Vol.2016-ARC-221 No.6
2016/8/8

PEGマシンのFPGA実装について

マイ マイクオン[1]　本多 峻[1]　倉光 君郎[1]

概要：解析表現文法 (PEG) は、2004 年に Ford によって提案された形式文法であり、正規表現や文脈自由文法の代替として人気が高まっている。本稿では、より高い性能要求を目指すため、PEG の FPGA 実装、特に PEG 演算子の命令化によるバーチャルマシン方式について報告し、性能に関する初期レポートを行う予定である。

キーワード：PEG, FPGA, 仮想マシン

FPGA Implementation of PEG Machine

Mai Maicuong[1]　Honda Shun[1]　Kuramitsu Kimio[1]

Abstract: Parsing Expression Grammar (PEG) is recognition-based foundation for describing syntax, formalized by Ford in 2004. PEG's powerful expressiveness is popular as the alternation of Regular Expression or Context-free Grammar. In this paper, we report about FPGA implementation of PEG, espeacially virtual machine method with making instruction of PEG operator to aim at higher performance, and make the initial report of the performance.

Keywords: PEG, FPGA, Virtual Machine

1. はじめに

近年、クラウドなどのデータセンターで使うコンピューティングデバイスとして性能向上や電力削減の期待から、FPGA が注目されている。例えば、Microsoft 社が Web エンジン「Bing」の処理を高速化するために、自社のデータセンター FPGA を導入すると発表した。また、中国のネット検索サービス大手の Baidu 社も画像検索サービスの実装に FPGA の導入を検討している。Intel 社でもサーバー CPU「Xeon」のパッケージに FPGA を収める製品を投入する予定である。

一方、データセンターでは、ウイルス対策などのセキュリティ対策が不可欠である。その対策の一つとして、侵入検知システム（IDS : Instruction Detection System) がある。IDS では、比較的処理の軽いホスト型 IDS が広く使われている。ホスト型 IDS は既存の不正アクセスパターンを記憶し、パターンマッチングにより、不正アクセスを検知する。それらのパターンは正規表現で記述されることが多い。

FPGA 上で正規表現を用いたマッチングマシンに関する研究は様々あり [3], [4], [7], [8]、FPGA を用いることによって処理時間を大幅に短縮することができる。しかし、パターンの複雑度に従い、パターンマッチング回路が非常に大きくなるのは問題である。

本研究では、コンパクト、かつ効率がよいマッチングマシンを実現することを目的としている。そのため、一般的に使われている正規表現の代わりに、解析表現文法（Parsing Expression Grammar）[1] を用いる。PEG は Ford によって提案され、正規表現や文脈自由文法の代替として人気が高まっている。PEG の特徴として、曖昧性がなく、字句解析が不要であり、また再帰的な構造の処理に向いている。本研究では、FPGA 上で PEG に特化したバーチャルマシンを実現する。必要最小限の回路を搭載し、また PEG 演算子に特化した専用回路によってコンパクトかつ効率がよいマッチングマシンが期待できる。

[1] 横浜国立大学

本稿の構成は次の通りである。第 2 節は PEG について述べる。次に第 3 節、第 4 節では、設計及び実装について説明する。第 5 節では性能評価を行い、最後に第 6 節に結論と今後の課題を述べる。

2. 解析表現文法

PEG は A ← e というルールの集合であり、その中に A は非終端記号、e は解析表現である。解析表現は表 1 によって表される。

表 1 PEG の演算子

解析表現	意味
'hoge'	文字リテラル
[a-zA-Z0-9]	文字クラス
.	任意の文字
A	非終端記号
(e)	グルーピング
e?	オプション
e*	0 個以上の繰り返し
e+	1 回以上の繰り返し
&e	肯定先読み
!e	否定先読み
e_1e_2	シーケンス
e_1/e_2	優先度付き選択

