令和 3 年度

修士学位論文

データ駆動型プロセッサの FPGA 向き回路最適化手法の検討

～データ転送制御回路に着目した最適化～

A Study on FPGA Circuit Optimization of Data-Driven Processor

- - Focusing on Data Transfer Control - -

1245117　　井上 聡

指導教員　　岩田 誠

2022 年 2 月 4 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

データ駆動型プロセッサの FPGA 向き回路最適化手法の検討
～データ転送制御回路に着目した最適化～

井上 聡

近年，IoT エッジ機器に対する高性能化・低消費電力化・多様化の要求が益々高くなっている．セルフタイム型パイプラインにより実現されたデータ駆動型プロセッサ DDP(Data-Driven Processor) は複数ストリームデータの多重処理性能が高く，低消費電力で動作するため，有効なアーキテクチャである．また，FPGA(Field Programmable Gate Array) を対象に実装することで，多様な用途に適応した回路を構成できるため，IoT エッジ機器の有効な実装法になりうる．一方，現行の FPGA 回路設計ツールは同期回路に最適化されており，DDP を非同期回路で実装する際，回路の最適化を充分に行えない．このため，FPGA を対象として，動作保証された DDP を設計することを目標として，DDP の設計自動化フローが提案された．しかしながら，この手法では全てのタイミング制約条件を網羅した検証ができていない．本研究では，先行研究のタイミング検証法を拡張し，データ転送制御回路のタイミング制約条件の網羅的検証による，該当パス遅延を最小に抑える手法を検討した．また，フロアプラン最適化手法を提案手法と併用することで回路の配線遅延を更に低減させた．その結果，提案手法は従来の設計手法と比較して，スループットは 12.2%向上，回路面積は 7.6%削減，設計時間は 75.4%短縮できることを確認した．

キーワード　Internet of Things，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, フロアプラン

Abstract

A Study on FPGA Circuit Optimization of Data-Driven Processor
- - Focusing on Data Transfer Control - -

Satoshi INOUE

In recent years, there is an increasing demand for higher performance, lower power consumption, and more flexibility in various IoT devices. Data-driven processor (DDP) realized by self-timed pipeline (STP) circuits is one of promising ways to meet those demands because the DDP can operate multiple data streams in parallel as well as STP can save power autonomously. If STP-based DDP ' s are implemented on field-programmable gate array (FPGA), they can be flexibly applied to various IoT devices.

However, there is no efficient FPGA circuit design tool for STP-based DDP ' s because commercial FPGA design automation tools (DAT ' s) are dedicated to clock synchronous circuits. Although a semi-automation tool for STP-based DDP ' s have been developed in our laboratory, it requires timing optimization of STP circuits in trial-and-error manner. This study proposes a delay timing optimization algorithm for STP circuit and a floorplan algorithm for DDP to reduce wiring length of STP. As a circuit design result of IoT-oriented STP-based DDP, the proposed algorithms achieve 12.2% throughput improvement, 7.6% reduction of circuit area, and 75.4% saving of design time compared to using the previous tool.

key words Internet of Things, Data-Driven Processor, Self-Timed Pipeline, FPGA, Floorplan

目次

目次

図目次

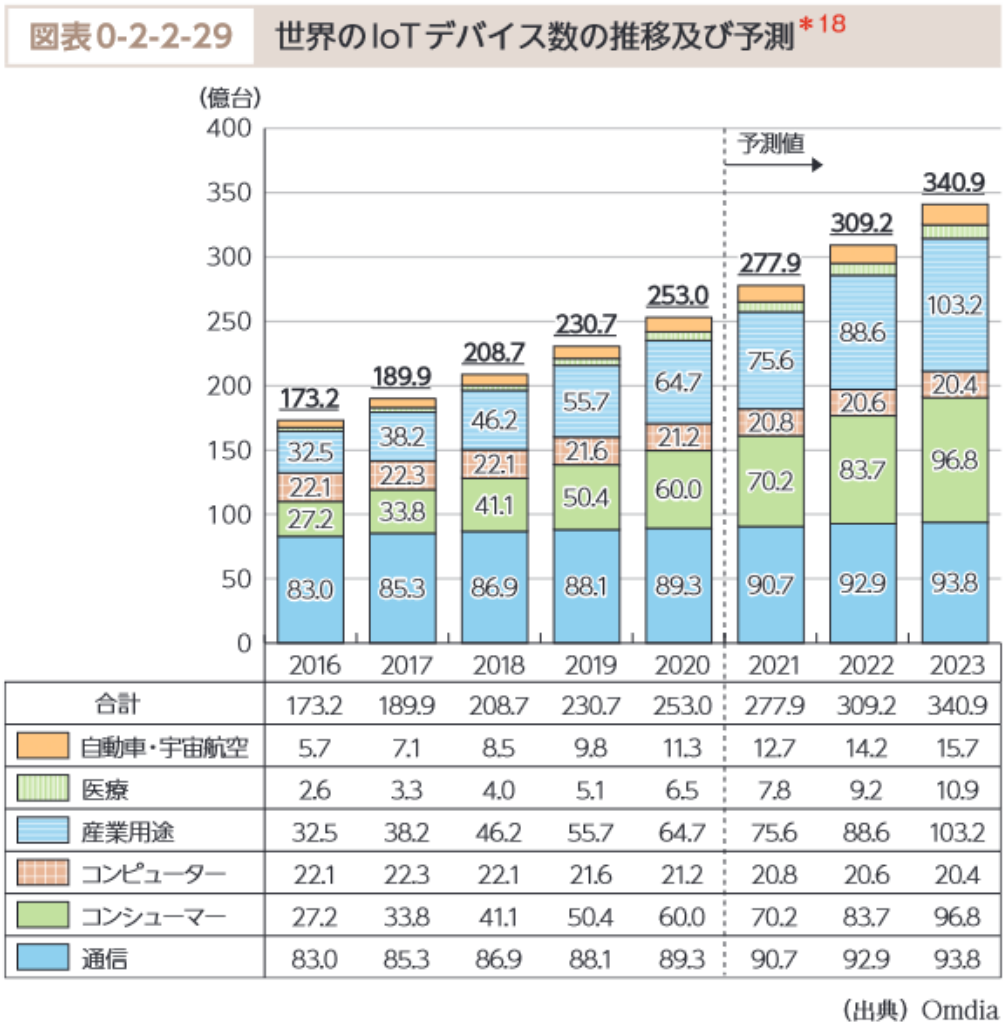
表目次

第 1 章

序論

近年，世界の IoT(Internet of Things) デバイスは一般社会への普及が加速しており，図 1.1 に示すように，2023 年には 340.9 億台に増加すると予測されている [1]．IoT 技術の普及は，クラウド技術や無線通信技術の発展による影響が大きいと考えられ，現在では PC やスマートフォンといった汎用的なインターネット接続端末に加え，家電や自動車，ビルシステムや工場の生産ライン等でもインターネットが接続されている．IoT システムの実現にあたって，システムの利用ユーザに近い端末やネットワーク内でデータを処理，集約するエッジコンピューティングが注目されている．これは現在広く活用されているクラウド技術の利用による，データセンターに処理を集約して行われるサービス提供ではなく，データ処理の一部をユーザに近いエッジデバイス側で行うことで，サーバへの負荷の分散や，リアルタイム性の向上を目指す手法である．これにより，自動運転車やセキュリティ関連技術を始めとした，リアルタイム性の高いシステムへの応用が期待され，特にセキュリティ関連技術ではクラウド技術と組合せることで，匿名性とリアルタイム性を兼ね備えた，より高性能なシステムが実現できると考えられる．エッジコンピューティングを用いたシステム開発は今後更に高まると予想され，IoT 技術の一般社会への普及を見据えて，より汎用的かつ高性能なエッジデバイスの開発が必要である．

代表的なエッジデバイス用のマイクロプロセッサとして，スマートフォンや組込みシステムなどで利用される ARM[2] や，カリフォルニア大学バークレイ校で開発された RISC-V[3] が挙げられる．しかし，これらのプロセッサはノイマン型プロセッサであり，グローバルなクロック信号で回路の制御を行う同期回路により実現されている．ノイマン型プロセッサが複数ストリームデータに対して多重処理を行う場合，割り込み処理や Simultaneous



	2016	2017	2018	2019	2020	2021	2022	2023
合計	173.2	189.9	208.7	230.7	253.0	277.9	309.2	340.9
自動車・宇宙航空	5.7	7.1	8.5	9.8	11.3	12.7	14.2	15.7
医療	2.6	3.3	4.0	5.1	6.5	7.8	9.2	10.9
産業用途	32.5	38.2	46.2	55.7	64.7	75.6	88.6	103.2
コンピューター	22.1	22.3	22.1	21.6	21.2	20.8	20.6	20.4
コンシューマー	27.2	33.8	41.1	50.4	60.0	70.2	83.7	96.8
通信	83.0	85.3	86.9	88.1	89.3	90.7	92.9	93.8

図 1.1　IoT デバイス数の推移及び予測 [1]

MultiThreading(SMT) 技術 [4] によってこれを実現している．しかし，割り込み処理の場合は前処理としてデータの退避や処理の復帰といった命令を実行する必要があり，SMT では実行時に利用可能な計算資源によって待機時間が変動する．そのため，総処理時間の増加や消費電力の増大を起こしてしまうことが課題であり，IoT エッジデバイスの高性能化，低消費電力化，多様化への需要が発生している．

データ駆動型プロセッサ DDP(Data-Driven Processor)[5] は，各パケットに対し，演算に必要な回路のみが動作する性質から省電力性を持ち，異なるパケットを多重に処理可能なことから，エッジデバイス用のアーキテクチャとして DDP は有効であると考えられる．

また，エッジデバイスには様々な用途が想定され，それぞれのデバイスに対して用途に合わせた回路設計を行うにあたり，最初から専用回路として設計される ASIC(Application Specific Integrated Circuit) と比較し，柔軟に内部の回路構成を改変可能な FPGA(Field Programmable Gate Array) が多品種少量生産に適していると考えられる．このことから，エッジコンピューティングを実現するにあたり，FPGA を用いた設計は有効であると考えられ，DDP をアーキテクチャとしたシステムに適用可能である．しかし，既存の Intel 社の Quartus や Xilinx 社の Vivado といった FPGA 設計ツールは同期式回路向けに最適化されており，DDP を始めとした非同期式回路の設計において動作を担保することが難しい．そのため，商用の FPGA に非同期式回路を実装するための研究 [6][7] がこれまで行われている．先行研究 [8] では FPGA を対象として，動作が保証された DDP を回路設計ツールの CUI コマンドや入力ファイルの自動生成スクリプトによって自動設計する手法が提案されている．また，FPGA における DDP の動作保証をするためのタイミング検証に関する研究 [9][7][10] も提案されており，設計時点で DDP の動作保証が実施可能な環境が整えられつつある．一方で，DDP の動作保証をするためのタイミング最適化手法が確立されておらず，設計者による試行錯誤によってタイミング検証が行われている．具体的には，データ転送制御回路で必要な遅延回路の個数を設計者が試行錯誤で決定し，その都度回路の再設計，タイミング検証を行い，膨大な時間をかけて回路の最適化を行っている．このため，DDP の設計に必要な時間が大きいことが課題となっている．また，従来手法 [8] では回路の配置配線を回路設計ツールの自動配置機能を用いて行っていたが，FPGA は配置配線による遅延の影響が ASIC と比べて大きく，回路の配置領域が大きくなる傾向にあり，それに伴い配線距離も長くなることで，配線遅延が増大する問題があった．そのため，回路の配置配線を巧妙に行うことで，配線長を低減し，高性能化を図る必要があり，これまで研究 [11][12] されている．

本研究では，DDP の動作保証と性能向上を目的とし，設計自動化フロー [8] を拡張した，データ転送制御回路に着目した回路最適化手法について検討する．

第 2 章では，DDP のアーキテクチャとこれに関する先行研究についてまとめ，先行研究

で発生している設計最適化に関する課題を提起する．

第 3 章では，第 2 章で述べた課題を解決するため，設計最適化の手法とこれを踏まえた設計自動化フローの拡張について述べる．データ転送制御回路間のタイミング調整法および先行研究の内容を応用した複合データ転送制御回路のタイミング調整法について提案を行い，回路の動作保証に関する検討を行う．また，配置配線の最適化により，DDP の性能向上に関する検討も行う．そして，検討内容を組み込んだ設計自動化フローの拡張についてまとめる．

第 4 章では，DDP を提案手法を用いて設計した場合と従来手法 [8] を用いて DDP を設計した場合で設計結果を比較，評価を行う．評価には Intel 社 MAX10 FPGA を，回路設計ツールには Intel Quartus Prime 18.0 を使用する．

第 5 章では，本研究で実施した内容についてまとめ，今後の課題について述べる．

第 2 章

DDP の設計最適化の課題

2.1 緒言

本章では，データ駆動型プロセッサ (DDP) を，FPGA 向けに実装するにあたり前提となる要素技術と課題について述べる．DDP を実現するためのデータ駆動型計算モデル，DDP アーキテクチャ，FPGA の要素技術についてまとめ，DDP の FPGA 向け実装に向けて取り組まれた先行研究について述べる．先行研究については本研究のベースとなる DDP の設計自動化フローに関する研究 [8] と，DDP のタイミング検証法に関する研究 [7][10] について述べる．そして，先行研究で残された課題について提起する．

2.2 データ駆動型計算モデル

DDP はデータ駆動型計算モデルによって実現されるプロセッサである．データ駆動型計算モデルは，図 2.1 に示すように，それぞれ命令が入ったノードの実行に必要なトークンが到着することで，実行可能な状態になる．実行可能になったノードはトークンの組を入力として演算を実行 (発火) し，演算結果を宛先に従って次に実行する命令の引数として出力される．このようにデータ駆動型計算モデルでは，各ノード間でトークンのやりとりが行われ，演算の実行順序はこれに依存する．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る．図 2.2 で (A+B)-(C+D) を計算するデータフローグラフについて表す．(A+B) を演算するノード，(C+D) を演算するノード，(A+B) と (C+D) の出力を必要とするノードが存在する．(A+B) と (C+D) はそれぞれデータ依存関係が無いため，並列に処理するこ

2.2　データ駆動型計算モデル

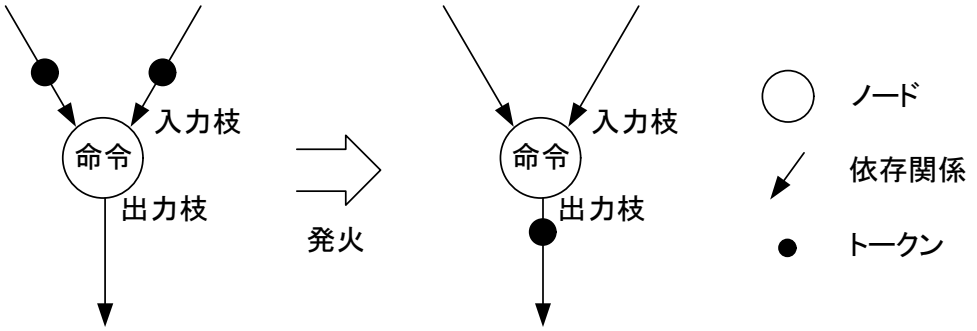


図 2.1　データ駆動計算モデルの動作原理

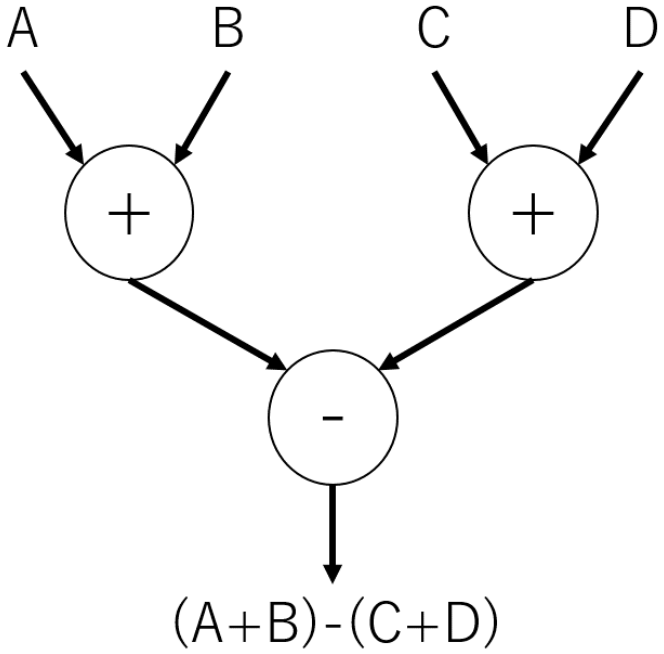


図 2.2　データ駆動計算モデルの動作例

とが可能である．一方，(A+B)-(C+D) のノードが実行可能状態になるためには必要なオペランドが演算完了している必要がある．これを実行規則に従うことで，データ駆動型計算モデルはプログラムの解釈・実行を行っている．

