シスラムＬＳＩ設計技術　103－24
（2001．11．30）

ハードウエア設計のマルチコンテキスト化手法

金子　直人[†]　宇野　正樹[†]　渡邊幸之介[†]　山本　淳二[††]　工藤　知宏[††]
天野　英晴[†]

† 慶応義塾大学 理工学部
†† 新情報処理開発機構

E-mail: †{kaneko,uno,watanabe,amano}@am.ics.keio.ac.jp, ††{junji,kudoh}@trc.rwcp.or.jp

あらまし　マルチコンテキストFPGAは複数のハードウエアコンテキストをチップ内に持ち，これらを切り替えながら実行することができる．我々はこのようなデバイスの利用に向けマルチコンテキスト設計モデルを提案し，本稿では特にそのコンテキストスケジューラについての議論を行う．スケジューラはディマンドドリブンに動作し，適当なコンテキストの動的選択を可能にする．また，コンテキストが資源待ちによりブロックされている間に別の処理を行うサスペンションも実現する．予備評価をルータチップの設計上で行い，その有効性を示すことができた．

キーワード　マルチコンテキストFPGA，コンテキスト スケジューリング，マルチコンテキスト設計モデル，リコンフィギュラブル コンピューティング

A Context Scheduling Mechanism for Multi-Context Hardware Designs

N. KANEKO[†], M. UNO[†], K. WATANABE[†], J. YAMAMOTO[††], T. KUDOH[††], and H. AMANO[†]

† Keio University
†† Real World Computing Partnership

E-mail: †{kaneko,uno,watanabe,amano}@am.ics.keio.ac.jp, ††{junji,kudoh}@trc.rwcp.or.jp

Abstract　Multi-context FPGA houses several hardware contexts on-chip, and is capable of switching among them at run time. This paper describes a context scheduling mechanism in our multi-context design model to utilize such devices. The demand driven scheduler provides the dynamic selection of an appropriate context. Also, it realizes the execution of an alternative process while a context is blocked for resources. The preliminary evaluation taken on a router design shows the validity of the mechanism.

Key words　Multi-context FPGA, context scheduling, multi-context design model, reconfigurable computing

1. はじめに

FPGA上に複数のConfiguration RAMを持ち，これを高速に切り替えるマルチコンテキストFPGAは，NECによるDRL [1] の開発によりいよいよ本格的な実用化時代を迎えようとしている．我々は，1992年よりマルチコンテキストFPGAを利用して，仮想ハードウェア機構を実現するためのシステムWASMII [2], HOSMII [3] の研究を行ってきた．これらの研究では，マルチコンテキストFPGAのConfiguration RAMを制御する手段としてデータ駆動型の原理を利用した．この方法は，「演算を目的とするハードウェア」に対しては有効であり，かなり広い応用分野で，少量のハードウェアで大規模な問題を処理することができることを示した．

本稿では，演算処理に限らず，マルチコンテキストFPGAを対象としたより一般的なハードウエア設計の分割モデルと，そのスケジューリング機構を提案する．その実現には (1) 設計のコンテキスト分割アルゴリズム，(2) コンテキストのスケジューリング法，そして (3) コンテキストのスイッチイング機構，を定める必要がある．これらのステップはそれぞれ難しい課題を含むが，ここでは，特に2番目のコンテキストのスケジューリング法について提案し，ルータチップのバーチャルチャネル部のマルチコンテキスト化に適用した結果を検討する．

2. マルチコンテキスト設計

本研究では，マルチコンテキスト設計の枠組みとして図1に示す階層的な概念的構成を提案する．

設計のトップレベルは常に回路化されている部分と，必要に応じて回路化される部分から構成される．前者をFixed Region，後者をShared Regionと呼ぶ．Shared Regionではコンテキストスイッチによりハードウェア構成が切り替わる．ここで，あるハードウェアコンテキストの回路情報が，実際の回路として用いられることを，コンテキストがアクティベイトされると呼ぶ．

Shared RegionはさらにContext Group，Context，Moduleの3階層を形成する．Shared Regionは1つ以上のContext Groupから，Context Groupは1つ以上のContextから，Contextは1つ以上のModuleからそれぞれ構成される．コンテキストのスケジューリングおよびスイッチングはContext Group単位で行われる．すなわち，各Context Groupで，ある瞬間にアクティベイトされているContextは常に一つ

