素因数分解装置の調査・検討に関する報告書

２００４年２月

株式会社富士通研究所

伊豆　哲也
伊藤　孝一
小暮　淳
小檜山　清之
下山　武司
武仲　正彦
鳥居　直哉
桝井　昇一

富士通株式会社

向田　健二

素因数分解装置の調査・検討に関する報告書

富士通株式会社

2004 年 2 月

本報告書について

本報告書は, 素因数分解装置 TWIRL の調査・検討結果を報告するものである. TWIRL とは, 2003 年 8 月に Adi Shamir と Eran Tromer によって提案された素因数分解専用ハードウェアデザインの名称である [ST03]. TWIRL は汎用的な ASIC 技術を用いることで, ふるいと呼ばれる処理を高い並列度で処理することを特徴とする. 提案者の試算によると, TWIRL が 1024-bit 合成数を素因数分解するのに必要なコストは, 1000 万ドル (約 10 億円) の費用と約 1 年の計算時間となっている. 公開鍵暗号の現在 (2004 年 1 月) のデファクトスタンダードである RSA 暗号の標準的な鍵の長さ 1024-bit であり, もしも TWIRL が現実に製造・動作可能であれば, RSA 暗号が解読可能であることを意味することになる. しかし提案者によるコスト試算は, 提案者も認める通り概算値でしかなく, TWIRL の実現性に関する詳細評価の必要性が指摘されていた [ST03, RSA03].

本報告書は TWIRL の実現可能性を, 理論面と現実面の両面から詳細に検討することを目的とする. 具体的には, TWIRL の提案論文を元にして動作原理を詳細に検討し, 提案論文の仕様に基づいた回路設計を行う. そして設計から算出される回路規模見積もりを元に, TWIRL の実現可能性を検討する. なお TWIRL の提案論文では, 分解対象となる合成数のサイズとして 512-bit, 768-bit, 1024-bit の 3 種類が挙げられているが, 本報告書は主に 1024-bit 合成数の場合についてのみ報告する.

本報告書の構成は以下の通りである. 第 1 章で総論を述べ, 評価結果概要を報告する. その後, 第 2 章では TWIRL の動作原理を (提案論文に従って) 説明し, 第 3 章では報告者による回路設計例を示すと共に, TWIRL の回路規模見積もりを算出する.

報告概要

TWIRL とは, 2003 年 8 月に Adi Shamir と Eran Tromer によって提案された素因数分解専用ハードウェアデザインの名称である [ST03]. TWIRL が 1024-bit 合成数を素因数分解するのに必要なコストは, 1000 万ドル (約 10 億円) の費用と約 1 年の計算時間であると提案者は試算している. この試算の妥当性を検討するために, 報告者は TWIRL の動作仕様を詳細に調査し, 基本回路設計を行うことで, 提案者の主張するハードウェアデザインと回路規模の見積もりの妥当性に関する検証を行った.

動作原理については, TWIRL を構成する大部分のユニットについては提案者の主張するハードウェアデザインおよび回路規模の見積もりは妥当であるが, 配送機構 (Largish Station の Buffer と Smallish Station の Funnel) ユニットについては, 提案者の見積もりよりも大規模な回路が必要であると報告者は考える.

次に報告者が検討した基本回路設計をもとに, TWIRL の実現可能性について検討を行った結果, 現時点 (2004 年 1 月) のテクノロジーを用いた場合, 提案者が想定する性能を持つ TWIRL の実現は不可能であるとの結論に至った. その主な理由は以下の 3 点である:

1. 提案者の主張通り TWIRL を単一の LSI に実装するには, 直径 100 mm 以上の巨大ウェハーを欠損なく製造する技術が必要となる. しかしこのような巨大なウェハーは現在の技術では製造不可能である.
2. TWIRL を複数の LSI に分割して実装したとしても必要 LSI 数が膨大となり, 単一のボードには実現不可能である.
3. 分割した LSI を複数のボードに実装したとしても, 提案者が主張する周波数 (1 GHz) を前提とした場合, ボード間の IO 数が多数であることから, 現在の技術では実現不可能である.

報告者の検討手順については第 1 章を参照されたい. また TWIRL の動作原理については第 2 章を, 報告者による検討内容については第 3 章を参照されたい.

目次

本報告書について i

報告概要 ii

目次 iii

第 1 章　総論 1

1.1　目的 1

1.2　TWIRL の特徴 2

1.3　調査・検討内容 3

1.4　検討結果 3

1.4.1　ウェハーの製造可能性 4

1.4.2　報告者による回路規模見積もり 5

1.4.3　複数 LSI を用いた実現可能性 5

1.4.4　複数ボードを用いた実現可能性 6

1.4.5　TWIRL 実現に必要なブレークスルー 6

第 2 章　動作原理 8

2.1　位置付け 8

2.2　数体ふるい法 9

2.2.1　アルゴリズム概要 9

2.2.2　ふるい処理 10

2.2.3　TWINKLE 13

2.3　基本構造 13

2.3.1　ふるいの並列処理 14

2.3.2 Largish Station 16
2.3.3 Smallish Station 22
2.3.4 Tiny Station 25
2.4 提案者によるコスト見積もり 26
2.4.1 1024-bit 合成数 26
2.4.2 512-bit 合成数 27
2.4.3 768-bit 合成数 28
2.4.4 ハードウェアに関するパラメータ詳細 30
2.4.5 数体ふるい法に関するパラメータ詳細 31
2.5 詳細デザイン 32
2.5.1 Delivery Line 32
2.5.2 Emitter の重複 33
2.5.3 Funnel 36
2.5.4 初期化 37
2.5.5 ふるい対象の削減 37
2.5.6 ふるいのカスケード 38
2.5.7 関係の導出 39
2.6 Largish Station における送信器の理論的サイズ 41
2.6.1 Rational TWIRL 41
2.6.2 Algebraic TWIRL 42

第 3 章 詳細検討 44
3.1 TWIRL を用いたふるい装置の概要 44
3.1.1 TWIRL cluster 44
3.1.2 Rational TWIRL 46
3.2 機能仕様について 47
3.3 Largish Station 48
3.3.1 Emission Triplet 発行処理部 53
3.3.2 DRAM 59
3.3.3 DRAM read 処理部 61
3.3.4 Processor 64
3.3.5 SRAM write 処理部 66
3.3.6 SRAM 69

3.3.7 SRAM write back 処理部 71
3.3.8 空きアドレス検出処理部 74
3.3.9 DRAM 初期化部 78
3.3.10 SRAM 初期化部 80
3.3.11 Buffer 81
3.3.12 Bubble Sort/Random Shuffle 84
3.3.13 *l*-sort 処理部 89
3.3.14 *k*-bit 値 *M*-4 × 2 分配処理部 92
3.3.15 *l*-重複削除処理部 96
3.3.16 Delivery Line 出力処理部 98
3.3.17 パイプライン加算処理部 102
3.4 Smallish Station 106
3.4.1 Emitter-Funnel 109
3.4.2 Emitter 113
3.4.3 1st/2nd Level Funnel (*N*-*M* Funnel) 115
3.4.4 *N*-データ列タイミング調整処理部 117
3.4.5 *N*-*M* 変換処理部 119
3.4.6 Segment 123
3.5 Tiny Station 126
3.6 詳細検討結果 129
3.6.1 回路規模見積もり詳細 129
3.6.2 複数の LSI を用いた TWIRL 実現に関する詳細検討 131
3.6.3 複数のボードを用いた TWIRL 実現に関する詳細検討 135

参考文献 137

第 1 章

総論

本章では素因数分解装置 TWIRL に関する調査・検討内容と検討結果概要を報告する. TWIRL の動作原理については第 2 章を, 検討詳細については第 3 章を参照されたい.

1.1 目的

公開鍵暗号技術のいくつかは, 素因数分解問題の難しさに安全性の根拠をおいている. 電子政府推奨暗号リスト [CRY02] では, 合成数サイズが 1024-bit 以上の場合には素因数分解は現実的には困難であり, 向こう 10 年は分解できないことを想定している. 従って, 現実的に素因数分解問題が困難であるという事実は, 安全な電子政府を構築する上で欠かせない前提となっている.

数体ふるい法は素因数分解アルゴリズムの中で最も優れた方法である [LL93]. 従来はソフトウェア実装による実験例しか報告されていなかったが, 最近になって専用ハードウェア, 特に LSI を用いたデザインが提案されている. TWIRL はイスラエルの暗号研究者 Adi Shamir [*1] と Eran Tromer によって 2003 年 8 月に提案されたハードウェアデザインであり [ST03], ふるいと呼ばれる処理を大規模に並列計算することを大きな特徴とする. 1024-bit 合成数をターゲットとした TWIRL による素因数分解のコストは, 1000 万ドル (約 10 億円) の費用[*2] と約 1 年の計算時間と提案者は試算しており, もしも試算通りに製造・動作が可能であれば, 素因数分解問題を利用した公開鍵暗号技術の安全性が損なわれることになる. しかしこの見積もりは, 提案者自身も認める通り概算値でしかなく, 実現可能性の可否も含めた再検討の必要性が指摘されていた [ST03, RSA03].

[*1] RSA 暗号の考案者として有名である (RSA の ”S”).

[*2] $1=100 円として換算, 以下同様.

表 1.1　TWIRL の回路規模見積もり

プロセスルール	算出者	Rational TWIRL	Algebraic TWIRL
0.13 μm	提案者	15960 mm^2	65960 mm^2
0.09 μm	提案者	7650 mm^2	31620 mm^2
	報告者	23892 mm^2	—

本報告書の目的は, 現在電子政府推奨暗号リストを採用した際に想定されているパラメータ (合成数サイズ 1024-bit) に対応する TWIRL の仕様を調査・検討し, 実現可能性・実現時期・費用等の詳細な評価を行うことである.

1.2　TWIRL の特徴

TWIRL (The Weizmann Institute Relation Locator) とは, 2003 年 8 月に Adi Shamir と Eran Tromer によって提案された素因数分解専用ハードウェアデザインの名称であり [ST03], 最良の素因数分解アルゴリズムである数体ふるい法 [LL93] のふるいステップをハードウェア実装したものである. このためふるいステップで使用する 2 種類のふるい (有理数的ふるい (rational sieve) と代数的ふるい (algebraic sieve)) に対応して, TWIRL は Rational TWIRL と Algebraic TWIRL の 2 つの構成要素からなる. TWIRL は LSI を効果的に利用することで, ふるい処理を高い並列度で計算することを特徴とする (1024-bit 合成数をターゲットとした Rational TWIRL の場合で 4096 並列). さらにこの並列処理は, ふるいに伴う対数値の加算処理のみを高い並列度で実行し, これらの加算処理に対する制御を一つの回路にて集中管理することで, 典型的なふるいの実装デザインと比較して大幅な回路規模削減を実現している. その結果, 1024-bit 合成数をターゲットとした TWIRL の回路規模は, 0.13 μm プロセスルールを使用した場合, Rational TWIRL の場合で 15960 mm^2, Algebraic TWIRL の場合で 65960 mm^2 になると提案者は試算している (表 1.1 参照[*3]). なお実際の分解にあたっては, Rational TWIRL 8 個と Algebraic TWIRL 1 個を組み合わせた TWIRL クラスタとして分解を行う. 1 年で分解を行うには 194 組の TWIRL クラスタが必要である.

TWIRL では複数の加算部に対する配送機能が必要になるものの, 提案者はこの機能に関しては必要性のみを述べるにとどまっており, 具体的な実現方法については触れられていない. 報

[*3] 0.09 μm プロセスルールを使用した場合の換算値は, Rational TWIRL の場合で $15960 \times (9/13)^2 \approx 7650$ mm^2, Algebraic TWIRL の場合で $65960 \times (9/13)^2 \approx 31620$ mm^2 となる.

告者の検討では, 配送機能を実現するのに必要な回路規模は非常に大きく, その結果, 全体の回路規模は提案者の主張よりも大きくなるという評価結論を得た. また TWIRL は基本的には汎用的な ASIC 技術を前提に設計されているが, 一部の回路については DRAM を論理回路と混載させる必要があるためフルカスタムで作成する必要があり, 通常の LSI 製造より割高になると予想する.

1.3 調査・検討内容

本報告では, 以下の手順によって検討を進めた.

まず TWIRL の提案論文 [ST03] に従い, 具体的な仕様を調査・検討した. いくつかのパラメータについては見直しが必要と考え, パラメータの理論的な最適値を数値実験を通して定めた. これら仕様・パラメータについては, 詳細を第 2 章にまとめた.

