令和 2 年度

修士学位論文

FPGA を対象とした データ駆動型プロセッサの 設計自動化フローの検討

A Study of FPGA Circuit Design Automation Flow for Data-Driven Processor

1235066　　長野 寛司

指導教員　　岩田 誠

2021 年 3 月 1 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

FPGA を対象とした
データ駆動型プロセッサの
設計自動化フローの検討

長野 寛司

近年，IoT（Internet of Things）デバイスが様々な用途に利用され，マイクロプロセッサアーキテクチャには省電力性と高性能が求められる．データ駆動型プロセッサ（DDP）[2]は，低消費電力で動作し，異なるデータ流を多重に処理可能で高性能なため，IoT システムに有効なアーキテクチャである．また，FPGA は応用に合わせた柔軟な回路設計が可能で，多様性が求められる IoT デバイスに有効である．よって，DDP を FPGA 上に実装して IoT デバイスを実現する手法が有望である．しかし，既存の FPGA と回路設計ツールは同期式回路に最適化で，非同期式回路の DDP を最適に設計することが困難だ [6]．FPGA に DDP を実現する過去の手法 [3] では，回路の動作保証が不十分であった．それに対し，[4] は動作保証が可能な手法である．しかし，回路設計ツールの機能を巧妙に利用し，独自の入力ファイルが必要で，複雑な GUI 操作と多大な設計時間を要していた．本研究では，動作が保証された DDP の設計時間を削減し，シンプルな入力で設計できるように，設計自動化フローの検討を行った．Perl を用いて，回路設計ツールの GUI 処理を，Tcl スクリプトに変換し，CUI 上で動作させ，タイミング検証ツール [4] に入力する JSON 形式のパス情報を自動生成することで実現した．従来手法 [3] と提案手法で DDP を設計したところ，回路規模が 1.8%増加し，スループットは 2%低下したが，設計時間は 27%削減できた．しかし，マルチコア構成を考慮した場合の配置・配線案は従来手法 [3] の方が有効であった．

キーワード　データ駆動型プロセッサ，セルフタイム型パイプライン，FPGA

Abstract

A Study of FPGA Circuit Design Automation Flow for Data-Driven Processor

Kanji NAGANO

In recent years, various IoT(Internet of Things) devices have been utilized for diverse applications. In general, the microprocessor architecture of such devices should be required higher porwer performance efficiency as well as flexible functionality. DDP(Data-Driven Processor)[2] is an effective architecture for IoT systems because DDP is capable of multiple parallel processing of different data strems and autonomous power saving. Also, FPGAs are useful for IoT devices because of their flexibility and versatility in circuit design according to the various application. Therefore, the method of implementing DDP on FPGA to realize IoT devices is promising. However, existing FPGAs and circuit design tools are optimized for synchronous circuits, and it is difficult to optimize the design of DDP for asynchronous circuits.[6]. In the past methods to realize DDP on FPGAs[3], the operation of the circuit was not guaranteed enough. On the other hand, the research[4] is a method that can guarantee operation. However, it cleverly uses the functions of circuit design tools and requires its own input files, resulting in complicated GUI operations and a large amount of design time. In this study, we investigated a design automation flow to design a DDP with guaranteed operation using simple inputs and to reduce the design time. Using Perl, the GUI process of the circuit design tool was converted to a Tcl script and run on the CUI, and the path information in JSON format to be input to the timing verification tool[4] was

automatically generated. When we designed the DDP using the conventional method[3] and the proposed method, the circuit size increased by 1.8% and the throughput decreased by 2%, but the design time was reduced by 27%.However, the conventional method[3] was more effective in terms of placement and routing when considering multi-core configurations.

key words Data-Driven Processor, Self-Timed Pipeline, FPGA

目次

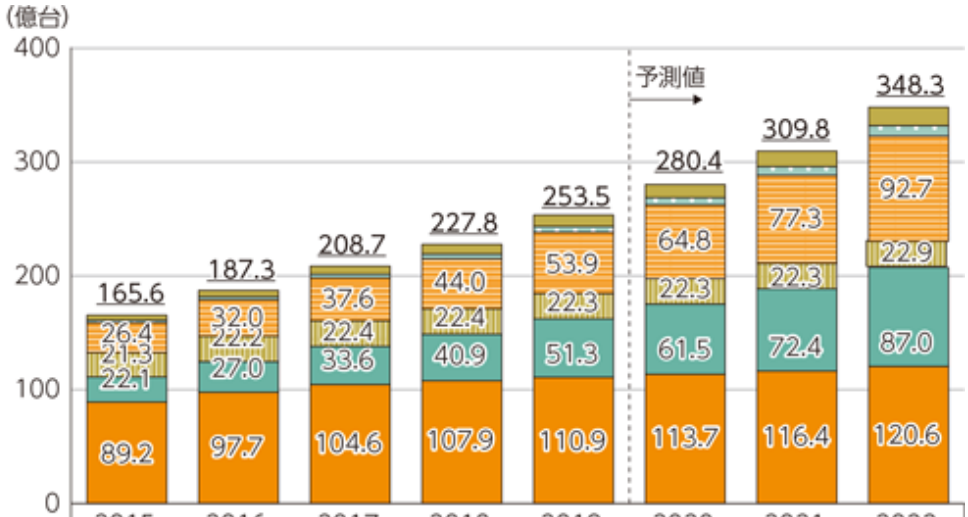
図目次

表目次

第 1 章

序論

近年，IoT（Internet of Things）技術の普及に伴って，IoT デバイスは図 1.1 で示すように，年々増加してきており，2022 年までには 348 億台にまで到達することが予測されている．



	2015	2016	2017	2018	2019	2020	2021	2022
合計	165.6	187.3	208.7	227.8	253.5	280.4	309.8	348.3
自動車・宇宙航空	4.5	5.7	7.1	8.6	9.9	11.7	13.7	16.1
医療	2.2	2.7	3.3	4.1	5.1	6.3	7.6	9.1
産業用途	26.4	32.0	37.6	44.0	53.9	64.8	77.3	92.7
コンピュータ	21.3	22.2	22.4	22.4	22.3	22.3	22.3	22.9
コンシューマ	22.1	27.0	33.6	40.9	51.3	61.5	72.4	87.0
通信	89.2	97.7	104.6	107.9	110.9	113.7	116.4	120.6

図 1.1　世界の IoT デバイス数の推移及び予測 (文献 [1] より引用)

IoT デバイスは，主にエッジ・コンピューティングやフォグ・コンピューティングのエッジ機器として用いられ，長時間稼働してセンサからデータを取得し，高速に処理することが

求められる．また図 1.1 から分かるように，医療，産業用途，自動車，宇宙航空など，多種多様な分野のシステムに応用することが期待されている．IoT デバイス用マイクロプロセッサは，スマートフォン等によく用いられる ARM や，カリフォルニア大学バークレイ校で開発された RISC-V 等が代表的である．これらは，グローバルなクロック信号で回路を制御するノイマン型の同期回路であり，複数のセンサからデータを取得した場合，割り込み処理等の前処理として処理中のデータの退避や処理を復帰させる命令の実行等が行われる．しかし，複数のセンサから頻繁にデータを取得する IoT デバイスにおいて，このような処理は総処理時間の増加や消費電力の増大を起こしてしまい問題となる．そのため，IoT デバイスの高性能化，低消費電力化，多様化に対する需要が高まっている．

データ駆動型プロセッサ（DDP:Data-Driven Processor）[2] は，低消費電力で動作し，異なるデータ流を多重に処理可能で高性能なため，IoT システムに有効なアーキテクチャである．また，FPGA は応用に合わせた柔軟な回路設計が可能で，多様性が求められる IoT エッジ機器に有効である．以上から，DDP を FPGA 上に実装して IoT エッジ機器を実現する方法が有望である．しかし，既存の FPGA と回路設計ツール（Intel 社 Quartus）は同期式回路に最適化され，非同期式回路の DDP を最適に設計することが困難である [6]．これまでに，商用の FPGA に非同期式回路を実現するために様々な研究が行われている [5][6][7][8]．特に，過去に取り組んだ研究 [3] では既存の回路設計ツール (Intel 社 Quartus) の機能を巧妙に使い，FPGA に手動で DDP を設計する手法を提案した．しかし，[3] では，回路の動作保証が十分ではなかった．それに対し，筑波大学の研究 [4] では，FPGA において DDP の動作を保証するためのシミュレーション方を提案しており，信頼性のある DDP の設計手法を確立している．これらの研究では，どれも既存の回路設計ツール（Intel 社 Quartus）の機能を巧妙に利用したり，安全に設計するための独自の入力ファイルを生成する必要があり，設計に多大な時間がかかってしまい，複雑な GUI 操作も行うため設計難易度が高い．

本研究では，動作が保証された DDP の設計にかかる時間を削減し，かつシンプルな入力ファイルで DDP の設計を行えるように，FPGA を対象とした DDP の設計自動化フローの検討を行う．

これ以降の本論文において，第 2 章では DDP のアーキテクチャと設計課題，設計課題に取り組まれている関連研究についてまとめ，DDP の設計を自動化する際の課題について述べる．

第 3 章では，DDP の設計自動化手法についてまとめる．先ず，DDP の設計を自動化するために改良したアーキテクチャ，次に，設計者が予め用意する必要のあるファイル，インストールする必要があるプログラミング言語など，DDP の設計を自動化する際の条件，最後に設計自動化フローの各ステップにおけるアルゴリズムについて詳細にまとめる．

第 4 章では，Intel 社製 FPGA ボード MAX10 DE10-LITE を対象として [3] の手法と今回の提案手法それぞれで DDP の設計を行い，設計時間やそれぞれの DDP の回路規模など性能の評価を行い，それらを比較して結果を考察して，提案手法の有効性について述べる．回路規模などの評価に用いる回路設計ツールは Intel 社製 Quartus Prime Standard Edition 18.0 を用いた．

第 5 章では，本研究のまとめと今後の課題について述べる．

第 2 章

DDP の設計自動化の課題

2.1 緒言

本章では，データ駆動型プロセッサ（DDP）で実現されている計算モデルのデータ駆動計算モデル，DDP のアーキテクチャ，FPGA（Field Programmable Gate Array）の要素技術について簡単にまとめ，DDP が既存の回路設計ツール（Quartus）での設計が困難であることを述べる．また，DDP の FPGA 設計における課題と，設計課題に取り組まれている関連研究についてまとめる．そして，DDP 設計を自動化するにあたって解決すべき点についてまとめる．

2.2 データ駆動計算モデル

DDP はデータ駆動計算モデルを実現したプロセッサで，このモデルは並列性を扱う一種であり，アルゴリズム自身が持つ本質的な並列性を表現できる．データ駆動計算モデルにおいて，プログラム中の各ノードは図 2.1 に示すように，その演算の実行に必要なパケットが揃うと実行可能になる．実行可能になったノードはパケットの組を入力として演算を実行し，演算結果を次のノードに転送する．このようにデータ駆動型計算モデルでは，各ノード間でデータの受渡しが行われるため，演算の実行順序はデータの受渡しの流れに依存する．その依存関係は図 2.2 のようなデータフローグラフで表され，データフローグラフ上で依存関係がないノードの演算は並列に実行することができる．

例えば，図 2.2 のような (A + B) - (C + D) の計算を考える．(A + B) の演算を行う

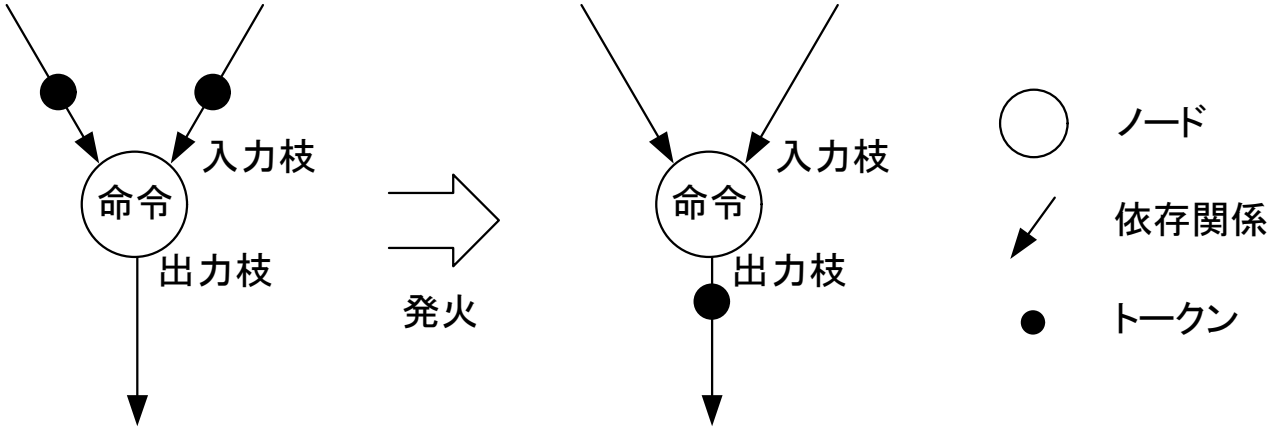


図 2.1　データ駆動計算モデルの動作原理

ノード (dest1) と (C + D) の計算を行うノード (dest2) はそれぞれデータ依存関係が無いため，並列に実行可能である．

2.3　DDP のアーキテクチャ

DDP はセルフタイム型パイプライン（STP: Self-Timed Pipeline）[2] で構成されている．STP は図 2.3 に示す様に，複数のデータ転送制御回路の C 素子（Coincidence flip-flop）で構成される．隣接するパイプラインステージ間の C 素子がデータ転送要求信号（Send 信号）とデータ転送許可信号（Ack 信号）でハンドシェイク通信することによって，データ・ラッチ解放信号（CP 信号）が立ち上がり，データ・ラッチ（DL: Data Latch）からロジック（logic）へデータが渡されて処理が行われる．これによって，隣接するパイプラインステージ間の回路のみ部分的に駆動するため，グローバル・クロックで回路全体が駆動する同期式回路と比べて，配線や電力消費を抑えられ，低消費電力で動作する特性を実現している．

STP は図 2.4 のタイミングチャートに基づいて動作している．CP 信号が立ち上がってから後段の CP 信号が立ち上がるまでの時間を Tf と呼び，Tf は前段の DL から logic を通り，後段の DL まで最長の経路（クリティカルパス）の遅延時間よりも長い必要があり（セットアップ時間制約），短いと logic の処理が終わらないままデータが取り込まれてしまうため，

