Vol.2019-ARC-235 No.55
Vol.2019-SLDM-187 No.55
Vol.2019-EMB-50 No.55
2019/3/18

レジスタ変数削減によるサイクルベース型シミュレーションの高速化手法

田宮 豊* 池 敦*

概要：近年のプロセッサ等の大規模回路は非常に多くのレジスタを有しており、論理シミュレーション中に行われるレジスタ変数の更新回数の多さがシミュレーション速度の低下要因となっている。本論文では、レジスタ変数を削減するように回路記述を変換することにより、サイクルベース型シミュレーションの高速化手法を提案する。先ず、我々は“ブロッキング変数”を新たに提案し、シミュレーション結果を変えずにレジスタ変数をブロッキング変数に置換できる事を示す。本来は並列に評価されるプロセスを逐次的に評価することで、レジスタ変数からブロッキング変数への置換を促進する効果がある事を示す。更に、この効果を、回路モデル中の全プロセスと全レジスタ変数に対する依存関係として抽出して、最適なプロセス評価順序を求めるレジスタ変数の削減問題として定式化する。実験評価では、SystemC 標準シミュレータを用いた社内開発プロセッサ向け性能評価用論理シミュレータに本手法を適用した。その結果、 SystemC 標準シミュレータを用いたシミュレーションは 3.16 倍高速化された。また、レジスタ変数削減により SystemC 標準シミュレータのマルチスレッド化が可能となり、最終的に SystemC の元記述に比べて 7.51 倍の高速化を達成した。

キーワード：サイクルベース型シミュレーション, レジスタ変数, ブロッキング代入, SystemC

Speed-up Method of Cycle-based Simulation by reducing Register Variables

Yutaka Tamiya* Atsushi Ike*

Abstract: Recently, large-scale digital circuits, such as CPU processors, have enormous number of registers, which is one of the causes that result in their long logic simulation time. In this paper, we propose a method to speed up the cycle-based simulation by reducing register variables as pre-processing of logic simulation. First, we introduce a “blocking variable”, with which a register variable in the target circuit can be replaced without changing the result of the logic simulation. By sequentially evaluating processes in the target circuit model, which originally shall be evaluated in parallel, we can increase the possibility to reduce the register variables of the model. Furthermore, by extracting the dependencies between all processes and all register variables of the circuit model, we formulate the register variables reduction problem that finds the optimum process evaluation order. In our experiments, we've applied our proposed method to our in-house performance evaluation simulator, which uses SystemC standard simulation engine. As a result, we've sped up the simulator by 3.16 times in single thread mode, and by 7.51 times in multi-thread mode.

Keywords: Cycle-based simulation, Register variable, Blocking assignment, SystemC

* (株)富士通研究所 Fujitsu Laboratories, Ltd.

ⓒ2019 Information Processing Society of Japan

Vol.2019-ARC-235 No.55
Vol.2019-SLDM-187 No.55
Vol.2019-EMB-50 No.55
2019/3/18

1. はじめに

一般的なデジタル回路設計では、設計回路のモデルをHDL (Hardware Description Language)[1][2]で記述し、論理シミュレーションを繰り返して、設計回路の機能と性能を検証する。論理シミュレーションの実行時間は、回路モデルの規模(プロセスとレジスタ変数の個数)に大きく依存する。

一方、近年のプロセッサ(マルチコア CPU[3], GPU[4], 機械学習専用プロセッサ[5]等)では、その処理性能のニーズ増大と半導体テクノロジの進化によって、回路規模は年々増加の一途を辿っている。このようなデジタル回路のHDL 記述は多くのプロセスとレジスタ変数を有しており、論理シミュレーションに費やされる膨大な実行時間が設計現場で問題になっている。

2. 論理シミュレーションの動作原理

2.1 クロック同期回路モデル

一般的なデジタル回路設計で採用されているモデルはクロック同期回路である為、本論文ではクロック同期回路に対象を絞って、その HDL 記述と論理シミュレーションを説明する。

