FPGAによる構文解析器の自動合成

A DISSERTATION
SUBMITTED TO THE COLLEGE OF ENGINEERING SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF YOKOHAMA NATIONAL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ENGINEERING

1264904 MAI MAI CUONG
February 2016

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Bachelor of Engineering.

(KURAMITSU Kimio) Principal Adviser

Approved for the University Committee on Graduate Studies

Abstract

本論文は FPGA(Field-Programmable Gate Array) を用いて、構文解析器の生成について述べる。

構文解析とは、定義された文法に従ってテキストの構造を解析することである。この技術はプログラミング言語処理系のみならず、現在 Web Page(HTML や XML) の読み込みや Twitter つぶやきの解析、通信パケットの不正検出など、あらゆる場面で活用されている。ビックデータ時代となり、ますます情報量が増えると予想される今、より高速な構文解析技法が求められている。

本研究では、より高速な構文解析を実現するため、FPGA（Field Programmable Gate Array）というデバイスを利用することを考える。FPGA とはプログラミング可能な大規模集積回路のことであり、ソフトウェア処理では実現できない高速処理に活用されている。通常の集積回路の場合、出荷された回路を変更することができないが、FPGA であれば、出荷された回路も使用者側で任意の論理機能を実装できる。FPGA は並列度の高い電子回路や機能の専門化によって機能向上した電子回路を容易に構成できるため、高速な構文解析器が期待できる。また、稼働中の機器のハードウェアも変更できるため、処理の対象や内容に合わせて、状況に応じた回路に書き換えることで、最小限の回路で、多様な処理を実行できる。

本研究では、定義された文法から FPGA での構文解析器を自動合成することに関する研究、合成された回路の機能評価、及び機能向上のための最適化を行う。

Acknowledgements

本論文は私が横浜国立大学大学院工学府物理情報工学専攻電気電子ネットワークコースに在籍中の研究成果をまとめたものである。本論文を執筆するにあたり、横浜国立大学工学部電子情報工学科准教授である倉光君郎先生には、指導教官として研究の方針をご指導頂きました。ここに深く感謝の意を表します。

加えて、私の学生生活を長きにわたって見守って下さった私の家族に深く感謝致します。

Contents

List of Figures

Chapter 1

はじめに

近年、ＩｏＴの発展につれ、様々な機器への組み込みやすさやより高度なパケット処理が必要になった。また、ビックデータ時代になった今は、より高速な構文解析技法が求められている。そこで、組み込みやすさと高速な処理の面から、FPGA() というデバイスが注目されている。FPGA を用いて、構文解析器を実現する研究がいくつかある。

例えば、研究１では正規表現を使って、ＦＰＧＡ上で構文解析器を作る。正規表現からＦＰＧＡ合成に必要なＨＤＬファイル、ネットリスト又はコンフィグレーションデータに変換する。しかし、パーサごとに回路を合成しなければならず、手間がかかる。また、開発者にはある程度ＦＰＧＡの知識が必要である。

また、研究２で文脈自由文法を使ってＦＰＧＡでパーサを実現する研究がある。こちらはサブ回路を作ってたくさん並ぶ。ただし、n 個のサブ回路が n+1 文字しか受理できず、性能的にはよいとは言えない。

FPGA で構文解析器を実現するのは非常に手間がかかる。例えば、ソフトウェア開発と異なり、FPGA の回路では、遅延が発生しているため、開発者は常に遅延時間をいしきしなければならない。また、FPGA 開発の際には、合理合成、配線などという段階あり、検証時間が大きい。さらに、新しいプロトコルが続々誕生されている今、

それに対応するには、構文解析器を書き換える必要がある。

そこで、本研究の目的は自動生成可能、高速な処理、そして組み込みやすいパーサを実現する。方法としては、解析表現文法からFPGA上でパーサが自動生成する。ユーザはＦＰＧＡの知識が必要なく、文法を定義するpegファイルとテキストを入力するだけで、結果が得られる。

構文を定義するために、解析表現文法（Parsing Expression Grammar、以降PEG）を用いる。PEGは形式言語を一連の規則を使って表したものであり、PEGを用いることにより、事前に字句分析する必要がなく、線形時間で構文解析可能というメリットがある。またFPGA構成を記述するために、VHDL（VHSIC Hardware Description Language）という言語を用いる。VHDLとはデジタル回路、特に集積回路を設計するための言語であり、具体的な回路を考慮せずに動作だけを記述すればハードウェアの動作を定義することができるため、ソフトウェア・プログラミングと同様な手法でハードウェアの設計を行うことができる。VHDLは現在標準化され、ハードウェア記述言語の標準的仕様として多く採用されている。FPGAで構文解析器を自動合成するために、先ずPEGファイルからVHDLファイルを自動生成し、そしてFPGA合成ツールを使ってVHDLからFPGAを合成する。

本論文の構成は次の通りである。
第１章：初めに
第２章：ＦＰＧＡ及び解析表現文法ＰＥＧ
第３章：実装方針
第４章：デザイン
第５章：実装
第６章：実験
第７章：関連研究
第８章：結論