'abc' 及び . はそれぞれ abc と任意の文字にマッチする。[abc] の場合、abc のいずれか 1 文字にマッチする。e?, e*, e+は正規表現と同様であるが、PEG ではできるだけ長い文字列をマッチさせる。e_1e_2 は順次に e_1、e_2 を評価し、どちらかが失敗した場合、最初の位置にバックトラックする。優先付き選択 (e_1/e_2) はまず e_1 を評価し、もし失敗した場合 e_2 を評価する。また、!e では e が失敗する時に成功し、e が成功した時に失敗する。

図 1 は四則演算を表す PEG の例である。PEG の演算子は非常に単純で、再帰的構造を処理するのに優れている。また、字句解析及び構文解析を分ける必要がある他の形式文法と違って、PEG では、字句解析を同時に行うことができる。

```
Expr    ← Sum
Sum     ← Product (('+' / '-') Product )*
Product ← Value (('*' / '/') Value )*
Value   ← [0-9]+ / '(' Expr ')'
```

図 1 PEG 例

3. 設計

PEG は packrat parsing[12][2][6] により線形時間で解析することができる。しかし、packrat parsing は大きな入力に対して莫大なメモリ容量を使用する。そのため、大きな入力を受理するため、Medeiros 氏が PEG のための Virtual Parsing Machine[5] を提案した。本研究で用いるバーチャルマシンは Medeiros 氏が提案したバーチャルマシンをベースにしている。命令セットは図 2 となる。

種類	命令名	意味	PEG例
基本命令	Byte	文字リテラル	'a'
	Set	文字クラス	[1-9]
	Any	任意の文字	.
特化命令	Obyte/ Oset	オプション	'a'?
	Rbyte/ Rset	0個以上	'a'*
	Nbyte/ Nset /Nany	否定先読み	!'a'
制御用命令	Call	呼び出し	
	Alt	Fail stackにpush	
	Fail	強制的にfail信号をハイレベルにする	
	Succ	Fail stackからpop	
	Ret	呼び出し先に戻る	
	Jump	指定された命令にジャンプ	

図 2 命令セット

本研究で用いる命令セットは Medeios 氏の提案した命令セットに、PEG の演算子を実行するための特化命令を追加している。特化命令では、オプション命令（Obyte, Oset)、0 個以上の繰り返し命令 (Rbyte, Rset）及び先読み命令 (Nbyte, Nset, Nany) がある。これらの命令は、実行する命令数を削減し、また特化回路によって、実行効率を上げるためにある。

メモリ使用量を削減するため、命令のワードは 16bit に収めた。第 15 ビットから第 11 ビットまでは、オペレーションフィールド（Op フィールド）であり、各命令に対応したコードが割り振られる。第 10 ビットから第 0 ビットまでは命令の対象データとなり、命令によってこのデータの意味が異なる。対象データの意味は表 2 に示すとおりである。命令のバイトコードの生成は構文解析プラットホーム Nez[10] を使用している。

表 2 対象データの意味

命令	対象データの意味
Byte/Obyte/Rbyte/Nbyte	文字
Set/Oset/Rset/Nset	Set テーブルのインデックス
Call/Alt/Jump	命令アドレス

4. 実装

4.1 全体図

全体のシステムは図 3 に示すとおりである。ホストとの通信は Ubuntu OS と FPGA の通信が可能にした、Xillybus

社が提供している Xillybus IP コアを用いる。また、メインメモリは FPGA に搭載しているブロックメモリで実装する。システムの動作では、まずホストから命令列のバイトコードを受け取り、メモリに一時的に保存する。次に文字列をホストから受け取り、解析を行い、結果をホストに返す。

