Lucas Moura Bastos

Implementação e análise da aplicação em C/C++ com Python embutido em um projeto de Microeletrônica

Campina Grande, Brasil

20 de dezembro de 2018

Lucas Moura Bastos

Implementação e análise da aplicação em C/C++ com Python embutido em um projeto de Microeletrônica

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Orientador: Gutemberg Gonçalves dos Santos Junior, D.Sc.

Campina Grande, Brasil

20 de dezembro de 2018

Lucas Moura Bastos

Implementação e análise da aplicação em C/C++ com Python embutido em um projeto de Microeletrônica

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Trabalho aprovado em: /12/2018

Gutemberg Gonçalves dos Santos Junior, D.Sc.
Orientador

Marcos Ricardo Alcântara Morais, D.Sc.
Avaliador

Campina Grande, Brasil
20 de dezembro de 2018

Dedico este trabalho aos meus pais, Marcus e Cláudia, ao meu irmão, Danilo, e aos Pseudomitos.

Agradecimentos

Primeiramente, agradeço aos meus pais, Marcus e Cláudia, e ao meu irmão, Danilo, pelo apoio e suporte incondicional em todas as minhas escolhas.

Aos meus amigos de faculdade, em especial aos *Pseudomitos*, que desde 2012 vêm se ajudando e suprindo as dificuldades acadêmicas em grupos de estudo. Sem eles essa jornada seria, certamente, impossível.

Aos companheiros e professores do laboratório XMEN, em especial ao meu orientador Gutemberg, essenciais para a minha capacitação na área de microeletrônica.

Aos colegas da Idea!, em especial Daniel, Tomaz, Eric e Leandro, com quem trabalho diretamente e me orientam em todas as dúvidas que surgem no decorrer do projeto.

"and this bird you cannot change",
Lynyrd Skynyrd

Resumo

A Interface de Programação de Aplicação é um conjunto de métodos que permite a extensão de linguagens de programação. Visando um melhor aproveitamento de recursos e tempo considerando o fluxo de projeto na área de microeletrônica com aplicação em fotônica, foi realizado um estudo da sua documentação e de exemplos encontrados na literatura a fim de facilitar o aprendizado e aplicação dessa ferramenta em projetos reais nas mais diferentes áreas.

Palavras-chaves: Interface de Programação de Aplicação; Extensão de linguagem de programação; Microeletrônica.

Abstract

The Application Programming Interface is a series of methods that allows the extension of programming languages. Intending the improvement of resources and time management when considering the design flow in the microelectronics field focused in photonics, a study of the documentation and examples found at the literature has been made aiming to to learn and apply this tool in real life projects.

Key-words: Application Programming Interface, programming language extension, Microelectronics.

Lista de ilustrações

Lista de abreviaturas e siglas

API	Application Programer's Interface
RTL	Register Transfer Level
HDL	Hardware Description Language
FIFO	First In, First Out
DPI	Direct Programming Interface
DSP	Digital Signal Processor

Sumário

1 Introdução

Um projeto de microeletrônica pode ser dividido em duas principais etapas: *front-end* e *back-end*. Na etapa *front-end* o cliente decide as especificações do seu chip e um modelo funcional do sistema é feito em linguagem de programação de alto nível. Ocorre, então, uma "tradução" para um modelo sistêmico (arquitetural) no qual tem-se o controle de latência e a suportabilidade para cálculos em ponto fixo. Em seguida, é projetado um modelo em linguagem de hardware HDL (Hardware Description Language). A última etapa consiste na comparação dos resultados obtidos pelos diferentes modelos, sob as mesmas condições de operação. Se o erro entre eles atender à tolerância especificada, o modelo em RTL é validado e uma otimização da lógica é feita através de uma Síntese Lógica. O back-end corresponde ao design físico. Nesta etapa, as representações de circuito dos componentes são convertidas em representações geométricas (layout) de formas que, quando fabricadas nas correspondentes camadas de materiais, garantirão o funcionamento necessário dos componentes (PEIXOTO, 2018).

Dentre os vários desafios encontrados em cada uma das etapas do fluxo supracitado, um dos primeiros a ser levado em consideração consiste em uma frequente dificuldade presente na transcrição do modelo em linguagem de alto nível, ou modelo de referência, e seu respectivo modelo arquitetural (sistêmico). Além da problemática da implementação da latência e das diferenças nos resultados, devido ao fato de um ter representação em ponto flutuante e o outro em ponto fixo, causando uma certa dificuldade na validação dos resultados, a codificação do sistema em uma linguagem mais baixo nível tende a ser muito mais trabalhosa e demorada.

Este trabalho visa expor uma alternativa que solucione, ou ao menos diminua, as dificuldades apresentadas. Tornando o fluxo de projeto mais rápido e seguro para que as especificações do cliente sejam satisfeitas e o produto entregue no prazo solicitado.

2 Microeletrônica

2.1 Fluxo de Projeto

O conjunto de etapas relativo à implementação em descrição de hardware, bem como sua verificação, recebe o nome de *frontend*. A implementação diz respeito à transformação de uma ideia e um conjunto de especificações para representações lógicas e físicas. Enquanto a verificação é responsável por garantir a funcionalidade da implementação no decorrer do projeto.

