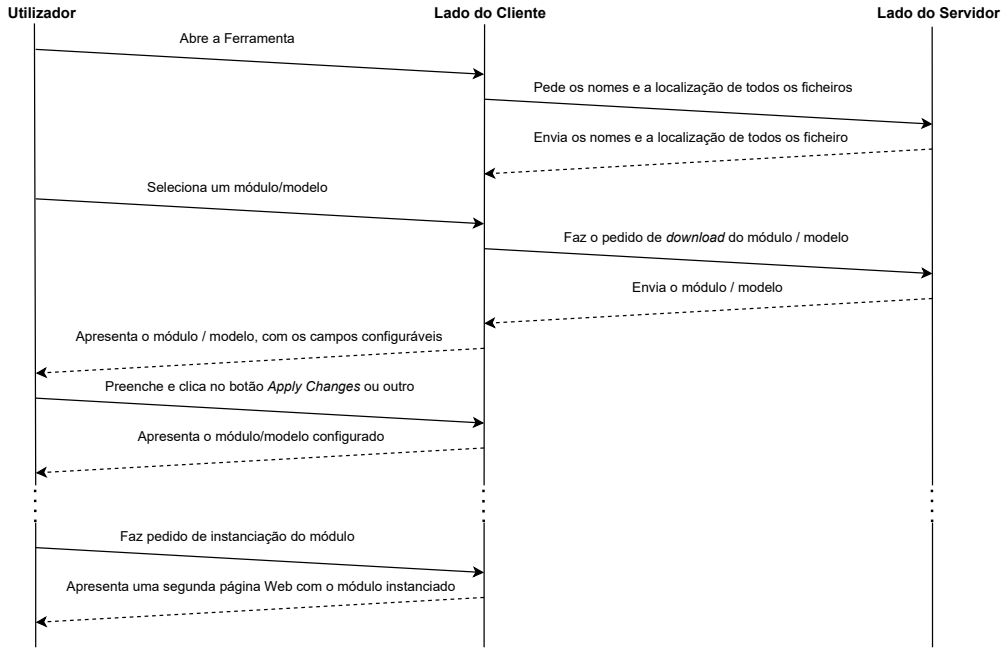

Figura 3.4: Diagrama de sequência do funcionamento da NOVA-HDL-Assist, para a configuração de um módulo/modelo e instanciação de um módulo.

3.2 Campos de Configuração

Os campos configuráveis pelo utilizador encontram-se resumidos na tabela 3.1. Para cada possibilidade de campos e opções de escolha por parte do utilizador, são apresentadas as respectivas modelações em VHDL. Note-se que cada palavra entre "<" e ">" será preenchida de acordo com as escolhas do utilizador. Importa também desde já referir, que sempre que aparecer MSB (*Most Significant Bit*) refere-se ao *bit* mais significativo e LSB (*Least Significant Bit*) refere-se ao *bit* menos significativo. Sempre que aparecer em VHDL, por exemplo, *<MSB>-1*, isto significa que é a posição do *bit* anterior ao MSB.

Tabela 3.1: Tabela com todos os campos de configuração e sua descrição.

Inicio da Tabela	
Campos	Descrição
<ARITHMETIC_DATA_-TYPE>	Por norma, refere-se ao tipo de dados de uma ou várias variáveis (*UNSIGNED* ou *SIGNED*)
<ASSIGNMENT_TYPE>	Por norma, refere se a atribuição deverá ser um sinal ou variável
<BORROW_BIT>	Por norma, refere-se ao nome da variável de saída correspondente ao *bit* de *borrow*

Continuação da Tabela 3.1	
Campos	Descrição
<CARRY_BIT>	Por norma, refere-se ao nome da variável de saída corresponde ao *bit* de *carry*
<CLOCK>	Por norma, refere-se ao nome do *clock* de um módulo
<DATA_TYPE>	Por norma, refere-se ao tipo de dados (*signal*, *variable* ou *constant*)
<DIFFERENCE_BIT>	Por norma, refere-se ao nome da variável de saída corresponde ao *bit* de *difference*
<EDGE_TYPE>	Por norma, refere se o módulo/modelo deverá ser sensível ao flanco ascendente ou descendente de um sinal
<FOSC_HZ>	Por norma, refere-se ao valor da frequência de oscilação do sinal de PWM em *Hertz*, presente no módulo *pulse width modulation*
<FREQ_HZ>	Por norma, refere-se ao valor da frequência do sinal *clock*, do módulo *pulse width modulation* ou do *debounce buttons*, em *Hertz*
<INPUT1>	Por norma, refere-se ao nome da primeira entrada
<INPUT2>	Por norma, refere-se ao nome da outra entrada
<INPUT_BIT1>	Por norma, refere-se ao nome da variável de entrada correspondente ao primeiro *bit*
<INPUT_BIT2>	Por norma, refere-se ao nome da variável de entrada correspondente ao segundo *bit*
<INPUT_BORROW_BIT>	Por norma, refere-se ao nome da variável de entrada correspondente ao *bit* de *borrow*
<NAME>	Por norma, refere-se ao nome que as *entities* deverão ter
<NET_NAME>	Por norma, refere-se ao nome da *Net* num ficheiro UCF (*User Constraint File*) ou XDC (*Xilinx Design Constraints file*)
<NUMBER_OF_BITS>	Por norma, refere-se ao número de *bits* de uma ou mais variáveis dentro de um módulo/modelo
<NUMBER_OF_STATES>	Por norma, refere-se ao múmero de estados que uma máquina de estados (módulo ou processo) deverá ter
<NUMBER_OF_BITS_OPA>	Por norma, refere-se ao número de *bits* do operando A
<NUMBER_OF_BITS_OPB>	Por norma, refere-se ao número de *bits* do operando B
<NUMBER_OF_BITS_RES>	Por norma, refere-se ao número de *bits* do operando correspondente ao resultado

Continuação da Tabela 3.1	
Campos	Descrição
<NUMBER_TO_CONVERT>	Por norma, refere-se ao número, num dado sistema numérico, que deverá ser convertido, para outro sistema numérico
<NUMBER_TO_CONVERT_SYSTEM>	Por norma, refere qual o sistema numérico (Binário, Decimal ou Hexadecimal) do número introduzido pelo utilizador
<OBJECT>	Por norma, refere-se ao nome de um objecto
<OUTPUT>	Por norma, refere-se ao nome da saída
<OPA_SUB_TYPE>	Por norma, refere-se ao tipo do operador A (*STD_LOGIC_VECTOR*, *UNSIGNED* ou *SIGNED*)
<OPB_SUB_TYPE>	Por norma, refere-se ao tipo do operador B (*STD_LOGIC_VECTOR*, *UNSIGNED* ou *SIGNED*)
<OPERATOR>	Por norma, refere-se a um operador lógico (*and*, *nand*, *or*, *nor*, *xor* ou *xnor*) ou relacional (*equal*, *not equal*, *less than*, *greater than*, *equal or less than* ou *equal or greater than*)
<OPERAND_A>	Por norma, refere-se a um operando
<OPERAND_B>	Por norma, refere-se a um operando
<OPERAND1>	Por norma, refere-se a um operando
<OPERAND2>	Por norma, refere-se a um operando
<OPERATION>	Por norma, indica o tipo de operação (adição, subtracção ou multiplicação)
<PERIOD_MS>	Por norma, refere-se ao valor do período do temporizador, em milissegundos
<PWM_PERCENT>	Por norma, refere-se à percentagem de um pulso (*Duty Cycle*) no módulo *pulse width modulation*
<RESET>	Por norma, refere-se ao nome do *reset* de um módulo
<RESULT>	Por norma, refere-se ao operando correspondente ao resultado
<RESET_TYPE>	Por norma, refere se o *reset* de um módulo/modelo deverá ser síncrono ou assíncrono
<RESET_STATE_NAME>	Por norma, refere-se ao estado a que a máquina de estados (módulo ou processo) deverá retornar sempre que é feito *reset*
<RES_SUB_TYPE>	Por norma, refere-se ao tipo do operador resultado (*STD_LOGIC_VECTOR*, *UNSIGNED* ou *SIGNED*)
<SELECT>	Por norma, refere-se ao nome da variável *select*, presente por exemplo nos *multiplexers* e *demultiplexers*

Continuação da Tabela 3.1	
Campos	Descrição
<SELECTABLE_VALUE>	Por norma, refere-se a um valor a atribuir, por exemplo a uma variável. Este valor difere de módulo para módulo, ou até mesmo diferir várias vezes no mesmo módulos, dependendo de outras configurações
<SUB_TYPE>	Por norma, refere-se ao tipo de sinal (*STD_LOGIC*, *STD_LOGIC_VECTOR*, *UNSIGNED* ou *SIGNED*)
<SIGVAR>	Por norma, refere-se ao nome de um sinal/variável
<SIGNAL_NAME>	Por norma, refere-se ao nome de sinais
<SIGNAL_MODE>	Por norma, refere-se ao modo de um sinal (*in*, *out* ou *inout*)
<SIGNAL_TYPE>	Por norma, refere-se ao tipo de sinal (*STD_LOGIC*, *STD_LOGIC_VECTOR*, *UNSIGNED*, *SIGNED* ou *INTEGER*)
<SIGNAL_NUMBER_OF_BITS>	Por norma, refere-se ao número de *bits* de um sinal
<SUM_BIT>	Por norma, refere-se ao nome da variável de saída correspondente ao *bit* de soma
<TIME_MS>	Por norma, refere-se ao valor temporal do *debounce buttons* em milissegundos
Fim da Tabela	

Cada um do campos configuráveis pelo utilizador, descritos na tabela 3.1, está guardado num ficheiro JSON que corresponde a um módulo ou modelo.
Para além dos campos de configuração presentes na tabela 3.1, qualquer outro campo de configuração pode ser usado, desde que seja colocado no ficheiro na base de dados. A ferramenta irá automaticamente reconhecer esse campo e agir de acordo com as suas configurações.

3.3 A base de dados

A base de dados é constituída por um conjunto de ficheiros gravados, em formato JSON, no *GitHub*, como se pode observar na figura 3.5. Cada um desses ficheiros guardados, com excepção do *content_list_of_modules.json*, corresponde a um excerto de código ou módulo em electrónica digital que será carregado para a página *Web*. O ficheiro denominado de *content_list_of_modules.json* permite que a ferramenta saiba o nome e o URL (*Uniform Resource Locator*) de cada um dos excertos de código e módulos, isto é observável na figura 3.6.

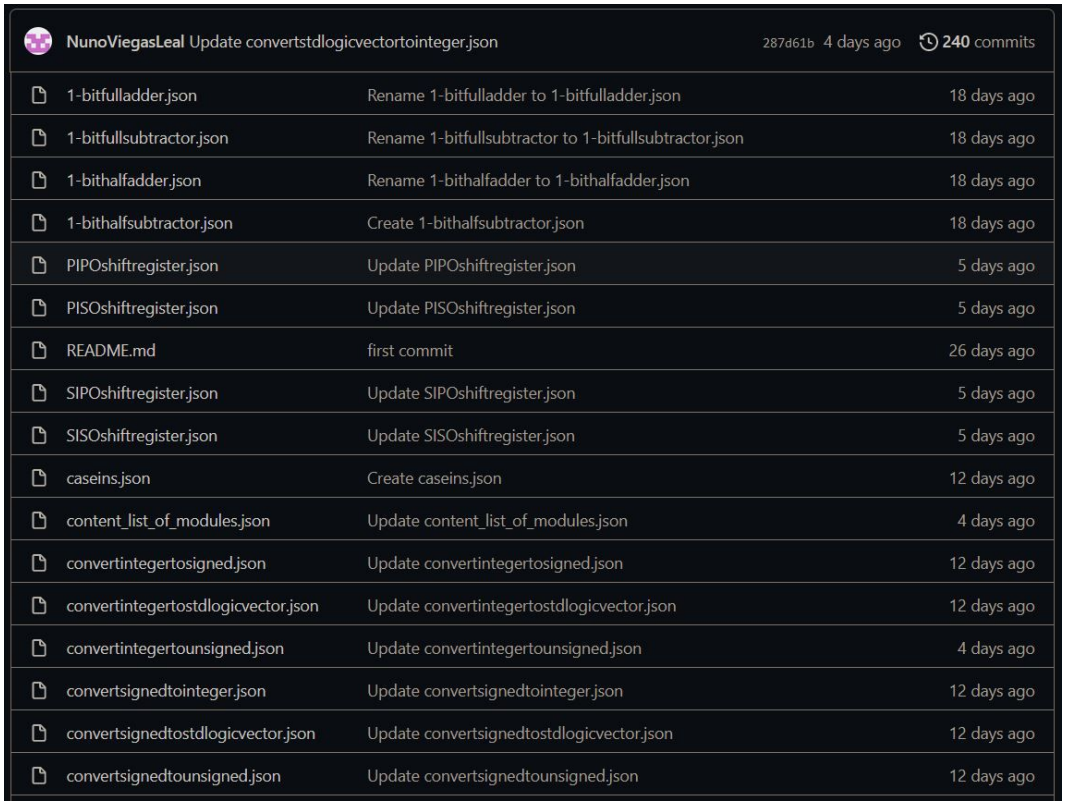

Figura 3.5: Alguns dos vários módulos presentes na base de dados do *GitHub*.

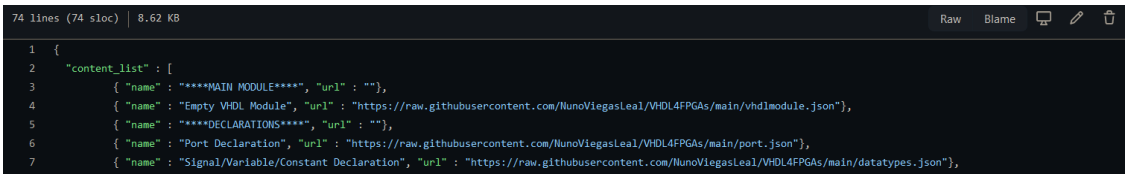

Figura 3.6: Figura com alguns excertos e módulos no ficheiro *content_list_of_modules.json* na base de dados. Este ficheiro contem o nome e URL de cada um dos excertos de código e módulos.

Cada ficheiro JSON é composto por um conjunto de campos, como se verifica na figura 3.7, cada um deles com uma finalidade. Os possíveis campos presentes num módulo são:

- *inputtext* - É interpretado pelo código em *JavaScript* como um comando para gerar caixas de texto para o utilizador preencher com o respectivo nome. Este campo é visível no rectângulo vermelho da figura 3.8;
- *buttons* - É interpretado pelo código em *JavaScript* como um comando para gerar botões. Cada botão será programado independentemente para uma acção específica ser realizada. Este campo é visível no rectângulo verde da figura 3.8;
- *selects* - É interpretado pelo código em *JavaScript* como um comando para gerar caixas de selecção com o nome presente em *name* e as opções presentes em *options*.

Este campo é visível no rectângulo azul da figura 3.8;

- *autogenerate* - Campo opcional. Corresponde ao campo que deverá ser gerado várias vezes pela ferramenta, dependendo do excerto de código ou módulo. Este campo é visível no rectângulo amarelo da figura 3.8;
- *moduleName* - Corresponde ao nome do módulo;
- *moduleCode* - É interpretado pelo código em *JavaScript* como o código em VHDL a ser carregado para a página *Web* e ser configurado de acordo com o valor introduzido nos campos disponibilizados ao utilizador;
- *moduleCode2* - Campo opcional. Está presente apenas em módulos em que seja necessário uma alteração profunda da descrição de *hardware* de acordo com as opções do utilizador, como acontece no caso de alguns módulos com *reset* síncrono ou assíncrono.

Figura 3.7: Exemplo do módulo *Flip-Flop* tipo D guardado em formato JSON no *GitHub*.

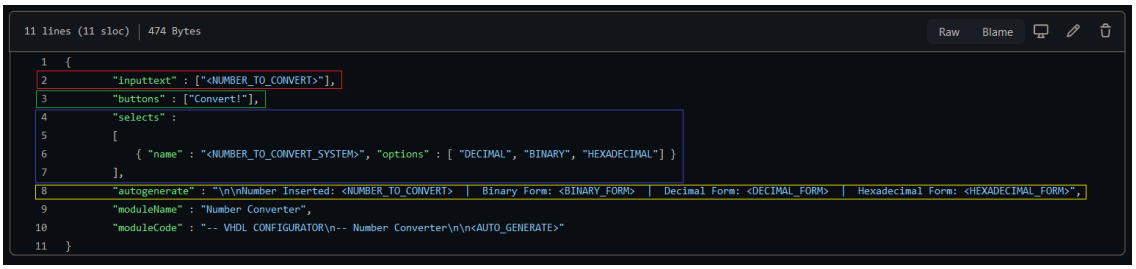

Figura 3.8: Exemplo do excerto de código *Number Converter* guardado em formato JSON no *GitHub*. A diferentes cores estão campos que serão interpretados pela ferramenta.

Realçar que, para o exemplo da figura 3.8, a ferramenta irá gerar um campo de texto com o nome *NUMBER_TO_CONVERT*; um botão com o nome *Convert!*; um campo de selecção múltipla com o nome *NUMBER_TO_CONVERT_SYSTEM*, cujas as opções de selecção serão *DECIMAL*, *BINARY* e *HEXADECIMAL*.
Os campos referidos anteriormente, irão alterar o valor dos respectivos campos delimitados pelos caracteres "<" e ">", que se encontra em *autogenerate* (rectângulo amarelo na figura 3.8).

3.4 Tecnologias utilizadas na implementação

Nesta secção são apresentadas as linguagens e métodos de comunicação utilizados. O *JavaScript* é uma linguagem de programação que torna uma página *Web* interactiva. Importa também fazer uma rápida referência ao HTML e CSS, que também foram usados, e servem para criar e definir os elementos e estilos das páginas *Web*, respectivamente.
Note-se, desde já que, o código em *JavaScript* (A versão usada para teste foi a *ECMAScript 2021*) foi implementado com funcionalidades para certas versões dos *browsers*, pelo que a sua actualização é recomendada. A NOVA-HDL-Assist foi testado no *Mozilla Firefox* (A versão usada para teste foi a 94.0), *Google Chrome* (A versão usada para teste foi a 96.0.4664.45) e *Microsoft Edge* (A versão usada para teste foi a 96.0.1054.34). É provável que a ferramenta seja suportada por outros *browsers* e outras versões, contudo a experiência poderá não ser a ideal.

3.5 Pedido de *request* à base de dados

Para se obter e descarregar os dados presentes na base de dados para uma *string*, foi implementado o código presente na listagem 3.1.

```
1 request.open('GET', requestURL, true);
2 request.responseType = 'json';
3 request.send();
```

Listagem 3.1: Pedido de *request*.

O método *HTTP GET* será realizado consoante o URL presente na *string requestURL*. Inicialmente, a página *Web* ao ser carregada, o valor presente na *string requestURL* corresponderá ao URL da lista com todos módulos disponíveis, ou seja da *content_list_of_modules.json* como mencionado na secção 3.3. Essa lista será guardada em variáveis internas e disponibilizada para a selecção do utilizador, como a figura 3.9 o exemplifica. O utilizador ao carregar num dos módulos irá substituir o valor presente na *string requestURL* pelo URL do módulo correspondente. Consequentemente esse módulo será carregado para a área de texto.

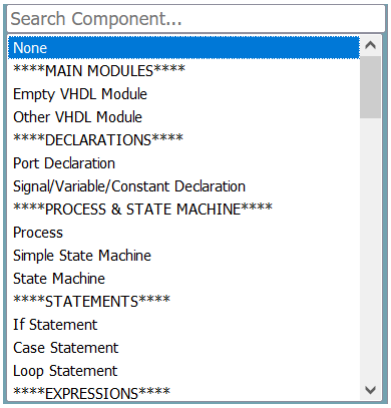

Figura 3.9: Exemplo de alguns módulos disponibilizados para selecção do utilizador.

3.6 Auto-geração dos Campos de Configuração

Na secção 3.3 foi elucidado como gerar os campos de configuração na base de dados, recorrendo a ficheiros JSON. Contudo, para além do JSON, é também necessário desenvolver um mecanismo no *JavaScript* que interprete o JSON e gere os campos necessários, bem como posicioná-los.

Uma vez recebido o ficheiro JSON proveniente da base de dados, os campos relativos ao ficheiro JSON serão guardados numa variável global chamada *globalJsonObj*. Através desta variável, por exemplo para criar caixas de texto, é necessário criar um ciclo de modo a ler todas as posições presentes no *array* correspondente. De seguida, é usado o *document.createElement* para se criar um novo elemento e atribuir-lhe algumas características, como a sua identificação. Por fim, para se escrever no elemento uma pequena identificação é usado o método *createTextNode* e usada a função *generateLabel* para se atribuir uma *label*. De modo a facilitar esta explicação, na listagem 3.2 está presente a parte do código em *JavaScript* responsável pela criação de caixas de texto que são editáveis pelo utilizador.

```
1  for (i in globalJsonObj.inputtext) {
2
3      createElement = document.createElement("INPUT");
4
5      //Gives an ID
6    createElement.setAttribute("id", globalJsonObj.inputtext[i]);
7
8    //Sets as required
9    createElement.setAttribute("required","");
10
11   //In order to populate the <RESET_STATE_NAME> options and forces the user
         to choose a correct option
12   if(globalJsonObj.inputtext[i] === "<NUMBER_OF_STATES>"){
13     createElement.setAttribute("onchange", "populateResetStateOptions()");
14   }
15
```

```
16    createTextNode = document.createTextNode(globalJsonObj.inputtext[i]);
17    createElement.appendChild(createTextNode);
18    document.body.appendChild(createElement);
19
20    //Generates an explanatory text
21    generateLabel("Insert " + globalJsonObj.inputtext[i] + ":",
          globalJsonObj.inputtext[i]);
22
23  }
```

Listagem 3.2: Exemplo da geração dos elementos em HTML através de *JavaScript*.

A figura 3.10 ilustra a disposição dos campos para o módulo da máquina de estados, após a leitura do ficheiro JSON e dos vários campos serem auto-gerados pela ferramenta.

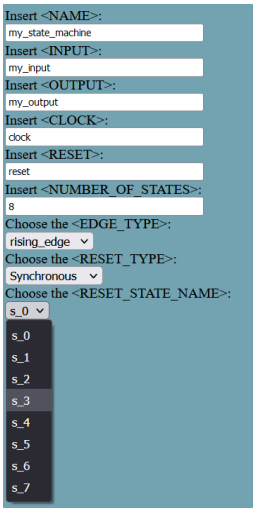

Figura 3.10: Conjunto de campos auto-gerados a serem preenchidos pelo utilizador. Neste caso os campos auto-gerados correspondem a uma máquina de estados.