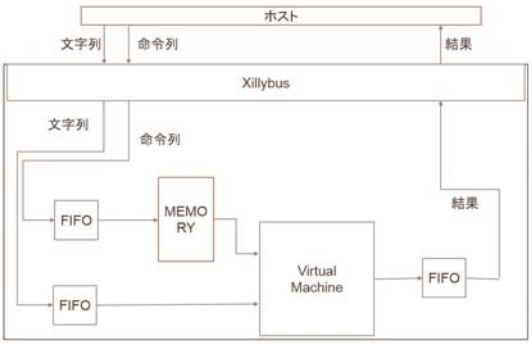


図 3 全体システム

また、PEG に特化した Virtual Machine(PEGVM) の全体図は図 4 となる。PEGVM には、書き換え機能つきプログラムレジスタ（PR) がある。PR は次に実行すべき命令が格納されたメモリアドレスを指定する。プログムの実行に従って順次にインクリメントされ、ただし、分岐命令や割り込みが実行された場合、分岐先のアドレスが PR に書き込まれる。また、Return スタックと Fail スタックがあり、それぞれのスタックがスタックポインタを持っている。スタックポインタは、インクリメンタとディクリメンタを持っており、信号によってインクリメントやディクリメントが適時実行される。他に、命令を解読するデコーダやそれぞれの命令に特化した命令用回路がある。また、メモリから読み込んだ命令データ、FIFO から受け取った文字データはそれぞれ命令レジスタ（IR)、文字レジスタ（TR) に一時的に保存される。

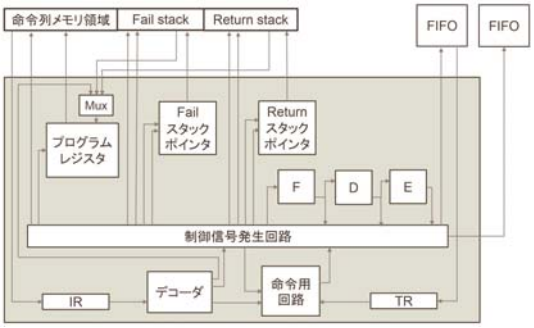


図 4 Virtual Machine の全体図

4.2 PEGVM の動作

PEGVM の動作は、命令フェッチ、文字データ読み込み、命令デコード、及び演算・データ転送を行う命令実行といった一連の処理の繰り返しであり、それらに対応した F、D、E という 3 つの状態がある。一般的に、命令フェッチは命令が格納されているメモリアドレスの設定、メモリデータレジスタへの読み込み、命令レジスタへの転送の 3 つの動作で構成される。しかし、本実装では FPGA に搭載している BlockRAM をメモリとして使用するため、命令フェッチは 1 クロックサイクルで実行した。実際に、メモリアドレスは事前に設定しておき、読み出し信号がある場合、データをメモリデータレジスタを通さず、直接命令レジスタに転送される。D 状態も 1 クロックサイクルで実行される。E 状態は基本的に 1 クロックサイクルで実行されるが、Rbyte、Rset や分岐命令の場合は例外であり、具体的には 4.3 節で説明する。

制御信号生成回路の役割は、制御信号を適時生成して、各回路に伝えることである。状態 F、D、E に対して、3 つのフリップフロップが直列に接続されている。まず前の命令の実行が成功した場合、状態 F に対するフリップフロップにフェッチ起動信号のハイレベル値が取り込まれる。そのクロックの間、メモリへの読み出し信号をハイレベルにする。その次のクロックの立ち上がりで、状態 D に対するフリップフロップにハイレベル値が取り込まれ、状態 F に対するフリップフロップの出力値はローレベルになる。そのクロックの間、IR が持っている命令データがデコードされる。同様にして、その次のクロックサイクルでは、命令実行のための信号をハイレベルにして、命令を実行する。そして、次のクロックから新たな命令フェッチを実行する。

4.3 各命令の実行

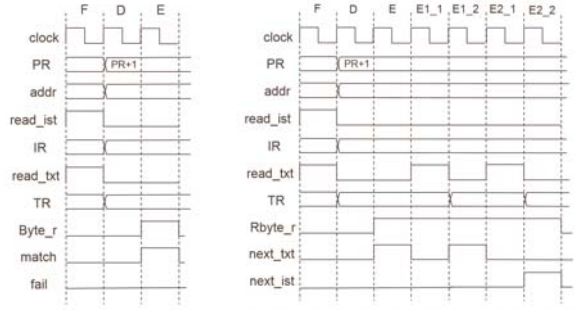
4.3.1 基本命令

Byte 命令実行のタイムチャートは図 5a に示す。F 状態では、クロックの立ち上がりで命令読み込み信号の `read_ist` 信号がハイレベルになり、命令が格納されるメモリにアクセスし、データを IR に転送される。アクセスアドレスは PR から転送されたアドレスである。同時に文字読み込み信号 `read_text` 信号もハイレベルになり、FIFO から 1 文字を読み出し、TR に転送される。