次に具体化させた仕様に基づき, 1024-bit 合成数をターゲットとした TWIRL の基本回路設計を行った. 設計方針としては, TWIRL の性能を最も左右するパラメータである Delivery Line と Sieve Location の移動速度が, 提案者の想定通り, 内部周波数 1 GHz のときに 1 clock cycle/cell となるように設計した[*4]. そしてこの回路設計を元に回路面積を算出し, TWIRL を複数 LSI に機能分割した場合の LSI 数を概算した. これら設計・検討結果の詳細は第 3 章にまとめた. ここで回路設計で用いたパラメータとしては, 基本的には提案論文から導出される理論的な値を用いたが, 実装に伴うさまざまな制約から, 一部については修正した値を用いた. なお本報告書の目的は提案論文の詳細検討であるため, 論文の内容を越える改良は行っていない. しかし論文で前提となっているが詳細が記されていない項目については, 適当な補完を行った.

1.4 検討結果

報告者が作成した基本回路設計を元に TWIRL の実現可能性を検討した結果, TWIRL は現時点 (2004 年 1 月) のテクノロジーでは, 実際に製造することは極めて困難であるとの結論を得た. その主な理由は以下の 3 点に要約される.

1. 提案者が主張するように TWIRL を単一の LSI として製造するためには巨大なウェハー (少なくとも直径 100 mm) を製造する必要があるが, このように巨大なウェハーを欠損なく製造することは現時点のテクノロジーでは不可能である.

[*4] 従って移動速度や内部周波数を変更させた場合については考察していない. これは変更させた場合に回路設計方針が大きく変わってしまうからである.

2. 次に TWIRL を現在のテクノロジーで製造可能な大きさの複数の LSI に分割して実現する方法が考えられる. しかし報告者が検討したところ, LSI 間の IO 数が多数必要なことから, Rational TWIRL だけでも多数の LSI (40 mm 角の LSI で 3362 個) を必要とする[*5] ため, 単一のボードに搭載不可能な規模[*6] となり, やはり現時点のテクノロジーでは製造不可能である.
3. 次に分割した LSI を複数のボードに実装する方法が考えられる. しかし報告者が検討したところ, Rational TWIRL の場合, 確かに 54 枚程度のボードに理論的には実装可能[*7] であるものの, 提案者の主張する動作周波数 (1 GHz) を前提とした場合, ボード間の IO 数が多数 (最大 1500-bit) 必要なことから, やはり現時点のテクノロジーでは実現不可能である[*8].

これら考察結果により, 現在のテクノロジーを前提とした場合, TWIRL を実際に実現することは極めて困難であると報告者は考える. なお本検討にあたっては, 結線の困難さは考慮していないため, 実際の実現可能性はさらに低いと思われる.

前述の通り, 報告者の回路設計は 1024-bit をターゲットとした TWIRL に対する提案者の仕様を忠実に反映させている. 従って半導体テクノロジーが劇的に向上した場合や, 仕様を越えた改良を施した場合の実現可能性は検討していない. また 512-bit 合成数や 768-bit 合成数をターゲットとした場合の TWIRL の実現可能性についても報告者は検討を行っていない.

以下, 各項目について詳細を述べる. 以下で述べるパラメータは, ことわりのない限り 1024-bit 合成数をターゲットとしたものである.

1.4.1 ウェハーの製造可能性

TWIRL を単一の LSI として製造するためには, 巨大な 1 枚のウェハーを製造する必要がある. 提案者の主張 (0.13 μm プロセスルールを使用) では, Rational TWIRL を製造するためには直径 150 mm ウェハー 1 枚, Algebraic TWIRL を製造するためには直径 300 mm ウェハー 1 枚を必要とする. また 0.09 μm プロセスルールを使用したとしても, 提案者の主張では Rational TWIRL に直径 104 mm ウェハー 1 枚[*9], Algebraic TWIRL に直径 208 mm ウェハー 1 枚を必要とし, 報告者による回路規模見積もりでは Rational TWIRL に直径 174 mm ウェハー 1 枚を

[*5] 0.09 μm プロセスルールを使用. また前述した配送機構の補正を追加している.
[*6] LSI 間の間隔を 10mm として正方形に配置した場合, (40mm+10mm) × 58 ≈ 3 m 角のボードを必要とする.
[*7] 0.09 μm プロセスルールを使用. また前述した配送機構の補正を追加している.
[*8] 動作周波数 1 GHz の場合, ボード間 IO は最大 400 bit が限界であろう.
[*9] ウェハーの直径は面積による単なる換算値である. 以下も同様.

必要とする (表 1.1 参照). しかしこれの大きさのウェハーを欠損なく製造することは, 現在の半導体技術では不可能である. 実際, 歩留まり率の関係から実用上製造可能な LSI は 40 mm 角が限界である.

1.4.2 報告者による回路規模見積もり

次に TWIRL の複数 LSI への分割実装を検討するために, Rational TWIRL の回路規模の再評価を行った. その結果, 0.09 μm プロセスルールを用いた場合の回路規模は 23892 mm^2 となった. 同じプロセスルールを用いた場合の提案者による回路規模見積もりは 7640 mm^2 となることから, 約 3.1 倍の回路規模となっている (表 1.1 参照).

このように報告者の検討結果が提案者の見積もり結果よりも大規模になる最大の理由は, 1.2 節で述べたように配送機構, 特に Largish Station の Buffer と呼ばれるユニットと Smallish Station の Funnel と呼ばれるユニットにおいて, 4096 並列された加算器に対するデータの配送機能の回路規模を提案者が考慮していないことが原因である. 報告者は Buffer/Funnel の回路デザインは慎重に行う必要があると考える. なぜなら, TWIRL が想定するパイプライン的な加算処理を実現するためには, 例えば Buffer の場合, 4096 通りの配送先を持つ対数値データのソート処理を 1 cycle に 2000 個以上処理する必要があるからである. このソート処理を典型的なバブルソートを用いて行った場合, 非常に巨大な回路 (5.6G gates 以上) を必要とする. この課題に対し報告者が検討を行った結果, 小さい回路 (329M gates) による実現が可能であるとの結論を得た[*10].

1.4.3 複数 LSI を用いた実現可能性

TWIRL を単一の LSI に実装することは不可能であることから, 前述の回路規模見積もりを用いて複数 LSI への分割実装を検討した. 検討にあたっては, 個々の LSI は 0.09 μm プロセスルールを用いて製造された 40 mm 角の大きさであり, 現時点 (2004 年 1 月) での標準的な仕様として, 回路規模 400M gates/LSI, IO ピン数 3000 pins/LSI (1500 bit を IO として使用可能[*11]) を仮定した. これら条件を元に Rational TWIRL を実装するのに必要な LSI 数を評価したところ, 3362 個の LSI が必要であるという結論を得た. LSI 間の IO ピン数を無視して回路規模のみの制約条件を考慮するならば 15 個の LSI で実現可能となることから, TWIRL の実装では

[*10] ただし報告者のソート処理のデザインは, 入力データに偏りが生じた場合非常に低い確率でデータ破棄が発生する. しかしこの確率は 0.000737 と非常に低く, 現実的には問題がない範囲であると考える.

[*11] 3000 pins のうち, IO として 1500 pins が使用可能であると仮定した (残りの pins は Vcc, GND 用である).

IO ピン数がボトルネックとなっていることがわかる. 3362 個の LSI を 1 枚のボードに実装することを考えると, 正方形に並べた場合, 58 × 58 で配置する必要ある. LSI の間隔を 10mm として実装するならば, およそ 3 m × 3 m のボードが必要となる. このように膨大な数の LSI を 1 枚のボードを実装・動作させることは, 現在の技術を遥かに超える範囲である. 以上により, 複数の LSI を用いて TWIRL を実現させた場合でも, LSI の IO ピン数がネックとなり, 必要となる LSI 数が膨大になり, 単一のボードに搭載して実現することは不可能であると考える.

1.4.4 複数ボードを用いた実現可能性

複数 LSI を用いた分割実装のさらなる可能性として, 複数ボードを用いた実装についても検討を行った. ボードあたりの LSI 数を 100 個と仮定した場合, Rational TWIRL の見積もり面積から算出した理論的なボード数は 54 枚となった. しかし TWIRL の実装にあたってはボード間に 1500 bit の IO を必要とし, 提案者の想定する周波数 (1 GHz) を考慮すると, これは現在の技術レベルを遥かに越えており, 実現は不可能であると考える[*12].

1.4.5 TWIRL 実現に必要なブレークスルー

これまでに述べたとおり, 既存技術だけを用いた場合, TWIRL を実際に実現することは極めて困難であると考える. しかし将来の技術進歩によって実現できる可能性は残されている. 前述の議論から, 単一の LSI を用いるよりも複数の LSI に分割する方が実現性は高いであろう. その場合 LSI 間およびボード間の IO 数がボトルネックとなっていたことから, TWIRL を実現するためには IO を高速かつ多ビット化する技術が必須であると考える[*13].

以下では IO を高速・多ビット化する技術の例と, それぞれの技術における問題点について簡単に説明する.

■シリアル-パラレル変換技術の高性能化 IO ピン数を仮想的に増加させる技術としてシリアル-パラレル変換 (シリ-パラ変換) が知られており, TWIRL の場合, 必要な LSI 数は LSI あたりの IO 数の 2 乗に反比例して小さくなることから, シリ-パラ変換を適用することで必要 LSI 数を減らすことができると考えられる.

しかしその代償として LSI 間 IO の周波数を上げる必要が生じる. 例えば 10:1 のシリ-パラ変換を用いた場合, LSI 間は内部の 10 倍の周波数で動作させる必要があり, TWIRL 提案者の主張に従い LSI 内部を 1 GHz で動作させた場合, LSI 間 IO は 10 GHz が必要となる. 現在実用

[*12] 1 GHz で動作する IO 数はおよそ 400 bit が限界と報告者は考える.
[*13] あくまでも必要条件であって, 十分条件ではない.

化されている 10 GHz I/F の消費電力を考慮すると, TWIRL の LSI 数の問題を解決できるほど多くの IO 数について適用することは, 当面は困難であろう.

■チップ間配線の微細化　チップ間の結線を微細化することによって, プリント版配線を用いた場合より IO 数を増加させることが可能となる. このような技術として ”Active Substrate Multi-chip Module Package” [Cha94] が知られている. しかし技術は論文レベルであり, 実用レベルでの適用例は知られていない. 本技術が現実的に使用可能であるか, さらには TWIRL に適用可能かであるかに関してはさらなる検討が必要である.

第 2 章

動作原理

本章は TWIRL の動作原理と仕様パラメータについてまとめる. まず 2.1 節で TWIRL の背景を述べた後, TWIRL が利用する数体ふるい法に関する準備を 2.2 節で簡単にまとめる. そして TWIRL の動作原理を 2.3 節で説明し, Shamir-Tromer によるコスト見積もりを 2.4 節で紹介する. TWIRL のさらなる詳細仕様については 2.5 節において触れる. また Largish Station における送信器の理論的サイズの数値例を 2.6 に挙げる.

本章の内容は TWIRL の提案論文 [ST03] に基づいており, ことわりのない限り報告者の主張はふくまれない. また本章の仕様パラメータは理論的なものであり, 実装に伴うさまざまな制約から, 実際の設計では異なる仕様パラメータを使用する必要がありえることにも注意が必要である (報告者の設計例については第 3 章を参照せよ).

2.1 位置付け

整数の素因数分解問題が困難であるという仮定は, 膨大な思考的・実験的経験から得られた結論であり, 情報セキュリティを考える上での大前提となっている. 実際, この前提を元にして RSA 暗号などのさまざまな暗号アルゴリズムが考案されてきている. 現時点 (2004 年 1 月, 以下同様) での素因数分解問題に対する最良のアルゴリズムは数体ふるい法 (NFS; Number Field Sieve method, [LL93]) であり, 2003 年 12 月にはソフトウェア実装による 576-bit 合成数の分解例が報告されている [F+03] (これは数体ふるい法による最大の分解例である). しかし現在の暗号で使用される合成数は 1024-bit 以上であり, ソフトウェア実装による同様なアプローチを用いた場合, 分解は非常に困難であることが予想されている. これに代わるアプローチとして, 最近になって専用ハードウェアを用いた方法が注目を集めている. 数体ふるい法は前処理 (多項式生成) ステップ, 関係式探索 (ふるい) ステップ, 線型代数ステップ, 後処理 (平方根計算) ス

テップの 4 つのステップから構成されているが, 専用ハードウェアが対象にしているのは, 数体ふるい法による素因数分解計算の大部分を占める関係式探索ステップと線型代数ステップである. 2001 年に Bernstein が提案した線型代数ステップに対するメッシュ回路を用いると, 1024-bit の合成数でも短時間で計算可能と報告されており [Ber01, LSTT02], 線型代数ステップは簡単な計算と見なされている. これに対し, 関係式探索ステップに対する専用ハードウェアとして TWINKLE [Sha99, LS00] やメッシュ回路 [GS03] が提案されたが, その効率は悪く, 1024-bit 合成数を扱うことは現実的には不可能であった.

