設 計 自 動 化 83－4
（1997. 2. 14）

FPGAを用いた論理エミュレータにおけるRTレベルのHDL分割手法

空岡 誠実† 田中 康一郎‡ 久我 守弘‡

† 九州工業大学大学院情報工学研究科
‡ 九州工業大学マイクロ化総合技術センター
〒820 福岡県飯塚市大字川津 680-4
E-mail {sora,tanaka,kuga}@cms.kyutech.ac.jp

あらまし 長期化する機能検証期間の短縮を図るために，機能検証方法は機能/タイミングの検証を行える多機能な論理シミュレータを用いる方法だけでなく，高速に機能検証を行える論理エミュレータも併せて利用せざるを得ない状況に成りつつある．しかしながら，論理エミュレーションを開始するまでに，“HDLコードの論理合成”，“回路分割”，および“デバイスへの実装”などの計算機負荷の重い処理を行う必要があり，高速な検証速度を充分に利用できていないのが現状である．本稿では，回路分割を論理合成後のネットリストを対象する従来の方法に代わり，HDLコードを対象とすることで論理エミュレータへの実装時間を短縮する方法を提案する．評価結果から，HDLコードを対象に回路分割を行うことで，実装時間の短縮が可能であることを確認した．

キーワード 回路分割，ハードウェア記述言語，論理エミュレータ，論理合成，FPGA

Partitioning Techniques on Hardware Description Language for FPGA-based Logic Emulator

Makoto Soraoka† Koichiro Tanaka‡ Morihiro Kuga‡

† Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology
‡ Center for Microelectronic Systems, Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka 820 Japan
E-mail {sora,tanaka,kuga}@cms.kyutech.ac.jp

abstract FPGA-based Logic Emulator attracts attention to speed up functional verification. Logic emulator processes functional verification about 100 to 1,000 times faster than logic simulator. However, it is problem that we don't use logic emulator efficiently on account of heavy processes; “Logic Synthesis”, “Logic Partitioning”, and “Placing and Routing”. This paper proposes partitioning techniques to reduce a processing time of functional verification. We suggest partitioning techniques for logic emulator on not gate-level netlist but RTL HDL code. As a results of evaluation, it is confirmed that our partitioning techniques reduce the processing time for functional verification.

keyword Logic Partitioning, Hardware Description Language, Logic Emulator, Logic Synthesis, FPGA

1 はじめに

近年の集積化技術の向上に伴いLSIに集積可能な回路規模が増大してきたことで，設計入力および機能検証に要する期間が共に長期化してきている．一方では，設計の短期化および効率化のため，設計手法は回路図エディタからハードウェア記述言語 (HDL：Hardware Description Language) を用いる方法に移りつつある．また，多量のテストパターンを実行し長時間を必要とする機能検証については，従来の論理シミュレータを用いる方法だけでなく，高速に機能検証を行えるFPGAを用いた論理エミュレータも併せて利用せざるを得ない状況に成りつつある．

機能検証における論理エミュレータの実行速度は，ソフトウェアによる論理シミュレータと比べ数百～数千倍程度高速である．しかしながら，論理エミュレータを使用するためには，図1に示すように設計入力終了後，“HDLコードに対するFPGA用の論理合成”，“複数のFPGAに対する回路分割”，および，“FPGAへの配置配線”といった3つの処理を必要とする．この処理は，論理エミュレータへ機能検証する回路を実装するために必要な処理であるとともに，設計の修正や変更のたびに行う必要がある．したがって，論理エミュレータへの回路実装処理時間を短縮できれば論理エミュレータの高速な検証能力をさらに活用することが可能となる．

論理エミュレータへ回路を分割実装するまでの処理としては，現在ゲートレベルのネットリストを対象としている [1]．したがって，HDLを用いて設計を行い論理エミュレータで機能検証するためには，HDLコード全体を論理合成した後にネットリストを分割しなければならない．つまり，記述の修正や変更を行い再び検証するためには，変更したモジュール全体の再論理合成を行う必要があり，実装までの処理時間が長期化してしまうことが問題となる．