クロック同期回路モデルの HDL 記述は、並列に動作する部分回路に相当する "プロセス(process)"と、信号値を保持する flip-flop に相当する "レジスタ変数(register variable)"から構成されるネットワークである。その論理シミュレーションの動作原理は、プロセスの評価(evaluation)と、レジスタ変数の更新(update)の繰り返しである。

プロセスの評価では、レジスタ変数を読み出した後に、回路モデルに記述された演算を行い、その演算結果を出力となるレジスタ変数に書き込む。

レジスタ変数の更新は、物理的な電気信号の伝達時間をモデル化している。あるプロセスの評価によってレジスタに書き込まれる値は、回路モデル記述された遅延時間(記述されない場合は微小時間(Δt とも呼ぶ))経った後に、他のプロセスから読み出しが可能になる。この遅延時間を伴うレジスタ変数の更新を実装するため、論理シミュレータは、各レジスタ変数に対して、現在値(curr)と次時刻値(next)の 2 つの値を持たせる。レジスタ変数の読み出し動作では現在値を返し、書き込み動作では次時刻値を上書きする。レジスタの更新では、次時刻値を現在値にコピーする。

本論文で使う回路モデル例とその Verilog 記述を図 1 に示す。このモデルは 4 個のプロセス(3 個の順序回路と、1 個の組合せ回路)と、4 個のレジスタ変数から構成されている。各レジスタ変数(A,B,C,D)への代入文は、互いに並行動作するプロセスと考えることができる(コメントで示すとおり、 process1~4 と命名する)。また、各レジスタ変数 X (={A,B,C,D})を registerX と読み替えれば、この Verilog 記述は同図の回路モデルと等価になる。

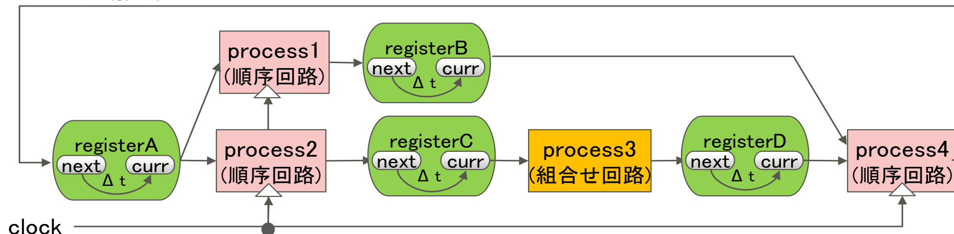
なお、HDL 記述におけるレジスタ変数への代入は、ノンブロッキング代入("<=")とブロッキング代入("=")の 2 種類が存在する。ノンブロッキング代入では、代入以降でレジスタ変数を読み出しても、更新前の値が読み出される。これは、レジスタ変数の更新が always 文ブロック(begin～end)の実行が完了する Δt 後まで遅延される為である。反対に、ブロッキング代入では、レジスタ変数の更新が即時行われ、代入以降のレジスタ変数の読み出しでは更新後の値が読み出される。

HDL の仕様で決められてはいないが、一般的な HDL 記述では、ノンブロッキング代入は順序回路(図 1 の process{1, 2, 4})に、ブロッキング代入は組合せ回路(同図の process3)に使われる。

2.2 サイクルベース型シミュレーション

イベント(プロセス評価とレジスタ変数更新)を原理通りに行う方式は、イベントドリブン型シミュレーションと呼ばれる。これは、イベントの動的管理が必要なためにシミュレーション速度が遅いという欠点を持つ。そこで、対象をクロック同期回路に限定することで、この欠点を改善するサイクルベース型シミュレーション(Cycle-based Simulation)[6]が提案された。1 クロック内に起こるプロセス評価とレジスタ変数更新のスケジューリングを予め決めておくことにより高速化する手法である。

前述の回路モデル例に対するサイクルベース型シミュレーションの実行例を図 2 に示す。ここでは、1 クロック



Verilog記述

```
module MOD (clock);
  input clock;
  reg [15:0] A=0, B=1, C=0, D=1;

  always @(posedge clock) begin
    B <= A + 1;      /* process1 */
    C <= A * 2;      /* process2 */
    A <= B * D;      /* process4 */
  end
  always @ (C)
    D = C + 1;       /* process3 */
endmodule
```