2003 年 8 月にイスラエルの研究者 Adi Shamir と Eran Tromer によって提案された TWIRL (The Weizmann Institute Relation Locator) は, 関係式探索ステップに対する専用ハードウェアデザインの名称である [ST03]. TWIRL の動作原理は TWINKLE の延長上にあるものの, 汎用的な ASIC 技術を利用することで大規模な並列計算を可能とし, 大幅な計算資源の削減ができると主張されている[*1]. Shamir-Tromer の試算によると, TWIRL を用いた場合に必要なコストは, 1024-bit 合成数の場合で 1000 万ドル (約 10 億円) の費用[*2]と約 1 年の計算時間, 512-bit 合成数の場合で 180 ドル (約 1.8 万円) の費用と 4.5 時間程度の計算時間となる (ただし初期投資費用は除く) [*3]. これらの見積もりコストは従来のデザインよりも 1000 倍以上効率的であり, 実際に製造可能な範囲内でもあることから, 提案当初から大きな注目を集めていた. しかし見積もりの精度は決して高くはないため, 詳細な見積もり算出の必要性も指摘されている [RSA03]. 実際, 提案者もその必要性を提案論文内で認めている.

2.2 数体ふるい法

本節では TWIRL の前提である数体ふるい法, 特に関係式探索ステップに対するふるい処理について述べる. また TWIRL のベースとなった TWINKLE についても簡単に紹介する. 以下では, 整数の集合を $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots, \}$ で, 自然数の集合を $\mathbb{N} = \{1, 2, \ldots, \}$ であらわす.

2.2.1 アルゴリズム概要

整数の素因数分解問題に対する数体ふるい法 (NFS; Number Field Sieve method, [LL93]) は現時点での最良のアルゴリズムとして知られている [IK03]. 数体ふるい法は, 前処理 (多項式生

[*1] 最近になってメッシュ回路ベースの改良案も提案されている [GS04].

[*2] 1 ドル = 100 円として換算, 以下同様

[*3] Shamir らは別の論文で 0.09 μm テクノロジーにおける試算も示している. その場合, 1024-bit 合成数に対しては 110 万ドル (約 1 億 1000 万円) の費用と 1 年の計算時間が必要とされている [LTS+03].

成) ステップ, 関係式探索 (ふるい) ステップ, 線型代数ステップ, 後処理 (平方根計算) ステップの 4 つのステップから構成される. このうち素因数分解計算の大部分を占めるのは, 関係式探索ステップと線型代数ステップである.

TWIRL と TWINKLE は関係式探索ステップを処理するデバイスであるので, 以下では関係式探索ステップの処理内容だけを説明していく. 数体ふるい法の動作原理, 他のステップの処理内容, 関係式探索ステップと他のステップとの関係などに関しては, 数体ふるい法に関する既存の文書 ([LL93, IK03] など) を参照されたい.

2.2.2 ふるい処理

整数の組 (a', b) が以下の 3 条件を満たすとき, 組 (a', b) を関係 (relation) と呼ぶ.

1. $\gcd(a', b) = 1$
   ただし $\gcd(a', b)$ は整数 a', b の最大公約数を表す.
2. $f_{\mathrm{R}}(a', b)$ は B_{R}-smooth
   ただし 1 次の整数係数 2 変数多項式 $f_{\mathrm{R}}(x, y) = mx + y$, 自然数 B_{R} は与えられているとする.
3. $f_{\mathrm{A}}(a', b)$ は B_{A}-smooth
   ただし d 次の整数係数 2 変数多項式 $f_{\mathrm{A}}(x, y) = (-x)^d F(-y/x)$, d 次の既約な整数係数 1 変数多項式 $F(z) = c_d z^d + \cdots + c_0$ $(c_i \in \mathbb{Z})$, 自然数 B_{A} は与えられているとする.

ここで自然数 N が自然数 B に対して B-smooth であるとは, N の最大素因数が B 以下のこと, あるいは N が B 以下の素因数の積に分解できること, つまり N が

$$N = \prod_{p_i \le B} p_i^{e_i}$$

と表せることをいう. このとき自然数 B を smoothness bound と呼ぶ.

数体ふるい法における関係式探索ステップの目標は, 与えられた 2 次元領域

$$\{\, (a', b) \in \mathbb{Z} \times \mathbb{Z} \mid -R/2 \le a' \le R/2,\ 1 \le b \le H \,\}$$

から, 上の 3 条件を満たす関係 (a', b) を規定の個数以上見つけることである. このとき見つけるべき関係は規定の個数以上であればどのような組み合わせでも良く, また全て見つける必要もない. 定数 $R \in \mathbb{N}$ をふるい幅 (sieve line width), $H \in \mathbb{N}$ をふるいの本数 (sieve line number) と呼ぶ. 例えば 1024-bit 合成数に対する TWIRL では $R = 1.1 \times 10^{15}$, $H = 2.7 \times 10^8$ を使用する.

以下では多項式 $f_{\mathrm{R}}(a', b)$, $f_{\mathrm{A}}(a', b)$ および自然数 R, H, B_{R}, B_{A} は関係式探索ステップへの入力として与えられていると仮定する. また関係の条件 2. と条件 3. は類似しているため, 以下で

は簡単のため, 多項式 $f(a', b)$ は $f_{\mathrm{R}}(a', b)$ または $f_{\mathrm{A}}(a'b)$ のいずれかを, B は B_{R} または B_{A} のいずれかを表すこととする.

関係式探索ステップの基本的な動作は, 固定された b に対し, $f(a', b)$ が B-smooth となる a' を $\{-R/2, \ldots, R/2\}$ の中から見つけ出すことである. ここで $f(a', b)$ が B-smooth であるとは

$$f(a', b) = \prod_{p_i \leq B} p_i^{e_i} \tag{2.2.1}$$

となることであったから, $f(a', b)$ を B 以下の素数 p_i で順番に割っていくことで, $f(a', b)$ が B-smooth であるかを判定することはできる. しかし除算は大量の計算リソースを必要とするため, 効率的に処理するには好ましくない.

そこで TWIRL は, 以下で紹介するふるい (sieve) を用いることで, 関係の効率的な探索を可能にしている. まず式 (2.2.1) は

$$\log f(a', b) = e_i \sum_{p_i \mid f(a', b),\ p_i \leq B} \log p_i \approx \sum_{p_i \mid f(a', b),\ p_i \leq B} \lfloor \log p_i \rceil \tag{2.2.2}$$

と近似できることに注意する. ここで $\lfloor x \rceil$ は実数 x の四捨五入, すなわち x に最も近い整数を表す. 式 (2.2.2) のように近似できるのは, ほとんどの大きな素因数 p_i に対しては $e_i = 1$ となることと, メモリ効率の観点から対数の値を整数に丸めた方が好ましく, 誤差は後で調整可能なことが理由である. 実際の処理は以下のようにして行う:

> あらかじめ $f(a', b)$ に対応する変数 v (初期値 0) を用意し, 整数 $f(a', b)$ が素数 p_i で割り切れる場合, v に $\lfloor \log p_i \rceil$ を加える. この操作を B 以下の素数全てに対して行い, 最終的に v の値が $\log f(a', b)$ に近い場合, (a', b) を関係の候補として扱う. 候補となった $(a'b)$ に対しては, 正しい素因数分解を通じて, 本当に関係になっているかを判定する.

なお v の値が $\log f(a', b)$ に近いかどうかの判定は, v の値が近似値であることを考慮して, b によって定まる値 $T(b)$ に対し, $v > T(b)$ かどうかで判定する. $T(b)$ としては $\log f(a', b)$ の 0.8 倍程度の値を使用する. $T(b)$ を閾値 (threshold) と呼ぶ. 一般に閾値を緩くすれば, たくさんの a' が候補として得られる.

ふるいは上記の処理を同時に行うことが特徴である. いま多項式の選択方法から

> $f(a', b)$ が p_i で割り切れるならば, $f(a' + p_i, b)$ も p_i で割り切れる

という性質が成立する. そこで上記の処理をまとめて行うために, 始めに $f(-R/2, b)$, $f(-R/2 + 1, b), \ldots, f(R/2, b)$ に対応する $R + 1$ 個の変数 $v_{-R/2}, v_{-R/2+1}, \ldots, v_{R/2}$ (初期値は全て 0) を用意する. 次に, 素数 p_i に対し $f(-R/2, b)$ から順に p_i で割り切れるかを調べていく. $f(a'_0, b)$

表 2.1　ふるいの計算例

a'	-3	-2	-1	0	1	2	3
$f(a')$	150 $= 2\times 3\times 5^2$	151 $= 151$	152 $= 2^3\times 19$	153 $= 3^2\times 17$	154 $2\times 7\times 11$	155 5×31	156 $2^2\times 3\times 13$
$p=2$	$+\lfloor\log 2\rceil$	—	$+\lfloor\log 2\rceil$	—	$+\lfloor\log 2\rceil$	—	$+\lfloor\log 2\rceil$
$p=3$	$+\lfloor\log 3\rceil$	—	—	$+\lfloor\log 3\rceil$	—	—	$+\lfloor\log 3\rceil$
$p=5$	$+\lfloor\log 5\rceil$	—	—	—	—	$+\lfloor\log 5\rceil$	—
$p=7$	—	—	—	—	$+\lfloor\log 7\rceil$	—	—
$p=11$	—	—	—	—	$+\lfloor\log 11\rceil$	—	—
$p=13$	—	—	—	—	—	—	$+\lfloor\log 13\rceil$
$p=17$	—	—	—	$+\lfloor\log 17\rceil$	—	—	—
$p=19$	—	—	$+\lfloor\log 19\rceil$	—	—	—	—
合計	5	0	5	6	7	2	7

で初めて p_i で割り切れるとすると, $f(a'_0+p_i,b)$ も p_i で割り切れることになるので, 結局 $f(a'_0,b), f(a'_0+p_i,b), f(a'_0+2p_i,b),\ldots$ は p_i で割り切れることが分かる. よって素数 p_i に対しては, 変数 $v_{a'_0}$, $v_{a'_0+p_i}$, $v_{a'_0+2p_i},\ldots$ に $\lfloor\log p_i\rceil$ を加えれば良い. この操作を B 以下の全ての素数 p_i について繰り返していけば, $f(a',b)$ が B-smooth となる a' を $-R/2\le a'\le R/2$ の中からまとめて見つけることができる. このように a' をまとめて見つける操作が, ふるい (sieve) と呼ばれる理由であり, 数体ふるい法の名前の由来でもある. a'_0 が見つかってしまえば, 除算は必要ないことに注意が必要である.

■計算例　ふるい処理の具体的な例として, $f(a',b)=a'+51b$, $b=3$, $B=20$ の場合を考える. 簡単のため $f(a')=f(a',b)=a'+153$ と記す. $R=6$ すなわち $-3\le a'\le 3$ とすると, $f(a')$ の具体的な値は表 2.1 の第 2 行目のようになる (参考のため各値の素因数分解を第 3 行目に示したが, 実際のふるいではこれら素因数分解は行わない). $B=20$ 以下の素数 $p=2,3,5,7,11,13,17,19$ に対し, $f(a')$ が p で割れる場合に $\lfloor\log_2 p\rceil$ を加えていった結果が最終行である. 対数値の合計と $\log_2 f(a')$ の代表値 $\lfloor\log_2 f(0)\rceil=7$ を比べると, 両者の値が等しい $f(1)$ と $f(3)$ が候補として得られる. 実際 $f(1)=2\times 7\times 11$, $f(3)=156=2^2\times 3\times 13$ であり, どちらも確かに B-smooth となっている. なお $f(-3)$, $f(-1)$, $f(0)$ も B-smooth であるが, この例ではべき指数の影響が大きいため, ふるいの結果からは漏れてしまっている. しかし閾値を, 例えば $T(b)=\lfloor 0.8\times\log_2 f(0)\rceil=6$ と設定すれば, $f(0)$ は候補として得られる.