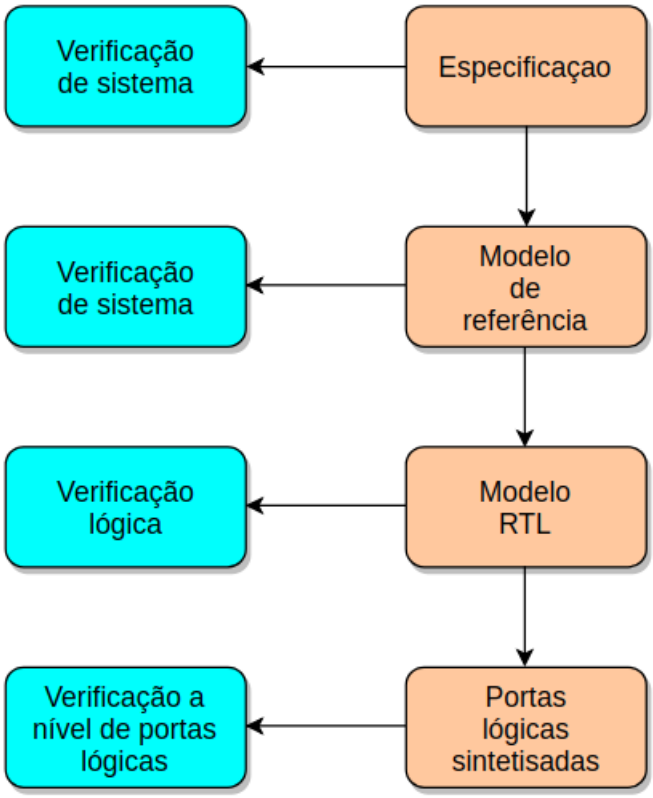

Figura 1 – Fluxo geral da etapa frontend de um projeto de microeletrônica.

A Figura 1 ilustra como dá-se o fluxo nas etapas de implementação e verificação. Ela pode ser, inicialmente, subdividida em duas para um melhor entendimento. A coluna à direita refere-se à implementação (*design*) do *hardware* e a coluna à esquerda, à verificação.

A etapa de especificação é onde acorda-se o comportamento geral e objetivos do *design* de forma textual ou gráfica. Definido isto, um estudo da microarquitetura é feito a fim de obter-se uma representação por *software* do comportamento especificado. O modelo de referência, implementado em linguagem de alto nivel, deve ser validado a nível

comportamental ou sistêmico para, então, dá-se início ao processo de implementação do circuito em uma linguagem de *hardware* (HDL - *Hardware Description Language*) e, uma vez obtendo-se um modelo funcional, ou estável, em HDL, o processo de verificação a nível de RTL (*Register Transfer Level*) pode ser iniciado por meio de testbenches que comparam as saídas dos modelos.

2.2 Estrutura de *testbench*

Em projetos de microeletrônica aplicados à, por exemplo, fotônica, o Matlab é certamente uma das ferramentas de modelagem mais utilizadas. Isto deve-se ao fato de existirem diversos pacotes e *toolboxes* com inúmeros recursos matemáticos muito utilizados no processamento de sinais. De forma geral, estes pacotes disponibilizam funções para analisar, pré-processar e extrair características dos sinais.

Via de regra, um ambiente de verificação, no qual o modelo de referência é feito em Matlab, possui uma estrutura básica semelhante à ilustrada na Figura 2. Um código gerador de dados fornece as entradas necessárias ao modelo de referência, ao passo que salva estes dados em arquivos. À medida que o modelo de referência recebe estas entradas, o mesmo gera as saídas esperadas que servirão para uma posterior comparação entre referência e modelo RTL.

Em se comparando o *Matlab* ao *Python*, algumas principais vantagens e desvantagens devem ser ressaltadas levando-se em conta a troca de dados entre os diferentes modelos, o consumo de memória, custo, desempenho e poder da linguagem.

Com o *Matlab* a troca de dados é feita via escrita e leitura de arquivos, que são tarefas que demandam tempo e um alto consumo de memória, piorando seu desempenho drasticamente. Em termos de poder da linguagem, o Matlab tem um desempenho matricial muito bom, mas que depende do "estilo" de codificação para fazer uso dela e, apesar de ter um bom número de funções específicas como mensionado anteriormente, deixa a desejar em componentes de propósito geral como *FIFOs* e bibliotecas de *multithreading/multiprocessing*. Além do fato dessa ferramenta ser código fechado o que muitas vezes pode levar à necessidade do uso do *Octave* por problemas de licenciamento. *Software*, este, que tem desempenho muito pior quando comparado com o *Matlab*.

Com relação aos quesitos mencionados, a utilização do *Python* traz as vantagens de que a troca de dados entre o *SystemVerilog* e C é feita via DPI (*Direct Programming Interface*) e entre C e Python, via a API, o que torna o consumo de memória muito mais eficiente. Ao mesmo tempo de que não há necessidade de licenciamento visto que o *Python* é uma linguagem de código aberto, a utilização de bibliotecas como *Numpy* possibilita uma codificação com alto desempenho e com funções essenciais para processamento de sinais análogas às *toolboxes* do *Matlab*. Além de um suporte a soluções de *software* (*third-*

