pamPython: proposta de um processador para executar algoritmos *Python*

Tulio P. Bitencourt[1], **Bruno S. Neves**[1]

[1]Universidade Federal do Pampa – (UNIPAMPA) – Bagé, RS – Brasil
Av. Maria Anunciação Gomes Godoy, 1650, Bagé – RS – CEP: 96460-000

tulio.pereirab@gmail.com, brunoneves@unipampa.edu.br

Abstract. *This paper describes the development process of a processor able to executes algorithm written in Python. This processor was developed using the description hardware language called VHDL and its main objective was follow the Python documentation and execute its respective Assembly code. It was reached, as result for this first version, a function general purpose architecture.*

Resumo. *Este artigo descreve o processo de desenvolvimento de um processador capaz de executar algoritmos escritos em Python. Esse processador foi desenvolvido utilizando-se a linguagem de descrição de hardware chamada VHDL e seu principal objetivo era seguir a documentação Python e executar seu respectivo código Assembly. Foi alcançado, como resultado desta primeira versão, uma arquitetura de propósito geral funcional.*

1. Introdução

A linguagem *Python* é uma das linguagens de programação mais utilizadas no mundo [Sandler 2018], sendo empregada em diversas áreas da ciência, que vão desde a neurociência [Peirce 2007] até inteligência artificial [Pedregosa et al. 2012], passando, inclusive, pela área de exploração de árvores [Huerta-Cepas et al. 2010]. Sua fama é decorrente, principalmente, de sua elevada portabilidade e simplicidade, tendo um algoritmo *Python* entre um terço a um quinto do tamanho de um algoritmo similar desenvolvido em C, C++ ou *Java* [Lutz 2007].

Python é considerada uma linguagem de programação interpretada, visto que possui um interpretador, desenvolvido em C, chamado *CPython* [Cannon 2005]. Esse interpretador, além de executar os algoritmos desenvolvidos, também age como compilador e apresenta ao usuário, por meio de uma biblioteca chamada *Disassembler* [Python 2017], o código *Assembly* que melhor representa o algoritmo. Com este projeto, tem-se como objetivo criar uma alternativa para a execução de algoritmos escritos em *Python* além da máquina virtual, utilizando uma arquitetura física. Existem diversas propostas de arquiteturas que visam criar alternativas para a execução de algoritmos *Java*, mas nem tantas opções para algoritmos *Python*. Este artigo descreve o desenvolvimento da versão inicial do um processador *Python*, a qual suporta funcionalidades básicas e pode ser considerada de propósito geral, não enfatizando nenhuma área, biblioteca ou algoritmo específico.

O restante deste artigo está organizado da seguinte forma: Seção 2: trabalhos correlatos; Seção 3: escolha das instruções a serem suportadas; Seção 4: projeto do RTL; Seção 5: implementação; e Seção 6: resultados e discussão.

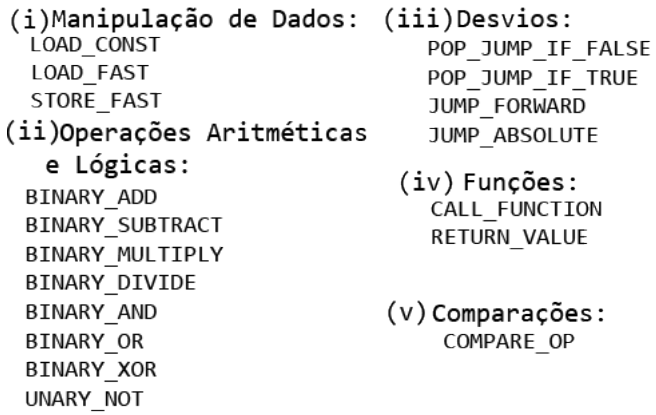

Figura 1. Instruções a serem suportadas no processador.

2. Trabalhos correlatos

Uma das motivações para esse projeto foi o fato de existirem diversas alternativas em *hardware* com foco na linguagem *Java* mas não na linguagem *Python*. Assim é possível perceber que os trabalhos correlatos apresentados foram feitos para a linguagem *Java*.