Chapter 2

関連知識

この章では、ＦＰＧＡ及びＰＥＧについて説明する。

2.1 FPGA

FPGA とは、ユーザー側で回路を変更することができるデバイスのことである。FPGA（Field Programmable Gate Array）とは、デジタル回路を構築する大規模集積回路の一種である。通常の集積回路の場合、出荷された回路を変更することができないが、FPGA の場合、出荷された回路も使用者側で任意の論理機能を実装できる。FPGA は並列度の高い電子回路や機能の専門化によって機能向上した電子回路を容易に構成できる。ＦＰＧＡは並列処理に擦れており、並列化によって高速化することが可能である。また、ユーザ側が自由に回路を変更することが可能であるため、ある問題に特化した専用回路を作り、性能を向上することができる。

さらに、稼働中の機器のハードウェアも変更できるため、処理の対象や内容に合わせて、状況に応じた回路に書き換えることで、最小限の回路で、多様な処理を実行できる。
一般にＦＰＧＡ開発の際、ハードウェア記述言語（ＨＤＬ）が用いられる。本研究では、ＨＤＬの一種であるＶＨＤＬを使用する。

2.2 PEG

構文解析とは、定義された文法に従ってテキストの構造を解析することである。本研究では、構文解析を実現するために、形式文法を使って、ＦＰＧＡ上でパーサを自動生成する。形式文法はたくさんの種類がある。例えば、解析表現文法（ＰＥＧ）や文脈自由文法（ＣＦＧ）などがある。

本研究では、パーサを構築するために、解析表現文法を使用する。ＰＥＧはＡ ¡= ｅというルールの集合である。解析表現は表（）にある値と演算子を組み合わせた式である。ＰＥＧのメリットは書きやすく、強力な表現力をもっている。また、Ｐａｃｔｒａｃｋ　を使って線形時間で処理することができる。

Chapter 3

実装方針

本研究では、Vitual Machine を採用している。PEG ファイルから、実行すべき命令列を生成する。命令セットは図 3.1 となる。

命令では、基本命令、特化命令と制御用命令がある。

基本命令は BYTE, SET, ANY がある。こちらの命令は実際に文法定義ファイルの文字とテキストからの文字を比較し、文字を消費するかを判断する。BYTE は文字リテラルを処理するための命令である。例えば、A = 'a' のＰＥＧである場合、[BYTE 'a'] という命令が生成される。この命令は、現在評価しているテキストの文字が'a' であれば成功でそうでなければ失敗である。
SET は文字クラスを処理するための命令である。例えば、A = [1-9] の場合、[SET [1-9]] という命令が生成される。
ANY は任意の文字を処理するための命令である。例えば、A = . の場合、[ANY] という命令がある。

非終端記号は CALL 命令を使って、非終端記号の定義されたところにジャンプする。ジャンプする前に、現在命令の次の命令を Return stack に push する。RET という命令の時、Return stack からジャンプ先を取得する。例えば、以下のＰＥＧファイルはこのような命令列が生成される。

種類	命令名	操作	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/ OANY	オプション	‘a’?
	RBYTE/ RSET/ RANY	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	Fail処理を発生	
	SUCC	Fail Stackからpop	
	POS	文字の位置を保存	
	BACK	保存された位置に戻る	
	RET	呼び出し先に戻る	

Figure 3.1: 命令セット

$$\begin{aligned} Add &= Value(+/-)Value \\ Value &= [1-9] \end{aligned}$$

生成された命令列：

$$\begin{array}{lll} Add & L1 & CallL5 \\ & L2 & Set[+/-] \\ & L3 & CallL5 \end{array}$$

$$\begin{array}{rl} & L4 \quad Ret \\ Value & L5 \quad Set[1-9] \\ & L6 \quad Ret \end{array}$$

まず非終端記号があるため、その非終端記号（L5) へジャンプする。Ｌ５にジャンプする前に、Ｌ２を Return stack に push される。ここで L5 が成功したら、L6 に進み、L6 は RET であるため、Return stack からジャンプ先を pop される。先ほど L2 を Return stack に push されたため、L2 にジャンプされる。
次に L2 が成功したら、L3 に進む。L3 で Value を呼び出し、Ｌ４を Return stack に push される。Value の命令が終ったら、Ｌ４に戻る。L4 で戻る先がないため、プログラムを終了する。どこから失敗した場合、プログラムが終了される。

ＡＬＴはジャンプ先を Fail stack に push する命令である。PEG では、Option, R , 選択という演算子があり、どこかの命令が失敗しても、処理が続けることがある。どこかが失敗した時、Fail stack からジャンプ先を取得する。ＡＬＴの使い道はグルーピング、非終端記号、選択を処理する時に使う。具体的に下記に述べる。

Option を処理するため、2 つの方法がある。対象となる Option は上記の基本要素であれば、その命令と組み合わせて一つの命令として実行する。例えば、'a'?である場合、[OBYTE 'a'] という Option とＢＹＴＥを組み合わせた命令が生成される。
対象となる Option はグルーピングまたは非終端記号の場合、ＡＬＴという命令を使って飛び先を stack に push する。例えば、A = ('a' 'b')?　'c' の場合、'a' を実行する前に、ALT で'c ' を評価するための位置を stack に push する。この場合、以下の命令列が生成される。

$$\begin{array}{rl} & L1 \quad ALTL4 \\ & L2 \quad BYTE'a' \\ & L3 \quad BYTE'b' \end{array}$$

$$L4 \quad BYTE'c'$$

'a', 'b' でマッチした場合、L2,L3,L4 の順に実行される。一方、例えば'a' で失敗した場合、Fail が発生し、stack から飛び先を取得する。先 ALT で L4 を stack に push されたため、L4 が pop されて、次に実行する命令が L4 となる。Ｏｐｔｉｏｎ対象が非終端記号の場合、ALT の後にＣＡＬＬが実行される。このように、ALT によって Option を処理することができる。