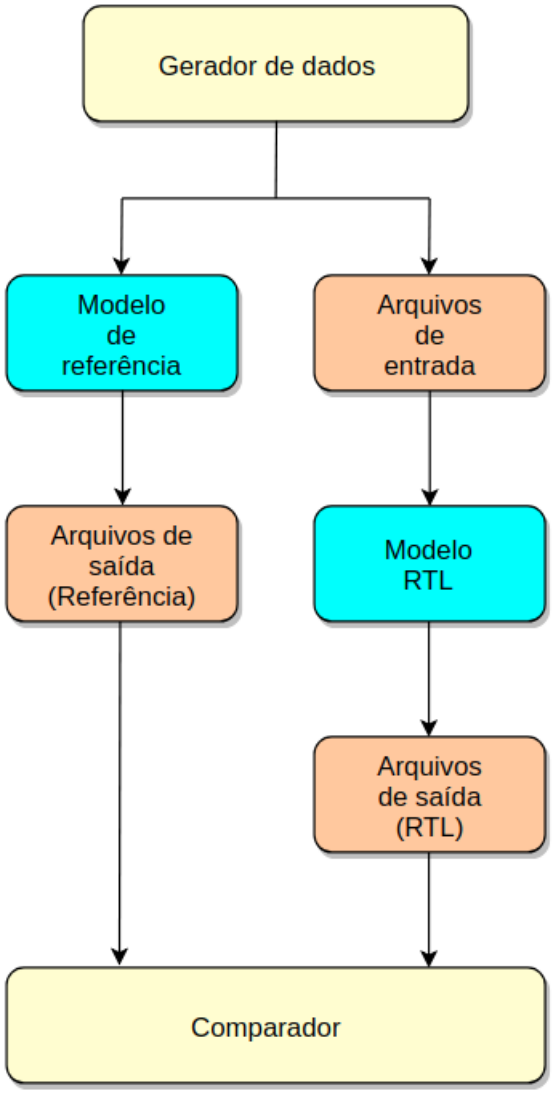

Figura 2 – Estrutura usual de um testbench.

party tools) extremamente amplo e um bom número de componentes de propósito geral (desde *FIFOs* até bibliotecas *multithreading*/*multitasking*).

2.3 Processamento Digital de Sinais

Processadores digitais de sinais (DSP - *Digital Signal Processor*) são microprocessadores especializados em processamento de sinais em tempo real ou offline.

O processamento consiste nas ferramentas matemáticas, algoritmos e nas técnicas utilizadas para manipular os sinais após terem sido convertidos para o formato digital (SMITH, 1997). Os sinais processados por esses dispositivos são sequências de amostras de uma variável contínua no domínio do tempo, espaço ou frequência.

A passagem do tempo contínuo para o discreto é possível através de um conversor

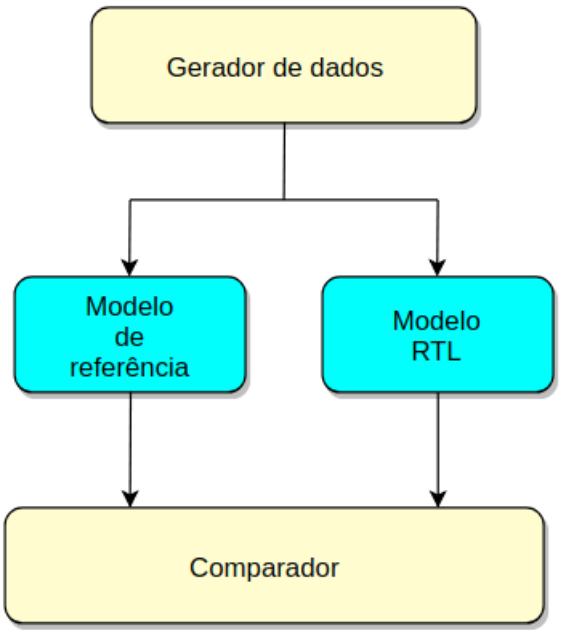

Figura 3 – Estrutura sugerida de um testbench.

analógico-digital (ADC - *Analog-to-Digital Converter*). Esta etapa é conhecida por amostragem e é realizado em dois momentos: discretização (amostragem) e quantização. No processo de discretização o sinal é dividido em intervalos de tempo iguais e cada intervalo é representado por uma única medida de amplitude. Após isso, a medida é aproximada para um valor finito. Aproximação, esta, conhecida como quantização.

3 Extendendo C/C++ com Python

No fluxo de um projeto, como explicado anteriormente, existe uma lacuna entre o momento em que se codifica o modelo em linguagem de alto nível, ou de referência, e seu respectivo modelo arquitetural. Essa distância deve-se ao fato de que, em muitas aplicações, a codificação de um modelo arquitetural pode tornar-se muito complexa devido à falta de bibliotecas matemáticas com as ferramentas necessárias para a implementação do processamento de sinais. Em se falando, por exemplo, de projeto de *chips* aplicados à telecomunicação, o sinal transmitido está sujeito a vários tipos de degradação, fazendo com que a mensagem recebida seja, muitas vezes, impossível de ser interpretada. Interferência intersimbólica, atenuação pelo canal e erros de transmissão, são apenas alguns dos efeitos que o DSP deve ser capaz de compensar. Para tanto, ferramentas matemáticas como convoluções, transformadas de Fourier, equalizações e filtragens são utilizadas. Infelizmente, dependendo da complexidade do DSP, a implementação desses algorítmos torna-se muito complexa e demorada.

Visando facilitar a passagem entre os modelos supracitados, a utilização de uma Interface de Programação de Aplicação (API - *Application Programer´s Interface*) mostrou-se essencial. Uma API é uma série de métodos de comunicação que permitem a interação entre duas ou mais linguagens de programação. Em se falando da API Python/C, ela disponibiliza para programadores de C e C++ acesso ao interpretador Python. Ou seja, *scripts*, funções e classes escritas em Python podem ser instanciadas no programa principal em C/C++ (PYTHON, 2018).

Existem dois motivos principais para o uso da API Python/C. A primeira é escrever extensões de módulos para um uso específico; esses são módulos em C que extendem o interpretador Python. O segundo motivo é utilizar o Python como um componente em uma aplicação maior; essa técnica é geralmente referida como Python embarcado. Ou seja, embarcar Python significa que o programa principal pode não ser implementado em Python, mas chama, ocasionalmente, seu interpretador para rodar algum código em Python.