O primeiro deles, considerado o mais semelhante com esse, é o *picoJava-I* [O'Connor and Tremblay 1997]. Ele é um processador de propósito geral que suporta 341 instruções *Assembly*. Sua frequência máxima de operação foi limitada a 100 MHz.

Os outros trabalhos correlatos possuem a mesma ideia aplicada ao *picoJava-I*, porém não são de propósito geral. Em sua maioria, são processadores que visam a execução de sistemas de tempo real, que não podem tolerar qualquer atraso. Esses trabalhos receberam os nomes de JAIP [Tsai et al. 2015a], JAIP-MP [Tsai et al. 2015b] e JOP [Schoeberl 2008]. A eles, são atribuídas funcionalidades específicas da linguagem *Java* e são utilizadas abordagens diferentes.

Dentre as diferenças esperadas entre este projeto *Python* para os processadores *Java* apresentados, pode-se citar a utilização de técnicas de *pipelining* e *multithreading* nos outros e não neste. Contudo, por se tratar da primeira versão, tais técnicas não foram consideradas neste momento, mas passam a ser objetivo para implementações futuras.

3. Escolha das instruções a serem suportadas

A primeira fase do projeto se define por ser o estudo das instruções *Assembly* da linguagem *Python* e, posteriormente, a escolha das instruções que seriam suportadas nesta primeira versão do processador *Python*.

Para isso, foram desenvolvidos algoritmos considerados de baixa complexidade com o único objetivo de que fossem analisadas as instruções *Assembly* que compunham esses algoritmos no baixo nível. Dentre os algoritmos, podem ser encontrados fluxos de repetição e comparações, assim como operações aritméticas e lógicas.

Após as análises, foram escolhidas as instruções, que são apresentadas na Figura 1, que iriam compor a primeira versão do processador. Na figura, as instruções aparecem divididas por classes, sendo que cada uma representa uma arquitetura parcial, termo que será apresentado na Seção 4.

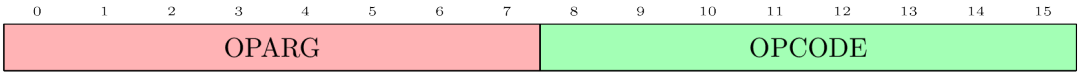

Figura 2. Formato das instruções *Python*.

As instruções apresentadas se classificam nas classes de (i) manipulação de dados, (ii) operações aritméticas e lógicas, (iii) desvios, (iv) funções e (v) comparações. Além de analisar o funcionamento de cada uma das instruções, é necessário também compreender o formato delas, o qual é apresentado na Figura 2 [Ike-Nwosu 2015].

4. Projeto do RTL

Após a conclusão da fase de análise e escolha das instruções, foi necessário que fosse projetada uma arquitetura capaz de suportar a execução de todas as instruções escolhidas. Para isso, utilizou-se uma abordagem que visou a criação de arquiteturas parciais que executam separadamente cada uma das classes de instruções apresentadas.

Porém, além das cinco arquiteturas parciais necessárias para as classes, foi necessário projetar outra capaz de realizar o controle do processador. Assim, as funcionalidades básicas do processador, como o *program counter* (PC) e o recebimento das instruções e argumentos ficou a cargo dessa nova arquitetura parcial, apresentada na Figura 3. O detalhe dessa arquitetura parcial é que existe o registrador de desvio (*regJump*) que é utilizado para gerar uma saída definida pela concatenação de uma palavra de memória completa (16 *bits*) com o argumento da instrução (8 *bits*), permitindo que as memórias possam ser endereçadas com até 24 *bits*.