そこで，本稿ではゲートレベルのネットリストを回路分割対象とするのではなく，HDLコードから回路分割を行うことにより，再論理合成の際に生じる不必要な処理を削減し，論理合成時間を短縮する手法 [2] を提案する．以下，2章では回路分割手法について従来手法と提案手法との差異を述べ提案手法の利点を明確にするとともに，HDL分割手法の概要について述べる．3章では，HDL分割手法における処理のひとつである細分化手法の詳細について述べる．4章では，提案手法にしたがって作成した細分化プログラムを用い，細分化手法の評価および考察を行う．最後に，5章を本稿のまとめとする．

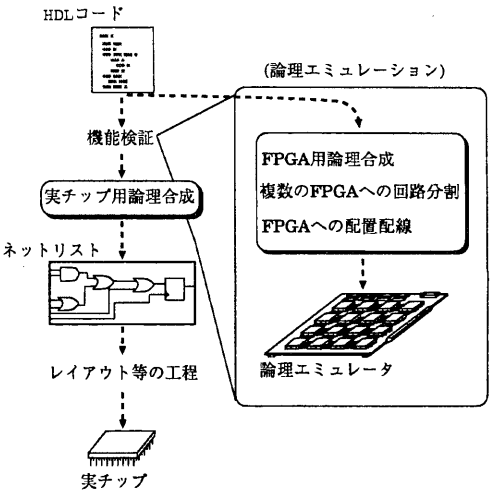

図 1: 論理エミュレータへの実装処理過程

2 HDL分割手法

2.1 提案分割手法の特長

HDLコードから論理エミュレータへ実装するまでの流れを図2に示す．論理エミュレータを対象とした現在の自動回路分割ツールでは，図2左側のフローで示すように，論理合成後のゲートレベルのネットリストを対象としている．一方，我々の提案する回路分割手法では，図2右側のフローで示すように，まずHDLにより記述された1つのモジュール (あるまとまった処理を行う機能ユニットの単位) を全く同じ動作をする複数のモジュールへ分割する．このとき分割後のモジュールはHDLコードとして表現する．この第1の処理を**細分化**と呼ぶ．細分化後のHDLコード各々に対して論理合成を行った後，細分化後のモジュールを論理エミュレータ上の各FPGAに対して実装処理を行う．この第2の処理を**グループ化**と呼ぶ．

回路分割処理を細分化とグループ化の2ステップにすることで，以下の3点の理由によりHDLコードから論理エミュレータへの実装までの時間の短縮を図ることが可能となる．

- HDLコードの記述を変更した後再び論理エミュレータへ実装する際には，細分化後のHDLコードが前回の細分化後のHDLコードから変化しているモジュールのみ細分化すれば良いため，変更されていない回路に対しては論理合成を行う必要がない．
- あるFPGAに対して実装されている細分化後のモジュール全てに変更がない場合には，各モ

ジュールの論理合成だけでなく，そのFPGAへの配置配線の処理も不要となる．

- 複数のワークステーション上で同時に論理合成ツールを実行できる環境が用意できるならば，細分化後のHDLコードを複数のワークステーション上で並列に処理することが可能となる．