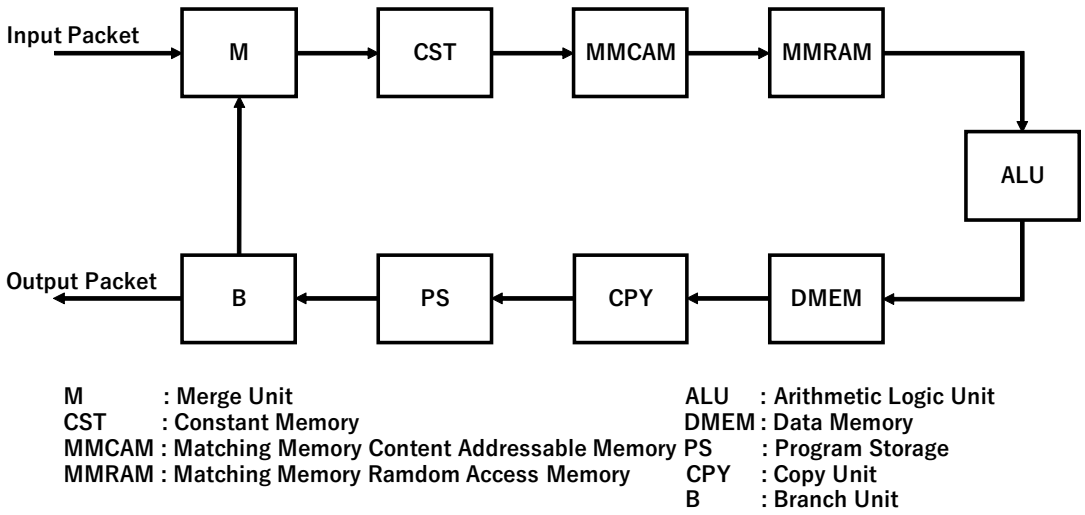


図 2.3　DDP のパイプラインステージ

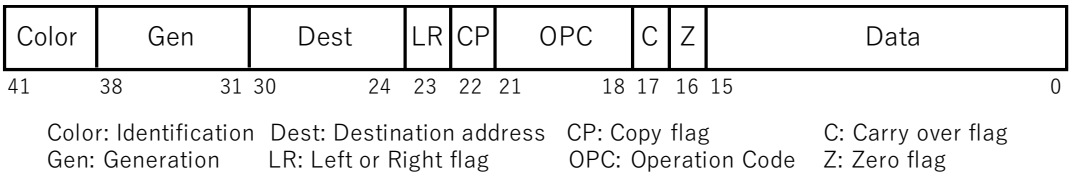


図 2.4　DDP の入力パケットフォーマット

2.3　DDP アーキテクチャと回路実現法

2.2 で示したモデルによって DDP が構成され，トークンはパケットとして実現される．本研究で対象とする DDP では図 2.3 に示すように，9 つのステージと 11 個の DL によって STP で構成されており，パケットレベルでデータ駆動型計算モデルの動作が忠実に実現される．DDP には図 2.4，表 2.1 に示されるパケットが入力され，パイプラインを周回する中でパケットに応じた処理が各ステージで実行され，処理が終わったら DDP の外へと出力される．DDP の演算には，単項演算と多項演算の二つがあり，多項演算ではペアとなるパケットが MMCAM で揃うことで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．これにより，低消費電力性と高性能を実現している．以下に DDP を構成する各ステージの詳細を示す．

表 2.1 DDP のパケット情報

フィールド名	名称	情報
color	Color	タスクの識別情報
gen	Generation	タスクの世代 (順番)
dest	Destination	パケットの宛先
LR	Left or Right	パケットの左右の識別
CP	Copy flag	コピーの有無
opc	Operation code	命令コード
C	Carry flag	キャリーフラグ
Z	Zero flag	ゼロフラグ
data	Data	演算データ

- パケット合流ステージ (Merge Stage)

  外部から入力されたパケットとパイプライン内を周回したパケットの合流を調停し，CST にパケットを出力する．

- 定数読み出しステージ (Constant Memory: CST)

  定数命令を実行する際に必要となる定数とオペレーションコードが格納されている．パケットが定数命令を保持する場合，パケットが持つ宛先情報から対応する定数とオペレーションコードを定数メモリの ROM(Read Only Memory) から読み出し，パケットに付加する．

- パケット連想メモリ (Matching Memory Content Addressable Memory: MMCAM)

  CST からパケットを受け取り，対応するパケットの待ち合わせを行うため，ペアとなるパケット情報の一時格納を行う．内部に連想記憶を行うレジスタが確保されており，対応するペアのパケットが入力されると発火信号が出力され，入力パケットと連想記憶の情報が MMRAM に出力される．定数命令はパケットの待ち合わせを行う必要が無いため，そのまま MMRAM に出力する．

- パケット待ち合わせステージ (Matching Memory Random Access Memory: MMRAM)

  パケット待ち合わせにおけるパケットデータの保持を行う．MMCAM からペアとなるパケットの連想記憶情報が入力されると，対応するデータを MMRAM(Random Access Memory) 内のメモリから読み出し，入力パケットに付加する．一方，MMCAM で対応するペアの連想記憶が無かった場合はパケットがメモリに保持され，ペアとなるパケット情報が届くまで待機する．定数命令など，パケットの待ち合わせが必要ない場合，入力パケットをそのまま ALU に出力する．

- 演算ステージ (Arithmatic Logic Unit: ALU)

  パケットが保持しているオペレーションコードを解読し，算術演算や論理演算，シフト命令等を実行する．また，ロード命令，ストア命令のような Data Memory にアクセスする命令では，アドレスの算出を行う．

- データメモリ (Data Memory: DMEM)

  ALU から受け取ったロード・ストア命令により，データの読み出し，書き込みを行う．ロード命令の場合は RAM から読み出したデータをパケットに付加し，ストア命令の場合はパケットのデータを RAM に書き込む．

- パケットコピーステージ (Copy Unit: CPY)

  入力パケットのコピーフラグを読み取り，フラグが有効であれば，入力パケットの複製を行う．

- 命令フェッチステージ (Program Storage: PS)

  命令が格納された ROM によって構成され，次のパイプライン周回で実行するオペレーションコードと宛先ノード番号が格納されている．入力パケットの宛先ノード番号から ROM の内容を参照することにより，入力パケットのオペレーションコードと新たな宛先ノード番号を書き換え，次のパイプライン周回時に命令の実行を行う．また，削除命令がパケットに付加された場合，パケットは削除される．

- 分岐ステージ (Branch: B)

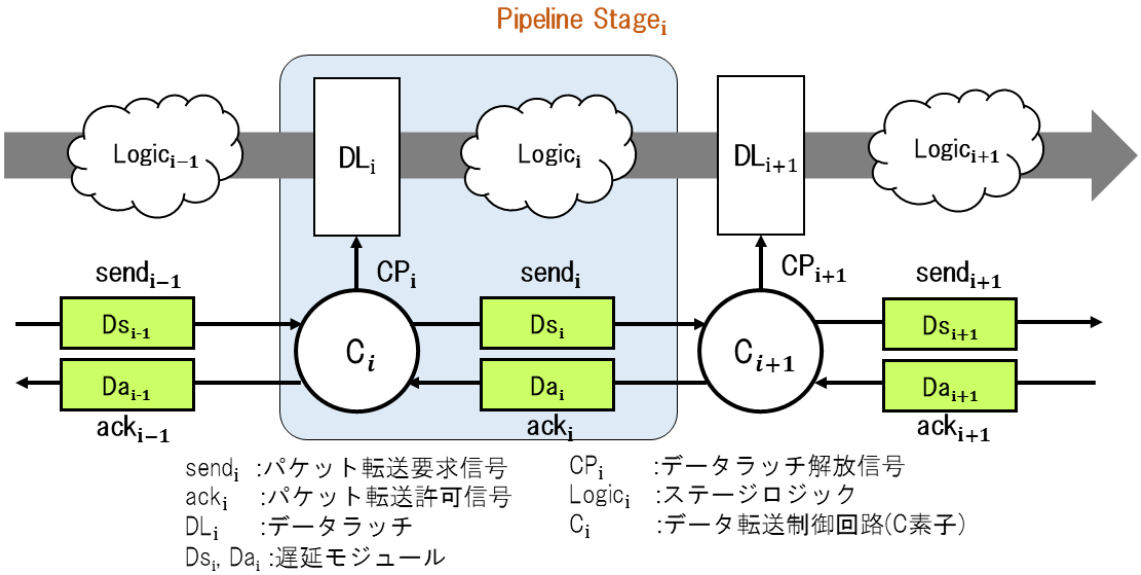


図 2.5　セルフタイム型パイプラインの構成図

パケットが保持する情報から，パケットを外部に出力するか再度パイプラインを周回するかを判断する．

DDP はセルフタイム型パイプライン (STP:Self-Timed-Pipeline) で構成されている．STP は図 2.5 のように複数のデータ転送制御回路ー C 素子 (Coincidence flip-flop) によって構成される．隣接するパイプラインステージ間で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) がシェイクハンド通信することにより，データラッチ開放信号 (CP 信号) が制御される．この信号を DL(Data Latch) が受信することにより，各パイプラインステージの Logic 回路にデータが転送され，各 Logic 回路で信号処理が行われる．これにより，隣接するパイプラインステージ間の回路のみが部分的に駆動し，必要な回路のみが動作する性質から，回路の低消費電力特性を実現している．

セルフタイム型パイプラインは図 2.6 のタイミングチャートに基づいて動作が行われる．前段の 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 T_f と呼ばれる．T_f は前段の DL から Logic 回路を通り，後段の DL までの最長経路（クリティカルパス）の遅延時間よりも長い必要があり，これをセットアップタイム制約と呼ぶ．セットアップタイム制約を満たしていない場合，Logic 回路の処理が完了しない段階で DL にデータが

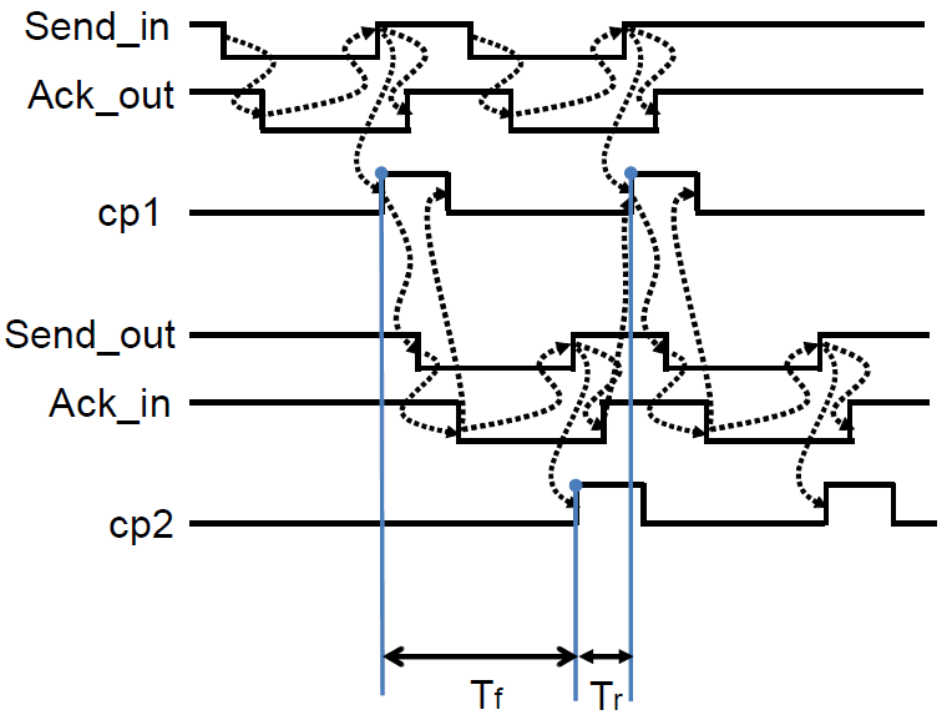


図 2.6　セルフタイム型パイプラインのタイミングチャート

取り込まれてしまい，正しいデータ処理が行われないことになる．そのため，図 2.5 の遅延回路 (Ds:Delay-send) を設計し，T_f 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，再度前段の CP 信号が立ち上がるまでの時間を T_r と呼び，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りも長い必要がある．これをホールドタイム制約と呼び，セットアップタイム制約と同様に回路を設計する上で考慮する必要がある．図 2.5 の場合，遅延回路 (Da:Delay-ack) によって，T_r を調整する．各ステージごとに Logic で必要となる回路規模が違うため，セットアップタイム制約およびホールドタイム制約もそれぞれ異なり，最終的なパケット転送時間も変わる．

以上の要素から，DDP のアーキテクチャが設計され，STP によって回路が実現されている．

2.4 FPGA

FPGA はユーザが手元で回路情報を変更できる点が大きな特徴であり，論理要素，入出力要素，配線要素の三つで構成されている．図 2.7 に FPGA 回路の一例として，Intel 社 MAX10 シリーズの回路図を示す．論理要素は LUT(Look Up Table) と FF(Flip Flop) から構成されており，これらが対になった LE(Logic Element) という素子で構成され，一定の個数の LE がまとまることで LAB(Logic Array Block) となる．LUT は 3 つないし 4 つの入力と 1 つの出力による多数決回路となっており，真理値表で表すことができる．ハードウェア記述言語で論理式を与えると，LUT の真理値表の情報が書き換わり，1 つの論理回路として扱うことが可能となる．FF は 1 つの LE に対して 1bit 保持することできるため，複数の LE を組み合わせることで任意の bit 幅を持つレジスタとして扱うことができる．また他の論理要素として，演算処理に特化した DSP（Degital Signal Processor）Block と，大量のデータを保持できる 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．FPGA の論理要素は LE 間を配線要素で接続することで LE 同士を自由に接続することが可能となり，この組み合わせによって，任意の論理回路を柔軟に設計することができる [13].

FPGA 回路の合成を行うと設計ツールによって自動的に LE が割り当てられ，回路が構成される．このとき，FPGA 回路の配置配線も同時に行われるが，LE 間の距離が離れているほど遅延が生じてしまう問題がある．これを解決するために，フロアプランと呼ばれる回路のレイアウト指定を行うことにより，使用するチップ面積の最小化や，配線による遅延時間の短縮が可能となる．

以上の要素から，ユーザは必要に応じて回路情報を書き換えられる汎用性があり，開発コストが抑えられることから，DDP の実装対象として FPGA を採用することは有効である．しかし，このためには解決すべき課題がいくつかあり，後述する．

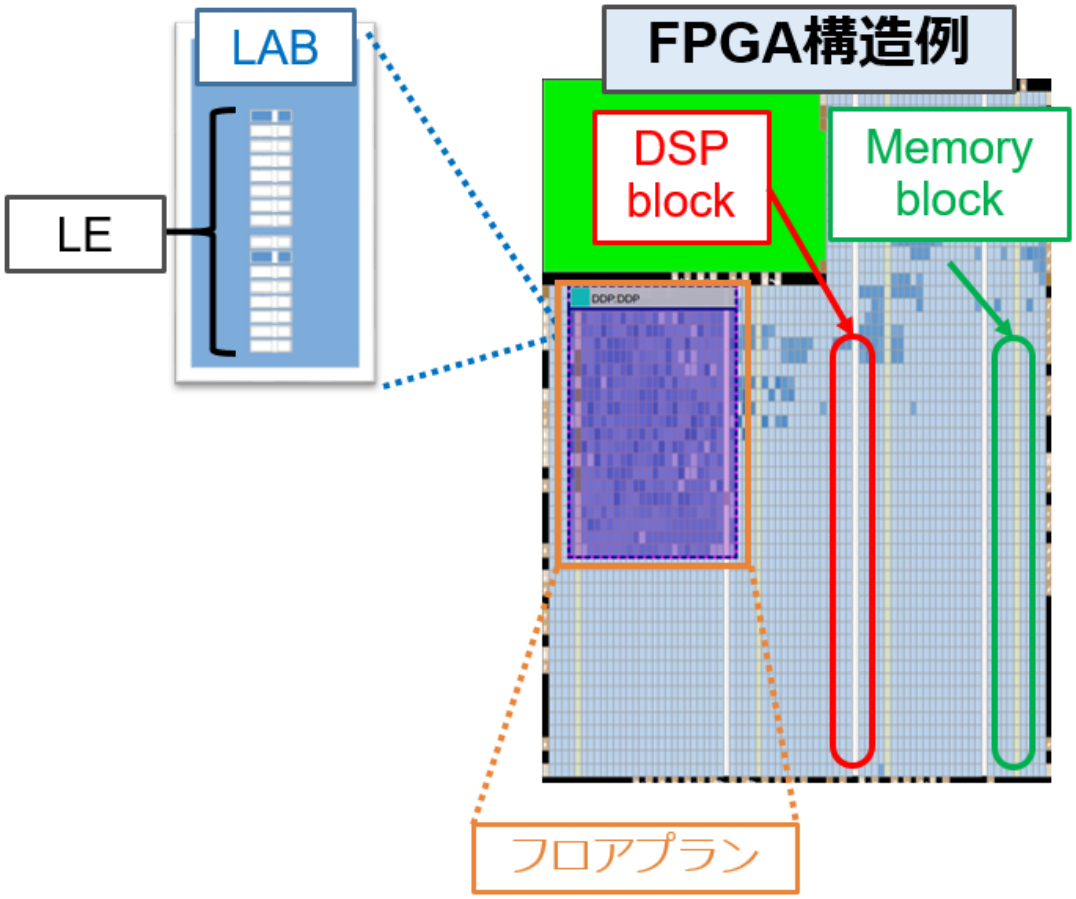


図 2.7　FPGA の構造 (Intel 社 MAX10-50[14])

2.5　DDP の設計最適化に対する検討

DDP の設計最適化に対する検討として，DDP の設計自動化フロー [8] がある．本節では，DDP 設計自動化フローとこれに付随するタイミング解析・検証と，フロアプラン最適化の課題について議論する．