次のクロックサイクルの立ち上がりで、IR と TR のデータが確立され、このクロックサイクルで IR のデータがデコードされ、どの命令を実行するかが決まる。今回は Byte 命令用回路が実行されることになる。同クロックサイクルで、PR はインクリメントされ、メモリアクセス用レジスタの `addr` にデータが転送される。

次のクロックサイクルの立ち上がりで、Byte 命令用回路のトリガーである `Byte_r` がハイレベルになり、Byte 命令用回路が実行される。IR が持っている文字データと TR

の文字データが一致するならば、match 信号がハイレベルになり、一致しなければ、fail 信号がハイレベルになる。match 信号及び fail 信号は、制御信号生成回路の入力であり、match 信号がハイレベルであれば、次のクロックから新たな命令フェッチを実行するように制御信号が生成される。一方、fail 信号がハイレベルの場合、Fail 処理を実行する制御信号が生成される。Fail 処理では、Fail スタックからデータをポップアップし、そのデータを PR に転送し、次の命令アドレスに設定される。



(a) Byte (b) Rbyte

図 5 Byte 命令及び Rbyte 命令実行のタイムチャート

Byte 命令は、IR を持っている 1 文字のデータと TR の文字データを比較するのに対して、Set 命令は TR の文字が複数の文字の中のどれかと一致するかを評価する命令である。Set 命令を実行するために、Set テーブルを使う。Set テーブルは複数の 256 ビット列からなる。256 ビット列に、ASCII 表の n 番目の文字とマッチさせるなら、n ビット目を'1' にし、マッチさせないなら、'0' にする。Set テーブルは命令のバイトコードと同時に生成されている。このようにして、与えられた文字を Set テーブルに照らし合わせて、対応したビットの値が'1' であればマッチ成功、'0' であればマッチ失敗となる。

4.3.2 特化命令

特化命令 Rbyte の実行では、F 状態、D 状態は Byte 命令と同様である。その様子は図 5b に示すとおりである。Ex 状態では、IR が持っている命令の対象データと TR の文字データが一致した場合、next_txt 信号がハイレベルとなる。この場合、次のクロックサイクルの立ち上がりで文字読み込み信号の read_txt がハイレベルとなり、FIFO から 1 文字を読み出す。次のクロックサイクルで、IR の持っている命令の対象データと新たな TR の文字データを比較する。一致すれば、また FIFO から新たな文字を読み込まれる。IR の持っている命令の対象データと TR の文字データが一致しなくなるまで、この処理が繰り返される。一致しない場合、next_ist 信号がハイレベルとなり、次のクロックから新たな命令フェッチを実行する。

オプション命令（Obyte, Oset) も同様に実行されるが、文字消費信号を持っているところが違う。オプション命令はマッチ成功した場合、文字消費信号がハイレベルになり、文字を消費する。一方、マッチ失敗した場合、文字を消費しないが、Fail 処理が起こらず、次の命令に進む。先読み命令（NByte, NSet) の実行も Byte,Set 命令の実行と類似するが、これらの命令では文字を消費しない。

4.3.3 分岐命令

分岐命令には、Jump 命令がある。Jump 命令実行のタイムチャートは図 6 に示すとおりである。E 状態では、デコードした結果、Jump 命令の実行のトリガーである Jump_r がハイレベルになり、PR のトリガーである PR_lat 信号もハイレベルになる。このとき、インクリメント信号の PR_inc はローレベルであるため、ジャンプ先のアドレスを持っている PC_data_in の値が PR の値に置き換えられる。次のクロックサイクルで、メモリアドレスレジスタ addr に少し遅れて PR の値が取り込まれる。また、次のクロックサイクルの立ち上がりで新たな命令フェッチが始まる。