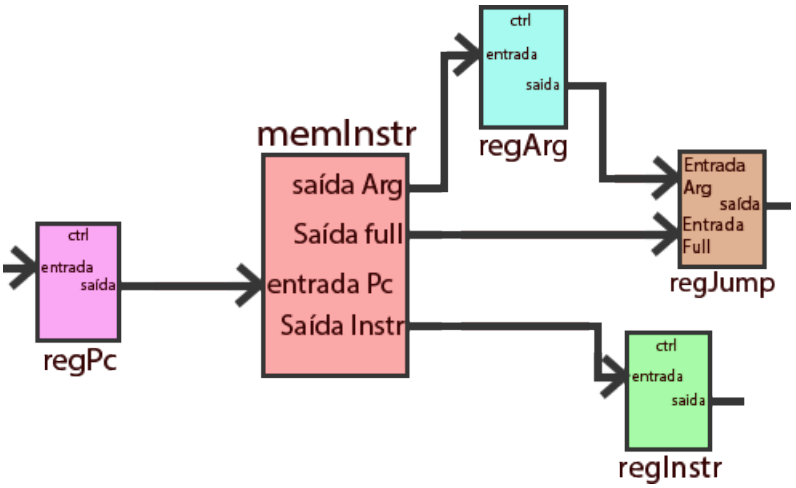

Figura 3. Arquitetura parcial responsável por funcionalidades básicas (controle de PC, recebimento de instruções e argumentos).

Após a criação de todas as arquiteturas parciais, foi necessário que elas fossem unidas, permitindo a obtenção de apenas uma arquitetura completa. Assim, criou-se o esboço apresentado na Figura 4, o qual mostra a arquitetura implementada para a primeira versão do *pamPython*. Percebe-se, na figura, a existência de multiplexadores, que foram adicionados para permitir a reutilização de componentes e reduzir a área ocupada pela arquitetura no FPGA.

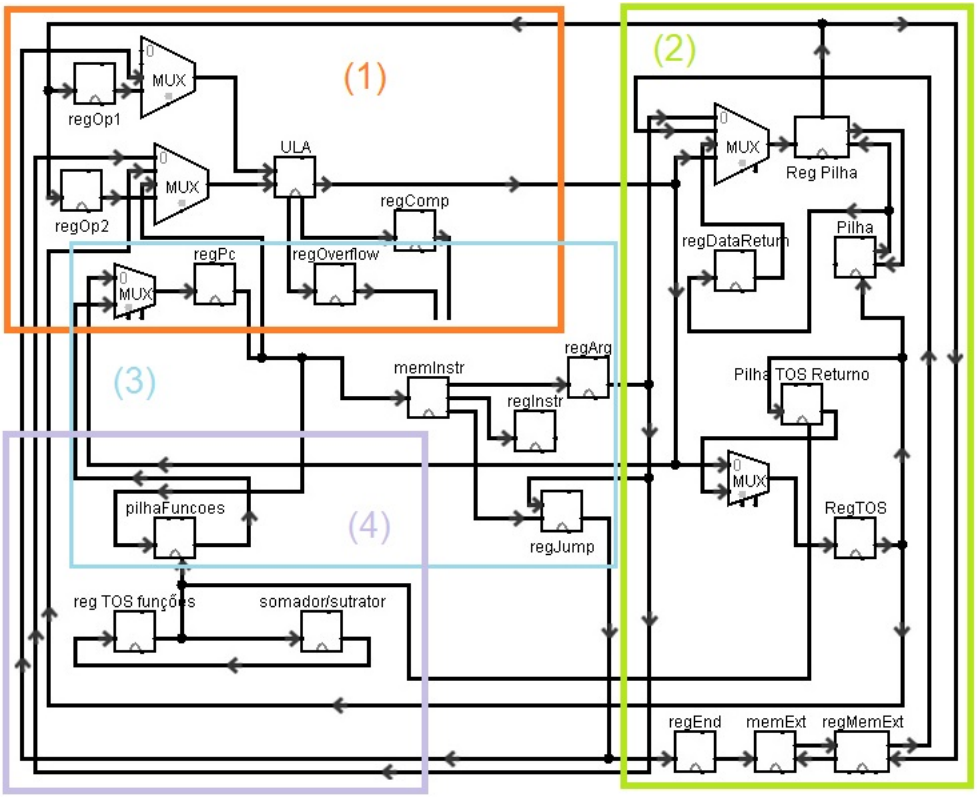

Figura 4. Esboço da arquitetura final para a primeira versão.