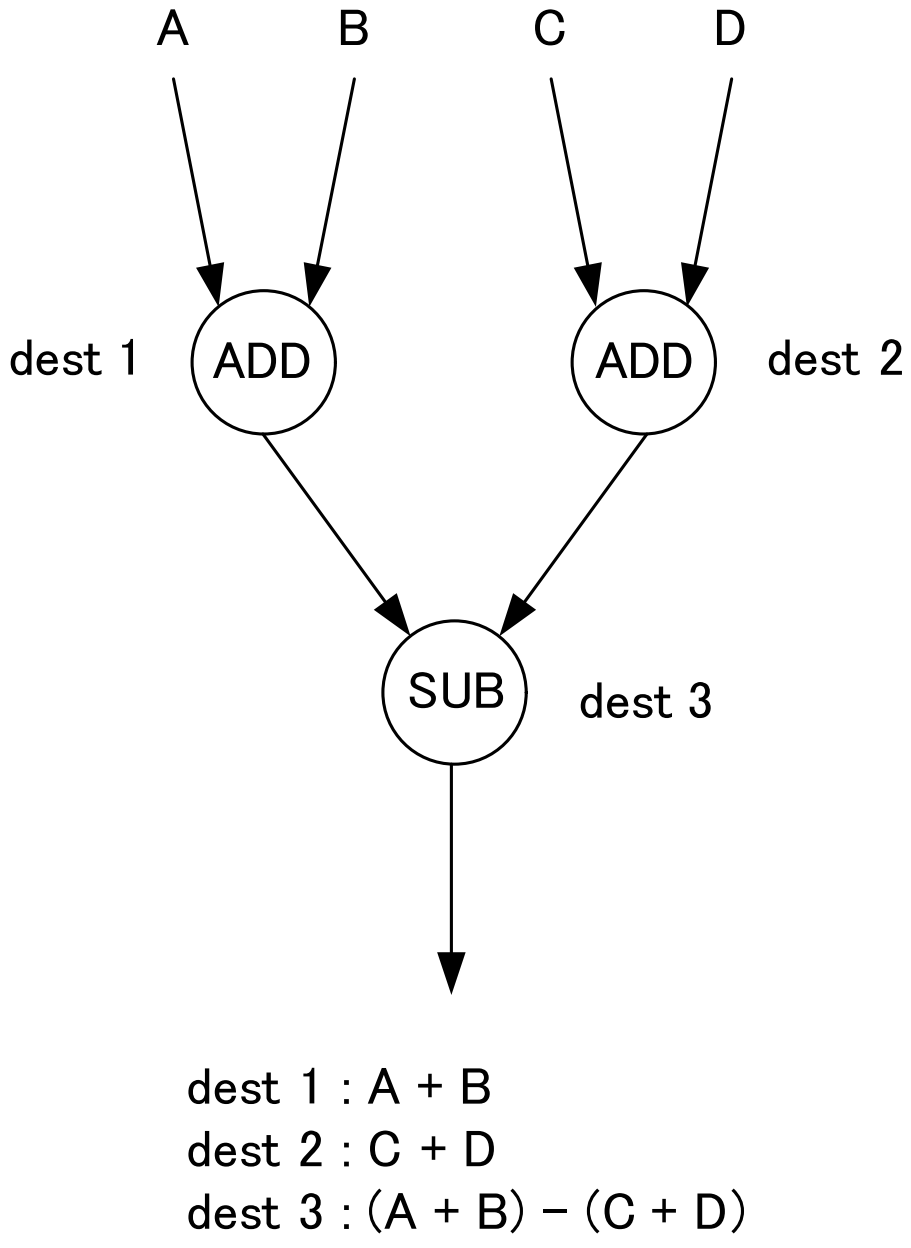


図 2.2　データフローグラフの例

正しいデータが得られない．そのために，図 2.3 の遅延回路 Ds を設計して，Tf の時間を調整する必要がある．

また，後段の CP 信号が立ち上がってから，もう一度全段の CP 信号が立ち上がるまでの時間を Tr と呼び，Tr は，後段の DL にデータが取り込まれてから安定するまでの時間よ

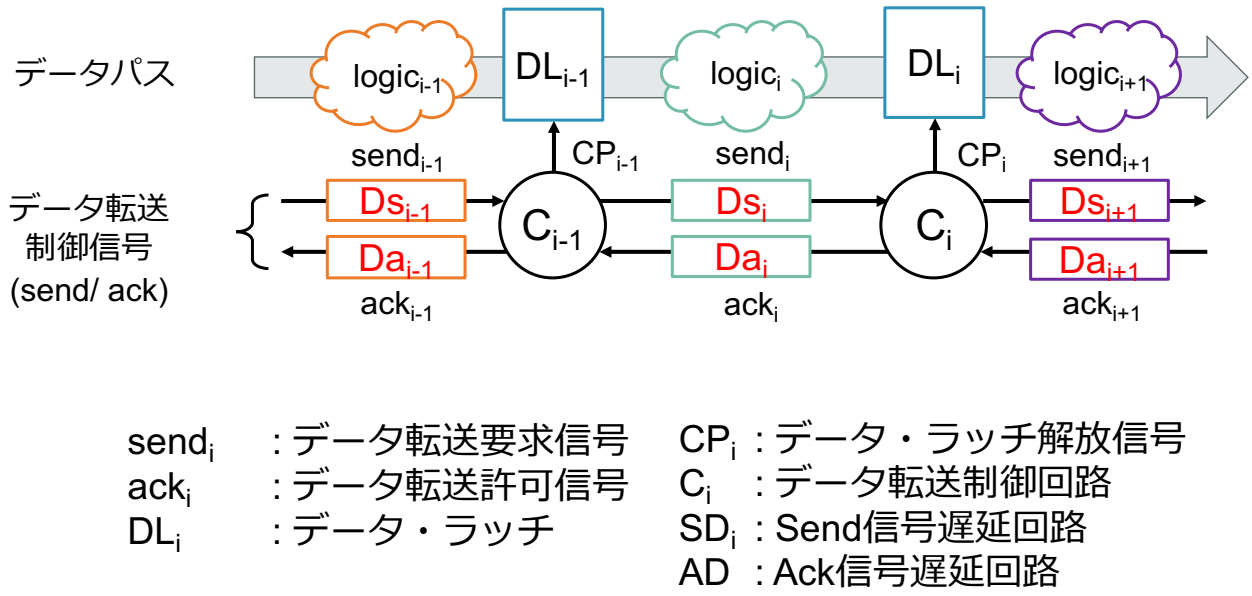


図 2.3　セルフタイム型パイプライン（STP）の構成

りも長い必要がある（ホールド時間制約）．そのために，図 2.3 の遅延回路 Da を設計して，Tr の時間を調整する必要がある．

DDP は図 2.5 に示す様に 9 つのパイプラインステージから構成され，それぞれの機能を以下に示す．DDP はパケット（図 2.6，表 2.1 が入力されパケットの持つ宛先情報（dest）から PS ステージで次の命令を読み出し各ステージを周回しながら処理が行われる．DDP の演算には，単項演算と多項演算があり，多項演算はペアとなるパケットが MMCAM で揃うと処理が実行されるため，アウトオブオーダー（Out-of-Order）実行が可能である．

- パケット合流機構（M : Merge）

  DDP はパケットが内部を周回することで，各ステージで処理が施されるアーキテクチャのため，外部からの入力パケットと内部から周回してきたパケット合流する部分があり，それがパケット合流機構に当たる．それらのパケットを調停し，先に到着した方のパケットから CST へパケットを出力する．

- 定数読み出し機構（CST : Constant Memory）

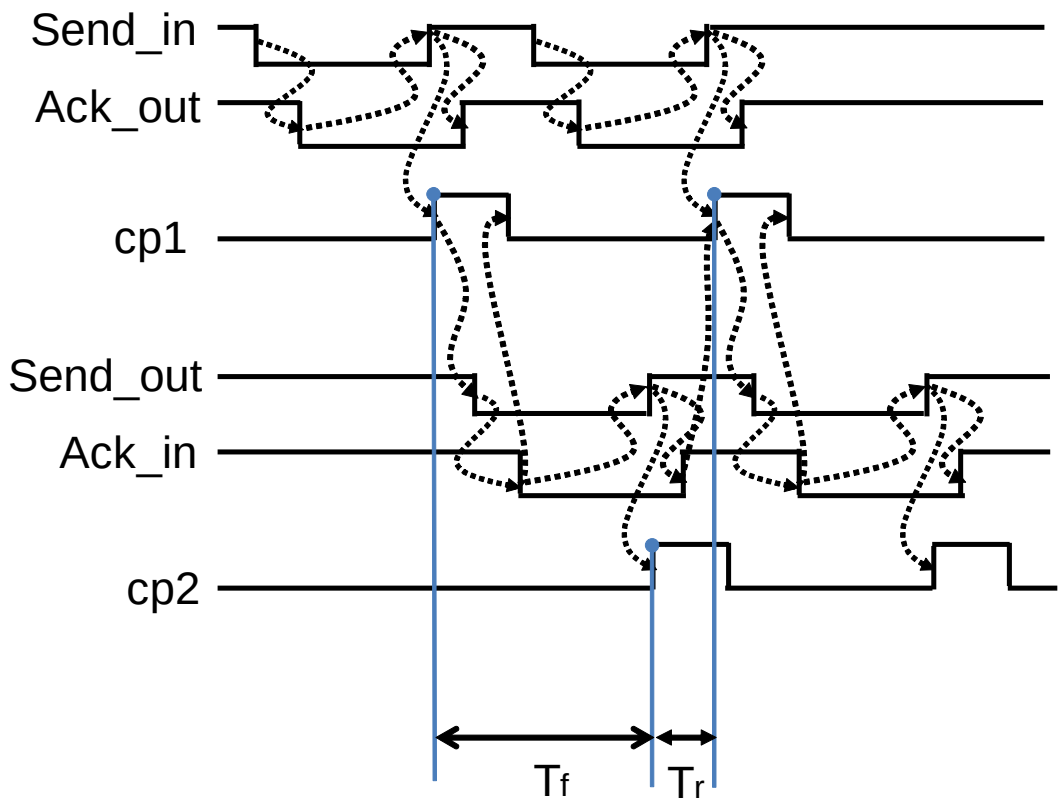


図 2.4　セルフタイム型パイプライン（STP）のタイミングチャート

定数演算を実行する場合，定数を格納しておく領域が必要となるため，定数読み出し機構内のメモリに，演算に用いる定数とオペレーションコードが格納されている．転送されてきたパケットに付加されている宛先情報（dest）から，対応する定数とオペレーションコードを読み出し，パケットに付加して MMCAM に出力する．

- パケット待ち合わせ機構（MMCAM : Matching Memory CAM）

  CST からパケットを受信し，コンテンツ・アドレッシング・メモリ（CAM) に格納されているペアのパケット検索に必要な情報を一時的に格納する．入力パケットに対応するペアのパケットが検索された場合，MMRAM で用いるアドレスを生成し，入力パケットと共に MMRAM に出力する．定数演算命令などペアのパケットが必要ない場合，入力パケットがそのまま MMRAM へ出力される．

2.3 DDP のアーキテクチャ

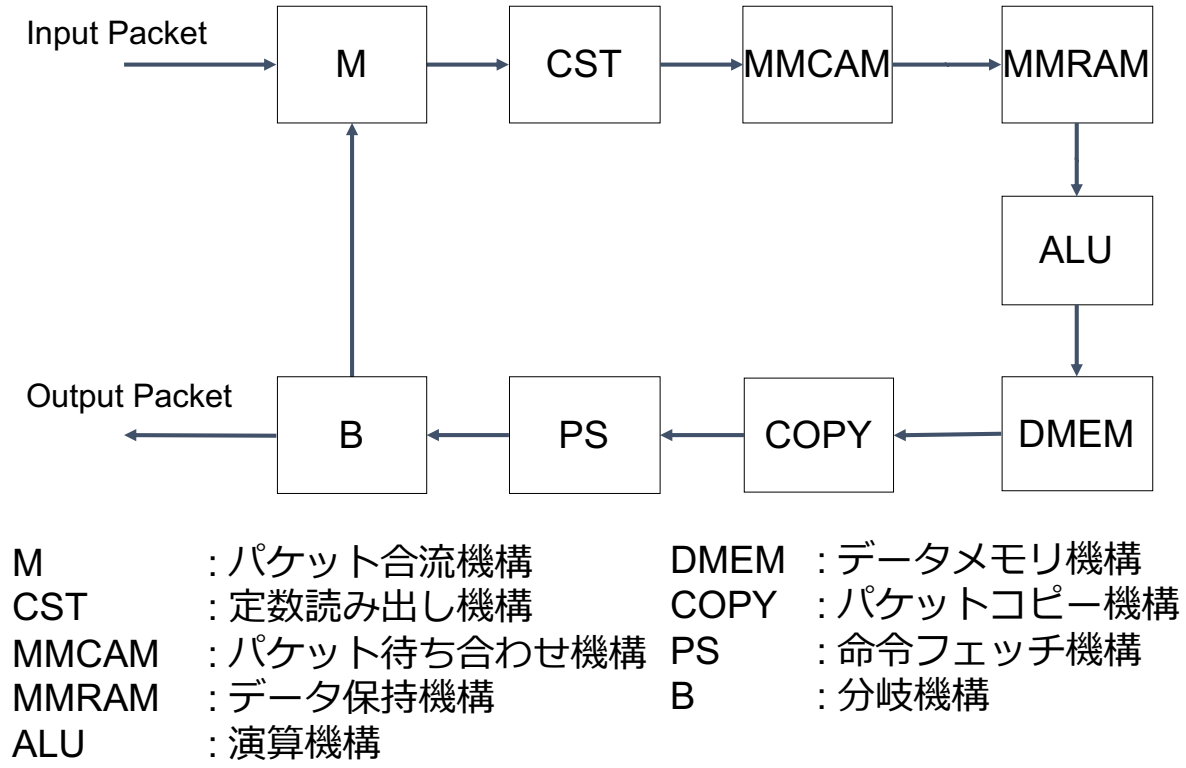


図 2.5 データ駆動型プロセッサ（DDP）の構成

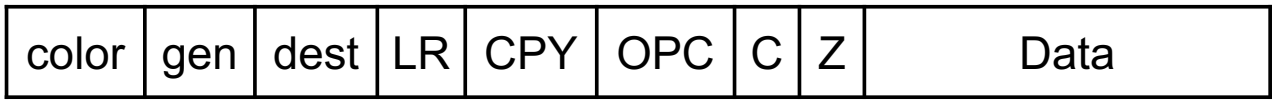


図 2.6 DDP の入力パケットフォーマット

- データ保持機構（MMRAM : Matching Memory RAM）

  MMCAM でペアのパケットが検索されたパケットは，対応するオペランドデータを MMRAM 内のメモリから読み出し，入力パケットに付加される．一方，MMCAM でペアが検索されず，ペアのパケットを待つ必要のあるパケットはメモリに保存される．定数演算命令など，ペアのパケットが必要ない場合，入力パケットがそのまま ALU へ出力される．

- 演算機構（ALU : Arithmetic Logic Unit）

表 2.1 パケットが保持する情報

フィールド名	情報	ビット数
color	パケットの識別	3
gen	パケットの世代	8
dest	パケットの宛先	7
LR	パケットの左右識別	1
CPY	コピーフラグ	1
OPC	命令	3
C	キャリーフラグ	1
Z	ゼロフラグ	1
Data	オペランドデータ	16

演算機構では，受診したパケットの保持するオペレーション・コードに従って，算術演算，論理演算などの演算処理を行う．ロード命令，ストア命令の場合は，DMEM で用いるアドレスを算出する．演算結果をキャリーフラグ（C）やゼロフラグ（Z）と共にパケットに書き込み，DMEM へ出力する．

- データメモリ機構（DMEM : Data Memory）

  ロード/ストア命令が実行されると，入力パケットに付加されているアドレスを用いて，データの読み出し/書き込みを行う．ロード命令で，読み出したデータをパケットに付加して COPY へ出力する．

- パケット・コピー機構（COPY : Copy Unit）

  入力パケットに付加されいるコピーフラグが有効であれば，パケットを複製する．有効でなければ，入力パケットをそのまま PS へ出力する．

- 命令フェッチ機構（PS : Program Storage）

実行するプログラムが PS 内のメモリに格納されており，入力パケットの宛先ノード番号を参照して次の宛先ノード番号，オペレーションコードを読み出し，入力パケット内の情報と書き換え，B に出力する．また，不要となったパケット削除する機能も備えており，削除命令の場合パケットを削除する．

- パケット分岐機構（B : Branch Unit）

  パケットの持つ情報から，出力先が M，もしくは外部なのかを判別し，対応した方へパケットを出力する．次に実行するプログラムがある場合，M へ出力する．それ以上実行するプログラムがない場合，外部へ出力する．

2.4 FPGA

FPGA は，ハードウェア記述言語を用いて自由に回路の書き換えが可能なデバイスである．FPGA が自由に回路の書き換えができる原理で，重要な役割を果たすのが LE（Logic Element）である．LE は，LUT（Look Up Table）と FF（Flip Flop）から構成される．また，LE は一定の数でまとまってブロック（LAB : Logic Array Block）となっている．図 2.7 の場合は 16 個の LE が 1 つの LAB を構成する．LUT は 4 入力 1 出力の回路となっており真理値表で表すことができ，その情報が保存されている．ハードウェア記述言語（Verilog HDL 等）で命令を記述すると，LUT の真理値表の情報が書き換わり、様々な論理として扱うことができる．FPGA 上には LE と，演算に特化した DSP（Degital Signal Processor）Block と，大量のデータの読み回帰が可能である Memory Block が並べられており，それぞれに配線が施されているため，互いに自由に接続が可能である．そのため，ハードウェア記述言語（Verilog HDL 等）で論理を書き換え，回路設計ツールの配置・配線によって接続することで柔軟に回路を設計することが可能となる．