3.7 Comandos de Configuração

O utilizador irá configurar o módulo através de um conjunto de campos auto-gerados, como a figura 3.10 o ilustra. O valor introduzido/seleccionado nesses campos irá corresponder a um conjunto de comandos sinalizados pelos caracteres "<" e ">" que serão posteriormente substituídos aquando as ordens do utilizador. No excerto de código em *JavaScript* presente na listagem 3.3 observa-se um exemplo de processamento desses comandos. Neste exemplo, é preciso ter em consideração qual é o tipo de campo com o qual a ferramenta está a trabalhar, podendo ser:

- Campo <*NUMBER_OF_BITS*> para os módulos e excertos de código no geral: Neste caso a ferramenta irá buscar o valor colocado pelo utilizador no campo <*NUMBER_OF_BITS*> e passar para uma variável chamada *inte*. Nas linhas 5, 6, 7, 8, 9 e 10, da

listagem 3.3, os valores irão ser substituídos no código a apresentar ao utilizador. Por exemplo na linha 10, o número colocado pelo utilizador irá ser convertido para binário e consequentemente substituído no código original;

- Campo <*NUMBER_OF_BITS*> para os módulos *Multiplexer*, *Demultiplexer*, *Encoder* e *Decoder*: Bastante semelhante ao descrito no ponto anterior, com a diferença que nestes módulos é necessário ter em conta a variação do número de *bits*, por exemplo, nas portas de entrada e *Select*.

```
1  //Fills some specific and unique values where it is necessary calculations
       (sometimes)
2  if (globalJsonObj.inputtext[i]  === "<NUMBER_OF_BITS>")
3  {
4    var inte = document.getElementById(globalJsonObj.inputtext[i]).value;
5    moduleCodeReceivedJSON = moduleCodeReceivedJSON.replaceAll("<MIDDLE_BIT>",
         inte/2);
6    moduleCodeReceivedJSON =
         moduleCodeReceivedJSON.replaceAll("<MIDDLE_BIT_PLUS_1>", (inte/2)+1);
7    moduleCodeReceivedJSON =
         moduleCodeReceivedJSON.replaceAll("<MIDDLE_BIT_MINUS_1>", (inte/2)-1);
8
9    moduleCodeReceivedJSON =
         moduleCodeReceivedJSON.replaceAll("<NUMBER_OF_BITS_MINUS_1>", inte-1);
10
11     moduleCodeReceivedJSON =
           moduleCodeReceivedJSON.replaceAll("<NUMBER_OF_BITS_BINARY>",
           decimalToBinary(inte-1,inte));
12   //Protect from outrun memory. Ex.: log(3) = outrun memory
13   if (globalJsonObj["moduleName"] == "Multiplexer" ||
         globalJsonObj["moduleName"] == "Demultiplexer"
14   || globalJsonObj["moduleName"] == "Encoder" || globalJsonObj["moduleName"]
         == "Decoder") {
15       moduleCodeReceivedJSON =
             moduleCodeReceivedJSON.replaceAll("<SELECT_NUMBER_OF_BITS>",
             Math.log2(inte)-1);
16       moduleCodeReceivedJSON =
             moduleCodeReceivedJSON.replaceAll("<SELECT_NUMBER_OF_BITS_BINARY>",
             decimalToBinary(inte-1, Math.log2(inte)));
17   }
18 }
```

Listagem 3.3: Exemplo do uso de comandos de configuração.

3.8 Interface Gráfica do Utilizador

O utilizador ao inicializar a página *Web* irá visualizar a lista de módulos disponíveis, como se pode ver do lado esquerdo da figura 3.11. Ao seleccionar um dos módulos, é feita a

comunicação com o servidor e respectivo *download*, sendo carregado para uma área de texto, como se pode observar na parte central da figura 3.11.

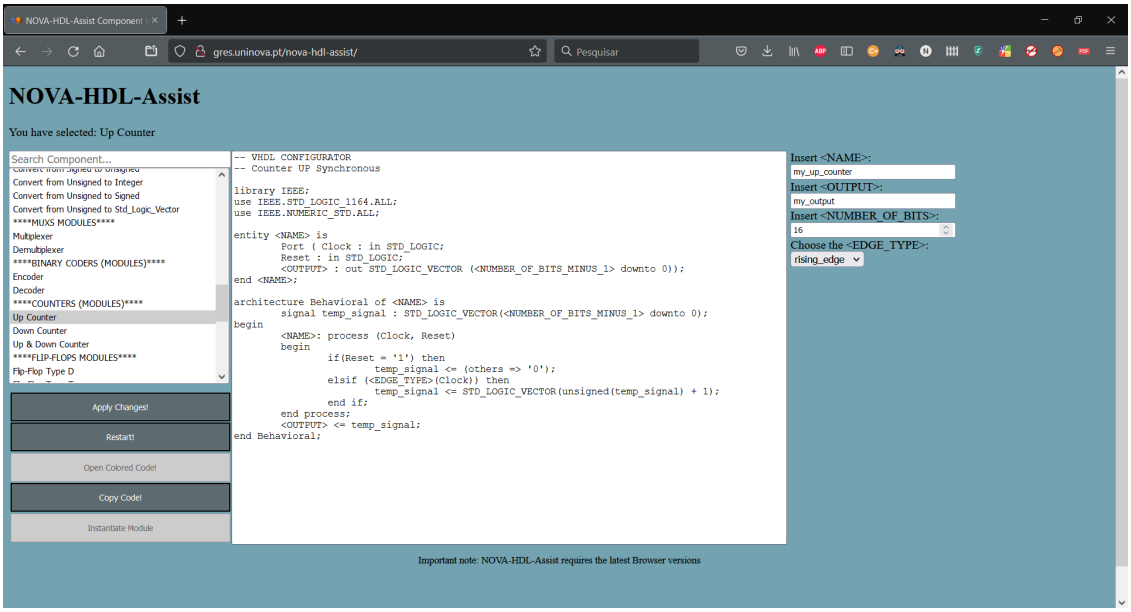

Figura 3.11: Ferramenta com o módulo de contador síncrono seleccionado.

O utilizador de seguida irá configurar o módulo através de um conjunto de campos auto-gerados, como o lado direito da figura 3.11 o ilustra. O valor introduzido/seleccionado nesses campos irá corresponder a um conjunto de comandos sinalizados pelos caracteres "<" e ">" que serão posteriormente substituídos aquando as ordens do utilizador.
Na zona central da figura 3.11 está presente o módulo correspondente a um contador síncrono cujas alterações desejadas pelo utilizador ainda não foram feitas, sendo possível reconhecer alguns comandos sinalizados pelos caracteres "<" e ">", como explicado anteriormente. Por exemplo, as *tags* <*NAME*> e <*OUTPUT*> visíveis no código VHDL irão ser substituídos pelos valores inseridos nos respectivos campos de entrada, visíveis no lado direito da figura 3.11. Importa realçar a *tag* <*EDGE_TYPE*> em que o utilizador terá de seleccionar uma das opções pré-definidas de uma lista (*rising_edge* ou *falling_edge*), em vez de escrever o nome desejado numa caixa de texto.

3.9 Instanciar de Módulos

O instanciar de módulos é uma mais valia da NOVA-HDL-Assist, uma vez que permite que o utilizador possa, em apenas um clique, instanciar e copiar para o seu projecto o módulo que configurou.
Esta funcionalidade em VHDL, é bastante interessante, uma vez que deste modo é possível adicionar componentes a outros ficheiros VHDL, podendo ter-se assim diferentes níveis de abstracção. O uso desta camada de abstracção é relevante, por exemplo, no caso de ser necessário propor aos alunos a implementação de um determinado sistema em que

certos módulos, com um elevado grau de complexidade, são fornecidos pelos tutores. Para instanciar um módulo na ferramenta em desenvolvimento o utilizador terá de preencher os campos auto-gerados para o módulo seleccionado e clicar no botão que corresponde ao instanciar do módulo. O instanciar do módulo será automaticamente gerado numa nova janela do *browser*, como se constata na figura 3.12.

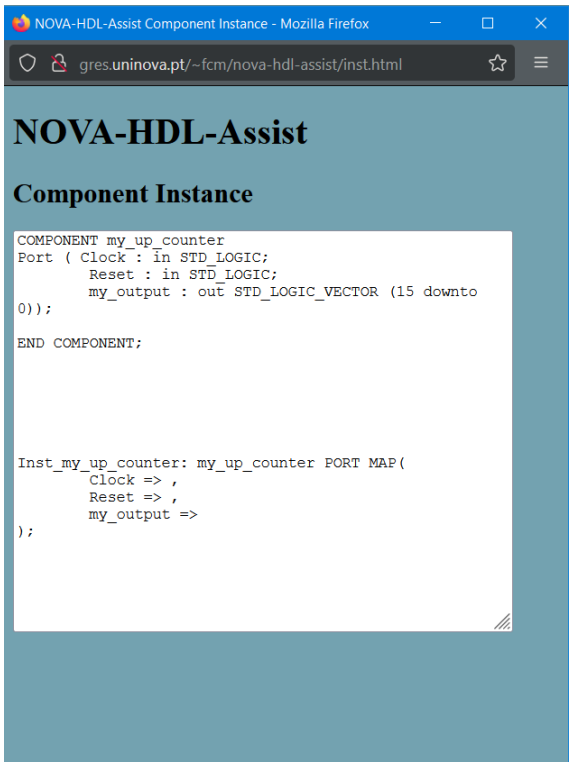

Figura 3.12: Módulo correspondente ao contador síncrono instanciado.

O código em *JavaScript* consiste numa *string* com todo o código em VHDL já configurado pelo utilizador. De modo a seguir a sintaxe e forma da VHDL essa *string* vai sofrer diversas transformações, sendo necessário obter todos os valores colocados pelo utilizador, iterar por todos e concatená-los, de modo a construir uma *string* que contenha tudo. Em 3.4 está um excerto de código responsável por concatenar todos os *inputs* e *outputs* e, ao mesmo tempo, limpar os valores numéricos. O resultado final, no caso do contador síncrono, é o que se encontra na figura 3.12.

```
1 for (var i = 0; i < inputsAndOutputsTreated.length; i++) {
2   if(isNaN(inputsAndOutputsTreated[i])){
3     textToLoad = textToLoad.concat("\n\t" + inputsAndOutputsTreated[i] + " =>
          " + ",");
4   }
5 }
```

Listagem 3.4: Excerto de código em que as entradas e saídas dos módulos são trabalhadas de acordo com a sintaxe e forma da VHDL.

3.10 Módulos e exemplos de descrições em VHDL oferecidos na Ferramenta

Nesta secção pretende-se dar a conhecer todos os módulos e exemplos de código VHDL, como conversores de dados, que estão presentes na ferramenta.
Os excertos de código e módulos presentes nesta secção foram desenvolvidos, testados e depois incorporados na base de dados para a sua apresentação na ferramenta ser possível.
Os excertos de código e módulos em VHDL presentes nesta secção foram escolhidos devido, não só à sua inevitabilidade do seu uso em projectos de nível baixo a intermédio, como devido à dificuldade que podem apresentar para alunos ou profissionais.

3.10.1 Declarações

3.10.1.1 Constantes, Variáveis e Sinais

Nesta subsecção é apresentado a forma como as constantes, variáveis e sinais estão presentes na ferramenta. Apesar de simples são bastante importantes e essenciais em qualquer descrição de *hardware*.
Na tabela 3.2 estão presentes os campos configuráveis, bem como a sua descrição.

Campos	Descrição
Tipo	O utilizador deverá indicar se pretende configurar uma variável, *signal* ou uma constante
Nome	O utilizador deverá indicar o nome da variável/constante/*signal*
Número de *bits*	O utilizador deverá indicar o número de *bits*
Indicação do subtipo	O utilizador deverá indicar o nome do subtipo, ou seja se é do tipo *unsigned*, *signed*, *std_logic* e *std_logic_vector*
Valor seleccionável	O utilizador deverá indicar o valor que deverá ter a declaração

Tabela 3.2: Tabela com os campos e descrição de configuração das constantes e variáveis.

Um exemplo genérico para a configuração de constantes está presente na listagem 3.5.

```
1 constant <Nome> : <Indicaçao_Subtipo> := Valor_do_Utilizador;
```

Listagem 3.5: Declaração genérica de uma constante.

Um exemplo genérico para a configuração de variáveis está presente na listagem 3.6.

```
1 variable <Nome> : <Indicaçao_Subtipo> := Valor_do_Utilizador;
```

Listagem 3.6: Declaração genérica de uma variável.

Já para a declaração de *signals*, um exemplo genérico para a configuração dos mesmos é apresentado na listagem 3.7.

```
1 signal <Nome> : <Indicaçao_Subtipo> := Valor_do_Utilizador;
```

Listagem 3.7: Declaração genérica de um *signal*.

Note-se que o espaço reservado aos valores seleccionáveis variam consoante o subtipo. Para o subtipo *std_logic*, o utilizador apenas poderá escolher o valor '0' ou '1'. Para os restantes subtipos (*unsigned*, *signed* e *std_logic_vector*) o utilizador terá a opção de atribuir a todos os *bits* o valor '0', a todos os *bits* o valor '1', ou qualquer outro valor binário atribuído pelo utilizador.

3.10.1.2 Declaração de Portas

Na tabela 3.3 estão presentes os campos configuráveis, bem como a sua descrição. Nas listagens 3.8 e 3.9, são apresentados duas declarações genéricas de um sinal.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do *signal*
Número de *bits*	O utilizador deverá indicar o número de *bits*
Indicação do tipo	O utilizador deverá indicar o nome do tipo, ou seja se é do tipo *unsigned*, *signed*, *std_logic*, *std_logic_vector* ou *integer*
Indicação do modo	O utilizador deverá indicar se o *signal* será *in*, *out* ou *inout*

Tabela 3.3: Tabela com os campos e descrição de configuração da declaração de portas.

```
1 Port ( <signal_name> : <signal_mode> <signal_type> );
```

Listagem 3.8: Declaração genérica de um *signal* quando o tipo do *signal* é *std_logic* ou *integer*.

```
Port ( <signal_name> : <signal_mode> <signal_type> (<number_of_bits>-1 downto
    0));
```

Listagem 3.9: Declaração genérica de um *signal* quando o tipo do *signal* é *std_logic_vector unsigned* ou *signed*.

Note-se que na declaração de portas, à semelhança da secção 3.10.1.1, os *signals* do tipo *unsigned*, *signed* e *std_logic_vector* permitem a configuração do número de *bits* por parte do utilizador. Para os tipos *std_logic* e *integer*, tal não é possível.

3.10.2 Instruções Sequenciais Não Configuráveis

As instruções sequeciais disponíveis na ferramenta são o *if*, *case* e *loop*.

3.10.2.1 Instrução *If*

Um exemplo genérico da instrução *If*, está presente na listagem 3.10.

```
if Condiçao then
    -- If Comportamento
elsif Condiçao then
    -- Elsif Comportamento
else
    -- Else Comportamento
end if;
```

Listagem 3.10: Exemplo genérico da instrução *if*.

3.10.2.2 Instrução *Case*

Um exemplo genérico da instrução *Case*, está presente na listagem 3.11.

```
case Tipo_dados is
    when Opçao1 => -- Execuçao
    when Opçao1 => -- Execuçao
end case;
```

Listagem 3.11: Exemplo genérico da instrução *case*.

3.10.2.3 Instrução *Loop*

Um exemplo genérico da instrução *Loop*, está presente na listagem 3.12.

```
loop
    -- wait until Condiçao; -- Se necessário

    --Comportamento do Loop

end loop;
```

Listagem 3.12: Exemplo genérico da instrução *loop*.

3.10.3 Subprogramas Não Configuráveis

A ferramenta disponibiliza um *template* genérico para *procedures* e *functions* para o utilizador copiar e colar no seu projecto. Apesar de serem módulos não configuráveis é dada a possibilidade de o utilizador atribuir o nome e, no caso da *function*, qual o tipo da variável de retorno. Isto visa facilitar e reduzir o tempo de pesquisa por parte do projectista.

3.10.3.1 *Procedures*

Um exemplo genérico de um *Procedure*, está presente na listagem 3.13.

```
procedure Nome_procedure (
    -- Signal Nome_variável : Out state_type;
) is
begin
    -- Comportamento do Procedure
end Nome_procedure;
```

Listagem 3.13: Exemplo genérico de um *procedure*.

3.10.3.2 *Functions*

Um exemplo genérico de uma *Function*, está presente na listagem 3.14.

```
function Nome_funçao (parameter_list) return tipo_de_return is
  -- Signal Nome_variável : bit;
begin
  -- Comportamento da funçao
  return nome_return;
end Nome_funçao;
```

Listagem 3.14: Exemplo genérico de uma *function*.

3.10.3.3 Processo

O processo presente na listagem 3.15 não é configurável, contudo a sua presença na ferramenta é uma mais-valia para utilizador pouco experientes, no sentido de que irá reduzir o tempo de pesquisa.

```
1 nome_processo: process(--lista de sensibilidades)
2   --Declaraçoes
3 begin
4   --Execuçao
5 end process;
```

Listagem 3.15: Exemplo genérico de um Processo.

3.10.4 Conversores de dados

3.10.4.1 Conversor geral de números

O conversor geral de números tem como objectivo facilitar a conversão de valores para outros sistemas numéricos, sem que se tenha de recorrer a outras aplicações. Desta forma o utilizador ao especificar os campos presentes na tabela 3.4, a ferramenta irá retornar o valor introduzido em binário, decimal e hexadecimal, como a listagem 3.16 o exemplifica.

Campos	Descrição
Número a converter	O utilizador deverá indicar o número a converter
Sistema numerário do número a converter	O utilizador deverá indicar qual o sistema numerário do número que deseja converter

Tabela 3.4: Tabela com os campos e descrição de configuração do conversor geral de dados.

```
1 Number Inserted: <NUMBER_TO_CONVERT>  |  Binary Form: <BINARY_FORM>  |
    Decimal Form: <DECIMAL_FORM>  |  Hexadecimal Form: <HEXADECIMAL_FORM>
```

Listagem 3.16: Exemplo genérico com os campos por configurar do Conversor geral de números.

3.10.4.2 Conversores de dados específicos

A ferramenta disponibiliza também um conjunto de exemplos que poderão auxiliar o projectista a converter diferentes tipos de dados. Para todos os exemplos de conversão é pedido que o utilizador preencha os campos presentes na tabela 3.5. Os exemplos de conversão estão presentes, em código VHDL, na listagem 3.17 que corresponde ao conversor de *Integer* para *Signed*, na listagem na 3.18 que corresponde ao conversor de *Integer*

para *Std_logic_vector*, na listagem 3.19 que corresponde ao conversor de *Integer* para *Unsigned*, na listagem 3.20 que corresponde ao conversor de *Std_logic_vector* para *Integer*, na listagem 3.21 que corresponde ao conversor de *Std_logic_vector* para *Signed*, na listagem 3.22 que corresponde ao conversor de *Std_logic_vector* para *Unsigned*, na listagem 3.23 que corresponde ao conversor de *Signed* para *Integer*, na listagem 3.24 que corresponde ao conversor de *Signed* para *Std_logic_vector*, na listagem 3.25 que corresponde ao conversor de *Signed* para *Unsigned*, na listagem 3.26 que corresponde ao conversor de *Unsigned* para *Integer*, na listagem 3.27 que corresponde ao conversor de *Unsigned* para *Signed* e na listagem 3.28 que corresponde ao conversor de *Unsigned* para *Std_logic_vector*. Estes conversores de dados foram adaptados de [32].

Campos	Descrição
Variável de entrada	O utilizador deverá indicar o nome da variável de entrada
Variável de saída	O utilizador deverá indicar o nome da variável de saída
Número de *bits*	O utilizador deverá indicar o número de *bits*

Tabela 3.5: Tabela com os campos e descrição de configuração dos conversores de dados.

```
1 signal <Variável de entrada>  : INTEGER;
2 signal <Variável de saída> : SIGNED(<Número de bits - 1> downto 0);
3
4 <Variável de saída> <= to_signed(<Variável de entrada>, <Variável de
      saída>'length);
```

Listagem 3.17: Conversor de *Integer* para *Signed*.

```
1 signal <Variável de entrada>  : INTEGER;
2 signal <Variável de saída> : STD_LOGIC_VECTOR(<Número de bits - 1> downto 0);
3
4 -- Como converter inteiros positivos
5 <Variável de saída> <= STD_LOGIC_VECTOR(to_unsigned(<Variável de entrada>,
      <Variável de saída>'length));
6
7 -- Como converter inteiros positivos ou negativos
8 <Variável de saída> <= STD_LOGIC_VECTOR(to_signed(<Variável de entrada>,
      <Variável de saída>'length));
```

Listagem 3.18: Conversor de *Integer* para *Std_logic_vector*.

```
signal <Variável de entrada>  : INTEGER;
signal <Variável de saída> : UNSIGNED(<Número de bits - 1> downto 0);

<Variável de saída> <= to_unsigned(<Variável de entrada>, <Variável de
    saída>'length);
```

Listagem 3.19: Conversor de *Integer* para *Unsigned*.

```
signal <Variável de entrada>  : STD_LOGIC_VECTOR(<Número de bits - 1> downto
    0);
signal <Variável de saída> : INTEGER;

-- Caso do unsigned
<Variável de saída> <= to_integer(UNSIGNED(<Variável de entrada>));

-- Caso do signed
<Variável de saída> <= to_integer(SIGNED(<Variável de entrada>));
```

Listagem 3.20: Conversor de *Std_logic_vector* para *Integer*.

```
signal <Variável de entrada>  : STD_LOGIC_VECTOR(<Número de bits - 1> downto
    0);
signal <Variável de saída> : SIGNED(<Número de bits - 1> downto 0);

<Variável de saída> <= SIGNED(<Variável de entrada>);
```

Listagem 3.21: Conversor de *Std_logic_vector* para *Signed*.

```
signal <Variável de entrada>  : STD_LOGIC_VECTOR(<Número de bits - 1> downto
    0);
signal <Variável de saída> : UNSIGNED(<Número de bits - 1> downto 0);

<Variável de saída> <= UNSIGNED(<Variável de entrada>);
```

Listagem 3.22: Conversor de *Std_logic_vector* para *Unsigned*.

```
signal <Variável de entrada>  : SIGNED(<Número de bits - 1> downto 0);
signal <Variável de saída> : INTEGER;

<Variável de saída> <= to_integer(<Variável de entrada>);
```

Listagem 3.23: Conversor de *Signed* para *Integer*.

```
signal <Variável de entrada>  : SIGNED(<Número de bits - 1> downto 0);
signal <Variável de saída> : STD_LOGIC_VECTOR(<Número de bits - 1> downto 0);

<Variável de saída> <= STD_LOGIC_VECTOR(<Variável de entrada>);
```

Listagem 3.24: Conversor de *Signed* para *Std_logic_vector*.

```
signal <Variável de entrada>  : SIGNED(<Número de bits - 1> downto 0);
signal <Variável de saída> : UNSIGNED(<Número de bits - 1> downto 0);

<Variável de saída> <= UNSIGNED(<Variável de entrada>);
```

Listagem 3.25: Conversor de *Signed* para *Unsigned*.

```
signal <Variável de entrada>  : UNSIGNED(<Número de bits - 1> downto 0);
signal <Variável de saída> : INTEGER;

<Variável de saída> <= to_integer(<Variável de entrada>);
```

Listagem 3.26: Conversor de *Unsigned* para *Integer*.

```
signal <Variável de entrada>  : UNSIGNED(<Número de bits - 1> downto 0);
signal <Variável de saída> : SIGNED(<Número de bits - 1> downto 0);

<Variável de saída> <= SIGNED(<Variável de entrada>);
```

Listagem 3.27: Conversor de *Unsigned* para *Signed*.

```
signal <Variável de entrada>  : UNSIGNED(<Número de bits - 1> downto 0);
signal <Variável de saída> : STD_LOGIC_VECTOR(<Número de bits - 1> downto 0);

<Variável de saída> <= STD_LOGIC_VECTOR(<Variável de entrada>);
```

Listagem 3.28: Conversor de *Unsigned* para *Std_logic_vector*.