A Figura 4 é dividida em quatro grandes blocos. O primeiro deles, localizado na parte superior esquerda (1), é o responsável por realizar operações aritméticas e lógicas. Ele é composto pela unidade lógica aritmética (ULA) e possui conexão com o segundo bloco. Esse, por sua vez, localizado ao lado direito da arquitetura (2), é definido como sendo responsável por interações com as memórias. É função desse bloco gerenciar a pilha e a memória externa, sendo essa última responsável pelo armazenamento de variáveis.

Na parte central da figura (3), é possível visualizar a única arquitetura parcial que foi apresentada neste artigo (Figura 3). Já na parte inferior esquerda da arquitetura (4), encontra-se o bloco responsável pelas instruções de chamada e retorno de funções.

A manipulação de funções é realizada salvando-se, durante a chamada de uma função, os valores atuais de PC e do indicador de topo da pilha (TOS). Já no retorno, ocorre a inserção dos valores salvos anteriormente nos registradores PC (*regPc*) e TOS (*regTos*). O bloco 4, contudo, apresenta apenas parte dessa funcionalidade. Os valores de retorno relacionados à pilha dependem de dois componentes localizados no bloco 2 (*regDataReturn* e "pilha TOS retorno"). O primeiro sendo responsável por retornar um valor vindo da função e o segundo sendo responsável por salvar os valores indicadores de TOS durante a chamada de uma função.

5. Implementação

5.1. Implementação do *Hardware*

Para iniciar a fase de implementação, foram definidas quais seriam as ferramentas utilizadas, tanto para a implementação quanto para a prototipação no FPGA e

```
1b: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
17: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
13: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
 f: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
 b: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
 7: 0000000000000000 0000000000000000 0000000000000000 0000000000001111
 3: 0000000000100000 0000001111101000 0000000001100000 0000111100001100
```

Figura 5. Formato do arquivo *.mem*.

realização das simulações. Assim, primeiramente, definiu-se que a ferramenta *Quartus* [Intel 2018b] seria utilizada para realização da prototipação no FPGA *Cyclone III* número EP3C25F324C6. Para a realização das simulações, foi utilizada a ferramenta [Intel 2018a]. O principal motivo que levou a escolha dessas ferramentas foi o fácil acesso do autor a todas elas.

Tendo como objetivo seguir rigorosamente o esboço apresentado anteriormente, na Figura 4, esperava-se que todas as instruções funcionassem corretamente. Assim, a fase implementação iniciou-se pela arquitetura parcial responsável pela funcionalidade básica do processador, composta por uma memória de instruções de até 24 *bits* de endereçamento, podendo esse valor ser alterado, juntamente com quatro registradores responsáveis pelo armazenamento do valor de PC, código da instrução, argumento e, por último, o valor concatenado de uma palavra de memória com o argumento.

Ao final da implementação das arquiteturas parciais, iniciou-se a implementação da unidade de controle. A criação dessa máquina de estados se baseou na mesma abordagem de dividir a complexidade e, nesse caso, foram criadas máquinas de estados para cada uma das instruções e, posteriormente, uniu-se todas elas. Em linhas gerais, a máquina de estados conta com 120 estados que são executados em sequências máximas de 12 estados, sendo cada sequência representando uma instrução específica.

5.2. Implementação de um conversor

Para executar algoritmos com maior agilidade, não sendo necessária a digitação do código binário na memória de instruções, e evitar que erros fossem comentidos no momento de inserir os códigos binários nos arquivos de inicialização da memória de instruções, foi desenvolvido um conversor. Esse conversor recebe as instruções *Assembly*, tais como as apresentadas na Figura 1, e as converte em código binário.

A atribuição dos códigos binários para cada uma das instruções escolhidas foi feita pelo autor deste artigo, visto que não foi encontrada qualquer menção de códigos binários padrões para a linguagem *Python* propostos pela organização mantenedora.