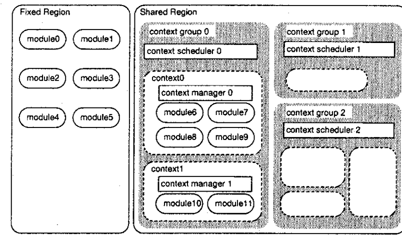

図1 設計の構成

のみである．

各ContextにはContext Managerが，各Context GroupにはContext Schedulerがそれぞれ付随しコンテキストスイッチングを実現するための各種制御を行う．Context Managerの役割は以下の5つである．

- 担当するContextの状態遷移の管理
- スケジューリングに必要な情報の保持
- Moduleの動作制御
- 隣接Contextとの通信
- Context Schedulerとの通信

また，Context Schedulerの役割は以下の4つである．

- 担当するContext Groupの状態遷移の管理
- Contextのスケジューリング
- 隣接Context Groupとの通信
- Context Managerとの通信

上記の構成は，DRLなど部分的にコンテキストスイッチを行う機能を持つFPGAにより実現することが可能である．

2.1 設計サイズ

マルチコンテキストFPGAを効率よく利用するためには，適切な大きさにContext Groupのサイズを決める必要がある．

マルチコンテキストにより実現する場合に必要な面積は式1のように表すことができる．

$$\text{設計のサイズ} = Fixed\ Region + \sum \text{各}\,Context\ Group\,\text{の最大}\,Context + \text{オーバヘッド} \quad (1)$$

ここで，オーバヘッドは，(1) コンテキストのスケジューリングに関わるハードウエア，(2) コンテキスト間のデータ受け渡しに用いるレジスタ等，常時アクティベイトされている必要があるハードウェアを指す．

Context Groupのサイズは，Group内の最大Contextによって決定される．従って，面積効率を最大に

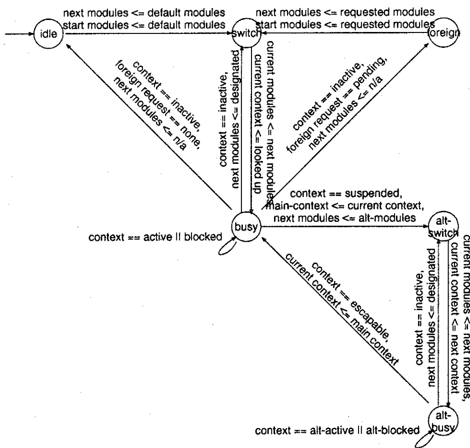

図 2　Context Group の状態遷移

するには Shared Region を多く取り，できるだけ等しい大きさで，デバイスの許す限り多くの Context へと分割する必要がある．

3. コンテキストスケジューリング

Shared Region の各 Context Group では，状況に応じて適切な Context を選びアクティベイトする必要がある．これがコンテキストスケジューリングである．Context の選択は Context Scheduler によってディマンドドリブンに行われダイナミックである．コンテキストスイッチ要求は Context Manager が管理下の Module より検出し，それを Context Scheduler へ伝える．

Context Scheduler の基本動作は，要求に従って Context Group 内の Context を切り替えることであるが，Context Group 外からの Context 利用要求に応えたり，逆に外部に対して要求を送る等の Context Group 間通信も行う．また，Context Group 内の処理が外部資源の利用待ちによりブロックされる状況で，別の処理を行う suspention も可能である．図 2，図 3 はこれらの一連の制御を司る状態遷移図である．

Context Group 内の処理は「流れ」でとらえる．起点となる Module から始まり，次々と Module の要求に応じて Context を起動し，もう次に起動すべき Context がなくなると終点となる．この軌跡が処理の流れである．通常の流れと suspention 中の流れが存在し，それぞれに適当な制御が必要である．状態遷移図中の，“alt-” で始まる状態は処理が suspention 中の流れにあることを表す．

3.1 基本的な動作

スケジューリングの基本動作は，idle 状態から始

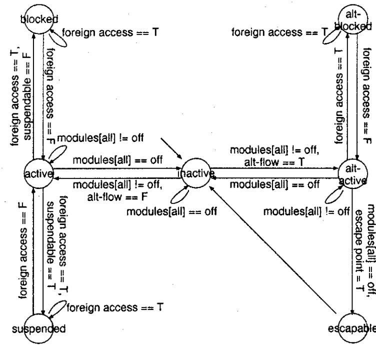

図 3　Context の状態遷移

まり，switch 状態と busy 状態の間を行き来し，再び idle 状態へ戻ることの繰り返しである．

初期状態から順に基本動作を解説する．各 Context Group には default modules が設定されている．idle 状態では，これらを次に処理を行う Module として next modules に指定してから switch 状態へ遷移する．

switch 状態ではこれを受けて next modules が所属する Context をアクティベイトし，busy 状態へと遷移する．この時，図 3 に示す通り選択された Context は inactive 状態から active 状態へと遷移する．inactive 状態は Context では何ら処理が行われていないことを，active 状態はアクティベイトされて何らかの処理が行われていることを表す．

active 状態の Context が何らかの処理を行っている間は Context Group は，busy 状態にとどまる．そして，現在の Context で行える処理が一通り終わりそれが inactive 状態となった時，条件に従い idle 状態，switch 状態，foreign 状態のいずれかへと遷移する．