3.10.4.3 Conversor de tipo de dados

O conversor de tipo de dados indica a sintaxe que irá permitir que o utilizador converta uma variável num determinado tipo de dados, noutro.
Na tabela 3.6, estão presentes os campos e descrição de configuração. Um exemplo genérico de um conversor de tipo de dados, está presente na listagem 3.29.

Campos	Descrição
Nome	O utilizador deverá indicar o nome da variável
Tipo de dados original	O utilizador deverá indicar qual o tipo de dados original da variável inserida
Tipo de dados a converter para	O utilizador deverá indicar qual o tipo de dados para o qual deseja converter a variável inserida

Tabela 3.6: Tabela com os campos e descrição de configuração do conversor de tipo de dados.

```
1 <DATA_TYPE_TO_CONVERT_TO>(<INPUT>)
```

Listagem 3.29: Exemplo genérico com os campos por configurar do Conversor de tipo de dados.

3.10.4.4 Conversor de decimal para binário

O conversor de decimal para binário, ao contrário dos outros conversores, foi concebido com o intuito de ser simples e directo. Isto porque, em princípio, de todas as conversões possíveis, a maioria das conversões a serem realizadas pelos utilizadores serão de decimal para binário.
Na tabela 3.7, estão presentes os campos e descrição de configuração. Um exemplo genérico de um Conversor de decimal para binário, está presente na listagem 3.30.

Campos	Descrição
Número a converter	O utilizador deverá indicar o número em decimal que deseja converter
Número de *bits*	O utilizador deverá indicar de quantos *bits* deverá ser o número em binário

Tabela 3.7: Tabela com os campos e descrição de configuração do conversor de decimal para binário.

```
1 Decimal Number Inserted: <NUMBER_TO_CONVERT>  |  Binary Form: <BINARY_FORM>
```

Listagem 3.30: Exemplo genérico com os campos por configurar do Conversor de decimal para binário.

3.10.5 Módulos Configuráveis

Nesta secção são apresentados os módulos configuráveis oferecidos pela ferramenta. A ferramenta ao ter como versatilidade a configuração dos módulos, é uma mais valia na edição de código VHDL por parte do utilizador.
Em termos de organização, cada subsecção é relativa a um módulo. Em cada módulo são expostos cada campo configurável e o seu respectivo exemplo em VHDL.

3.10.5.1 *Flip-Flop*

Como [33] menciona, enquanto que uma porta lógica é o elemento mais básico da lógica combinatória, já o seu equivalente em lógica sequencial é o *flip-flop*. A importância e aplicabilidade dos *flip-flop* não é exclusiva do seu uso como bloco individual em vários sistemas, mas também como o elemento básico de várias funções lógicas complexas. De seguida, na tabela 3.8, são apresentados os campos a preencher pela ferramenta bem como a sua descrição. Nas listagens 3.31, 3.32, 3.33 e 3.34 estão descritos, respectivamente, em VHDL os *flip-flops* dos tipos SR, JK, D e T. Estes módulos foram adaptados de [34].

Campos	Descrição
Nome	O utilizador deverá indicar o nome do *flip-flop* a ser configurado
Tipo de *Edge Triggered*	Modifica o estado do *flip-flop* com base no estado positivo (*rising-edge*) ou negativo (*falling-edge*) do pulso do *clock*

Tabela 3.8: Tabela com os campos e descrição de configuração do *Flip-Flop*.

```
1 entity <Nome>_SR_flipflop is
2     Port ( Set: in STD_LOGIC;
3            Reset: in STD_LOGIC;
4            Clock: in STD_LOGIC;
5            --
6            Q_Signal: out STD_LOGIC);
7 end <Nome>_SR_flipflop;
8
9 architecture Behavioral of <Nome>_SR_flipflop is
```

```
10
11 begin
12
13     SR_flipflop: process(Clock)
14     begin
15         if(<Tipo_Edge>(Clock)) then
16             if(Set='0' and Reset='0')then
17
18             elsif(Set='1' and Reset='1')then
19                 Q_Signal<='Z';
20             elsif(Set='0' and Reset='1')then
21                 Q_Signal<='0';
22             else
23                 Q_Signal<='1';
24             end if;
25         end if;
26     end process;
27
28 end Behavioral;
```

Listagem 3.31: *Flip-flop* tipo SR configurável.

```
1  entity <Nome>_JK_flipflop is
2      Port ( J: in STD_LOGIC;
3             K: in STD_LOGIC;
4             Clock: in STD_LOGIC;
5             --
6             Q_Signal: out STD_LOGIC);
7  end <Nome>_JK_flipflop;
8
9  architecture Behavioral of <Nome>_JK_flipflop is
10
11 begin
12
13     JK_flipflop: process(Clock)
14         variable temp_signal: STD_LOGIC;
15     begin
16         if (<Tipo_Edge>(Clock)) then
17             if (J='0' and K='0') then
18                 temp_signal:=temp_signal;
19             elsif (J='1' and K='1') then
20                 temp_signal:=not temp_signal;
21             elsif (J='0' and K='1') then
22                 temp_signal:='0';
23             else
24                 temp_signal:='1';
25             end if;
26         end if;
27         Q_Signal<=temp_signal;
28     end process;
```

```
29
30 end Behavioral;
```

Listagem 3.32: *Flip-flop* tipo JK configurável.

```
1  entity <Nome>_D_flipflop is
2      Port ( D : in STD_LOGIC;
3             Clock : in STD_LOGIC;
4             --
5             Q_Signal : out STD_LOGIC);
6  end <Nome>_D_flipflop;
7
8  architecture Behavioral of <Nome>_D_flipflop is
9
10 begin
11     D_flipflop: process(Clock)
12     begin
13         if(<Tipo_Edge>(Clock)) then
14             Q_Signal <= D;
15         end if;
16     end process;
17
18 end Behavioral;
```

Listagem 3.33: *Flip-flop* tipo D configurável.

```
1  entity <Nome>_T_flipflop is
2      Port ( T: in STD_LOGIC;
3             Clock: in STD_LOGIC;
4             --
5             Q_Signal: out STD_LOGIC);
6  end <Nome>_T_flipflop;
7
8  architecture Behavioral of <Nome>_T_flipflop is
9
10 begin
11
12     T_flipflop: process (Clock)
13         variable temp_sig: STD_LOGIC := '0';
14     begin
15         if <Tipo_Edge>(Clock) then
16             if T='0' then
17                 temp_sig := temp_sig;
18             elsif T='1' then
19                 temp_sig := not (temp_sig);
20             end if;
21         end if;
22         Q_Signal <= temp_sig;
23     end process;
24
```

```
25 end Behavioral;
```

Listagem 3.34: *Flip-flop* tipo T configurável.

3.10.5.2 Contadores

Como [33] refere, um contador é usado maioritariamente para aplicações relacionadas com a contagem, onde, por exemplo, face a um sinal é necessário medir a sua frequência, ou o intervalo de tempo entre dois instantes. Os contadores foram adaptados de [34].

Contador Síncrono

O contador síncrono tem este nome devido ao sincronismo do *clock*, ou seja, todos os *flip-flops* envolvidos na contagem mudam de estado ao mesmo tempo. Na ferramenta são oferecidos os contadores *up*, *down* e *up & down*. Apesar de, em seguida serem apresentados apenas os *process* relativo a cada um deles, cada um destes módulos está presente na ferramenta de forma independente. A tabela 3.9 contém os campos a serem preenchidos pelo utilizador. O preenchimento desses campos irá modelar as descrições em VHDL presentes nas listagens 3.35, 3.36 e 3.37.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do contador a ser configurado
Nome da variável de saída	Indica o nome da variável de saída
Número de *bits* do contador	Indica de quantos *bits* deverá ser o contador
Tipo de *Edge*	Indica se o contador deverá ser sensível ao *rising edge* ou *falling edge* do *clock*

Tabela 3.9: Tabela com os campos e descrição de configuração do Contador.

No caso de o utilizador optar pelo tipo de contagem decrescente ou crescente e decrescente, os processos são os presentes nas listagens 3.36 e 3.37.

```
entity <Nome> is
    Port (
        Clock: in STD_LOGIC;
        Reset: in STD_LOGIC;
        --
        <Variável de Saída>: out STD_LOGIC_VECTOR (<Número de bits - 1>
            downto 0)
        );
end <Nome>;

architecture Behavioral of <Nome> is
    signal temp_signal : STD_LOGIC_VECTOR(<Número de bits - 1> downto 0);
begin

    <Nome>_Sync_Up_Counter: process (Clock, Reset)

    begin

        if(Reset = '1') then
           temp_signal <= (others => '0');
        elsif (<Tipo de Edge>(Clock)) then
           temp_signal <= STD_LOGIC_VECTOR(unsigned(temp_signal) + 1);
        end if;
    end process;
    <Variável de Saída> <= temp_signal;

end Behavioral;
```

Listagem 3.35: Contador configurável com o processo *up* síncrono.

```
<Nome>_Sync_Down_Counter:
process (Clock, Reset)

begin

    if(Reset = '1') then
       temp_signal <= (others => '1');
    elsif (<Tipo de Edge>(Clock)) then
       temp_signal <= STD_LOGIC_VECTOR(unsigned(temp_signal) - 1);
    end if;
end process;
```

Listagem 3.36: Processo *down* síncrono para o contador configurável.

```
1  <Nome>_Sync_UpDown_Counter:
2  process (Clock, Reset)
3
4  begin
5
6      if(Reset = '1') then
7         temp_signal <= (others => '0');
8      elsif (<Tipo de Edge>(Clock)) then
9         if (UpOrDown = '0') then
10             temp_signal <= STD_LOGIC_VECTOR(unsigned(temp_signal) + 1);
11         elsif (UpOrDown = '1') then
12             temp_signal <= STD_LOGIC_VECTOR(unsigned(temp_signal) - 1);
13         end if;
14     end if;
15 end process;
```

Listagem 3.37: Processo *up* e *down* síncrono para o contador configurável.

3.10.5.3 *Shift Register*

O *shift register* é um componente em sistemas digitais composto por vários *flip-flops* de um único *bit*, usado para o armazenamento e transferência de dados [33].
Para criar este módulo, os *flip-flops* são conectados em série de modo a que a saída de um seja a entrada do seguinte. A saída do *shift register* pode consistir numa saída de dados em série ou em paralelo, a mesma flexibilidade é aplicada na entrada, por isso os *shift registers* podem ser baseados em vários métodos de leitura e escrita de dados. Esses métodos podem ser:

- *Serial-in to Parallel-out* (SIPO) - A escrita no registo é feito em série, *bit* a *bit*. A leitura é feita simultâneamente de todos os *flip-flops* presentes, ou seja em paralelo. O processo relativo à sua descrição em VHDL encontra-se na listagem 3.39;

- *Serial-in to Serial-out* (SISO) - A escrita no registo é feito em série, *bit* a *bit*. A leitura é também feita em série. O processo relativo à sua descrição em VHDL encontra-se na listagem 3.40;

- *Parallel-in to Serial-out* (PISO) - A escrita é feita *simultâneamente* de todos os *flip-flops* presentes, ou seja em paralelo. A leitura no registo é feito em série, *bit* a *bit*. O processo relativo à sua descrição em VHDL encontra-se na listagem 3.41;

- *Parallel-in to Parallel-out* (PIPO) - A escrita é feita simultâneamente em todos os *flip-flops* presentes, ou seja em paralelo. A leitura é também feita em paralelo. O processo relativo à sua descrição em VHDL encontra-se na listagem 3.42;

Cada um dos processos auto-gerados de acordo com a selecção do *Método de escrita e leitura de dados* por parte do utilizador, na tabela 3.10, são apresentados de seguida. Note-se

Campos	Descrição
Nome	O utilizador deverá indicar o nome do registo a ser configurado
Nome da variável de saída	O utilizador deverá indicar o nome da variável de saída
Nome da variável de entrada	O utilizador deverá indicar o nome da variável de entrada
Nome do *Clock*	O utilizador deverá indicar o nome do *clock*
Tipo de *reset*	O utilizador deverá indicar se o registo terá um *reset* assíncrono ou síncrono
Nome do *reset*	O utilizador deverá indicar o nome do *reset*
Número de *bits*	Indica de quantos *bits* deverá ser o registo
Tipo de *Edge* de um pulso	O utilizador deverá escolher entre *rising edge* e *falling edge* de um pulso

Tabela 3.10: Tabela com os campos e descrição de configuração do Registo.

que foram adaptados de [35]. Importa referir que, na ferramenta, os métodos apresentados anteriormente serão adaptados ao registo configurável sem processos, presente na listagem 3.38.

```
entity <Nome> is
    Port ( <Clock>: in STD_LOGIC;
           <Reset>: in STD_LOGIC;
           <Variável de Entrada>;
           --
           <Variável de Saída>);
end <Nome>;

architecture Behavioral of <Nome> is

signal Q_signal: STD_LOGIC_VECTOR((<Número de bits - 1>) downto 0);

begin
     --Processo relativo ao registo
     --( SIPO, SISO, PIPO ou PISO) virá aqui
end Behavioral;
```

Listagem 3.38: Registo configurável sem processos.

```
SIPO_REGISTER: process(<Clock>)
begin
    if <Tipo_Edge>(<Clock>) then
        if <Reset> = '1' then
            Q_signal <= (others => '0');
        else
            Q_signal((<Número de bits - 1>) downto (<Número de bits>/2-1)) <=
                Q_signal((<Número de bits>/2) downto 0);

            Q_signal(0) <= <Variável de Entrada>;
        end if;
    end if;
end process;

<Variável de Saída> <= Q_signal;
```

Listagem 3.39: Processo relativo ao SIPO do registo configurável.

```
SISO_REGISTER: process(<Clock>)
begin
    if <Tipo_Edge>(<Clock>) then
        if <Reset> = '1' then
            Q_signal <= (others => '0');
        else
            Q_signal((<Número de bits - 1>) downto (<Número de bits>/2-1)) <=
                Q_signal((<Número de bits>/2) downto 0);

            Q_signal(0) <= <Variável de Entrada>;
        end if;
    end if;
end process;

<Variável de Saída> <= Q_signal(<Número do
Bit mais significativo>);
```

Listagem 3.40: Processo relativo ao SISO do registo configurável.

```
PISO_REGISTER: process(<Clock>)
begin
    if <Tipo_Edge>(<Clock>) then
        if <Reset> = '1' then
            Q_signal <= (others => '0');
        else
            if LOAD = '1' then
                Q_signal((<Número de bits>-1) downto (<Número de bits>/2-1))
                    <= Q_signal((<Número de bits>/2) downto 0);
            else
                Q_signal(<MSB>) <= <Variável de Entrada>(<MSB>);
                Q_signal(<MSB>-1) <= <Variável de Entrada>(<MSB>-1);
                -- Continua de acordo com o número de bits
                -- inserido. Autogerado
                Q_signal(1) <= <Variável de Entrada>(1);
            end if;
            Q_signal(0) <= <Variável de Entrada>(0);
        end if;
    end if;
end process;

<Variável de Saída> <= Q_signal(<Número do Bit mais significativo>);
```

Listagem 3.41: Processo relativo ao PISO do registo configurável.

```
1  PIPO_REGISTER: process(<Clock>)
2  begin
3      if <Tipo_Edge>(<Clock>) then
4          if <Reset> = '1' then
5              Q_signal <= (others => '0');
6          else
7              if LOAD = '1' then
8                  Q_signal((<Número de bits-1>) downto (<Número de bits>/2-1))
                       <= Q_signal((<Número de bits>/2) downto 0);
9              else
10                 Q_signal(<MSB>) <= <Variável de Entrada>(<MSB>);
11                 Q_signal(<MSB-1>) <= <Variável de Entrada>(<MSB-1>);
12                 -- Continua de acordo com o número de bits
13                 -- inserido. Autogerado
14                 Q_signal(1) <= <Variável de Entrada>(1);
15             end if;
16             Q_signal(0) <= <Variável de Entrada>(0);
17         end if;
18     end if;
19 end process;
20
21 <Variável de Saída> <= Q_signal;
```

Listagem 3.42: Processo relativo ao PIPO do registo configurável.

Relativamente ao *reset* assíncrono e síncrono, na secção 3.10.5.9 são apresentados e comparados ambos os casos.

3.10.5.4 *Multiplexers* e *Demultiplexers*

Os *multiplexers* e *demultiplexers* são circuitos que seleccionam as entradas para as saídas, ou seleccionam as saídas para as entradas, através do recebido nas entradas responsáveis pela selecção [33].

Multiplexer

O *multiplexer* é um circuito lógico composto por um máximo de 2^n entradas e n entradas de selecção. As entradas de selecção orientam a informação presente na entrada até à respectiva saída seleccionada [33]. O *multiplexer* foi adaptado de [36].
No *multiplexer* é pedido que o utilizador preencha os cinco campos descritos na tabela 3.11. Existe uma limitação do número de *bits* de entrada, em função do número de *bits* de selecção inserido pelo utilizador.
A descrição em VHDL, presente genericamente na listagem 3.43, corresponde à ferramenta quando esta gerar todas as combinações de *bits* possíveis, com base no número de *bits* inseridos pelo utilizador em cada um dos campos descritos na tabela 3.11.

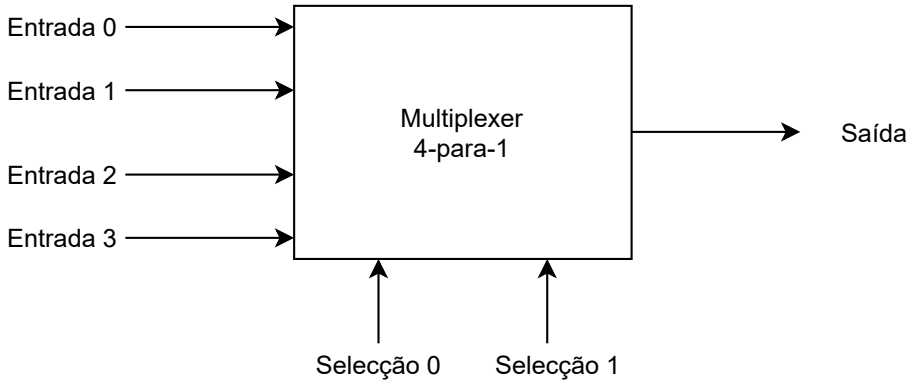

Figura 3.13: Representação do módulo correspondente ao *multiplexer*. Adaptado de [33].

Campos	Descrição
Nome	O utilizador deverá indicar o nome do *multiplexer* a ser configurado
Nome da variável de entrada	Indica o nome da variável de entrada
Nome da variável de saída	Indica o nome da variável de saída
Nome da variável de selecção	Indica o nome da variável de selecção
Número de *bits*	Indica de quantos *bits* deverá ser o módulo

Tabela 3.11: Tabela com os campos e descrição de configuração do *Multiplexer*.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada> : in STD_LOGIC;
3             <Variável de Selecçao> : in STD_LOGIC_VECTOR (<Número de bits de
                   selecçao> downto 0);
4             --
5             <Variável de Saída> : out STD_LOGIC_VECTOR (<Número de bits - 1>
                   downto 0));
6  end <Nome>;
7
8  architecture Behavioral of <Nome> is
9
10 begin
```

```
11
12      with <Variável de Selecçao> select output <=
13          <Variável de Entrada> when <Variável de Selecçao>,
14          <Variável de Entrada> when <Variável de Selecçao>,
15          <Variável de Entrada> when <Variável de Selecçao>,
16          <Variável de Entrada> when <Variável de Selecçao>,
17          -- ...  Continua dependendo do
18          -- número de bits inseridos
19          '0'  when others;
20
21  end Behavioral;
```

Listagem 3.43: Código VHDL da ferramenta por configurar. Este código corresponde ao *Multiplexer* configurável.

Demultiplexer

O *demultiplexer* é um circuito lógico bastante semelhante ao descodificador. Este módulo é composto no máximo por 2^n saídas e n entradas de selecção. As entradas de selecção orientam a informação presente na entrada até à respectiva saída seleccionada [33]. O *demultiplexer* foi adaptado de [36].

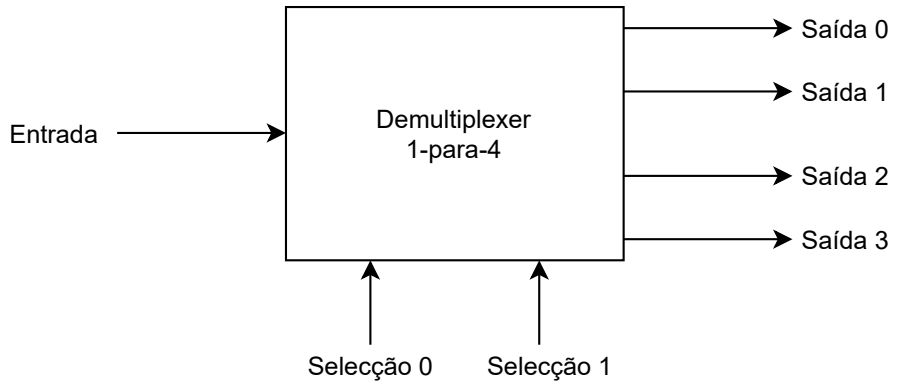

Figura 3.14: Representação do módulo correspondente ao *demultiplexer*. Adaptado de [33].

No *demultiplexer* é pedido que o utilizador preencha os cinco campos descritos na tabela 3.12. Existe uma limitação no número de *bits* de saída em função do número de *bits* de selecção inserido pelo utilizador.
A descrição em VHDL, presente genericamente na listagem 3.44, a ferramenta irá gerar todas as combinações de *bits* possíveis, com base no número de *bits* inseridos pelo utilizador em cada um dos campos descritos na tabela 3.12.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do *demultiplexer* a ser configurado
Nome da variável de entrada	Indica o nome da variável de entrada
Nome da variável de saída	Indica o nome da variável de saída
Nome da variável de selecção	Indica o nome da variável de selecção
Número de *bits*	Indica de quantos *bits* deverá ser o módulo

Tabela 3.12: Tabela com os campos e descrição de configuração do *Demultiplexer*.

Note-se que as variáveis relativas ao número de *bits* de saída e de selecção são dependentes entre si. A relação entre ambas foi explicada no início desta secção.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada> : in STD_LOGIC;
3             <Variável de Selecçao> : in STD_LOGIC_VECTOR (<Número de bits de
                   selecçao> downto 0);
4             --
5             <Variável de Saída> : out STD_LOGIC_VECTOR (<Número de bits - 1>
                   downto 0));
6  end <Nome>;
7
8  architecture Behavioral of <Nome> is
9
10 begin
11
12     process(<Variável de Selecçao>, <Variável de Entrada>)
13     begin
14         case <Variável de Selecçao> is
15             when "00" => <Variável de Saída> <= <Variável de Entrada>;
16             when "01" => <Variável de Saída> <= <Variável de Entrada>;
17             when "10" => <Variável de Saída> <= <Variável de Entrada>;
18             when "11" => <Variável de Saída> <= <Variável de Entrada>;
19             -- ...  Continua dependendo do
20             -- número de bits inseridos
21             when others => <Variável de Saída> <= "X";
22         end case;
23     end process;
```

```
24
25 end Behavioral;
```

Listagem 3.44: Código VHDL da ferramenta por configurar. Este código corresponde ao Demultiplexer configurável

3.10.5.5 Operações Aritméticas, Somadores e Subtractores

Operações Aritméticas

A ferramenta disponibiliza uma forma de o utilizador poder obter a expressão de uma operação aritmética. Este componente é bastante interessante e útil, uma vez que o utilizador poderá trabalhar com variáveis com diferentes números de *bits* e diferentes tipos de sub-tipos sem se preocupar se a sintaxe está correcta. Caso o utilizador seleccione uma operação com sub-tipos ilegais em VHDL, o mesmo será alertado e obrigado a seleccionar a forma correcta.