Embarcando uma linguagem em outra faz com que uma linguagem seja capaz de implementar algumas das funcionalidades presentes na outra.

3.1 Descrição das ferramentas

Antes de escrever os códigos em si, faz-se necessária a preparação do ambiente. O presente relatório não conterá detalhes de como fazê-lo, mas a lista a seguir serve de guia

para o que o leitor deve prestar atenção.

- Compilador C/C++;
- Interpretador Python (recomendação: plataforma Anaconda);
- Bibliotecas necessárias devidamente instaladas (para C/C++ e Python);
- Diretório correto para as opções de *linking* (**-I** e **-L**) quando da compilação do programa em C/C++.

Versões das ferramentas utilizadas para esse relatório:

- GCC: gcc (Ubuntu 7.3.0-16ubuntu3) 7.3.0;
- Python: Python 3.6.5 :: Anaconda, Inc.

3.1.1 PYTHONPATH

Um PATH é uma variavel de ambiente que diz ao sistema operacional em quais diretórios procurar por arquivos executáveis. O PYTHONPATH é utilizado para dizer ao interpretador Python onde procurar por módulos para importar.

```
$ export PYTHONPATH=dir/to/python/file
```

Se existirem múltiplos módulos Python em diretórios diferentes

```
$ PYTHONPATH=~/one/location:$PYTHONPATH
$ PYTHONPATH=~/second/location:$PYTHONPATH
$ export PYTHONPATH
```

3.2 Inicializando o interpretador Python

Em uma aplicação de Python embarcado, a função *Py_Initialize()* deve ser chamada antes de qualquer outra função da API; com exceção de algumas funções de configuração e de alocação de memória e das variáveis de configuração global (PYTHON, 2018).

Para garantir que todas as configurações e ferramentas estão devidamente preparadas, o interpretador será inicializado e finalizado, e a função *Py_IsInitialized()* será utilizada para verificar se a inicialização fora realizada.

initialization.cpp

```
1
2 #include <Python.h>
3 #include <iostream>
4 using namespace std;
5
6 int main(int argc, char* argv[]) {
7
8     Py_Initialize(); // inicializacao do interpretador Python
9
10    cout << "Python is Initialized = " << Py_IsInitialized() << endl;
11
12    Py_Finalize();
13
14    cout << "Python is Initialized = " << Py_IsInitialized() << endl;
15
16    return(0);
17 }
```

- void Py_Initialize()
  Inicializa o interpretador Python.

- void Py_Finalize()
  Finaliza o interpretador Python.

- int Py_IsInitialized()
  Retorna verdadeiro (não zero) se o interpretador está inicializado, falso (zero) caso contrário.

3.3 Funções de importação de módulos Python

- PyObject* Py_BuildValue(const char *format, val)
  Retorna uma nova referência.
  Cria um novo valor baseado em um formato string. Retorna o novo valor ou NULL no caso de um erro.

- PyObject *PyUnicode_FromString(const char *u)
  Cria um objeto Unicode de um buffer de char u. Necessária para a importação de um módulo quando da utilização da função PyImport_Import(PyObject *name).

- PyObject* PyImport_Import(PyObject *name)
  Retorna uma nova referência para o módulo importado, ou NULL em caso de falha.

- PyObject* PyImport_ImportModule(const char *name)
  Retorna uma nova referência para o módulo importado, ou NULL em caso de falha. Essa função sozinha substitutui a utilização das duas funções supracitadas.

- PyObject* PyObject_GetAttrString(PyObject *o, const char *attr_name)
  Retorna um atributo de nome *attr_name* do objeto *o*. Retorna o valor do atributo em caso de sucesso, ou NULL em caso de falha.

- PyObject* PyObject_GetAttr(PyObject *o, PyObject *attr_name)
  Retorna um atributo de nome *attr_name* do objeto *o*. Retorna o valor do atributo em caso de sucesso, ou NULL em caso de falha.

- PyObject* PyObject_SetAttr(PyObject *o, PyObject *attr_name, PyObject *v)
  Atribui o valor a um atributo de nome *attr_name* do objeto *o*. Levanta uma exceção e retorna -1 em caso de falha; retorna 0 em caso de sucesso.

  Se *v* é NULL, o atributo é apagado. Análogo à utilização de PyObject_DelAttr().

- PyObject* PyObject_SetAttrString(PyObject *o, const char *attr_name, PyObject *v)
  Retorna um atributo de nome *attr_name* do objeto *o*. Retorna o valor do atributo em caso de sucesso, ou NULL em caso de falha.

- PyObject* PyObject_HasAttr(PyObject *o, PyObject *attr_name)
  Retorna 1 se *o* tem o atributo de nome *attr_name*, e 0 caso contrário.

- PyObject* PyObject_DelAttr(PyObject *o, PyObject *attr_name)
  Deleta o atributo de nome *attr_name*, do objeto *o*. Retorna -1 em caso de falha. Equivalente ao del o.attr_name do Python.

- PyObject* PyObject_DelAttrString(PyObject *o, const char *attr_name)
  Deleta o atributo de nome *attr_name*, do objeto *o*. Retorna -1 em caso de falha. Equivalente ao del o.attr_name do Python.

- PyObject* PyCallable_Check(PyObject *o)
  Determina se o objeto *o* existe. Retorna 1 se ele existe, 0 caso contrário.

- PyObject* PyObject_CallObject(PyObject *callable, PyObject *args)
  Chama um objecto Python de nome *callable*, com argumentos dados pela tupla *args*. Se não houverem argumentos, *args* pode ser NULL. Retorna o resultado em caso de sucesso, ou NULL em caso de falha.