Assim, a função do conversor é receber os códigos *Assembly* e gerar os arquivos de inicialização da memória, tanto para simulações quanto para execuções no FPGA. O arquivo responsável por inicializar a memória em simulações no *Modelsim* é no formato *.mem*, cujo exemplo é apresentado na Figura 5. Já o arquivo responsável por simulações no *Quartus* e execuções no FPGA é no formato *.mif*, cujo exemplo é apresentado na Figura 6.

5.3. Verificação de erros

Além das implementações do *hardware* e do *software*, foi necessário ainda adicionar meios de reconhecer quando existem erros no algoritmo que possam tornar a execução

```
WIDTH=16;
DEPTH=4096;

ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;

CONTENT BEGIN
        0       :       0000111100001100;
        1       :       0010111000001100;
        2       :       0000000000100000;
        3       :       0001010000001100;
        4       :       0001100100000010;
        5       :       0000000000110001;
        6       :       0000000000001010;
        7       :       1111111100001100;
        8       :       0000000000000000;
        9       :       0000000000000000;
        10      :       1111111100001100;
        [11..4095]      :       0000000000000000;
END;
```

Figura 6. Formato do arquivo *.mif*.

Tabela 1. Códigos dos erros verificados em *hardware*.

CÓDIGO	SIGNIFICADO	CLASSIFICAÇÃO
11111111	Ponteiro de TOS inválido	Crítico
00111100	Instrução inexistente	Crítico
11000011	Endereço de memória inválido	Alerta
00001001	*Overflow* de soma	Alerta
00001010	*Overflow* de subtração	Alerta
00001011	*Overflow* de multiplicação	Alerta
00001100	*Overflow* de divisão	Alerta
00111100	Desvio condicional sem comparação	Alerta

sobre a arquitetura inválida. O mais clássico deles é o da leitura na pilha sem que haja um empilhamento antes, tornando o ponteiro da pilha um valor negativo e, portanto, inválido.

Assim, foram adicionadas três camadas de verificação de erros, sendo duas em *software* e a última em *hardware*. A primeira delas, em *software*, é a verificação de erros de digitação ou falta de argumento. Essa camada também verifica a validade do argumento, ou seja, se é ou não suportado em 8 *bits* quando for um dado ou em 24 *bits* quando for um endereço.

A segunda camada, e última em *software*, nada mais é do que um simulador da própria arquitetura física que realiza a execução de cada uma das instruções exatamente como a arquitetura o fará. Esse simulador possui os mesmos sinais internos da arquitetura e, portanto, essa verificação de erros é feita através da análise desses sinais. Caso algum algoritmo torne um desses sinais inválidos, existe uma enorme probabilidade de que esse mesmo sinal fique inválido quando o algoritmo for executado na arquitetura.

A última camada de verificação de erros, implementada em *hardware*, é definida como sendo o monitoramento, por parte da unidade de controle, dos sinais básicos da arquitetura. Ao perceber que existe problema em algum dos sinais, um código de erro contendo 8 *bits* é salvo em um registrador chamado *regError*, o qual indicará qual erro ocorreu. Os códigos de erros são definidos pela Tabela 1.

Cada um dos erros também recebe uma classificação, podendo ser um erro crítico ou apenas um alerta (*warning*). No caso de um erro crítico, o processador em *hardware* é finalizado e passa a operar em um estado de erro até que haja uma reinicialização (*reset*). Já um alerta nada mais é do que um aviso de que o resultado apresentado pode estar inválido.

6. Resultados e Discussão

Ao final da implementação, foi possível obter um processador capaz de executar todas as funcionalidades previstas inicialmente. Essas funcionalidades são: execução de comparações, desvios condicionais e incondicionais, operações aritméticas e lógicas, e chamada e retorno de funções. Para realizar a validação da execução, foram desenvolvidos algoritmos *Python* e, utilizando-se a ferramenta *Disassembler*, obteve-se seus códigos *Assembly*. Posteriormente, esses códigos *Assembly* foram inseridos no conversor e gerou-se os arquivos de inicialização de memória de instruções. Assim, considerou-se como uma correta execução quando os resultados encontrados em *software* e hw eram os mesmos.