Campos	Descrição
Operando A	O utilizador deverá indicar o nome do operando A
Operando B	O utilizador deverá indicar o nome do operando B
Nome do resultado	O utilizador deverá indicar o nome da variável do resultado
Número de *bits* do operando A	O utilizador deverá indicar de quantos *bits* o operando A deverá ser
Número de *bits* do operando B	O utilizador deverá indicar de quantos *bits* o operando B deverá ser
Número de *bits* do resultado	O utilizador deverá indicar de quantos *bits* a variável relativa ao resultado deverá ser
Sub-tipo do operando A	O utilizador deverá indicar se o operando A é do subtipo *std_logic_vector*, *unsigned* ou *signed*
Sub-tipo do operando B	O utilizador deverá indicar se o operando B é do subtipo *std_logic_vector*, *unsigned* ou *signed*
Sub-tipo do operando C	O utilizador deverá indicar se o operando C é do subtipo *std_logic_vector*, *unsigned* ou *signed*
Tipo de operação	O utilizador deverá indicar se a operação será uma adição, subtracção ou multiplicação

Tabela 3.13: Tabela com os campos e descrição de configuração das operações aritméticas.

```
1 use IEEE.NUMERIC_STD.ALL;
2
3 Signal <OPERAND_A> : <OPA_SUB_TYPE> (<NUMBER_OF_BITS_OPA> downto 0);
4 Signal <OPERAND_B> : <OPB_SUB_TYPE> (<NUMBER_OF_BITS_OPB> downto 0);
5 Signal <RESULT> : <RES_SUB_TYPE> (<NUMBER_OF_BITS_RES> downto 0);
6
7 <RESULT> <= <CMD_OPF>(<CMD_OPA>(<OPERAND_A>) <OPERATION>
    <CMD_OPB>(<OPERAND_B>));
```

Listagem 3.45: Código VHDL das Operações aritméticas por configurar.

Somador

Um somador consiste num circuito combinatório aritmético em que dois números binários de entrada são somados um ao outro [33]. Todos os somadores foram adaptados de [37].

Meio Somador de um *Bit*

O meio somador é um circuito com duas variáveis de entrada que correspondem a dois números binários. A soma de estas duas variáveis irá reproduzir-se num valor que corresponde à soma de ambos os números e de um *carry*. Cada um destes valores corresponde a uma variável de saída [33].

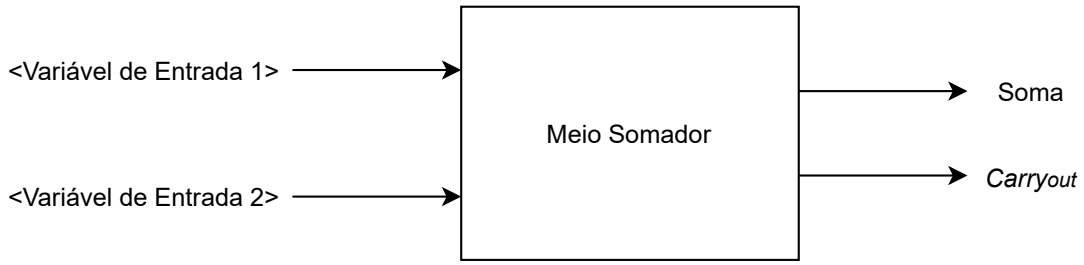

Figura 3.15: Representação do módulo correspondente ao meio somador. Adaptado de [33].

No meio somador de um *bit* é pedido que o utilizador preencha os três campos descritos na tabela 3.14 para que se possa modelar o código presente na listagem 3.46.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Variável de Saída	Indica o nome da variável de saída (*Sum Bit*)
Bit de *Carry*	Indica o nome da variável do *bit* de *carry*

Tabela 3.14: Tabela com os campos e descrição de configuração do meio somador.

```
entity <Nome> is
    Port ( <Variável de Entrada 1> : in STD_LOGIC;
           <Variável de Entrada 2> : in STD_LOGIC;
           --
           <Bit de Carry> : out STD_LOGIC;
           <Variável de Saída> : out STD_LOGIC);
end <Nome>;

architecture Behavioral of <Nome> is

begin

    process(<Variável de Entrada 1>, <Variável de Entrada 2>)
    begin
        if <Variável de Entrada 1> = '1' then
            <Variável de Saída> <= not <Variável de Entrada 2>;
            <Bit de Carry> <= <Variável de Entrada 2>;
        else
            <Variável de Saída> <= <Variável de Entrada 2>;
            <Bit de Carry> <= '0';
        end if;
    end process;

end Behavioral;
```

Listagem 3.46: Código VHDL da ferramenta por configurar. Este código corresponde ao meio Somador de um *bit* configurável.

Somador Completo de um *Bit*

O funcionamento do somador completo é bastante semelhante ao meio somador, apresentado na secção 3.10.5.5, com a diferença de que neste circuito para além das duas variáveis de entrada presentes num circuito meio somador, existe uma terceira que corresponde ao *carry* [33].

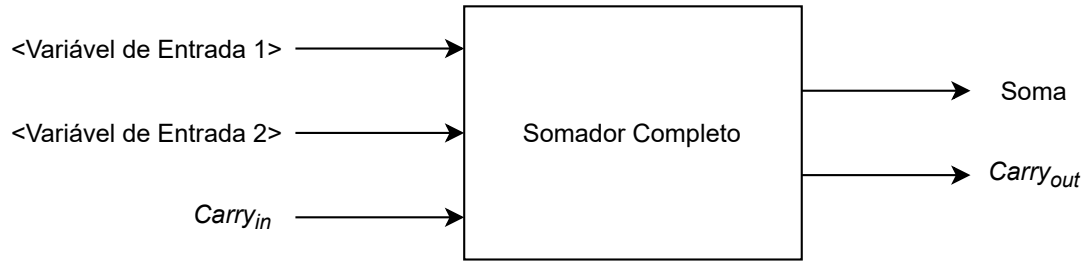

Figura 3.16: Representação do módulo correspondente ao somador completo. Adaptado de [33].

No somador completo de um *bit* é pedido que o utilizador preencha os três campos descritos na tabela 3.15 para que se possa modelar o código presente na listagem 3.47.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada 1> : in STD_LOGIC;
3             <Variável de Entrada 2> : in STD_LOGIC;
4             <Bit de Carry de entrada> : in STD_LOGIC;
5             --
6             <Bit de Carry> : out STD_LOGIC;
7             <Variável de Saída> : out STD_LOGIC);
8  end <Nome>;
9
10 architecture Behavioral of <Nome> is
11
12 begin
13
14     process(<Variável de Entrada 1>, <Variável de Entrada 2>, <Bit de Carry
           de entrada>)
15     begin
16         if <Variável de Entrada 1> = '0' then
17             if <Variável de Entrada 2> = '1' and <Bit de Carry de entrada> =
                   '1' then
18                 <Variável de Saída> <= '0';
19                 <Bit de Carry> <= '1';
20             else
21                 if <Variável de Entrada 2> = '0' and <Bit de Carry de
                       entrada> = '0' then
22                     <Variável de Saída> <= '0';
23                 else
24                     <Variável de Saída> <= '1';
25                 end if;
26                 <Bit de Carry> <= '0';
27             end if;
28         else
29             if <Variável de Entrada 2> = '1' or <Bit de Carry de entrada> =
                   '1' then
30                 <Bit de Carry> <= '1';
31             else
32                 <Bit de Carry> <= '0';
33             end if;
34             if (<Variável de Entrada 2> = '1' and <Bit de Carry de entrada> =
                   '1') or
35             (<Variável de Entrada 2> = '0' and <Bit de Carry de entrada> =
                   '0') then
36                 <Variável de Saída> <= '1';
37             else
38                 <Variável de Saída> <= '0';
39             end if;
40         end if;
41     end process;
42
43 end Behavioral;
```

Listagem 3.47: Código VHDL da ferramenta por configurar. Este código corresponde ao Somador completo de um *bit* configurável.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Bit de *Carry*	Indica o nome da variável do *bit* de *carry* de saída
Bit de *Carry* de entrada	Indica o nome da variável do *bit* de *carry* de entrada
Variável de Saída	Indica o nome da variável de saída

Tabela 3.15: Tabela com os campos e descrição de configuração do somador completo de um *bit*.

Somador de N-*bits*

O somador de N-*bits* representado na figura 3.17 com a descrição genérica em VHDL presente na listagem 3.48 permite a configuração por parte do utilizador através dos campos na tabela 3.16.

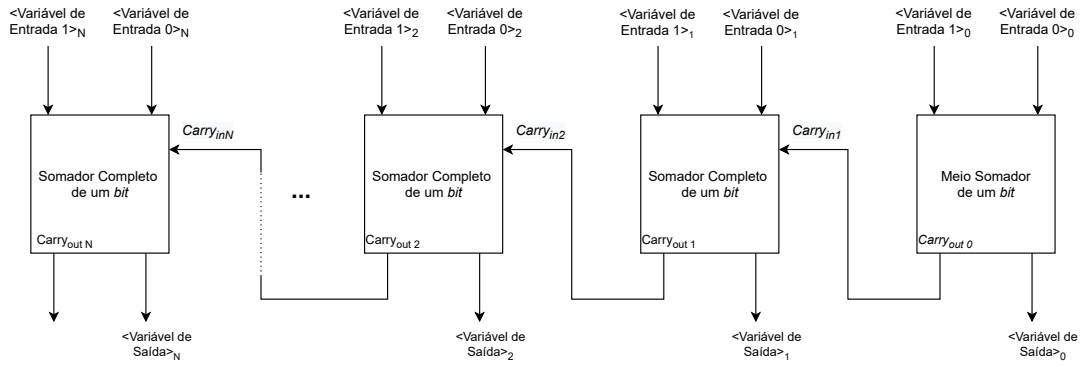

Figura 3.17: Representação do módulo correspondente ao somador de N-*bits*. Adaptado de [33].

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Variável de Saída	Indica o nome da variável de saída
Número de *bits*	Indica de quantos *bits* deverá ser o módulo

Tabela 3.16: Tabela com os campos e descrição de configuração do somador de N-*bits*.

```
1  entity <Nome>_adder_n_bits is
2      Port ( <Variável de Entrada 1> : in STD_LOGIC_VECTOR (<Número de bits>
           downto 0);
3            <Variável de Entrada 2> : in STD_LOGIC_VECTOR (<Número de bits>
               downto 0);
4            --
5            <Variável de Saída> : out STD_LOGIC_VECTOR (<Número de bits> downto
               0));
6  end <Nome>_adder_n_bits;
7
8  architecture Behavioral of <Nome>_adder_n_bits is
9      signal c0, c1, c2, ..., cN : STD_LOGIC;
10
11
12 begin
13
14     Half_Adder: entity work.half_adder port map (input_bit1 => <Variável de
           Entrada 1>(0), input_bit2 => <Variável de Entrada 2>(0), carry_bit =>
           c0, sum_bit => <Variável de Saída>(0));
15
16     Full_Adder0: entity work.full_adder port map (input_bit1 => <Variável de
           Entrada 1>(1), input_bit2 => <Variável de Entrada 2>(1),
           input_carry_bit => c0, carry_bit => c1, sum_bit => <Variável de
           Saída>(1));
17
```

```
18      Full_Adder1: entity work.full_adder port map (input_bit1 => <Variável de
            Entrada 1>(2), input_bit2 => <Variável de Entrada 2>(2),
            input_carry_bit => c1, carry_bit => c2, sum_bit => <Variável de
            Saída>(2));
19
20      -- ... Continua dependendo do número de bits
21
22      Full_AdderN: entity work.full_adder port map (input_bit1 => <Variável de
            Entrada 1>(N), input_bit2 => <Variável de Entrada 2>(N),
            input_carry_bit => cN-1, carry_bit => cN, sum_bit => <Variável de
            Saída>(N));
23
24  end Behavioral;
```

Listagem 3.48: Código VHDL por configurar. Este código corresponde ao Somador de N-*bits* configurável.

3.10.5.6 Subtractor

Um subtractor consiste num circuito combinatório aritmético em que dois números binários de entrada são subtraídos um ao outro [33]. Todos os subtractores foram adaptados de [37].

Meio Subtractor de um *Bit*

O meio subtractor terá como entradas duas variáveis que correspondem a dois valores binários. O módulo irá subtrair as duas entradas uma à outra e apresentar o resultado numa das variáveis de saída, a segunda variável de saída corresponde ao *borrow* [33].

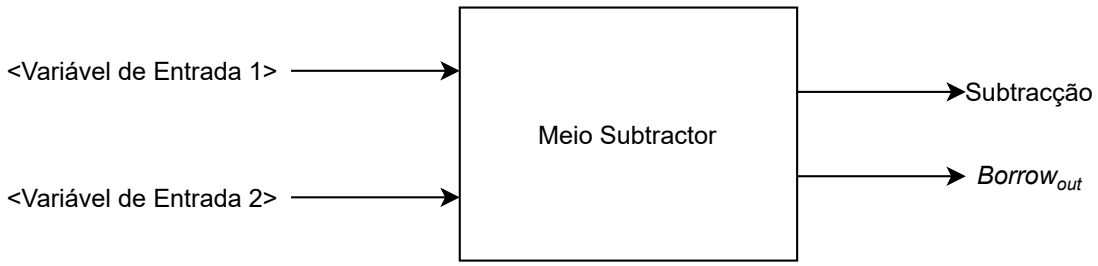

Figura 3.18: Representação do módulo correspondente ao meio subtractor. Adaptado de [33].

Na tabela 3.17 estão descritos os campos que o utilizador deverá preencher de modo a que a descrição em VHDL, presente na listagem 3.49, seja gerada.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Bit de diferença	Indica o nome da variável do *bit* de diferença de saída
Bit de *borrow*	Indica o nome da variável do *bit* de *borrow* de saída

Tabela 3.17: Tabela com os campos e descrição de configuração do meio subtractor.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada 1> : in STD_LOGIC;
3             <Variável de Entrada 2> : in STD_LOGIC;
4             --
5             <Bit de Borrow> : out STD_LOGIC;
6             <Variável de Saída> : out STD_LOGIC);
7  end <Nome>;
8
9  architecture Behavioral of <Nome> is
10
11 begin
12
13     process(<Variável de Entrada 1>, <Variável de Entrada 2>)
14     begin
15         if <Variável de Entrada 1> = '1' then
16             <Variável de Saída> <= not <Variável de Entrada 2>;
17             <Bit de Borrow> <= '0';
18         else
19             if <Variável de Entrada 2> = '1' then
20                 <Variável de Saída> <= '1';
21                 <Bit de Borrow> <= '1';
22             else
23                 <Variável de Saída> <= '0';
24                 <Bit de Borrow> <= '0';
25             end if;
26         end if;
27     end process;
28
```

```
29 end Behavioral;
```

Listagem 3.49: Código VHDL da ferramenta por configurar. Este código corresponde ao Meio subtractor de um *bit* configurável.

Subtractor Completo de um *Bit*

Ao contrário do meio subtractor, o subtractor completo, para além das duas entradas descritas na secção 3.10.5.6, tem também como entrada o *borrow* [33], como se pode ver na figura 3.18.

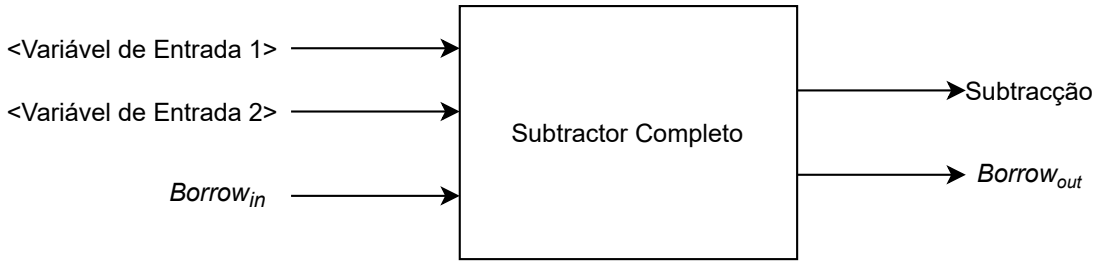

Figura 3.19: Representação do módulo correspondente ao subtractor completo. Adaptado de [33].

O módulo em questão, de forma semelhante ao meio subtractor referido na secção 3.10.5.6, é constituído por duas saídas. Uma destinada ao resultado da divisão e outra que corresponde ao *borrow*.
Na tabela 3.18 estão descritos os campos que o utilizador deverá preencher de modo a que a descrição em VHDL, presente na listagem 3.50, seja gerada.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Bit de *borrow* de entrada	Indica o nome da variável do *bit* de *borrow* de entrada
Bit de diferença	Indica o nome da variável do *bit* de diferença de saída
Bit de *borrow*	Indica o nome da variável do *bit* de *borrow* de saída

Tabela 3.18: Tabela com os campos e descrição de configuração do subtractor completo de um *bit*.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada 1> : in STD_LOGIC;
3             <Variável de Entrada 2> : in STD_LOGIC;
4             <Bit de borrow de entrada> : in STD_LOGIC;
5             --
6             <Bit de borrow> : out STD_LOGIC;
7             <Variável de Saída> : in STD_LOGIC);
8  end <Nome>;
9
10 architecture Behavioral of <Nome> is
11
12 begin
13
14     process(<Variável de Entrada 1>, <Variável de Entrada 2>, <Bit de borrow
           de entrada>)
15     begin
16         if <Variável de Entrada 1> = '0' then
17             if <Variável de Entrada 2> = '1' or <Bit de borrow de entrada> =
                   '1' then
18                 <Bit de borrow> <= '1';
19                 <Variável de Saída> <= '1';
20                 if <Variável de Entrada 2> = '1' and <Bit de borrow de
                       entrada> = '1' then
21                     <Variável de Saída> <= '0';
```

```
22                  end if;
23              else
24                  <Bit de borrow> <= '0';
25                  <Variável de Saída> <= '0';
26              end if;
27          else
28              if <Variável de Entrada 2> = '1' and <Bit de borrow de entrada> =
                    '1' then
29                  <Bit de borrow> <= '1';
30                  <Variável de Saída> <= '1';
31              else
32                  if <Variável de Entrada 2> = '0' and <Bit de borrow de
                        entrada> = '0' then
33                      <Bit de borrow> <= '0';
34                      <Variável de Saída> <= '1';
35                  else
36                      <Bit de borrow> <= '0';
37                      <Variável de Saída> <= '0';
38                  end if;
39              end if;
40
41          end if;
42      end process;
43
44  end Behavioral;
```

Listagem 3.50: Código VHDL da ferramenta por configurar. Este código corresponde ao Subtractor completo de um *bit* configurável.

Subtractor de N-*bits*

O subtractor de N-*bits* representado na figura 3.20 com a descrição genérica em VHDL presente na listagem 3.51 permite a configuração por parte do utilizador através dos campos na tabela 3.19.

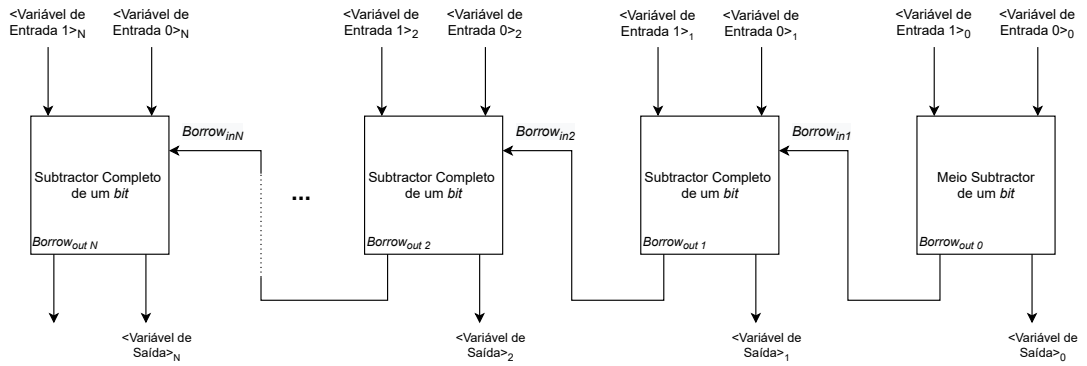

Figura 3.20: Representação do módulo correspondente ao subtractor de N-*bits*. Adaptado de [33].

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Variável de Saída	Indica o nome da variável de saída
Número de *bits*	Indica de quantos *bits* deverá ser o módulo

Tabela 3.19: Tabela com os campos e descrição de configuração do subtractor de N-*bits*.

```
1  entity <Nome>_subtractor_n_bits is
2      Port ( <Variável de Entrada 1> : in STD_LOGIC_VECTOR (<Número de bits>
           downto 0);
3            <Variável de Entrada 2> : in STD_LOGIC_VECTOR (<Número de bits>
               downto 0);
4            --
5            <Variável de Saída> : out STD_LOGIC_VECTOR (<Número de bits> downto
               0));
6  end <Nome>_subtractor_n_bits;
7
8  architecture Behavioral of <Nome>_subtractor_n_bits is
9      signal b0, b1, b2, ..., bN : STD_LOGIC;
10
11
12 begin
13
14     Half_Subtractor: entity work.half_subtractor port map (input_bit1 =>
           <Variável de Entrada 1>(0), input_bit2 => <Variável de Entrada 2>(0),
           borrow_bit => b0, sub_bit => <Variável de Saída>(0));
15
16     Full_Subtractor0: entity work.full_subtractor port map (input_bit1 =>
           <Variável de Entrada 1>(1), input_bit2 => <Variável de Entrada 2>(1),
           input_carry_bit => b0, borrow_bit => b1, sub_bit => <Variável de
           Saída>(1));
17
```

```
18      Full_Subtractor1: entity work.full_subtractor port map (input_bit1 =>
            <Variável de Entrada 1>(2), input_bit2 => <Variável de Entrada 2>(2),
            input_carry_bit => b1, borrow_bit => b2, sub_bit => <Variável de
            Saída>(2));
19
20      -- ... Continua dependendo do número de bits
21
22      Full_SubtractorN: entity work.full_subtractor port map (input_bit1 =>
            <Variável de Entrada 1>(N), input_bit2 => <Variável de Entrada 2>(N),
            input_carry_bit => bN-1, borrow_bit => bN, sub_bit => <Variável de
            Saída>(N));
23
24  end Behavioral;
```

Listagem 3.51: Código VHDL por configurar. Este código corresponde ao Subtractor de N-*bits* configurável.

3.10.5.7 Somador/Subtractor de alto nível

Nesta secção dá-se a conhecer um módulo presente na ferramenta que consiste num somador/subtractor de alto nível. De alto nível porque existe uma maior abstração na descrição do código VHDL, presente na listagem 3.52, relativamente aos módulos apresentados nas secções 3.10.5.6 e 3.10.5.5 que tratam da manipulação ao nível de um *bit*. O somador/subtractor de alto nível foi adaptado de [38].
Na tabela 3.20 estão ilustrados os campos que o utilizador deverá completar.