また，FPGA 上に回路の配置・配線を行うには，Intel 社の FPGA 設計用ツール Quartus[9] が用いられる．回路の合成を行うとツールによって自動的にプログラムに合わせて LE が割り当てられるが，LE 間の通信時間は，LE 間の距離が離れているほど遅延が

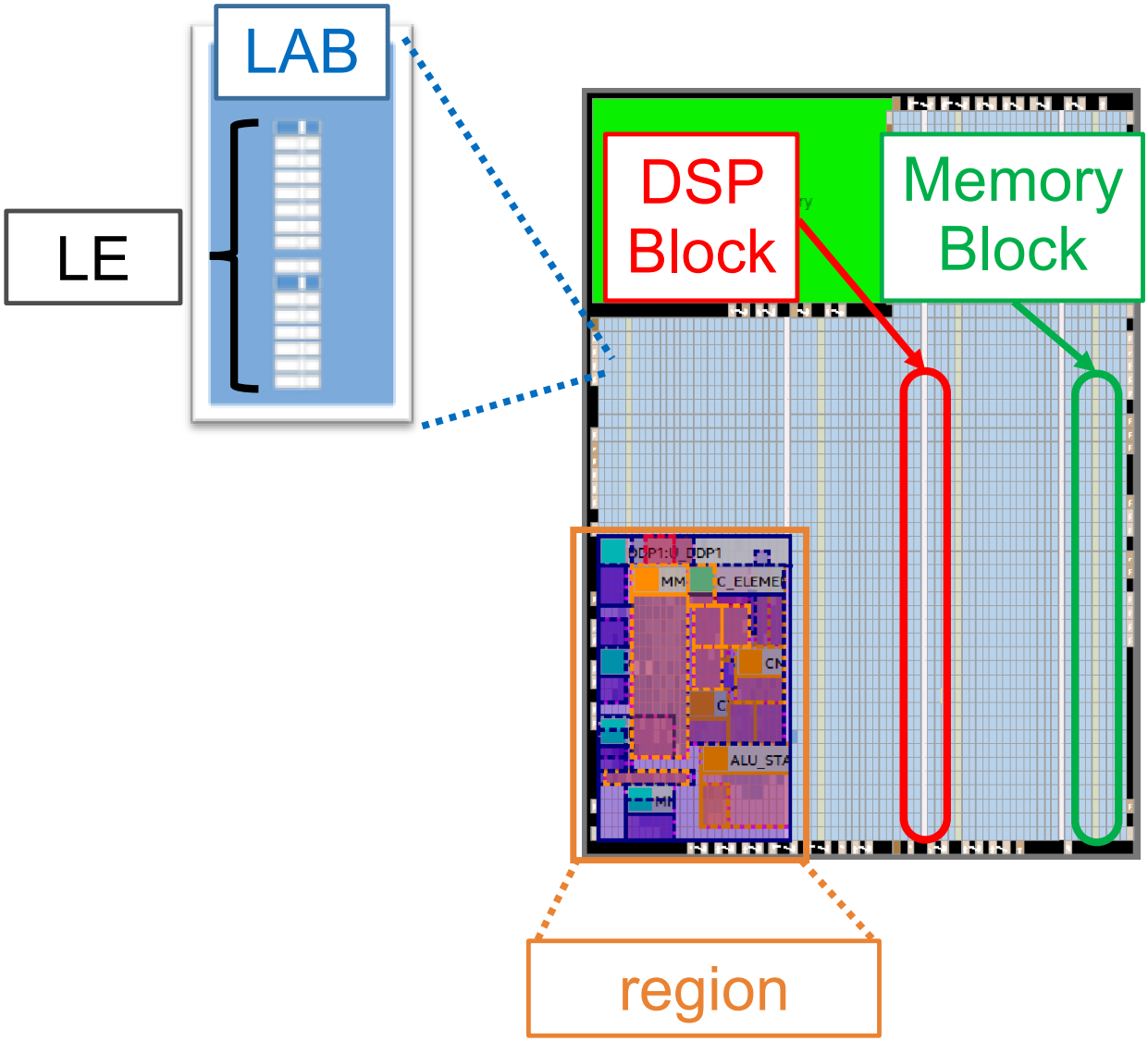


図 2.7　FPGA の構造（Intel 社 MAX10 の場合）

生じてしまう．そのため，ユーザが図 2.7 の様に，意図的に LE の配置を変更することによって遅延を削減することも可能である．

2.5　DDP の設計課題

一般的な同期式回路の設計フローは図 2.8 のようになっている．しかし，DDP は非同期式回路のため，Quartus の機能（特にタイミング解析・検証のステップ）をそのまま用いる

ことができず，動作を保証した DDP の設計が困難である．そのため，Quartus の機能を同期式回路と同様に用いることができるように．DDP に適した設計手法と設計フローが必要である．

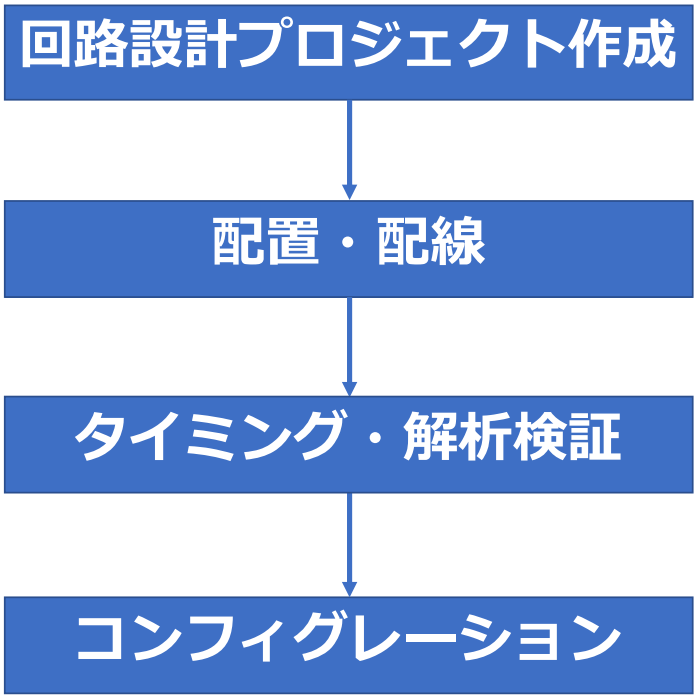


図 2.8　一般的な同期回路の設計フロー

2.5.1　タイミング制約を満たす最適な遅延回路の設計

DDP のアーキテクチャでも述べたように，STP ベースの DDP が正しく動作するには遅延回路の Ds と Da を設計することで，Tf と Tr の時間を調整する必要がある．ただし，単純に遅延回路の遅延量を過剰に増やすだけでは，STP の動作が遅くなりスループットが低下してしまう．そのため，各ステージのクリティカルパスに合わせて，最適な遅延回路を設計することで DDP のスループットを最大限に向上できる．最適な遅延回路を設計するには logic のクリティカルパスの経路や，Tf と Tr のタイミング情報を正確に求める必要があり，その方法が [4] の研究で確立されている．

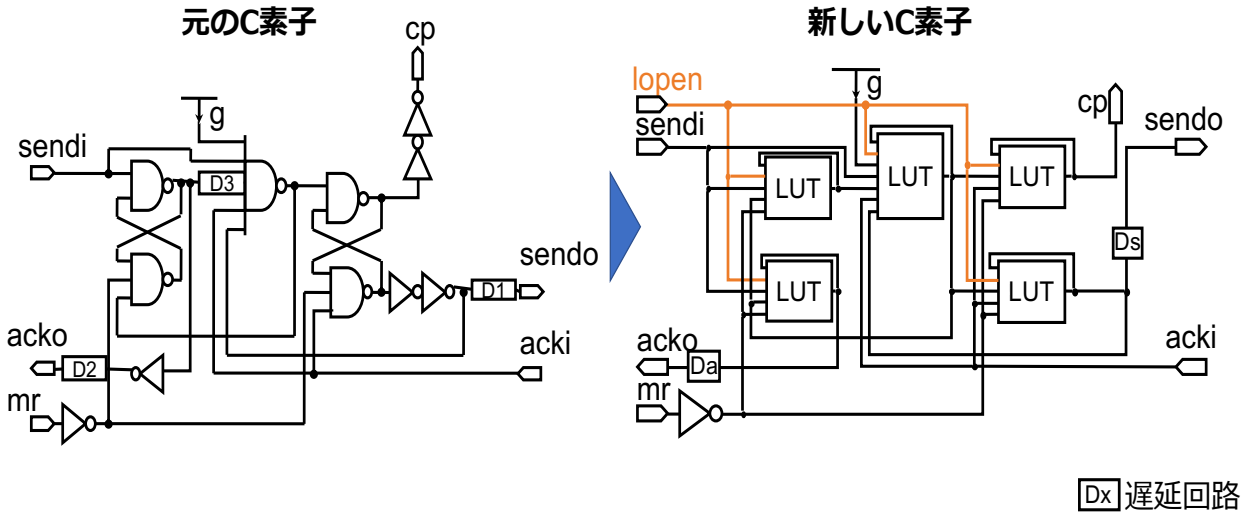


図 2.9　元の C 素子（左）と新しい C 素子（CCORE）[4]（右）

C 素子は元々図 2.9 の左側のアーキテクチャの様に組み合わせ回路で構成されていて，既存の回路設計ツール（Quartus）では，タイミング情報を抽出することができず，Tf と Tr がタイミング制約を満たしているかどうか確認できなかった．[4] では図 2.9 の右側のアーキテクチャの C 素子（以降 CCORE と呼ぶ）を用いることで，この問題を解決している．LUT は RS フリップフロップの役割を持っており，元の NAND ゲートの上下それぞれの機能を 1 つで実現している．さらに，回路記述に本来不要なラッチ（lopen）を追加することで回路設計ツールにレジスタとして認識させている．これにより LUT 間のタイミング情報を抽出できるため，Tf と Tr のタイミング情報は図 2.10 の経路で抽出できる．また，CP 信号を出力している LUT（図 2.10 の LUT3 と LUT8）に SDC（設計制約）ファイルでクロック特性を持たせることで logic のクリティカルパスのタイミング情報を抽出できる．Quartus のタイミング解析機能によって，タイミング情報を抽出し，JSON 形式のモジュール間パス情報と併せて，独自のタイミング検証ツールへ入力することで，タイミング制約を満たしているかどうか確認できる．これによって，最適な遅延回路を設計できる．回路合成時に Quartus のパーティションと呼ばれる，回路の最適化を防止するための機能を，全ての回路モジュールに設定する必要がある．

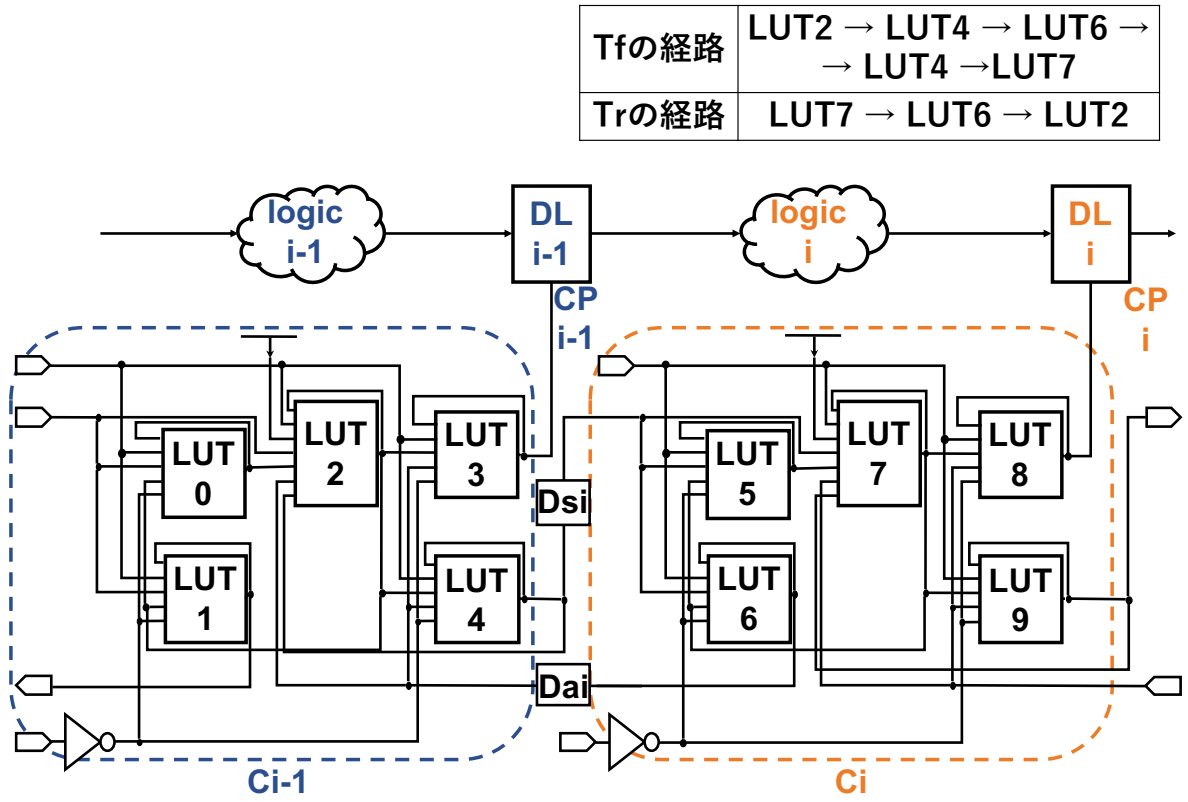


図 2.10　Tf と Tr の経路 [4]

しかし，この手法が適用できるのは通常の C 素子で，DDP には通常の C 素子から派生させたタイミング制御回路が複数含まれているため，それらに応用させる方法を考える必要がある．

2.5.2　配置・配線

FPGA は配置・配線による遅延の影響が ASIC と比べて大きいため，回路を凝集して配置・配線することで，配線遅延を削減することが望ましい．また，IoT 向けエッジ機器は，マルチコア構成に拡張可能なように設計にするため，リロケータブルな配置・配線案が必要である．

[3] の研究では，Quartus の Region の機能を使用することで DDP 全体の配置・配線される領域の面積の削減，また隣接するステージを密接して配置・配線することで，配線遅延の

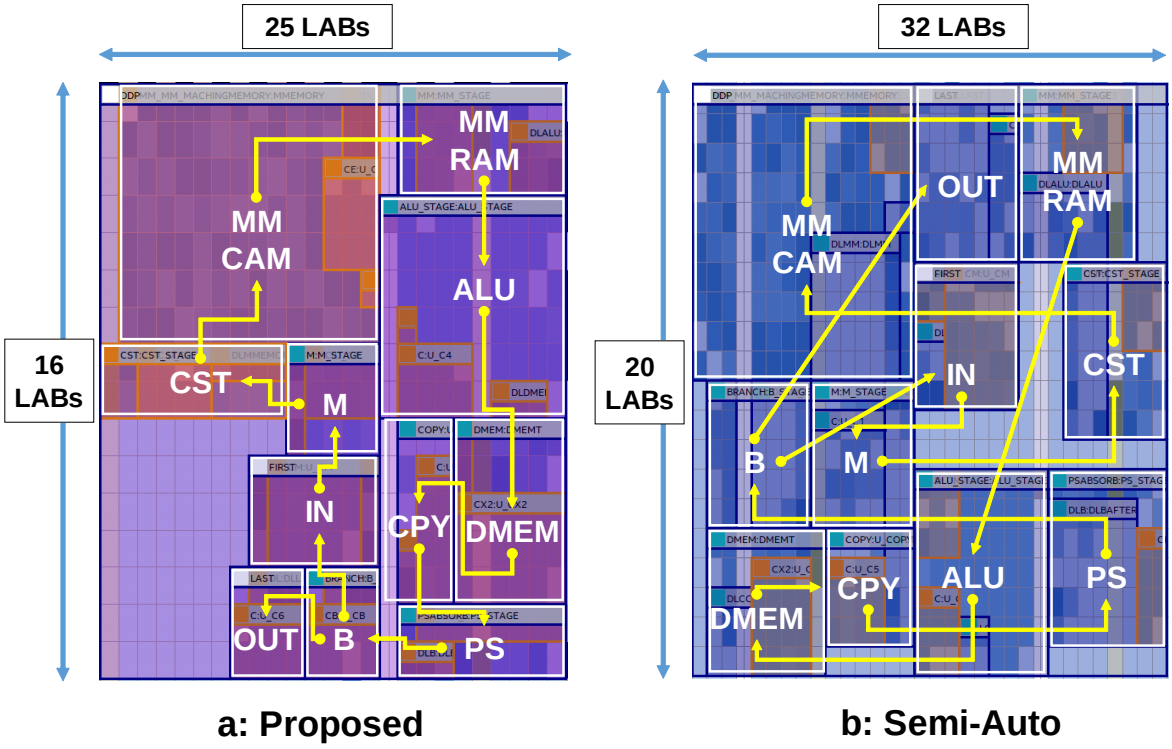


図 2.11　Quartus の Region 機能を用いた面積削減 [3]