数体ふるい法では, 各 line に対して 2 種類のふるい処理が必要で, 1 つ目のふるいを有理数的

ふるい (rational sieve), 2 つ目のふるいを代数的ふるい (algebraic sieve) と呼んで区別する. そして 2 つのふるいのどちらにも残った (a', b) を候補 (candidate) と呼び, 候補はふるいの後に以下の最終チェックを受ける. 最終チェックでは 2 種類の関数値 $f_{\mathrm{R}}(a', b)$, $f_{\mathrm{A}}(a', b)$ を実際に分解し, 本当に B-smooth であるかを確かめる[*4]. そしてこのチェックにパスした (a', b) を関係 (relation) と呼ぶ. 関係式探索 (ふるいス) テップの出力は関係の集合である.

以下では簡単のため, また文献 [ST03] の記述に合わせるため, $a = a' + R/2$ ($a \in \{0, \ldots, R\}$) とする. また a', b の多項式 $f(a', b)$ と a, b の多項式 $f(a, b)$ を上の変数変換の下で同一視する.

2.2.3 TWINKLE

1999 年に Shamir が提案した TWINKLE (The Weizman INstitute Key Locating Engine) は, 前節のふるい操作に基く, 関係式探索ステップの高速処理を目的とした専用ハードウェア・デザインである [Sha99]. TWIRL は TWINKLE のアイディアに基づいているため, 以下簡単に TWINKLE について説明する. TWINKLE は光学素子を使用することを特徴とする. 構造は発光部と受光部に分かれ, 発光部は独立した多数のセル (LED) を持つウェハーとなっている. ここで各セルは素数 p_i に対応しており, 時刻 a のとき, $f(a, b)$ が素数 p_i で割り切れるような p_i に対応するセルは $\lfloor \log p_i \rceil$ に比例する光を放つようになっている. ウェハーの反対側の受光部には $f(a, b)$ に対応する変数 v_a を意味する光学素子が設置され, セルから放たれた光を受光する. 最終的な受光量 $\sum \lfloor \log p_i \rceil$ が閾値を越えた場合, その a は候補として残す. 各 LED が周期的に発光する (twinkle) ことが名称の由来である.

TWINKLE の長所は原理が簡単なことである. 各 LED は固有の周期で発光すれば良いので, ハードウェアによる実現も容易である. しかし欠点として, 受光部に光学素子 (GaAs, ヒ化ガリウム) を前提としているため費用が高価であり, さらなる効率化を目的として並列化した場合, 発光部と受光部をそれぞれ複数用意しなければならないことから, 並列化による効果が低いことが挙げられる. このため TWINKLE が 1024-bit の合成数のふるい処理を行うことは, 現実的には不可能であると評価されている [LS00].

2.3 基本構造

2003 年 8 月に Shamir と Tromer が提案した TWIRL (The Weizmann Institute Relation Locator) は, TWINKLE のアイディアに基いたふるい処理専用ハードウェアのデザインである [ST03]. TWINKLE の持つ材料的な問題に対しては汎用的な ASIC 技術を使用することで

[*4] 実際には 2+2 large prime variation を用いる [LL93].

対応し, 並列化の効果を上げるために, 1 組の Emission controller (発光部) が複数組の Bus Line (受光部) を担当するような設計とした. このため TWINKLE やメッシュ回路に比べても効率が大幅に向上し, 1024-bit の合成数であっても現実的な時間内でふるい計算が可能となったと主張されている.

TWIRL は, 関係式探索ステップの有理数的ふるいと代数的ふるいに対応して, 有理数的 TWIRL (Rational TWIRL) と代数的 TWIRL (Algebraic TWIRL) がそれぞれ提案されているが, パラメータ値以外の構造はほとんど同じであるため, 以下の説明では主に Rational TWIRL に絞って説明していく. ただし Algebraic TWIRL は Rational TWILRL に合格した a だけを処理するカスケード構造になっている (詳細は 2.5.6 節を参照).

以下では 1024-bit 合成数の素因数分解を目的としたパラメータを用いて説明する. 具体的なパラメータは 2 重括弧 $\langle\!\langle \cdot \rangle\!\rangle$ をつけて表す[*5]. そのパラメータが Rational TWIRL に関するならば $\langle\!\langle x \rangle\!\rangle_{\mathrm{R}}$, Algebraic TWIRL に関するならば $\langle\!\langle x \rangle\!\rangle_{\mathrm{A}}$, 両者に関すならば単に $\langle\!\langle x \rangle\!\rangle$ のように記す.

2.3.1 ふるいの並列処理

関係式探索ステップが処理するラインの本数を H $\langle\!\langle = 2.7 \times 10^8 \rangle\!\rangle$ 本, 各ラインの幅を R $\langle\!\langle = 1.1 \times 10^{15} \rangle\!\rangle$ とする. また smoothness bound を B $\langle\!\langle = 3.5 \times 10^9 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle = 2.6 \times 10^{10} \rangle\!\rangle_{\mathrm{A}}$ と定める.

TWIRL の大きな特徴は, 関係式探索ステップのふるい計算を並列に処理する点である. まず TWINKLE のように 1 clock cycle あたり 1 つのふるい位置 a を処理するデバイスを考える. 閾値を $T = T(b)$ とするとき, このデバイスは $\log_2 T$ $\langle\!\langle = 10 \rangle\!\rangle$ bit 幅の一方向性なデータバスを持ち, 何百万もの条件判断機能付き加算器 (conditional adder) を連続して持つ. それぞれの条件判断機能付き加算機は 1 つの Progression $P_i = \{\, a = a_0^{(i)} + kp_i \mid k = 0, 1, \dots \,\}$ を担当し, 動作時刻になるとバスに $\lfloor \log p_i \rceil$ を加える[*6]. データ $\lfloor \log p_i \rceil$ を p_i に対応する Contribution と呼ぶ. 時刻 t の時, z 番目の加算器は $t - z$ 番目のふるい位置を処理する. パイプラインには 1 clock cycle あたり 1 つの速さでふるい位置のデータが流れ, 0 を先頭に $1, \dots, R$ が流れていく (図 2.1 (a) 参照).

次にこのデバイスの並列化を行う. 先ほどと同じ $\log_2 T$ bit 幅のバスを s $\langle\!\langle = 4,096 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle = 32,768 \rangle\!\rangle_{\mathrm{A}}$ 本用意し, 時刻 t の時には z 番目の加算器の i 本目のバスは $(t-z)s+i$ 番目のふるい位置を処理するように変更する. ここで各パイプラインの先頭になるふるい位置を $0, 1, \dots, s-1$ とする (図 2.1 (b) 参照). このように変更することで, 1 clock cycle あたりに処理するデータを

[*5] これらパラメータは提案者の推奨値である.

[*6] ここで $\lfloor \log p_i \rceil$ は対数値 $\log p_i$ に最も近い整数 (つまり四捨五入) を表す.

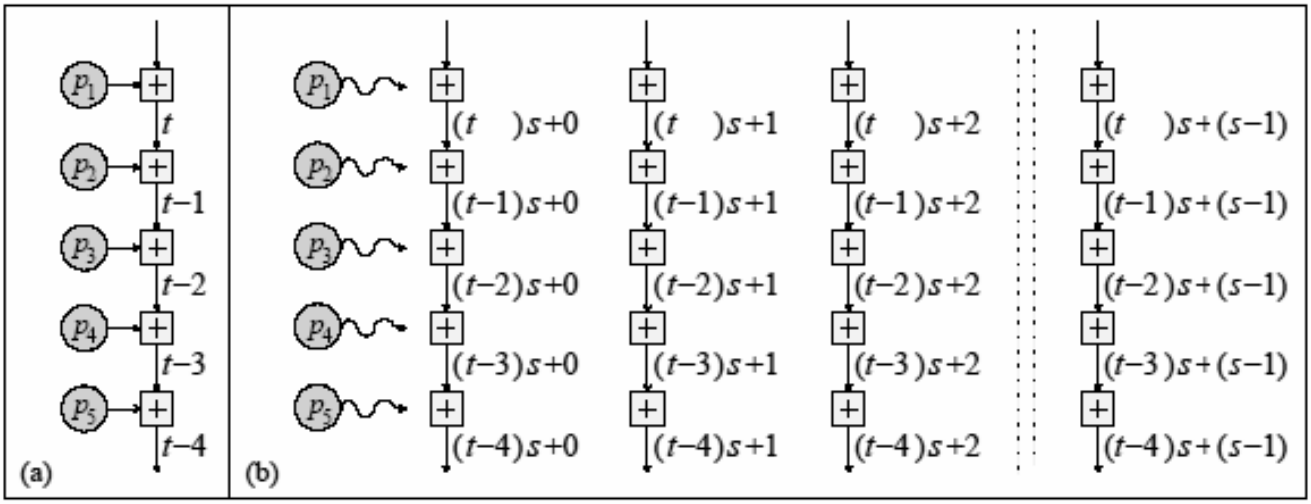


図 2.1　ふるい位置の配置 (a)=単一の加算器列 (TWINKLE), (b)=複数の加算器列 (TWIRL) [ST03]

s 個にすることが可能となる. ただし 2 つの問題が生じてしまう. 1 つは時刻の進みが $1/s$ になるため, それに応じた処理が必要になることである. もう 1 つは s 本のバスを同時に並列処理するために, 同じ Progression P_i に対応する加算 $\lfloor \log p_i \rceil$ を同時に行う必要が生じることである[*7]. そこで同じ Progression P_i を担当する送信器を複数使用するのだが, 素数によって頻度が異なるため, 素数の大きさによって 3 種類の Station と呼ばれるユニットを併用する. 具体的には, 大きな素数 (Largish Prime) に対しては Largish Station, 小さな素数 (Smallish Prime) に対しては Smallish Station, とても小さな素数 (Tiny Prime) に対しては Tiny Station を用いる[*8]. 各 Station は決められた範囲の素数を担当し, パイプライン状に連結されている. 最後の Station の出力後に閾値判定を行うユニットを配置し, その出力をデバイスの出力とする. Rational/Algebraic TWIRL の概念図を図 2.2 に示す. ただし Station によって Progression の送信器の構造が異なるため, 図では省略した.

素因数分解 (の関係式探索) 装置としての TWIRL は, 複数個の Rational TWIRL が 1 個の Algebraic TWIRL に連結され, クラスタとして機能する. 1024-bit 合成数に対しては, 《8》 個の Rational TWIRL が 1 個の Algebraic TWIRL に連結される (図 2.3 参照). クラスタの出力は関係 (a, b) の集合である. なお提案者によれば, 1024-bit 合成数の分解に必要なクラスタ数は

[*7] もちろん Progression 送信器を s 組用意すれば良いが, それでは並列化による効果は不十分である (s 個の TWINKLE を同時に処理しているのと同等の効果しか得られない).

[*8] 数体ふるい法では Large/Small Prime という概念もあるが, Largish/Smallish/Tiny Primes とは別の概念であり, 注意が必要である.

《194》組と試算されている (2.4.1 節参照).

回路の大部分を占めるのは 3 つの Station であるので, 以下, 各 Station の構造を簡単に説明する. 詳細な構造やパラメータについては 2.5 節を参照されたい. なおふるい制御部と閾値判定部については説明を省略する[*9].

2.3.2 Largish Station

素数 p_i に対応する Progression P_i の送信器は, Contribution $\lfloor \log p_i \rceil$ を周期 p_i/s で発行する. 従って $p_i/s \gg 1$ の時の発行頻度は非常に小さいため, 図 2.1 のように 1 個の送信器が 1 組の Progression を担当する構造は非常に効率が悪いといえる. そこでこのような場合に, 送信器は複数の Progression を担当し, Progression 情報の通り道となる Delivery Line を複数の素数で共有するようにすれば, 大幅な回路削減に直結する. これが Largish Station の基本的なアイディアである. 実際には素数 p_i が $\langle\!\langle 5.2 \times 10^5 < p_i < 3.5 \times 10^9 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 4.2 \times 10^6 < p_i < 2.6 \times 10^{10} \rangle\!\rangle_{\mathrm{A}}$ の場合を Largish Prime と呼び (このとき $127 < p_i/s_{\mathrm{R}} < 854493$, $127 < p_i/s_{\mathrm{A}} < 793458$ となる), 以下の構造を用いる.