```
1   entity <Nome> is
2       Port ( <Variável de Entrada 1> : in <Tipo de dados>(<Número de bits - 1>
            downto 0);
3              <Variável de Entrada 2> : in <Tipo de dados>(<Número de bits - 1>
                   downto 0);
4              --
5              <Variável de Saída> : out <Tipo de dados>(<Número de bits - 1>
                   downto 0));
6       end <Nome>;
7
8   architecture Behavioral of <Nome> is
9
10  begin
11
12      <Variável de Saída> <= <Variável de Entrada 1> + <Variável de Entrada 2>;
13
14  end Behavioral;
```

Listagem 3.52: Somador/Subtractor de alto nível.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo a ser configurado
Variável de Entrada 1	Indica o nome da variável de entrada
Variável de Entrada 2	Indica o nome da variável de entrada
Variável de Saída	Indica o nome da variável de saída
Número de *bits*	O utilizador deverá indicar o número de *bits* que o módulo deverá ter
Tipo de dados	O utilizador deverá indicar se as variáveis irão ser *unsigned* ou *signed*

Tabela 3.20: Tabela com os campos e descrição de configuração do somador/subtractor de alto nível.

3.10.5.8 Codificador e Descodificador

Codificador

O codificador é um circuito lógico que tem 2^n entradas e n saídas, como é referido no livro [33]. O codificador converte o que foi recebido nas entradas num código binário equivalente na saída.

Campos	Descrição
Nome do módulo	O utilizador deverá indicar o nome do codificador a ser configurado
Nome da variável de entrada	O utilizador deverá indicar o nome da variável de entrada
Nome da variável de saída	O utilizador deverá indicar o nome da variável de saída
Número de *bits* de entrada	Indica quantos *bits* deverá ter a variável de entrada

Tabela 3.21: Tabela com os campos e descrição de configuração do codificador.

Neste módulo o utilizador irá preencher os campos presentes na tabela 3.21 e consequentemente dentro do *process* irão ser geradas combinações entre o valor das entradas e o valor que as saídas deverão apresentar. Na listagem 3.53 está o código VHDL genérico correspondente ao codificador configurável.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada> : in STD_LOGIC_VECTOR (<Número de bits>
           downto 0);
3             --
4             <Variável de Saída>) : out STD_LOGIC_VECTOR (<Número de bits de
                  selecçao> downto 0);
5  end <Nome>;
6
7  architecture Behavioral of <Nome> is
8
9  begin
10
11     process(<Variável de Entrada>)
12     begin
13         case <Variável de Entrada> is
14             when "1000" => <Variável de Saída> <= "00";
15             when "0100" => <Variável de Saída> <= "01";
16             when "0010" => <Variável de Saída> <= "10";
17             when "0001" => <Variável de Saída> <= "11";
18             -- ...  Continua dependendo do
19             -- número de bits inseridos
20             when others => <Variável de Saída> <= "ZZ";
21         end case;
22     end process;
23
```

```
24 end Behavioral;
```

Listagem 3.53: Código VHDL da ferramenta por configurar. Este código corresponde ao Codificador configurável.

Descodificador

O descodificador, ao contrário do Codificador, é um circuito lógico que tem n entradas e 2^n saídas, como é referido no livro [33]. O descodificador converte o que foi recebido nas entradas num código binário equivalente na saída. Neste módulo o utilizador irá

Campos	Descrição
Nome	O utilizador deverá indicar o nome do descodificador a ser configurado
Nome da variável de entrada	O utilizador deverá indicar o nome do codificador a ser configurado
Nome da variável de saída	O utilizador deverá indicar o nome da variável de saída
Número de *bits* de saída	Indica quantos *bits* deverá ter a variável de saída

Tabela 3.22: Tabela com os campos e descrição de configuração do descodificador.

preencher os campos presentes na tabela 3.22 e consequentemente dentro do *process* irão ser geradas combinações entre o valor das entradas e o valor que as saídas deverão apresentar. Na listagem 3.54 está o código VHDL genérico correspondente ao descodificador configurável.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada> : in STD_LOGIC_VECTOR (<Número de bits de
           selecçao> downto 0);;
3             --
4             <Variável de Saída> : out STD_LOGIC_VECTOR (<Número de bits>
                  downto 0););
5  end <Nome>;
6
7  architecture Behavioral of <Nome> is
8
9  begin
10
11     process(<Variável de Entrada>)
```

```
12      begin
13          case <Variável de Entrada> is
14              when "00" => <Variável de Saída> <= "0001";
15              when "01" => <Variável de Saída> <= "0010";
16              when "10" => <Variável de Saída> <= "0100";
17              when "11" => <Variável de Saída> <= "1000";
18              -- ...  Continua dependendo do
19              -- número de bits inseridos
20          end case;
21      end process;
22
23  end Behavioral;
```

Listagem 3.54: Código VHDL da ferramenta por configurar. Este código corresponde ao Descodificador configurável.

3.10.5.9 Máquinas de Estados

Módulo da Máquina de Estados

Importa relembrar que as máquinas de estados, como os autores de [39] referem, é um modelo bastante útil no desenvolvimento de sistemas que consistem em sequências bem definidas. As máquinas de estado, em suma, são uma técnica de modulação de circuitos lógicos sequenciais.
Para uma configuração rápida e com o menor número de erros possíveis, a ferramenta pede que o utilizador preencha os campos presentes na tabela 3.22. A ferramenta irá gerar o código VHDL de acordo com a informação fornecida pelo o utilizador. O código genérico a ser gerado encontra-se na listagem 3.55.
Importa referir que os *procedures*, como o *maq1_proc_s0* e *maq1_proc_s1*, permitem que o utilizador mais facilmente descreva o comportamento de cada estado. Esta divisão irá organizar e auxiliar o projectista na descrição em VHDL.
Relativamente aos restante processos que compõem a máquina de estados, a descrição em VHDL sobre o, *clock* síncrono e assíncrono encontra-se nas listagens 3.57 e 3.56, respectivamente; sobre o processo relativo ao próximo estado encontra-se na listagem 3.58 e o processo relativo à saída da máquina de estados na listagem 3.59.

Campos	Descrição
Nome	O utilizador deverá indicar o nome da máquina de estados a ser configurada
Nome do *clock*	O utilizador deverá indicar o nome do *clock*
Nome do *reset*	O utilizador deverá indicar o nome do *reset*
Identificação do estado inicial	O utilizador deverá identificar o estado inicial. Caso alguma condição falhe, dever-se-á regressar a este estado
Tipo de *reset*	O utilizador deverá indicar se a máquina de estados terá um *reset* assíncrono ou síncrono
Tipo de *Edge*	O utilizador deverá escolher entre *rising edge* e *falling edge*
Número de estados	O utilizador deverá indicar o quantos estados deverá ter a máquina de estados

Tabela 3.23: Tabela com os campos e descrição de configuração da máquina de estados.

```
1  entity <Nome> is
2      Port ( <Variável de Entrada>: in STD_LOGIC;
3             <Reset>: in STD_LOGIC;
4             <Clock>: in STD_LOGIC;
5             --
6             <Variável de Saída>: out STD_LOGIC);
7  end <Nome>;
8
9  architecture Behavioral of <Nome> is
10
11     type state_type is ( s_0, ..., s_N);
12     signal estado_actual, estado_seguinte: state_type;
13
14     procedure maq_procedure_s0 (estado_actual, estado_seguinte) is
15     begin
16         -- Comportamento a ser descrito
17     end maq_procedure_s0;
18
19     procedure maq_procedure_sN (estado_actual, estado_seguinte) is
20     begin
21         -- Comportamento a ser descrito
```

```
22      end maq_procedure_sN;
23
24  begin
25
26      -- Processo relativo ao clock síncrono ou assíncrono
27      -- será gerado aqui
28
29      -- NEXT_STATE_PROCESS será gerado aqui
30
31      -- OUTPUT_PROCESS será gerado aqui
32
33  end Behavioral;
```

Listagem 3.55: Máquina de estados de *N* estados configurável sem o processo de *clock*.

No caso de o utilizador escolher *reset* assíncrono, o processo relativo ao *clock* assíncrono será gerado. Se for escolhido *clock* com *reset* síncrono, será gerado o processo síncrono.

```
1
2   -- No caso de o utilizador escolher
3   -- reset assíncrono
4   CLOCK_PROCESS_ASYNC: process(<Clock>, <Reset>)
5   begin
6       if <Reset> = '1' then
7           estado_actual <= s0; -- No caso de ser s0
8       else
9           estado_actual <= estado_seguinte;
10      end if;
11  end process;
```

Listagem 3.56: Processo de *clock* assíncrono da máquina de estados de *N* estados configurável.

```
1
2   -- No caso de o utilizador escolher
3   -- reset síncrono
4   CLOCK_PROCESS_SYNC: process(<Clock>)
5   begin
6       if <Tipo de Edge>(<Clock>) then
7           if <Reset> = '1' then
8               estado_actual <= s0; -- No caso de ser s0
9           else
10              estado_actual <= estado_seguinte;
11          end if;
12      end if;
13  end process;
```

Listagem 3.57: Processo de *clock* síncrono da máquina de estados de *N* estados configurável.

O processo *NEXT_STATE_PROCESS* juntamente com cada uma das funções dentro da palavra-chave *case* (*maq_procedure_s0()*, *maq_procedure_s1()*, etc) é responsável por definir o próximo estado, através da variável do estado actual e de entrada.

```
1  NEXT_STATE_PROCESS: process (<Clock>)
2  begin
3      estado_seguinte<=estado_actual;
4      case estado_actual is
5          when s_0=>
6              maq_procedure_s0(estado_actual, estado_seguinte);
7          -- ... Continua,
8          -- dependendo do número
9          -- de estados inseridos
10         when s_N =>
11             maq_procedure_sN(estado_actual, estado_seguinte);
12     end case;
13 end process;
```

Listagem 3.58: Processo relativo ao próximo estado da máquina de estados de *N* estados configurável.

O processo *OUTPUT_PROCESS* tem como finalidade dar ao utilizador a facilidade de definir um valor para a variável de saída consoante as condições que se considere necessárias.

```
1  OUTPUT_PROCESS: process( )
2  begin
3
4      -- <Variável de Saída><='0';
5
6      -- <Variável de Saída><='1';
7
8  end process;
```

Listagem 3.59: Processo relativo à saída da máquina de estados configurável de *N* estados.

Processo de máquina de estados

O processo de máquina de estados tem como finalidade apresentar uma versão mais simples do que foi apresentado em 3.10.5.9, facilitando e permitindo que os utilizadores menos experientes compreendam melhor, não apenas o funcionamento de uma máquina de estados mas também da VHDL. O módulo presente em 3.10.5.9 requer um nível mais avançado em linguagens de descrição de *hardware*, por parte do utilizador.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do processo da máquina de estados a ser configurada
Nome do *clock*	O utilizador deverá indicar o nome do *clock*
Nome do *reset*	O utilizador deverá indicar o nome do *reset*
Identificação do estado inicial	O utilizador deverá identificar o estado inicial. Caso alguma condição falhe, dever-se-á regressar a este estado
Tipo de *reset*	O utilizador deverá indicar se a máquina de estados terá um *reset* assíncrono ou síncrono
Tipo de *Edge*	O utilizador deverá escolher entre *rising edge* e *falling edge*
Número de estados	O utilizador deverá indicar o quantos estados deverá ter a máquina de estados

Tabela 3.24: Tabela com os campos e descrição de configuração do processo da máquina de estados.

```
1  type <NAME>_type is (<AUTO_GENERATE_STATES>);
2  signal <NAME>_state: <NAME>_type;
3
4  <NAME>_process: process(<CLOCK>)
5  begin
6    if <EDGE_TYPE>(<CLOCK>) then
7      if <RESET> = '1' then
8        <NAME>_state <= s_0;
9      else
10       case (<NAME>_state) is
11         <AUTO_GENERATE>
12       end case;
13     end if;
14   end if;
15 end process;
```

Listagem 3.60: Código VHDL da ferramenta por configurar. Este código corresponde ao Processo da máquina de estados.

3.10.5.10 Declaração de expressões

Expressões lógicas e relacionais

As configuração expressões lógicas e relacionais são oferecidas na ferramenta, agilizando e facilitando o uso e aprendizagem dos operadores lógicos (*and*, *or*, *xor*, *nand*, *nor* e *xnor*) e relacionais (*equal*, *not equal*, *less than*, *greater than*, *equal or less than* e *equal or greater than*).

Campos	Descrição
Nome do operando 1	O utilizador deverá indicar o nome do primeiro operando
Nome do operando 2	O utilizador deverá indicar o nome do segundo operando
Operador	O utilizador deverá indicar qual o operador que deseja

Tabela 3.25: Tabela com os campos e descrição de configuração das expressões lógicas e relacionais.

```
1 <OPERAND1> <OPERATOR> <OPERAND2>
```

Listagem 3.61: Expressão lógica e relacional por configurar.

Expressões de atribuições

A configuração de expressões de atribuição presente na ferramenta, ao ensinar a sua sintaxe, torna-se proveitosa para utilizadores que estejam a dar os primeiros passos em VHDL.

```
1 <SIGVAR> <ASSIGMENT_TYPE> <OBJECT>
```

Listagem 3.62: Expressão de atribuição por configurar.

Campos	Descrição
Nome do sinal ou variável	O utilizador deverá indicar o nome do sinal ou variável
Nome do objecto	O utilizador deverá indicar o valor a ser atribuído
Tipo de atribuição	O utilizador deverá especificar se deseja optar por um sinal ou uma variável

Tabela 3.26: Tabela com os campos e descrição de configuração das expressões de atribuição.

3.10.6 Ficheiros UCF e XDC

A ferramenta oferece a possibilidade de gerar os ficheiros UCF (*User Constraint File*) e XDC (*Xilinx Design Constraints file*). Estes ficheiros permitem que o utilizador configure, nos ambientes de desenvolvimento ISE e Vivado da Xilinx, os *pins* a serem usados.

3.10.6.1 Ficheiro UCF

A tabela 3.27 contém os campos de configuração do ficheiro UCF e na listagem 3.63 está presente um exemplo genérico com os campos do ficheiro UCF por configurar.

Campos	Descrição
Nome	O utilizador deverá indicar o *net name*
Número de *bits*	O utilizador deverá indicar o número de *bits* da *net*

Tabela 3.27: Tabela com os campos e descrição de configuração do ficheiro UCF.

```
1 NET "<NET_NAME>" LOC = "LOCNAME"
```

Listagem 3.63: Campos do ficheiro UCF por configurar.

3.10.6.2 Ficheiro XDC

A tabela 3.28 contém os campos de configuração do ficheiro XDC e na listagem 3.64 está presente um exemplo genérico com os campos do ficheiro XDC por configurar.

Campos	Descrição
Nome	O utilizador deverá indicar o *net name*
Número de *bits*	O utilizador deverá indicar o número de *bits* da *net*

Tabela 3.28: Tabela com os campos e descrição de configuração do ficheiro XDC.

```
1 NET "<NET_NAME>" LOC = "LOCNAME" | IOSTANDARD = LVTTL
```

Listagem 3.64: Campos do ficheiro XDC por configurar.

3.10.7 Outros módulos

3.10.7.1 *Empty VHDL module*

O *Empty VHDL module* é um módulo bastante simples que pretende dar a conhecer a sintaxe e forma de um módulo em VHDL.

Campos	Descrição
Nome	O utilizador deverá indicar o do módulo

Tabela 3.29: Tabela com os campos e descrição de configuração do *Empty VHDL module*.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  --additional LIBRARY DECLARATIONS here
4
5  entity <NAME> is
6
7  --add PORT here
8  end <NAME>;
9
10 architecture Behavioral of <NAME> is
11
12 --TYPES, SIGNALS, FUNCTIONS, PROCEDURES, and COMPONENTS here
13
14 begin
15
16 --COMPONENTS INSTANTIATION, PROCESSES, SIGNALS ASSIGNMENT, and OUTPUTS
       ASSIGNMENT here
```

```
17
18 end Behavioral;
```

Listagem 3.65: Código VHDL do *Empty VHDL module* por configurar.

4

Validação

Este capítulo tem como objectivo apresentar as validações efectuadas através das interacções entre o utilizador e a ferramenta. Na duas primeiras secções são apresentados os testes efectuados à ferramenta, bem como o inquérito realizado. Por fim, são apresentados um conjunto de módulos desenvolvidos por um dos participantes do inquérito, demonstrando novas possibilidades e potencialidades para futuras versões da ferramenta.

4.1 Testes

4.1.1 Abertura dos módulos

Neste teste pretende-se averiguar como a ferramenta reage quando um utilizador clica no módulo que pretende configurar. Na figura 4.1 está representado o ambiente inicial que deverá ser apresentado ao utilizador aquando a inicialização da página Web.

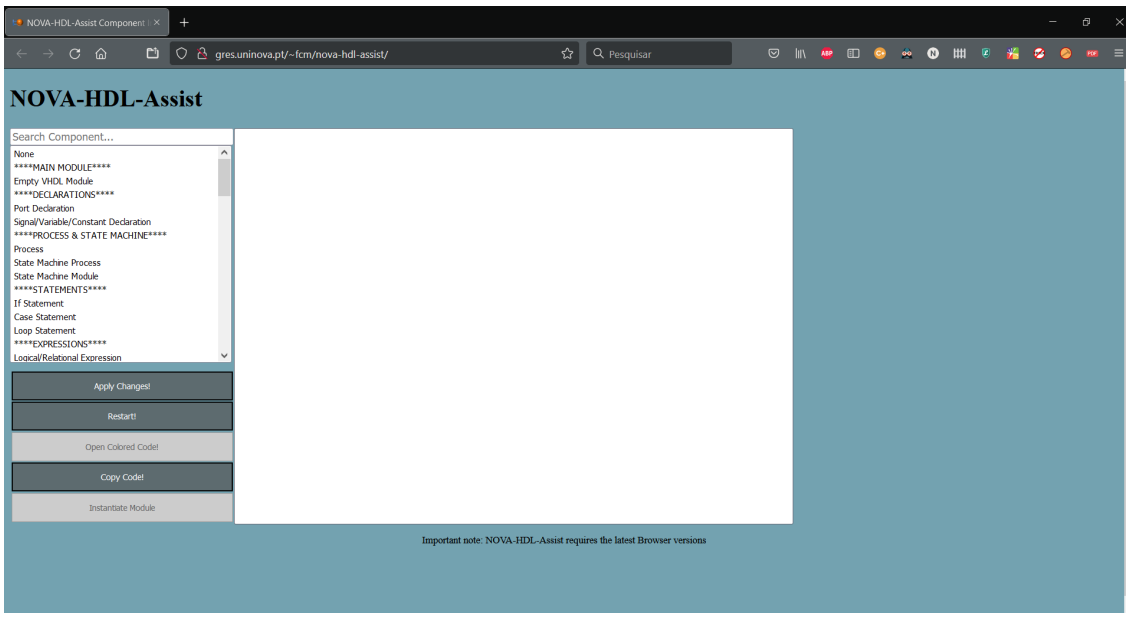

Figura 4.1: Ambiente inicial do NOVA–HDL–Assist.

O teste foi realizado para três componentes, o módulo *State Machine Process*, representado na figura 4.2, o módulo *Encoder*, representado na figura 4.3 e o componente relativo às operações aritméticas, o *Arithmetic Operation*, visível na figura 4.4.

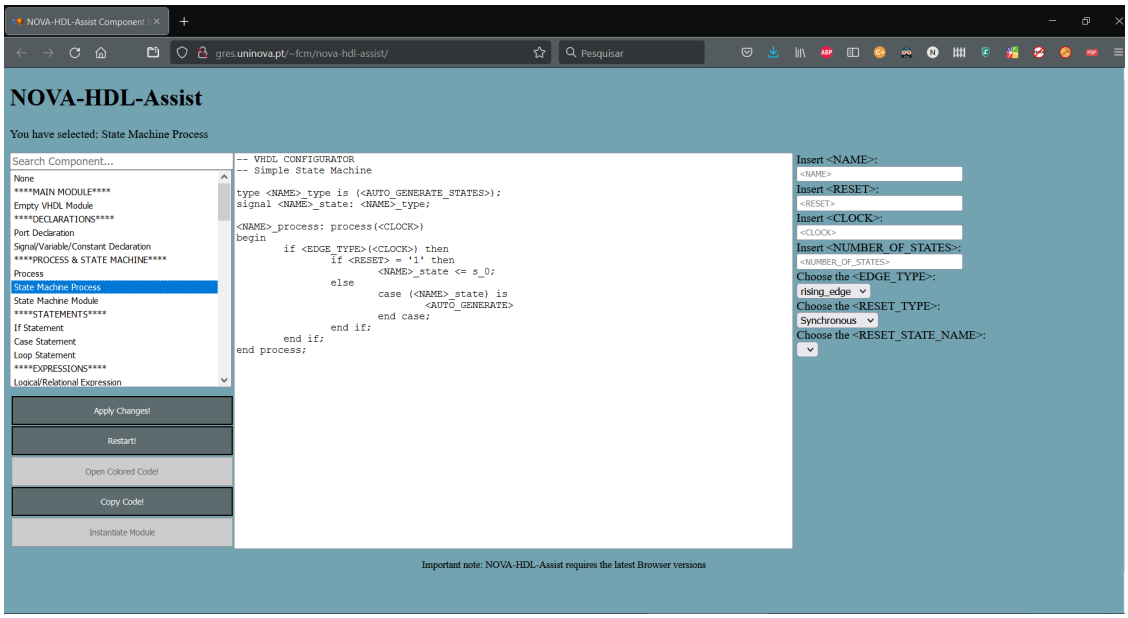

Figura 4.2: Teste aquando a selecção do *State Machine Process*.

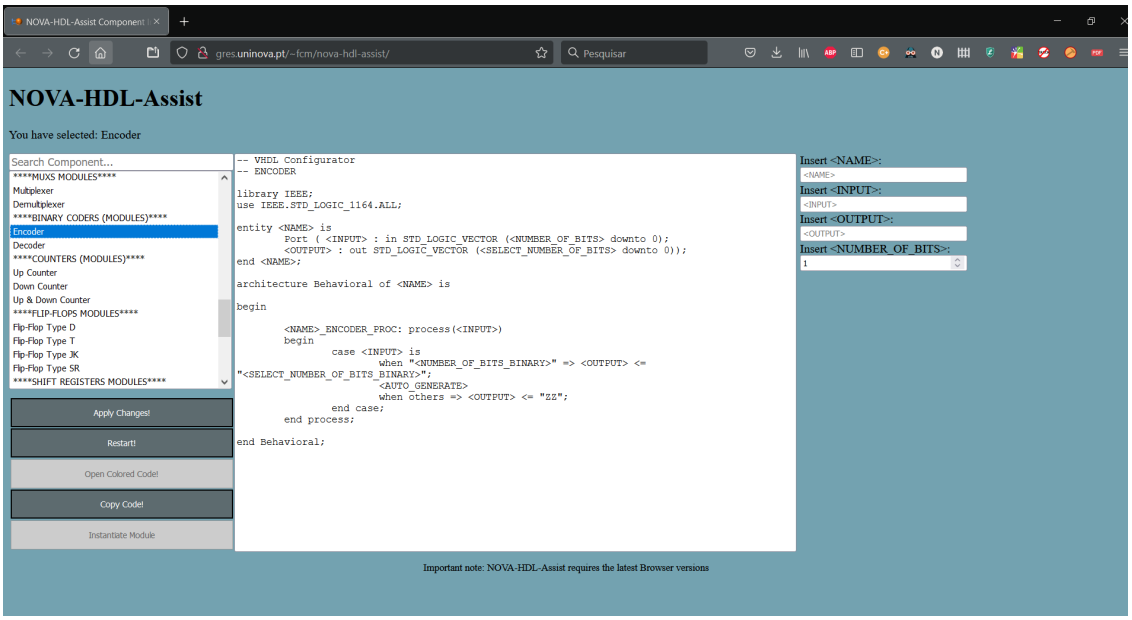

Figura 4.3: Teste aquando a selecção do *Encoder*.

