ANALISE DO CÓDIGO ASSEMBLY DA LINGUAGEM PYTHON PARA FUTURA IMPLEMENTAÇÃO DE UM PROCESSADOR

Tulio Pereira Bitencourt [1]

Tulio Bitencourt [2]

Bruno Silveira Neves [3]

Resumo:

Este trabalho se baseia em uma análise sobre o que é e como é feita a conversão de um algoritmo escrito em Python, ou seja, em alto nível, para um código Assembly da mesma linguagem, baixo nível. Nele, são apresentados conceitos básicos como o que é um código Assembly e como ele funciona, assim como instruções específicas do código Assembly da linguagem Python. Como tal linguagem é executada utilizando um interpretador, é possível tornar este interpretador como sendo um processador físico. Logo, com esse objetivo, está em desenvolvimento um protótipo de processador, implementado em VHDL e executado em FPGA, capaz de executar algoritmos Python. Porém, este artigo se baseia em apenas uma parte deste projeto, que é a análise das instruções e código Assembly que o interpretador lê e executa. Portanto, antes de iniciar-se um desenvolvimento e um processador, é necessário sempre definir seu código Assembly específico, pois sua arquitetura deverá ser desenvolvida especificamente para executar as instruções contidas nele. Cada processador possui o seu código Assembly, com instruções definidas e diferentes, porém, a maioria deles possui instruções básicas que são semelhantes a todos os outros.

Palavras-chave: Assembly Python Bytecode Processador FPGA Programação Unipampa

Modalidade de Participação: Iniciação Científica

ANALISE DO CÓDIGO ASSEMBLY DA LINGUAGEM PYTHON PARA FUTURA IMPLEMENTAÇÃO DE UM PROCESSADOR

[1] Aluno de graduação. tulio.pereirab@gmail.com. Autor principal

[2] Aluno de graduação. tulio.pereirab@gmail.com. Apresentador

[3] Docente. brunoneves@unipampa.edu.br. Orientador

ANALISE DO CÓDIGO ASSEMBLY DA LINGUAGEM PYTHON PARA FUTURA IMPLEMENTAÇÃO DE UM PROCESSADOR

1 INTRODUÇÃO

Este artigo apresenta uma análise sobre o código *Assembly* (linguagem de programação que se baseia em um conjunto de instruções) responsável pela execução interna da *Máquina Virtual Python* (PVM) (IKE-NWOSU, 2015). Desempenhar tal análise é de fundamental importância para que seja realizada uma implementação de um processador, em VHDL, capaz de executar algoritmos escritos em *Python*. Uma implementação física da linguagem *Python* pode ser capaz de melhorar as taxas de desempenho em aplicações pesadas.

Indubitavelmente, a tecnologia está crescendo cada vez mais a cada dia. Todas as áreas que rodeiam a vida de qualquer pessoa, tais como a de telefonia, computadores pessoais, sistemas de lojas, sistemas de gerenciamento de empresas, algoritmos que visem a execução de cálculos complexos e etc., estão em constante evolução e não param de se modernizar. O grande pilar que rege tal evolução, e no qual ela se apoia fortemente, é a programação. É através dela que todos temos acesso a sistemas que, por sua vez, substituem diversas tarefas que demandariam, caso não fossem automatizadas, de profundo conhecimento técnico e tempo para serem realizadas.

Dentro da área da computação responsável pela programação, encontram-se as linguagens de programação, que servem para que rotinas de execução possam ser descritas de forma a poderem ser executadas por um computador. Tais linguagens são comumente classificadas como linguagens de alto nível ou baixo nível, sendo que na primeira classe se encaixam linguagens que possuem grande abstração e, por conseguinte, têm seu uso facilitado. Já na segunda classificação, podem ser encontradas linguagens que estão fortemente relacionadas a conceitos puramente físicos e lógicos, sendo, portanto, de mais complexo entendimento.

A linguagem de programação chamada de *Python* é classificada como sendo uma linguagem de alto nível. Ela foi criada em 1990 e lançada em 1991 pelo seu criador, Guido Van Rossum. Ela é, assim como a linguagem Java, considerada uma linguagem interpretada. Isso porque, diferentemente de linguagens como C, ela não é executada diretamente em um processador físico (Intel, AMD, etc.), mas sim em uma máquina virtual (VM – *Virtual Machine*). Logo, para um algoritmo *Python* ser executado, ele precisa ser escrito em alto nível, em *Python*, e ser convertido para uma linguagem de baixo nível, *Assembly*, para que, somente então, a VM possa lê-lo e interpretá-lo. A linguagem *Java*, que também é interpretada e funciona utilizando uma máquina virtual para executar suas instruções *Assembly*, tem sua VM descrita em (LINDHOLM et al., 2018).