この条件は以下の通りである．

（1） 次に処理を行いたい Module として next modules が指定されている場合は switch 状態へと戻る．Context Scheduler は適当な Context をアクティベイトして現在の処理の流れを続行する．

（2） next modules が指定されておらず外部からの Context 利用要求も届いておらずもはや何もするべきことがなければ，idle 状態へ還る．これでまた処理の振り出しに戻ることになる．

（3） しかし，後に述べる Context Group 間通信により，Context 利用要求が届いていれば，foreign 状態へ遷移しこれに応える．

3.2 Context Group 間通信

外部の Context Group にある Context を foreign

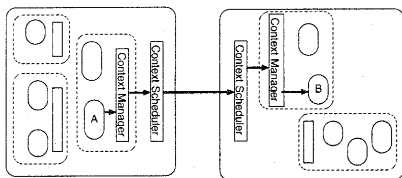

図 4　Context Group 間通信

context，これを利用するための Context Group 同士のやり取りを Context Group 間通信と呼ぶ．これは Module 同士の通信であるが，図 4 に示すように Context Manager と Context Scheduler を介して行い諸々の整合をとる．図中では Module A が Module B へと要求を送っている．外部から届く Context 利用要求を foreign request，逆に外部へ送る Context 利用要求を foreign access とする．

3.2.1　foreign request の処理

foreign request は Context Scheduler がペンドしておく．条件が揃い Context Group が foreign 状態になったら，要求された Module を next modules に指定して switch 状態へ遷移する．その後の処理は通常の Context Group 内部からの要求と同様である．

foreign 状態へ遷移するのは，Context Group 内の処理が一段落し，要求がなければ idle 状態に戻る場合である．このため，Context Group 内の資源をどのように用いても書き潰しなどの障害が生じることはなく，安全である．

3.2.2　foreign access の実行

foreign access は目的の Module の所属する Context Group の Context Scheduler に対して行われる．相手側ではこれを foreign request として処理する．foreign access を行う Context は，処理が返ってくるのを待たねばならない．

この待ち方には blocked 状態と suspended 状態の二種類がある．blocked 状態では Context Group 全体が完全に止まってしまう．一方，suspended 状態では待っている間に別の処理を許す．しかし，suspention の実現には以下に示す制約がある．

3.3　suspention の実現

suspention が発生し得るのは foreign acceess を行って，処理が戻ってくるのを待っている時である．例えば，Context Group 間で共有されるメモリからデータを取得する場合や，いっぱいになった FIFO に空きができるのを待っている間などが考えられる．suspention の実現は，処理の安全性を守るために制限が必要であり，また foreign access の結果として長く待たされることが予測される場合でないと，かえって性能の低下を引き起こしかねない．

suspention を完全に実現するには，全ての Context において，許容する suspention の深さに対応するだけ記憶資源を持つ必要がある．このような資源があれば，複数の状態を同時に保持することができ，処理の流れごとに切り替えて動作させることができる．しかし，この方法は大量の記憶資源を必要とするため，現在 suspention 中の Context の処理に関わらない Context のみを使った処理を許す方法をとる．こうすることで可能な処理は限られたものになるが，追加資源なしで一段階の suspention が実現できる．

suspention の仕組みを考える場合，Context Group 内の処理の流れを意識しなければならない．まず start modules を起点とし，現在 suspend している Module までの処理の流れを main-flow とする．start modules になり得るのは default modules と foreign request により利用の要求を受けた Module の 2 種類である．

一方で suspention 中に実行可能な処理の流れを alt-flow と呼ぶ．ある Module からの suspention が可能であるかどうかは，main-flow と重ならないような，alt-flow を形成することができるかどうかによって決まる．

3.3.1　alt-flow の形成

alt-flow を形成するためには，suspention を行う Module が適切な alt-modules を用意する必要がある．適当な alt-modules が見つからなければ，Module は suspention できない．alt-modules とは suspention 中の処理の起点となる Module で，以下の二つの条件を満足する必要がある．