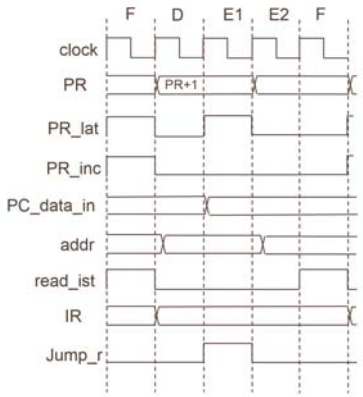


図 6 Jump 命令実行のタイムチャート

4.3.4 スタック操作命令

スタック操作命令には、Call、Alt、Return、Succ がある。

Non-Terminal を呼び出す命令が Call 命令であり、それを呼び出したプログラムへ実行制御を返すのが Return 命令である。Call 命令は、プログラムレジスタ PR の値を Return スタックにプッシュダウンして退避させ、Non-Terminal の先頭番地であるアドレスを PR に転送する。Call 命令の実行は Jump 命令と似ており、ただし新たなアドレスを PR に転送すると同時に、Return スタックにプッシュダウンを行う。

Return 命令は、Call 命令によってスタックに退避した Non-Terminal からの戻り番地をポップアップして PR へ転送する。これによって、Non-Terminal を呼び出したプログラムへ実行制御が返される。Return 命令実行のタイムチャートは図 7 に示すとおりである。

E1 状態のクロックサイクルの立ち上がりで、Return 命令実行のトリガーである Return_r 信号がハイレベルにな

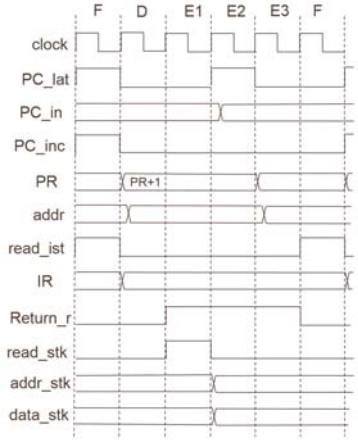


図 7 Return 命令実行のタイムチャート

る。同クロックサイクルで Return スタックの読み出し信号 `read_stk` がハイレベルになり、データが `data_stk` レジスタに転送される。次のクロックサイクルで PR の書き換えデータ `PC_data_in` に少し遅れて `data_stk` のデータが取り込まれる。次のクロックサイクルの立ち上がりで PR の新たなアドレスが確立し、少し遅れて命令アドレスの addr レジスタに転送される。この時点で Return スタックからポップアップしたアドレスは次の命令を指すようになる。次に新たな命令フェッチが始まる。

一方、PEG では、バックトラックがある [9], [11]。バックトラックは、選択肢の中でマッチが失敗した場合、前の状態に戻り、別の選択肢を評価する仕組みである。選択がある場合、選択肢を評価する前に、バクトラックが起こるときの戻り先を Fail スタックにプッシュダウンする。この操作を行うのは Alt 命令である。

また、どこかでマッチが失敗した場合、バックトラックが起こる。このとき、Fail 処理が行われ、Fail スタックからポップアップされたアドレスを PR に転送され、次に実行される命令のアドレスになる。一方、無事にマッチできた場合、他の選択肢を評価する必要がなくなるため、Fail スタックに保存したアドレスが除去される。この操作を行うのは Succ 命令である。

Alt 命令と Succ 命令の動作は Call 命令、Return 命令と似ているが、両者の違いは Alt 命令と Succ 命令がスタックを操作するだけで、PR にデータを転送しない点である。

5. 性能評価

本研究では、Xilinx 社の Zynq xc7z010-1clg400c を搭載した Zynq-7000 評価ボードで実装した。また、VHDL による RLT 記述で行い、論理合成や配置配線、シミュレーションなどには Vivado Suite Design 2015.3 を用いている。クロック周波数は 125MHz である。