図 1 クロック同期回路モデル例

ⓒ2019 Information Processing Society of Japan

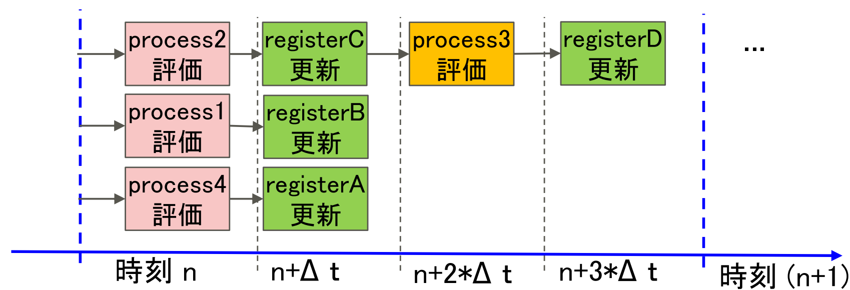


図 2　サイクルベース型シミュレーションの実行例

内のプロセス評価とレジスタ変数更新のスケジューリングを表している。先ず、時刻 n において、クロックに同期して process{1,2,4}が評価される。次に、時刻 n+Δt において、それらの出力 register{A,B,C}が更新される。次の Δt 後に組合せ回路 process3 が評価され、最後に registerD が更新される。なお、時刻 n における process{1 ,2, 4}は並列で評価が可能である。同様に、時刻 n+Δt における register{A,B,C}の更新も並列化が可能である。

サイクルベース型シミュレーションは、シミュレーション中に生じるイベントの動的管理を不要にしたことと、プロセス評価とレジスタ変数更新を並列に実行することにより、イベントドリブン型シミュレーションの数倍の高速化を達成した。大部分のデジタル回路はクロック同期で設計されているため、サイクルベース型シミュレーションは広く使われるに至っている。

しかしながら、近年の大規模な回路モデルに対しては、サイクルベース型シミュレーションの実行速度が不十分な事が分かってきている。特に、近年のプロセッサは SIMD 命令によって大量の演算器とレジスタが同時に動作するため、シミュレーション実行時間の問題は深刻である。

シミュレーション実行時間の大部分は、プロセス評価とレジスタ変数更新で占められる。この内、プロセス評価は、設計対象の回路が実現する論理機能そのものである為、これを削減することは出来ない。一方、レジスタ変数更新は、回路のモデル化に由来する処理であり、条件次第で削減が可能なことを我々は発見した。

レジスタ削減の手法として、re-timing[7]が提案されている。しかし、適用範囲が限られている上、回路構成によっては必ずしもレジスタの個数が減らないなど、効果が十分とは言えない。

本論文では、レジスタ変数の削減によってサイクルベース型シミュレーションを高速化する手法を提案する。我々の貢献は以下の 4 点である:

A) ブロッキング変数を新たに提案し、シミュレーション結果を変えずにレジスタ変数をブロッキング変数に置換できる事を示した。
B) 本来は並列評価されるプロセスを逐次評価にスケジューリングすることで、レジスタ変数からブロッキング変数への置換を促進できる事を示した。
C) 上記の性質を回路モデル中の全プロセスと全レジスタ変数に対する依存関係を抽出することにより、最適なプロセス評価順序を求めるレジスタ変数の削減問題として定式化した。
D) 提案手法を SystemC[8]記述の実プロセッサ開発用向け論理シミュレータに適用して、その高速性を示した。

本論文の以降の構成では、第 3 章では提案手法について、第 4 章では本手法の実設計への適用評価について、それぞれ述べる。最後に第 5 章で本論文をまとめる。

3. 提案手法

3.1 ブロッキング変数

我々は、シミュレーション結果を変えずにレジスタ変数を削減するために、新たに “ブロッキング変数(blocking variable)”を提案する。ブロッキング変数の定義は以下の通りである:

(定義)ブロッキング変数とは、論理シミュレーションの回路モデルにおいて、値を保持するレジスタを表す変数の一種で、書き込まれた値は、即時、読み出しが可能

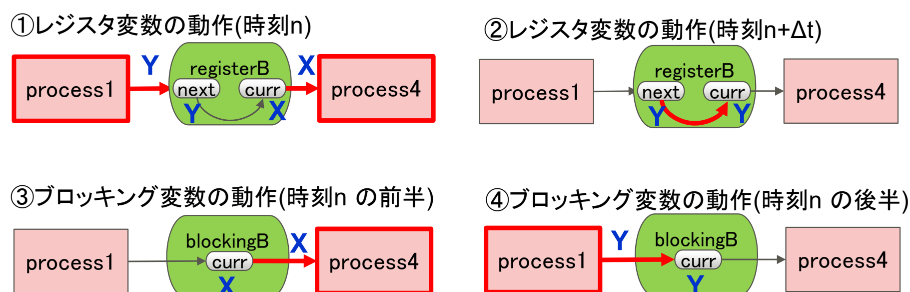


図 3　レジスタ変数とブロッキング変数の動作

ⓒ2019 Information Processing Society of Japan

となる変数。

ブロッキング変数は、HDL 記述上は、全ての代入にブロッキング代入(Verilog 文法では “=”)が使われるレジスタ変数と同じとなる。但し、論理シミュレータの実装観点からは異なっている。即ち、常にブロッキング代入で使われる為、ブロッキング変数は、現在値(curr) の 1 つのみを値として保持すれば良い。

レジスタ変数とブロッキング変数の違いを図 3 に示す。この図は、図 2 の論理シミュレーションにおいて、 プロセス process{1, 4}とレジスタ変数 registerB を取り出している。時刻 n において(図 3 ①)、registerB が持つ値は X とする。process1 と process 4 のプロセス評価は、 registerB に対して、それぞれ、書き込みと読み込みを行う。この時のレジスタ変数の動作では、process1 は次時刻値(next) として Y を書き込み、process4 は現在値(curr)から X を読み出す。その後の時刻 n+Δt において(図 3 ②)、registerB は更新されて現在値に値 Y がコピーされる。

このように、プロセスが並列に評価される時、同一のレジスタ変数に書き込みと読み書きが同時刻に行われても正しい動作をするよう、論理シミュレーションの実装では、レジスタ変数に 2 つの値(現在値と次時刻値)を持たせている。また、2 つの値を持つが故に、2 つ値を同期させる処理として、レジスタ変数の更新が必要となっている。

次に、レジスタ変数 registerB をブロッキング変数 blockingB に置換する(図 3 ③④)。ここで、ブロッキング変数が元のレジスタ変数と同じ動作をするように、プロセスを逐次的に評価することを考える。それには、ブロッキング変数を読み出すプロセスを先に評価し、その後、書き込むプロセスを評価すれば良い。本例では、先ず process4 を評価して値 X を読み出させ(図 3 ③)、その後(しかし、シミュレーション時刻としては同時刻)、process1 を評価して値 Y を書き込ませる(図 3 ④)。各プロセスは、レジスタ変数の場合(図 3①)と同じ値を読み書きするため、論理シミュレーションの動作が変わらない事が保証される。

ところで、プロセス process1 と process4 を本来の回路モデルの通りに並列に評価してしまうと、ブロッキング変数 blockingB に対する書き込みが読み込みよりも早くなる可能性が有り、その時はシミュレーション動作が異なってしまう。逆の見方をすると、本来は並列に評価されるプロセスを逐次的に評価することで、ブロッキング変数へ置換可能なレジスタ変数を増やすことが出来る。

このように、レジスタ変数をブロッキング変数に置換し、かつ、適切なプロセス評価順序を与えれば、論理シミュレーションの動作を変えずに、そのレジスタ変数を削減できる事が分かった。なお、実際のデジタル回路において、ブロッキング変数に相当する回路要素は存在しない。あくまでも、論理シミュレーションを高速化するためのモデル概念に過ぎない。