A linguagem *Python* é uma linguagem amplamente usada no mundo todo. Segundo lista divulgada na revista *Business Insider*, que mostra as linguagens de programação mais utilizadas atualmente (SANDLER, 2018), a linguagem *Java* aparece em quinto lugar e a linguagem *Python* um pouco atrás, em sétimo. Nesta mesma lista, a linguagem *Javascript*, utilizada principalmente para a confecção de *websites*, aparece em primeiro lugar. Já ocupando o último lugar da lista, na posição 14ª, aparece a linguagem *Swift*, desenvolvida pela empresa *Apple* e utilizada para desenvolvimentos dentro dos seus sistemas operacionais.

Embora existam grandes vantagens para se desenvolver um sistema baseado em máquina virtual, em certas ocasiões pode ser custoso. A maior vantagem de se utilizar uma linguagem deste tipo é que um mesmo algoritmo já compilado pode rodar em todos os dispositivos que a linguagem provê suporte. Quando se utiliza uma linguagem não interpretada, em cada processador que um algoritmo já compilado precisar ser rodado, é

necessário fazer uma nova compilação, para que o código *Assembly* seja o mesmo utilizado para o processador. Enquanto isso, com uma linguagem interpretada, o código *Assembly* que será executado diretamente no processador será o da máquina virtual, podendo, portanto, executar o mesmo algoritmo compilado *Python* em qualquer processador.

Todavia, a principal desvantagem em se utilizar uma linguagem interpretada é a perda de desempenho, visto que o algoritmo em si passará para uma máquina virtual executá-lo e, em seguida, essa VM terá de ser executada em um processador físico. Com esses procedimentos, perde-se em desempenho, visto que executando um código *Assembly* diretamente processado em um processador físico alvo seria mais rápido. Contudo, em aplicações simples e rotineiras, não é possível perceber tal problema. Já em aplicações que necessitam de cálculos pesados e que precisam ser entregues em determinado período de tempo (sensíveis ao tempo), os problemas podem ser percebidos claramente, podendo, inclusive, inviabilizar a execução de determinado sistema.

2 METODOLOGIA

2.1 Códigos *Assembly*

Para iniciar a análise do código *Assembly* que a máquina virtual *Python* utiliza para executar algoritmos escritos nesta linguagem, vamos falar um pouco sobre o que compõe um código *Assembly* (HUTTY, 1981). As linguagens *Assembly* utiliza como princípio quebrar instruções de alto nível ao máximo, ou seja, quando programamos em alto nível, utilizamos funções complexas, mas que podem ser manipuladas de forma simples e executar operações mais rebuscadas. Porém, em baixo nível, essa linha de código única do alto nível precisa ser quebrada em diversas operações simples, que executam uma função extremamente específica dentro da máquina virtual ou de um processador.

Para entender melhor tal conversão, devemos lembrar que um processador somente executa suas funções baseando-se em eletricidade e em componentes físicos. Então, é necessário que as instruções sejam as mais simples possíveis para que o processador possa compreendê-las e executá-las rapidamente, a fim de garantir que a complexidade do circuito seja a menor possível. Então, cada processador dispõe de um conjunto de instruções, chamado de ISA (*Instructions Set Architecture*). Para que uma linguagem de alto nível possa ser executada em determinado processador, é necessário que todas as funcionalidades da linguagem possam ser executadas apenas utilizando o conjunto de instruções. A esse conjunto, denominamos código *Assembly*.

Como códigos *Assembly* nem sempre são simples de serem analisados, todos são classificados como sendo de baixo nível, já que, quando programamos em uma linguagem de alto nível, apenas uma linha de código pode criar dezenas de instruções *Assembly*. Todavia, devemos também lembrar que um código *Assembly* pode ser representado tanto com palavras chaves quanto por *bits*. A primeira representação é a utilizada, principalmente, para facilitar o entendimento de quem lerá o código e o escreverá. Já a segunda, que irá desempenhar exatamente o mesmo funcionamento que a primeira, é utilizada para que o algoritmo possa ser entendido pelo processador.

2.2 Apresentação do código *Assembly* da linguagem *Python*

A linguagem *Python*, mesmo tendo sido desenvolvida para rodar em uma máquina virtual, necessita ser convertida para um código *Assembly* (GALLARDO, 2016), já que é extremamente difícil para a máquina virtual ler diretamente a linguagem escrita em alto nível e executar de forma eficiente. Então, para que isso seja possível, a máquina virtual *Python*

dispõe de um conjunto de instruções (ISA) bem definido que capaz de executar qualquer algoritmo escrito em *Python*.