０回以上の場合も処理が２つに分かれる。対象がＢＹＴＥ，ＳＥＴ，ＡＮＹの場合、RBYTE, RSET, RANY として組み合わせた命令を実行する。

対象はグルーピング又は非終端記号の場合、ＡＬＴ及びＪＵＭＰを使って処理する。まずＡＬＴでこの処理が終ったときのジャンプ先を Fail Stack に push する。次に、グルーピング、非終端記号の要素を順番に実行し、成功したら、ＪＵＭＰでグルーピング・非終端記号の最初の要素に戻る。この処理をＦａｉｌが発生するまで、繰り返される。Ｆａｉｌが発生したら、先ＡＬＴで push した Fail stack から、ジャンプ先を取得する。

例えば、以下のＰＥＧはこのような命令列が生成される。

```
Value = '1' ( '+' '2')* '+' '3'
```

$$L1 \quad BYTE'1'$$
$$L2 \quad ALTL6$$
$$L3 \quad BYTE'+'$$
$$L4 \quad BYTE'2'$$
$$L5 \quad JUMPL3$$
$$L6 \quad BYTE'+'$$

$$L7 \quad BYTE'3'$$
$$L8 \quad RE$$

まず、ＡＬＴでＬ６を Fail stack に push する。つぎに順番に L3, L4 を実行し、終わったら JUMP 命令で L3 に戻る。 L3, L4 のどこかで失敗するまで、繰り返しが続く。一方、失敗すると、Fail stack に L6 を pop されて、Ｌ６が実行される。１回以上の場合、基本命令と０回以上にみなして実行する。例えば、A = 'a'+ の場合は、[BYTE 'a'] 及び [RBYTE 'a'] のように命令が生成される。

Choise の場合、backtrack を削減するために、選択肢の最初の文字によって、最初にどの選択にするかを決める。この処理をする命令は FIRST という。複数の選択肢の最初の文字が同じである場合、各選択肢を評価する前に、この選択肢が失敗である場合、別の選択肢に移すためのジャンプ先を ALT 命令で stack に保存する。
例えば、以下のＰＥＧはこのような命令が生成される。

```
Value = ('1' '1')/('1' '2')/'3'
```

$$L1 \quad FIRST$$
$$'1' \quad -> L3$$
$$'3' \quad -> L8$$
$$default \quad -> L2$$
$$L2 \quad FAIL$$
$$L3 \quad BYTE'1'$$
$$L4 \quad ALTL7$$
$$L5 \quad BYTE'1'$$
$$L6 \quad RET$$

$$L7 \quad BYTE'2'jumpL6$$
$$L8 \quad BYTE'3'jumpL6$$

まず、最初の文字によって、jump 先が分かれている。最初の文字が'1' であれば L3 に、'3' であれば L8 に、そうでなければ L2 に移ってＦａｉ処理が発生する。Ｌ３にジャンプした場合、まず'1' を消費する。この場合、選択肢が'1' '1' 又は'1'　'2' の２つがあるため、'1' '1' の選択肢を評価する前に、この選択肢が失敗した場合のジャンプ先をＡＬＴ命令で Fail stack に push する。もし、この選択肢が成功した場合、L6 の RET でプログラムを終了させる。一方、失敗した場合、選択肢'1' '2' を評価し、つまり L7 が実行される。

否定先読みの場合、処理が２つに分かれる。対象がＢＹＴＥ，ＳＥＴ，ＡＮＹの場合、N BYTE, N SET, N ANY として組み合わせた命令を実行する。対象はグルーピング又は非終端記号の場合、まずグルーピング・非終端記号を評価する前に、失敗した時のジャンプ先を Fail stack に push する。その後、グルーピング・非終端記号の要素を順序に評価する。どこかで失敗が発生したら、ＡＬＴで保存したジャンプ先にジャンプする。この場合は次の解析表現に進むことになる。
ただし、グルーピング・非終端記号が成功した場合、つまり否定先読みが失敗した場合、もし Fail 処理を発生させるとしたら、ＡＬＴで保存されたジャンプ先に飛ぶことになるため、次の解析表現に進んでしまう。そのため、まず SUCC 命令でＡＬＴで保存したジャンプ先を消してから、Fail 処理を発生させる。
例えば、以下のＰＥＧはこのような命令列が生成される。

$$Value = !('1'+)[1-9]+$$

$$L1 \quad ALTL6$$
$$L2 \quad BYTE'1'$$
$$L3 \quad RBYTE'1'$$

$L4 \quad SUCC$

$L5 \quad FAIL$

$L6 \quad SET[1-9]$

$L7 \quad RSET[1-9]$

$L8 \quad RET$

肯定先読みの場合、対象の表現を順序に評価することになるが、文字を消費しないため、評価する前に、ＰＯＳ命令で今の文字の位置を保存しておく。評価が終ったら、ＢＡＣＫ命令で、保存された位置に戻る。例えば以下のＰＥＧはこのような命令列が生成される。

```
Value =&('1') [1-9]+
```