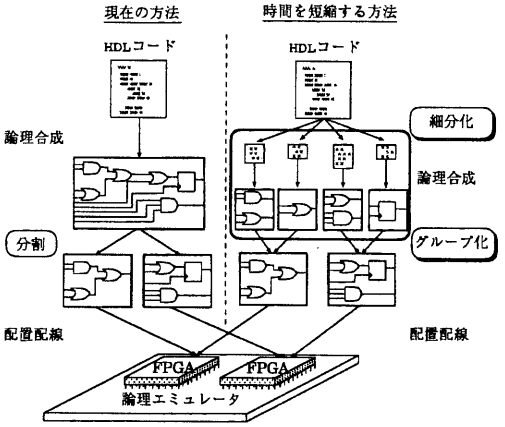

図 2: 実装までの流れ

2.2 提案アルゴリズムの概要

2.2.1 対象とするHDLの記述レベル

実設計において広く使用されているHDLにはVerilog HDLとVHDLがあるが，現在はVerilog HDLを対象とした．

LSI設計において，実際に用いるHDLの記述レベルには，ビヘイビアレベル，レジスタトランスファレベル (RTL:Register Transfer Level)，ゲートレベルの3つがある．本稿におけるビヘイビアレベルのHDLコードとは，回路の動作に必要なクロック数を明確に決定していない記述レベルを指す．したがって，ビヘイビアレベルでは伝搬遅延とクロックサイクル数のトレードオフからクロックサイクルを決定するスケジューリング機能を備えた動作合成ツール [3] の使用を前提とする．本分割手法で扱う記述レベルは，回路のクロックレベルの動作が明確に記述されているRTLを対象とし，ビヘイビアレベルのHDLコードは動作合成しなければ回路の動作を決定できないため対象外とする．なお，ゲートレベルの記述については混在させていても構わない．また，RTLのHDLコードであっても，論理合成ツールがサポートしている記述と，サポートしていない記述が存在する．本研究では，HDLコードを論理合成することを前提としているため論理合成可能なHDLコードを対象とする．論理合成ツールによりサポートされている記述が異なるが，現在筆者らが使用している米国Synopsys社製Design Compilerがサポートしている記述レベル [4] を対象とした．

2.2.2 細分化

HDLコードはある任意の論理回路を表現している．その論理回路がどのような要素から構成されているか考えてみると，それは図3に示すように以下の4つの回路構成要素から成ると考えることができる．

- フリップフロップとそのフリップフロップへの入力を選択するセレクタとなる部分．
- 設計者が階層を意識して記述したALUなどの機能ブロックの部分．
- ある機能ブロックへの入力を選択するために必要なセレクタとなる部分．
- セレクタへの制御信号を生成する組合せ回路部分．

分割されていないHDLコードを論理合成する場合は，論理合成ツールがこれらの各部分を同時に論理合成することになる．

HDLコードを上記4つに分類される回路構成要素を表現する複数のHDLコードに分割すれば，各々個別に論理合成を行うことが可能になる．細分化処理では，HDLコードをパーサを用いて構文解析を行い，分割できる回路構成要素を抽出するとともに，回路構成要素毎に論理合成可能なHDLコードを生成する．分割の際には，回路構成要素毎に入出力端子数や相互接続信号数等をレポートとして報告し，グループ化を行う際の情報として利用できるようにする．

2.2.3 グループ化

グループ化では図4に示すように，各々のFPGAに対して割り当てる細分化後のモジュールを，以下の3つの情報を基にして決定する．

- 細分化処理の際に作成される各モジュール間の接続情報．
- 細分化後のHDLコードを論理合成しテクノロジマッピングを行った際にログとして報告される各モジュールの回路規模情報．
- 実装対象である論理エミュレータの構成情報 [5]．これには，実装されているFPGAの個数・規模，FPGA間の結合形態 (結合ネット数，結合網のトポロジ) 等の情報がある．