- PyObject* PyObject_Call(PyObject *callable, PyObject *args, PyObject *kwargs)
  Retorna uma nova referência.
  Chama um objecto Python de nome *callable*, com argumentos dados pela tupla *args*

e argumentos nomeados dados pelo dicionário *kwargs*. *args* não pode ser NULL, use uma tupla vazia se argumentos forem desnecessários. Se argumentos nomeados não forem necessários, *kwargs* pode ser NULL

Retorna o resultado em caso de sucesso, ou NULL em caso de falha.

- PyObject* PyObject_CallFunctionObjArgs(PyObject *callable, ..., NULL)
  Chama o objeto Python *callable*, com um número variável de argumentos *PyObject**. O último argumento deve ser NULL.

  Retorna o resultado da chamada em caso de sucesso, ou NULL em caso de falha.

- PyObject* PyObject_CallMethodObjArgs(PyObject *obj, PyObject *name, ..., NULL)
  Chama um método do objeto python *obj*, no qual o nome do método é dado por uma string Python *name*. É chamado com um número variável de argumentos *PyObject**. O último argumento deve ser NULL.

  Retorna o resultado da chamada em caso de sucesso, ou NULL em caso de falha.

- int PyObject_IsInstance(PyObject *inst, PyObject *cls)
  Retorna 1 se *inst* é uma instância da classe ou subclasse *cls*, ou 0 caso contrário. Em caso de erro, retorna -1 e levanta uma exceção.

3.4 Subtipos de objetos Python

3.4.1 Listas

- int PyList_Check(PyObject *p)
  Retorna verdadeiro se *p* é um objeto do tipo lista ou uma instância de um subtipo do tipo lista.

- PyObject* PyList_New(length)
  Retorne uma nova referência.
  Retorne uma nova lista de tamnho *length* em caso de sucesso, ou NULL em caso de falha.

  Observação: se *length* for maior que zero, os ítens da lista de retorno são inicializadas com NULL. Portanto, funções como PySequence_SetItem(), ou passar o objeto para o Python antes de associar cada elemento da lista, não podem ser utilizadas.

- Py_ssize_t PyList_Size(PyObject* list)
  Retorna o tamanho da lista.

- PyObject* PyList_GetItem(PyObject *list, Py_ssize_t index)
  Retorna uma referência emprestada.

Retorna o objeto da posição *index* na lista. A posição deve ser positiva. A função não suporta indexação pelo fim da lista. Se *index* estiver fora dos limites, o retorno é NULL.

- int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)
  Associa um ítem à posição *index* da lista. Retorna 0 em caso de sucesso ou -1 em caso de falha.

  Observação: essa função "rouba"uma referência de *item* e descarta a referência para um ítem que já está na lista na posição indicada.

- int PyList_Insert(PyObject* list, Py_ssize_t index, PyObject *item)
  Insere o ítem *item* na lista na frente da posição *index*. Retorna 0 em caso de sucesso, -1 em caso de falha.

- int PyList_Append(PyObject* list, PyObject *item)
  Insere o ítem *item* no final da lista. Retorna 0 em caso de sucesso, -1 em caso de falha.

3.4.2 Tuplas

- int PyTuple_Check(PyObject *p)
  Retorna verdadeiro se *p* é um objeto do tipo tupla ou uma instância de um subtipo do tipo tupla.

- PyObject* PyTuple_New(length)
  Retorna uma nova referência.
  Retorna uma nova tupla de tamnho *length* em caso de sucesso, ou NULL em caso de falha.

- Py_ssize_t PyTuple_Size(PyObject* p)
  Retorna o tamanho da lista.

- PyObject* PyTuple_GetItem(PyObject *p, Py_ssize_t pos)
  Retorna uma referência emprestada.
  Retorna o objeto da posição *pos* da tupla. A posição deve ser positiva. A função não suporta indexação pelo fim da lista. Se *pos* estiver fora dos limites, o retorno é NULL.

- int PyTuple_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)
  Associa o ítem *o* à posição *pos* da tupla. Retorna 0 em caso de sucesso ou -1 em caso de falha.

  Observação: essa função "rouba" uma referência de *o*.

3.4.3 Dicionários

- int PyDict_Check(PyObject *p)
  Retorna verdadeiro se *p* é um objeto do tipo dicionário.

- PyObject* PyDict_New()
  Retorne uma nova referência.
  Retorne um novo dicionário vazio, ou NULL em caso de falha.

- void PyDict_Clear(PyObject *p)
  Esvazia o dicionário de todos os pares chave-valor.

- int PyDict_Contains(PyObject *p, PyObject *key)
  Determina se *key* está contida no dicionário *p*. Retorna 1 em caso verdadeiro, ou 0 caso contrário. Em caso de erro, retorna -1.

- int PyDict_SetItem(PyObject *p, PyObject *key, PyObject *val)
  Insere o valor *val* no dicionário *p* com a chave *key*.

- int PyDict_SetItemString(PyObject *p, const char *key, PyObject *val)
  Insere o valor *val* no dicionário *p* com a chave *key*. *key* deve ser uma string.

- int PyDict_DelItem(PyObject *p, PyObject *key)
  Remove o valor com a chave *key* no dicionário *p*.

- int PyDict_DelItemString(PyObject *p, const char *key)
  Remove o valor com a chave *key* no dicionário *p*. *key* deve ser uma string.

- PyObject* PyDict_GetItem(PyObject *p, PyObject *key)
  Retorna uma referência emprestada.
  Retorna o objeto do dicionário *p* relativo à chave *key*. Retorna NULL se a chave não está no dicionário.