$L1 \quad POS$

$L2 \quad BYTE'1'$

$L3 \quad BACK$

$L4 \quad SET[1-9]$

$L5 \quad RSET[1-9]$

$L6 \quad RET$

シーケンスは連続な命令で実行される。

ＦＰＧＡに情報を渡す前に、前処理としてＰＥＧファイルから命令列を生成する。この処理は本研究室が開発しているＮｅｚライブラリを使う。ＦＰＧＡは命令列を受け取り、テキストを解析する。

Chapter 4

設計

まず、各命令に対応するのサブ回路を作る。制御部の信号によってどのサブ回路が実行されるのか決められる。

全体回路図は図 4.1 となる。

4.1 制御部

実効ステップは以下となる。
ステップ 1 :命令をロード、文字をロード
ステップ 2 ：命令をデコード、命令実行
ステップ 2 が終ったら、またステップ 1 に戻る。

基本的に命令実行に 2 つのクロックが必要である。1 つ目のクロックでは、命令及び文字を並に専用のレジスタにロードする。Instruction Address Register(IAR) に保存されたアドレスに格納された命令データを Instruction Data Register(IDR) にロードされる。同時に Text Address Register(TAR) に保存されたアドレスに格納されたテキストを Text Data Register(TDR) にロードされる。(図 4.2)

次のクロックでは、IDR のデータをデコーダに転送され、デコーダによって、実行される命令用回路が決まる。また、命令に応じて必要な情報を実行回路に送る。同クロックで命令用回路が実行される (図 4.3)。実行結果によって、次の処理が決まる。

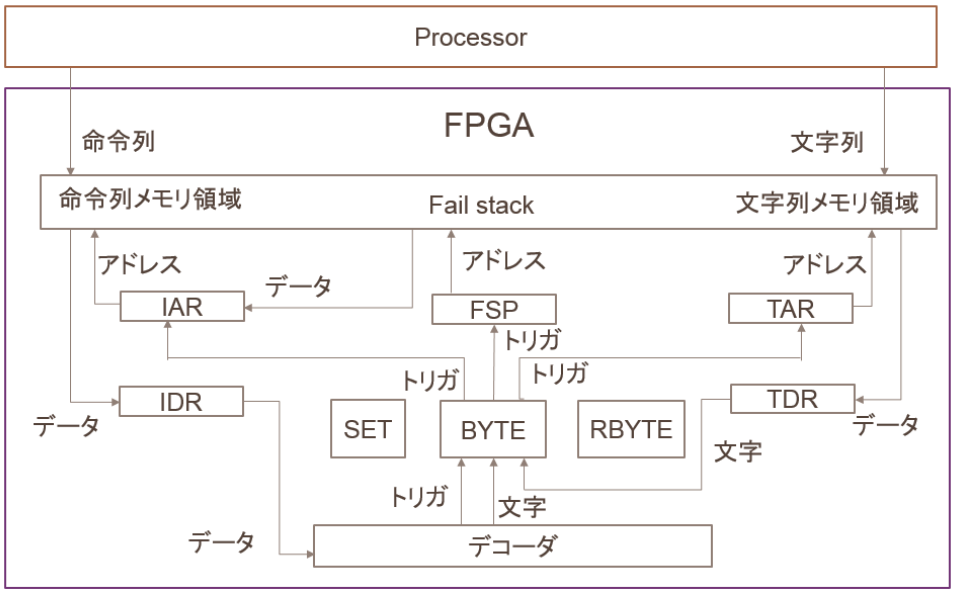

Figure 4.1: 全体回路

制御用命令を除いて命令用回路の出力はnext char, next ist, failである。next char= '1' の時、文字を消費することになる。next ist = '1' の時、次の命令に進む (図 4.4)。fail = '1' の時、Fail 処理を発生させる (図 4.5)。

Fail stackは失敗した時の戻り先を保存するためのスタックである。例えば、選択の場合、第 1 選択肢を評価する前に、もし失敗した場合、どこに戻るかを事前に保存しておく。第 1 選択肢が失敗した時、保存された戻り先にジャンプする。
どこかの回路のfail信号が'1' になったとき、Fail stackにアクセスする。Fail stackに要素があれば、Fail stackをpopし、ジャンプ先を取得する。Fail stackに要素がなければ、パース失敗となり、プログラムを終了する。