2.5.1　DDP の設計自動化フロー

DDP は非同期式回路のため，同期回路に最適化された既存の回路設計ツールでは，タイミング解析・検証をそのまま実施することができず，DDP の動作保証が困難である．そのため，回路設計ツールによって同期式回路と同様に DDP を設計するために，図 2.8 のよう

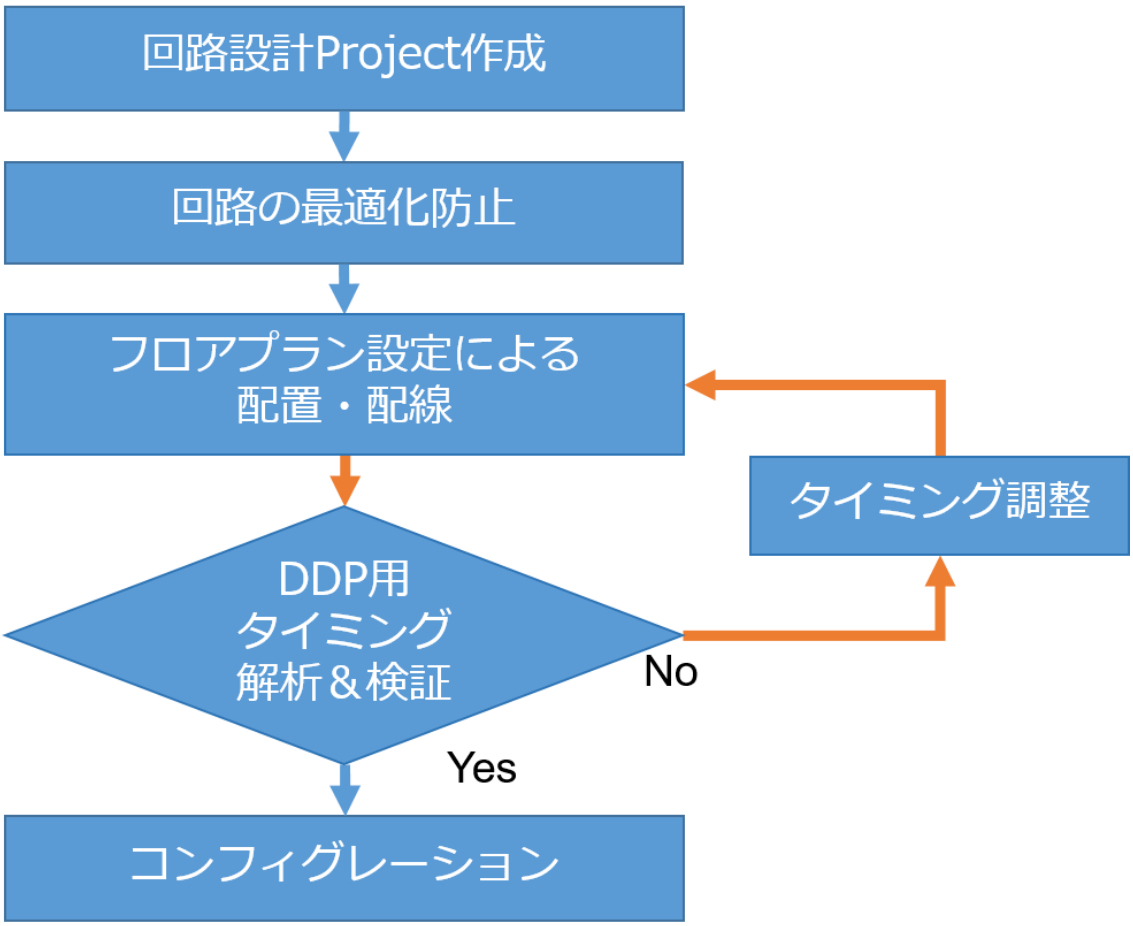


図 2.8　DDP の設計フロー

な設計フローが提案された．図 2.8 によって DDP を設計する際，設計作業を手作業で行うと必要な操作が膨大になり，設計に費やす時間が長くなる．そのため，設計作業を CUI 上で実行可能な環境を整え，コマンドライン上で DDP の設計を行うための手法が提案された．[8] 具体的には，Intel 社 FPGA を対象とした DDP の設計を行うことを想定し，Quartus の GUI 作業に対応した Tcl スクリプトや，2.5.2 のタイミング検証で利用する JSON ファイルを始めとした入出力ファイルの自動生成，DDP の設計を自動化する手法が提案された．設計の自動化は Perl と Python スクリプトを用いて行われ，図 2.8 で示された回路設計プロジェクト作成，最適化防止設定，フロアプラン設定と配置・配線，タイミング解析・検証，コンフィグレーションといった作業が自動化された．

回路設計プロジェクト作成では，デバイスの情報や FPGA チップの入出力信号の抽出，Verilog コードから回路の合成を行う．このプロジェクト作成を行う Tcl スクリプトを設計自動化フローで提案された手法で自動生成する．

最適化防止設定について，回路設計ツールの最適化機能は，論理的に不要と判断した回路を削除する恐れがあり，最適化を防止するために，全ての回路モジュールに最適化防止の設定する必要がある．Intel 社 Quartus の場合はパーティションと呼ばれる機能 [15] がこれにあたり，本研究ではこれを利用する．DDP は回路モジュールが多く，手動で行うと膨大な時間がかかるため，設計自動化フローによってパーティション設定を行うための Tcl スクリプトを自動生成する．

フロアプラン設定について，2.4 で説明されたフロアプランを行うための設定を行う．このフロアプラン設定によって，より最適化された配置・配線が可能となり，DDP の性能向上を行うことができる．フロアプランは Quartus の Region 設定機能を利用し，自動配置機能を用いるものとし，を行う Tcl スクリプトも，設計自動化フローによって自動生成する．

タイミング解析・検証について，設計自動化フローを通して SDC(制約) ファイルと回路情報ファイルの生成が行われるため，これを用いてタイミング解析を行う．タイミング解析も上で述べた方法と同様に，Tcl スクリプトによって自動化する．また，2.5.2 のタイミング検証もこの時点で行われるため，検証に必要な JSON ファイルを始めとする入力ファイルの生成を行う．JSON ファイルおよびタイミング解析結果を入力とし，タイミング検証を行い，設計した DDP がタイミング制約を満たしているかを判断する．

以上が DDP 設計自動化フローで提案された手法である．

2.5.2　データ転送制御回路のタイミング検証法

2.3 でも述べたように，DDP が正常に動作するには遅延回路の設計によって，T_f と T_r の時間を調整する必要がある．しかし，単純に遅延回路の遅延量を過剰に増やすだけでは，回路の動作が遅くなりスループットの低下を招く．そのため，各ステージのクリティカルパスに対応した，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させ

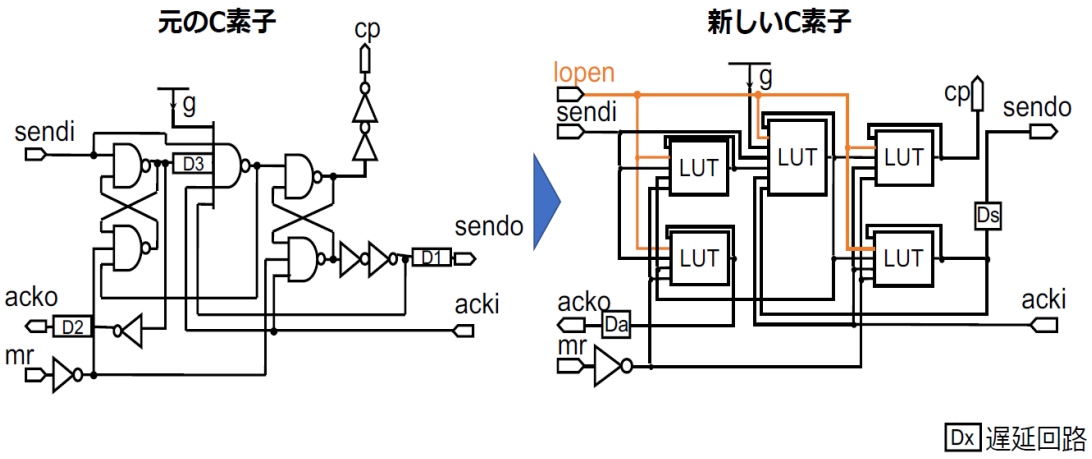


図 2.9　従来の C 素子（左）と新しい C 素子（CCORE）[7]（右）

る必要がある．最適な遅延回路を設計するには Logic 回路のクリティカルパスの経路や，T_f と T_r のタイミング情報を正確に求める必要があり，これによる遅延時間の算出手法が提案されている．[7]

従来の C 素子は図 2.9 の左側のアーキテクチャの様に NAND ゲートラッチを組み合わせることにより構成されていた．既存の回路設計ツールではこの C 素子内部のタイミング情報が抽出できず，C 素子の T_f と T_r がタイミング制約を満たしているかを確認できなかった．図 2.9 の右側のアーキテクチャで構成される C 素子（以下 CCORE）は，NAND ゲートラッチを FPGA 上の組み合わせ回路の構成単位である LUT(Look Up Table) に置換し，疑似的な SR- FlipFlop を構成している．SR-FlipFlop には，回路記述で本来不要なラッチ制御信号（lopen）を追加することで回路設計ツールに LUT をレジスタとして認識させ，タイミング解析を可能にした．

CCORE の T_f と T_r のタイミング情報を抽出するにあたり，CP 信号を出力している LUT（図 2.9 の右図の右上の LUT）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 回路のクリティカルパスのタイミング情報を抽出できる．これにより，JSON 形式による CCORE 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．これにより，C

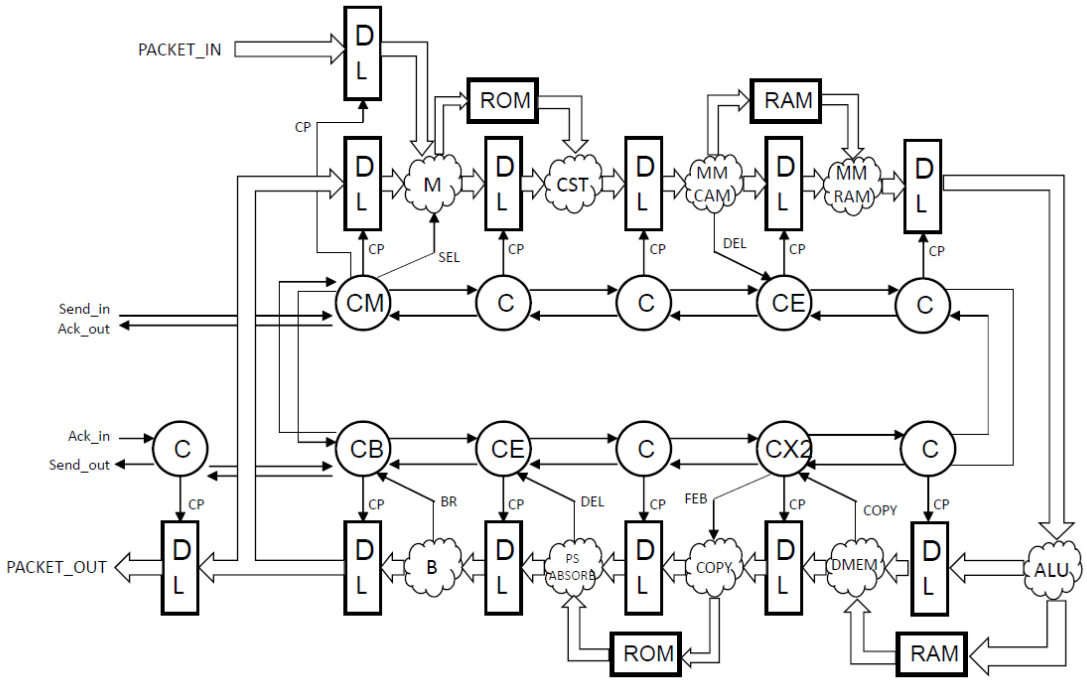


図 2.10　DDP 全体の構成図

素子のタイミング検証が可能となり，最適な遅延量を持つ遅延回路が設計可能となる．

2.5.3　複合データ転送制御回路のタイミング検証法

DDP は，図 2.10 に示すように，データの消去や複製，パイプラインの分流及び合流といった機能を持つ，複合データ転送制御回路（CM，CE，CX2，CB）を通常の C 素子とともに構成することで設計される．複合データ転送制御回路は，通常 C 素子に論理ゲートやラッチを追加することで設計されている．この論理ゲートやラッチでも固有のタイミング制約条件が存在しており，2.5.2 で提案されているタイミング検証手法では動作保証が十分にできない．そのため，これらの複合データ転送制御回路固有のタイミング検証を行うために，回路構成を改変する必要が生じた．

各複合データ転送制御回路が持つ固有の経路のうち，複合制御を行うためのクリティカルな制約条件を持つ経路を調べ，その経路が含まれる論理ゲート・ラッチを CCORE と同様

に LUT に変換する手法が部分的に提案されている [10]．これにより複合データ転送制御回路が持つ固有の制約条件が検証可能となった．

2.5.4　フロアプラン最適化に関する検討

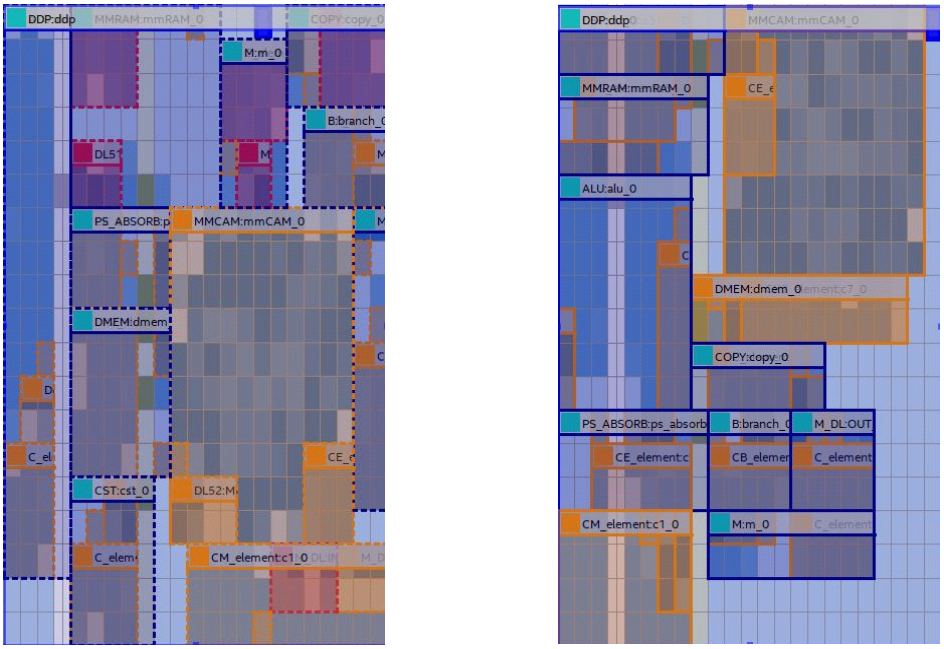
図 2.8 の配置配線の項目について，DDP のフロアプラン最適化の検討 [12] が提案されている．従来の DDP 設計では回路設計ツールによる自動配置や，手作業で Region 設定することでフロアプランを行っていたが，提案手法ではヒューリスティックな配置配線によるフロアプラン設計アルゴリズムを作成することで，DDP を実装する際のフロアプランを設計者に依存せずに設計することを目指した．この手法では，FPGA の回路資源量と配置領域から，DDP の各ステージの座標 (X, Y) および領域 (Width, Height) の決定，DDP のステージ間の関連性を考慮した最適化を行うアルゴリズムを提案した．アルゴリズムは各ステージの初期配置と初期領域の決定，初期配置から各ステージごとの関連性と LAB 数を考慮した配置調整と領域調整によって行われ，最終的な DDP 全体の Width, Height を出力する．

図 2.11 に提案手法によるフロアプランの結果を示す．Quartus の自動配置機能によるフロアプランと提案手法を用いて設計されたフロアプランで比較を行った結果，回路面積の維持を確認した．ただし，Quartus の自動配置機能によるフロアプランでは，DDP の各ステージ間の配線遅延を考慮できていないため，DDP の回路全体の性能向上には繋がらず，提案手法のような DDP 独自のフロアプラン設計手法の検討が重要であると評価する．2.5.1 では配置配線の最適化を行っていないことから，本手法を設計自動化フローに導入することで配線遅延を低減し，DDP 全体の性能向上を図ることができると考えられる．

2.6　DDP の設計最適化における課題

以上より，DDP の設計最適化に対する検討が行われてきた．しかし，2.3 で提案された DDP の設計自動化フローにはいくつかの課題が存在する．

まず，図 2.8 の設計自動化フローにおいて，タイミング調整が設計者の試行錯誤によって



(a)Quartus の自動配置機能による　　(b) 提案手法を用いた設計による

フロアプラン　　フロアプラン

図 2.11　提案されたフロアプラン最適化手法の比較

実施されていることが挙げられる．具体的には，設計自動化フローにおけるタイミング解析・検証では，タイミング検証結果の出力までが自動化されており，新たに必要な遅延回路の導出手法が確立されていない．そのため，遅延回路の調整作業は現状，設計者が試行錯誤で決定しており，タイミング検証結果が良くなるまで設計自動化フローを何度も繰り返すことで，DDP の最適化が行われている．よって，設計者に依存しないタイミング調整法を新たに提案することで，動作保証された DDP 設計を，より短時間で行うことができる．