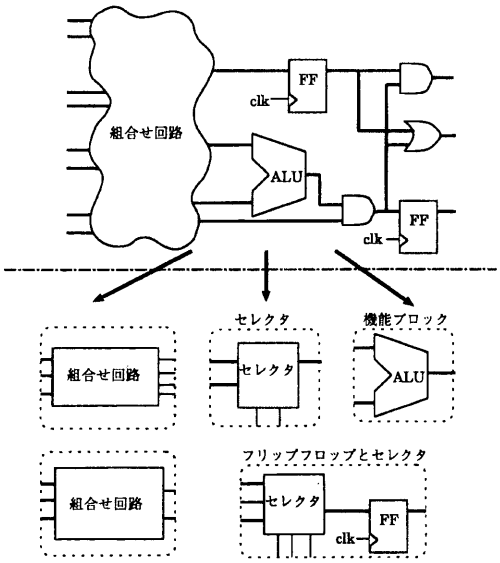

図 3: 細分化の概念

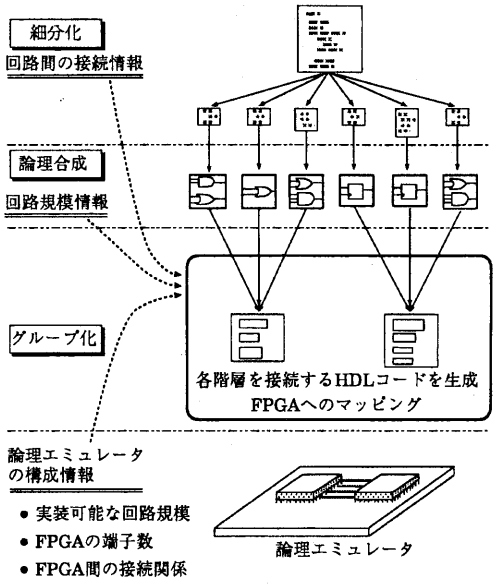

図 4: グループ化の概念

なお，グループ化のアルゴリズムについては現在研究途上であり，将来報告する研究報告を参照されたい．

3 細分化手法

3.1 細分化の基本方針

細分化を行い論理合成する場合は，細分化せずに論理合成する場合と比較して，論理合成ツールは回路全体を見渡す広範囲な最適化を行うことができなくなる．したがって，論理合成における合成時間の短縮と論理合成後の回路の質 (回路規模，動作速度等) とにはトレードオフが生じる．しかし，論理エミュレータへ実装する場合は通常その論理合成結果は機能検証にのみ使用されるため，多少回路規模が増大しても論理エミュレータへ実装までの時間を短縮した方が良いと考えられる．そこで，論理エミュレータへ実装する回路規模が多少大きくなってでも実装時間の短縮が図れる細分化手法を採用する．細分化を行う際の最適化としては，演算器のような機能ユニット単位での共有 (シェアリング) 等を行う程度にとどめる．また，必須となる回路構成要素のみを生成するよう細分化することで回路規模の増大をできるだけ抑える．

HDL では，並行動作記述と C 言語のような順次実行動作記述の両方を用いて回路を表現する．それぞれ，以下に示す方針で細分化を行う．

(1) 並行動作記述の細分化

Verilog HDL における並行動作記述には継続代入文および always 文があり，それぞれが並行に動作するとともに，これらの文に評価順序は存在しない．つまり，並列に動作する部分はそれぞれ個別のモジュールとして取扱うことができる．したがって，各々の並行動作部分をモジュールとし，個々に論理合成できるよう細分化する．

継続代入文については，それぞれその継続代入文のみを含むモジュールとして細分化する．

(2) 順次実行動作記述の細分化

順次実行動作記述を行う always 文では，always 文内の先頭から終りまでを逐次的に評価することで一連の動作を表現している．したがって，細分化する際にはその動作を実現するためにどのような回路が必要になるのか判別しなければならない．

always 文による同期回路の表現方法には，図 5(a) のように always 文の先頭でのみ同期を取る方法と，図 5(b) のように always 文内の複数箇所で同期を取る方法の 2 種類に大別できる．同期回路を always 文で表現する場合，always 文内の条件分岐などが複雑になり表現される回路が大規模になるほど，図 5(a) のように always 文の先頭でのみ同期を取る記述の方が容易に記述できる．現在は，always 文での先頭でのみ同期を取る always 文のみを細分化の対象としている．図 5の例は，順序回路であるが，組合せ回路を記述する always 文の場合も同様に，

```
always@(posedge clk)begin
    .
    .
    .
end
```

(a) 同期は先頭のみ

```
always begin
  @(posedge clk);
      .
      .
  @(posedge clk);
      .
      .
end
```

(b) 複数箇所で同期

図 5: always 文での同期表現

always 文の先頭でのみイベント待ちする always 文を対象とする.

always 文内で使用される構文要素には, 代入文としてブロッキング代入文とノンブロッキング代入文がある. 制御文として if 文, case 文 (casex 文, casez 文も含む), サブルーチンの呼出しを表現するための function_call, task_enable がある. 次節以降, 上記構文要素における細分化手法について説明する. なお, ループ文として repeat 文, for 文, while 文, forever 文, があるが, これらについては現在検討中であるため割愛する.