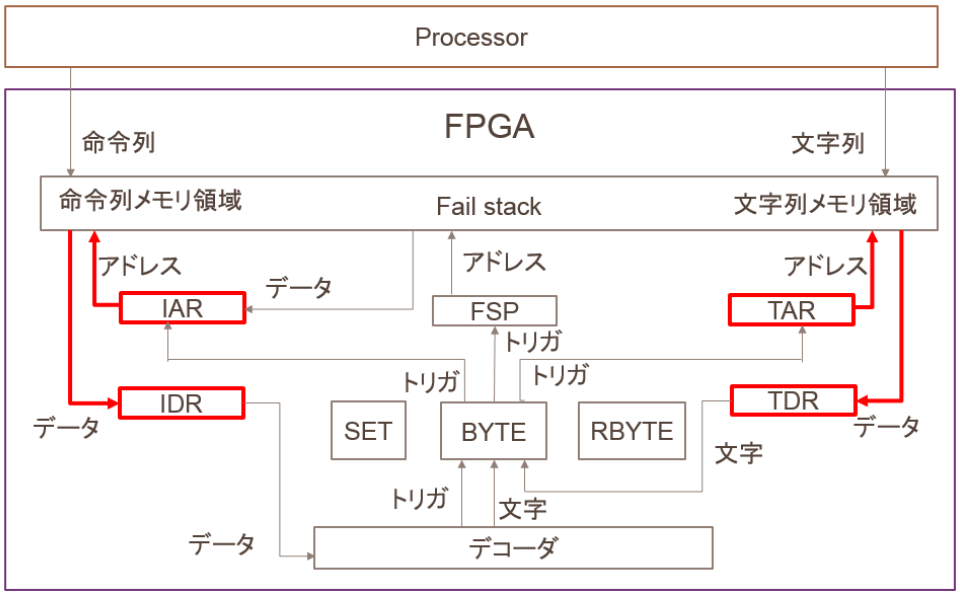

Figure 4.2: 実行ステップ 1/2

4.2 命令用回路

4.2.1 基本命令

文字操作命令は BYTE, SET, ANY がある。

BYTE 命令は、テキストの文字と形式文法の文字が一致するかどうかを判断する。ＢＹＴＥ命令用回路の入力は TRG, PEG からの文字、テキストの文字がある。一致する場合、match 信号が'1' となり、逆に一致しない場合は fail 信号が'1' となる。Trg がある場合だけ、作動する。Ｔｒｇがないときは match 及び fail が'0' となる。match が'1' になった場合、次の命令に進む処理をする。fail が'1' の場合、Alt 命令で保存されたスタックからとび先をもらう。スタックにデータが存在しない場合、プログラムを終了し、パース失敗となる。
BYTE 回路は図 4.6 となる。

ANY 命令は、１文字を消費するという命令である。ＡＮＹ命令用の回路の入力は TRG で、出力は nextchar 信号である。TRG がある場合だけ、nextchar が'1' となり、

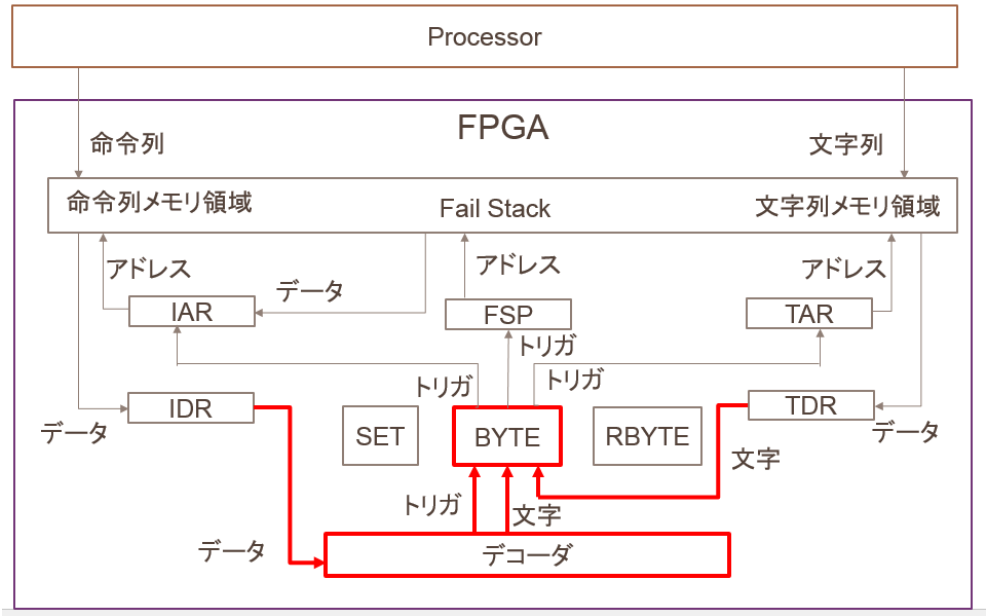

Figure 4.3: 実行ステップ 2/2

文字を消費する処理を始める。

SET 命令用回路の入力は、TRG, PEG からの 2 の文字、テキストの文字がある。文字の範囲か、文字の選択かによって、比較条件が異なる。マッチした場合、match が'1' となり、マッチしない場合、fail が'1' となる。

4.2.2 特化命令

特化命令は Option, O 個以上、否定先読みを BYTE, SET, ANY を組み合わせた命令である。

Option の場合、0 個マッチしてもよいため、TRG がある場合、match が'1' となり、この回路から fail 信号が発生されない。しかし、文字を消費するかを決めるため、1 個マッチした場合、nextchar が'1' となり、文字を消費する。