Para converter os códigos escritos em alto nível para um código *Assembly* e entender o funcionamento de cada instrução *Assembly*, a organização mantedora da linguagem *Python* disponibilizou, em seu website oficial, uma documentação explicando em detalhes como é feito o *disassembler* (PYTHON, 2017), que é a possibilidade de visualizar as instruções de baixo nível que compõem determinado algoritmo. Também, na mesma documentação, existem detalhes do funcionamento de cada instrução.

Neste artigo, serão descritas apenas instruções *Assembly* da linguagem *Python* que são necessárias para implementações de algoritmos simples, tais como operações aritméticas, operações de condições e repetições (*loops*). A linguagem possui diversas outras instruções que são consideradas mais complexas e que não serão abordadas nesta ocasião. O motivo para isso é que, ao implementar um processador como sendo um trabalho e conclusão de curso, torna-se inviável desenvolvê-lo em sua totalidade, visto que existem dezenas de instruções diferentes que precisariam ser cobertas pela arquitetura.

2.3 Instruções do código *Assembly* da linguagem *Python*

As instruções *Assembly* da linguagem *Python* são bem definidas e precisas, executam operações específicas e que são necessárias para o correto funcionamento da linguagem. Grande parte das instruções pode ser comparada às instruções *Assembly* de outros processadores, tais como *Neander* (WEBER, 1998) e *MIPS* (SWEETMAN, 2006), que são geralmente utilizados para fins didáticos.

É importante que se saiba também que a máquina virtual da linguagem *Python* é implementada utilizando uma pilha para todas as manipulações com variáveis. Abaixo estão listadas algumas instruções *Assembly* da linguagem *Python*, juntamente com suas respectivas funcionalidades.

i. LOAD_CONST: carrega, para o topo da pilha, uma constante apresentada como argumento na chamada da função;
ii. LOAD_FAST: carrega, para o topo da pilha, uma variável salva em um endereço de memória apresentado como argumento;
iii. STORE_FAST: salva o valor presente no topo da pilha na memória, em um endereço determinado como argumento, e o remove do topo da pilha;
iv. COMPARE_OP: compara o topo da pilha com o valor logo abaixo, utilizando como método de comparação um valor informado, que fará referência a uma operação lógica (por exemplo: menor que, igual ou maior que);
v. POP_JUMP_IF_FALSE: executará um pulo para a posição de memória passada no argumento caso a operação de comparação retorne-se falsa;
vi. BINARY_ADD: executa uma soma simples entre o topo da pilha e a posição logo abaixo, excluindo o topo da pilha e sobrescrevendo a antiga segunda posição da pilha com o resultado;
vii. BINARY_SUBTRACT: do mesmo modo que a instrução BINARY_ADD, executa uma subtração.

2.4 Análise e método de *disassembler*

Para analisar o conjunto de instruções da PVM (*Python Virtual Machine*), foi necessária, além da documentação, a escrita de algoritmos simples que executassem exatamente o que se queria analisar. Logo, foram escritos algoritmos que implementavam carregamento e escrita de dados na pilha e memória, respectivamente, comparações e funções de repetição.

Então, utilizando uma ferramenta própria da linguagem *Python*, chamada de *CPython* (CANNON, 2005), foi possível somente adicionar os códigos escritos em alto nível para que o módulo do compilador produzisse os respectivos algoritmos em *Assembly*.

3 RESULTADOS e DISCUSSÃO

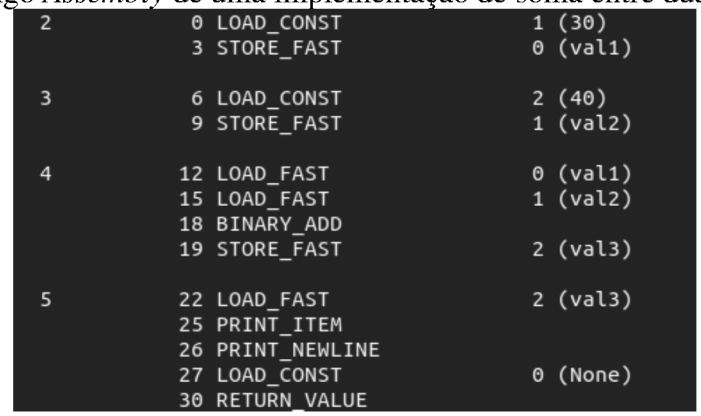
Utilizando, então, a ferramenta citada na subseção 2.4, foi possível gerar os códigos *Assembly* dos algoritmos escritos em *Python* e, também, foi possível distinguir exatamente qual sequência de instruções de baixo nível se referia a qual linha de código de alto nível. Conhecendo previamente os outros conjuntos de instruções citados na subseção 2.3, foi possível perceber que o código da PVM é bem similar em muitos pontos.

