AccuCellとAccuCoreにおける、ラッチおよびフリップ・フロップのユーザ定義による論理認識

はじめに

AccuCell と AccuCore には、ラッチやフリップ・フロップの論理ファンクションを認識する自動アルゴリズムが採用されていますが、ユーザ定義による論理認識も可能です。本稿では、その方法とシンタックスの例を挙げながら説明します。

ユーザ定義による論理認識でのファンクション抽出

セルの論理を自動認識できない場合、ベクトル・テーブル・ファイル (.tbl) を使用する前に、まずファンクション・イクエージョン・ファイル (.eqn) を試してください。イクエージョンを記述する際は、次の構成ルールを適用する必要があります。

1) すべての出力は少なくとも 1 つのイクエージョンを持つ必要がある。
2) 演算の順番を指定するために、論理演算子を括弧 ( ) で囲んで使用することができる。
3) 論理積「&」は、論理和「|」よりも優先される。
4) 論理否定「~」は、論理積「&」よりも優先される。
5) 状態が 0 と 1 の場合、イクエージョンは入力と保存状態によって表現される。
6) ユーザ定義による論理認識で書き込まれたイクエージョンは、自動生成されたイクエージョンより優先される。

EBNF で記述されたイクエージョン・ファイル (.eqn) のシンタックス

```
<legal_char> ::= {{a…z} | {A…Z} | {0…9} |“ _” |“ .”}
// under-score period

<illegal_char> ::= ~ <legal_char> // NOT(<legal_
char>)
<escaped_char> ::= “\” <illegal_char>
<char> ::= <legal_char> | <illegal_char>
<string> ::= <char> [<char>]
<legal_signal_name_char> ::= <legal_char> |
<escaped_char>
<legal_signal_name> ::= <legal_signal_name_char>
[<legal_signal_name_char>]
<semi> ::= “;” // semi-colon
<assignment> ::= “:=” // colon equal-sign
<not_back_slash> ::= ~ “\” // NOT(back-slash)
<new_line> ::= <not_back_slash> \n // new_line_char
<line_continue> ::= “\” \n // back-slash must be
the last char in the line
<white_space> ::= {“ ” | \t} [<white_space>] //
tab-char
<separator> ::= <white_space> [<line_continue>]
<comment_to_end_of_line> ::= “/” “/” <string> <new_
line> // double fwd-slash
<term> ::= <semi> <new_line> [<new_line>]
<hyphen> ::= “-”
<not> ::= “~”
<and> ::= “&”
<or> ::= “|”
<oper> ::= <and> | <or>
<rising_edge_suffix> ::= “+”
<falling_edge_suffix> ::= <hyphen>
<previous_state_prefix> ::= <hyphen>
<input_signal> ::= <legal_signal_name> // defined in
.cfg INPUTS cmd
<bidir_signal> ::= <legal_signal_name> // defined in
.cfg INOUTS cmd
<clock_signal> ::= <legal_signal_name> // defined in
.cfg CLOCKS cmd
<output_signal> ::= <legal_signal_name> // defined
in .cfg OUTPUTS cmd
<input> ::= {[<previous_state_prefix>]
<input_signal>} |\
{[<previous_state_prefix>] <bidir_signal>} |\
<clock_signal> {<rising_edge_suffix> |
<falling_edge_suffix>}}
```

```
<output> ::= <output_signal> | <bidir_signal>
<not_input> ::= <not> <input>
<grouping> ::= "(" [<separator>] <expr>
[<separator>] ")"
<logical_exp> ::= {<grouping> | <not_input> |
<input>} [<oper>] [<grouping> |\
<not_input> | <input>] // bidir_signal may
not self-refer
<output.0_expr> ::= <output>".0" [<separator>]
<assignment> [<separator>]\
<logical_expr> <term>
<output.1_expr>::=<output>".1"[<separator>]
<assignment> [<separator>]\
<logical_exp> <term>
<output.z_expr> ::= <output>".z" [<separator>]
<assignment> [<separator>]\
<logical_expr> <term>
<output.x_expr> ::= <output>".x" [<separator>]
<assignment> [<separator>]\
<logical_expr> <term>
<eqn_file> ::= [<comment_to_end_of_line>] |
<separator> | <new_line>]\
<output.0_expr> [<comment_to_end_of_line> |
<separator> | <new_line>]\
<output.1_expr> [<comment_to_end_of_line> |
<separator> | <new_line>]\
[<output.z_expr>] [<comment_to_end_of_line> |
<separator> | <new_line>]\
[<output.x_expr>] [<comment_to_end_of_line> |
<separator> | <new_line>] <eof>
```