Na figura 4.5 está demonstrado uma situação em que é seleccionado o módulo por parte do utilizador, contudo este ou não é descarregado ou é descarregado parcialmente da base de dados. A causa deste erro advém principalmente da má ligação de *internet* por

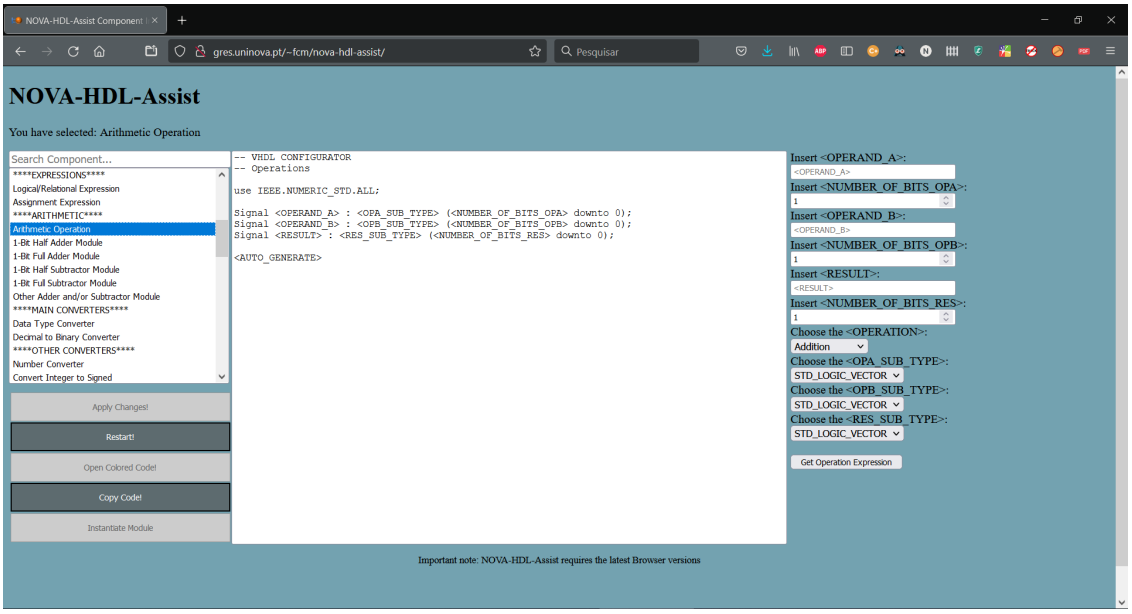

Figura 4.4: Teste aquando a selecção do *Arithmetic Operation*.

parte do utilizador, ou por problemas no servidor. Realçar que a vermelho na figura 4.5, observa-se a mensagem de erro.

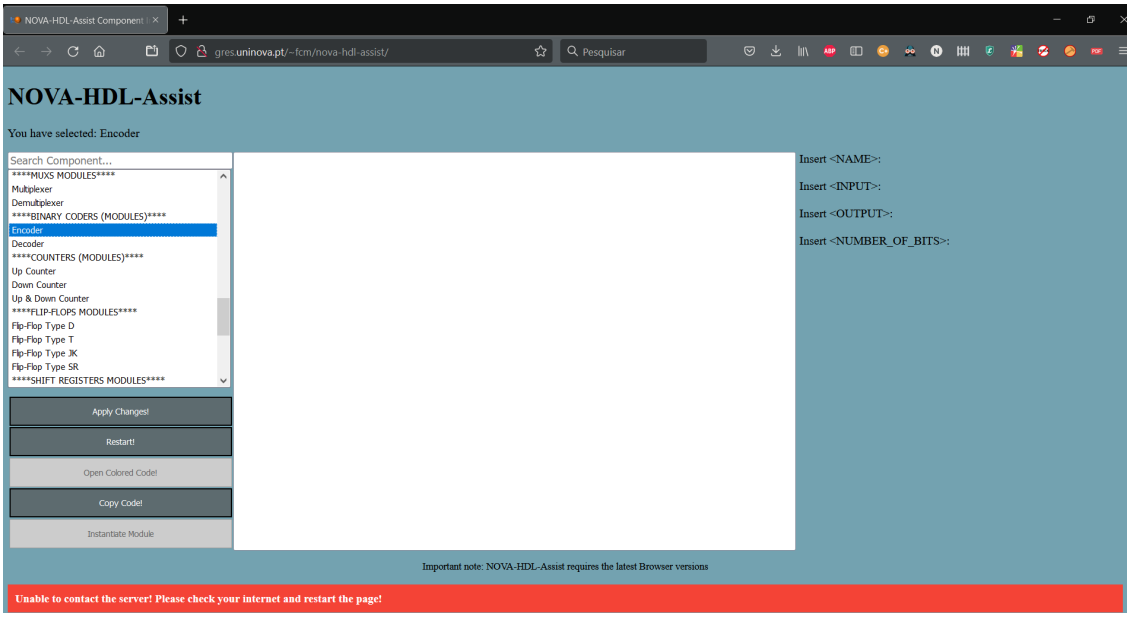

Figura 4.5: Teste aquando a selecção de um módulo com erro de falha da conexão de *internet*, ou falha do servidor.

4.1.2 Configuração dos módulos

Dado que o módulo *State Machine Module* é um dos mais complexos na ferramenta, este foi escolhido em primeiro lugar para se testar a configuração de módulos por parte do utilizador. Nas figuras 4.6, 4.7 e 4.8, observa-se o módulo configurado (após o clique em *Apply Changes!*). As diferentes figuras correspondem às diferentes selecções nos items de selecção.
De seguida foram também seleccionados, aleatoriamente, o módulo *Flip-Flop* do tipo D e o módulo *Multiplexer*. O teste relativo ao *Flip-Flop* tipo D está presente nas figuras 4.9 e 4.10 e 4.11. O teste realizado com recurso ao *Multiplexer* encontra-se nas figuras 4.12 e 4.13.
De forma a melhor elucidar o bom funcionamento da ferramenta e o sucesso dos testes, em algumas figuras são apresentadas caixas com cores em que cada cor faz corresponder o valor colocado pelo utilizador nos campos de configuração, e a alteração realizada pela ferramenta que é visível no painel onde o código configurado é mostrado.
Note-se ainda que em alguns módulos dado existirem várias possibilidades de configuração, são apresentadas figuras com todas a possibilidades de configuração, como sensibilidade ao flanco ascendente ou descendente, e *reset* síncrono ou assíncrono. Por exemplo, na figura 4.6 é apresentado o módulo *State Machine Process* com sensibilidade do *clock* ao flanco ascendente e com *reset* síncrono, e na figura 4.7 é apresentado o módulo *State Machine Process* com sensibilidade do *clock* ao flanco descendente e com *reset* síncrono.
Importa referir que na figura 4.6, no lado esquerdo a laranja, está assinalado o número de estados que o utilizador deseja para o seu módulo e, no centro a laranja, está o código auto-gerado pela ferramenta que corresponde ao número de estados indicados pelo utilizador.

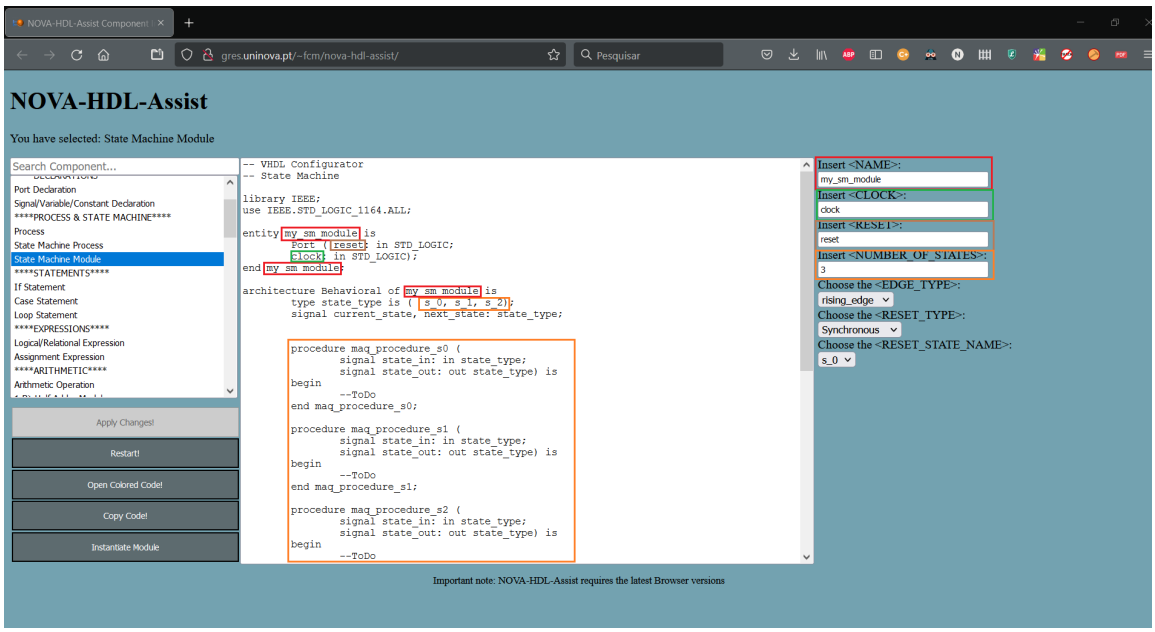

Figura 4.6: Teste à configuração do módulo *State Machine Process* com *rising edge* e *reset* síncrono.

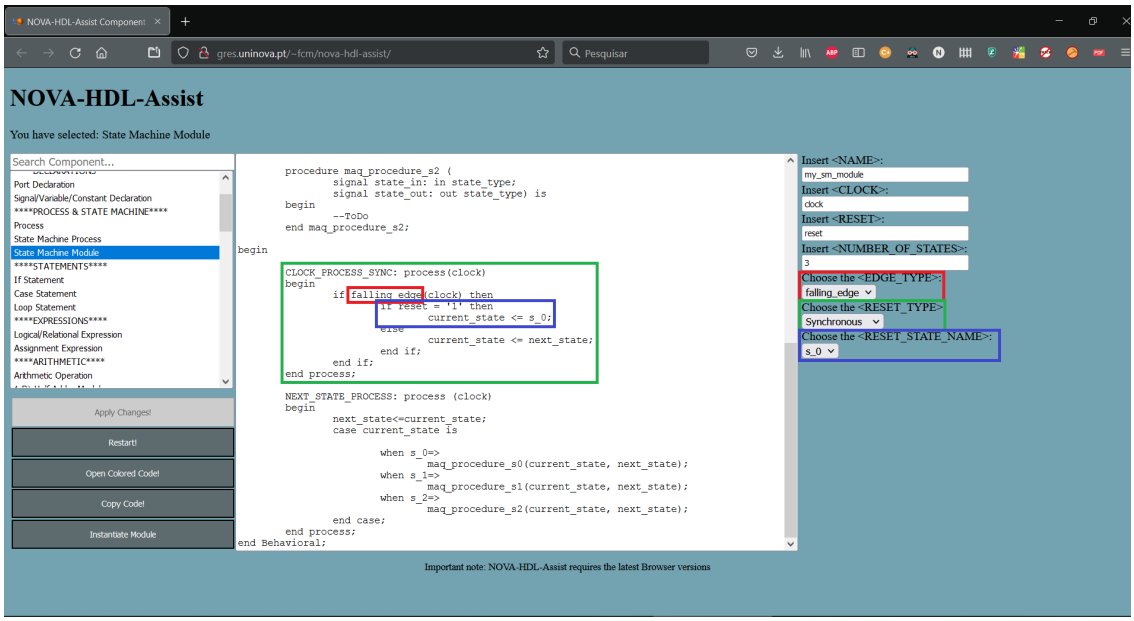

Figura 4.7: Teste à configuração do módulo *State Machine Process* com *falling edge* e *reset* síncrono.

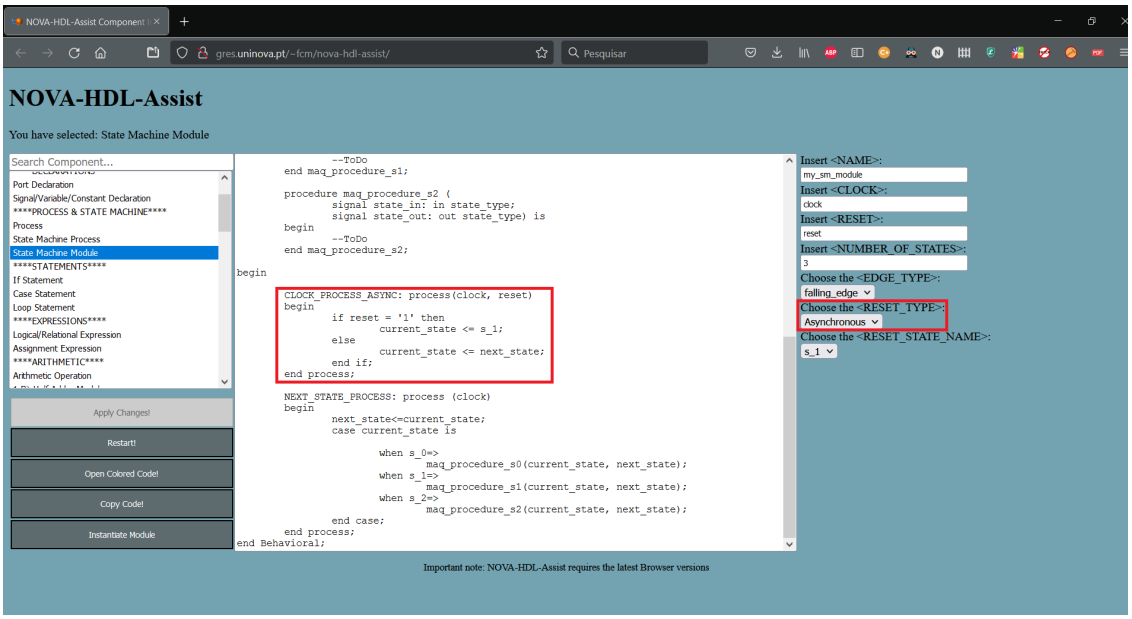

Figura 4.8: Teste à configuração do módulo *State Machine Process* com *reset* assíncrono. Note-se que como o *reset* é assíncrono, o valor do *edge type* é desprezável.

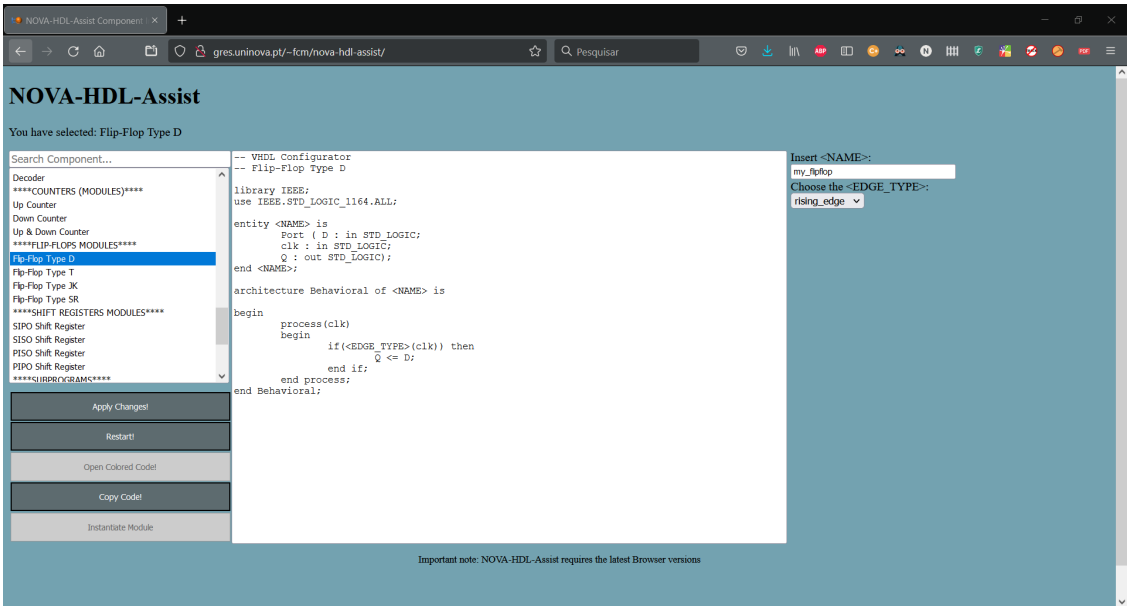

Figura 4.9: Teste à configuração do módulo *Flip-Flop* do tipo D com sensibilidade ao flanco ascendente do *clock*. Antes de se clicar em *Apply Changes!*.

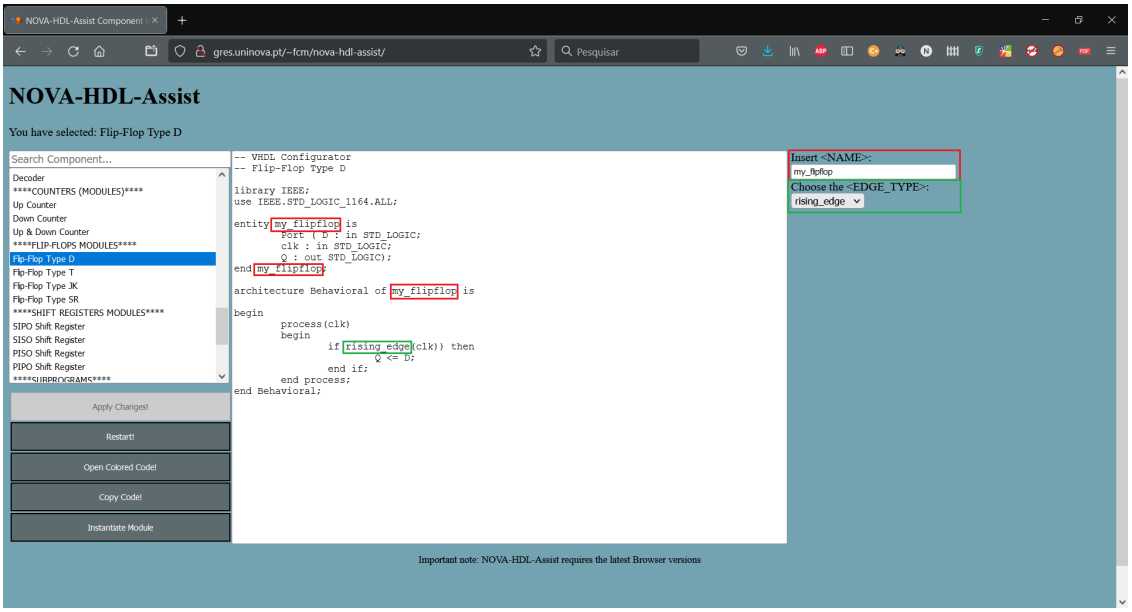

Figura 4.10: Teste à configuração do módulo *Flip-Flop* do tipo D com sensibilidade ao flanco ascendente do *clock*. Depois de se clicar em *Apply Changes!*.

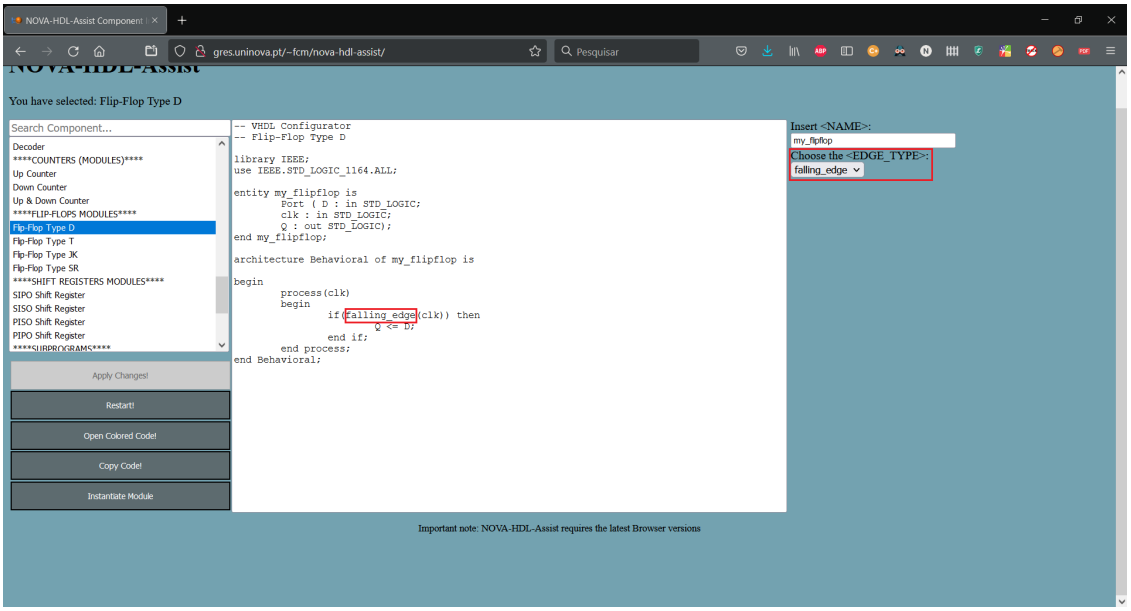

Figura 4.11: Teste à configuração do módulo *Flip-Flop* do tipo D com sensibilidade ao flanco descendente do *clock*. Depois de se clicar em *Apply Changes!*.

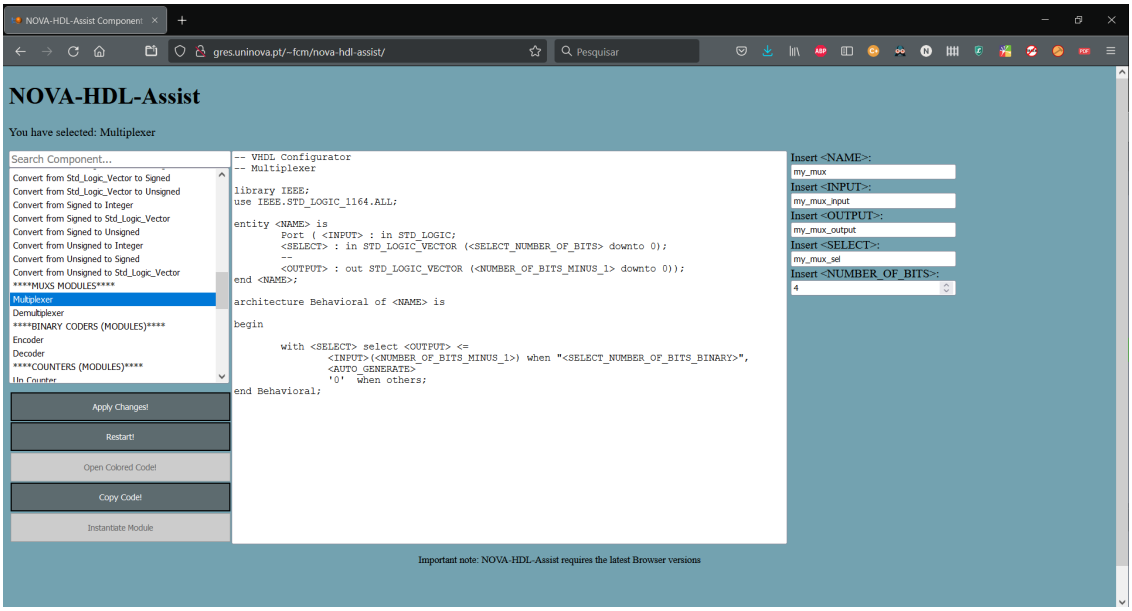

Figura 4.12: Teste à configuração do módulo *Multiplexer*. Antes de se clicar em *Apply Changes!*.