- main-flow と交わらない alt-flow をいずれかの escape point と共に形成できること．ここで，escape point とは main-flow なら idle 状態へ遷移できる状況に相当し，安全に処理を終えることができる．図 5 にこの様子を表す．
- 処理の流れが escape point へ到達し，処理が終了することが保証されていること．到達し得る escape point は複数あってもかまわないが，main-flow との資源依存によるデッドロックや処理が永遠に巡回するようなことが起ってはならない．

尚，これらの条件が alt-modules 選定に関して保証するのは安全性のみである．この alt-flow で suspention を行って処理上の辻褄が合うかどうかは設計の対象に依存する．

また start modules が違えば main-flow も変化する

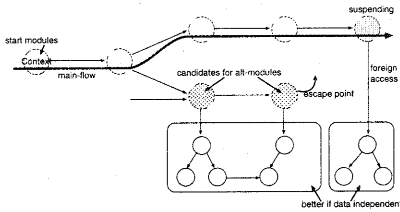

図 5　alt-modules の選定

ため，suspention 可能であるかの判断と alt-modules の選定は，可能性のある start modules ごとに行う必要がある．従って，alt-modules は suspention 可能な Module と start module の組み合わせに対して与える必要がある．

性能向上につながる suspention を行うためには，処理の並列性を考慮した alt-modules の選択が不可欠である．このため，alt-flow 上の Context から辿れる foreign context と，suspention 中の Module が利用している foreign context は独立していることが望ましい．折角 suspention をしても，資源依存で処理が直列化されるようでは意味がなくなってしまう．

3.3.2　suspention 中のスケジューリング

suspention 中もスケジューリングは通常とほぼ同様である．通常のスケジューリングの switch 状態に相当する alt-switch 状態，busy 状態に相当する alt-busy 状態が存在し，同様に機能する．

suspention は Context Group が busy 状態から alt-switch 状態へと遷移することで開始する．この際，next modules には alt-modules を設定する．suspention を行う Context は suspended 状態になり，後の復帰のために main-context として記録される．

alt-flow の処理は，alt-switch 状態と alt-busy 状態の間を行き来してを進められる．alt-flow 中の Context は通常の active 状態のかわりに alt-active 状態で起動される．こうして alt-busy 状態へ遷移するのだが，その動作には busy 状態と違う点が 3 つある．

- alt-flow からの suspention は行わない．suspention 可能な Module から foreign access を行う場合でも Context は alt-blocked 状態になるだけである．
- alt-flow 中は foreign request を受け付けない．これは alt-flow が終了した時点では Context Group 中には main-flow の処理が残っており，迂闊にこれらに関わる状態を変えるわけにはいかないためである．
- escape point へ達すると alt-busy 状態は busy 状態へと戻る．これは通常の制御ならば idle 状態へと遷移する状況に相当する．Context は alt-flow で escape point へ達すると escapable 状態になってそのことを知らせる．

これで suspention 処理自体は実現されるが，main-flow と alt-flow の処理内容の負荷バランスには注意を払う必要がある．alt-flow が escape point に達するまでは main-flow へ処理は戻されないため，場合によってはシステム全体の効率が悪化する．

4.　評　　価

マルチコンテキスト設計の一例として，仮想チャネルを持つルータチップを設計した．

4.1　対象の設計

設計したルータは 2 入力 2 出力で，それぞれ 2 本ずつの仮想チャネルを持つ．図 6 にその構成を示す．パケットヘッダに記述される目的地アドレスでルーティングテーブルを引き，クロスバースイッチを制御して適当な出力線から送出する．

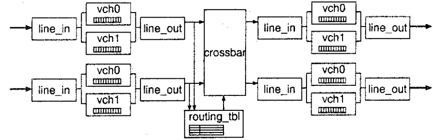

図 6　ルータの設計

入力部と出力部を構成するモジュールは同一で，マルチコンテキスト化したのはこの部分である．図 7 のように，line_in, line_out は Fixed Region の Module とし，仮想チャネル用のバッファを持つ vch0 と vch1 を Context Group 内でそれぞれ Context へ割り当てこれらを切り替えながら使用する．

この切り替えが発生するのはバッファ内のデータが目的地へ届いた時，あるいはバッファは空いているが入力データが存在しない時である．起動する Context の選択ルールは図 2 の状態遷移に倣って以下のように定める．

（1）　line_in からの foreign request が届いていれば，foriegn 状態になりこれに応じる．foreign request で指定されるのは，パケットが届いている方のチャネルの Module である．

（2）　foreign request が届いていなければ idle 状態になり，default modules である vch0 を起動し，パケットの到着を待機する．