O princípio básico de programação em *Assembly* se baseia em, sempre que necessário, executar um carregamento para a pilha (ou registradores, como é o caso de outros processadores) e, somente então, executar o que é preciso executar. Ao finalizar a execução, deve-se sempre salvar o resultado obtido na memória, para que ele não seja perdido.

Como podemos ver na Figura 1, o código *Assembly* primeiramente define o valor da variável que foi criada na primeira linha de código *Python*. Para isso, é utilizada a instrução de carregamento de constantes (LOAD_CONST), passando o valor definido por parâmetro. Tal chamada de instrução colocará o valor no topo da pilha. Após isso, então, para finalizar a execução da primeira linha de código, é feito um salvamento do dado na memória, utilizando a instrução STORE_FAST, indicando que agora trata-se de uma variável. Na segunda linha de código *Python*, podemos perceber outra criação e atribuição de valor para uma variável. Logo, repete-se o mesmo procedimento.

Já na terceira linha, temos uma operação de soma com as duas variáveis salvas na memória. Então, seguindo o princípio de carregar sempre que for necessário utilizar, carregam-se as duas variáveis para as duas posições ao topo da pilha utilizando um LOAD FAST, que é a instrução destinada para carregamento de variáveis da memória. Então, com as variáveis carregadas, é chamada a instrução BINARY_ADD, que executa uma soma entre as duas variáveis ao topo da pilha e salva o resultado na segunda posição da pilha, excluindo o topo e tornando tal posição o novo topo da pilha.

Figura 1 – código *Assembly* de uma implementação de soma entre duas variáveis.



Fonte: do autor, 2018.

Para finalizar a execução da terceira linha, executa-se, então, uma instrução de salvamento na memória. Tal instrução terá como objetivo salvar o valor na posição de memória destinada ao valor da variável que recebeu o resultado da operação.

4 CONSIDERAÇÕES FINAIS

A análise de códigos *Assembly* de linguagens das quais são máquinas virtuais possam virar processador é de fundamental importância, pois os processadores deverão ser capazes de executar exatamente as instruções presentes na ISA. Portanto, o primeiro passo para a criação de qualquer processador é definir a melhor ISA possível para esse processador ou, caso já exista uma ISA, deve-se utilizá-la com completa fidelidade ao que ela prega.

O trabalho do qual veio este artigo prevê, como resultado, um protótipo de processador, a ser executado em um FPGA (*Field Programmable Gate Array*), capaz de executar qualquer algoritmo escrito em *Python* que utilize instruções que serão implementadas. Futuramente, o número de instruções implementadas pode ser ampliado, caso exista interesse.

REFERÊNCIAS

CANNON, B. **Design of the CPython Compiler**. 2005. Disponível em: <https://www.python.org/dev/peps/pep-0339/>.

GALLARDO, B. R. **Executing Code in The Python Virtual Machine**. 2016. Disponível em: <https://www.ics.uci.edu/{∼}brgallar/week9{\\ }3>.

HUTTY, R. **Z80 Assembly Language Programming for Students**. Londres: Macmillan Education UK, 1981. ISBN 978-0-333-32295-6. Disponível em: <http://link.springer.com/10.1007/978-1-349-06155-6>.

IKE-NWOSU, O. **Inside the Python Virtual Machine**. Lean Publishing, 2015. Disponível em: <https://leanpub.com/insidethepythonvirtualmachine/ read{\\#}fnref-t>.

LINDHOLM, T. et al. **The Java Virtual Machine Specification**. The Java Virtual Machine Specification, Oracle America, Inc, 2018. Disponível em: <https://docs.oracle.com/javase/specs/jvms/se10/jvms10.pdf>.

PYTHON. **31.12. dis - Disassembler for Python bytecode**. 2017. Disponível em: <https://docs.python.org/3.3/library/dis.html>.

SANDLER, R. **The 14 most popular programming languages, according to a study of 100,000 developers**. Business Insider, 2018. Disponível em: <https://www.businessinsider.com/ 14-most-popular-programming-languages-stack-overflow-developer-survey-2018-4>.

SWEETMAN, D. **See MIPS Run**. 2. ed. San Francisco: Morgan Kaufmann, 2006. ISBN 9780120884216,0120884216.

WEBER, R. F. **Fundamentos de Arquitetura de Computadores**. 2a . ed. Porto Alegre: Sagra Luzzato, 1998. 199 p. ISBN 8524106352.