0 個以上の場合、マッチしたら、nextchar 信号が'1' となり、文字を消費する。ま

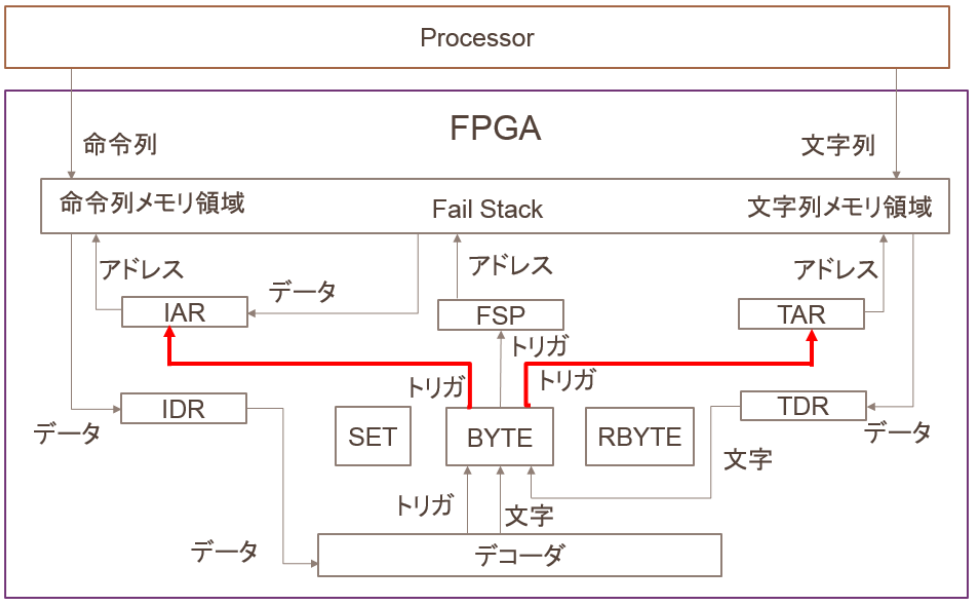

Figure 4.4: 成功した場合

た、nextchar 信号が TRG とつながっており、マッチした場合、命令をもう一度実行する。ただし、新たな文字をロードするには、１クロックが必要であるため、nextchar を１クロック遅らせてＴＲＧに入る。このように、マッチしなくなるまで、テキストを消費しながら、回路を動作する。一方、マッチされなくなったら、nextcmd 信号が'1' になり、次の命令に進む。
RBYTE 用回路は図 4.7 となる。

否定先読みの場合、条件を逆にするだけである。

4.2.3 制御用命令

Return stack は呼び出し先を保存するための stack である。cmd addr レジスタは現在の文字の位置を保存するレジスタである。CALL 命令用回路は、ＴＲＧがあるとき、まず Return stack に (cmd addr + 1) を push し、Return stack のスタックポインターをインクリメントする。同時に命令が持っているジャンプ用の命令アドレスをプログラムレジスタにロードする。

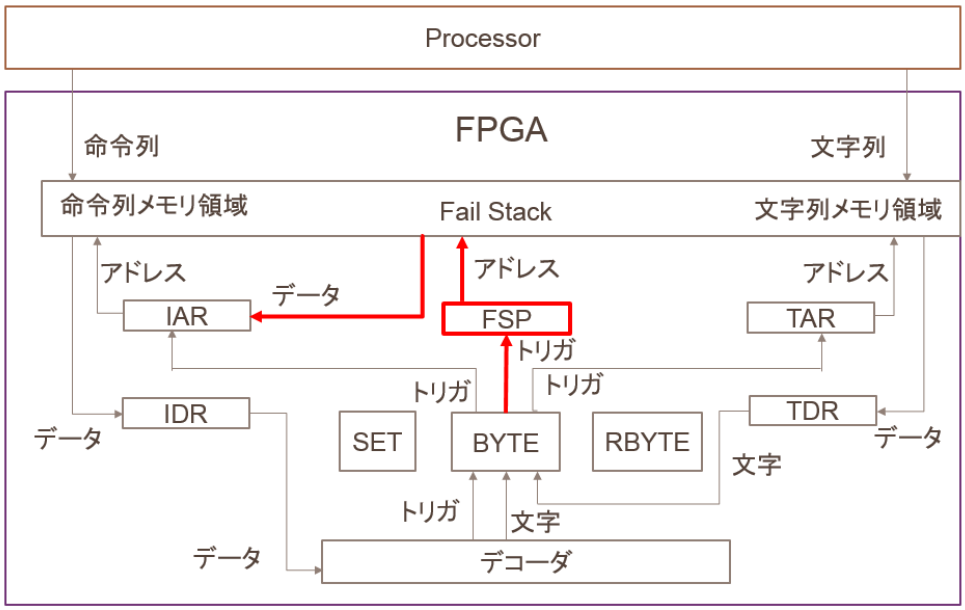

Figure 4.5: Fail 処理

RET 命令用回路は、TRG があるとき、Return stack から pop し、cmd addr に転送する。Return stack に要素がなければ、パース成功となり、プログラムを終了させる。
Fail stack は fail 処理が発生するときの命令アドレスを保存するスタックである。ＡＬＴ命令用回路は、ＴＲＧがあるとき、Fail stack に命令が持っている保存用アドレスを push し、Fail stack のスタックポインターをインクリメントする。

SUCC 命令用回路は、ＴＲＧがあるとき、Fail stack からスタックポインターが指す要素を消し、Fail stack のスタックポイントをデクリメントする。FAIL 命令用回路は、ＴＲＧがあるとき、fail を'1' となり、Fail 処理を発生させる。

text addr レジスタは文字の位置を保存するためのレジスタである。current text addr レジスタは現在の文字の位置を指すレジスタである。POS 命令用回路は current text addr を text addr に転送し、現在文字の位置を保存す。BACK 命令用回路は text addr を current text addr に転送し、現在文字を text addr に切り替える。