削減を検討した．その結果，図 2.11 のように面積の削減に成功した．しかし，DDP の動作保証が不十分であった．

2.6　設計自動化の課題

以上から DDP を設計するフローは図 2.12 のようになる．しかし，この設計作業を全て手作業で行うと多大な時間がかかることが見込まれる．特にパーティション設定や，タイミング解析・検証のための JSON 形式のモジュール間パス情報などは，回路規模が大きくなるにつれて作業が多くなり，時間がかかり，またミスも多くなることが予想される．そのため，設計を自動化することが望まれる．

DDP の設計を自動化するためには，設計作業を CUI 上で実行する必要がある．そのためには，Quartus の GUI 作業に対応した Tcl スクリプトや，タイミング検証ツール [4] で使

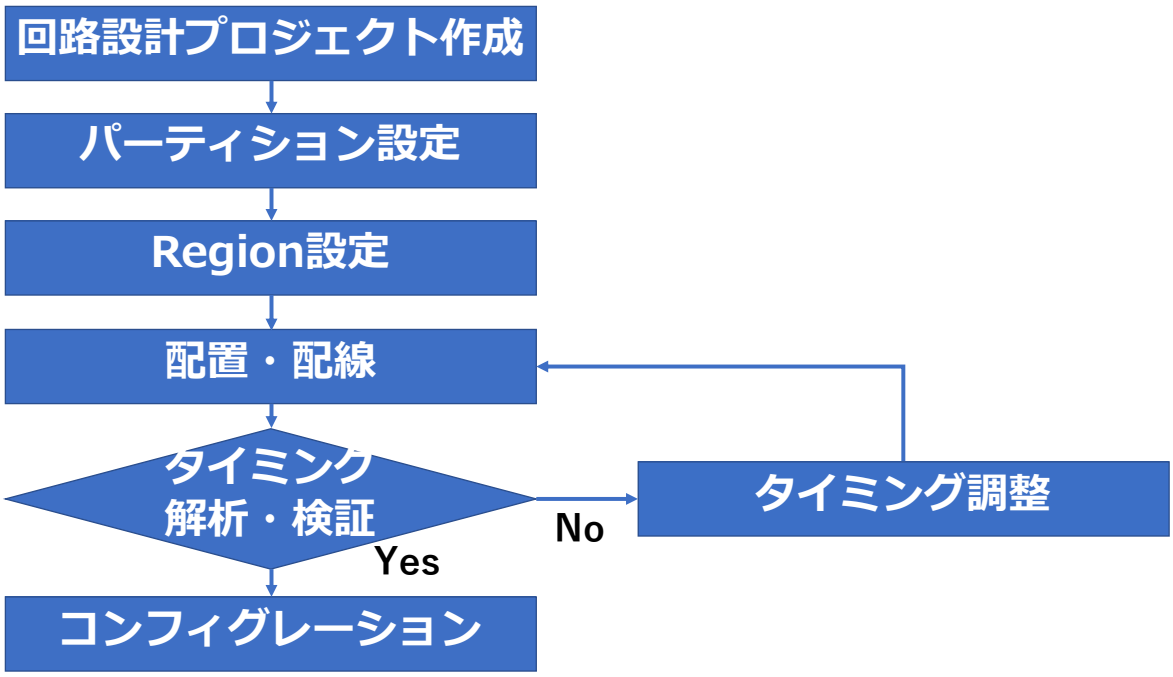


図 2.12　DDP の設計フロー

用する JSON ファイルなどを自動生成する必要がある．

2.7　結言

本章では，まず DDP のアーキテクチャについてまとめ，セルフタイムパイプライン（STP）で構成されているため低消費電力動作し，またプログラムの順番に関係なく，データの到着順で処理を実行し，複数の異なるデータ流を多重処理可能であるため高性能であることを述べ，FPGA の構成について簡単にまとめた．次に，既存の回路設計ツール（Quartus）では，非同期式回路の DDP のタイミング解析・検証を行うことが困難なため，信頼性のある設計をするには，DDP 独自の設計フローが必要であることを述べた．また，FPGA における STP のタイミング解析・検証方法を確立している研究 [4] についてまとめ，DDP に応用するには，C 素子以外のデータ転送制御回路のアーキテクチャを変更しなければならないことを述べた．最後に，DDP の設計を自動化するためには，Quartus の GUI 操作に対応する Tcl スクリプトや，タイミング解析・検証に用いられる JSON 形式のパス情

報などを，自動で生成する必要があると主張した．

第 3 章

DDP の設計自動化手法

3.1　緒言

本章では，まず DDP の設計自動化手法の前に，[4] の技術を用いて，動作を保証した DDP を設計するためにデータ転送制御回路の変更したアーキテクチャについてまとめる．その後に，Perl を用いて Tcl スクリプトやタイミング検証ツールに用いる JSON 形式のパス情報などを自動生成して，DDP の設計を自動化する手法についてまとめる．DDP 設計自動化の実行には，各設計ステップに入力するファイルや，インストールが必要なプログラミング言語や CUI などの条件があるため，それらについてまとめる．

3.2　データ転送制御回路のアーキテクチャの変更

タイミング検証ツール [4] を用いるために，データ転送制御回路（C 素子）のアーキテクチャを，2 章で示した LUT を用いたアーキテクチャに変更する必要がある．しかし，DDP には，図 3.1 に示すように，通常の C 素子だけではなく，C 素子から派生させたデータ転送制御回路（CM，CE，CX2，CB）を含んでいるため，これらのアーキテクチャの変更も必要である．

3.2.1　CM

データ転送制御回路（CM）は，2 つの経路から転送されてきたパケット（図 3.1 の外部からの入力パケットと B からの周回パケット）を調停し，先に到着した方のパケットから

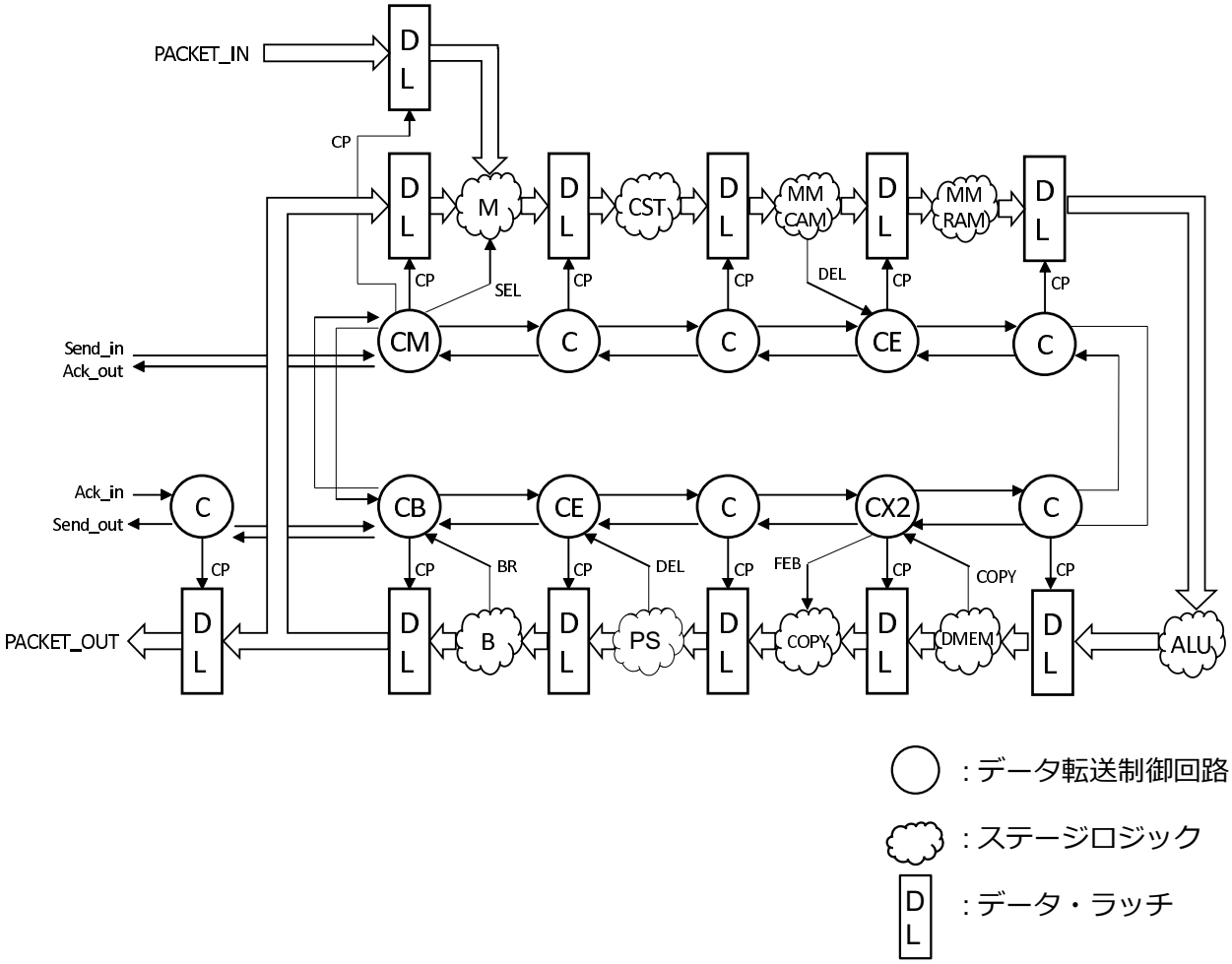


図 3.1　DDP 全体のアーキテクチャ

順番に次のステージへ転送するための回路である．元の CM のアーキテクチャは図 3.2 のようになっており，主に 2 つの CJ と呼ばれる回路と ARB と呼ばれる調停回路から構成される．CJ のアーキテクチャは図 3.3 の左側のようになっており，通常の C 素子との相違点は ARB からの入力（z と ga 図 3.3 の）があることと，出力 (sendo）が not で反転していること（図 3.3 の）である．そこで，CJ のアーキテクチャを図 3.3 の右側のように変更し，これを新たに CJCORE と名付ける．そして，図 3.4 のように，CM を CJCORE を用いたアーキテクチャに変更し，CJCORE の出力（sendo）を not で反転させる必要があるため，not 回路を新たに追加した．

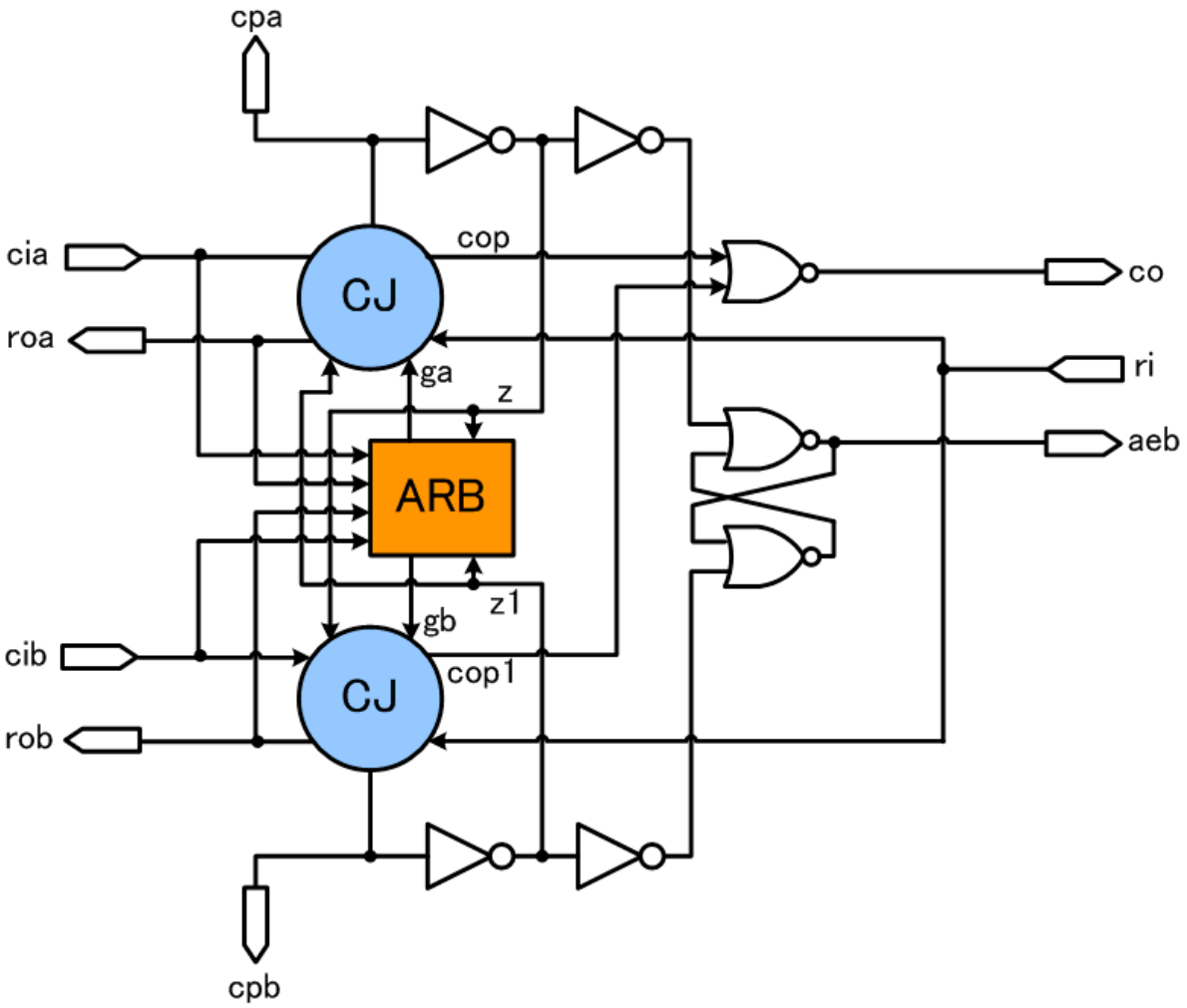


図 3.2　元の CM

3.2.2　CE

データ転送制御回路（CE）は，転送されてきたパケットを消去する機能を備えた回路である．待ち合わせが必要なパケットを次のステージへ送らない場合や，消去命令のパケットが流れてきた場合に，消去機能が働く．元の CE のアーキテクチャは図 3.5 のようになっており，CF と呼ばれる回路と，その他複数の論理回路から構成される．CF のアーキテクチャは通常の C 素子とアーキテクチャと同じであるため，CF は新しい C 素子（CCORE）で置き換え可能である．そのため，CE のアーキテクチャを図 3.10 のように，CF を CCORE に置き換えたアーキテクチャに変更した．