図 2.4 は提案者による Largish Station のデザインである. ここで Largish Station は, Progression 送信器 (DRAM Memory + Cache + Processor), Buffer, Delivery Line から構成される. C $\langle\!\langle = 8,490 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle = 59,400 \rangle\!\rangle_{\mathrm{A}}$ 個の送信器は唯一の Buffer に接続され, Progression 情報は送信部から Buffer を経て Delivery Line に送られる. Progression は Progression Triplet として表現され, Memory に保存されている. Progression Triplet は Processor によって周期的にチェックを受け, 該当する場合には情報が Emission Triplet として Buffer に送信されるとともに, 新しい状態情報に更新される. Buffer は複数のプロセッサから送信された Emission Triplet を受け取り, 時刻情報の順に保持し, 適当なタイミングで Delivery Pair を発行する. Delivery Pair はパイプライン化された Delivery Line (Cell が鎖状に連結したもので, 各 Cell はデータの転送と加算器の機能を持つ) に送信される.

以下, 各部分を詳細に説明する.

■Progression 情報の更新　Progression 情報は送信器内の Memory に保管されている. ここで Largish Prime に対する Progression の集合 $\{P_i\}$ は分割して送信器に保管され, 重複はないものとする. j 番目の送信器が保持する Progression の個数を d_j 個とする (ただし $\langle\!\langle 32 \le d_j \le 2.2 \times 10^5 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 32 \le d_j \le 2.0 \times 10^5 \rangle\!\rangle_{\mathrm{A}}$ で, 個数は送信器によって異なる). Progression

[*9] TWIRL の提案論文でも詳細は示されていない.

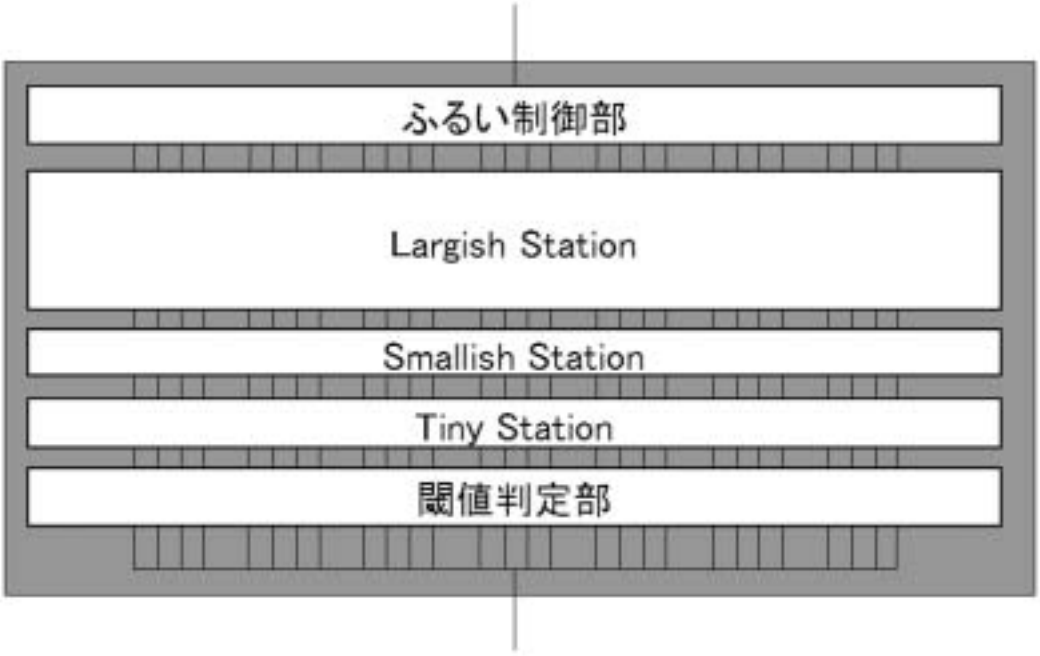


図 2.2　Rational/Algebraic TWIRL の概念図

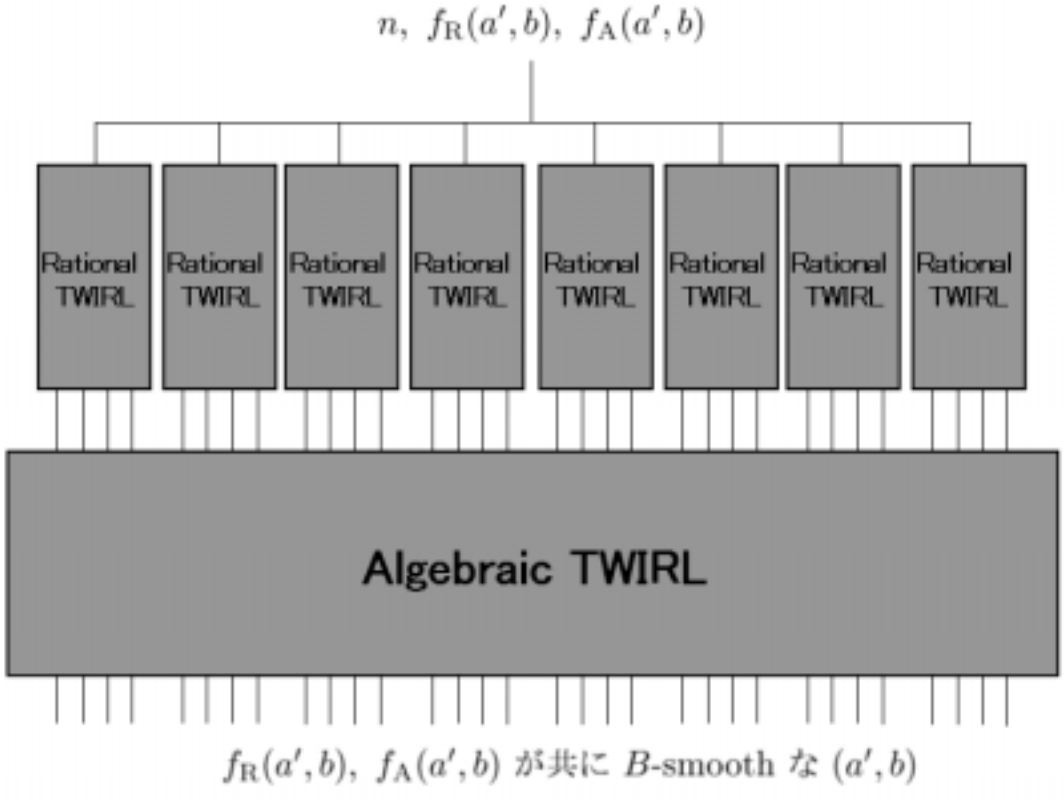


図 2.3　TWIRL クラスタ

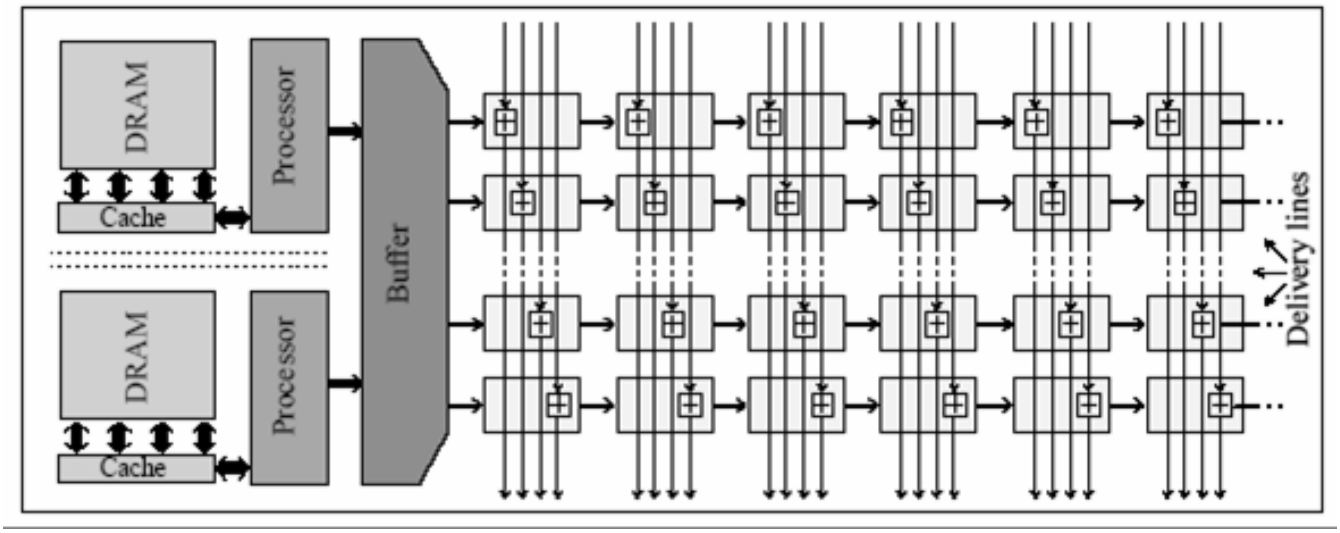


図 2.4　Largish Station の構造 [ST03]

Triplet は $(p_i, \ell^{(i)}, \tau^{(i)})$ で表されるデータの組で, p_i は素数の値, 次の発行情報 $a^{(i)} \in P_i$ に対して $\tau^{(i)} = \lfloor a^{(i)}/s \rfloor$ は Contribution が足される時刻, $\ell^{(i)} = a^{(i)} \bmod s$ は Contribution が足される Bus Line 番号を表す. (ただし $a^{(i)}$ は陽には保管されていない).

Processor は以下の操作をパイプライン方式で実行する:

1. Progression Triplet $(p_i, \ell^{(i)}, \tau^{(i)})$ を読み込み, このデータを Memory から消去する.
2. Emission Triplet $(\lfloor \log p_i \rceil, \ell^{(i)}, \tau^{(i)})$ を Buffer に送信する.
3. $\tilde{\ell}^{(i)} \leftarrow (\ell^{(i)} + p_i) \bmod s$, $\tilde{\tau}^{(i)} \leftarrow \tau^{(i)} + \lfloor p_i/s \rfloor + w$ を計算する. ただし $w = 1$ if $\tilde{\ell}^{(i)} < \ell^{(i)}$, $w = 0$ otherwise.
4. Progression Triplet $(p_i, \tilde{\ell}^{(i)}, \tilde{\tau}^{(i)})$ を Memory に書き込む.

Emission Triplet $(\lfloor \log p_i \rceil, \ell^{(i)}, \tau^{(i)})$ が Buffer に送信されるのは, 時刻 τ_i の直前であって欲しい (早すぎると Buffer のメモリがあふれてしまうし, 遅すぎると Delivery Line への送信に間に合わない). 従って Processor が読み込む Progression Triplet は発行直前のものを読み込むようにしたい. 各 Progression Triplet の読み書き位置を固定し, 必要な情報をスキャンする単純な方式は, 今のように Progression Triplet の個数 d_j が大きいときには非効率的である. 次節のように時刻 $\tau^{(i)}$ によってインデックス付けされた位置に保管することで, Memory 上で効率的な配置が可能となる.

■Progression **の保管方法**　Processor は Memory (DRAM) 上に記憶された Progression Triplet を一定の速度で連続的・巡回的に読み込む (《 1 triplet あたり 2 clock cycle 》). 読み込んだデータが空の場合には何もせずに次のデータを読み込む. そうでない場合, Processor は前述のようにして新しいデータを計算し, 適当な位置, つまり時刻 $\tilde{\tau}^{(i)}$ の少し前に読み込まれる位置に更新されたデータを書き込む. このようにして Processor は発行直前の Progression Triplet を読み込んでいく. Progression Triplet は最初の空の位置に書き込まれる (ただしインデックスをさかのぼる) ようにして, 書き込み位置が空である確率を高めるために, Memory は必要量の《2》倍を確保する.

もしも適切な場所から《64》個以内に空の場所がない場合[*10], Progression Triplet は任意の場所 (または専用のオーバーフロー領域) に書き込まれ, 後に適切な位置に移動するようにするが, 場合によってはこれらの Emission が失われる可能性もある. しかしこのようなことが起きる確率は希であるし, 例え失われたとしても十分に許容範囲内である (つまり関係式探索ステップで関係を規定個数以上集められる) と提案者は主張している.

j 番目の送信器は d_j 個の連続した素数 $\{p_{\min}, \ldots, p_{\max}\}$ を担当するとしよう. この情報を記録するのに必要な Memory 量は $p_{\max}/s$ ワード (ただし 1 ワードは 1 個の Progression Triplet を記録するのに必要なデータ量) である. Progression Triplet を読み込んだ際, $p_i = p_{\max}$ の場合には, 更新後の Progression Triplet は読み込み位置から最も離れている. p_i がそれより小さい場合には, だんだんと読み込み位置に近づくことになり, $p_i = p_{\min}$ の場合が最も近い. 素数定理によると $p_{\max}/s - p_{\min}/s \approx d_j \times \log_e(p_{\max})/s$ となるから, 更新後の書き込み位置の幅は

$$d_j \times \log_e(p_{\max})/s < d_j \times \log_e(B)/s \ll d_j$$

となり, その幅は非常に狭い (図 2.5 参照). したがって Memory をシリンダ状に配置し, 小さなウィンドウが一定の速度で Memory の周りを動くようにすれば, 読み書きに伴う Memory へのアクセスはウィンドウを通して行われる[*11]. シリンダ状の Memory は素数の個数に応じてさまざまな大きさで各送信器に配置される.