例

次に記述する例は記述例の紹介だけを目的としたもので、ユーザが定義する必要のあるファンクションの例 ではありません。

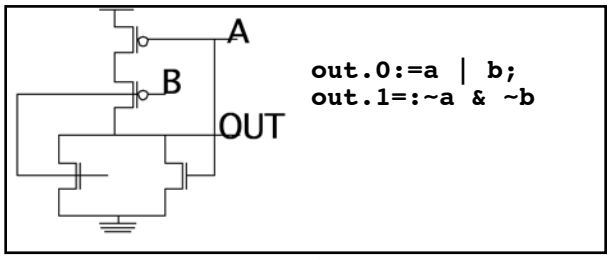


図 1:　2 入力 NOR とイクエージョン

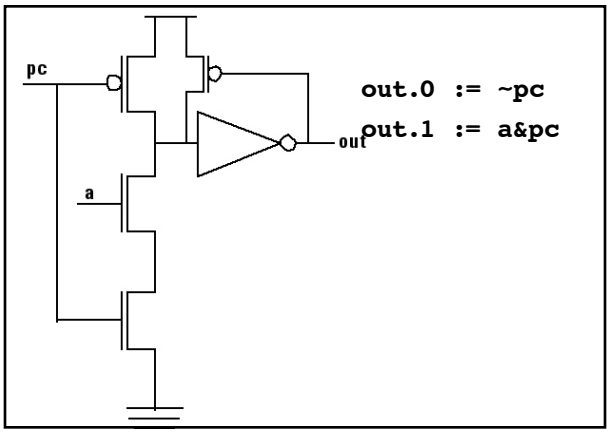


図 2:　標準的フット付ドミノ・バッファとイクエージョン

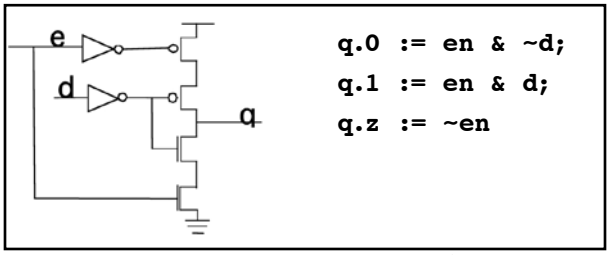
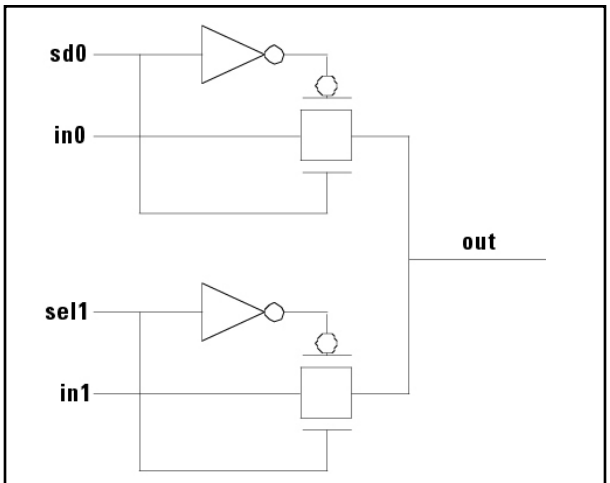