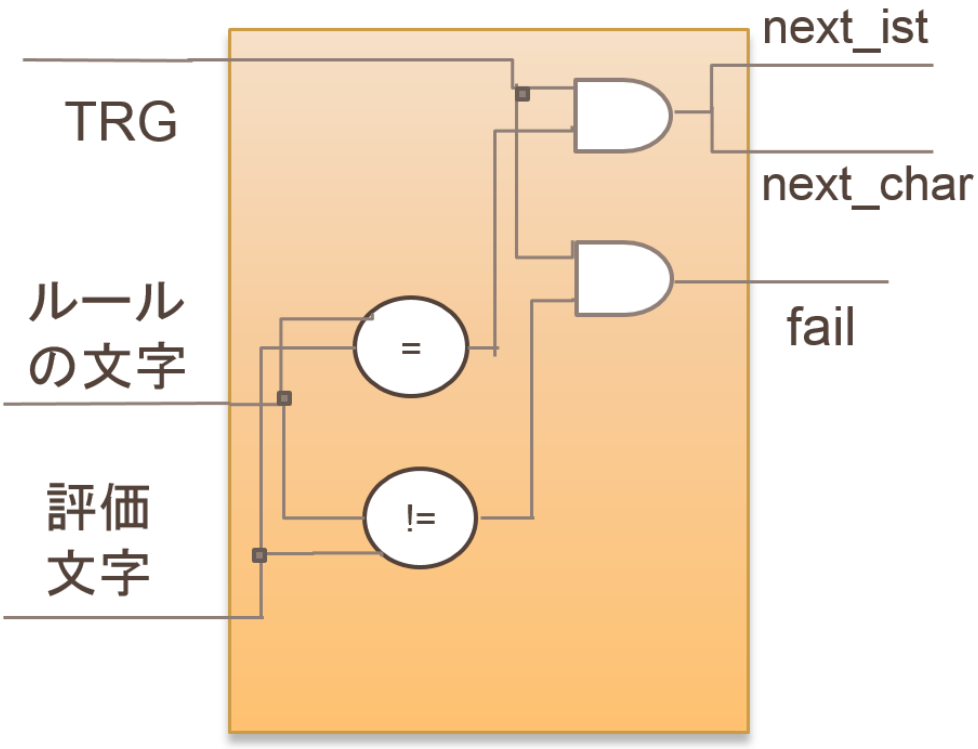

Figure 4.6: BYTE 用回路

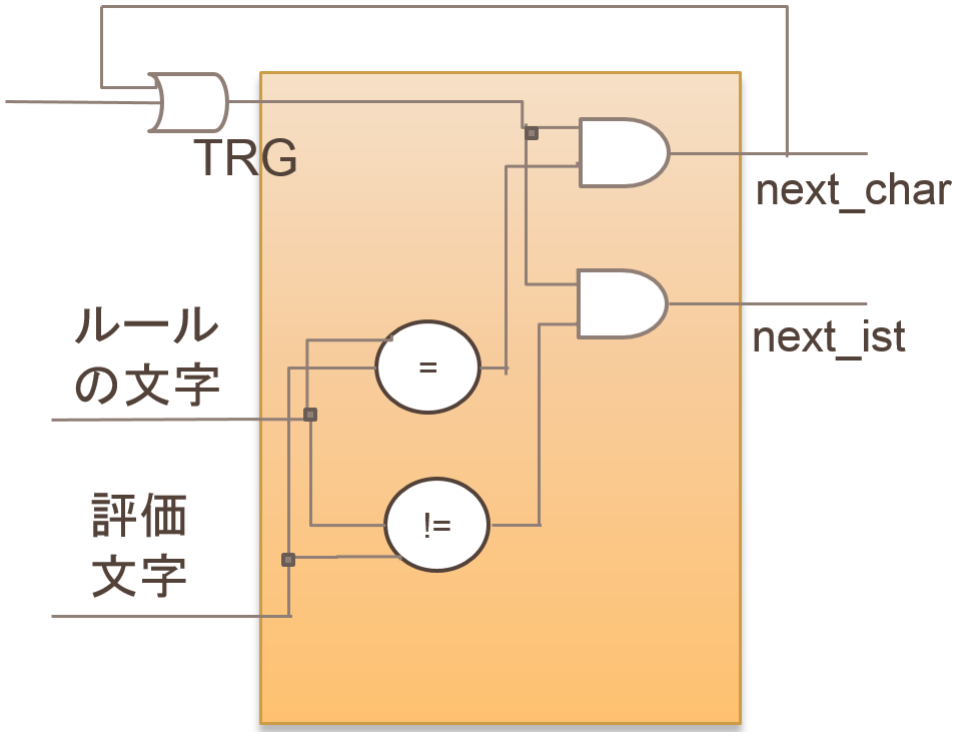

Figure 4.7: RBYTE 用回路

Chapter 5

実装

本研究は、Zynq-7000 Development Board　で実装を行う。詳細環境は以下である。

- Xilinx Zynq Z-7010 搭載
- 最大動作周波数：450MHz
- 512MB DDR3

このボードは、SoC の一種で、つまり FPGA の他に、ARM Processor も搭載している。ARM　Processer 側に Linux 環境を起動させ、通信を行った。ARM-FPGA 間では、Xillybus 社の xillybus というインタフェースを使った。