ウィンドウは SRAM ベースの高速なキャッシュによって実現できる. 最も古いデータを DRAM に書き込むことと, 次の DRAM のデータを読み込むことによって, キャッシュのデータはシフトされていく. よって SRAM と DRAM の間に適切なインターフェイスを用いれば, DRAM の高レイテンシ[*12] を緩和することが可能となると同時に, DRAM の構造が簡単

[*10] 1 つの Progression Triplet が必要とするメモリ量を 1 個と表現している.

[*11] これがデバイス名 ”twirl”(回転する) の本当の由来であろう.

[*12] データの読み出し要求を行ってから, 実際にデータが転送されるまでにかかる遅延時間.

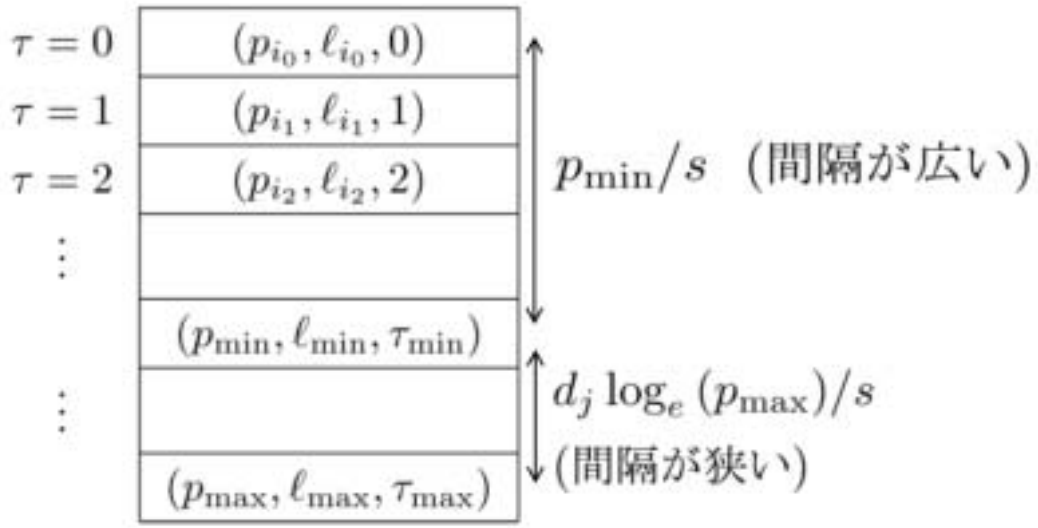


図 2.5　Progression の保管

に, さらには DRAM は少量で済むことになる. なおキャッシュミスは起こり得ない. なぜなら Processor と Memory の間にはインターフェイスは「次の Memory 位置を読む」と「与えられたアドレスよりも前の最初の空の Memory 位置に Triplet のデータを書き込む」だけであり, 後者はキャッシュを用いて回路が実現できるからである.

■送信器の個数　Rational/Algebraic TWIRL が使用する送信器の個数 C $\langle\!\langle = 8,490 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle = 59,400 \rangle\!\rangle_{\mathrm{A}}$ の算出理由を簡単に説明する. j 番目の送信器が担当する, 連続する d_j 個の素数を $\{p_{\min}, \ldots, p_{\max}\}$ とすると, 送信器が用意すべきメモリは $\lceil p_{\max}/s \rceil$ ワードとなる. このメモリ領域に d_j 個の Progression Triplet を格納することになるが, メモリからの読み込みに 2 clock cycle 必要なことと, メモリは必要量の 2 倍を用意することから, $d_j \times 2 \times 2 = p_{\max}/s$ でなければならず, 結局 $4s \times d_j = p_{\max}$ なる関係式が成立しなければならない. あとは数値的に調べていくことで, 各送信器が保持する Progression の個数 d_j と, 送信器の総数 C が得られる. Rational/Algebraic TWIRL に対する計算例 (抜粋) を 2.6 節に挙げる. なお計算値と提案者の推奨値が異なっているが, 理由は不明である.

■Buffer　Buffer は複数の Processor から Emission Triplet を受け取り, Delivery Line に対して Delivery Pair を送信する. 従って Buffer の中心となる動作は, 受け取った Emission Triplet を

時刻情報を基に適切に配置し, 適切なタイミングで変換した Delivery Pair を Delivery Line に送信することである. つまり Emmision Triplet $(\lfloor \log p_i \rceil, \ell^{(i)}, \tau^{(i)})$ は, Delivery Pair $(\lfloor \log p_i \rceil, \ell^{(i)})$ に変換され, $\ell^{(i)}$ に応じて適切な Delivery Line に時刻 $\tau^{(i)}$ の時に送信される.

これらの条件を満足させるために, TWIRL では以下のように Buffer を設計する. Buffer は複数のキューとパイプライン状のネットワークから構成される. まず入力される全ての Emission Triplet は時刻情報をインデックスとしたキューに入力される. ここでキューはメッシュ状になっていて, 行方向では常にバブルソートが, 列方向では常にランダムシャッフルが行われている. 最後の数行のデータの時刻情報は現在の時刻情報と比較され, マッチしたデータはパイプライン状のネットワークにおいて Bus Line 番号をインデックスとして並べ換えられた後, 適当な Delivery Line に対して送信される. 輻輳の影響により Delivery Line への送信が遅れることもあり得るが, その場合にはデータを破棄することにする (しかし適切なパラメータの下では, このようなことが起きるのは極めて希であると提案者は主張している).

Buffer のサイズは, 送信前の Emission Triplet をどれだけ保持するかということ (これは Processor の構造から小さい値となる) と, 各プロセッサが発行する Progression Triplet の速度 (およそ $\langle\!\langle$ 4 clock cycle / triplet $\rangle\!\rangle$) によって決まる.

■Delivery Line　Delivery Line は Buffer から Delivery Pair $(\lfloor \log p_i, \ell^{(i)} \rceil)$ を受け取り, $\lfloor \ell^{(i)}/k \rfloor$ clock cycle 後に加算を行う (ここで Delivery Pair は 1 clock cycle あたり k $\langle\!\langle = 4 \rangle\!\rangle$ 本の Bus Line を進むとする). Delivery Line は Bus Line と直交する Cell の 1 次元配列として実装され, Cell は 1 組の Delivery Pair を保持することができる. Cell は自分の Bus Line 番号 $\ell^{(i)}$ を比較し, 等しい場合には Bus Line に $\lfloor \log p_i \rceil$ を加えてから Deliver Pair を破棄する. 等しくない場合には, シフトレジスタのように, 隣の Cell に Delivery Pair を送る.

各送信器から発行される Progression Triplet は, 平均的には 4 clock cycle につき 1 個であるから, Delivery Line は送信器のの 1/4 程度の割合で用意すれば良い. 提案者による Delivery Line の本数は $\langle\!\langle 2, 100 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 14, 900 \rangle\!\rangle_{\mathrm{A}}$ 本であり, [*13]. 装置の大部分を占めることになる. コスト削減のために, interleaving と carry-save 加算器が利用可能であることを 2.5.1 節で, さらには代数的ふるいにおいてはこれら加算器がほとんど不要となることを 2.5.6 節で説明する.

■Progression Triplet のサイズ　Progression Triplet (p_i, ℓ, τ) を保持するには, データの有無を表すフラグに 1 bit, 素数 p_i を保持するのに $|p_i|$ $\langle\!\langle = 32 \rangle\!\rangle_{\mathrm{R}}$ bit, Bus Line 番号 ℓ を保持するのに $|s_{\mathrm{R}}|$ $\langle\!\langle = 12 \rangle\!\rangle_{\mathrm{R}}$ bit が必要である. また時刻情報 τ については, 前述の Processor, Memory, Buffer

[*13] Rational TWIRL における Delivery Line の本数は, 論文には ”2,100120” 本と記載されているが, 正しくは ”2,100” 本 (程度) である. これは回路規模見積もりからの逆算からも明らかであるし, 実際 TWIRL の提案者である Eran Tromer 氏に確認したところ, 誤りを認めている.

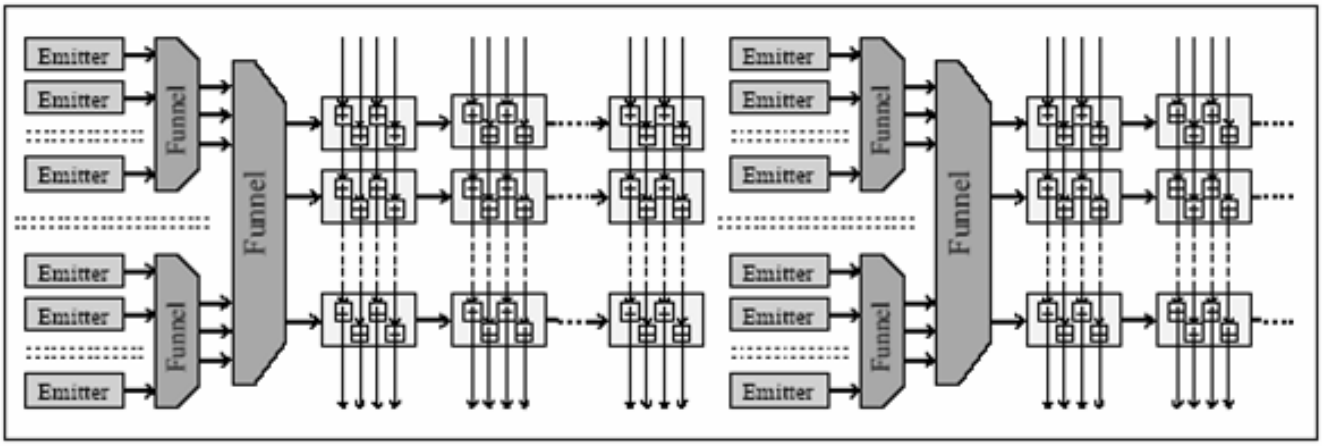


図 2.6　Smallish Station の構造 [ST03]

の説明では clock cycle を表す任意の整数であることを仮定していたが, τ は Progression Triplet が Memory から読み込まれてから, Buffer から送信されるまでの時間を表せれば良いので, 適当な整数 $\langle\!\langle 2048 \rangle\!\rangle$ による剰余で現実的には十分である. よって τ を保持するには $\log_2 2048 = 11$ bit で良い. さらに初期化 (2.5.4 節) で必要となる $|p_i|$ $\langle\!\langle = 32 \rangle\!\rangle_{\mathrm{R}}$ bit を考慮すると, Progression Triplet のサイズは $12 + 2|p_i| + \log_2 s$ $\langle\!\langle = 88 \rangle\!\rangle_{\mathrm{R}}$ bit となる.

2.3.3　Smallish Station

送信器が担当する素数 p_i が s と同じくらいの大きさの時 ($\langle\!\langle 256 < p_i < 5.2 \times 10^5 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 256 < p_i < 4.2 \times 10^6 \rangle\!\rangle_{\mathrm{A}}$ すなわち $\langle\!\langle 0 < p_i/s < 127 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 0 < p_i/s < 129 \rangle\!\rangle_{\mathrm{A}}$), Progression の発行周期は Largish Station の時に比べて短くなる. 送信器は $\langle\!\langle 2 \rangle\!\rangle$ clock cycle 毎に高々 1 個の Progression しか発行できないので, Smallish Prime に対しては各送信器が担当する素数をあまり多くすることができない. つまり Largish Station のような構造はコスト的に不利になってしまう. また Progression が頻繁に発行されることから, 各 Progression が一ヶ所から発行される場合, 遠くの Bus Line まで送信するコストは大きい. 従ってこれらの Smallish Prime に対しては, Largish Station とは異なる別のデザインが必要であり, これが Smallish Station を提案する理由となる. 提案者によるデザインを図 2.6 に示す.

以下, 簡単に Smallish Station の構造を説明する.