図 3:　トライステート・バッファとイクエージョン




```
out.0 := (~sel1&sel0&~in0) |
(sel1&~in1&~sel0) | (~in1&sel0&~in0)
; // out is 0
out.1 := (in1&sel0&in0) |
(~sel1&sel0&in0) | (sel1&in1&~sel0) ;
// out is 1
out.z := (~sel1&~sel0) ; //out is
floating
out.x := (sel1&in1&sel0&~in0) |
(sel1&~in1&sel0&in0) ; // out is un-
known
```

図 4:　マルチプレクサとイクエージョン

順序デバイス

順序セルには、それぞれのセル・コンフィグ・ファイルに CLOCKS コマンドが含まれている必要があります。そして、セルはドメインによってグループ分けされ、位相によって、さらにひとつのドメイン内で識別されます。

非同期セット信号と非同期クリア信号を持つフリップ・フロップもまた、イクエージョンを使用して表現することができます。アクティブ・ハイ信号で「セット (set)」、アクティブ・ハイ信号で「クリア (clr)」であるフリップ・フロップで考えて見ましょう。「セット」をアクティブにすると、このフリップ・フロップは q 出力に 1 を出力しますが、「クリア」をアクティブにすると、0 を出力します。

下図の例では、「d」および「m」(main)、「!m」(main) は、セット信号とクリア信号とは無関係な状態にあります。

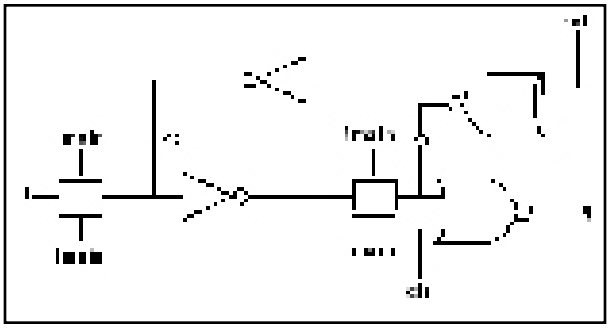


図5:　非同期「セット(set)」と非同期「クリア(clr)」を持つフリップ・フロップとそのイクエージョン

```
q.0 := ~d & main+ & ~set & ~clr | clr;
q.1 := d & main+ & ~set & ~clr | set & ~clr;
```

.cfg ファイルにおける前処理

AccuCell では、.eqn または .tbl ファイルを使用する場合、下記のようにセル・レベル .cfg 内にファイルを指定する必要があります。

1) EQN_FILE_NAME myfile.eqn または、

2) TBL_FILE_NAME myfile.tbl

AccuCore では、ブロック・レベル .cfg 内に、次のいずれかのコマンドを指定します。

1) KEEP_SUBCKT （階層型ネットリストの場合）
2) FIND_SUBCKT （フラット型のネットリストの場合）
   これにより、まず回路の一部が分離され、その論理がコマンド中で指定された .eqn ファイルまたは .tbl ファイルで上書きされます。

```
KEEP _ SUBCKT <subckt _ name> {<inputs>} {<out-
puts>} {bidirs>} {<clocks>} {EQN <eqn _
file>|<tbl _ file>}
FIND _ SUBCKT <subckt _ name><pattern _ file>
{<powers>}{<grounds>} {<inputs>} {<out-
puts>} {bidirs>} {<clocks>} {EQN <eqn _
file>|<tbl _ file>}
```

注記：マッチングのためのパターン･ファイルとしては、フラットなネットリストが必要となります。

まとめ

.tbl ファイルは、あるファンクション抽出におけるキャラクタライゼーション・プロセスを完全にコントロールする上で必要になりますが、.eqn ファイルは、ファンクション抽出向けにより簡単でかつ適切なソリューションとして提供されています。これらのファイルは、AccuCell、AccuCore の自動ベクタ生成機能を手助けしています。