次に，2.5.3 で述べた複合データ転送制御回路の制約条件に関する研究 [10] は，従来の設計自動化フローに導入されていないため，タイミング検証が完全でないという問題が存在する．そのため，複合データ転送制御回路の制約条件に関するタイミング検証を，2.5.2 のタイミング検証ツールを拡張することで，設計自動化フローで実行可能にする．また，2.5.3 の時点では，複合データ転送制御回路固有の制約条件に関するタイミング調整も，設計者の試行錯誤で行われているため，これに対応したタイミング調整法も検討する必要がある．

さらに，従来の設計自動化フローの配置配線手法では，DDP 全体を一つの Region として設定し，回路設計ツール (Quartus) の自動配置機能を用いて配置配線が行われていた．しかし，より高性能な DDP を設計するにあたり，DDP の各ステージごとに対応した Region 設定を行う必要があると考えられる．設計自動化フローに DDP の配置配線に関する検討 [12] を踏まえた拡張を行うことで，従来の配置配線手法よりも各回路の配線距離を短くし，配線遅延の削減によって DDP の高性能化を図る．

以上の課題と解決策から，DDP 設計自動化フローを拡張することにより，DDP の性能向上と制約条件の網羅的検証による動作保証を実現する．

2.7 結言

本章では，まず DDP を実現するデータ駆動型モデルと，このモデルに基づく DDP アーキテクチャについて説明し，STP による低消費電力性の実現について述べた．また，DDP の実装対象である FPGA についての説明を行った．次に，DDP は非同期回路によって設計されることから，信頼性の高い DDP を設計するためにはタイミング解析・検証手法の確立を始めとした DDP 独自の設計フローが必要であることについて説明した．設計フローを構成する要素として，FPGA における STP の考え方を基にした C 素子のタイミング解析・検証方法に関する研究 [7] や，これを拡張した複合データ転送制御回路のタイミング検証に関する研究 [10] について説明した．そして，タイミング検証手法を盛り込んだ DDP の設計フローの自動化に関する提案 [8] について説明した．また，DDP における配置配線の最適化手法として，フロアプラン最適化に関する研究 [12] について説明した．最後に，先行研究で生じている設計最適化に関する課題について問題提起し，課題の解決策を提案した．次章からは課題の解決策について詳しく説明していく．

第 3 章

データ転送制御回路に着目した最適化の検討

3.1 緒言

本章では，まずデータ転送制御回路 (C 素子) の回路最適化の方針について述べ，回路最適化の手法について簡単に説明する．タイミング検証を行うにあたり実施した，複合データ転送制御回路の改良についても述べる．その後，データ転送制御回路におけるタイミング最適化で重要な要素となる遅延回路の最小遅延時間の解析手法について説明し，解析結果について述べる．続いて，C 素子間のタイミング調整法，および複合データ転送制御回路の制約条件に関するタイミング調整法についてそれぞれ提案する．また，回路の性能向上を行うためのフロアプラン最適化手法のアルゴリズムについて提案する．最後に，従来の設計自動化フロー [8] では，複合データ転送制御回路のタイミング検証手法やフロアプラン最適化手法に対応していないため，これらを導入した DDP 設計自動化フローの拡張手法について説明する．

3.2 回路最適化の方針

データ転送制御回路の回路最適化は以下の 2 つの手法によって実施する．

1. データ転送制御回路のタイミング調整法の確立
2. フロアプランの最適化によるデータ転送制御回路の遅延時間の低減

3.2　回路最適化の方針

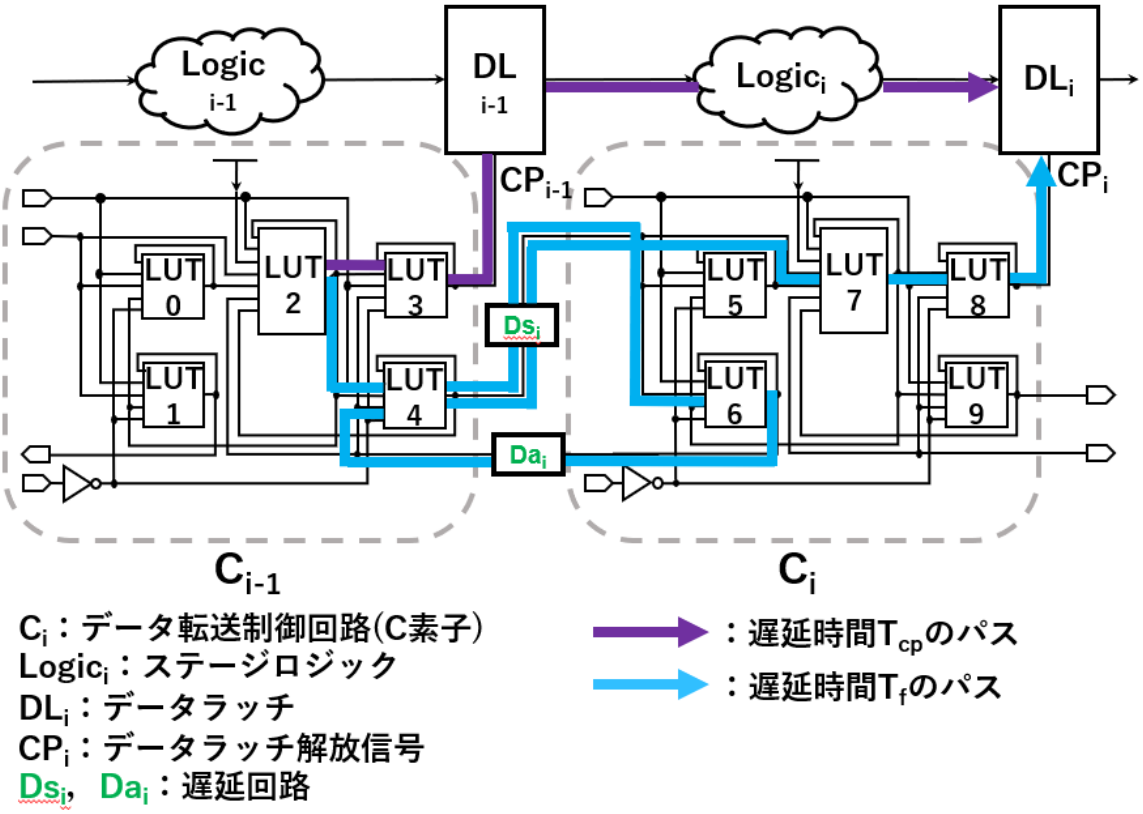


図 3.1　データ転送制御回路の信号パス

データ転送制御回路のタイミング調整は，回路の動作保証が可能かつより少ない個数の遅延回路を設計することで最適化を行う．フロアプランの最適化は，従来手法よりも無駄なく回路資源を利用し，配線遅延を抑えることができる回路配置を導出することで最適化を行う．

タイミング調整は図 3.1 の Ds，Da の遅延回路の個数を調整することで行われるが，2.6 で述べたように，設計者による試行錯誤で行われている．そのため，設計者に依存しない，効率的かつ定量的に行うタイミング調整法を新たに確立することで，設計時間の短縮と DDP の動作保証を図る．

タイミング調整は以下の流れで実施される．

1. 遅延回路の最小遅延時間を解析
2. 解析結果と各経路のタイミング検証結果から，タイミング調整で必要な遅延回路の個数

を推測する

3. 遅延回路の個数を変更し，再度 DDP の回路設計を行い，タイミング検証を行う
4. タイミング調整が完了するまでこれを繰り返す

遅延回路の最小遅延時間を解析することで，データ転送制御回路のタイミング調整および，複合データ転送制御回路固有の制約条件のタイミング調整を行う．複合データ転送制御回路については，タイミング情報抽出のために回路構成の改良も行う．

また 2.6 で提起したように，従来の設計自動化フローではフロアプラン設計を回路設計ツールの自動配置機能により実施していた．しかし，自動配置機能による配置配線では，回路の配置領域が大きくなる傾向にあり，それに伴い配線距離も長くなり，配線遅延が増大する問題があった．これを解決すべく，2.5.4 のフロアプラン最適化手法を用いた設計により配置配線を最適化し，DDP の性能向上を図る必要がある．図 3.1 の遅延時間 T_{cp} と遅延時間 T_f のパスは，遅延回路 Ds, Da だけでなく，配置配線による配線遅延の影響も受ける．このことから，配置配線面積の縮小を行うことは，T_{cp} と T_f で必要な配線遅延を低減することに繋がる．T_{cp} の配線遅延を低減することで，タイミング調整法で決定される T_f も低減することができ，最終的に DDP の性能向上に繋げられる．

3.3　複合データ転送制御回路の改良

複合データ転送制御回路のタイミング検証を行うにあたり，タイミング情報を抽出可能な回路構成に改良する必要がある．本節では複合データ転送制御回路（CM，CE，CX2，CB）の回路構成の改良について説明する．

3.3.1　データ転送合流制御 (CM) 回路の改良

CM 回路の回路図を図 3.2 に示す．

CM 回路では，前段に 2 つのデータ転送制御回路，後段に 1 つのデータ転送制御回路が接続されており，パケットの合流調停を行うことで，先の到着したパケットから順に後段の

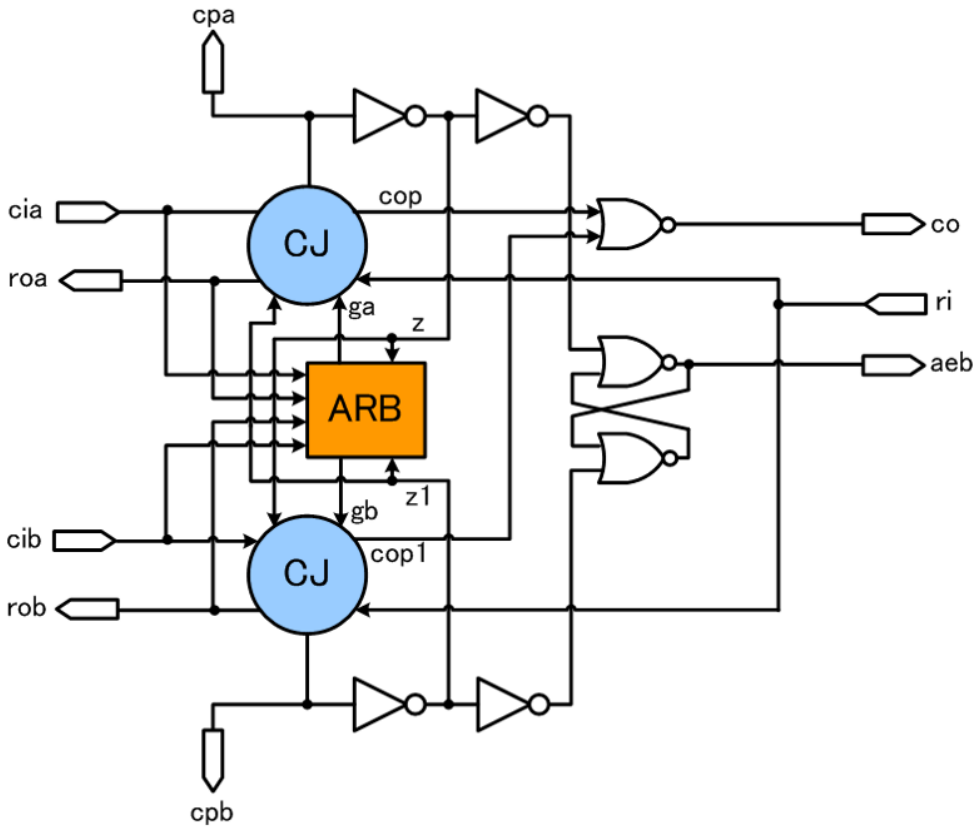


図 3.2　CM 回路の回路構成

データ転送制御回路に転送する．CM 回路は 2 つの CJ 回路と呼ばれる通常 C 素子を元にした回路と，ARB と呼ばれる調停回路から構成される．CJ 回路は ARB からの入力を受け付ける機構が特徴で，CM 回路内部の NOR ゲートで処理を可能にするため，出力が反転している．CM 回路固有のタイミング制約条件は大きく 2 つあり，1 つは前段データ転送制御回路との間の制約条件，もう 1 つは出力信号の 1 つである arb 信号に接続する MUX に関する制約条件である．

タイミング検証に伴う CM 素子の改良について，CJ 回路は CJCORE に置き換えることで，2.5.2 のタイミング検証手法を利用して従来の経路を検証可能にした．前段データ転送制御回路との制約条件は前段の CCORE を用いることでタイミング検証が可能である．arb 信号に接続する MUX に関する制約条件は, 周辺回路の拡張が必要となる．設計した回路を図 3.3 に示す．

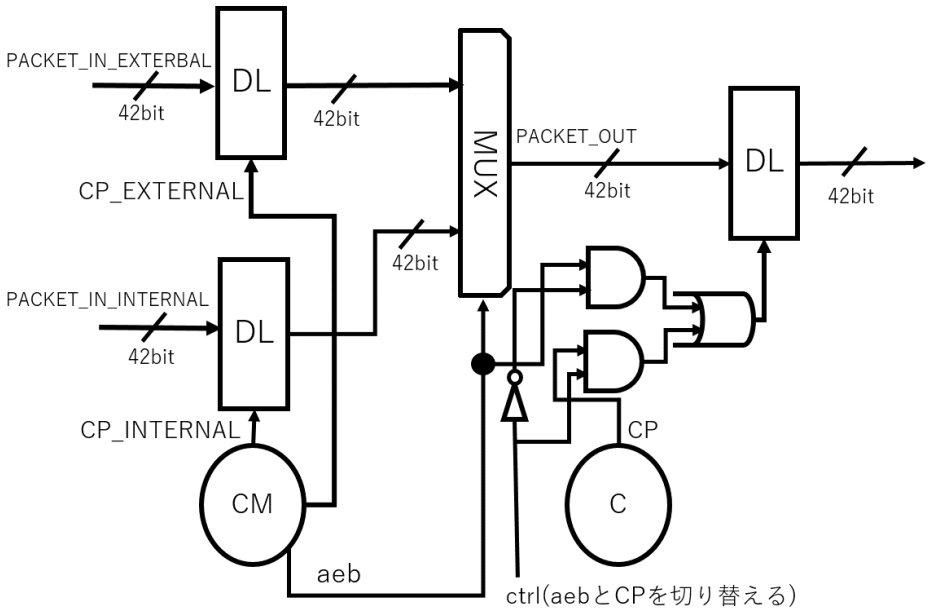


図 3.3　aeb 信号の検証のため拡張した CM の周辺回路

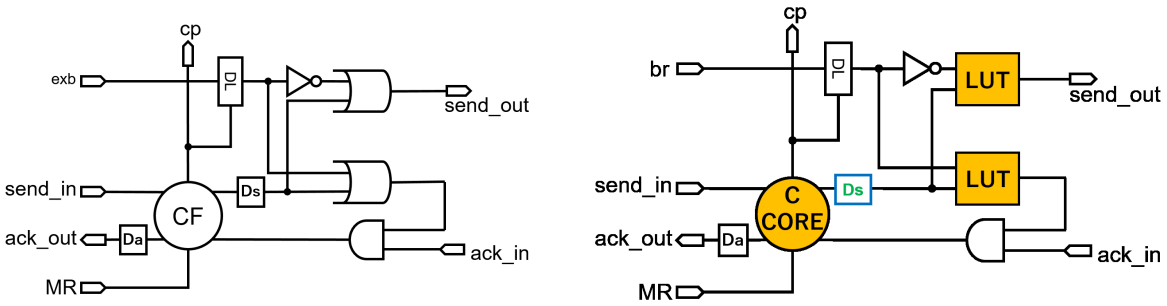
aeb 信号は 2 つのパケットを合流させるための MUX 制御信号として利用する信号である．CM 回路から MUX の経路でタイミング制約条件が存在するため，検証する必要があるが，MUX は順序回路でないため，タイミング検証ができなかった．そのため，タイミング検証時のみ MUX の後段の DL に aeb を接続することで，タイミング検証を可能にした．後段 DL の CP 信号と aeb 信号の切り替えは，新たに ctrl 信号を設け，AND ゲートと OR ゲートを用いて回路設計することで，DDP の機能に影響が出ないように工夫した．これにより，CM 回路のタイミング制約条件を検証可能にした．

2.5.2 のタイミング検証ツールを拡張することで，CM 回路特有のタイミング制約条件の検証が可能となる．

3.3.2　データ転送削除制御 (CE) 回路の改良

CE 回路の回路図を図 3.4 の (a) に示す．

CE 回路では，通常のデータ転送制御に加え，前段 DL から転送されるパケットに付加した制御信号 (exb 信号) により，パケットの削除を行う．CE 回路は，CF 回路と呼ばれる通