Além disso, as características básicas do processador criado aqui podem ser comparadas com as dos trabalhos correlatos apresentados. Então, foi encontrado como frequência máxima de operação o valor de 84,35 MHz, se mantendo logo pouco abaixo do *picoJava-I*. Mas é importante salientar que nessa versão não houve qualquer refinamento no projeto visando o aumento da frequência.

Quanto ao valor correspondente à área ocupada no FPGA, pode-se considerar que foi em torno de 6% da capacidade total do dispositivo utilizado. Tal valor foi retirado diretamente da ferramenta *Quartus*, a qual indica, por exemplo, um número total de 728 elementos lógicos na arquitetura. Outra informação importante de um projeto de *hardware* é a potência térmica dissipada. Nesse processador, foi encontrado um consumo de 132,56 mW.

Ao realizar as simulações, foram comparados o resultado esperado com o resultado encontrado. Um exemplo disso é uma operação aritmética utilizado-se um valor retornado de uma função como sendo o segundo operando. Caso o resultado final, o qual foi salvo na memória externa, não correspondesse ao resultado esperado, considerar-se-ia que o resultado estava inválido e, portanto, a arquitetura estava com algum problema.

Ao final deste trabalho, foi possível verificar que todas as funcionalidades implementadas estavam sendo corretamente executadas. Assim, considerou-se que este processador estava funcional.

7. Considerações Finais

Como dito anteriormente, a implementação deste processador é considerada uma implementação básica de um processador *Python*. Em outras palavras, essa é a primeira versão de uma implementação que poderá ser ainda maior. Ela não possui nenhum foco em alguma área específica e, portanto, pode ser considerada como sendo de propósito geral.

Essa arquitetura foi projetada para receber o acoplamento de novos módulos, que poderão ser criados visando executar algoritmos específicos. Um exemplo disso são algoritmos que utilizam matrizes e vetores para serem executados. Na primeira versão,

esses algoritmos não são suportados, mas com o acoplamento de um módulo específico contendo as instruções responsáveis por essa funcionalidade será possível executar tais algoritmos.

Portanto, este projeto serve de ponto de partida para a implementação de uma arquitetura *Python* mais completa, formada pelo núcleo básico (já desenvolvido) e componentes avançados para suporte à novas funcionalidades ou aprimoramento do desempenho. Neste momento, tal realidade ainda é distante, contudo ainda considera-se como viável.

Referências

Cannon, B. (2005). Design of the CPython Compiler.

Huerta-Cepas, J., Dopazo, J., and Gabaldón, T. (2010). ETE: A python Environment for Tree Exploration. *BMC Bioinformatics*, 11.

Ike-Nwosu, O. (2015). *Inside the Python Virtual Machine*. Lean Publishing.

Intel (2018a). ModelSim*-Intel® FPGA Edition Software.

Intel (2018b). Power Analysis and Optimization User Guide Intel Quartus Prime Pro Edition. *Power Analysis and Optimization User Guide Intel Quartus Prime Pro Edition*, 18.

Lutz, M. (2007). *Learning Python: Powerful Object-Oriented Programming*. Number 1. O'Reilly Media, Inc., Sebastopol, CA, USA, 3 edition.

O'Connor, J. and Tremblay, M. (1997). picoJava-I: the Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É. (2012). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Peirce, J. W. (2007). PsychoPy–Psychophysics software in Python. *Journal of neuroscience methods*, 162(1-2):8–13.

Python (2017). 31.12. dis - Disassembler for Python bytecode.

Sandler, R. (2018). The 14 most popular programming languages, according to a study of 100,000 developers. *Business Insider*.

Schoeberl, M. (2008). A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1-2):265–286.

Tsai, C.-J., Kuo, H.-W., Lin, Z., Guo, Z.-J., and Wang, J.-F. (2015a). A Java Processor IP Design for Embedded SoC. *ACM Transactions on Embedded Computing Systems*, 14(2):1–25.

Tsai, C.-J., Wu, T.-H., and Su, H.-C. (2015b). JAIP-MP: A four-core Java application processor. In *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, volume 2015-Octob, pages 189–194. IEEE.