3.2 レジスタ変数削減問題の定式化

前節の例では、1 つのレジスタ変数に注目してプロセスの評価順序を決めれば、そのレジスタ変数をブロッキング変数に置換できることを示した。実際の回路では、1 つプロセスが複数のレジスタ変数を読み書きするし、1 つのレジスタ変数も複数のプロセスから読み書きされる。その為、最適なプロセス評価順序の決定と、レジスタ変数を最大限に削減するためには、全プロセスと全レジスタ変数を同時に考慮することが必要になる。本節では、回路モデル中の全プロセスと全レジスタ変数の依存関係を Read-Write 制約として抽出し、これを 1 つの表に纏めることにより、レジスタ変数の削減は、最適なプロセス評価順序を求める問題として定式化できることを示す。

先ず、回路中の全プロセスと全レジスタ変数を抽出し、どのプロセスがどのレジスタ変数を読み出し(Read)、または、書き込み(Write)しているかを調べる。また、各プロセスについて、順序回路か組合せ回路かのプロセス種類も調べる。プロセスを行に、レジスタ変数を列にした表を作成し、先に抽出した情報を記入する。この表をレジスタ変数依存表と呼ぶ。

表 1 回路モデル例のレジスタ変数依存表とその最適化結果

(1)オリジナルのレジスタ変数依存表

	プロセス種類	registerA	registerB	registerC	registerD
process1	順序回路	Read	Write		
process2	順序回路	Read		Write	
process3	組合せ回路			Read	Write
process4	順序回路	Write	Read		Read

(2)最適プロセス実行順序にレジスタ変数依存表

	プロセス種類	registerA	registerB	registerC	registerD
process4	順序回路	Write	Read		Read
process1	順序回路	Read	Write		
process2	順序回路	Read		Write	
process3	組合せ回路			Read	Write

次に、各レジスタ変数の Read と Write に対して、下記の Read-Write 制約に従った矢印を表に記入する。

(Read-Write 制約)

A) 順序回路プロセスの Read から、他プロセスの(複数 Write が有る場合は最初の)Write へ矢印を付ける(表 1 では赤実線矢印)。レジスタ変数がこの制約を満たす時、全ての Read が実行された後で Write されることが保証される。

B) 組合せ回路プロセスの Read へ、 他プロセスの(複数 Write が有る場合は最後の)Write から矢印を付ける(同表では緑破線矢印)。この制約を満たすレジスタ変数は、組合せ回路プロセスへの入力として正しい動作が保証される。

C) 同一レジスタ変数へ複数プロセスの Write がある場合、その順序に従って矢印を付ける(青太線矢印。但し同表では存在しない)。この制約は、レジスタ変数への複数回のノンブロッキング代入(書き込み)は、最後の書き込みが有効になるという HDL 文法を反映している。

全ての Read-Write 制約を満たすレジスタ変数は、これをブロッキング変数に置換してもシミュレーション動作が変わらないことが保証される。

図 1 に対するレジスタ変数依存表を表 1(1)に示す。ここで、レジスタ変数依存表におけるプロセスの評価順序を、行の一番上から下へと定めると、全ての矢印が上から下に向かっているレジスタ変数は Read-Write 制約を満たす。表 1 (1)では register{A,C}が Read-Write 制約を満たす。よって、これらのレジスタ変数はブロッキング変数に置換して削減できる。逆に、他の register{B,D}はブロッキング変数に置換できず、レジスタ変数として残る。この事実により、レジスタ変数の削減は、レジスタ変数依存表において Read-Write 制約を満たすレジスタ変数の個数を最大化する行順序を求める問題として定式化できる。

この例では、register{A,B}の Read-Write 制約が循環しており、全てのレジスタ変数について Read-Write 制約を満たすプロセス評価順序は存在しない。しかし、process{4, 1, 2, 3}の評価順序にすれば、Read-Write 制約を満たすレジスタ変数が 3 個(register{B, C, D})と最適になる(同表(2))。