(a) 従来の CE 回路の構成図　　(b) 改良した CE 回路の構成図

図 3.4　CE 回路の回路構成

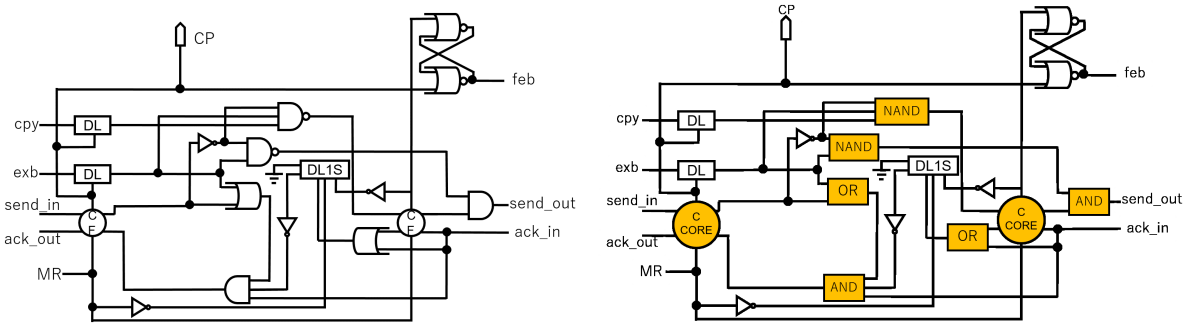
常 C 素子と同様の役割を果たす回路と，パケットの削除を実現するための 2 つの OR ゲートから構成される．exb 信号は後段のデータ転送制御回路に転送させるかを判断するため，どちらの OR ゲートに send_out 信号を出力するかを決定する．パケットを転送しない場合は，ack_in 信号と接続している AND ゲートに send_out 信号を送り，疑似的に ack_in 信号を出し，パケットの削除を実現している．そのため，send_out 信号が OR ゲートに伝達されるよりも早く exb 信号が OR ゲートに伝達される必要がある．

改良した CE 回路を図 3.4 の (b) に示す．CF 回路は CCORE に置き換えることで，2.5.2 のタイミング検証手法を利用して従来の経路を検証可能にした．OR ゲートも同様に疑似クロック付き LUT に置き換えることで，exb 信号による制御信号の遅延時間を計測可能にした．これにより，2.5.2 のタイミング検証ツールを拡張することで，exb 信号によるタイミング制約条件の検証が可能となる．

3.3.3　データ転送複製制御 (CX2) 回路の改良

CX2 回路の回路図を図 3.5 の (a) に示す．

CX2 回路では，通常のデータ転送制御に加え，前段 DL から転送されるパケットに付加した制御信号 (exb 信号, cpy 信号) により，パケットの削除と複製が可能である．CX2 回路は，2 つの CF 回路と，複数の論理ゲートを組み合わせることで構成される．パケットの削除を行う場合は 1 つ目の CF 回路からの send_out 信号を CX2 内部で処理し，後段のデー



(a) 従来の CX2 回路の構成図　　(b) 改良した CX2 回路の構成図

図 3.5　CX2 回路の回路構成

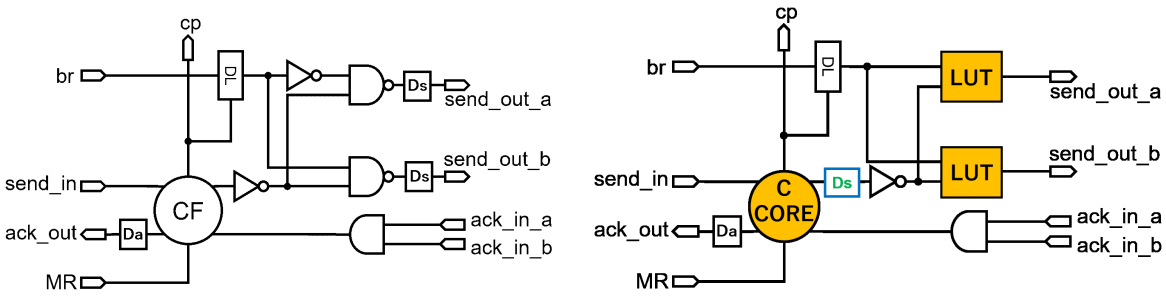
タ転送制御回路への転送制御通信を防ぐ．パケットの複製を行う場合は，CX2 内部の一時記憶回路 DL1S に send_out 信号を保存し，後段のデータ転送制御回路がデータ転送を追えた後に再度 send_out 信号を送信し，パケットの複製を行う．cpy 信号，exb 信号はパケットの複製機能，削除機能を制御する信号であり，CX2 内の DL に格納され，CP 信号に同期して出力される．CX2 回路の動作は，通常機能（複製なし，削除なし），複製機能，削除機能の大きく 3 つに分けられ，それぞれの機能を実現するために各論理ゲートにタイミング制約条件が存在する．そのため，各論理ゲートに対してタイミング検証を行う必要がある．

改良した CX2 回路を図 3.5 の (b) に示す．CX2 回路の機能を実現する論理ゲートも CCORE と同様に疑似クロック付き LUT に置き換え，cpy 信号および exb 信号による制御信号の遅延時間を計測可能にした．

3.3.4　データ転送分流制御 (CB) 回路の改良

CB 回路の回路図を図 3.6 の (a) に示す．

CB 回路では，前段に 1 つのデータ転送制御回路，後段に 2 つのデータ転送制御回路に接続されており，前段 DL から転送されるパケットに付加した分流方向を示す信号 (br 信号) により，パケットを send_out_a，send_out_b のどちらに転送するかを決定する．後段のデータ転送制御回路から返ってきた ack_in 信号は AND ゲートで受け取り，2 つのうちどちらの回路からも信号処理可能となる．CB 回路は，CF 回路と分流を実現するための 2



(a) 従来の CB 回路の構成図　　(b) 改良した CB 回路の構成図

図 3.6　CB 回路の回路構成

つの NAND ゲートから構成される．br 信号は後段のデータ転送制御回路と信号処理を行うにあたり，どちらの NAND ゲートに send_out 信号を出力するかを決定する．そのため，send_out 信号が NAND ゲートに伝達されるよりも早く br 信号が NAND ゲートに伝達される必要がある．

改良した CB 回路を図 3.6 の (b) に示す．CF 回路は CCORE に置き換えることで，2.5.2 のタイミング検証手法を利用して従来の経路を検証可能にした．NAND ゲートも同様に疑似クロック付き LUT に置き換えることで，br 信号による制御信号の遅延時間を計測可能にした．これにより，2.5.2 のタイミング検証ツールを拡張することで，br 信号によるタイミング制約条件の検証が可能となる．

3.4　遅延回路の最小遅延時間の解析

タイミング調整で利用するため，遅延回路の最小遅延時間の解析について説明する．最小遅延時間の解析手法について述べたのち，解析結果について説明する．

3.4.1　最小遅延時間の解析手法

DDP とは別に新規の回路設計プロジェクトを作成し，図 3.7 に示す回路を設計する．回路は 1bit の入力レジスタと出力レジスタ，後述する個数 N_d の遅延回路（LCELL）からな

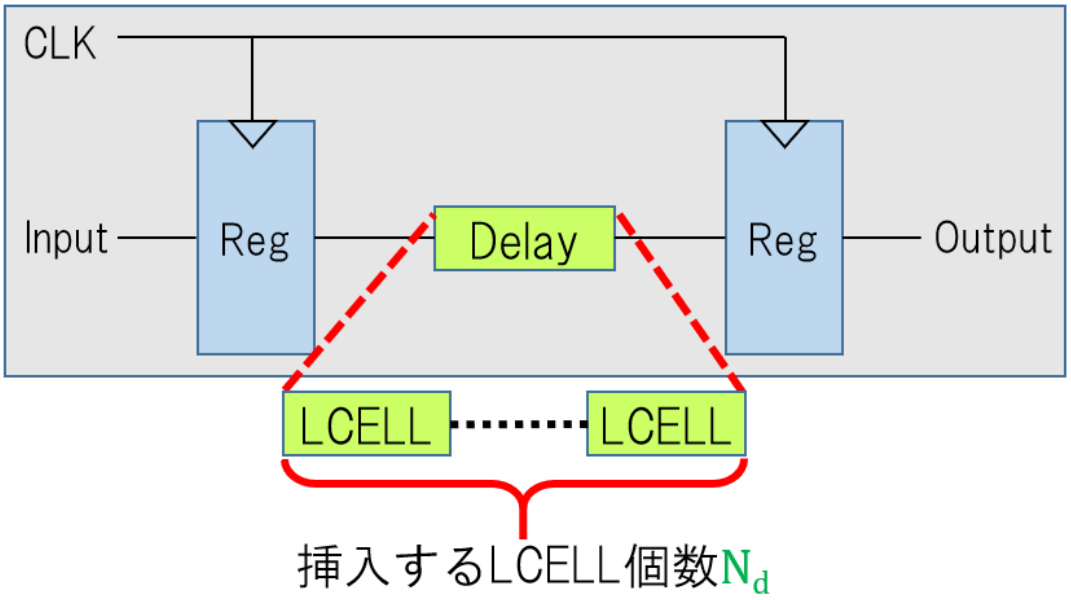


図 3.7　最小遅延時間を解析するための回路構成

る．LCELL の個数 N_d は，DDP の実装対象となる FPGA の LAB に含まれる LE の個数からレジスタの個数 2 を引いた個数とする．LCELL を複数個挿入することで，タイミング解析を行う際の配線遅延による影響をなるべく無くし，LCELL による遅延のみを取り出す．また，回路設計時は 1 個の LAB に回路がまとまるよう配置配線を行い，LAB 間の配線遅延の影響を無くす．また，タイミング解析時に利用するクロック信号およびタイミング検証モデルは，DDP のタイミング検証で利用する条件と一致させる．

以上から，最小遅延時間を解析するための計算式を示す．

$$LCELL1\text{ 個あたり遅延時間 }T_{min} = \frac{\text{遅延時間 } T_d}{LCELL\text{ 個数 } N_d + 1}$$

タイミング解析はレジスタの出力ポート間を解析範囲としているため，LCELL による遅延時間と出力レジスタ内の遅延時間の和が T_d に含まれる．このため，LCELL1 個あたりの遅延時間 T_{min} は，LCELL 個数 N_d に 1 を足した値で T_d を割ることで導出される．

3.4.2　解析結果

3.4.1 の手法を用いて最小遅延時間の解析を実施した結果について述べる．DDP の実装対象は Intel 社 MAX10 FPGA と想定し，50MHz のクロック信号で動作するものとする．また，LAB に含まれる LE 数は 16 個である．レジスタで利用する LE を除くと，$N_d = 16 - 2 = 14$ で，LCELL 個数は 14 個である．タイミング解析モデルは “Slow model 1200mv 0C” を利用した．これらの条件によって，回路のタイミング解析を行った．解析結果を表 3.1 に示す．

LCELL の最小遅延時間は $T_{min} = \frac{5.347}{15} = 0.3564$ となった．この解析結果から，データ転送制御回路のタイミング調整を行う．

表 3.1　最小遅延時間の解析結果

T_d	[ns]	5.347
N_d	[個]	14
T_{min}	[ns]	0.3564

3.5　データ転送制御回路間のタイミング調整

データ転送制御回路は 2.3 で説明したように，回路間でハンドシェイク通信を行うことでデータ転送を実現している．図 3.1 に示す遅延時間 T_{cp} と T_f のパスにおいて，タイミング制約条件 $T_{cp} < T_f$ を満たさなければ，DDP が正常に動作しない．このため，T_f のパス上に存在する遅延回路 Ds, Da の LCELL 個数 N_{Ds}，N_{Da} を調整することで，制約条件を満たすデータ転送制御回路を設計することができる．

しかし，制約条件を満たすために無条件に LCELL 個数を増やすと，DDP のスループットが低下し，高性能化を実現できない．よって，制約条件を満たす範囲で遅延回路 Ds，Da を最小化し，DDP の動作保証をした上で性能向上を図る必要がある．

各データ転送制御回路は，以下の手順でタイミング調整を行う．

1. データ転送制御回路の LCELL の個数 N_{Ds}, N_{Da} を初期値に設定し，DDP のコンパイル，タイミング検証を行い，パス遅延時間 T_f を求める．
2. タイミング制約 $(T_{cp} < T_f)$ の充足を確認し，未充足の場合，充足に必要な LCELL の個数 N'_{Ds} を求める
3. 新たな LCELL 個数 N'_{Ds} でコンパイルとタイミング検証を行い，全ての経路でタイミング制約を充足するまで，2 を繰り返す．

手順 1 について，LCELL 個数 N_{Ds},N_{Da} の初期値は 1 個とする．これは，最小構成の遅延回路で DDP をコンパイルすることで，最もデータ転送時間の少ない場合のデータ転送制御回路のタイミング検証を行うためである．なお，LCELL 初期値を 0 個とした場合は，回路設計ツールが遅延回路を見つけられないことでエラーが出力されるため，LCELL を少なくとも 1 個挿入する必要がある．

手順 2 について，タイミング制約 $(T_{cp} < T_f)$ が満たされていない場合，T_f で不足している遅延時間を求める必要がある．これに際して，以下のような連立方程式が立てられる．

$$\begin{cases} T_f < T_{cp} \cdots 1 \\ T_f = T_{fbasic} + (2N_{Ds} + N_{Da})T_{min} \cdots 2 \end{cases}$$

T_f は，3.4.2 で求めた LCELL の最小遅延時間 T_{min} と T_f パス上で通過する LCELL 個数 N_{Ds} を掛けた遅延時間と，T_f パスそのものが持つ遅延時間 T_{fbasic} によって決まる．T_f のパスは遅延回路 Ds を 2 回，遅延回路 Da を 1 回通過しているため，パス全体で通過する遅延回路の合計は $2N_{Ds} + N_{Da}$ となる．この式を元に，遅延時間を充足するために新たに必要な LCELL 個数 N'_{Ds} を決定したときについて考えると，1 と 2 の数式より遅延時間 T'_f について新たに連立方程式が立てられ，

$$\begin{cases} T_{cp} - T'_f > 0 \cdots 3 \\ T'_f = T_{fbasic} + (2(N_{Ds} + N'_{Ds}) + N_{Da})T_{min} = T_f + 2N'_{Ds}T_{min} \cdots 4 \end{cases}$$

となる．

4 を 3 に代入し，

$$T_{cp} - (T_f + 2N'_{Ds}T_{min}) > 0 \cdots 5$$

5 を N_{Ds} について変形すると，

$$2N'_{Ds}T_{min} < T_{cp} - T_f$$

$$N'_{Ds} < \frac{T_{cp} - T_f}{2T_{min}}$$

$$N'_{Ds} = \lceil \frac{T_{cp} - T_f}{2T_{min}} \rceil \cdots 6$$

となる．以上により，新たに追加する LCELL 個数 N'_{Ds} を決定する数式 6 が導出され，これにより決まった LCELL 個数を新たに遅延回路 Ds に追加する．

一方，回路がタイミング制約条件を満たしている場合，$N'_{Ds} = 0$ とし，LCELL 個数を変更しない．

手順 3 について，全ての経路について遅延回路の修正を行ったのち，再度 DDP のコンパイルとタイミング検証を行い，新たなパス遅延時間を求める．タイミング制約条件が満たされていない経路がある場合は，その経路全ての遅延回路 Ds について手順 2 を行い，LCELL 個数 N'_{Ds} の調整を再度行う．これを繰り返し，タイミングの調整を実施する．

以上より，データ転送制御回路のタイミング調整が行われる．

3.6　複合データ転送制御回路のタイミング調整

複合データ転送制御回路には 2.5.3 で示したように，固有のタイミング制約条件が存在する．タイミング制約条件で使用するパスは，複合制御のための制御信号の伝搬時間 T_{short} と，制約条件を満たすために制御対象信号の伝搬時間 T_{long} の 2 つからなり，$T_{short} < T_{long}$ を満たす必要がある．2.5.3 を例にして考えると，T_{short} は br 信号，T_{long} は send_out 信号に対応する．T_{long} は制約条件を満たすため，T_{long} のパス上にある遅延回路 Dc によってタイミング調整が行われる．

複合データ転送制御回路固有の制約条件を検証するにあたり，DDP で利用する全ての複合データ転送制御回路 (CM, CE, CX2, CB) について，2.5.3 で述べたように，複合制御を行うためのクリティカルな制約条件を持つ経路の調査し，その経路が含まれる論理ゲート・

ラッチを CCORE と同様に LUT に変換する．LUT に変換することで，タイミング検証パスの始点と終点の入力を行うことができる．

複合データ転送制御回路の遅延回路 Dc は，以下の手順でタイミング調整を行う．基本的には，3.5 と同様の手法である．

1. 複合データ転送制御回路の LCELL の個数 N_{Dc} を初期値に設定し，DDP のコンパイル，タイミング検証を行い，パス遅延時間 T_{long} を求める．
2. タイミング制約 ($T_{short} < T_{long}$) の充足を確認し，未充足の場合，充足に必要な LCELL の個数 N'_{Dc} を求める
3. 新たな LCELL 個数 N'_{Dc} でコンパイルとタイミング検証を行い，全ての経路でタイミング制約を充足するまで，2 を繰り返す．

手順 1 について，LCELL 個数 N_{Dc} の初期値は 1 個とする．

手順 2 について，タイミング制約 ($T_{short} < T_{long}$) が満たされていない場合，T_{long} で不足している遅延時間を求める必要がある．これについて，以下のような連立方程式が立てられる．

$$\begin{cases} T_{long} < T_{short} \cdots 1 \\ T_{long} = T_{longbasic} + N_{Dc}T_{min} \cdots 2 \end{cases}$$