- PyObject* PyDict_GetItemString(PyObject *p, const char *key)
  Retorna uma referência emprestada.
  Retorna o objeto do dicionário *p* relativo à chave *key*. Retorna NULL se a chave não está no dicionário. *key* deve ser uma string.

- Py_ssize_t PyDict_Size(PyObject *p)
  Retorna o numero de ítens no dicionário.

3.4.4 Array - API do Numpy para C/C++

O Numpy disponibliza uma API para C/C++ habilitando os usuários a ter acesso a objetos do tipo *array*.

Suas principais funções são:

- int PyArray_NDIM(PyArrayObject *arr)
  Retorna o número de dimensões do array.
- npy_intp PyArray_Size(PyArrayObject *obj)
  Retorna 0 se *obj* não for uma subclasse de ndarray. Caso contrário, retorna o número total de elementos no array.
- void* PyArray_GETPTR1(PyArrayObject *obj, npy_intp i)
- void* PyArray_GETPTR2(PyArrayObject *obj, npy_intp i,npy_intp j)
- void* PyArray_GETPTR3(PyArrayObject *obj, npy_intp i, npy_intp j, npy_intp k)
  Acesso rápido ao elemento correspondente às coordenadas i, j e k do objecto *obj*. Utilize *typecast* para converter o apontador de *void* para o tipo de dado apropriado.

3.5 Funções de tratamento de erro e administração de memória

3.5.1 Tratamento de erro

Existem várias funções da API que permitem ao programador "configurar" as exceções de erros causadas por problemas nas demais funções da API. Existe um indicador global (por *thread*) do último erro ocorrido. Quando um indicador é levantado, as funções de erro utilizam-no para imprimir uma mensagem apropriada que ajuda ao programador identificar onde está o erro.

Como a utilização, ou não, desse tipo de função depende de cada programador, este relatório traz apenas a mais simples delas a seguir. Aconselha-se ler a documentação na sessão *Exception Handling* para maiores informações.

- void PyErr_Print()
  Imprime uma mensagem de erro padrão.
  Chamar essa função apenas quando o indicador de erro for ativado.

3.5.2 Administração de memória

Em linguagens como C/C++, o programador é responsável pela alocação e desalocação dinâmica da memória. Se um bloco de memória for alocado, este deve, posteriormente, ser desalocado para que possa ser usado, ou aquele espaço não poderá ser utilizado

até que o programa termine. A isto, dá-se o nome de vazamento de memória (do inglês, *memory leak*).

Para contornar essa problemática, o Python utiliza um método de *contador de referência*, que baseia-se no seguinte princípio: todo objeto tem um contador, que é incrementado quando uma referência a esse objeto é armazenada em algum lugar e que é decrementada quando uma referência a ele é apagada. Quando este contador atinge zero, ou seja, a última referência ao objeto é apagada, o objeto é liberado.

- void Py_INCREF(PyObject *o)
  Incrementa o contador de referência para o objeto *o*. O objeto não deve ser NULL; usar Py_XINCREF(PyObject *o) no caso dessa possibilidade.

- void Py_DECREF(PyObject *o)
  Decrementa o contador de referência para o objeto *o*. O objeto não deve ser NULL; usar Py_XDECCREF(PyObject *o) no caso dessa possibilidade.

4 Exemplo

4.1 Aplicação: FFT (*Fast Fourier Transform*)

Na área de comunicações ópticas, a Transformada Discreta de Fourier (DFT - *Discrete Fourier Transform*) é certamente uma ferramenta matemática muito utilizada para o processamento digital de sinais. Para o cálculo dela, projetistas utilizam os algoritmos de FFT que são preferidos devido à sua eficiência pois a complexidade no cálculo da DFT diminui de N^2 para $Nlog_2N$.

Estes algoritmos são baseados na decomposição do cálculo de uma DFT de uma sequência de tamanho N em DFTs de sequências menores combinadas (Figura 4).

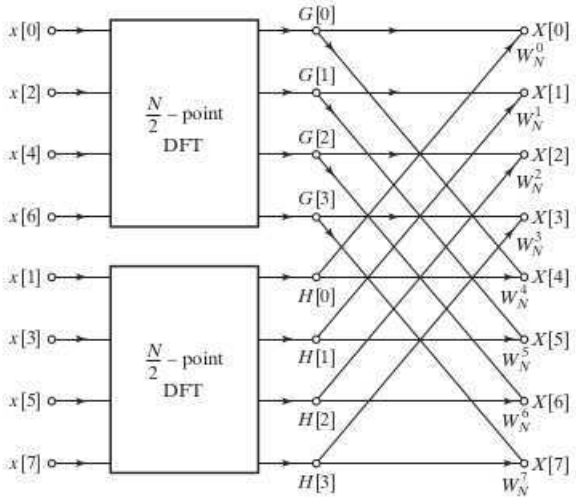

Figura 4 – Estrutura de decimação no tempo de uma DFT de tamanho 8 em duas de tamanho 4 (OPPENHEIM; SCHAFER, 1989).

Essa ferramenta matemática pode, muitas vezes, ser de difícil implementação em uma linguagem como C/C++, mas a partir dos conhecimentos da API Python/C pode-se utilizar a biblioteca *Numpy* que já tem uma função para o cálculo da mesma. Portanto, a implementação de uma FFT resume-se em fazer uma conexão de C para Python.