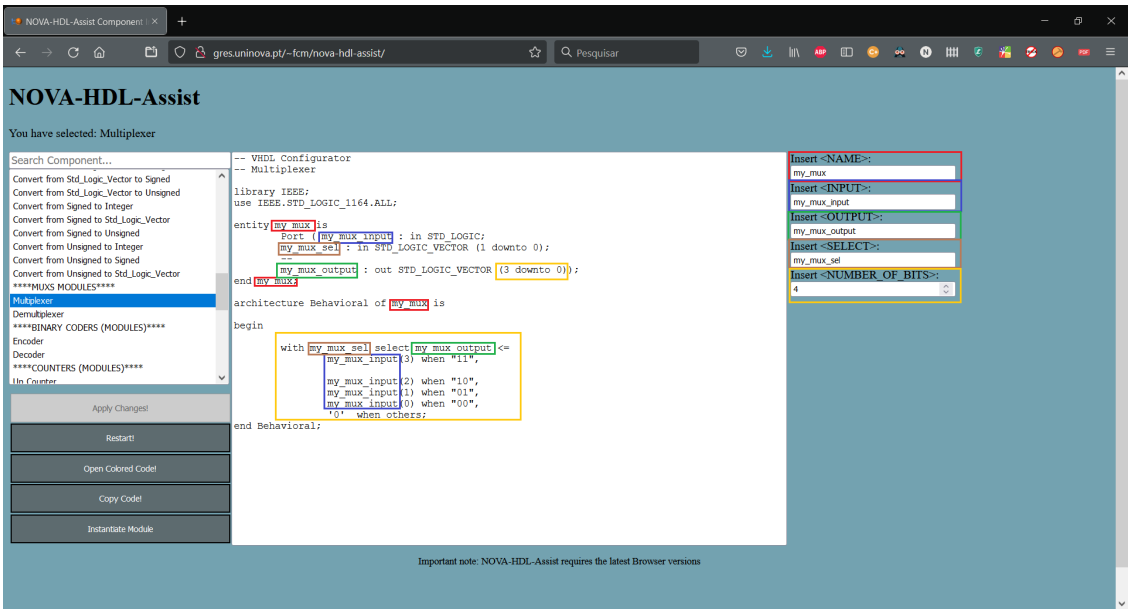

Figura 4.13: Teste à configuração do módulo *Multiplexer*. Depois de se clicar em *Apply Changes!*.

Na figura 4.14 observa-se que a ferramenta mostra com sucesso uma mensagem de erro quando o utilizador, ao configurar um módulo, introduz uma palavra reservado em VHDL.

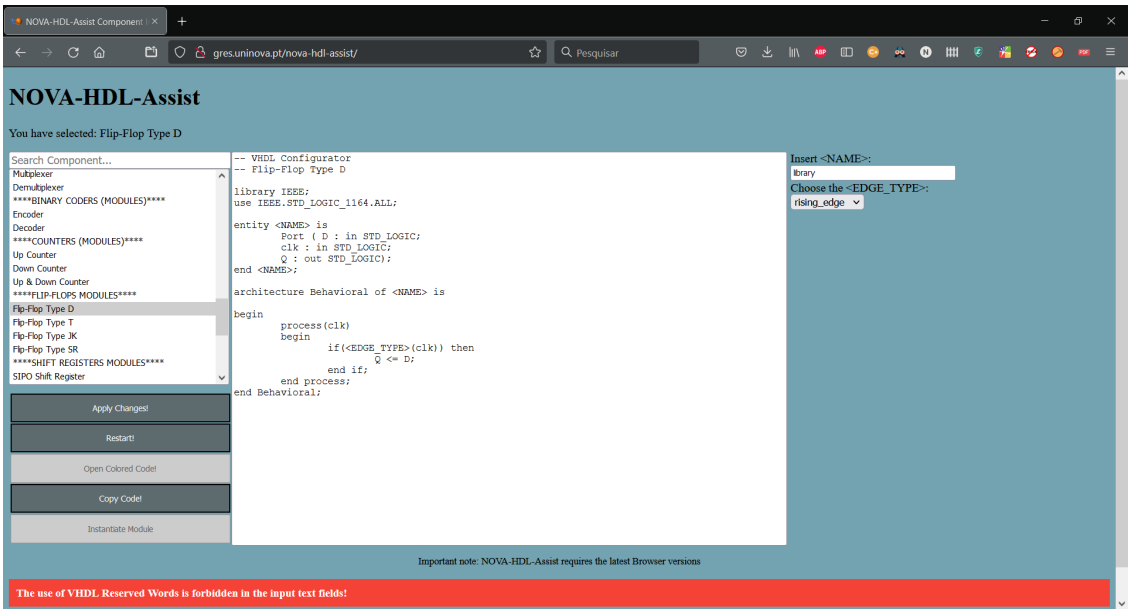

Figura 4.14: Teste à configuração do módulo *Flip-Flop Type D* com o uso da palavra reservada *library* em VHDL. Depois de se clicar em *Apply Changes!*.

4.1.3 Instanciação dos módulos

A instanciação de módulos foi testada com recurso ao módulo *Up Counter* como a figura 4.15 o demonstra. Para a instanciação de um módulo, o utilizador deverá escolher o módulo que deseja, configurá-lo e depois de clicar para implementar as alterações, clicar em *Instantiate Module*.

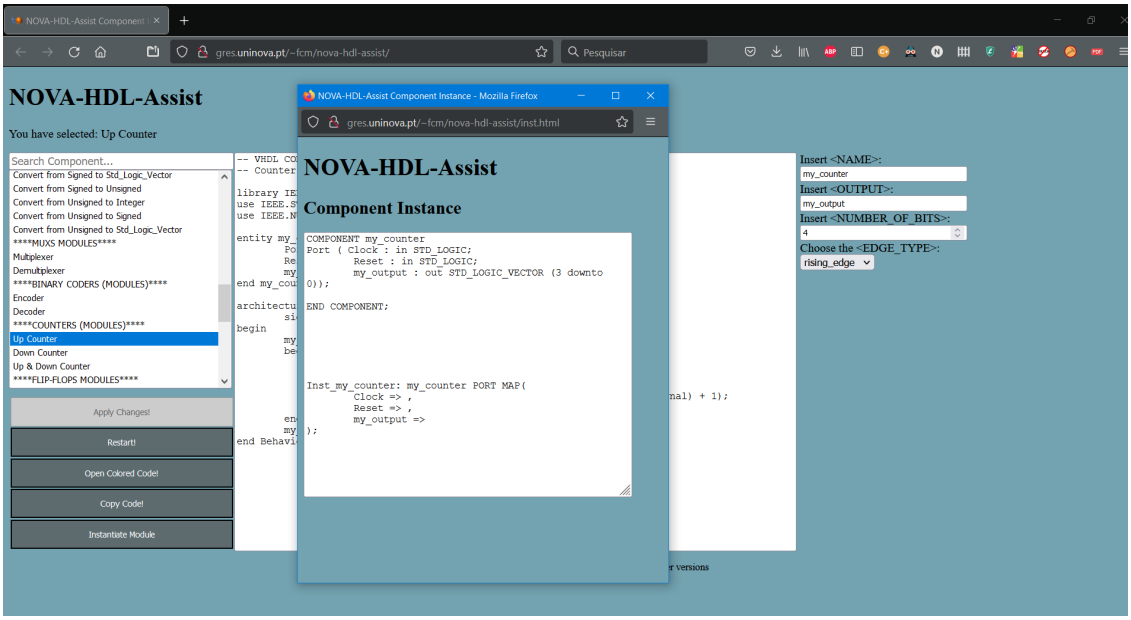

Figura 4.15: Teste à instanciação do módulo *Up Counter*.

4.2 Inquérito

Para validar a NOVA-HDL-Assist foi realizado um inquérito a dois grupos de alunos e ex-alunos do Mestrado Integrado em Engenharia Electrotécnica e de Computadores da FCT NOVA, com recurso ao *Google Forms*. Os grupos caracterizam-se da seguinte forma:

- Primeiro grupo (mais experiente) - Composto por doze alunos e ex-alunos (um do sexo feminino e onze do sexo masculino) que tinham aprendido VHDL durante o curso.

- Segundo grupo (menos experiente) - Composto por dez alunos (um do sexo feminino e nove do sexo masculino) que à data da realização do inquérito estavam a aprender VHDL na Unidade Curricular de Concepção de Sistemas Digitais.

O inquérito foi dividido em três partes. A primeira parte do questionário refere-se ao aspecto gráfico da ferramenta; a segunda aos módulos VHDL disponibilizados pela ferramenta; e a última é relativa à componente de descrição de *hardware*. Cada subsecção desta secção corresponde a uma parte do questionário e a última subsecção (4.2.4) corresponde a uma breve análise dos dados estatísticos obtidos.
Salientar que nos gráficos apresentados nesta secção, o valor "1" corresponde a uma classificação baixa e o valor "5" corresponde a uma classificação alta. Por exemplo, no gráfico 4.16, o valor "1" corresponde a uma interface gráfica pouco simples e com um aspecto gráfico mau, enquanto que o valor "5" corresponde a uma interface gráfica simples e com um bom aspecto gráfico.

4.2.1 Aspecto gráfico da interface

No desenvolvimento da ferramenta um dos principais objectivos era conceber uma ferramenta o mais graficamente simples e intuitiva, de forma a que os utilizadores percam o menor tempo possível a tentar compreender o seu funcionamento.
Este objectivo parece ter sido alcançado, tendo tanto o grupo mais experiente como o menos experiente dado uma boa classificação.
Nos gráficos 4.16 e 4.17 estão presentes os resultados relativamente ao grupo mais experiente e ao menos experiente, respectivamente. Estes gráficos referem-se à interface gráfica quanto ao seu aspecto e à sua simplicidade.

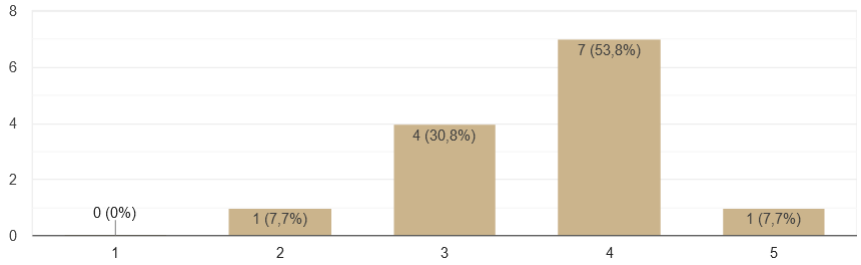

Figura 4.16: Classificação da interface gráfica quanto à sua simplicidade e aspecto gráfico por parte dos inquiridos. Classificação do grupo mais experiente

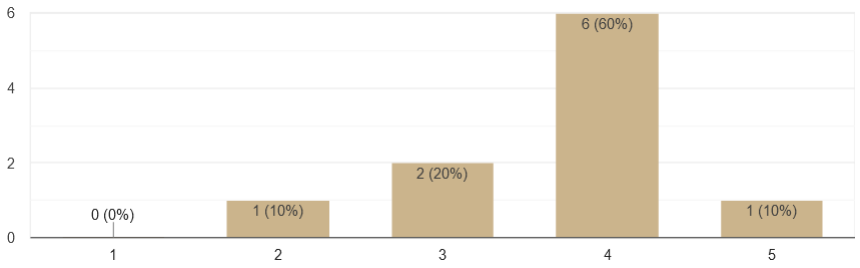

Figura 4.17: Classificação da interface gráfica quanto à sua simplicidade e aspecto gráfico por parte dos inquiridos. Classificação do grupo menos experiente

Nos gráficos 4.18 e 4.19 é possível concluir que os inquiridos mostraram-se satisfeitos quanto à intuitividade e facilidade do uso da ferramenta.

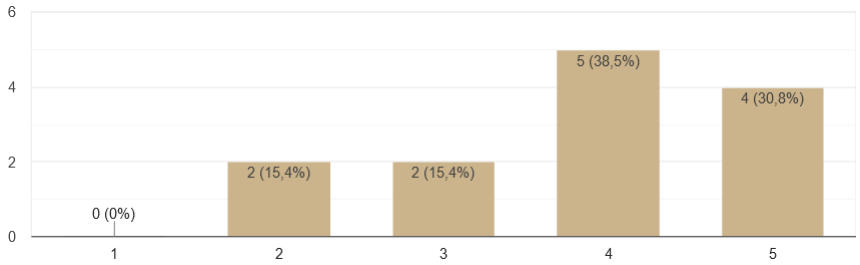

Figura 4.18: Classificação da interface gráfica quanto à sua facilidade de uso por parte dos inquiridos. Classificação do grupo mais experiente.

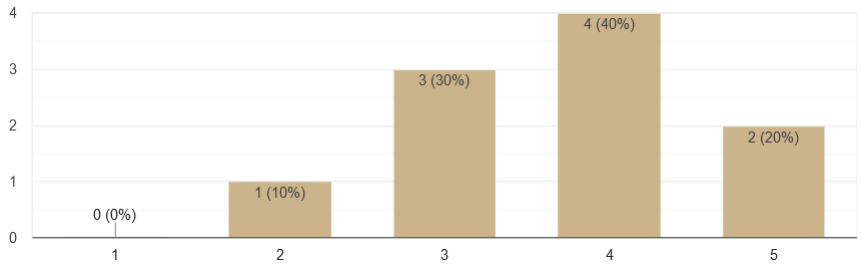

Figura 4.19: Classificação da interface gráfica quanto à sua facilidade de uso por parte dos inquiridos. Classificação do grupo menos experiente.

4.2.2 Módulos VHDL disponibilizados

Quando terminada a primeira versão da NOVA-HDL-Assist, partiu-se do princípio que os módulos disponibilizados não eram suficientes, pelo que nas futuras versões novos módulos teriam de ser adicionados. Contudo, contra as expectativas, a maioria dos inquiridos referiu que os módulos oferecidos são suficientes, como os gráficos 4.20 e 4.21 o constatam. Esta peripécia poderá ser explicada pelo facto das pessoas com pouca experiência em VHDL não necessitarem de muitos mais módulos para além dos oferecidos pela ferramenta.

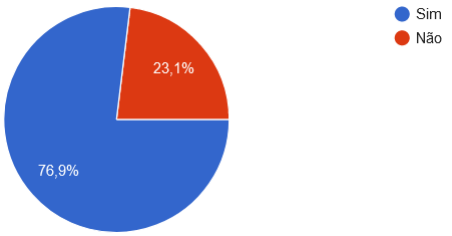

Figura 4.20: Classificação do número e dos módulos disponibilizados pela ferramenta, por parte dos inquiridos. Classificação do grupo mais experiente.

Ainda nesta secção os utilizadores disseram que alguns dos módulos mais úteis foram o *Arithmetic Operations*, *State Machine & Process* e os conversores. Quanto aos menos úteis o *If Statement* foi escolhido pela maioria dos inquiridos.

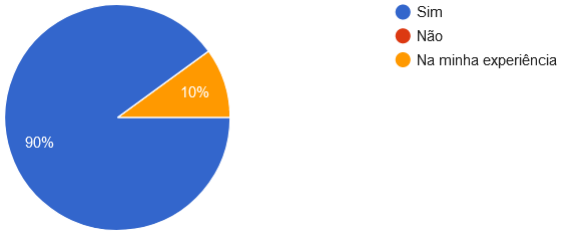

Figura 4.21: Classificação do número e dos módulos disponibilizados pela ferramenta, por parte dos inquiridos. Classificação do grupo menos experiente.

4.2.3 Componente de descrição de *hardware*

Nesta secção, mais uma vez o resultado dos inquéritos foi coincidente com os objectivos definidos inicialmente.
Dos inquiridos, 100% referiu que a ferramenta permitiu reduzir o tempo de pesquisa e reduzir os erros de sintaxe, como os gráficos 4.22 e 4.23 o referem.

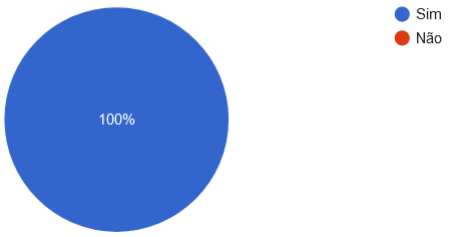

Figura 4.22: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist permitiu reduzir o tempo de pesquisa. Classificação atribuída tanto pelo grupo menos experiente como o mais experiente.

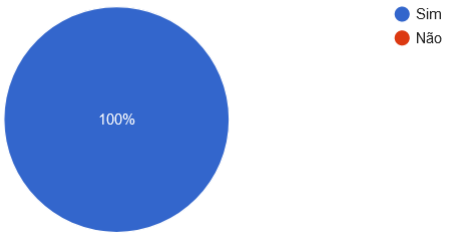

Figura 4.23: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist permitiu reduzir os erros de sintaxe. Classificação atribuída tanto pelo grupo menos experiente como o mais experiente.

Os inquiridos referiram também que a ferramenta tornou a VHDL uma linguagem mais simples e intuitiva (4.24 e 4.25), que aumentou a velocidade de desenvolvimento dos seus projectos (4.26 e 4.27), que incentivou a boas práticas de descrição de *hardware* (4.28 e

4.29) e 100% dos inquiridos considerou que a NOVA-HDL-Assist é uma mais valia na aprendizagem de VHDL (4.30 e 4.31).

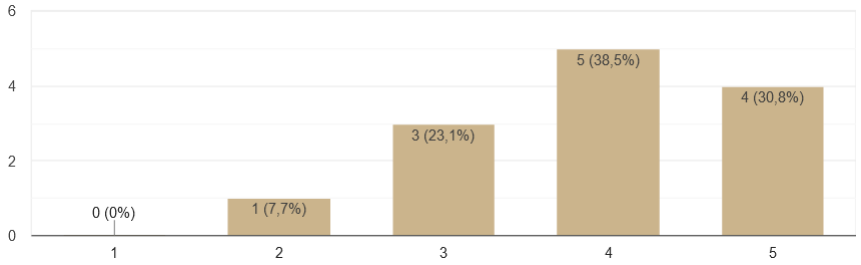

Figura 4.24: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist tornou a VHDL uma linguagem mais simples e intuitiva. Classificação atribuída pelo grupo mais experiente.

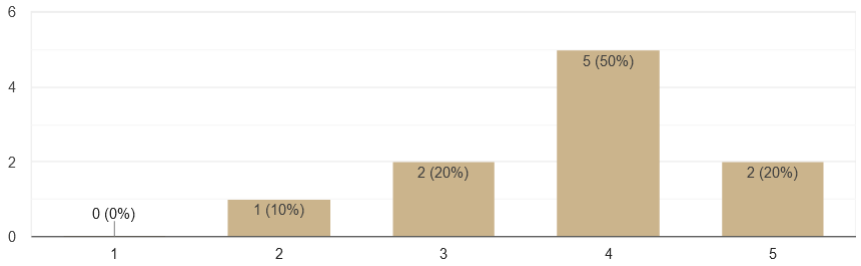

Figura 4.25: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist tornou a VHDL uma linguagem mais simples e intuitiva. Classificação atribuída pelo grupo menos experiente.

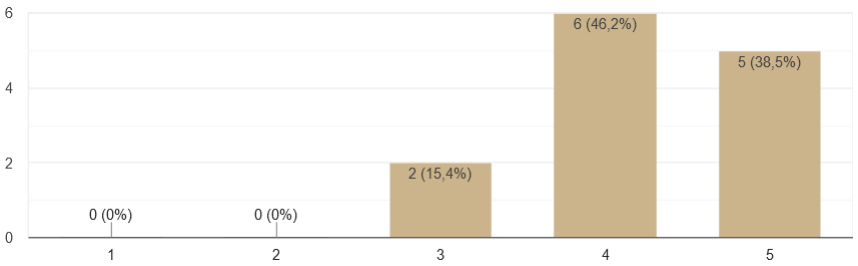

Figura 4.26: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist aumentou a velocidade de desenvolvimento dos seus projectos. Classificação pertencente ao grupo mais experiente.

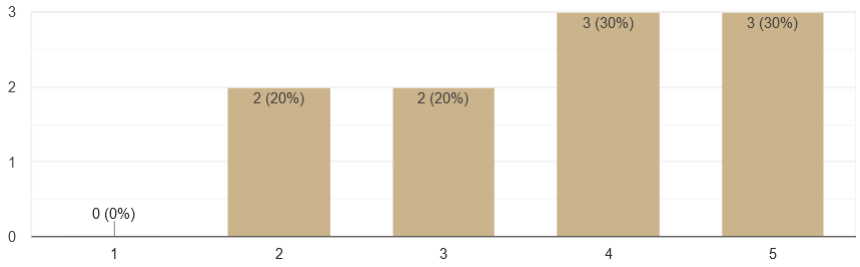

Figura 4.27: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist aumentou a velocidade de desenvolvimento dos seus projectos. Classificação pertencente ao grupo menos experiente.

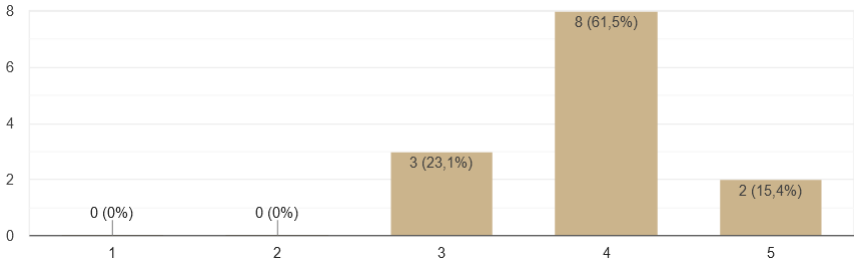

Figura 4.28: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist incentivou a boas práticas de desenvolvimento de *hardware*. Classificação pertencente ao grupo mais experiente.

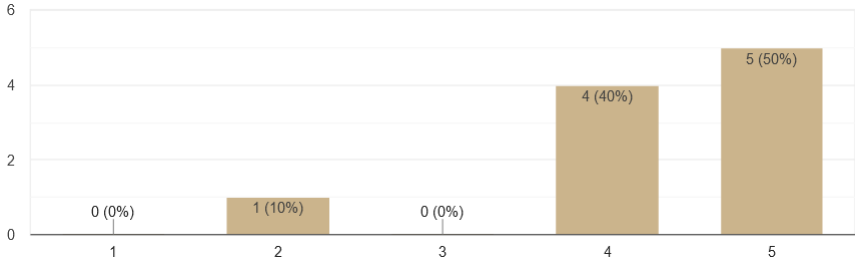

Figura 4.29: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist incentivou a boas práticas de desenvolvimento de *hardware*. Classificação pertencente ao grupo menos experiente.