この式を元に，新しい LCELL 個数 N'_{Dc} が決定したときについて考えると，1 と 2 の数式より遅延時間 T'_{long} について新たに連立方程式が立てられ，3.5 の 2 と同様に変形することで，

$$N'_{Dc} = \lceil \frac{T_{short} - T_{long}}{T_{min}} \rceil$$

となる．以上により，新たに追加する LCELL 個数 N'_{Dc} を決定する数式が導出され，新たな遅延回路 Dc' に追加する．

一方，回路がタイミング制約条件を満たしている場合，$N'_{Dc} = 0$ とし，LCELL 個数を変更しない．

3.5 と同様に，タイミング制約条件が満たされていない経路がある場合は，解消されるまで手順 2 と手順 3 を繰り返す．

以上より，複合データ転送制御回路固有の制約条件に関するタイミング調整が行われる．

3.7　フロアプランの最適化手法

2.6 で述べたように，高性能な DDP を設計するにあたり，DDP の各ステージごとに対応したフロアプランの設計が必要であると考えられる．よって，先行研究 [12] で提案されたフロアプランの最適化手法を用いて，DDP の高性能化を図る．

3.7.1　フロアプラン最適化のアルゴリズム

フロアプランの最適化は以下のアルゴリズムによって行われる．

1. DDP で必要となる FPGA 回路資源量 (各ステージの LAB 数，Memorybits，Embedded Multiplier 9-bit elements 数)，および FPGA の配置領域 (基準座標 (x,y)，Width, Height，MemorybitsBlock・DSPBlock の x 座標) を決定する．
2. $\lceil\sqrt{LAB\ 数}\rceil$ より，各ステージの Width, Height を決定する．
3. 各ステージの配置座標 (x,y) をそれぞれの隣接関係に従って順に配置し，座標の仮決定を行う．
4. 各ステージにおいて，($Width * Height > LAB$ 数) を満たす範囲で Width と Height を調整する．
5. 総面積が最小になるよう各ステージの配置座標 (x,y) を調整する．

これから得られたフロアプランから，DDP の配置配線を行うことで，より最適化されたデータ転送制御でを実現する．

3.7.2　データの定義

DDP のデータは表 3.2 のように分類されており，各ステージでそれぞれ用意される．

Embedded Multiplier 9-bit elements(以下 EM 数) は乗算器の実装を行う際に必要とな

表 3.2　回路資源量の定義

項目	内容
Embedded Multiplier 9-bit elements	DSP ブロック (乗算機) を利用する個数
Memorybits	M9K ブロックで利用するメモリ Bits
DL	Data Latch の LAB 数
Logic	Stage Logic の LAB 数
C	C 素子の LAB 数

る素子の数で用いられる DSP Block で実装を行う．

Memory Bits(以下 MEM 値) は ROM や RAM を実装する際に必要な bit 数を示しており，MAX10 FPGA を対象とした設計を行う場合，M9K Block で実装される．1 つの M9KBlock で 8196bit のデータを保持することができる．

DL，Logic，C は，各ステージの Data Latch，StageLogic，C 素子で必要となる LAB 数を示す．

表 3.3　FPGA の配置領域の定義

項目	内容
FPGAWidth，FPGAHeight	対象 FPGA で利用可能な配置領域の幅と高さ
座標	DDP を配置する基準座標 (x,y)
Width	DDP の配置領域の幅
DSP 座標*	対象 FPGA の DSP Block の x 座標の集合
M9K 座標*	FPGA 上の M9K Block の x 座標の集合

表 3.3 に，対象 FPGA に DDP を配置するための配置領域について定義する．FPGAWidth，FPGAHeight は DDP の実装対象となる FPGA 回路の領域を示し，フロアプラン結果が FPGA の領域外に設定されることを防ぐ．座標 (x，y) は，フロアプランを行うための基準となる座標であり，各ステージはこの座標を起点に決定される．座標は実装領域の左下

をとる．

実装領域の Width は，FPGA 上に DDP を実装する幅の最大値を入力する．これは実装範囲を決定するときに FPGA のどの領域に回路の配置を行うか制約を設ける必要があるからである．Width を定数として事前に決定しておくことで，Width 方向でステージ間の空白を減らすことを目指す．値は設計者が任意で決めることができる．

DSP Block と M9K Block の x 座標は，実装対象の FPGA 回路の Block 情報を保持しておくことで，DSP Block，M9K Block が必要なステージの制約条件とする．

なお，実装領域の Height に関してはフロアプランの設計完了時に，基準座標から y 軸方向に最も遠く配置されたステージの y 座標を利用することで求める．

3.7.3　初期配置

各ステージの情報から最低限必要となる領域 (Width，Height) と，配置位置の座標 (x，y) を算出し，初期配置を決定する．

各ステージ ($Stage_n$) の DL，Logic，C の LAB 数からステージ全体の LAB 数として算出する数式を以下に示す．

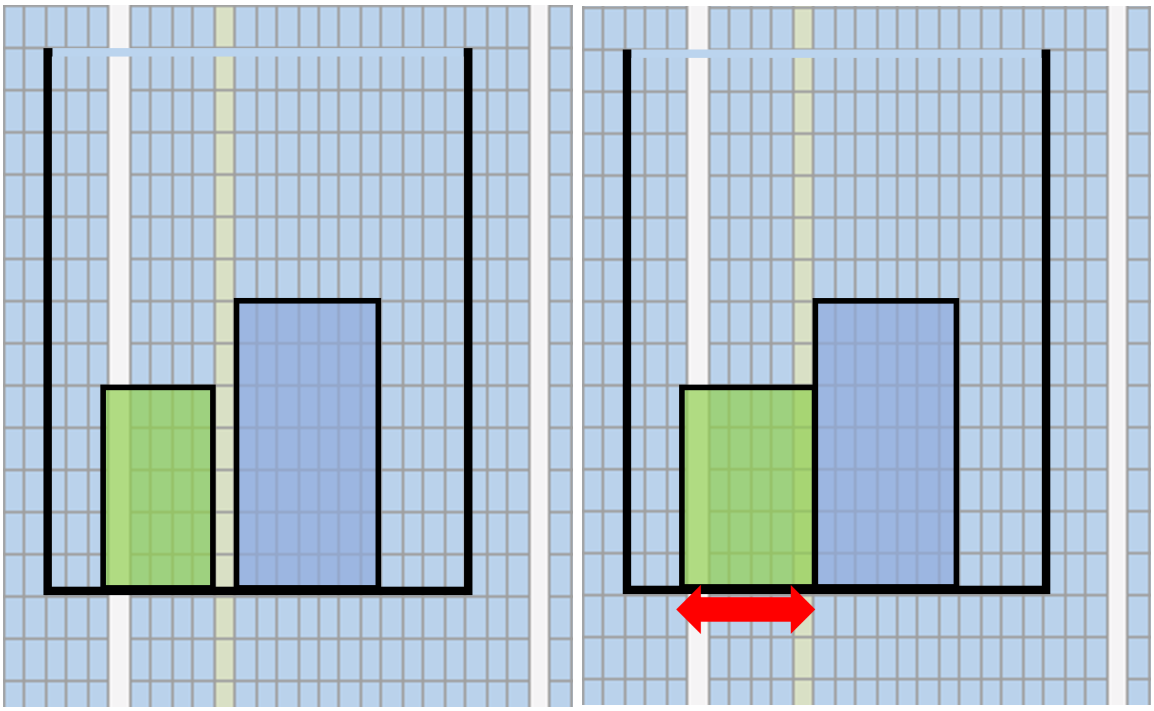
$$LAB\_Stage_n = LAB\_DL_n + LAB\_Logic_n + LAB\_C_n$$

また，EM 値と MEM 値についても，以下の数式より DSP Block と M9K Block の個数を算出する．

$$DSP_n = \lceil \frac{DSP_n}{DSPnum} \rceil$$
$$M9K_n = \lceil \frac{MEM_n}{M9Knum} \rceil$$

EM 値は 9 × 9bit について割り振られており，MAX10FPGA では 1 つの DSP Block で 2 個の乗算器を使用する．そのため本稿では $DSPnum = 2$ として，DSP Block の個数を求める．

先述した通り，M9K Block は 1 つの Block で 8196bit のデータを格納するため，上の数式で $M9Knum = 8196$ として，M9K Block の個数を求める．



(a) 通常ステージの初期領域の仮決定　　(b)DSP,M9KBlock を考慮した初期領域

図 3.8　初期領域の仮決定

続いて，各ステージの LAB 数に応じて Width と Height の仮決定を行う．

$$Width_n = \lceil \sqrt{LAB_n} \rceil$$

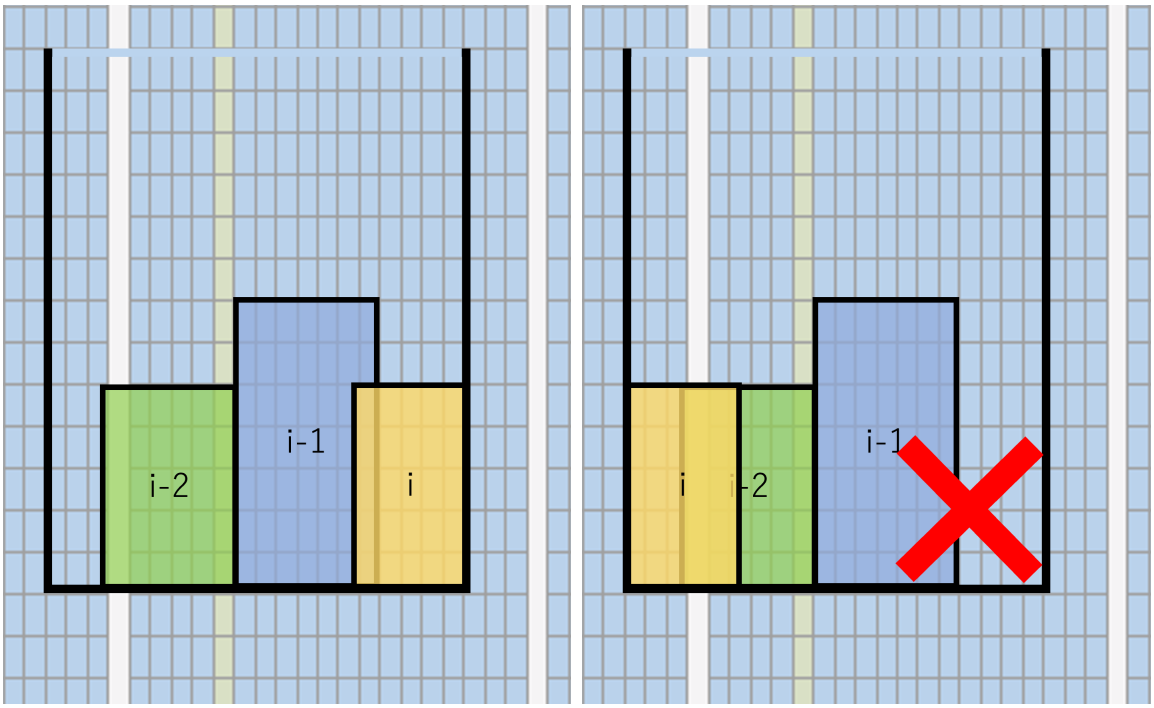
$$Height_n = \lceil \sqrt{LAB_n} \rceil$$

$$Stage_n.DSP \neq 0 \cup Stage_n.M9K \neq 0 \{Width_n ++\}$$

Width と Height は，$Stage_n$ の LAB 数の平方根を切り上げることで求める．

また，各 Stage の DSP 値と MEM 値に 0 以外の値が入っている場合，LAB 数とは別に M9K Block と DSP Block をステージの範囲内に確保する必要があるため，Width をインクリメントする．初期領域の仮決定の様子を図 3.8 に示す．(a) は通常ステージの初期領域の仮決定の様子，(b) は M9K Block と DSP Block の範囲を確保した際の初期領域の様子である．

初期領域の仮決定結果から，配置座標の仮決定を行う．以下のアルゴリズムを，各ステージの隣接関係に従った配置順で実行する．



(a) 初期配置 x 座標の仮決定 (step1)　　(b) 初期配置 x 座標の仮決定 (step2)

図 3.9　初期配置 x 座標の仮決定

隣接関係は設計者が事前に決定しておき，本稿では DDP のパイプラインに沿った，MMCAM，MMRAM，ALU，DMEM，CPY，PS，B，IN，M，CST，OUT の配置順とする．ただし，このアルゴリズムは直前に配置したステージに依存した決定手法となっているため，一番最初に配置されるステージに関しては別途指定して配置を行う．

各ステージの x 座標を，以下のアルゴリズムで仮配置する．仮配置の一例を図 3.9 と図 3.10 に示す．

1. 1 つ前に配置したステージ ($Stage_{n-1}$) の座標の右側に新しくステージ ($Stage_n$) を配置できるか調べる．図 3.9(a) にその様子を示す．
   （a）$Stage_{n-1}$ の座標 (x+width,y) と FPGA の配置可能領域 (x+width,y) の間に $Stage_n$ が配置可能な空間があれば，$Stage_n$ を $Stage_{n-1}$ の右に配置する．
   （b）1a 項が達成されない場合，$Stage_{n-1}$ の右側に配置できないものとし，2 に移る．
   （c）1a 項を達成したとき，$Stage_n.M9K \neq 0$ あるいは $Stage_n.DSP \neq 0$ である場合，

配置領域に M9KBlock，DSPBlock の x 座標が含まれているか確認し，含まれていない場合は決定した座標を破棄，2 で再度座標の決定を行う．

2. 1 項でステージを配置できなかった場合，1 つ前に配置したステージ ($Stage_{n-1}$) の左側にステージを配置できるか調べる．図 3.9(b) にその様子を示す．
   （a）$Stage_{n-1}$ の座標 (x,y) と FPGA の配置可能領域 (x,y) の間に $Stage_n$ が配置可能な空間があれば，$Stage_n$ を $Stage_{n-1}$ の左に配置する．
   （b）2a 項が達成されない場合は，$Stage_{n-1}$ の左側に配置できないものとし，2 に移る．
   （c）2a 項の処理が行われた上で，$Stage_n.M9K \neq 0$ あるいは $Stage_n.DSP \neq 0$ である場合，配置領域に M9KBlock，DSPBlock の x 座標が含まれているか確認し，含まれていない場合は決定した座標を破棄，3 で再度座標の決定を行う．
3. 2 項でステージを配置できなかった場合は，$Stage_n$ の座標 x=FPGA の配置可能領域 x とし，y 軸方向に $Stage_n$ を配置できるか確認する．図 3.10 にその様子を示す．
   （a）$Stage_{n-1}$ 以前に配置したステージについて，$Stage_n$ の x 軸方向の配置領域が重なっているステージ $Stage_{search}$ を探索する．$Stage_{search}.(y+h)$ が最も大きいステージについて，$maxYH = Stage_{search}.(y+h)$ とし，$Stage_n.y = maxYH$ とする．
   （b）$Stage_n.M9K \neq 0$ の場合，$Stage_n$ の x 軸方向の配置領域に M9KBlock の x 座標が含まれていることを確認し，含まれていない場合，$Stage_n.(x+w) - Field.M9K.x$ に従って，$Stage_n.x$ を右に移動させる．
   （c）$Stage_n.DSP \neq 0$ の場合，$Stage_n$ の x 軸方向の配置領域に DSPBlock の x 座標が含まれていることを確認し，含まれていない場合，$Stage_n.(x+w) - Field.DSP.x$ に従って，$Stage_n.x$ を右に移動させる．

また，各ステージの y 座標は，3a で決まった y 座標を除き，x 座標の結果を用いて仮決定する．以下に仮決定のアルゴリズムを説明する．

1. $Stage_{n-1}$ 以前に配置したステージについて，$Stage_n$ の x 軸方向の配置領域が重なっ

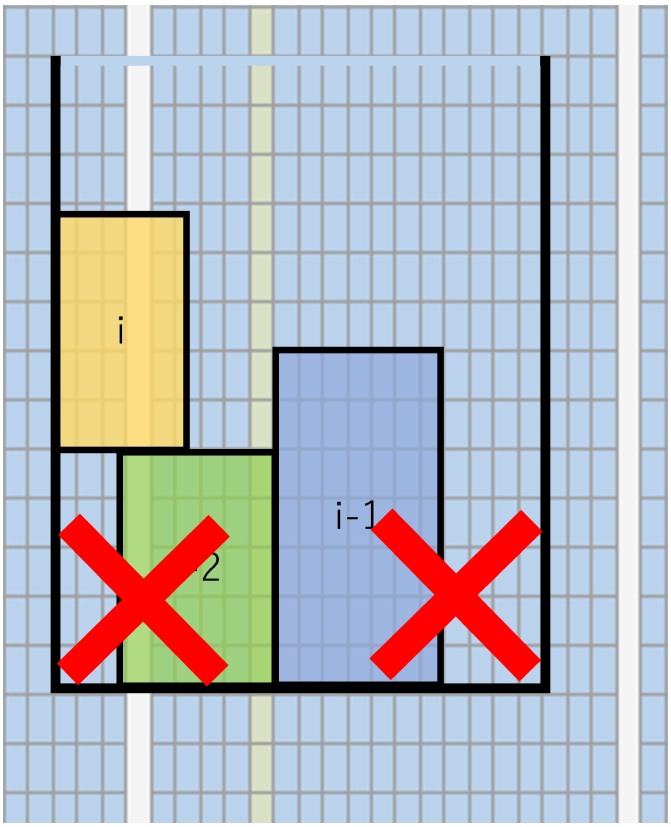


図 3.10　初期配置 x 座標の仮決定 (step3)