ホストとのインターフェースを除いた Virtual Machine 本体の実装で用いたリソース使用量は表 3 に示すとおりである。

表 3 リソース使用量

リソース	使用量	利用可能	使用率 (%)
LUT	323	17600	1.84
FF	196	35200	0.56
ロジックセル	565	28000	2.02

現時点では、四則演算を表すなどの簡単な PEG に対して、正しく動作することが確認できた。表 3 に記載したリソースは PEG ファイル及び文字列の複雑度に依存しない。ただし、BlockRAM の使用量は PEG ファイル及び文字列の複雑度に依存する。

BlockRAM は、命令列領域、スタック領域、Set テーブルで使われている。命令列領域及び Set テーブルに用いられるメモリ量は PEG ファイルに依存する。例えば、図 1 に示す四則演算を表す PEG の場合、4 つのルールから 34 の命令が生成される。一つの命令は 16bit であるため、命令列領域に 34×16bit が使われている。また、Set テーブルに 3 列が必要であり、つまり 3×256bit が使われている。命令列領域及び Set テーブルで使われる BlockRAM は合計で 1312bit となり、搭載された 240KByte BlockRAM の 0.068%である。

一方、スタックに使われるメモリ量は文字列の長さや構造に依存するため、どの程度のメモリを確保すればよいかは今後の課題となる。

6. まとめ

本稿では、解析表現文法（PEG）に特化した Virtual Machine について述べた。必要最小限の回路だけ搭載し、また PEG 演算子に特化した回路により、コンパクトかつ効率がよいマッチングマシンが実現できた。Virtual Machine 本体が非常に小さいため、同じ FPGA に複数の Virtual Machine を載せ、並列で動作させることによって、高いスループットのマッチングマシンが期待できる。

現時点では、四則演算を表すなどの簡単な PEG ファイルに対して正しく動作できた。スタックに使われるメモリの見積、及びより複雑なデータ構造を処理できるように回路を拡張するのは今後の課題となる。

参考文献

[1] Ford, B.: Parsing expression grammars: A recognition-basedsyntactic foundation. In Proc. of POPL04, 2004.
[2] Ford, B.: Packrat Parsing and Parsing Expression Grammars Pages.http://bford.info/packrat/
[3] Sidhu, R. and Prasanna, V.K.: Fast Regular Expression Matching using FPGAs. In Proc. of FCCM01, 2001.

[4] Yang, Y.E., Jiang, W. and Prasanna, V.K.: Compact Architecture for High Throughput Regular Expression Matching on FPGA. In Proc. of ANCS08, pp. 30-39, 2008.
[5] Medeiros, S. and Ierusalimschy, R.: A Parsing Machine for PEGs. In Proc. of DLS08, 2008.
[6] Kuramitsu, K.: Packrat Parsing with Elastic Sliding Window. In IPSJ Programming Meeting, 2014.
[7] Yamagaki, N., Sidhu, R. and Kamiya, S.: High-speed regular expression matching engine using multi-character nfa. In Proc. of FPL 08, 2008.
[8] Lin, C.H., Jiang, C.P. and Chang, S.C.: Optimization of Pattern Matching Circuits for Regular Expression on FPGA. In Proc. of IEEE TVLSI06, 2006.
[9] Aho, A. V., and Ullman, J. D.: The Theory of Parsing, Translation and Compiling, Vol. I, “Parsing” . Prentice Hall, 1972.
[10] Kuramitsu, K.: Nez: practical open grammar language. In Proc. of ACM SPLASH/Onward 2016, 2016.
[11] Birman, A., and Ullman, J. D.: Parsing algorithms with backtrack. Information and Control 23, pp. 1âĂŞ34, 1973.
[12] Ford, B.: Packrat parsing: simple, powerful, lazy, linear time. In Proc.of ICFP02, pp. 36-47, 2002.