この最適なプロセス評価順序とレジスタ変数のブロッキング変数への置換結果の Verilog 記述を図 4 に示す。代入文の順序(プロセス評価順序)を変更し、かつ、{B,C,D}のレジスタ変数に対するノンブロッキング代入(<=)をブロッキング代入(=)に変更している。代入にブロッキング代入しか使われていない{B,C,D}は、ブロッキング変数へ置換されている。レジスタ変数 A は、Read-Write 制約を満たさないため、ブロッキング変数に置換せず、代入文はノンブロッキング文(<=)のままにする。この時のシミュレーション動作を同図に示す。クロックに同期して、process{4, 1, 2, 3}の順で評価する。その後、registerA を更新する。図 2 と比較すると、register{B, C, D}の更新が削減されており、この更新に伴う計算量,および,データアクセス量が削減され、シミュレーションの高速化が期待される。

4. 実験評価

本手法の評価として、SystemC[8] 標準シミュレータを用いた社内開発プロセッサ向け性能評価用論理シミュレータに適用した。SystemC 文法において、レジスタ変数はテンプレートクラス sc_signal に、ブロッキング変数はモジュールクラスのメンバ変数に、それぞれ対応する。従来はブロッキング変数の概念が無かったため、これらがレジスタ変数削減に使われてなかった。

SystemC 記述でのブロッキング変数への置換例を図 5 に示す(本例では B, C, D が置換されている)。今回のプロセスの評価順序は、第 3 章で述べた定式化をした後に、発見的手法で順次 Read-Write 制約のループを切るように定めた。定められた通りの順序でプロセス評価を実現するためには、モジュールのプロセス評価を SystemC 標準シミュレータのスケジューラに任せず、トップモジュールのプロセスから直接実行するように実装した。また、レジスタ変数の削減によって全ての sc_signal が削除されたプロセスは、

レジスタ変数削減後のVerilog記述

```
module MOD (clock);
  input clock;
  reg [15:0] A=0;
  reg [15:0] B=1, C=0, D=1;//blocking
  always @(posedge clock) begin
    A <= B * D;      /* process4 */
    B = A + 1;       /* process1 */
    C = A * 2;       /* process2 */
    D = C + 1;       /* process3 */
  end
endmodule
```