Código Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def fft_call(i_data):
5   result = np.fft.fft(i_data)
6   return result
```

Código C++

```
#include <iostream>
#include <fstream>
#include "Python.h"
#include "arrayobject.h"
#include <stdlib.h>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {

    Py_Initialize(); // inicializacao do interpretador Python

    PyObject *pName, *pModule, *pFunc;
    PyObject *pList;
    PyObject *pOut1;

    int list_size = 5000;

    std::string file_x_in;
    file_x_in = "input_file.txt";

    std::ifstream file_x(file_x_in.c_str());
    double input;
    pModule = PyImport_ImportModule("fft_tcc");

    if(pModule != NULL)
    {
        pFunc = PyObject_GetAttrString(pModule, "fft_call");
        if(pFunc && PyCallable_Check(pFunc))
        {

            pList = PyList_New(list_size);

            for(int pos = 0; pos < list_size; pos++)
            {
                file_x >> input;
                PyList_SetItem(pList, pos, Py_BuildValue("f", input ));

            }

            // call function
            pOut1 = PyObject_CallFunctionObjArgs(pFunc, pList, NULL);

            // esse metodo precisa que a saida seja um numpy array
            PyArrayObject *npArray0 = (PyArrayObject*)(pOut1);
            void* npArray0_ptr;
```

```
47
48              for (int i=0; i < list_size; i++)
49              {
50                  npArray0_ptr = PyArray_GETPTR1(npArray0, i);
51                  // cout << "---> " << *(double*)(npArray0_ptr) << endl;
52              }
53
54          }
55          else{
56              printf("ERRO. Class not imported()\n");
57              PyErr_Print();
58          }
59
60      }else{
61          printf("ERRO. Module not imported\n");
62          PyErr_Print();
63      }
64
65
66      Py_Finalize();
67      return(0);
68  }
```

4.2 Exemplos gerais

Considerando que a API Python/C pode ser utilizada para vários tipo de projetos e atividades. A seguir estão dois exemplos apenas mostrando como funcionam os principais métodos.

Código Python:

```
1  import numpy as np
2
3
4  class general_ex(object):
5      """docstring for general_example"""
6      def __init__(self, param1 = 0, param2 = 0) :
7          print("Construtor")
8          print(param1)
9          print(param2)
10         self.out_list = np.array(10)
11         self.out_scalar = 0
12
13
14     def def1(self, in_list, in_scalar):
15         self.out_list = np.array(in_list)
16         self.out_scalar = in_scalar
```

```
17          print(in_list)
18          print(in_scalar)
19
20          return np.array(self.out_list)
21
22      def def2(self, arg1 = 0, arg2 = 0):
23          out1 = np.array(arg1)
24          print(np.shape(out1))
25          out2 = np.array(arg2)
26          print(out1)
27          print(out2)
28
29          return {"output1": out1, "output2": out2}
```

Instanciando o método "def1". Utilizando listas e passando parâmetros por posição.

```
1  #include <iostream>
2  #include "arrayobject.h"
3  using namespace std;
4
5  int main(int argc, char* argv[]) {
6
7      Py_Initialize(); // inicializacao do interpretador Python
8
9      PyObject *pName, *pModule, *pClass, *pInstance;
10     PyObject *pDef1;
11     PyObject *pList, *pScalar;
12     PyObject *pOut1;
13
14     int list_size = 10;
15
16     pName = PyUnicode_FromString("general_example"); //build the name
       object
17     pModule = PyImport_Import(pName); //load the module object
18     // pModule = PyImport_ImportModule("general_example");
19
20     if(pModule != NULL)
21     {
22         pClass = PyObject_GetAttrString(pModule, "general_ex");
23         if(pClass && PyCallable_Check(pClass))
24         {
25             // call constructor
26             pInstance = PyObject_CallObject(pClass,NULL);
27
28             pList = PyList_New(list_size);
29
30             for(int pos = 0; pos < list_size; pos++)
31             {
```

```
32                 PyList_SetItem(pList, pos, Py_BuildValue("f", pos + 0.2 ));
33             }
34 
35             pScalar = Py_BuildValue("f", 52.3);
36 
37             pDef1 = Py_BuildValue("s", "def1");
38             pOut1 = PyObject_CallMethodObjArgs(pInstance, pDef1, pList,
   pScalar, NULL);
39 
40 
41             // cast output: perigoso por forcar a saida para o tipo
   requerido
42             double *Out = reinterpret_cast<double*>(PyArray_DATA(
   reinterpret_cast<PyArrayObject*>(pOut1)));
43             for (int i=0; i < list_size; i++)
44             {
45                 cout << "-> " << Out[i]<<endl;
46             }
47 
48 
49             // esse metodo precisa que a saida seja um numpy array
50             PyArrayObject *npArray0 = (PyArrayObject*)(pOut1);
51             void* npArray0_ptr;
52 
53             for (int i=0; i < list_size; i++)
54             {
55                 npArray0_ptr = PyArray_GETPTR1(npArray0, i);
56                 cout << "----> " << *(double*)(npArray0_ptr) << endl;
57             }
58 
59 
60             // extrair atributos da classe diretamente por nome
61             PyObject *pOut_scalar = Py_BuildValue("s", "out_scalar");
62             if(PyObject_HasAttr(pInstance,pOut_scalar))
63             {
64                 PyObject *ret_attr_scalar = PyObject_GetAttr(pInstance,
   pOut_scalar);
65                 double *ret_attr_scalar_out = reinterpret_cast<double*>(
   PyArray_DATA(reinterpret_cast<PyArrayObject*>(ret_attr_scalar)));
66 
67                 cout << "----------> " << PyFloat_AsDouble(ret_attr_scalar)
   <<endl;
68             }
69 
70 
71             // extrair atributos da classe diretamente por nome
72             PyObject *pAttr = Py_BuildValue("s", "out_list");
```