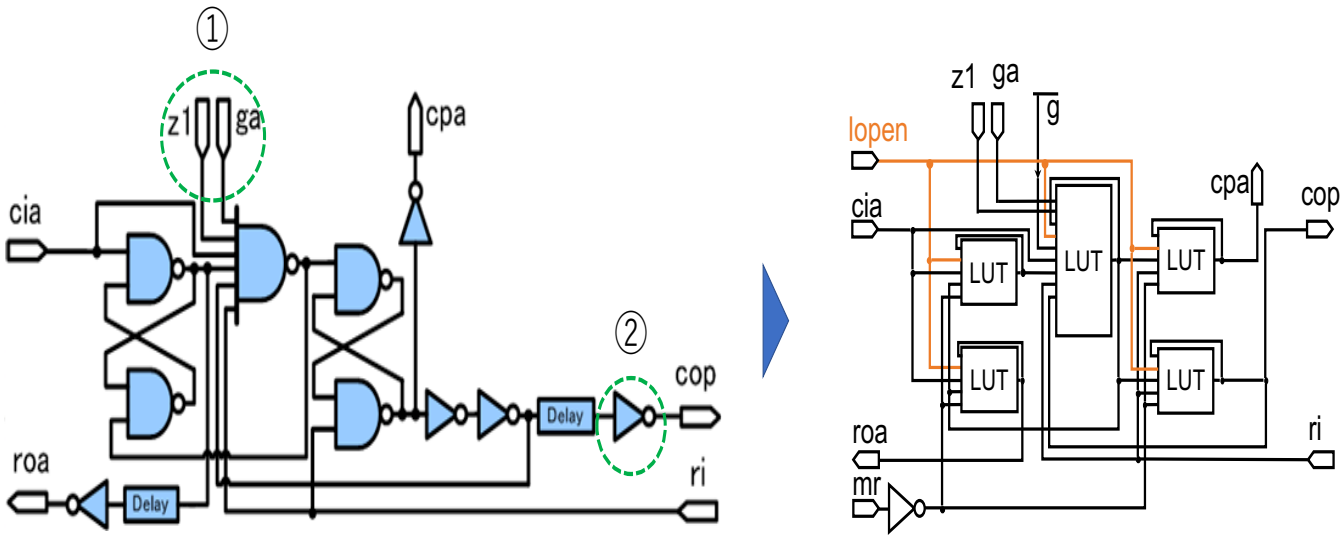


図 3.3　元の CJ（左）と新しい CJCORE（右）

3.2.3　CX2

データ転送制御回路（CX2）は，転送されてきたパケットを消去する機能とコピーする機能備えた回路である．コピーフラグが有効なパケットが流れてきた場合に，コピー機能が働く．元の CX2 のアーキテクチャは図 3.7 のようになっており，CE と同様で CF と呼ばれる回路と，その他複数の論理回路から構成される．そのため，CX2 のアーキテクチャを図 3.8 のように，CF を CCORE に置き換えたアーキテクチャに変更した．

3.2.4　CB

データ転送制御回路（CB）は，パケットの分岐を制御する回路である．転送されてきたパケットに付加されているブランチフラグを参照してパケットを M ステージへ転送するか，外部へ転送するか決定する．元の CB のアーキテクチャは図 3.9 のようになっており，CE と同様で CF と呼ばれる回路とその他複数の論理回路から構成される．そのため，CB のアーキテクチャを図**??**のように，CF を CCORE に置き換えたアーキテクチャに変更した．

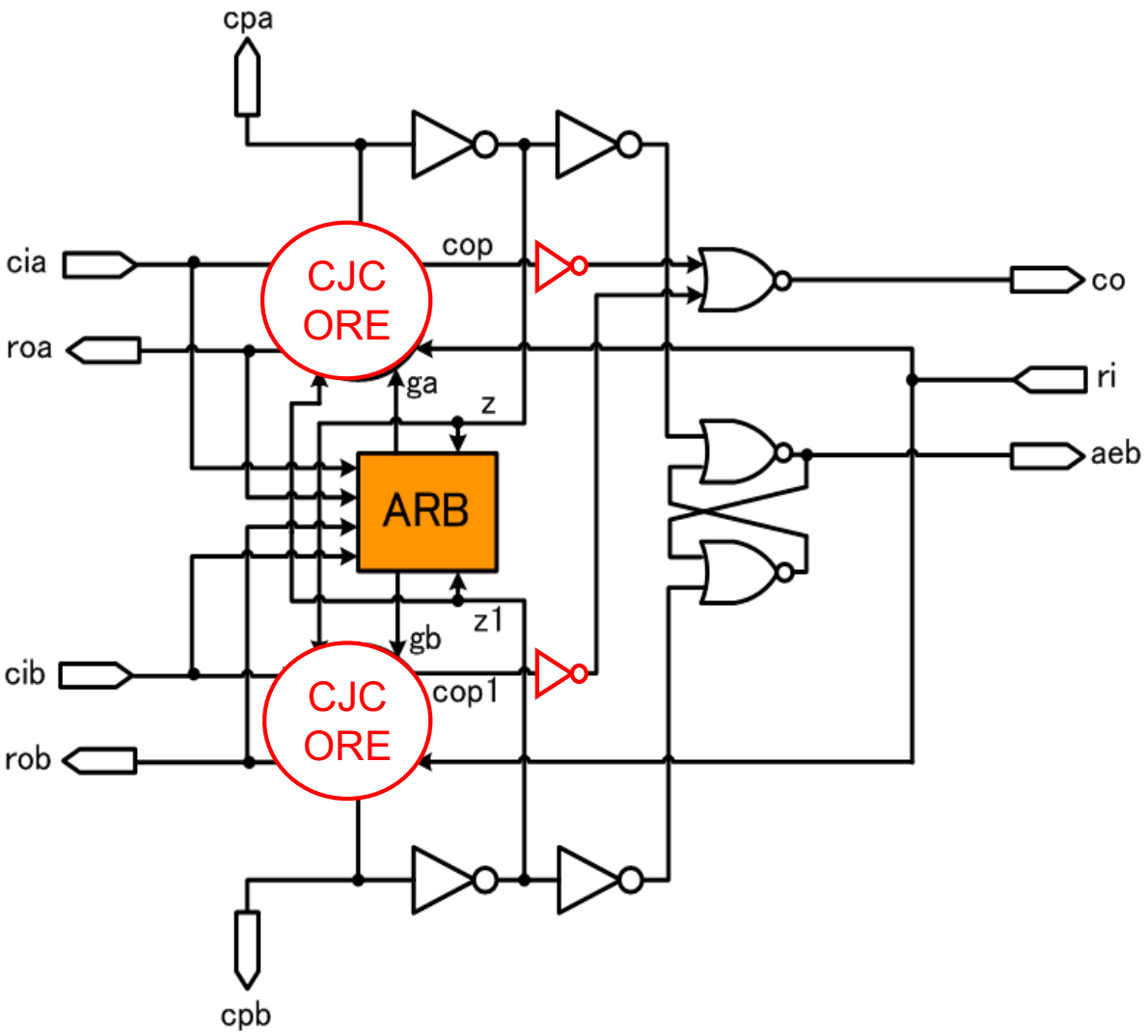


図 3.4　新しい CM

3.3　DDP 設計自動化の条件

DDP 設計を自動化するにあたって，満たすべき条件についてまとめる．

3.3.1　実行環境

DDP の自動設計を実行する際に必要な環境を，表 3.1 にまとめる．

CUI の NiosII Command Shell は回路設計ツールの Quartus をインストールすると同時

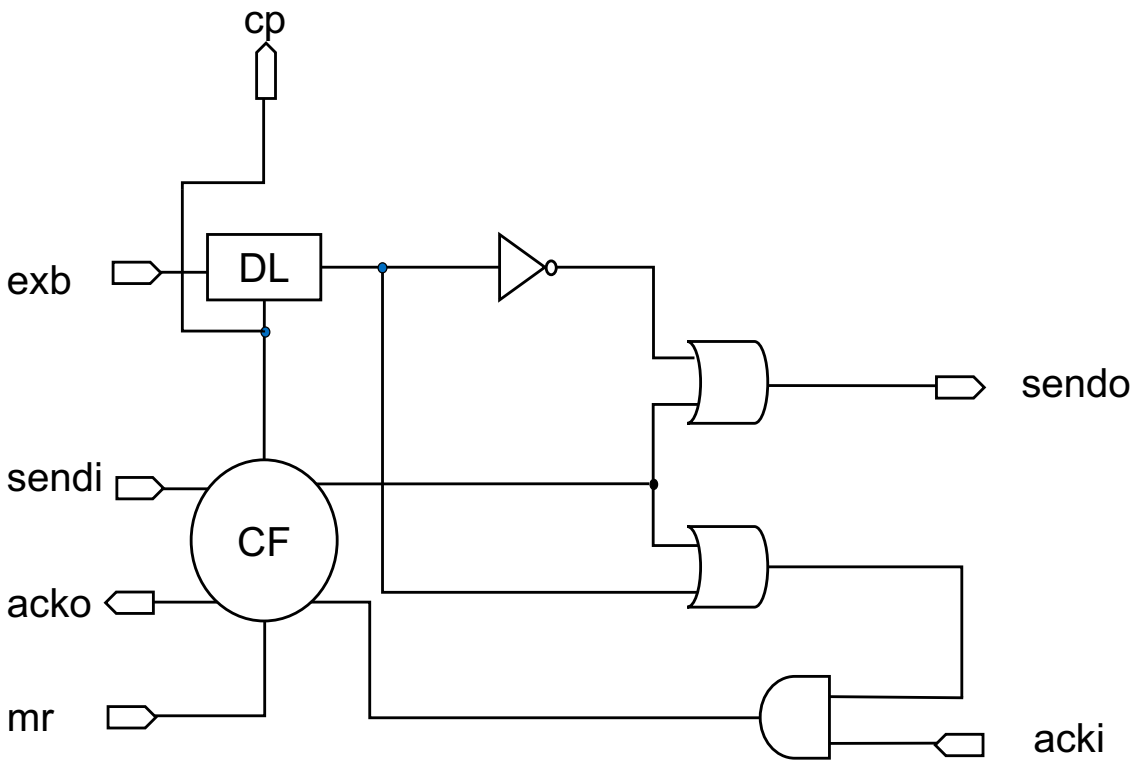


図 3.5　元の CE

表 3.1　DDP 自動設計の実行環境

OS	Windows10
プログラミング言語	Perl, Python
CUI	NiosII Command Shell

に，自動でインストールされる．

3.3.2　用意するファイル

設計自動化にあたり，用意する必要があるファイルを以下にまとめる．

- 設計対象 FPGA のデフォルトの qsf（回路設計プロジェクト設定）ファイル

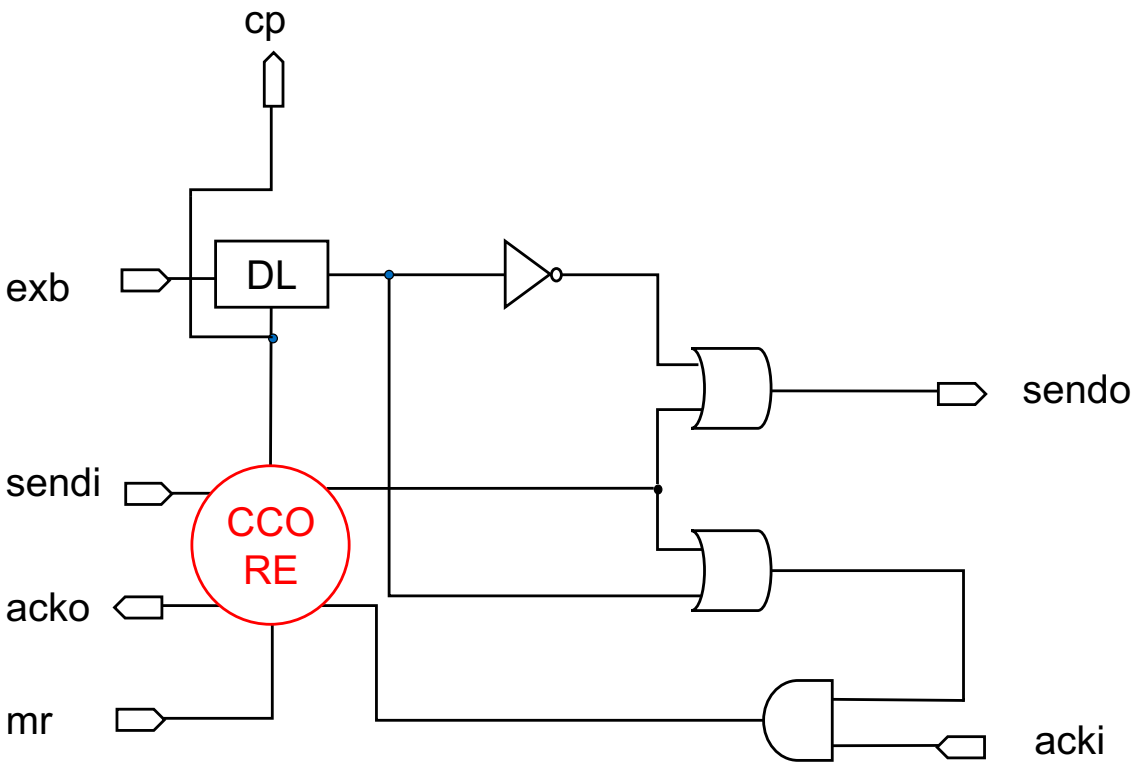


図 3.6　新しい CE

qsf ファイルとは，回路設計プロジェクト内で設定されている情報をまとめているファイルである．FPGA デバイス名や，FPGA チップ周りの入出力信号の接続情報などの情報が含まれており，対象の FPGA の回路設計プロジェクトを作成する際に用いられる．

- 設計対象 FPGA のデフォルトの SDC（設計制約）ファイル

  SDC（設計制約）ファイルとは，Quartus が回路を合成する際の制約情報がまとめられているファイルである．qsf ファイル内でプロジェクトに用いられる SDC ファイル名が指定されているので，その名前で SDC ファイルを用意する．

- 設計対象 FPGA のデフォルトのトップモジュールの Verilog ソースコード

  デフォルトのトップモジュールの Verilog ソースコードとは，対象の FPGA チップ周

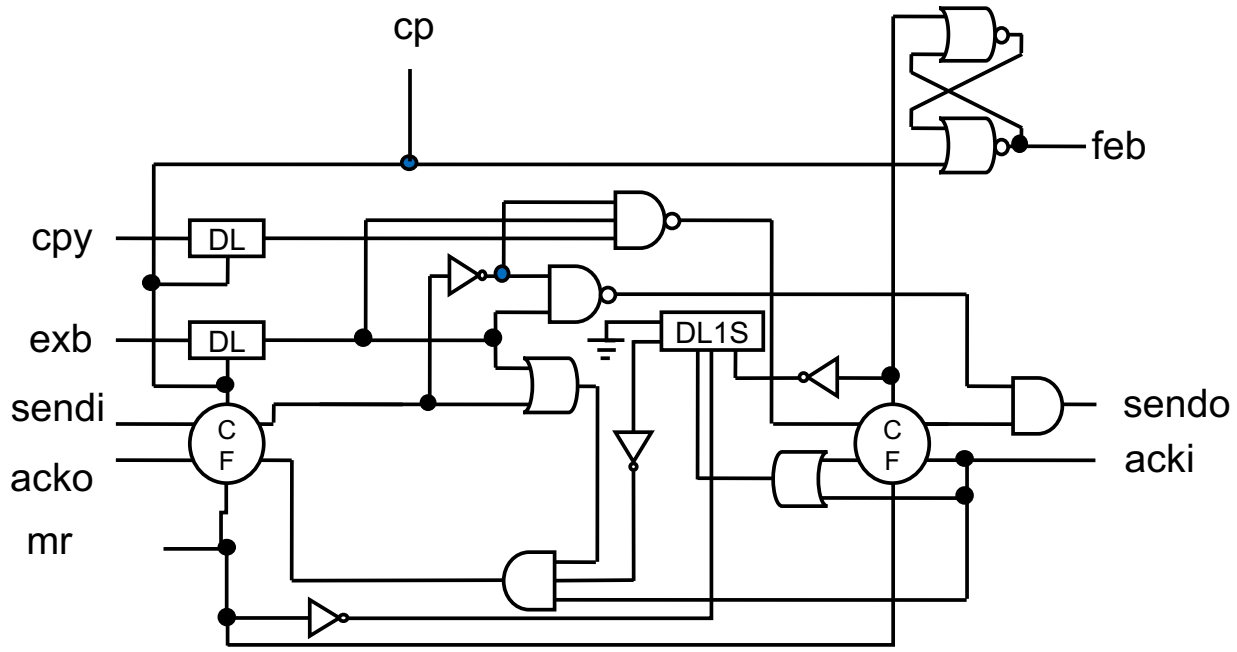


図 3.7 元の CX2

りの入出力信号名が宣言されている Verilog ソースコードのことである．デフォルトの qsf ファイルで回路設計プロジェクトの最上位モジュールに指定されており，用意しなければ回路が合成できない．また，トップモジュールのソースコード内に DDP のモジュールをインスタンス化しておく．