ているステージ $Stage_{search}$ を探索する．

2. 1 項より，$Stage_{search}.(y+h)$ の座標で最も大きい値を $maxYH$ とし，探索を行う．
3. 2 項の結果から，$Stage_n.y = maxYH$ とする．

以上から，各ステージの配置座標 (x,y) が仮決定される．

3.7.4　配置座標と領域の調整

3.7.3 の配置位置では，各ステージの Height と Width が冗長であり，配置領域をまんべんなく使用可能な設計が出来ていない．これを解決すべく，各ステージにおいて，Height を小さく，Width を大きく配置設計を行うことで，より集積度の高い配置配線を行える．よって，各ステージの Width と Height を周囲の配置配線状況に合わせて最適化し，その結果に従って対応した位置に修正することで，より最適化されたフロアプランを行う．

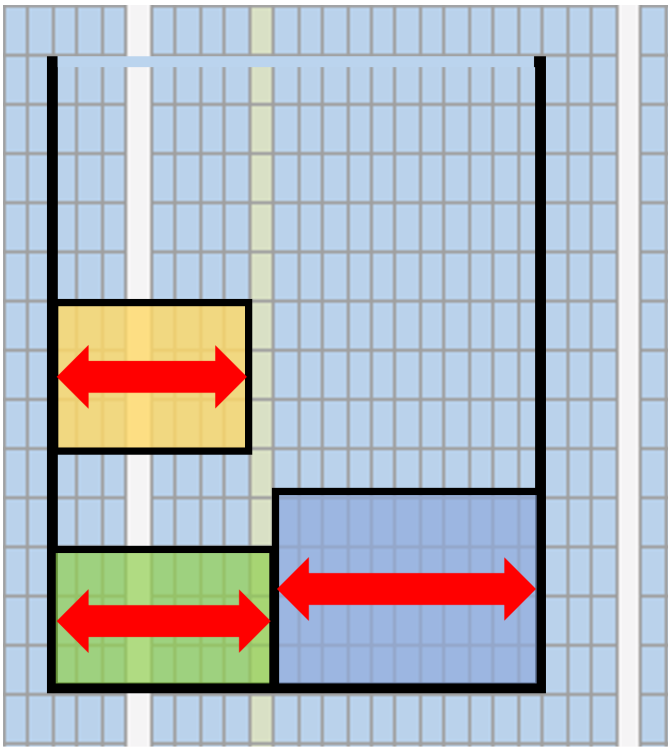


図 3.11　配置領域の最適化

配置領域の調整は，以下に示すアルゴリズムを 3.7.3 で設定した配置順に従って実施することで行う．

1. $Stage_n$ において，LAB 数 $< Width * Height$ ならば Height をデクリメント，LAB 数 $> Width * Height$ ならば Width をインクリメントする．
2. Height をデクリメントするごとに，LAB 数 $\leq Width * Height$ となるまで Width のインクリメントを繰り返す．
3. Width のインクリメント時，$Stage_n$ の左に他のステージが配置されていないならば，配置リソースを確保するために $Stage_n.x$ をデクリメントする．

   この一連の動作は以下の条件に当てはまるまで繰り返し行われる．

   （a）$Stage_n.(x + w)$ が DDP の配置領域を越える場合，あるいは $Stage_n$ が既に配置されている他のステージに重なる場合

   （b）$Stage_n$ の Height が 2 を切る場合

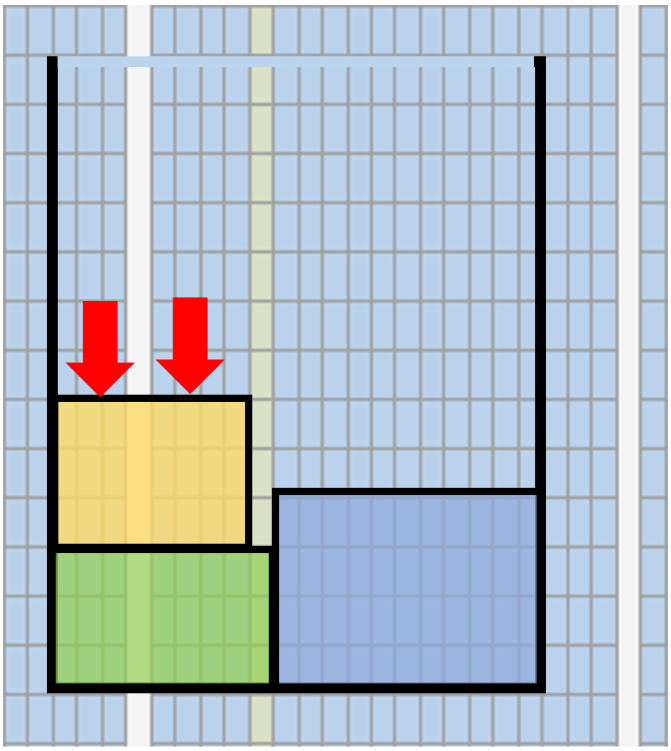


図 3.12　配置座標の最適化

（c）$Stage_n$ の Height が，$Stage_n$ が保持する M9K Block 数，DSP Block 数よりも下回る場合

これにより，DDP 全体の Height が小さくなるように，各ステージの Width，Height が調整され，図 3.11 のようになる．

続いて，配置座標の調整を行う．この段階では，配置領域の調整によって各ステージで配置領域に空白が発生しているため，ステージを詰め，DDP 全体の配置領域を集約する．配置座標の調整を行う様子を図 3.12 に示す．

x 軸方向の座標調整は以下のアルゴリズムで実施する．

1. $Stage_n.x > Stage_{n-1}.(x+w)$ の場合，$Stage_n.x = Stage_{n-1}.(x+w)$ とする．
2. $Stage_n.x > Stage_{n-1}.x$ の場合，

   $Field.(x+w) - Stage_{n-1}.(x+w) > Stage_n.w$ ならば $Stage_n.x = Stage_{n-1}.(x+w)$ とする．

それ以外の場合は $Stage_n.x = Stage_{n-1}.x$ とする．

3. $Stage_n.x < Stage_{n-1}.x$ の場合，

   $Stage_n.(x+w) > Stage_{n-1}.x$ ならば $Stage_n.x = Stage_{n-1}.x - Stage_n.w$ とする．

   $Stage_n.x < Field.x$ ならば $Stage_n.x = Field.x$ とする．

4. $Stage_n$ に M9K Block あるいは DSP Block が含まれる場合，配置領域にブロックが含まれていない場合は，1 から 3 を無視して M9K Block，DSP Block を含む位置に座標を移動させる．

以上より，$Stage_{n-1}$ の相対座標から $Stage_n$ の配置座標を決定する．

y 軸方向の座標調整は，x 軸方向の調整結果より，y 軸方向に重なっている他のステージとの隙間を密集させるように配置位置を決定する．

1. $Stage_{n-1}$ 以前に配置したステージについて，$Stage_n$ の x 軸方向の配置領域が重なっているステージ $Stage_{search}$ を探索する．
2. $Stage_n.y > Stage_{search}.(y+h)$ ならば，$Stage_n.y = Stage_{search}.(y+h)$ とし，y 軸方向にステージを詰める．

これらの手順を踏むことで，各ステージのフロアプランが完了する．DDP 全体の高さは，一番上の座標に配置したステージの座標 y+h から決定する．この結果を設計自動化フローに出力することで，フロアプラン最適化を行った DDP の設計ができる．

3.8　設計自動化フローの改良

本研究で提案する最適化手法によって DDP を設計するにあたり，図 2.8 の設計自動化フローの改良を行う必要がある．設計自動化フローの改良箇所について説明する．

3.8.1　フロアプラン設計に関する改良

3.7 で提案した手法を用いたフロアプラン最適化ツールを設計自動化フローの Region 設定に反映するための環境作りを行う．順序としては，Partition 設定と Region 設定の間でフロアプラン最適化の処理を行い，Region 設定で利用可能にする．

フロアプラン最適化を行うためには，DDP の回路情報を抽出する必要がある．回路情報抽出のための入出力ファイルとプログラムを表 3.4 に示す．

表 3.4　回路情報の抽出で利用するファイル

入力ファイル	Stage_info.txt, fit.rpt
プログラム	Circuit_Report.pl
出力ファイル	FloorplanInput.csv

Circuit_Report.pl は，DDP のステージ情報を保持する Stage_info.txt と，DDP の回路情報が含まれる fit.rpt を入力とし，フロアプラン最適化ツールで利用する，FloorplanInput.csv の出力を行う．Stage_info.txt からは DDP のステージ名を抽出する．fit.rpt からは DDP の各 Stage Logic, DataLatch, C 素子で利用する LE 数を抽出する．Circuit_Report.pl では，Stage_info.txt のステージ名から各ステージの Stage Logic, DataLatch, C 素子を調査し，FloorplanInput.csv に結果を出力する．FloorplanInput.csv では，各ステージにおける 3.2 の情報がまとめられている．

続いて，フロアプラン最適化ツールの入出力ファイルとプログラムを表 3.5 に示す．

表 3.5　フロアプラン最適化ツールで利用するファイル

入力ファイル	FloorplanInput.csv
プログラム	Floorplan.py
出力ファイル	FloorplanResult.txt

Floorplan.py は Floorplaninput.csv によって回路情報が入力され，フロアプラン最適化

を実施し，結果を FloorplanResult.txt に出力する．Floorplan.py は 3.7.4 で示したアルゴリズムによって実装されており，フロアプラン最適化を行う．最適化の結果は Floorplan-Result.txt で出力され，この情報を Region 設定で利用することで，各ステージごとに配置配線する．

以降の Region 設定の流れは，従来の設計自動化フロー [8] と同様である．

3.8.2　タイミング検証ツールの改良

従来の設計自動化フローで利用していたタイミング検証ツールでは，複合データ転送制御回路固有の制約条件を検証できなかった．そのため，タイミング検証ツールと入出力ファイルに関する改良を行い，これを検証可能にする．

改良するのはタイミング検証ツール TimingChecker.py と，タイミング検証ツールの入力ファイルである JSON ファイルを生成する JsonPath_create.pl である．

表 3.6 に JSON ファイル生成のための入出力ファイルとプログラムを示す．

表 3.6　JSON ファイル生成で必要なファイル

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

path.pl はタイミング検証ツールに入力するための JSON 形式のモジュールパス情報を出力する．JsonPath.txt と ConstraintsPath.txt により，fit.rpt から抽出するモジュールパス情報を入力する．Jsonpath.txt ではデータ転送制御回路のモジュール間パス情報を，ConstraintsPath.txt では複合データ転送制御回路回路のモジュール内パス情報を保持する．入力されたパス情報より，fit.rpt から対象モジュールのノード名を抽出し，JSON 形式のモジュールパス情報を作成，path.json として出力する．

ConstraintsPath.txt を新たに作成し，path.pl を改良することで，従来手法で作成された path.json に加えて，複合データ転送制御回路のモジュールパス情報を含んだ JSON ファ

イルが作成可能となった．

表 3.7 にタイミング検証で利用する入出力ファイルとプログラムを示す．

表 3.7　タイミング検証で利用するファイル

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

拡張対象は TimingChecker.py であり，入出力ファイルは path.json の内容が追加されたこと以外は従来の設計自動化フロー [8] と変更点はない．タイミング検証によって，path.json で指定されたパスにおいてタイミング制約を満たしているかどうか確認する．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は，3.5, 3.6 で述べた手法を用いてタイミング調整を行う．

3.9　結言

本章では，データ転送制御回路に着目した回路最適化手法について検討した．まずデータ転送制御回路のタイミング最適化の方針について述べ，データ転送制御回路のタイミング調整法の確立と，フロアプランの最適化によるデータ転送制御回路の遅延時間の低減によって実施されることを説明した．タイミング調整法の確立は，遅延回路の最小遅延時間を解析することで，C 素子間のタイミング調整，複合データ転送制御回路固有の制約条件に関するタイミング調整が可能であることを説明した．フロアプランの最適化によるデータ転送制御回路の遅延時間の低減については，フロアプラン最適化手法によって各ステージで配置配線をまとめることで可能なことを説明した．そして，DDP を提案した最適化手法によって設計するため，設計自動化フローの拡張内容について説明した．

しかし，この最適化手法はまだ最適とは言えず，いくつか課題がある．

3.4.1 は，最小遅延時間の解析内容に依存した手法であり，FPGA チップの変更や，解析

で利用する Clock 信号の周波数による精度の違いといった影響を大きく受ける．また，3.5 の手法では遅延回路の個数を増やしてタイミング調整を行うため，最適とされる遅延回路よりも余裕を持った調整がされる可能性がある．このため，タイミング制約を満たしていたとしても，余剰な遅延回路が設計されていないかを検討する必要がある．

3.7.1 は，先行研究 [12] で評価されたように設計者の手作業によるフロアプランを最適解とし，これに近づけたヒューリスティックなアルゴリズムによる設計自動化を行った．しかしながら，フロアプラン領域の分割方法や，回路の配置場所といったフロアプランの解はいくつも存在するため，本提案が最適であるとは言い難く，いくつかあるであろう解の 1 つという認識が正しいと考えられる．また本提案によるフロアプラン最適化手法は，特定の FPGA チップに対してのみ検討されており，他の FPGA チップに実装する際の検討が行われていない．そのため，FPGA チップに依存せずに，ヒューリスティックな配置配線が生成可能な手法を，引き続き検討する必要がある

第 4 章

評価

4.1　緒言

本章では，第 2 章，第 3 章で述べた手法を用いて作成した DDP の設計結果について評価し，その結果について述べる．評価は従来の設計自動化フローを用いて設計した DDP[8] と，提案手法を用いて設計した DDP を比較することで行った．

]

4.2　評価条件

表 4.1 に本研究の提案手法の評価を行う際の評価条件を示す．

表 4.1　評価条件

FPGA ボード (使用 FPGA チップ)	MAX10 DE10-Lite (10M50DAF484C7G)
回路設計ツール	Quartus Prime 18.0
タイミング解析モデル	Slow 1200mv 0C model

評価対象は Intel 社 MAX10 DE10-Lite(図 4.1) で，FPGA チップは MAX10 10M50DAF484C7G である．回路設計ツールは Intel 社 Quartus Prime 18.0 を使用した．

タイミング解析で用いる解析モデルは “Slow 1200mv 0C model” を利用した．これは FPGA で回路の動作電圧を 1200mV，動作温度を 0 ℃，動作環境が理想状態のときのタイミング解析を行うことを示している．

4.2　評価条件

図 4.1　MAX10 DE10-Lite ボード

設計対象となる DDP は 2.1 のフォーマットで設計し，従来手法，提案手法ともに同一の回路を利用した．設計対象 DDP の仕様は表 4.2，表 4.3 の通りである．

また，従来手法ではタイミング調整が設計者による試行錯誤で行われていたことから，理論的な調整手法が存在しない．そのため，従来手法で用いるタイミング調整法として，二分探索を用いた調整法によって提案手法との比較を行う．以下に従来手法として用いたタイミング調整法を示す．

1. 二分探索を行うにあたり，$max = 16, min = 1$ とする．
2. 全ステージの遅延回路 Ds の個数 N_{Ds} を最大値に設定し，設計を行う．
3. タイミング検証の結果から，制約条件を満たす場合は，$N_{Ds} = \frac{max+min}{2}$ とし，$max = N_{Ds}$ とする．満たさない場合は $min = N_{Ds}$ とする．
4. これを繰り返し，タイミング調整を行う．

遅延回路 Ds は，DDP が確実に動作する個数として，16 個で設定した．遅延回路 Da は，遅延回路の個数が 1 個でも DDP の動作に影響が及ばないことが経験則で判明しているた

表 4.2　DDP のパケット情報

フィールド名	ビット数
color	3bit
gen	8bit
dest	7bit
LR	1bit
CP	1bit
opc	4bit
C	1bit
Z	1bit
data	16bit

表 4.3　設計対象 DDP の仕様

ステージ数	11		stages
入出力パケット	42		bits
CST 容量	20 bits	*	64 words
MMCAM 容量	19 bits	*	64 words
MMRAM 容量	24 bits	*	64 words
DMEM 容量	16 bits	*	1024 words
PS 容量	13 bits	*	128 words

め，1 個で設定した．なお，二分探索が終了してもタイミング調整が完了しなかった場合は，該当ステージの遅延回路の個数を 1 個ずつ増やし，タイミング調整を続ける．

4.3　DDP の設計時間の評価

提案手法と従来手法 [8] で DDP の設計時間を比較した．設計時間は，DDP の回路プロジェクト作成からタイミング調整完了までとする．

表 4.4 に設計時間の比較結果を示す．

表 4.4　設計時間の比較

		従来手法	提案手法
総設計時間	[min]	480	118
コンパイル回数	[回]	12	4

設計時間は 75.4%の短縮に成功し，コンパイル回数も 66%削減することができた．従来手法ではコンパイル 1 回あたりの時間は約 30 分，タイミング調整で必要とする時間は約 10 分と，1 回の回路設計で約 40 分要した．

一方提案手法では，1 回目のコンパイルで約 30 分，2 回目以降は 25 分と約 5 分短縮した．これは，1 回目のコンパイル時にフロアプラン最適化を行ったのち，2 回目以降はこの設定を引き継ぐために Region 設定を実行しなかったため，設計時間の短縮に繋がったと考えられる．また，タイミング調整で必要な時間も約 5 分と短くなり，提案手法によってタイミング調整で要する時間が削減できたことが分かった．