3.2 順次実行動作記述の細分化の詳細

(1) 順序回路を生成する always 文の取扱い

図 6に順序回路を生成する always 文の細分化事例を示す. 図 6(a) の HDL コードにおいて代入されている変数に着目し, その変数への入力を選択するセレクタ (図 6(b-2)) と, 変数に対応するフリップフロップ (図 6(b-3)) を作成するための always 文をモジュールとして生成する. 上記の処理を代入されている変数各々に対して行い, 各々の変数に対応するモジュールを生成する. そして, それらのモジュールへ制御信号を生成するモジュール (図 6(b-1)) をひとつ生成する.

フリップフロップを作成するモジュールを生成する際には, 以下の点に注意する必要がある. always 文内で代入される変数であっても, その always 文中でしか参照されず, かつ, あるクロックサイクルで参照される場合, 参照部分より上で必ず同一クロックサイクル中にブロッキング代入される場合は, その変数に対応するフリップフロップは必要としない. したがって, 代入文同士の依存関係を条件文のネストにより解析し, フリップフロップが必要でない場合はフリップフロップを作る always 文は生成しない. また, 非同期リセットがある場合には, フリップフロップを作る always 文として非同期リセットを備えたものを生成する. 上記の処理を代入されている変数全てについて実行する. また, それらの新たに生成したモジュールに対して制御

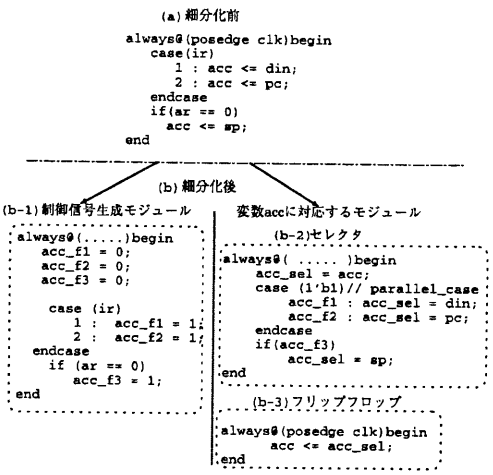

図 6: 細分化の事例

信号を送るモジュールを 1 つ生成する.

一方, セレクタを作成するモジュールを生成する際には, 以下の点に注意する必要がある. ある変数への代入が同一サイクル中に同時に起こるかどうか解析し, 同時に起こらない場合にはそれを考慮してセレクタを生成しなければ不必要な回路を付加してしまうことになる. そこで図 6(b-2) のセレクタとなる always 文中の case 文のように, 論理合成ツールに対し parallel_case を指定する. このように, 細分化によりモジュールを生成する際には case 文に対し parallel_case や full_case を指定する. この文字列は論理合成ツールによって異なるため細分化プログラムの設定ファイルに指定するかオプションとして入力することで文字列を変更する.

また, ブロッキング代入される変数を参照する場合, 以下のような処理を行う必要がある. セレクタは, 図 6(b-2) のようにあるひとつの変数値を決定する. しかし, 細分化前の always 文が図 7のように, ブロッキング代入された変数をそれより下の代入文で参照している場合は, 他の変数に対応したモジュールのセレクタに対する入力も生成する必要がある. ブロッキング代入された変数が条件文の条件内で参照されている場合も, 基本的には代入文で参照する場合と同じである. 例を図 8 に示す.

(2) 組合せ回路を生成する always 文の取扱い

組合せ回路を生成する always 文の細分化手法は, 基本的に順序回路の always 文の場合と同じである. しかし, 図 6の順序回路の例に示すようなある変数に対応するフリップフロップを生成するための always 文 (図 6(b-3)) は不要である.