- DDP の Verilog ソースコード

  RTL シミュレーション（論理シミュレーション）が済んでいる状態の，DDP の Verilog ソースコードを用意する．

- メモリの初期化ファイル

  DDP のステージ（CST，MMRAM，DMEM，PS）で使用されるメモリを初期化するファイルである．Quartus で作成できるため，予め用意しておく．

- タイミング検証ツールプログラム [4]

  [4] で提案された，DDP 用のタイミング検証ツールプログラム．

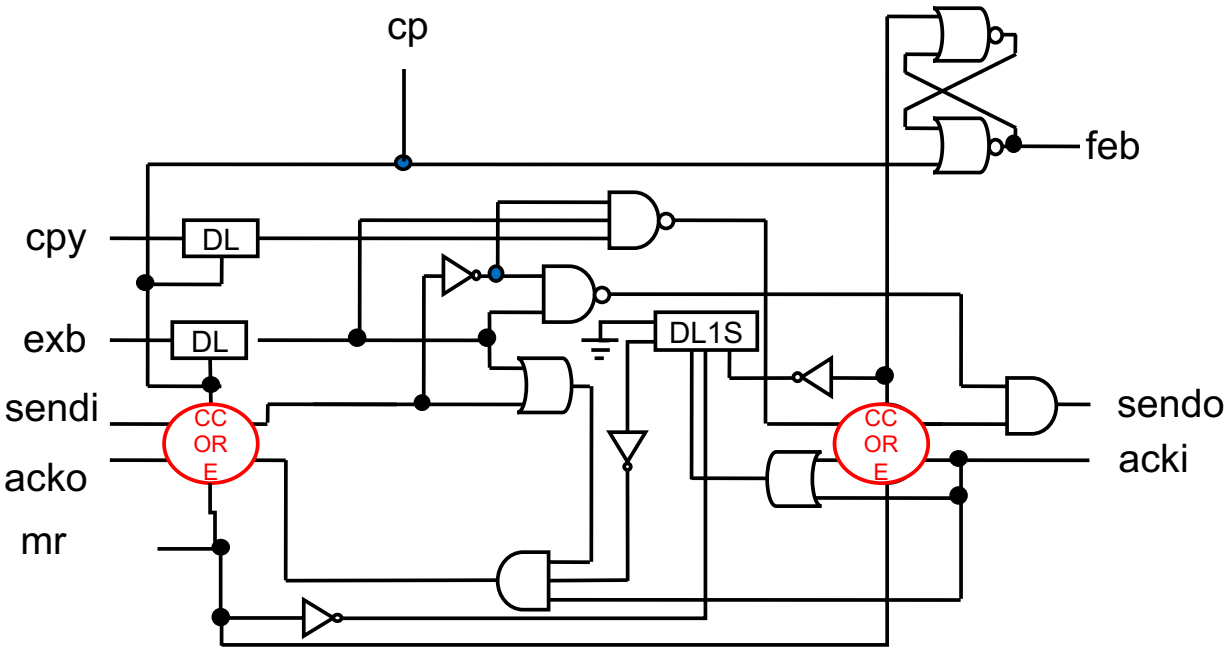


図 3.8　新しい CX2

- 自動設計に用いる Perl プログラム

  今回独自で作成した，自動設計のために Tcl スクリプトや JSON 形式のパス情報ファイルなどを出力するプログラム．

- プロジェクト情報ファイル：Project_info.txt

  作成するプロジェクト名と，使用する Quartus のバージョンが記述されたファイル．回路設計プロジェクトを作成する際に用いられる．以下のように記述されている．

  ```
  Quartus_Version="18.0.0 Standard Edition "
  Project_Name=DDP
  ```

- Verilog ソースがあるディレクトリへのパス情報ファイル：Path.txt

  Verilog ソースを置いているディレクトリへの相対パスが記述されているファイル．回路設計プロジェクト作成時に，プロジェクトへ Verilog ソースコードを入力する際に用いられる．記述は以下のように行う．

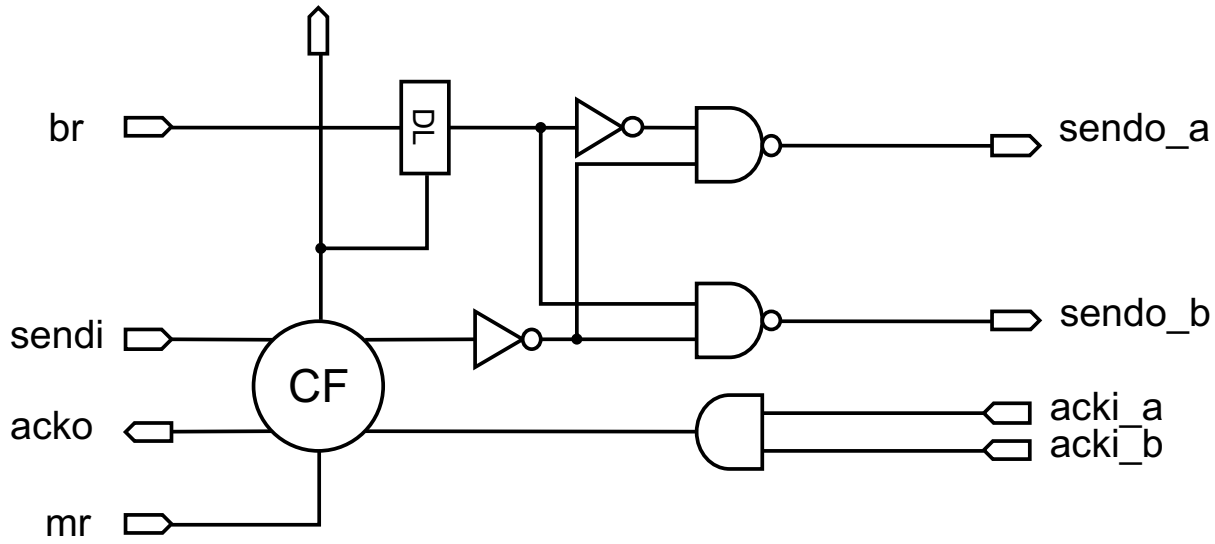


図 3.9　元の CB

```
../Verilog_Sorce/*
../Verilog_Sorce/ALU/*
```

Verilog_Sorce は，デフォルトの Verilog ソースを置いておくためのディレクトリである

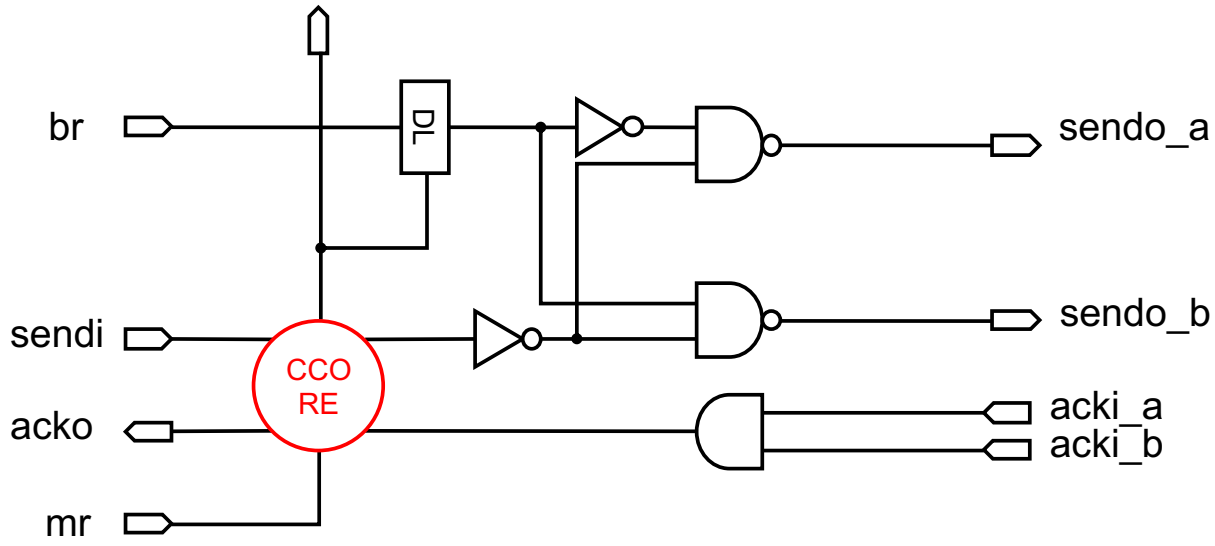


図 3.10　新しい CB

（3.3.3 ディレクトリ構造 参照）．さらにそれ以下にディレクトリを作る場合（ALU 等）2 行目の様に追加していく．

- モジュール接続情報ファイル：JsonPath.txt

  Perl で JSON 形式のモジュール間パス情報を出力する際に入力されるファイル．

- SDC ファイル入力情報ファイル：SDC_input.txt

  Perl で SDC ファイルを自動編集する際に，入力されるファイル．C 素子内の LUT をレジスタとして認識させるために必要な入力信号に，クロック特性を持たせるために用いられる．

- シェルスクリプトファイル：auto.sh

  DDP の自動設計に必要な実行プログラムやスクリプトが，設計フローの順序で記述されている．このファイルを実行することで，DDP の自動設計を行うことができる．

3.4　設計自動化フロー

DDP の設計フローは第 2 章の図**??**の通りである．今回，自動化した設計ステップは，回路設計プロジェクト作成，パーティション設定，Region 設定，配置・配線，タイミング解析・検証，コンフィグレーションである．タイミング調整はタイミング解析・検証の結果を受けて，設計者が直接遅延回路の Verilog ソースコードを修正して調整を行う．以降で各ステップにおける入力ファイル，動作させるプログラム，出力ファイルについて説明する．

3.4.1　回路設計プロジェクト作成

回路設計プロジェクト作成のステップにおける，入出力ファイルとプログラムを表 3.2 にまとめる．

Project_Create.pl は，デフォルトの qsf ファイル，Verilog ソースコードのディレクトリへの相対パス情報（Path.txt），プロジェクト情報（Project_info.txt）を入力として新たな

表 3.2　回路設計プロジェクト作成

入力ファイル	デフォルトの qsf ファイル，Path.txt，Project_info.txt
プログラム	Project_Create.pl
出力ファイル	Project_Create.tcl

回路設計プロジェクトを作成する Tcl スクリプトファイル Project_Create.tcl を出力する．デフォルトの qsf ファイルからは，デバイスの情報や FPGA チップ周りの入出力信号を抽出する．Path.txt からは，相対パス情報を抽出し，それを利用して対象のディレクトリ以下にある Verilog ソースコードのファイル名を全て取得する．Project_info.txt からは，プロジェクト名と Quartus のバージョンを抽出する．抽出した情報を利用して，新たな qsf ファイルを作成する Tcl スクリプトを作成し，Project_Create.tcl として出力する．これを，以下のような，NiosII Command Shell 独自のコマンドで実行すると新たな回路設計プロジェクトが作成される．ディレクトリから対象ファイルを自動的にプロジェクトに追加することができるため，ファイルの追加漏れを防ぐことができ，またファイルが多いほど効率的になる．

```
quartus_sh -t Project_Create.tcl
```

プロジェクト作成後，以下のコマンドでコンパイルを行う．

```
quartus_sh --flow compile プロジェクト名
```

コンパイルが完了すると，配置・配線結果レポート（fit.rpt）が出力される．

3.4.2　パーティション設定

パーティション設定における，入出力ファイルとプログラムを表 3.3 にまとめる．

Partition_Setting.pl は，Project_info.txt，fit.rpt を入力として，パーティションを設定する Tcl スクリプトファイル Partition_Setting.tcl を出力する．Project_info.txt からはプロジェクト名を抽出する．fit.rpt からは全ての回路モジュールのプロジェクト内における

表 3.3　パーティション設定

入力ファイル	Project_info.txt，fit.rpt
プログラム	Partition_Setting.pl
出力ファイル	Partition_Setting.tcl

ノード名を抽出する．抽出したプロジェクト名からは，プロジェクトを開くための Tcl スクリプト，ノード名からパーティションを設定する Tcl スクリプトを作りだし，それらを Tcl ファイルの Partition_Setting.tcl に出力する．以下のコマンドで qsf ファイルにパーティションの設定を反映させた後に，コンパイルを行う．全てのモジュールをまとめてパーティション設定することができるため，GUI で 1 つずつ回路モジュールを選んでパーティションを設定するより効率的である．DDP では約 350 個のモジュールにパーティションを設定するため，GUI 作業ではとても行えない．

```
quartus_sh -t Partition_Setting.tcl
```

3.4.3　Region 設定・配置・配線

Region 設定，配置・配線における，入出力ファイルとプログラムを表 3.4 にまとめる．

表 3.4　Region 設定

入力ファイル	Project_info.txt，fit.rpt
プログラム	Region_Setting.pl
出力ファイル	Region_Setting.tcl

Region_Setting.pl は，Project_info.txt，fit.rpt を入力として，DDP 全体の Region を設定する Tcl スクリプトファイル Region_Setting.tcl を出力する．Project_info.txt からはプロジェクト名を抽出する．fit.rpt からは DDP モジュールのノード名を抽出する．プロジェ

クト名からは，プロジェクトを開くための Tcl スクリプト，ノード名から Regioin を設定する Tcl スクリプトを作りだし，それらを Tcl ファイルの Region_Setting.tcl に出力する．以下のコマンドで qsf ファイルに Region の設定を反映させた後に，コンパイルを行うことで，Quartus が Region の設定を反映させて，自動配置・配線を行う．

```
quartus_sh -t Region_Setting.tcl
```

3.4.4　タイミング解析・検証

タイミング解析・検証における，入出力ファイルとプログラムを表 3.5 から表 3.8 にまとめる．

表 3.5　SDC ファイル

入力ファイル	デフォルトの SDC ファイル，SDC_input.txt，fit.rpt
プログラム	SDC.pl
出力ファイル	新たな SDC ファイル

SDC.pl は，デフォルトの SDC ファイル，SDC_input.txt，fit.rpt を入力として，新たな SDC ファイルを出力する．SDC_input.txt から，DDP のリセット信号，Ack_In 信号，lopen に接続している信号名を抽出する．fit.rpt からは CP 信号を出力している LUT のノード名を抽出する．それらに create_clock で 20MHz のクロック特性を与え，Quartus に DDP を 20MHz で動作する同期回路として認識させる．

表 3.6　タイミング解析

入力ファイル	Project_info.txt
プログラム	timingcheck_tcl.pl
出力ファイル	timingcheck.tcl

timingcheck_tcl.pl は，Project_info.txt を入力として，タイミング解析を行う Tcl スクリプトファイル timngcheck.tcl を出力する．Project_info.txt からプロジェクト名を抽出して，プロジェクトを開く Tcl スクリプトを作成し，また，ネットリストを作り出す Tcl スクリプト create_timing_netlist，SDC ファイルを読み込む Tcl スクリプト read_sdc，ネットリストのアップデートを行う Tcl スクリプト update_timing_netlist，回路のタイミング情報を抽出する Tcl スクリプト report_timing を作成し，Tcl ファイルの timingcheck.tcl へ出力する．以下のコマンドでタイミング解析を行う．

```
quartus_sta -t timingcheck.tcl
```