また，コンパイル回数も従来手法と比較して大幅に削減することができた．これは，提案手法が本来必要となる遅延回路をタイミング検証結果から予測して調整を行う手法なのに対し，従来手法はタイミング検証を何度も行うことで必要となる遅延回路を試行錯誤する手法であることが，要因だと考えられる．

この結果から，設計時間の削減が確認でき，提案手法は有効であると考えられる．

4.4　回路面積，回路規模の評価

DDP の回路面積と回路規模を比較し，提案手法が有効であるか示す．回路面積は，DDP 全体の Region の Width と Height から求める．回路規模は，回路合成で使用される素子の LE（LogicElement）の総数で比較する．回路面積と回路規模の評価結果を表 4.5 にまとめる．

表 4.5　回路面積の比較

		従来手法	提案手法
回路規模	[LE 数]	4574	4566
回路面積	[W*H]	26*22	23*23
集積率	[%]	50.0	53.9

従来手法 [8] と比較すると，回路規模は 0.2%削減とほぼ変わらず，回路面積は 7.6%削減ができた．今回設計対象とした DDP の回路構成はどちらの手法でも変わらないため，回路規模にほとんど変化は確認できなかった．従来手法では DDP 全体の Region のみ設定して，配置・配線は Quartus の配置配線機能に任せているため，最小構成の Region ではなく，回路面積が大きくなる傾向にあった．提案手法では DDP の各ステージごとに Region 設定され，ステージごとに回路が集約されると同時に，DDP 全体の面積が小さくなるように配置配線されるため，回路面積が従来手法よりも小さくなった．このことから，設計された回路の集積率も提案手法のほうが 3.9%高くなり，提案手法が有効であると言える．

4.5　スループットの評価

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．

$$\frac{1}{T_f + T_r \text{の最大値}}$$

タイミング検証の結果より，最も $T_f + T_r$ が大きいステージのスループットを算出し，結果，表 4.6 のようになった．

表 4.6　スループットの比較

	従来手法	提案手法
スループット　[packet/s]	25.4M	28.5M

スループットは，従来手法 [8] と比べて，12.2%向上した．従来手法では配置配線時に各ステージの回路（Stage Logic, Data Latch, C 素子）が DDP 全体に自由に配置され，配線遅延が安定しなかった．一方，提案手法では，各ステージで回路が集約され，回路の配線遅延を抑制することができた．また，各ステージの隣接関係を考慮したフロアプラン最適化を行っていることから，ステージ間の配線も従来手法と比べて抑えることができた．配線遅延を抑えることで，データ転送制御回路で必要な遅延時間も低減することができ，スループットの向上ができたと考えられる．そのため，提案手法によって回路の性能向上が確認できたと判断でき，提案手法は有効であると言える．

4.6　データ転送制御回路の遅延時間の比較

4.6.1　ステージ間の遅延時間の比較

DDP 内の各ステージ間の遅延時間の比較結果を表 4.7 に示す．

比較した結果，ステージごとに遅延時間の減少と増加が共に見られ，一概に提案手法が良くなったとは言い難い結果となった．しかし，Tf+Tr の値が最も大きな DMEM ステージでは遅延時間の減少が見られたため，4.5 で述べた通り，DDP 全体のスループット向上を達成したものとする．

表 4.7　各ステージ間の遅延時間の比較

従来手法	Tf [ns]	Tr [ns]	提案手法	Tf [ns]	Tr [ns]
IN	11.6	6.5	IN	12.2	6.5
M	18.8	4.8	M	18.9	4.3
CST	10.7	6.5	CST	10.9	5.2
MMCAM	25.2	5.2	MMCAM	27.4	5.6
MMRAM	15.1	4.2	MMRAM	15.9	4.4
ALU	9.5	5.4	ALU	10.2	6.5
DMEM	31.9	7.4	DMEM	27.4	7.7
CPY	10.6	6.7	CPY	9.9	5.9
PS	23.1	5.1	PS	25.5	5.4
B	31.4	6.5	B	26.6	5.6

4.6.2　複合データ転送制御回路内タイミング制約条件における遅延時間の比較

各複合データ転送制御回路に存在するタイミング制約条件で生じる遅延時間について比較し，結果を表 4.8 に示す.

結果，ほぼすべての経路で遅延時間の減少を確認することができた．これにより，提案したタイミング調整法による遅延回路の調整とフロアプラン最適化による配線遅延の低減は有効であると判断できた.

4.7　結言

本章では，Intel 社製の MAX10 FPGA を対象とし，従来手法 [8] と提案手法で DDP を設計し，提案手法が従来手法よりも有効な設計結果が得られているかについて述べた．評価指標は，DDP の設計時間，DDP 全体の回路規模，回路面積，スループットである.

表 4.8 複合データ転送制御回路のタイミング制約条件における遅延時間の比較

従来手法	Tsend [ns]	Tctrl [ns]	提案手法	Tsend [ns]	Tctrl [ns]
CM	29.6	15.2	CM	24.6	13.1
CE_MMCAM	10.0	8.6	CE_MMCAM	9.9	8.1
CX2_CPYDL	9.4	8.7	CX2_CPYDL	8.9	7.0
CX2_EXBDL1	9.4	8.0	CX2_EXBDL1	8.9	6.9
CX2_EXBDL2	9.4	7.9	CX2_EXBDL2	9.0	7.1
CX2_ORADL	9.6	8.1	CX2_ORADL	8.7	7.2
CX2_ORB	12.8	11.1	CX2_ORB	13.2	11.0
CX2_AND3L	10.0	9.8	CX2_AND3L	9.6	8.6
CE_PS	8.9	8.3	CE_PS	9.5	7.8
CB_B1	8.9	8.8	CB_B1	7.8	7.2
CB_B2	8.7	8.7	CB_B2	8.2	7.6

設計時間においては，設計時間を 75.4%短縮，DDP 設計時の回路のコンパイル回数を 66%削減できたことから，提案したタイミング調整法が有効であることを述べた．回路規模は，提案手法が 0.2%削減と回路を保持しており，回路面積は 7.6%削減に成功したため，より集積率の高い回路設計ができ，提案したフロアプラン最適化手法が有効であると述べた．スループットにおいては，提案手法が 12.2%上昇したことから，フロアプラン最適化によるデータ転送制御回路の性能向上が見られたと判断し，提案手法が有効であることを述べた．

ただし，本提案手法の評価結果は，4.2 で設定した条件の元で行った結果であり，他の FPGA チップを用いた場合や，DDP の仕様を変更した場合は結果が変わることが予想される．

第 5 章

結論

近年，世界の IoT(Internet of Things) デバイスは一般社会への普及が加速しており，2023 年には 340.9 億台に増加すると予測されている．IoT 技術の普及は，クラウド技術や無線通信技術の発展による影響が大きいと考えられ，現在では PC やスマートフォンといった汎用的なインターネット接続端末に加え，家電や自動車，ビルシステムや工場の生産ライン等でもインターネットが接続されている．IoT システムの実現にあたって，システムの利用ユーザに近い端末やネットワーク内でデータを処理，集約するエッジコンピューティングが注目されている．エッジコンピューティングを用いたシステム開発は今後更に高まると予想され，IoT 技術の一般社会への普及を見据えて，より汎用的かつ高性能なエッジデバイスの開発が必要である．

データ駆動型プロセッサ DDP(Data-Driven Processor)[5] は，クロック信号を用いない非ノイマン型プロセッサであるため，割り込み処理の必要がなく，複数ストリームデータに対する多重処理が高速に可能である．また，各パケットに対し，演算に必要な回路のみが動作し，演算で使用しない回路は電力を消費しないことから，省電力性も兼ね備えている．これらの理由から，エッジデバイス用のアーキテクチャとして，DDP は有効であると考えられる．

また，エッジデバイスには様々な用途が想定され，それぞれのデバイスに対して回路設計を行うにあたり，柔軟に内部の回路構成を改変可能な FPGA が有効である．このことから，エッジコンピューティングを実現するにあたり，FPGA を用いた設計は有効であると考えられ，DDP をアーキテクチャとしたシステムに適用可能である．

しかし，既存の FPGA 設計ツールは同期式回路向けに最適化されており，DDP を始め

とした非同期式回路の設計には適していない．そのため，商用の FPGA に非同期式回路を実装するための手法を検討する必要があり，これまでの研究 [7][8][10] より，動作保証された DDP を自動で設計することが可能となりつつある．一方で，DDP の動作保証をするためのタイミング最適化手法が確立されておらず，設計者による試行錯誤によってタイミング調整が行われている．このため，DDP の設計時間の損失が大きな課題となっている．また，FPGA は配置配線による遅延の影響が ASIC と比べて大きく，回路設計ツールによる自動配置機能では配線遅延が増大する問題があった．

本研究では，DDP の動作保証と性能向上を目的とし，設計自動化フロー [8] の拡張による，データ転送制御回路に着目した回路最適化手法について検討した．回路最適化は，データ転送制御回路のタイミング調整法の確立と，フロアプラン最適化によるデータ転送制御回路の遅延時間の低減の，大きく 2 つの手法によって行われた．データ転送制御回路間のタイミング調整および，複合データ転送制御回路固有の制約条件に基づいたタイミング調整法を確立することで，DDP の動作保証と設計時間の短縮を図った．また，フロアプラン最適化によって，データ転送制御回路で生じる配線遅延を低減させ，DDP の回路面積の縮小と性能向上を図った．

提案手法によって設計した DDP と従来手法 [8] によって設計した DDP を評価し，設計時間，回路規模，回路面積，スループットについて，有効性を評価した．設計時間は 75.4% の短縮に成功し，コンパイル回数は 66%削減することができた．回路規模は 0.2%削減と変化がほぼ見られず，回路面積は 7.6%削減することができた．スループットは，12.2%上昇した．これら全ての指標において提案手法が優位であったため，提案手法は有効性のある設計手法であると結論づけた．

本研究における最適化手法に関して，データ転送制御回路のタイミング調整とフロアプランの最適化から考察する．

本研究で提案したデータ転送制御回路のタイミング調整法は，遅延回路の最小遅延時間の解析を事前に行い，その結果から必要となる遅延回路を調整する．そのため，最小遅延時間の解析内容に依存した手法であり，FPGA チップの変更や，解析で利用する Clock 信号の

周波数による精度の違いといった影響を大きく受ける．また提案したタイミング調整法では，最小遅延時間の解析結果を用いてタイミング調整で必要となる遅延回路の個数を決定するが，遅延回路の個数を増やしてタイミング調整を行うため，最適とされる遅延回路よりも余裕を持った調整がされる可能性がある．このため，タイミング制約を満たしていたとしても，余剰な遅延回路が設計されていないかを検討する必要がある．

フロアプランの最適化は，先行研究 [12] で評価したように設計者の手作業によるフロアプランを最適解とし，これに近づけたヒューリスティックなアルゴリズムによる設計自動化を行った．しかしながら，フロアプラン領域の分割方法や，回路の配置場所といったフロアプランの解はいくつも存在するため，本提案が最適であるとは言い難く，いくつかあるであろう解の 1 つという認識が正しいと考えられる．また本提案による配置結果も，空白領域が生じていたりステージ配置が最適かといった懸念点があるため，今後も引き続き検討を行う必要がある．

またその他の課題として，以下のものが挙げられる．

- マルチコア設計を考慮した設計手法の検討

  本提案手法で用いたフロアプラン最適化手法では，単一コアによる DDP の配置配線を想定していたが，マルチコア設計を行うことで，より高性能な DDP の設計が可能となる．そのため，DDP をマルチコア設計可能なフロアプランの生成手法を今後検討する必要がある..
- Xilinx 社製 FPGA に対する評価

  本提案手法では，Intel 社製 FPGA に対して評価を行ったが，商用 FPGA を対象としてより柔軟な設計を可能にするためには，Xilinx 社製 FPGA に対する設計自動化フローについても検討する必要がある．そのため，回路設計プロジェクト作成，配置配線設定，最適化防止設定，タイミング解析機能の CUI コマンドの調査をする必要がある．
- タイミング調整の更なる自動化

  本提案手法ではデータ転送制御回路のタイミング調整で必要な遅延回路の個数を決定

することはできたが，遅延回路個数の修正は設計者が Verilog コードを変更することで行っている．そのため，遅延回路が含まれる該当の Verilog コードを自動で修正するスクリプトを新たに組むことで，タイミング調整の更なる自動化を行うことができると考えられる．

今後 DDP を FPGA 向きに設計するにあたり，本提案に加えて上記の課題を解決することで，設計者に依存しない設計が可能となり，DDP を用途に応じて多品種少量生産することが容易になることが予想される．また，タイミング調整法の更なる改良と自動化を行うことで，高性能な DDP を安定生産できるようになり，IoT エッジデバイスとして実用化されることができるようになる．

謝辞

本研究を進めるにあたり，日頃より懇切丁寧にご指導，ご鞭撻を賜りました岩田誠教授に心より感謝の意を表すと同時に，ここで厚く御礼申し上げます．研究室活動を通じて，知識や技術の習得のみならず，人間的にも成長することができました．本当にお世話になりました．

本研究の論文の副査をお引き受け下さり，中間発表等で貴重なご意見や疑問点を指摘して頂きました，横山和俊教授，松崎公紀教授に厚く御礼申し上げます．

本研究の基礎を築き上げてくださり，また，在学中は回路設計に関する知識や技術をご教示くださった，研究室 OB の長野寛司氏に心より感謝致します．

本研究の要素の一つである，複合データ転送制御回路のタイミング検証法について，日頃から相談や質問に応じてくれた，修士 1 年の尾ノ井嶺卓氏，学部 4 年の仁野槙人氏に心より感謝致します．研究内容の完成度を上げることができました．

研究を進めるにあたって，研究室ミーティングを始めとした研究室活動の中で，様々な疑問点や改善点等を指摘して頂いた，修士 1 年の岡野秀平氏，筧拓也氏，古田雄大氏に心より感謝申し上げます．

博士 1 年の張震氏には論文英訳の添削等，学部 4 年の高橋龍一氏，渡邊竜也氏には，日頃の研究室活動を行うにあたり広くご支援ご協力を頂き，研究を円滑に進めることができました．心より感謝申し上げます．

最後になりましたが，日頃からご支援頂きました関係者の皆様に心より御礼を申し上げます．

参考文献

[1] 総務省, “令和元年度情報通信白書第 2 節デジタル経済を支える ICT の動向,” https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r01/pdf/n1200000.pdf, 2020 年 2 月 10 日参照.

[2] ARM, “ARM のアーキテクチャ,” https://www.arm.com/ja/architecture, 2020 年 2 月 24 日参照.

[3] RISC-V International, “About RISC-V,” https://riscv.org/about/, 2020 年 2 月 24 日参照.

[4] 大原雄介, “Core i シリーズにも使われる「SMT」の利点と欠点,” https://ascii.jp/elem/000/000/560/560386/, 2020 年 2 月 24 日参照.

[5] Hiroki Terada, Souichi Miyata, and Makoto Iwata, “DDMP's: Self-Timed Super-Pipelined Data-Driven Multimedia Processors,” Proceeding of the IEEE, vol. 87, no. 2, pp.282–296, Feb. 1999.

[6] 滝澤恵多朗, 齋藤寛, “束データ方式による非同期回路の FPGA 設計支援環境の構築,” 情報処理学会研究報告システムと LSI の設計技術（SLDM）, 2015-SLDM-171, pp.1-6, May. 2015.

[7] 吉川千里, 三宮秀次, 岩田誠, 佐藤聡, 西川博昭,“FPGA 向き自己同期型パイプライン回路構成法,” 情報処理学会研究報告システム・アーキテクチャ（ARC）,2021-ARC-243(25), pp.1-7, Jan. 2021.

[8] 長野寛司, “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,” 修士学位論文, Feb. 2021.

[9] 吉見宗真, 齋藤寛, “束データ方式による非同期式回路の遅延調整に関する考察” 情報処理学会研究報告システムと LSI の設計技術（SLDM）, 2016-SLDM-176, pp.1-6, May. 2016.

[10] 尾ノ井嶺卓, “セルフタイム型複合データ転送制御回路の FPGA 実装用タイミング検証法,” 学士学位論文, Feb. 2021.

[11] 清田紘司, 藤吉邦洋, “Sequence-Pair 表記された一般構造フロアプランの Simulated Anneling 法探索,” 電子情報通信学会論文誌 A, Vol.J84-A No.7, Jul. 2001

[12] 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,” 学士学位論文, Feb. 2020.

[13] 天野英晴編, “FPGA の原理と構成,” オーム社, pp.35-37, Apr. 2016.

[14] terasic, “DE10-Lite Board: Terasic - All FPGA Boards - MAX 10,” https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1021, 2020 年 2 月 28 日参照

[15] Intel, “ブロックベース・デザインユーザーガイドインテル®Quartus®Prime プロ・エディション,” https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/ug/ug-qpp-block-based-design_j.pdf, 2022 年 1 月 15 日参照.

[16] ALTERA, “デザイン・フロアプランの解析および最適化,” https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/hb/qts/qts_qii52006_j.pdf, 2020 年 2 月 8 日参照

[17] Intel, “Intel Quartus Prime Standard Edition User Guide: Getting Started,” https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html, Feb. 2020.

[18] Intel, “インテル Quartus Prime プロ・エディションユーザーガイド:デザインのコンパイル,” https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/ug/ug-qpp-compiler.pdf, Feb. 2020.