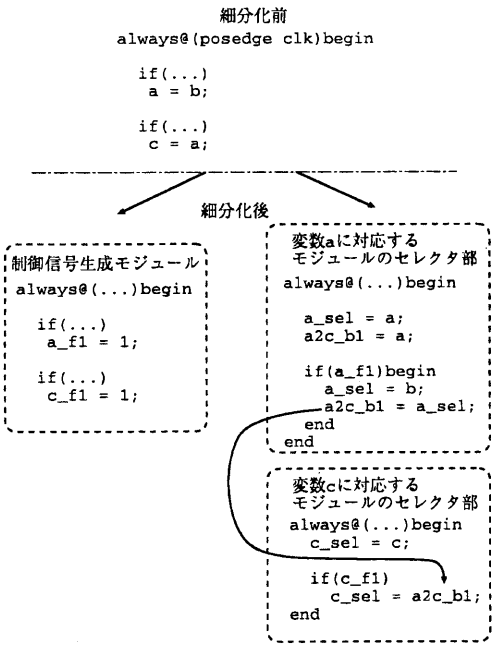

図 7: ブロッキング代入された変数の代入文での参照

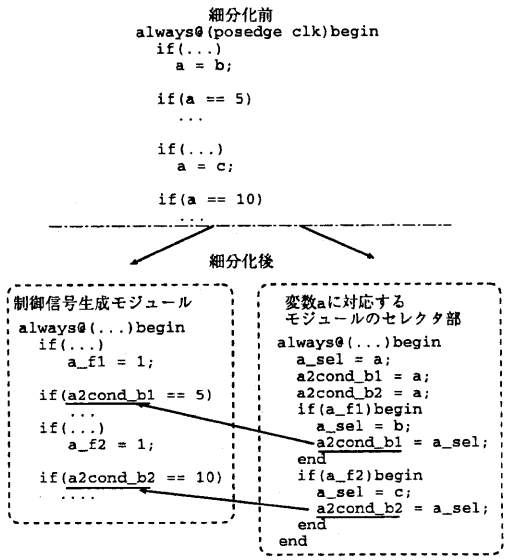

図 8: ブロッキング代入された変数の条件内での参照

3.3 タスク/ファンクションの取扱い

タスクは always 文中でのみ呼出されるサブルーチンである．Verilog HDL の文法上は数クロック後に結果が戻るような状態遷移を持つタスクを記述できる．しかし，現状の論理合成ツールでは一般にサポートされていないため，図 9に示すような組合せ回路のみを生成するタスクを細分化の対象とする．タスクが always 文中の複数箇所で呼出されている場合，どの呼出し箇所が同時に利用されるかを解析する．新たに生成したモジュールを共有できる場合は，新たに生成したモジュールへの入力を決定するセレクタとなるモジュールも同時に生成する．このセレクタにより共有可能なモジュールを共有するように制御することが可能となる．なお，タスク呼出しの解析は，条件文 (if 文，case 文) のネストから判断する．

ファンクションはタスクと違い組合せ回路のみ表現する．したがって，ファンクションも組合せ回路を生成するタスク場合と同様に組合せ回路のモジュールを新たに生成できる．複数のファンクションの呼出しがある場合には，同時に利用されるものを解析する．解析の結果共有可能な場合には，モジュールへの入力を選択するセレクタもタスク場合と同様に生成する．また，ファンクションは，always 文中だけではなく継続代入文でも使用できる．継続代入文で使用された場合，図 10のように継続代入文をモジュール化する際に新たに生成したモジュールを接続する．

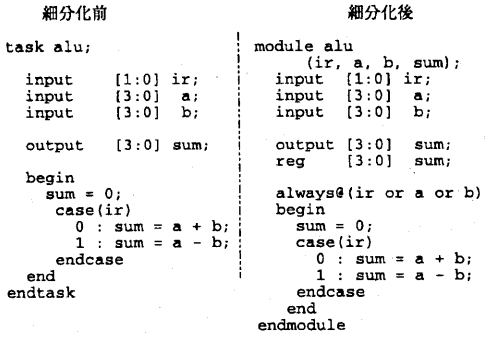

図 9: タスクのモジュール化

3.4 生成モジュールの再細分化

always 文の制御構造が複雑な場合，および，条件文の条件判定部分の比較演算などが複雑またはビット数が多い場合などでは，生成した各モジュールのセレクタへ制御信号を送るモジュールの回路規模が大きくなる．また，このモジュールはセレクタ