タイミング解析の結果は，timingreport.txt に出力される．

表 3.7　JSON 形式パス

入力ファイル	JsonPath.txt，fit.rpt
プログラム	JsonPath_create.pl
出力ファイル	path.json

JsonPath_create.pl はタイミング検証ツールに入力するための JSON 形式のモジュール間パス情報を出力する．fit.rpt から抽出する必要のあるモジュールを，JsonPath.txt から抽出し，それを参考にして，fit.rpt から対象のモジュールのノード名を抽出し，JSON 形式のモジュール間パス情報を作成し，path.json に出力する．

表 3.8　タイミング検証

入力ファイル	timingreport.txt，path.json
プログラム	TimingChecker.py
出力ファイル	timingresult.txt

TimingChecker.py はタイミング解析の結果（timingreport.txt）と JSON 形式のモジュー

ル間パス情報（path.json）を入力として，タイミング検証を行い，タイミング制約を満たしているかどうかチェックする．タイミング検証の結果は timingresult.txt に出力され，タイミング違反がある場合，設計者はタイミング調整を行う．タイミング違反がない場合，コンフィグレーションへ進む．

3.4.5　コンフィグレーション

コンフィグレーションにおける，入力ファイルとプログラムを表 3.9 にまとめる．出力ファイルは存在しない．

表 3.9　コンフィグレーション

入力ファイル	sof ファイル
プログラム	Configuration.pl
出力ファイル	-

Configuration.pl は，コンパイルで Quartus が出力する sof ファイル（FPGA に回路情報を書き込むためのファイル）を入力としてコンフィグレーションを行う．

3.5　結言

本章では，動作を保証した DDP の設計のために，データ転送制御回路（CM，CE，CX2，CB）のアーキテクチャを，第 2 章で述べた CCORE と同様に，LUT を用いたアーキテクチャに変更したたことについてまとめた．CM では新しく CJCORE**??**という回路を作成することで，LUT を含むアーキテクチャに変更した．CE，CX2，CB では，元の CF と呼ばれる回路を，CCORE に置き換えることで LUT を含むアーキテクチャに変更した．DDP の設計自動化は，独自で作成した Perl プログラムを用いて，DDP の設計フローにおける Quartus の GUI 処理を，対応する Tcl スクリプトに変換した CUI 上（NiosII Command Shell）で動作させ，また，タイミング検証ツール [4] に入力する JSON 形式の回路モジュー

ル間パス情報を自動生成することによって実現した.

第 4 章

評価

4.1 緒言

本章では，第 3 章で提案した手法と過去に提案した手法 [3] それぞれの方法で DDP の設計を行い，各評価指標それぞれにおいて，どちらの設計手法が有効であるか議論していく．評価指標は，DDP の設計時間（タイミング調整の終了まで），DDP 全体の回路規模と各ステージの遅延回路の回路規模，DDP 全体の回路面積，スループットである．設計対象の FPGA ボードは Intel 社製 MAX10 DE10-LITE（図 4.1）で，FPGA チップは MAX10 10M50DAF484C7G である．FPGA 用回路設計ツールは Intel 社製 Quartus Prime Standard Edition 18.0 を用いた．

4.2 評価条件

評価対象は，今回提案した手法で設計した DDP と，過去に提案した手法 [3]（以降従来手法）で設計した DDP である．設計する DDP の仕様は表 4.1 の通りである．設計対象の FPGA ボードは Intel 社製 MAX10 DE10-LITE（図 4.1 で，FPGA チップは MAX10 10M50DAF484C7G である．また，回路設計ツールは Intel 社製 Quartus Prime Standard Edition 18.0 を用いた．

従来手法 [3] ではパーティションの設定を行っていなかったが，タイミング検証ツール [4] を用いて性能評価を行うために，今回はパーティションを設定して合成する．

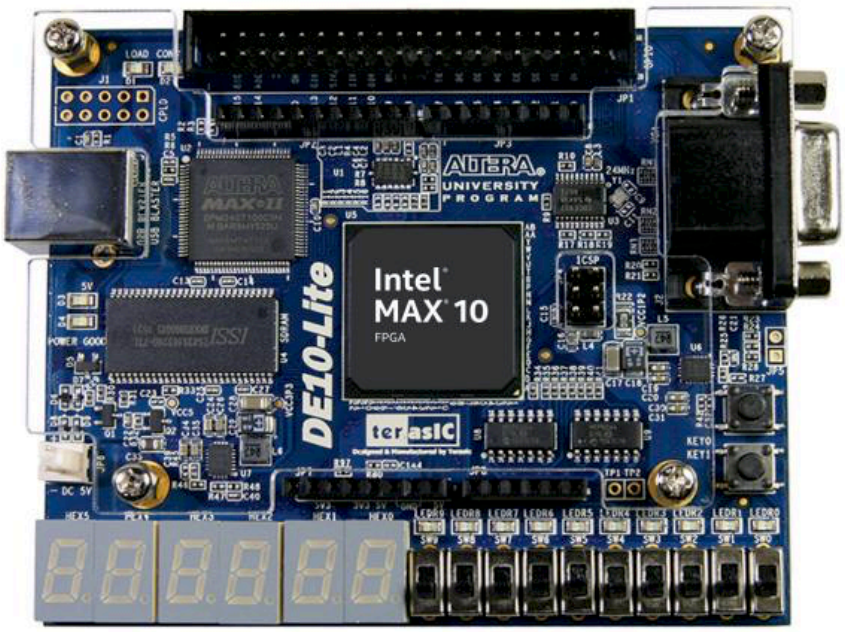


図 4.1　MAX10 DE-10LITE

4.3　DDP の設計時間の評価

提案手法と従来手法 [3] で DDP の設計時間を比較し，どちらが有効であるかを示す．設計時間は回路設計プロジェクト作成からタイミング調整完了までとする．

設計時間は表 4.2 のようになった．提案手法は従来手法 [3] と比べて 27%設計時間を削減できたため，提案手法の方が有効である．50%以上削減可能であると予想していたが，タイミングを調整してから，回路の再合成に想像以上に時間がかかってしまった．遅延回路 1 つの変更だけでも，再合成に 20 分前後かかっていた．パーティションを設定すると，Verilog ソースコードに変更があった場合，設定しない場合よりも回路合成に時間がかかってしまうため，それが原因であると考えられる．

表 4.1　設計する DDP の仕様

パケットフォーマット	col：3bit
	gen：8bit
	dest：7bit
	LR：1bit
	CPY：1bit
	C：1bit
	Z：1bit
	Data：16bit
メモリサイズ	CST：20bit × 64 words
	MMCAM：20bit × 64words
	MMRAM：24bit × 64words
	DMEM：16bit × 1024words
	PS：13bit × 128words

表 4.2　設計時間 [min.] と時間削減率 [%]

従来手法 [3]	提案手法	時間削減率
324	237	27

4.4　回路規模の評価

それぞれの手法で設計した DDP 全体の回路規模と，ステージごとの遅延回路の規模を比較し，どちらが有効であるか示す．回路規模は，回路の合成に使用される素子の LE（Logic Element）の数で比較する．DDP 全体の回路規模を表 4.3，ステージごとの遅延回路（Ds と Da）の規模を表 4.4 にまとめる．

DDP 全体の回路規模は従来手法と比べて 1.3%増加した．各パイプラインステージの logc

表 4.3　DDP の回路規模 [LEs] と比率

従来手法 [3]	提案手法	比率
4830	4893	1.013

表 4.4　ステージごとの遅延回路規模 [LEs]

ステージ	従来手法 [3]		提案手法		比率	
	Ds	Da	Ds	Da	Ds	Da
M	5	1	12	1	2.4	1
CST	23	1	16	1	0.7	1
MMCAM	4	1	16	1	4	1
MMRAM	9	1	16	1	1.7	1
ALU	19	1	16	1	0.84	1
DMEM	6	1	16	1	2.6	1
COPY	4	1	8	1	2	1
PS	10	1	16	1	1.6	1
B	5	1	16	1	3.2	1

やデータ・ラッチの構成は全く一緒であり，遅延回路の規模だけ相違があるため，妥当な結果である．

ステージごとの遅延回路の規模を比べてみると，全てのステージにおいて Tr を調整する遅延回路（Da）の規模は同じであった．Tf を調整する回路（Ds）の規模は全体的に増加しており，最大 4 倍になっているステージもある．その一方で，CST ステージと ALU ステージの遅延回路（Ds）の規模はそれぞれ 30%と 16%減少した．

また，提案手法の遅延回路（Ds）は従来手法 [3] と比べて，ステージごとにばらつきがなかった．これは，各ステージのクリティカルパスの遅延時間にそこまで差がなく，20[ns] 前

後であったためである．ステージによって logic の複雑さが違うため，クリティカルパスの遅延時間に差が出るように考えられたが，おそらく SDC（設計制約）ファイルが影響している．今回 SDC ファイルでは，全ての C 素子の CP 信号を出力している LUT に 50MHz のクロック特性を持たせている．つまり，全てのステージが 50MHz 動作するように設定しているため，Quartus がその制約を満たすような同期式回路と同等な DDP を合成したと考えられる．

一方で従来手法では設計時に，DDP を同期式回路に一時的に変換してクリティカルパスの遅延時間を計測しているが，この時に SDC ファイルで何の制約も設定していない，そのため，Quartus がステージの logic ごとに最速で動作する同期式回路を設計してしまい，その結果クリティカルパスの遅延時間に差がでたと考えられる．そこで，従来手法 [3] で設計後に SDC ファイルを従来手法と同様に設定して再合成を行い，タイミング検証ツールを用いて調べたところ，遅延回路（Ds）の規模が大きい CST ステージと ALU ステージ以外でタイミング違反を起こしていた．そのため，従来手法 [3] では DDP の動作保証ができないため，提案手法が有効であるといえる．

4.5　DDP の回路面積の評価

それぞれの手法で設計した DDP の回路面積を比較し，どちらが有効であるか示す．回路面積は，DDP 全体の Region の幅と高さから求める．回路面積の結果を表 4.5 にまとめる．また，従来手法と提案手法の配置・配線結果を 4.2 に示す．

従来手法 [3] と比較すると，回路面積は 9.3%増加した．従来手法 [3] は，DDP の各ステージの logic，データ・ラッチ，C 素子に Region を設定して，全ての Region が最小にかつ，

表 4.5　DDP の回路面積 [W * H] と比率

従来手法 [3]	提案手法	比率
32 * 16	20 * 28	1.093

隣接するステージが近くに配置にされるように手動で配置・配線を行っている．これは,，マルチコア構成へ拡張することを想定して，面積の最小化と，各回路モジュールが配置・配線される場所を固定することによって，配線遅延を最大限固定化させることを目的としている．一方，提案手法では DDP 全体の Region のみ設定して，配置・配線は Quartus の自動配置・配線機能に任せているため，Region は最小でなく，また，各回路モジュール（ステージの logic，データ・ラッチ等）が Region 内に自由に配置され，回路モジュール間の配線遅延が提案手法と比べて安定しない．FPGA にシングルコアで実装する場合であれば問題はないが，IoT デバイスはマルチコア構成で性能を向上することが望まれるため，配置・配線手法は従来手法 [3] の方が有効である．

4.6　スループットの評価

それぞれの手法で設計した DDP のスループットを比較し，どちらが有効であるか示す．スループットとは単位時間あたりに処理可能なパケット数のことであり，DDP のスループットは以下の式から求まる．

$$\frac{1}{Tf + Tr \text{ の最大値}}$$

それぞれのスループットは表 4.6 のようになった．

表 4.6　DDP のスループット [packets/s] と比率

従来手法 [3]	提案手法	比率
31.4M	30.8M	0.98

スループットは，従来手法 [3] と比べて, 2%低下した．しかし，これは回路が正しく動作することが前提の結果であり，回路規模の評価でも述べたように，従来手法 [3] で設計した DDP は，タイミング違反を起こしている．そのため，従来手法 [3] におけるスループットは信頼性が低い．そのため，スループットは若干劣っているが提案手法の方が有効であるといえる．

4.7　結言

本章では，Intel 社製の FPGA チップ MAX10 を対象として，従来手法 [3] と提案手法のそれぞれで DDP を設計し，評価指標においてどちらが有効であるかを述べた．評価指標は，DDP の設計時間（タイミング調整の終了まで），DDP 全体の回路規模と各ステージの遅延回路の回路規模，DDP 全体の回路面積，スループットである．DDP の設計時間において，提案手法が 27%設計時間を短縮することができたため，設計時間は提案手法が有効であることを述べた．回路規模においては，提案手法が 1.3%増加する結果であったが，従来手法 [3] はタイミング制約を満たしていないため，提案手法が有効であると述べた．回路面積においては，提案手法が 9.3%増加し，また，DDP 内の回路モジュールの配置・配線が安定しないため従来手法の配置・配線方法が有効であると述べた．最後にスループットにおいて，提案手法が 2%低下する結果となったが，回路規模の評価で述べたように，従来手法で設計した DDP はタイミング違反があるため，提案手法が有効であると述べた．4 つの評価指標のうち，3 つで提案手法が有効である結果となった．

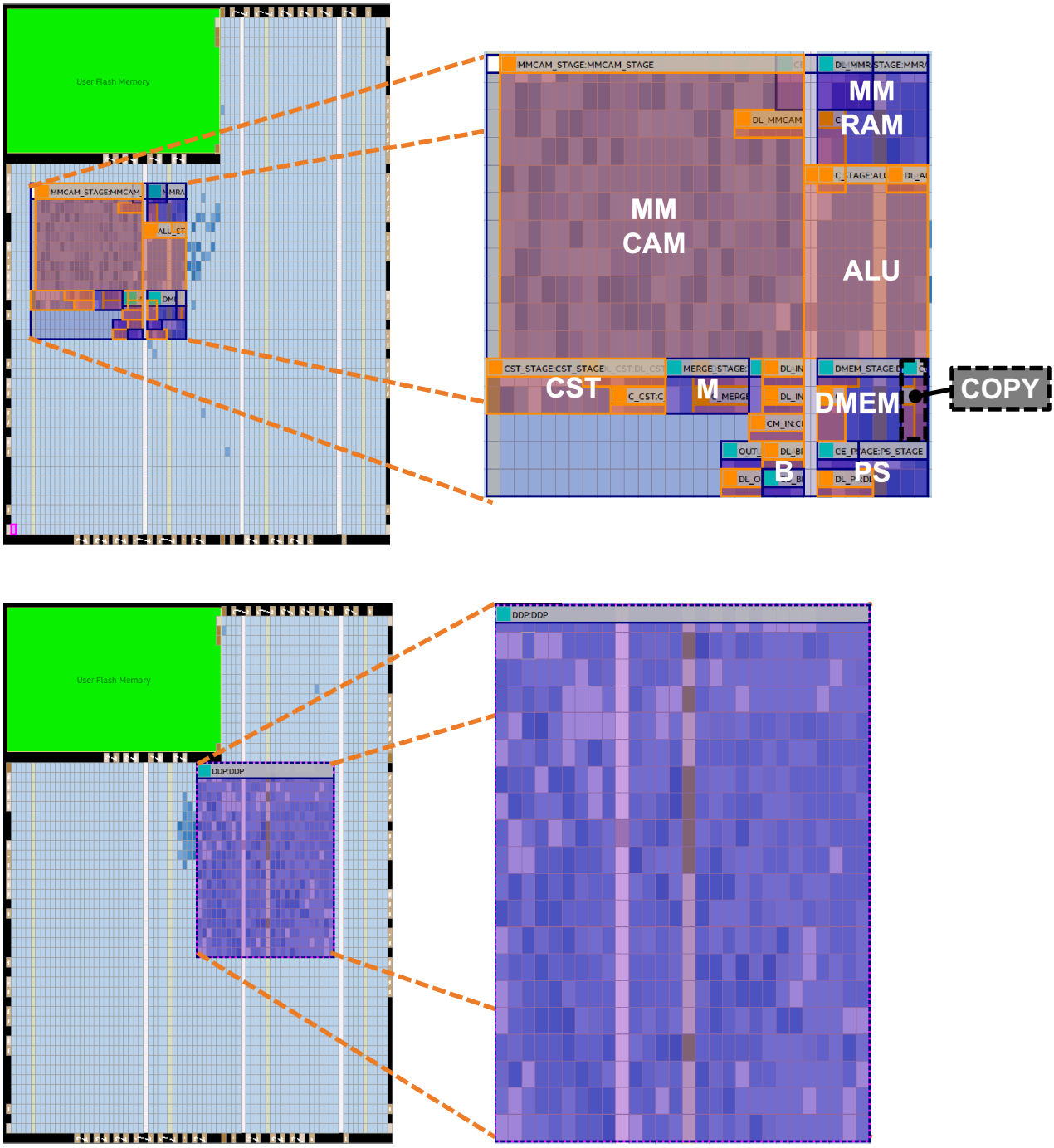


図 4.2　従来手法 [3]（上）と提案手法（下）の配置・配線結果

第 5 章

結論

近年，IoT（Internet of Things）技術の普及に伴って，IoT デバイス数は年々増加してきており，2022 年までには 348 億台にまで到達することが予測されている．IoT デバイスは，主にエッジ・コンピューティングやフォグ・コンピューティングのエッジ機器として用いられ，長時間稼働してセンサからデータを取得し，高速に処理することが求められ，また，多種多様な分野のシステムに応用することが期待されている．主な IoT デバイス用マイクロプロセッサは，グローバルなクロック信号で回路を制御するノイマン型の同期回路であり，複数のセンサからデータを取得した場合，割り込み処理等の前処理として処理中のデータの退避や処理を復帰させる命令の実行等が行われる．しかし，複数のセンサから頻繁にデータを取得する IoT デバイスにおいて，このような処理は総処理時間の増加や消費電力の増大を起こしてしまい問題となり，IoT デバイスの高性能化，低消費電力化，多様化に対する需要が高まっている．