■Emitter　Smallish Station における Progression の送信器を Emitter と呼ぶ. Emitter は単一の Progression を担当し, Delivery Pair を生成する小さな回路である. Emitter は Progression

の状態を内部に持つレジスタに保持し, 適宜 Delivery Pair を発行する. ここで Delivery Pair $(\lfloor \log p_i \rceil, \ell^{(i)})$ は, 一定時刻後に $\ell^{(i)}$ 番目の Bus Line に対して $\lfloor \log p_i \rceil$ が加算されることを表す. Emitter の効率的なデザイン法については 2.5.2 節を参照されたい.

Smallish Station では発行周期が短いため, 同じ素数 p_i を担当する Emitter を複数用意して, 各 Emitter の周期を増加させる. これを Emitter の重複 (duplication) と呼ぶ. ただし素数 p_i に対応する Emitter の個数を $n_i = \lfloor s/2\sqrt{p_i} \rfloor_2$ 個とする ($n_i = \langle\!\langle 2, 4, 8, 16, 32, 64 \rangle\!\rangle_{\mathrm{R}}$) [*14]. 各 Emitter は内部のカウンター情報によって随時更新され, Delivery Pair を T_i ($= \lfloor \sqrt{p_i}/2 \rfloor_2$) clock cycle に 1 回発行する ($T_i = \langle\!\langle 8, 16, 32, 64, 128 \rangle\!\rangle_{\mathrm{R}}$).

Emitter の重複によって, Delivery Pair の生成源から目的の Bus Line までの距離を短くすることができる (図 2.7 参照). 各素数に対応する Progression は n_i 個の Emitter によって保持され, これら Emitter はバス上に一定間隔で配置される. これに呼応して Delivery Line を s/n_i ($\approx 2\sqrt{p_i}$) 本ずつの Segment に分割する. 各 Emitter は異なる Segment に接続され, この Segment に対して Progression を T_i ($= n_i p_i / s \approx \sqrt{p_i}/2$) clock cycle ごとに送信する. Progression が到着するまでの時間は短縮されるので, Delivery Line の本数を減らすことができる. またいずれの Emitter においても Emission の周期は短くなるので, Smallish Station では s 程度の (あるいはそれよりも小さい) 素数 p_i を扱うことができる. Smallish Station 全体で必要な Delivery Line の本数はおよそ $\langle\!\langle 501 \rangle\!\rangle_{\mathrm{R}}$ 本で, これら Delivery Line は異なるサイズの Segment に分割される.

■Emitter の個数　Rational/Algebraic TWIRL のそれぞれにおいて, 素数 p_i の大きさに応じた各変数の値を表 2.2, 表 2.3 に示す. ただし Rational TWIRL では $\sum \#p_i = 43007$, $\sum n_i \#p_i = 184562$, Algebraic TWIRL では $\sum \#p_i = 296260$, $\sum n_i \#p_i = 476779$ となっている.

■Delivery Line の本数　各素数 p_i に対応する Progression の送信周期は $T_i \approx \sqrt{p_i}/2$ であるので, 1 clock cycle あたりに送信される Contribution は, 平均で $\sum 1/T_i$ 個となる. Funnel の構造を考慮すると, Delivery Line としてこの 2 倍程度の本数を用意すれば良い (2.5.3 節参照). 実際, 具体的に計算すると,

$$\text{Rational} : 2 \times \sum_{256 < p_i < 5.2 \times 10^5} \frac{1}{T_i} \approx 2 \times \sum_{256 < p_i < 5.2 \times 10^5} \frac{2}{\sqrt{p_i}} = 2 \times 254.6 = 509.2$$

$$\text{Algebraic} : 2 \times \sum_{256 < p_i < 4.6 \times 10^6} \frac{1}{T_i} \approx 2 \times \sum_{256 < p_i < 4.6 \times 10^6} \frac{2}{\sqrt{p_i}} = 2 \times 624.4 = 1248.8$$

となる. ここで計算値と提案者の推奨値が異なっているが, 理由は不明である.

[*14] $\lfloor x \rfloor_2$ は x を 2 のべきに丸めた値, すなわち $\lfloor x \rfloor_2 = 2^{\lfloor \log_2 x \rfloor}$ を表す.

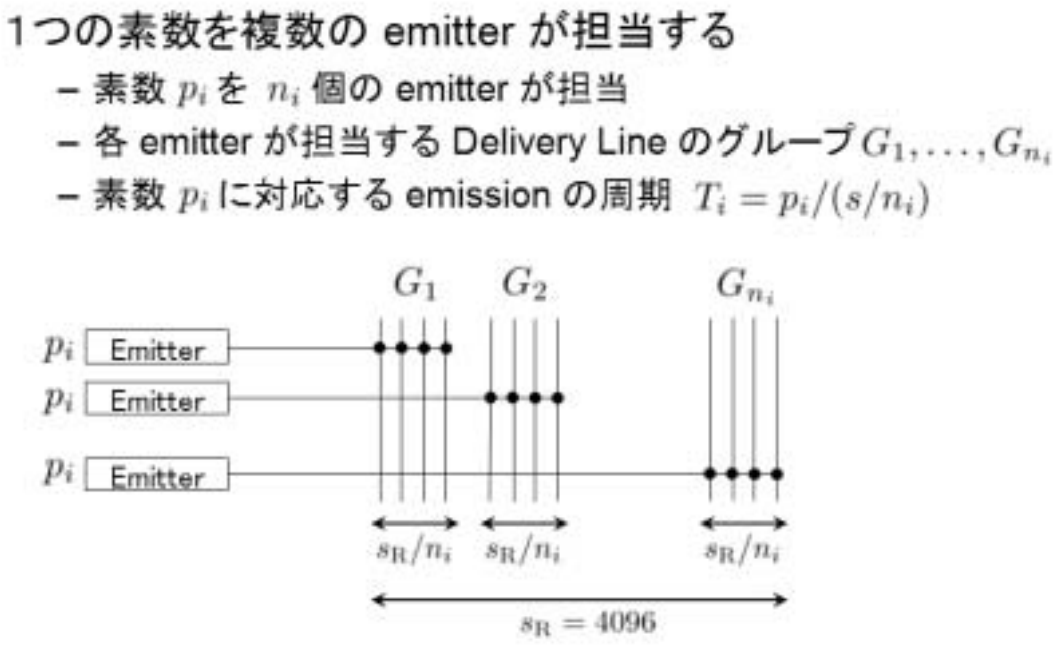


図 2.7　Segment 分割

表 2.2　Smallish Station のパラメータ (Rational TWIRL)

p_i	2^8	$\cdots$	2^{10}	$\cdots$	2^{12}	$\cdots$	2^{14}	$\cdots$	2^{16}	$\cdots$	2^{18}	$\cdots$	5.2×10^5
$\#p_i$		118		392		1336		4642		16458		20061	
n_i		64		32		16		8		4		2	
$\lvert G_i \rvert$		64		128		256		512		1024		2048	
β_i		6		7		8		9		10		11	
T_i		8		16		32		64		128		256	

表 2.3　Smallish Station のパラメータ (Algebraic TWIRL)

p_i	2^8	$\cdots$	2^{10}	$\cdots$	2^{12}	$\cdots$	2^{14}	$\cdots$	2^{16}	$\cdots$	2^{18}	$\cdots$	2^{20}	$\cdots$	4.2×10^6
$\#p_i$		118		392		1336		4642		16458		59025		214289	
n_i		64		32		16		8		4		2		1	
$\lvert G_i \rvert$		64		128		256		512		1024		2048		4096	
β_i		6		7		8		9		10		11		12	
T_i		8		16		32		64		128		256		512	

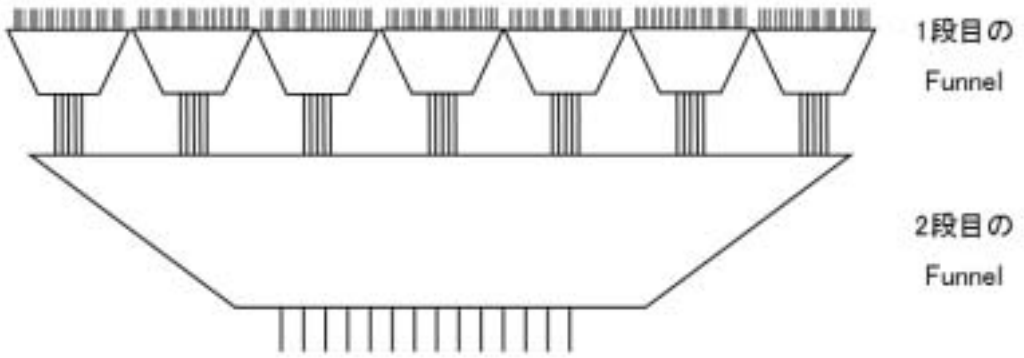


図 2.8　Funnel

■Funnel　Smallish Station でも Largish Station と同様に複数の Progression で Bus Line を共有するが, 各 Emitter を全ての Bus Line と接続するのは無駄が多い. そこで Funnel と呼ばれる 2 段のユニットを用いて, まばらな入力を以下のようにして圧縮する (図 2.8). 各 Funnel の入力は多数の Emitter の出力に接続される (各 Emitter の接続先となる Funnel は 1 個). Funnel の入力は 1 次元配列 (ただしそのほとんどの成分は空) と考えることもできる. Funnel の出力も 1 次元配列だが, その長さはずっと短く, 出力の配列の空でない成分の個数は, 何 clock cycle か前の入力となった配列の空でない成分の個数と同一になっている. Funnel の出力は Delivery Line に接続される. 改良版シフトレジスタを使用した Funnel の実装法を 2.5.3 節で述べる. なお Funnel は Largish Station における Buffer に相当するユニットであるが, 時間情報によるソート機能が省略されている.

2.3.4　Tiny Station

とても小さな素数《$p_i < 256$》に対しては, Buffer や Funnel のコストは不利である. またこのような素数は各 clock cycle ごとに複数個の Emission を必要とするので, いくつかの Progression に Delivery Line を割り当てるのは得策でない. 従ってこれら Tiny Primes に対しては 3 つめのデザインが必要となる. 提案者による Tiny Station のデザインを図 2.9 に示す. Tiny Station では Smallish Station と同様に, 同じ素数に対して複数の Emitter を用意する.

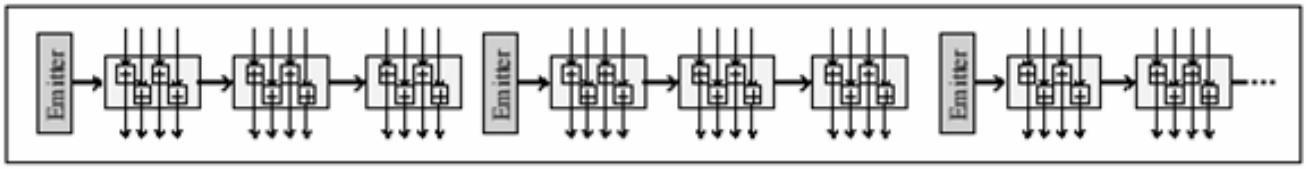


図 2.9　Tiny Station の構造 [ST03]

2.4　提案者によるコスト見積もり

本節では, 提案者による TWIRL のコスト見積もりを示す. 以下では現在の LSI テクノロジーを前提とし, 素因数分解問題のサイズごと (1024-bit, 512-bit, 768-bit) に算出する. ただしこれら見積もりは大まかなものであって, 多数の概算値と仮定を用いているため, 提案者も認める通り現実のコストとは定数のオーダーで異なる可能性があることを考慮しなければならない. 本節の内容の正当性を検証することが本報告書の主目的の一つである.

本節の見積もりで使用する TWIRL のパラメータを表 2.4 にまとまる.

2.4.1　1024-bit 合成数

1024-bit 合成数に対しては, 2.3 節の基本構造に加え, ふるいのカスケードを用いることを仮定する (2.5.6 節参照). 概算によると, 1 つの Rational TWIRL が必要とするシリコンの面積は 15,960 mm^2 (30 cm シリコンウェハー (約 66,000 mm^2) の 1/4) となる. このうち Largish Station は 76 % (DRAM は全体の 37 %), Smallish Station は 21 %, Timy Station は残りである 3 % を占めている. 同様に, 1 つの Algebraic TWIRL が必要とするシリコンの面積は 65,900 mm^2 (30 cm シリコンウェハー 1 枚) であり, このうち Largish Station は 94 % (DRAM は全体の 66 %), Smallish Station は 6 % を占めている. この他の詳細なパラメータは 2.5 節を参照されたい.