細分化前

```
    .
    .
    .
assign
  a = func(b,c);
    .
    .
```

細分化後

```
module a2m(b,c,a);
  input b,c;
  output a;
  wire [2:0] func_out;

  assign a = func_out;

  func2m
    func2m_inst
     (.in1(b),
      .in2(c),
      .out(func_out));

endmodule
```

図 10: 継続代入文内のファンクション

のモジュールと比較して接続が集中する．そこで，この各セレクタに対して制御信号を送るモジュールは，図 11のように制御文ごとに別モジュールとし，ある条件が成立したことを他のモジュールへ送るようにさらに細分化する．このようにさらに細分化を行うことで，条件判定に必要な回路を別々のモジュールとすることができ，また 1 モジュールへの接続の集中を解消できる．

同様に，モジュール化したタスクやファンクションの回路規模が大きい場合，および，接続数が多い場合にも，再細分化を行う．モジュール化したタスクやファンクションは組合せ回路を生成する always 文となるため，3.2 節 (2) で述べた手法により再細分化を行うことができる．

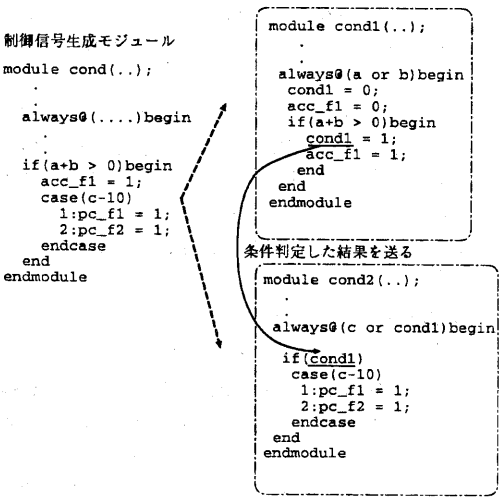

図 11: 条件生成部分の再細分化

4 細分化手法の評価

4.1 評価項目および環境

細分化を行うプログラムを作成し，そのプログラムを用いて 3 章で述べた細分化手法の評価を行った．評価項目として細分化前後での以下の 4 項目に着目した．

1. 動作の同一性
2. 論理合成時間
3. 回路規模
4. 動作速度

使用したベンチマーク回路は，検証環境が整っている本学で教育用に開発された KITE-1 マイクロプロセッサ [6] を用いた．単一のモジュールで記述した HDL コードを細分化した後，その全 HDL コードを結合するために必要となる最上位階層を生成し，1 つの FPGA に実装する．

論理シミュレータと FPGA の実装された評価用ボードの両方で動作検証を行い，細分化前後での動作の同一性を確認する．

論理合成は，EDA ツールの GUI 環境からの操作ではなくスクリプトファイルからの実行により行い，論理合成時間の測定には UNIX の time コマンドを用いターンアラウンドタイムを測定した．回路規模は配置配線時のログファイルより確認する．また，動作速度は米国 Xilinx 社製の遅延解析ツールを用いて測定した．

評価環境 (計算機，EDA ツール等) については以下の通りである．

- 使用計算機:米国 Sun Microsystems 社製 SPARC station 20 model 151 (主記憶 64 Mbyte)
- 論理シミュレータ:米国 Cadence 社製 VERILOG-XL Ver.2.3.3
- 論理合成ツール：米国 Synopsys 社製 Design Compiler Ver.3.5a
- FPGA 開発ツール (配置配線，遅延解析など)：米国 Xilinx 社製 XACTstep Ver.5.2.1
- テストボード:KITE Microprocessor Board PLUS$^+$ (米国 Xilinx 社製 XC4013pg223-5[約 13,000 ゲートを実装可能] を搭載)

4.2 評価結果

評価結果として，論理合成時間の比較を図 12に，回路規模の比較を図 13に示す．論理合成は，細分化後の HDL コードそれぞれについてソフトウェアを起動して行った時間を示している．最も時間を必要とした HDL コードは，細分化後の HDL コード全てを接続している最上位階層の HDL コードであり，合成時間は約 2 分であった．また，ALU と各