データ駆動型プロセッサ（DDP:Data-Driven Processor）[2] は，低消費電力で動作し，異なるデータ流を多重に処理可能で高性能なため，IoT システムに有効なアーキテクチャである．また，FPGA は応用に合わせた柔軟な回路設計が可能で，多様性が求められる IoT エッジ機器に有効である．以上から，DDP を FPGA 上に実装して IoT エッジ機器を実現する方法が有望である．しかし，既存の FPGA と回路設計ツール（Intel 社 Quartus）は同期式回路に最適化され，非同期式回路の DDP を最適に設計することが困難である．商用の FPGA に非同期式回路を実現するために様々な研究が行われているが，既存の回路設計ツール（Intel 社 Quartus）の機能を巧妙に利用したり，安全に設計するための独自の入力ファイルを生成する必要があり，設計に多大な時間がかかってしまい，複雑な GUI 操作も

行うため設計難易度が高い．そのため，本研究では，動作が保証された DDP の設計にかかる時間を削減し，かつシンプルな入力ファイルで DDP の設計を行えるように，FPGA を対象とした DDP の設計自動化フローの検討を行った．DDP の設計自動化は，独自で作成した Perl プログラムを用いて，DDP の設計フローにおける Quartus の GUI 処理を，対応する Tcl スクリプトに変換した CUI 上（NiosII Command Shell）で動作させ，また，タイミング検証ツール [4] に入力する JSON 形式の回路モジュール間パス情報を自動生成することによって実現した．

第 4 章で，今回の提案手法と，過去に提案した手法 [3] を用いて Intel 社製 FPGA チップ MAX10 を対象として DDP を設計し，設計時間，DDP 全体の回路規模と各ステージの遅延回路の規模，回路面積，スループットを比較して，有効性を評価した．設計時間は 27%削減でき，回路規模は 1.8%増加し，スループットが 2%低下したが，従来手法 [3] で設計した DDP をタイミング検証を行うとタイミング違反があったため，提案手法の設計方法が有効であるといえる．しかし，回路面積においては，9.3%増加し，また従来手法 [3] の配置・配線と比較すると，提案手法は回路モジュールの配置・配線場所が安定せずマルチコア設計を考慮した場合は，従来手法 [3] の方が有効であるといえる．4 つの評価指標のうち 3 つが提案手法の方が優位を示したため，提案手法は有効性のある設計方法であるといえる．

今後の課題として一番に挙げられるのは，従来手法 [3] のような，マルチコア設計を考慮したヒューリスティックな配置・配線案を自動で作り出し，提案手法に組み込むことである．[10] の研究では，DDP を特定の FPGA チップに対してのみ，[3] のように，ヒューリスティックな配置・配線案を作り出す手法を提案している．そのため [10] の提案手法を改良し，FPGA チップに依存せずに，ヒューリスティックな配置・配線案を生成できるようにする必要がある．

次に挙げられるのは，設計時間のさらなる短縮方法の提案である．従来手法 [3] と比べると設計時間は減少したが，まだ，4 時間近く設計にかかっている．最も時間がかかっているのは，遅延回路の Verilog ソースコードを修正し，回路の再合成を行う時間である．遅延回路 1 つの修正だけでも再合成に 20 分以上かかっているため，もっと高速に再合成できるよ

うな設定方法があれば，より効率的に動作が保証された DDP を設計することが可能となる．

最後に，今回のタイミング検証では，全てのステージにおいて，クリティカルパス遅延時間がほぼ同じで 20ns になってしまっていた．これは，タイミング解析結果の Data Required Time（データ要求時間）をタイミング検証のプログラムで取得してクリティカルパス遅延時間として比較していたためである．Data Required Time とは，同期回路で後段の DL のクロック信号が立ち上がるまでの時間であり，図 5.1 のから読み取れるようにクリティカルパス遅延時間は関係しておらず，SDC ファイルで設定した値（今回は 20ns）に Clock Delay（図 5.2 の LUT8 から DL_i までの遅延時間）を加えた時間である．そのため，実際のステージのクリティカルパス遅延時間と異なる可能性がある．正しいクリティカルパス遅延時間を取得する場合は図 5.1 の Data Arrival Time（データ到着時間）を取得しなければならない．Data Arrival Time は図 5.1 と図 5.2 で示すように，Clock Delay（LUT3 から DL_{i-1} までの遅延時間）に Data Delay（DL_{i-1} から DL_i までの遅延時間）を加えた時間である．これをタイミング解析結果から抽出し Tf と比較することで，より正確なタイミング検証ができ，またスループット性能が改善される可能性があるため，Data Required Time と Data Arrival Time を抽出し，それぞれで遅延回路を設計して実機で動作確認をして評価する必要がある．

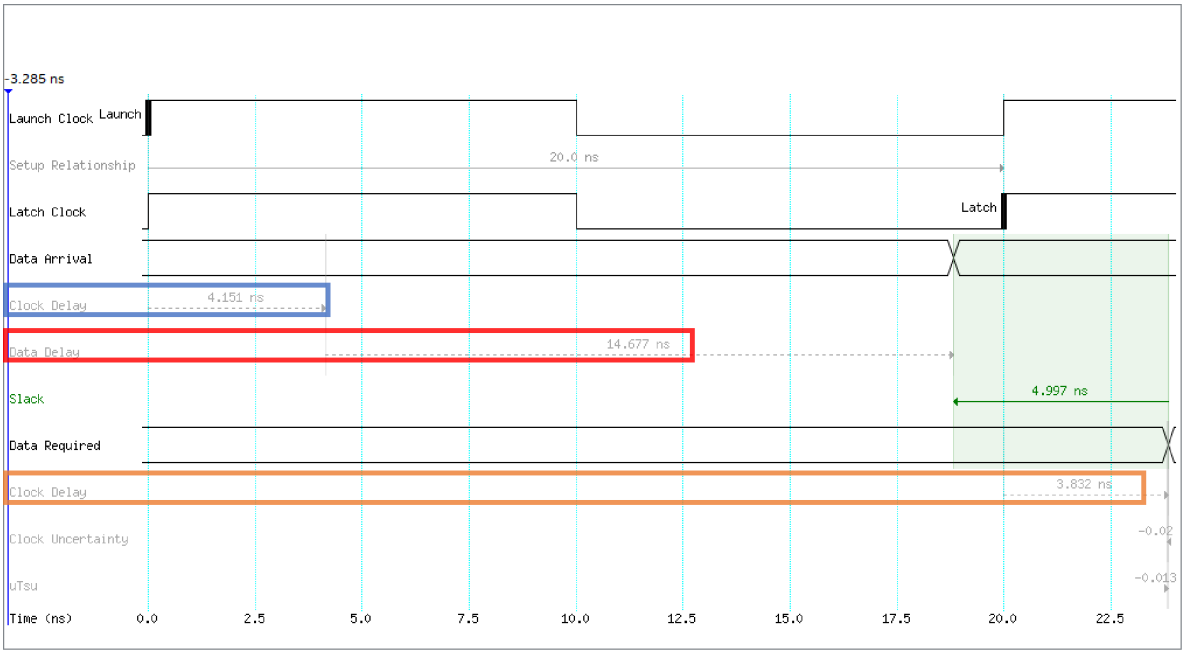
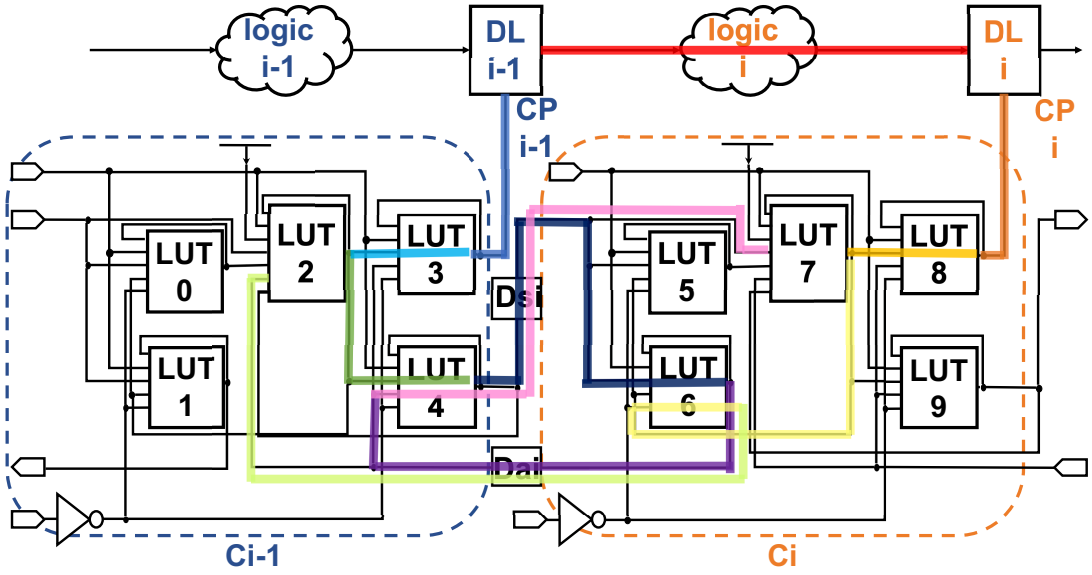


図 5.1　タイミング解析による信号の波形



Data Arrival Time = Clock Delay$_{i-1}$ + Data Delay

Tf = T_2 + T_3 + T_4 + T_5

Tr = T_6 + T_7

Tf > T_1 + Data Arrival Time - T_8 - Clock Delay$_i$

Tr > T_1 + Clock Delay$_{i-1}$ - T_8 - Clock Delay$_i$

図 5.2　STP のタイミング制約に用いられる時間の経路と制約条件

謝辞

本研究に際して，様々なご指導をいただきました岩田誠教授に深く感謝申し上げます．お忙しい中，貴重なお時間を割いて相談に乗って頂いたおかげで本研究を進めることができました．学部 1 年次の授業から始まり，学部 3 年次の研究室配属から修士 2 年次までの間，就職活動や精神的なサポートなど，勉学以外でも大変お世話になりました．ここに感謝の意を表します．

本研究の進めるにあたり，大変貴重な技術提供をしていただきました，筑波大学修士 1 年の吉川千里様，助教の三宮秀次様に深く感謝申し上げます．

ご多忙の中，本研究の副査を務めて下さり，様々な疑問点や改善点などを指摘していただいた横山和俊教授，松崎公紀教授に深く感謝申し上げます．

また，研究室に配属されてから同期として大学院まで共に歩んでくださった，楠田健太氏，汐見興明氏に心より感謝致します． 研究室の後輩として，日頃からご支援，ご協力頂いた修士 1 年の井上聡氏，学部 4 年の岡野秀平氏，尾ノ井嶺卓氏，筧拓也氏，古田雄大氏，小谷拳聖氏，高見結衣氏，学部 3 年の仁野槙人氏，渡邊竜也氏，高橋龍一氏に心より感謝致します．

最後になりましたが，親兄弟をはじめ，日頃からご支援いただきました関係者の皆様に心より御礼申し上げます．

参考文献

[1] 総務省, “令和 2 年版 情報通信白書 IoT デバイスの急速な普及,” https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r02/html/nd114120.html, 2021 年 2 月 2 日参照.

[2] Hiroki Terada, Souichi Miyata, and Makoto Iwata, “DDMP’S: Self-Timed Super-Pipelined Data-Driven Multimedia Processors,” Proceedings of the IEEE, Vol. 87, No. 2, pp. 282–296, Feb. 1999.

[3] Kanji Nagano and Makoto Iwata, “Area Afficiant Implementation of Data-Driven Processors,” ISFT2019, pp. 58, Aug. 2019.

[4] Senri YOSHIKAWA, Shuji SANNOMIYA, Makoto IWATA and Hiroaki NISHIKAWA, “Pipeline Stage Level Simulation Method for Self-Timed Data-Driven Processor on FPGA,” 2020 8th International Electrical Engineering Congress (iEECON), Chiang Mai, Thailand, 2020, pp. 1–5, doi: 10.1109/iEECON48109.2020.229515, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9077529&isnumber=9077410.

[5] 滝澤 恵太郎, 齋藤 寛, “束データ方式による非同期式回路の FPGA 設計支援環境の構築,” 情報処理学会研究報告, Vol. 2015-SLDM-171, No, 5, pp. 1–6, May. 2015, https://ipsj.ixsq.nii.ac.jp/ej/index.php?active_action=repository_view_main_item_detail&page_id=13&block_id=8&item_id=141655&item_no=1.

[6] Keitaro Takisawa, Shunya Hosaka and Hiroshi Saito, “A design support tool set for asynchronous circuits with bundled-data implementation on FPGAs,” 2014 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, 2014, pp. 1-4, doi: 10.1109/FPL.2014.6927435, https://ieeexplore.ieee.org/document/6927435.

[7] Maruzio Tranchero and Leonald M. Reyneri, “Exploiting synchronous placement for asynchronous circuits onto mommercial FPGAs,” Proceeding of FPL, pp. 622–625, 2009.2009 International Conference on Field Programmable Logic and Applications, Prague, 2009, pp. 622-625, doi: 10.1109/FPL.2009.5272378, https://ieeexplore.ieee.org/document/5272378.

[8] Maruzio Tranchero and Leonald M. Reyneri, “Implementation of Self-Timed Cirucuits onto FPGAs Using Commercial Tools,” 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, Parma, 2008, pp. 373-380, doi: 10.1109/DSD.2008.73, https://ieeexplore.ieee.org/document/4669259.

[9] Intel, “Intel Quartus Prime Standard Edition User Guide: Getting Started, ” https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html, 2021 年 2 月 2 日参照.

[10] 井上 聡, “データ駆動型プロセッサの FPGA 実装におけるフロアプラン最適化の検討,” 高知工科大学卒業論文, 2020, https://www.kochi-tech.ac.jp/library/ron/pdf/2019/03/13/a1200290.pdf.