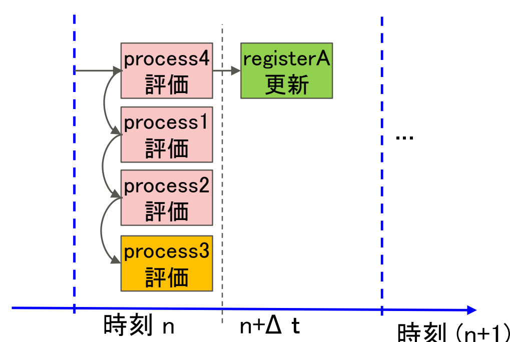


図 4 レジスタ変数削減後の Verilog 記述とシミュレーション動作

ⓒ2019 Information Processing Society of Japan

Vol.2019-ARC-235 No.55
Vol.2019-SLDM-187 No.55
Vol.2019-EMB-50 No.55
2019/3/18

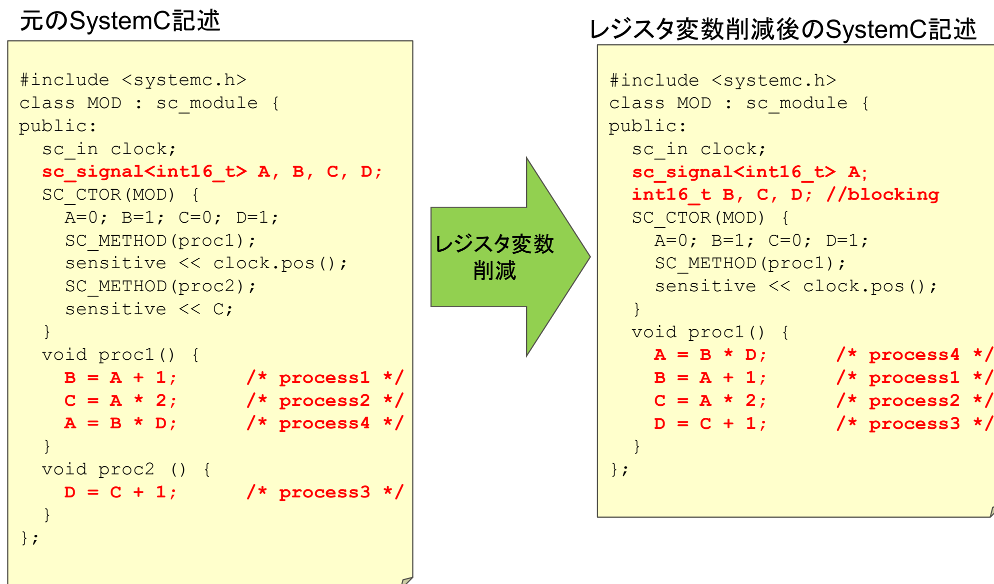


図 5　SystemC 記述におけるレジスタ削減例

SystemC 標準シミュレータを用いているにも関わらず、マルチスレッドによる並列実行シミュレーションが可能となるという副次的効果も得られた。

2 種類のプログラムコードに対するシミュレーション時間の比較を表 2 に示す(実行マシンは、Intel CPU Xeon E5-2637v4, 3.5GHz (8 論理コア) x 2CPU)。今回はレジスタ変数の 92.9%が削減されており、この効果が最大 3.16 倍のシミュレーション高速化結果に表れている(表中のシングルスレッドの列)。マルチスレッド実行では最大 7.51 倍と、更に高速化されることが分かる。但し、スレッド数が 128 にも関わらず、マルチスレッド効果は 2.36 倍(=17,251 秒÷7,312 秒)に留まった。これには、スレッド数とスレッド粒度の設定に改善の余地があると考えられる。

5.　おわりに

本論文では、レジスタ変数を削減し、サイクルベース型シミュレーションを高速化する手法を提案した。その特徴は、レジスタ変数のブロッキング変数への置換と、プロセスの逐次的評価順序の決定にある。実設計の論理シミュレータへの適用を通じて、本手法の効果を示した。

今後の課題として、マルチスレッド実行のスレッド粒度の最適化を挙げる。本手法を別の側面から見ると、本来並列に評価されるプロセスを逐次的に評価することによって、削減可能なレジスタ変数を増やしている。もしも、全てのプロセスを逐次化すると、最大限のレジスタ変数削減が可能になる一方で、スレッド数が 1 になってしまい、シミュレーション全体として高速化されるとは限らない。この事は、スレッド粒度とレジスタ変数の削減との間にトレードオフが存在することを示唆している。今後は、このトレードオフを本論文で定式化したレジスタ変数削減の最適化問題に取り込むとともに、この最適化問題の効率的な解法に取り組みたい。

参考文献

[1] “IEEE Standard for Verilog Hardware Description Language”, IEEE 1364-2005, 2005.
[2] “IEEE Standard VHDL Language Reference Manual”, IEEE 1076-2008, 2008.
[3] “The K computer”, FUJITSU SCIENTIFIC & TECHNICAL JOURNAL, 2012-7 (Vol.48, No.3), 2012.
[4] “NVIDIA Volta AI Architecture NVIDIA”, https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/, 2018.
[5] “Google Cloud TPU”, https://cloud.google.com/tpu/, 2018.
[6] Kei-Yong Koo and Alan N. Willson, Jr., “Cycle-Based Timing Simulations Using Event-Streams”, Proc of ICCD 1996, pp. 460-465, 1996.
[7] Kumar N. Lalgudi and Marios C. Papaefthymiou, “Retiming Edge-Triggered Circuits Under General Delay Models”, IEEE Trans. On CAD, Vol. 16. No. 12, 1997.
[8] “IEEE Standard for Standard SystemC Language Reference Manual”, IEEE 1666-2011, 2011.

表 2　本手法の実設計適用結果

	実時間 [秒]		
	オリジナル	レジスタ変数削減 (シングルスレッド)	レジスタ変数削減 +128マルチスレッド
プログラムコード1	18,567	8,979 (2.44倍)	2,713 (6.84倍)
プログラムコード2	54,540	17,251 (3.16倍)	7,312 (7.51倍)

ⓒ2019 Information Processing Society of Japan