FPGA 設計するために、VHDL を用いた。開発の際、Xilinx 社の Vivado Suite Design というソフトウェアを使った。

Chapter 6

実験

性能評価のために、同じ仮想マシンをソフトウェアで実装したプログラムと実行時間を比較する。ソフトウェアの詳細は以下である。

- CPU: Intel(R) Core i7-4770 3.40GHz
- RAM: 8.00GB
- OS: Ubuntu 14.04.1 LTS

実験の際に発見したバグ：大きいデータファイルを受理できない。
原因 : 現在の実装では、プロセッサから受信されたデータストリームは一旦リングバッファ(レジスタアレイ）に一時的に保存する。大きいデータファイルの場合、バッファにまだ処理されていない文字が新しい文字に上書きされてしまう
限界： FPGA ではレジスタ数が限られているため、現在の実装は正常に受理できるのは 2KB 以下のデータファイル
今後の対策：一度に一定の長さのデータ列を受信し、処理が終わるまで受信せず、送信側に待機を要求する

結果対象は 4 つの json ファイルである。サイズは 2KB である。
データの収集方法: FPGA の実行時間は常に一定しているが、moz を実行する際、PC 環境に影響されているため、各 json ファイルに 100 回実行し、その中で処理時間が最も小さい値を取得する。

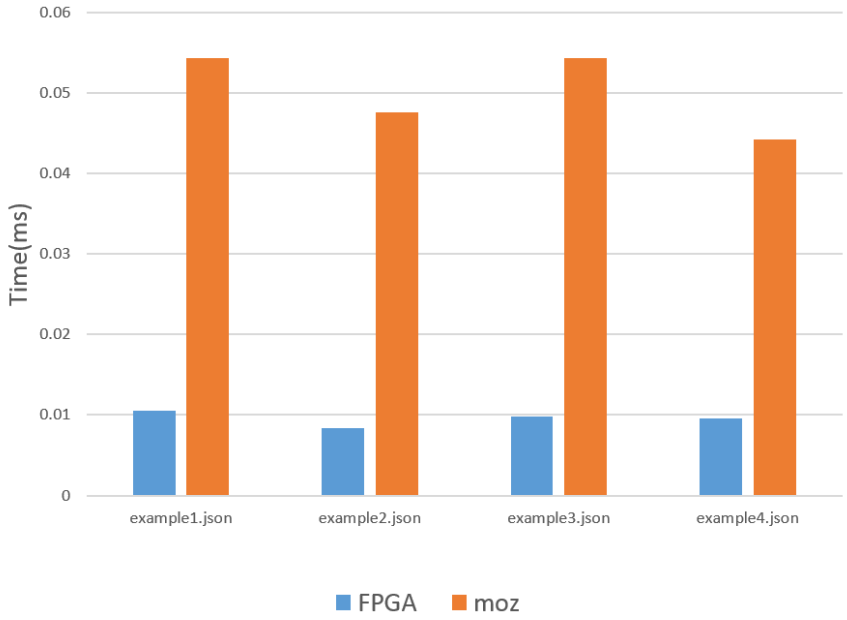

Figure 6.1: 実験結果

環境の動作周波数が異なるため、環境動作周波数を 1GHz と想定して、計算する。結果は図 6.1 となる。

Chapter 7

関連研究

[1] Cristian Ciressan, Eduardo Sanchez, Martin Rajman, Jean-Cedric Chappelier, “ An FPGA-Based Coprocessor for the Parsing of Context-Free Grammars ”, Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines 2000, Page 236

[2]Yi-Hua E.Yang, Weirong Jiang and Vicktor K.Prasanna,Proceedings of the 4th ACM IEEE Symposium on Architectures for Networking and Communications Systems Pages 30-39 , ” Compact Architecture for High Throughput Regular Expression Matching on FPGA ” ,2008

[3]Reetinder Sidhu and Vicktor K.Prasanna, Fast Regular Expression Matching using FPGAs, Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Pages 227238, , 2001

[4] Joao Bispo, Ioannis Sourdis, Joao M.P.Cardoso, and Stamatis Vassiliadis, “Regular Expression Matching for Feconfi

gurable Packet Inspection”, FPT ’06: Proceedings of the IEEE International Conference on Field Programmable Technology, 2006., Dec. 2006, pp. 119-126.

[5] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang,

"Optimization of regular expression pattern matching circuits on FPGA", DATE '06: Proceedings of the conference on Design, automation and test in Europe (3001 Leuven, Belgium, Belgium), European Design and Automation Association, 2006, pp. 12-17.

[6]Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya, "High-Speed Regular Expression Matching Engine Using Multi-Character NFA", FPL '08: Proceedings of the International Conference on Field Programmable Logic and Applications, 2008., Aug. 2008, pp. 697-701.

[7] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia", ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (New York, NY, USA), ACM, 2007,pp. 155-164.

Chapter 8

結論

本研究は仮想マシン方式を使って、FPGA で構文解析器を実現した。結果として、2KB 以下のファイルの場合、C 言語で実装したプログラムと比べて、FPGA では約 5 倍高速化できた。

今後の課題としては、まずバグを修正し、大きいファイルの場合の性能評価を行う。また、パイプライン・メモ化を導入し、より高速化を図る。さらに、仮想マシン以外の方式を検討する。