```
73              void* npArray_list;
74              if(PyObject_HasAttr(pInstance,pAttr))
75              {
76                  PyObject *ret_attr = PyObject_GetAttr(pInstance, pAttr);
77                  PyArrayObject *npArrayList = (PyArrayObject*)(ret_attr);
78                  for (int i=0; i < list_size; i++)
79                  {
80                      npArray_list = PyArray_GETPTR1(npArrayList,i);
81                      cout << "-> " << *(double*)(npArray_list) <<endl;
82                  }
83
84              }
85
86
87
88          }
89          else{
90              printf("ERRO. Class not imported()\n");
91              PyErr_Print();
92          }
93
94      }else{
95          printf("ERRO. Module not imported\n");
96          PyErr_Print();
97      }
98
99
100     Py_Finalize();
101     return(0);
102 }
```

Instanciando o método "def2". Passando parâmetros por nome através da utilização de dicionários.

```
1  #include <iostream>
2  #include "arrayobject.h"
3  using namespace std;
4
5  int main(int argc, char* argv[]) {
6
7      Py_Initialize(); // inicializacao do interpretador Python
8
9      PyObject *pName, *pModule, *pClass, *pInstance;
10     PyObject *pDict_param, *pDict_method;
11     PyObject *pDef2;
12     PyObject *pList_param;
13     PyObject *pOut2;
14
```

```
15      int list_size = 10;
16
17      pName = PyUnicode_FromString("general_example"); //build the name
        object
18      pModule = PyImport_Import(pName); //load the module object
19      // pModule = PyImport_ImportModule("general_example");
20
21      if(pModule != NULL)
22      {
23          pClass = PyObject_GetAttrString(pModule, "general_ex");
24          if(pClass && PyCallable_Check(pClass))
25          {
26              pList_param = PyList_New(list_size);
27
28              for(int pos = 0; pos < list_size; pos++)
29              {
30                  PyList_SetItem(pList_param, pos, Py_BuildValue("f", pos +
        0.2 ));
31              }
32
33              pDict_param = PyDict_New();
34              cout << "Set param 1 = " << PyDict_SetItemString(pDict_param, "
        param1", pList_param) << endl;
35              cout << "Set param 2 = " << PyDict_SetItemString(pDict_param, "
        param2", Py_BuildValue("i", 98)) << endl;
36              // call constructor
37              pInstance = PyObject_Call(pClass, PyTuple_New(0), pDict_param);
38
39
40              pDict_method = PyDict_New();
41              cout << "Set arg 1 = " << PyDict_SetItemString(pDict_method, "
        arg1", pList_param) << endl;
42              cout << "Set arg 2 = " << PyDict_SetItemString(pDict_method, "
        arg2", Py_BuildValue("f", 0.78)) << endl;
43
44              pDef2 = PyObject_GetAttrString(pInstance, "def2");
45              pOut2 = PyObject_Call(pDef2, PyTuple_New(0), pDict_method);
46
47              cout << "A saida eh um dicionario: " << PyDict_Check(pOut2) <<
        endl;
48
49              cout << "O dicionario contem output1: " << PyDict_Contains(
        pOut2, Py_BuildValue("s", "output1")) <<endl;
50              cout << "O dicionario contem output2: " << PyDict_Contains(
        pOut2, Py_BuildValue("s", "output2")) <<endl;
51
52              PyObject *output1 = PyDict_GetItemString(pOut2, "output1");
```

```
53              PyObject *output2 = PyDict_GetItemString(pOut2, "output2");
54
55              cout << "Numero de dimensoes de output1: " << PyArray_NDIM(
    output1) << endl;
56              cout << "Numero de dimensoes de output2: " << PyArray_NDIM(
    output2) << endl;
57
58              npy_intp *array_shape;
59              array_shape = PyArray_SHAPE((PyArrayObject*)output1);
60              cout << "dimensao do array: " << *array_shape << endl;
61
62
63
64          }
65          else{
66              printf("ERRO. Class not imported()\n");
67              PyErr_Print();
68          }
69
70      }else{
71          printf("ERRO. Module not imported\n");
72          PyErr_Print();
73      }
74
75
76      Py_Finalize();
77      return(0);
78  };
```

5 Considerações Finais

Com este trabalho, foi possível analisar as vantagens da utilização da linguagem de programação Python no fluxo de projeto. Como o Matlab, o Python tem bibliotecas com funções apropriadas para o processamento de sinais, além da vantagem de não precisar de licenciamento.

A partir dessa migração de linguagens, percebe-se uma melhoria no que diz respeito ao consumo de memória visto que não há a necessidade de um constante armazenamento de arquivos de dados de entradas e saídas para posteriores comparações via *testbench*.

Além disso, uma nova estrutura de ambiente de verificação pôde ser implementada a partir da utilização da API Python/C. Esta estrutura consiste na conexão direta, Figura 3, entre o gerador de dados e os modelos que deseja-se comparar.

Referências

OPPENHEIM, A. V.; SCHAFER, R. W. *Discrete-Time Signal Processing*. Third edition. [S.l.]: Pearson, 1989. Citado 2 vezes nas páginas 8 e 15.

PEIXOTO, J. L. *Design físico de um IP*. Tese (Graduation dissertation) — Universidade Federal de Campina Grande, 2018. Citado na página 1.

PYTHON. *Python/C API Reference Manual*. 2018. Disponível em: <https://docs.python.org/3.6/c-api/index.html>. Citado 2 vezes nas páginas 6 e 7.

SMITH, S. W. *The Scientist and Engineer´s Guide to Digital Signal Processing*. [S.l.: s.n.], 1997. Citado na página 4.