セレクタに対して制御信号を生成するモジュールのHDLコードが1分から2分であり，それ以外のセレクタまたはセレクタとフリップフロップのHDLコードの合成時間は30秒から50秒であった．また，細分化に必要であった時間は約1秒である．この結果から，細分化を行うことで論理合成時間の短縮化を確認した．細分化後のHDLコード全てを論理合成する場合は，細分化前の方が論理合成時間が短いようにみえるが，EDAツールを再起動せずに細分化後の全HDLコードを論理合成するスクリプトを生成し実行した場合の論理合成時間は約8分であり，細分化前よりも短時間であった．

細分化前後を比較すると，回路規模は細分化後の方が小さくなり，クリティカルパスはほぼ同じとなった．回路規模(図13)が縮小した部分は，LUT(Look Up Table)で実現される論理ゲートであり，フリップフロップの使用数は同じであった．一方，クリティカルパス(図14)となっている部分は同じであったため，細分化後の回路規模が縮小している部分はそれ以外の部分と考えられる．詳しいことは，現在調査中である．

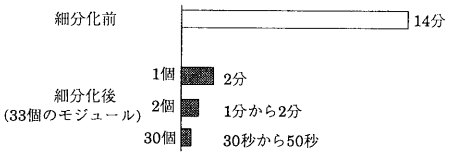

図 12: 論理合成時間の比較

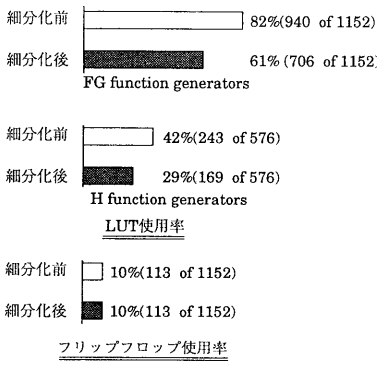

図 13: 回路規模の比較

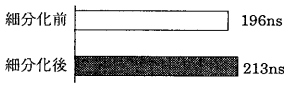

図 14: クリティカルパスの比較

5 おわりに

本稿では，FPGAを用いた論理エミュレータ上の複数のFPGAに対して，HDLレベルで回路を分割する手法について述べた．それは，HDLコードの細分化と合成結果のグループ化の2段階で行うと述べたが，本稿では細分化の手法について説明し現状での評価を述べ，論理合成時間の短縮を図れることを確認した．

今後の課題を以下に示す．

- **対応文法の拡張**：複数箇所で同期するalways文やループ文への対応を検討する．
- **リソースシェアリング**：リソースシェアリングを行うことで，あるリソースに対して接続が集中し論理エミュレータへの実装が困難になる場合がある．したがって，リソースシェアリングは実装する論理エミュレータの構成を考慮して行う必要がある．
- **多ビット変数への対応**：本稿で述べた細分化手法は変数に着目し細分化している．そのため変数のビット数が多くなり細分化後のモジュールが大規模になった場合に，現在の方法ではそれ以上細分化できない．多ビットの演算器などがある場合も同様に問題となる．このような場合は，ビットスライスによりあるビット数ごとに細分化する必要がある．

参考文献

[1] Hauck, S. and Borriello, G.: "Logic Partition Orderings for Multi-FPGA Systems," *Proceeding of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp.32–38, 1995.

[2] 空岡，田中，久我："HDL記述によるディジタル回路の複数FPGAに対する分割手法，" 情報処理学会 第53回 (平成8年後期) 全国大会講演論文集 (1) 1-25，1996.

[3] Synopsys, Inc.："Behavioral Compiler ユーザーガイド，" 1996.

[4] Synopsys,Inc.："HDL Compiler for Verilog リファレンスマニュアル，" 1996.

[5] Scott Hauck: "The Roles of FPGAs in Reprogrammable Systems," http://www.eecs.nwu.edu/ hauck/springbok/index.html

[6] 末吉，田中，久我："教育用マイクロプロセッサKITEによる設計教育事例 (第2報)，" 情報処理学会研究報告 (93-ARC-110-13, 93-DA-73-13)，1994.