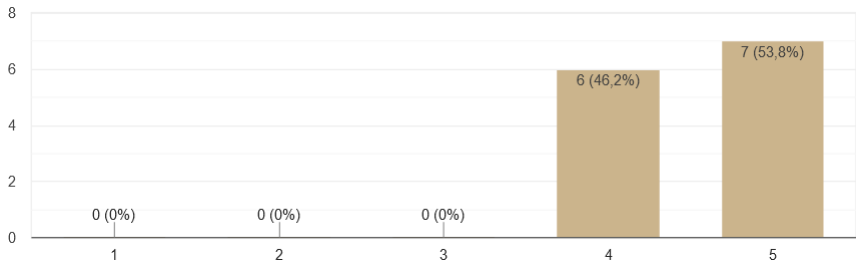

Figura 4.30: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist é uma mais-valia na aprendizagem das HDL. Classificação pertencente ao grupo mais experiente.

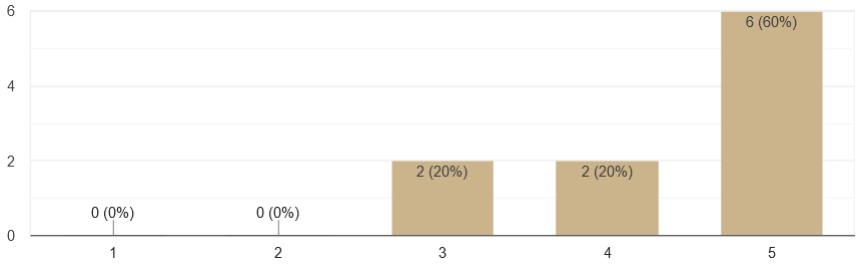

Figura 4.31: Classificação, por parte dos inquiridos, se a NOVA-HDL-Assist é uma mais-valia na aprendizagem das HDL. Classificação pertencente ao grupo menos experiente.

4.2.4 Análise de Resultados

Os resultados para ambos os grupos são bastante semelhantes em todas as perguntas, contudo ao longo do inquérito realizado é possível inferir através das respostas, que existem diferenças entre os dois grupos que participaram no inquérito.
Relativamente ao número de módulos disponibilizados (figuras 4.20 e 4.21) é curioso verificar que a maioria, em ambos os grupos, refere que os módulos disponibilizados pela ferramenta são suficientes. Contudo é de notar que 23,1% dos inquiridos do grupo mais experiente considera que os módulos oferecidos não são suficientes, contrastando com 0% do grupo menos experiente. Isto poderá ser explicado pela maior proficiência nas HDL dos inquiridos do grupo mais experiente, que levará os mesmos a necessitar e procurar por módulos mais complexos.
Na questão se a ferramenta aumenta a velocidade de desenvolvimento dos projectos (figuras 4.26 e 4.27), 84,7% dos inquiridos do grupo mais experiente atribuiu nota de '4' e '5' à ferramenta. Já no grupo menos experiente, este intervalo, fica-se pelos 60%. Uma possível explicação para este acontecimento poderá ser devido ao facto de um utilizador mais experiente ter mais autonomia, já conhecer as HDL e saber o que necessita de consultar e configurar.

4.3 Desenvolvimento de novos módulos

Os módulos apresentados nesta secção foram integralmente projectados e concebidos por um dos inquiridos, o que demonstra a possibilidade de no futuro os utilizadores poderem contribuir com novos módulos e excertos de código de forma massiva.
Os módulos desenvolvidos pelo inquirido foram concebidos e testados pelo mesmo, tendo depois traduzido os seus módulos para o formato JSON, de modo a que seja interpretável pela ferramenta. Os ficheiros JSON foram de seguida inseridos na base de dados.
O sucesso de desta abordagem revela que futuramente existe espaço para explorar técnicas e métodos relativos à contribuição por utilizadores. O sucesso desta abordagem deverá passar por criar um sistema amigo do utilizador que permita a submissão de módulos ou modelos de código de descrição de *hardware*.

4.3.1 Temporizador

No módulo temporizador o utilizador deverá configurar a frequência de funcionamento e o seu período. Sempre que um período de tempo for completado, o sinal de saída irá tomar o valor binário de '1', durante um *clock*.

Campos	Descrição
Nome	O utilizador deverá indicar o do módulo
Nome de *reset*	O utilizador deverá indicar o nome do *reset*
Nome de *clock*	O utilizador deverá indicar o nome do *clock*
Nome do *output*	O utilizador deverá indicar o nome do *output*
Valor da frequência	O utilizador deverá indicar da frequência do temporizador
Valor do período	O utilizador deverá indicar o valor do período do temporizador
Tipo de *edge*	O utilizador deverá escolher entre *rising edge* e *falling edge*
Tipo de *reset*	O utilizador deverá indicar se a máquina de estados terá um *reset* assíncrono ou síncrono

Tabela 4.1: Tabela com os campos e descrição de configuração do temporizador.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity <NAME> is
5
6  Port (
7    <CLOCK> : in STD_LOGIC;
8    <RESET>: in STD_LOGIC;
9    <OUTPUT> : out STD_LOGIC
10 );
11
12
13 end <NAME>;
14
15 architecture Behavioral of <NAME> is
16
17 signal cont : integer range 0 to <FREQ_HZ>*<PERIOD_MS>/1000 := 0;
18
19 begin
```

```
20   process(<CLOCK>)
21     begin
22     if <EDGE_TYPE>(<CLOCK>) then
23
24       if <RESET> = '1' then
25         cont <= 0;
26         else
27           if cont = (((<FREQ_HZ>/1000)*<PERIOD_MS>)-1) then
28             cont   <= 0;
29             <OUTPUT> <= '1';
30           else
31             cont <= cont + 1;
32             <OUTPUT> <= '0';
33             end if;
34         end if;
35     end if;
36   end process;
37
38
39 end Behavioral;
```

Listagem 4.1: Código VHDL da ferramenta por configurar. Este código corresponde ao temporizador.

4.3.2 *Debounce*

O módulo de *debounce* é utilizado devido ao facto de, por exemplo, os botões e interruptores mecânicos, ao serem clicados terem um comportamento inesperado (ressalto), e como tal é necessário desenvolver uma técnica e aguardar um período de tempo para que o sinal de entrada seja um sinal limpo, estável e pronto para ser processado.

Campos	Descrição
Nome	O utilizador deverá indicar o do módulo
Nome de *reset*	O utilizador deverá indicar o nome do *reset*
Nome de *clock*	O utilizador deverá indicar o nome do *clock*
Nome do *input*	O utilizador deverá indicar o nome do *input*
Nome do *output*	O utilizador deverá indicar o nome do *output*
Valor da frequência	O utilizador deverá indicar da frequência do *debouncer*
Valor do período	O utilizador deverá indicar o valor do período do *debouncer*
Tipo de *edge*	O utilizador deverá escolher entre *rising edge* e *falling edge*
Tipo de *reset*	O utilizador deverá indicar se a máquina de estados terá um *reset* assíncrono ou síncrono

Tabela 4.2: Tabela com os campos e descrição de configuração do *debounce*.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity <NAME> is
5
6  Port (
7    <CLOCK> : in STD_LOGIC;
8    <RESET>: in STD_LOGIC;
9    <INPUT>: in STD_LOGIC;
10   <OUTPUT> : out STD_LOGIC := '0'
```

```
11 );
12
13 end <NAME>;
14
15 architecture Behavioral of <NAME> is
16
17   signal cont : integer range 0 to (((<FREQ_HZ>/1000)*<TIME_MS>)-1);
18   signal internal_sig : std_logic := '0';
19
20 begin
21
22   process(<CLOCK>)
23   begin
24     if <EDGE_TYPE>(<CLOCK>) then
25       if <RESET> = '1' then
26         cont <= 0;
27         internal_sig <= <INPUT>;
28       else
29         if cont = (((<FREQ_HZ>/1000)*<TIME_MS>)-1) then
30           cont <= 0;
31           internal_sig <= <INPUT>;
32         else
33           if <INPUT> /= internal_sig then
34             cont <= cont + 1;
35           else
36             cont <= 0;
37           end if;
38         end if;
39       end if;
40     end if;
41
42   end process;
43
44 <OUTPUT> <= internal_sig;
45
46 end Behavioral;
```

Listagem 4.2: Código VHDL da ferramenta por configurar. Este código corresponde ao *debounce*.

4.3.3 Modulação por largura de pulso

A modulação por largura de pulso é um módulo interessante de se incluir na ferramenta devido ao seu alargado uso. Este tipo de modulação funciona como um interruptor em que consoante a informação transmitida (ex.: *duty cycle*) o modo de funcionamento de um determinado dispositivo receptor deverá alterar-se. Para a configuração deste módulo o utilizador deverá ter em conta a frequência de oscilação (frequência de repetição do sinal) e o valor em percentagem da largura de pulso.

Campos	Descrição
Nome	O utilizador deverá indicar o nome do módulo
Nome de *reset*	O utilizador deverá indicar o nome do *reset*
Nome de *clock*	O utilizador deverá indicar o nome do *clock*
Nome do *output*	O utilizador deverá indicar o nome do *output*
Valor da frequência máxima	O utilizador deverá indicar o valor da frequência máxima de funcionamento do módulo. Esta frequência deverá ser superior à frequência de oscilação
Valor da frequência de oscilação	O utilizador deverá indicar da frequência de oscilação
Valor em percentagem da largura de pulso	O utilizador deverá indicar a largura de pulso
Tipo de *edge*	O utilizador deverá escolher entre *rising edge* e *falling edge*
Tipo de *reset*	O utilizador deverá indicar se a máquina de estados terá um *reset* assíncrono ou síncrono

Tabela 4.3: Tabela com os campos e descrição de configuração do módulo de modulação por largura de pulso.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity <NAME> is
```

```
5
6  Port (
7    <CLOCK> : in STD_LOGIC;
8    <RESET> : in STD_LOGIC;
9    <OUTPUT> : out STD_LOGIC := '0'
10 );
11
12
13 end <NAME>;
14
15 architecture Behavioral of <NAME> is
16
17   signal cont : integer range 0 to (<FREQ_HZ>/<FOSC_HZ>-1) := '0';
18
19 begin
20
21   process(<CLOCK>)
22   begin
23     if <EDGE_TYPE>(<CLOCK>) then
24       if <RESET> = '1' then
25         cont <= '0';
26         <OUTPUT> <= '0';
27       else
28         if cont = ((<FREQ_HZ>/<FOSC_HZ>)-1) then
29           cont <= 0;
30           <OUTPUT> <= '0';
31         else
32           if cont < (((<FREQ_HZ>/<FOSC_HZ>)*<PWM_PERCENT>)/100) then
33             <OUTPUT> <= '1';
34             cont <= cont + 1;
35           else
36             <OUTPUT> <= '0';
37             cont <= cont + 1;
38           end if;
39         end if;
40       end if;
41     end if;
42   end process;
43 end Behavioral;
```

Listagem 4.3: Código VHDL do módulo de modulação por largura de pulso por configurar.

5

Conclusões e Trabalhos Futuros

É importante relembrar que existe um crescente interesse, por parte da comunidade, no ensino de linguagens de descrição de *hardware* e desenvolvimento de aplicações e métodos que complementem e facilitem esse ensino. O trabalho de investigação realizado no âmbito desta dissertação leva à reflexão que, apesar desse crescente interesse, as ferramentas disponibilizadas continuam escassas e/ou difíceis de encontrar.
É da opinião de alguns autores, alguns abordados neste documento, que uma ferramenta intuitiva e interactiva é uma mais-valia no ensino das linguagens de descrição de *hardware*, daí ter sido desenvolvido a NOVA-HDL-Assist. Sendo imperativo lembrar que o facto de não serem conhecidas ferramentas semelhantes a esta, propiciou o seu desenvolvimento.
A ferramenta apresentada neste documento e respectiva abordagem poderão ter um impacto significativo no ensino das linguagens de descrição de *hardware*.
No capitulo 4.2 relativo ao inquérito e à classificação atribuída pelos inquiridos, podemos concluir que características como a rápida configuração do código VHDL permite reduzir o tempo de pesquisa de exemplos de código e mitigar os erros de sintaxe, incentivando ainda a boas práticas de descrição de *hardware* para FPGA, aliada à facilidade e diminuição da curva de aprendizagem, levam à reflexão de que a NOVA-HDL-Assist será uma opção para quem deseja melhorar os seus conhecimentos nesta área.
Sendo que o público-alvo desta ferramenta será em grande parte estudantes, entusiastas e outros profissionais com pouca ou nenhuma experiência em linguagens de descrição de *hardware*, importa não descartar a sua potencial utilidade para profissionais nesta área. Esta ferramenta poderá ser vista como um complemento aos ambientes de desenvolvimento comerciais mencionados neste documento.
A NOVA-HDL-Assist pode ser encontrada em `http://gres.uninova.pt/nova-hdl-assist/`. No âmbito desta dissertação de mestrado foi publicado um artigo com o título "Ferramenta de Edição e Configuração de Código VHDL" [31].

Futuramente, após a publicação da primeira versão, deverão surgir novas versões com novas características e correcções de *bugs*, potenciando e explorando novas funcionalidades

latentes a esta ferramenta.
São vários os caminhos que podem ser seguidos no desenvolvimento da ferramenta, bem como as funcionalidades que poderão ser desenvolvidas nos anos que se seguirão. São sugeridas algumas para considerar em futuras versões, como:

- Desenvolvimento de mais, e mais complexos módulos;
- Melhoramentos no instanciar de módulos. Permitindo, por exemplo, instanciar vários módulos de uma só vez;
- Permitir o desenvolvimento de um projecto de forma estruturada na ferramenta, reduzindo a dependência da ferramenta relativamente aos ambientes de desenvolvimento comerciais;
- Desenvolvimento de um sistema que gere *test bench* de forma simples e automática;
- Desenvolvimento de um sistema integrado de teste por formas de onda, à semelhança do GTKWave;
- Integração da ferramenta com outros *softwares*, como o GHDL;
- Para além da VHDL, adaptar a ferramenta de modo a que outras linguagens de descrição de *hardware* possam ser utilizadas, como Verilog, SystemVerilog, SystemC e outras linguagens menos relevantes mas igualmente interessantes.

Bibliografia

[1] R. W. Nowlin e R. Sundararajan. "A VHDL Course For Electronics Engineering Technology". Em: *1997 Annual Conference*. 10.18260/1-2–6893. https://peer.asee.org/6893. Milwaukee, Wisconsin: ASEE Conferences, jun. de 1997 (ver p. 1).

[2] R. Rodriguez-Ponce e J. Rodriguez-Resendiz. "Integrating VHDL into an undergraduate digital design course". Em: *Proceedings of 2013 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*. 2013, pp. 172–177. DOI: 10.1109/TALE.2013.6654423 (ver pp. 1, 16).

[3] Y. A. Badamasi. "The working principle of an Arduino". Em: *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*. 2014, pp. 1–4. DOI: 10.1109/ICECCO.2014.6997578 (ver p. 2).

[4] A. Wu. "Interactive learning toolbox for logic synthesis with VHDL". Em: *Proceedings of International Conference on Microelectronic Systems Education*. 1997, pp. 77–78. DOI: 10.1109/MSE.1997.612555 (ver pp. 2, 17, 18).

[5] A. Etxebarria, I. J. Oleagordia e M. Sanchez. "An educational environment for VHDL hardware description language using the WWW and specific workbench". Em: 1 (2001), T2C–2. DOI: 10.1109/FIE.2001.963876 (ver pp. 2, 18).

[6] A. Madanayake, C. Wijenayake, R. M. Joshi, J. Grover, J. Carletta, J. Adams, T. Hartley e T. Ogunfunmi. "Teaching freshmen VHDL-based digital design". Em: *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2012, pp. 2701–2704. DOI: 10.1109/ISCAS.2012.6271865 (ver pp. 2, 17).

[7] G. R. Garay, A. Tchernykh, A. Y. Drozdov, S. N. Garichev, S. Nesmachnow e M. Torres-Martinez. "Visualization of VHDL-based simulations as a pedagogical tool for supporting computer science education". Em: *Journal of Computational Science* 36 (2019), p. 100652. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2017.04.004. URL: http://www.sciencedirect.com/science/article/pii/S187775031730385X (ver pp. 2, 19, 24).

[8] L. Gomes e A. Costa. "Hardware-level Design Languages". Em: *The Industrial Information Technology Handbook*. n/a. 2005, pp. 1–18 (ver p. 4).

[9] S. Sjoholm e L. Lindh. *VHDL for Designers*. Prentice Hall PTR, 1997 (ver p. 4).

[10] D. A. T. Peter J. Ashenden Gregory D. Peterson. *The System Designer's Guide to VHDL-AMS*. Elsevier, 2003. ISBN: 978-1-55860-749-1 (ver p. 5).

[11] S. D. Automation. "VHDL Reference Manual". Em: *Redmond, Washington* (1997) (ver pp. 5–7).

[12] Z. Navabi. "Elements of VHDL for description of hardware: a tutorial view". Em: (1990), T/7.1–T/7.9. DOI: 10.1109/ASIC.1990.186076 (ver p. 5).

[13] X. University. *Modeling Concepts*. URL: https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/lab1.pdf (acedido em 02/12/2020) (ver p. 6).

[14] IBM. *Operadores e Expressões*. URL: https://www.ibm.com/docs/pt-br/tcamfma/6.3.0?topic=tesl-operators-expressions (acedido em 08/11/2021) (ver p. 8).

[15] J. L. Peter J. Ashenden. *The Designer's Guide To VHDL*. Terceira Edição(Systems on Silicon). Elsevier, 2008. ISBN: 978-0-12-088785-9 (ver pp. 9–14).

[16] N. Botros. *HDL With Digital Design VHDL AND VERILOG*. Mercury Learning e Information, 1981. ISBN: 978-1-938549-81-6 (ver pp. 14–16).

[17] IEEE. *IEEE Standard VHDL Language Reference Manual*. Institute of Electrical e Electronics Engineers, Inc, 2000. ISBN: 0-7381-1949-0 (ver p. 14).

[18] X. University. *Functions, Procedures, and Testbenches*. URL: https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/lab4.pdf (acedido em 02/12/2020) (ver p. 15).

[19] V. A. Pedroni. "Teaching design-oriented VHDL". Em: *Proceedings 2003 IEEE International Conference on Microelectronic Systems Education. MSE'03.* 2003, pp. 6–7. DOI: 10.1109/MSE.2003.1205229 (ver p. 17).

[20] A. I. Hussein, D. M. Gruenbacher e N. M. Ibrahim. "Design and verification techniques used in a graduate level VHDL course". Em: 2 (nov. de 1999), 13A4/28–13A4/31 vol.2. ISSN: 0190-5848. DOI: 10.1109/FIE.1999.841737 (ver p. 17).

[21] I. Said e M. S. Çavuş. "ALU DESIGN BY VHDL USING FPGA TECHNOLOGY AND MICRO LEARNING IN ENGINEERING EDUCATION". Em: (jan. de 2018) (ver p. 19).

[22] F. Pereira, F. Moutinho e L. Gomes. "IOPT-tools — Towards cloud design automation of digital controllers with Petri nets". Em: *2014 International Conference on Mechatronics and Control (ICMC)*. 2014, pp. 2414–2419. DOI: 10.1109/ICMC.2014.7232002 (ver p. 20).

[23] F. Pereira e L. Gomes. "Automatic synthesis of VHDL hardware components from IOPT Petri net models". Em: *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*. 2013, pp. 2214–2219. DOI: 10.1109/IECON.2013.6699475 (ver p. 20).

[24] F. Pereira e L. Gomes. "The IOPT-Flow framework pairing Petri nets and data-flows for embedded controller development". Em: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. 2016, pp. 4832–4837. DOI: 10.1109/IECON.2016.7794152 (ver p. 20).

[25] R. E. Haskell e D. M. Hanna. *Introduction to Digital Design: Using Digilent FPGA Boards; Block Diagram, VHDL Examples;[30 VHDL, Active-HDL Examples]*. LBE Books, 2009 (ver pp. 20, 21, 23).

[26] B. Zeidman. "Introduction to CPLD and FPGA Design". Em: *Embedded System Conference, San Fransisco*. 2004 (ver pp. 21, 22).

[27] P. Bellows e B. Hutchings. "JHDL-an HDL for reconfigurable systems". Em: *Proceedings. IEEE symposium on FPGAs for custom computing machines (Cat. No. 98TB10025 1)*. IEEE. 1998, pp. 175–184 (ver p. 23).

[28] V. Boscaino, G. Capponi, G. M. Di Blasi, P. Livreri e F. Marino. "Modeling and simulation of a digital control design approach for power supply systems". Em: *2006 IEEE Workshops on Computers in Power Electronics*. 2006, pp. 246–249. DOI: 10.1109/COMPEL.2006.305638 (ver pp. 25, 26).

[29] Cadence. *Stratus High-Level Synthesis*. URL: https://www.cadence.com/content/dam/cadence-www/global/en%5C_US/documents/tools/digital-design-signoff/stratus-ds.pdf (acedido em 24/06/2021) (ver p. 26).

[30] L. Stornaiuolo, M. Santambrogio e D. Sciuto. "On how to efficiently implement deep learning algorithms on pynq platform". Em: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2018, pp. 587–590 (ver p. 26).

[31] N. Leal, F. Moutinho, L. Gomes e A. Costa. "Ferramenta de edição e configuração de código VHDL". Em: Porto Portugal. Zenodo, 2021. DOI: 10.5281/zenodo.5090012. URL: https://doi.org/10.5281/zenodo.5090012 (ver pp. 28, 113).

[32] Nandland. *Examples of VHDL Conversions*. URL: https://www.nandland.com/vhdl/tips/tip-convert-numeric-std-logic-vector-to-integer.html (acedido em 15/04/2021) (ver p. 47).

[33] A. K. Maini. *Digital electronics: principles, devices and applications*. John Wiley & Sons, 2007 (ver pp. 51, 54, 56, 60–62, 66, 67, 69, 71, 73, 75, 78, 80).

[34] TutorialsPoint. *VHDL Programming for Sequential Circuits*. URL: https://www.tutorialspoint.com/vlsi_design/vhdl_programming_for_sequential_circuits.htm (acedido em 15/04/2021) (ver pp. 51, 54).

[35] A. A. FPGA. *VHDL Code for 4-Bit Shift Register*. URL: https://allaboutfpga.com/vhdl-code-for-4-bit-shift-register/ (acedido em 15/04/2021) (ver p. 57).

[36] A. A. FPGA. *VHDL Code for 4 to 2 Encoder*. URL: https://allaboutfpga.com/vhdl-code-for-4-to-2-encoder/ (acedido em 15/04/2021) (ver pp. 60, 62).

[37] B. Tech. *VHDL Code for Half Adder*. URL: https://buzztech.in/vhdl-code-half-adder-structural-behavioral-dataflow/ (acedido em 15/04/2021) (ver pp. 66, 71).

[38] S. Arar. *Review of VHDL Signed/Unsigned Data Types*. https://www.allaboutcircuits.com/technical-articles/review-of-vhdl-signed-unsigned-data-types/. (Acedido em 15/04/2021) (ver p. 77).

[39] I. Chiuchisan, A. D. Potorac e A. Graur. "Finite state machine design and VHDL coding techniques". Em: *Development and Application Systems* (2010), p. 75 (ver p. 81).