4.2　マルチコンテキスト設計の実現

マルチコンテキスト FPGA をシミュレーションするため，アクティブな Context 以外はクロックを停め，出力線をセレクタで切断することで擬似的にこ

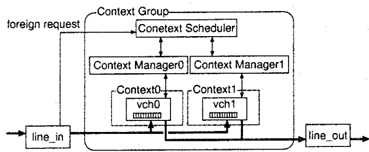

図 7　マルチコンテキスト化後のチャネル部

表 1　マルチコンテキスト化前後のセル面積

Module	before	after
line_in	0.001906	0.001985
vch	0.063118×2	0.065422
line_out	0.023509	0.008330
Context Manager	–	0.017630
Context Scheduler	–	0.040021
合計	0.151651	0.133388

れを実現した．

マルチコンテキスト化のために，元の設計に対しておこなった作業は，(1)Context Scheduler と Context Manager の作成，(2)vch に Context Manager との情報をやりとりするための組み合わせ回路を付加，(3)line_in に foreign request を発行する組み合わせ回路を付加である．

設計自体への変更点としては line_in，line_out をより簡素なチャネルが 1 本しかない場合と同じ設計にした．マルチコンテキスト化後は回路として存在するチャネルは常にどちらかの片方であるためこれが可能になった．

4.3　機能の評価

上記のルータを Verilog-HDL で記述し，Cadence 社の Signalscan を用いてシミュレーションした．また，Synopsys 社の Design Compiler と Rohm 社の 0.6μ プロセスのライブラリを用いて論理合成を行った．ルータ自体の機能を正しいことを確認するため，疑似乱数で決定した目的地アドレスとチャネルを持つパケットを 100 万個送信し，マルチコンテキスト化の前後ともに，全てのパケットが目的の出力線の意図するチャネルへ到達することを確認した．

4.4　面積の評価

表 1 に示す通り，マルチコンテキスト化前後で論理合成した結果のセル面積の合計を減らすことができた．line_in と vch が若干大きくなったのは式 1 に示されるオーバヘッドである．一方，設計の簡素化により line_out は大分小さくなった．今回の例では，マルチコンテキスト化の最大の目的である面積削減効果があまり実現されていないが，以下の 2 点より将来性は高いと考えられる．

一つは今回の Context Scheduler と Context Manager は，その機能を忠実に実現することを最優先し，対象の設計に依存しないよう配慮したため，最適化されたものではないことである．すなわち，より大きな設計をマルチコンテキスト化する場合でも，最大でもこの程度のオーバヘッドで実現できる．

次に，今回の設計で大きな面積を消費している Context Scheduler は，Context を追加しても増やす必要がない．したがってルータの仮想チャネル数を追加して 4 チャネルにする場合，Context Manager ふたつ分の面積 (0.035260) で実現できる．これは，vch ふたつ分の面積 (0.130844) と比較して大きく有利になる．

4.5　性能の評価

マルチコンテキスト化後の性能は約 4 分の 1 へ落ちてしまった．元の設計ではクリティカルパスが 3.30ns で，1 パケットが通過には平均 3clock を要した．しかし，マルチコンテキスト化することで，これらがそれぞれ 6.85ns，5.7clock へと延びてしまった．

クリティカルパスについては，既述のように今回の実装が特定の設計に対して最適なものではないことから大いに改善の余地がある．

所要クロック数については，Context の効率の良い分割および切り替え条件の探索が今後の課題となる．また，switch 状態を busy 状態へ圧縮できる可能性についても検討している．

5. おわりに

本稿では，ハードウエア設計の分割モデルと，そのスケジューリング機構を提案し，特にコンテキストのスケジューリング法について議論とルータの設計例に基づく評価を行った．その結果，提案したスケジューラで正しくマルチコンテキスト動作し，面積削減効果が期待できることがわかった．今後は，性能面でも満足のゆくマルチコンテキスト設計を実現ために，スイッチング機構の最適化を行う必要がある．また，分割法について，現実的な検討を行う必要がある．

文　献

[1] M.Yamashina, M.Motomura, "Reconfigurable Computing: Its concept and practical embodiment using newly developed DRL LSI," Proc. ASP-DAC 2000, pp.329–332, (2000).

[2] X.-P. Ling, H. Amano, "WASMII: A Data Driven Computer on a Virtual Hardware" Proc. FCCM '93, pp. 33–42, 1993.

[3] Y.Shibata, H.Miyazaki, X.Ling, H.Amano, "HOSMII: A Virtual Hardware Integrated with DRAM," Proc. of Parallel and Distributed Processing Workshop, LNCS 1388, pp. 85-90, (1998).

盛光印刷所