1 組の TWIRL クラスタは, 8 個の Rational TWIRL と 1 個の Algebraic TWIRL から構成され, 全体で必要なシリコンの面積はウェハー 3 枚 ($1/4 \times 8 + 1 = 3$) となる. 各 Rational TWIRL は Algebraic TWIRL に対して単方向的接続をしていて, 13 bit/clock cycle でデータを転送する. クラスタは 1 本のラインをふるうのに, スループットで R/s_{A} clock cycle (周波数 1GHz で約 33.4 秒) を必要とするので, 全領域 (H 本のライン) をふるうには, 約 286 年 ($33.4 \times H = 9.02 \times 10^9$ 秒) が必要である. さらに 2.5.5 節で述べる高速化技法を用いると約 33

表 2.4　TWIRL のパラメータ

パラメータ	意味	1024-bit	768-bit	512-bit
R	Line 幅	1.1×10^{15}	3.4×10^{13}	1.8×10^{10}
H	Line 数	2.7×10^{8}	8.9×10^{6}	9.0×10^{5}
B_{R}	Smoothness bound (有理数側)	3.5×10^{9}	1.0×10^{8}	1.7×10^{7}
B_{A}	Smoothness bound (代数側)	2.6×10^{10}	1.0×10^{9}	1.7×10^{7}

% の高速化が可能であり, 約 194 年 (= 286×0.67) に改善できる. 従って 194 組のクラスタを用いれば, 約 1 年でふるいが終了することになる. ウェハー 1 枚の製造コストを 5,000 ドル (約 50 万円) と仮定した場合, このときの製造コストは合計で約 290 万ドル (約 2 億 9000 万円) となる.

パッケージ, 電力供給, 冷却装置などの費用を考慮し, エラーマージンを加味すると, 1024-bit 合成数の素因数分解に必要なふるいを 1 年で終了させるのに必要な製造コストは, およそ 1000 万ドル (約 10 億円) となる[*15]. この他に設計, シミュレーション, マスク製造などに対し, 初期開発費 (NRE; Non-Recurring Expenses) として 2000 万ドル (20 億円) 程度が必要となる. 以上をまとめると表 2.5 のようになる.

■詳細な回路規模見積もり　ハードウェアに関する詳細パラメータ (2.4.4 節) をもとにした, 1024-bit 合成数用の TWIRL の回路規模の詳細見積もりを表 2.6 (Rational) と表 2.7 (Algebraic) に示す.

■0.09 μm での見積もり　Shamir 等はその後の論文において, プロセスルールを 0.09 μm に変更した場合の見積もりを示している [LTS+03]. その論文によると, 変更に伴って回路面積は約 1/2 倍, 速度は約 2 倍となり, パラメータを改良することで, 全体として約 9 倍のコスト改良が可能であるとしており, 0.13 μm の時と同じクラスタ数が 110 万ドル (約 1 億 1000 万円) で製造可能と結論づけている (必要な計算時間は 1 年のまま変わらない).

2.4.2　512-bit 合成数

512-bit 合成数に対しては, 2.3 節の基本構造を用いる. ただし B_{R}, B_{A}, R, H, s_{R}, s_{A} 以外のパラメータは 1024-bit の場合と同じ値を用いた. このとき 1 枚の TWIRL チップが必要とする

[*15] このような場合, 通常はシリコンの製造コストの 2 倍を見積もるのに対し, 提案者は 3 倍として見積もっている.

表 2.5　提案者による TWIRL の概算コスト (1024-bit)

	Rational TWIRL	Algebraic TWIRL
プロセスルール	0.13 μm	
周波数	1 GHz	
集積度	論理回路: 2.8×10^{-6}mm^2/transistor, DRAM: 0.2×10^{-6}mm^2/bit	
Largish Station の比率 (DRAM の比率)	76 % (37%)	94 % (66%)
Smallish Station の比率	21 %	6 %
Tiny Station の比率	3 %	0 %
チップ面積	15,960 mm^2/1 個 (30 cm ウェハー 1/4 枚)	66,000 mm^2/1 個 (30 cm ウェハー 1 枚)
	合計 30 cm ウェハー 3 枚/クラスタ 1 組	
計算時間 × クラスタ数	1 年 × 194 組	
コスト	製造コスト: 1000 万ドル (約 10 億円)/一式 NRE: 2000 万ドル (約 20 億円)	

シリコンの面積は 800 mm^2 となり, うち Largish Station が 56%, 残りのほとんどを Smallish Station が占める. 1 本のラインをふるうのに必要な時間は, スループットで R/s clock cycles (周波数 1 GHz で約 0.018 秒) であり, 全領域 (H 本のライン) をふるうには, 約 4.5 時間が必要となる. 以上をまとめると表 2.8 のようになる.

2.4.3　768-bit 合成数

768-bit 合成数に対しても, 2.3 節の基本構造を用いる. ただし B_{R}, B_{A}, R, H, s_{R}, s_{A} 以外のパラメータは 1024-bit の場合と同じ値を用いた. このとき Rational TWIRL は 1330 mm^2, Algebraic TWIRL は 4430 mm^2 を必要とする. 1 組の TWIRL クラスタは 4 個の Rational TWIRL と 1 個の Algebraic TWIRL から構成され, 合計で 9750 mm^2 を必要とする. 1 本のラインをふるうのに, スループットで R/s_{A} clock cycles (周波数 1 GHz で約 8.3 秒) を必要とするので, 全領域をふるうには約 855 日 (約 2.3 年) が必要となる. 1024-bit の場合と同様な高速化 (2.5.5 節) が可能であることと, 1 枚のウェハーから 6 組のクラスタがとれることを加味すると, 768-bit 合成数のふるい処理に必要なコストは, ウェハー 1 枚を用いた場合, 約 95 日 (約 3 ヶ月) となる. 以上をまとめると表 2.9 のようになる.

表 2.6　提案者による Rationl TWIRL の詳細コスト (1024-bit)

Largish Station		12,130 mm^2　76 %
	DRAM	5,905 mm^2　37 % 167268917 個 × 2 倍 × 88 bit/triplet × (0.2 ×10^{-6}mm^2/bit)
	CPU	2,292 mm^2　14 % 96400 trans/個 × 8490 個 × (2.8 ×10^{-6}mm^2/trans)
	Cell	3,191 mm^2　20 % 530 trans/個 × 2100 本 × 1024 本 × (2.8 ×10^{-6}mm^2/trans)
	Buffer など	742 mm^2　5 %
Smallish Station		3,352 mm^2　21 %
	Emitter	1,053 mm^2　7 % 2037 trans/個 × 184562 個 × (2.8 ×10^{-6}mm^2/bit)
	Cell	1,752 mm^2　11 % 1220 trans/個 × 501 本 × 1024 本 × (2.8 ×10^{-6}mm^2/bit)
	Funnel など	584 mm^2　3 %
Tiny Station		478 mm^2　3 %
合計		15,960 mm^2　100 %

表 2.7　提案者による Algebraic TWIRL の詳細コスト (1024-bit)

Largish Station		61,946 mm^2　94 %
	DRAM	43,494 mm^2　66 % 1133598328 個 × 2 倍 × 94 bit/triplet × (0.2 ×10^{-6}mm^2/bit)
	CPU	16,033 mm^2　24 % 96400 trans/個 × 59400 個 × (2.8 ×10^{-6}mm^2/trans)
	Cell	708 mm^2　1 % 530 trans/個 × 14900 本 × 32 本 × (2.8 ×10^{-6}mm^2/trans)
	Buffer など	1,711 mm^2　3 %
Smallish Station		3,954 mm^2　6 %
	Emitter	2,719 mm^2　4 % 2037 trans/個 × 476779 個 × (2.8 ×10^{-6}mm^2/bit)
	Cell	137 mm^2　0 % 1220 trans/個 × 1250 本 × 32 本 × (2.8 ×10^{-6}mm^2/bit)
	Funnel など	1,098 mm^2　2 %
Tiny Station		— mm^2　0 %
合計		65,900 mm^2　100 %

表 2.8　提案者による TWIRL の概算コスト (512-bit)

	Rational/Algebraic TWIRL (兼用)
プロセスルール	0.13 μm
周波数	1 GHz
集積度	論理回路: 2.38×10^{-6}mm^2/transistor, DRAM: 0.7×10^{-6}mm^2/bit
Largish Station の比率	56 %
Smallish Station の比率	44 %
Tiny Station の比率	0 %
チップ面積	800 mm^2/1 個
計算時間 × チップ数	4.5 時間 × 1 個 3.4 分 × 79 個 (ウェハ 1 枚)
コスト	製造コスト: 180 ドル (約 1 万 8000 円)/1 個

表 2.9　提案者による TWIRL の概算コスト (768-bit)

	Rational TWIRL	Algebraic TWIRL
プロセスルール	0.13 μm	
周波数	1 GHz	
集積度	論理回路: 2.8×10^{-6}mm^2/transistor, DRAM: 0.2×10^{-6}mm^2/bit	
チップ面積	1,330 mm^2/1 個	4,430 mm^2/1 個
	合計 9,750 mm^2/クラスタ 1 組 ウェハー 1/6 枚/クラスタ 1 組	
計算時間 × クラスタ数	2.3 年 × 1 組 3 ヶ月 × 6 組 (ウェハ 1 枚)	
コスト	製造コスト: 2200 ドル (約 22 万円)/組	

2.4.4　ハードウェアに関するパラメータ詳細

コスト算出に用いたハードウェアに関するパラメータは以下の通りである [LSTT02]: シリコンウェハーは 0.13 μm テクノロジーで考え, 300 mm の標準的なウェハー 1 枚の製造コストは 5,000 ドル (約 50 万円) とする. 1024-bit 合成数と 768-bit 合成数に対しては DRAM タイプのウェハーを使用し, トランジスタ密度は 2.8×10^{-6} mm^2/transistor (論理領域での平均値), DRAM 密度は 0.2×10^{-6} mm^2/bit (DRAM バンク領域での平均値) とする. 512-bit 合成数に対してはロジックタイプのウェハーを使用し, トランジスタ密度は 2.38×10^{-6} mm^2/transistor, DRAM 密度は 0.7×10^{-6} mm^2/bit とする. いずれの場合でもパイプラインを使用することか

ら, 周波数は 1 GHz として考える.

以下 2.3 節で使用したパラメータを用いた場合の 1024-bit 合成数について, パラメータの詳細を説明する. Largish Station で使用する特殊なプロセッサが必要とする transistor 数は $\langle\!\langle 96,400 \rangle\!\rangle_{\mathrm{R}}$ であると仮定する (ただしバッファーと $\langle\!\langle$14Kbit$\rangle\!\rangle$ のキャッシュメモリを含む. キャッシュの大きさは p_i とは独立である). Smallish Station で使用する Emitter が必要とする transistor 数は $\langle\!\langle 2,037 \rangle\!\rangle_{\mathrm{R}}$ (Funnel も含む), Tiny Station で使用する Emitter が必要とする transistor 数は $\langle\!\langle 522 \rangle\!\rangle_{\mathrm{R}}$ であると仮定する. Delivery Line 上の cell は, interleaving あり (Largish Station 用) で $\langle\!\langle 530 \rangle\!\rangle_{\mathrm{R}}$ transistor, interleaving なし (Smallish/Tiny Station 用) で $\langle\!\langle 1,220 \rangle\!\rangle_{\mathrm{R}}$ transistor が必要であるとする. 2.3.2 節で説明したメモリ構造では, 通常の DRAM に比べてスラック領域とキャッシュ領域が多めに必要となるので, 通常よりも $\langle\!\langle 2.5 \rangle\!\rangle_{\mathrm{R}}$ 倍の領域を占めるとする. Bus Line では, ワイヤの重層実装が可能なことから, ラインは領域を必要とせず, レジスタのみ領域を使用するとする. ワイヤのクロスには $\langle\!\langle 0.5 \rangle\!\rangle$ bit/μm を使用するとする.

TWIRL では異なる大きさの内部結合ユニットを多数使用することから, 効率的なレイアウトを実現するのは容易でない. しかしここで用いられているユニットの種類は, LSI の世界で通常行われているデザインでの種類数に比べて少ないので, 考慮する余地は大いに残っている.

2.4.5 数体ふるい法に関するパラメータ詳細

1024-bit 合成数に対する TWIRL で使用したパラメータは, 特定の 1024-bit 合成数 (RSA-1024) に対するものである. 従って同じコストで他の 1024-bit 合成数が分解できる保証はないが, TWIRL の性能を把握する上で大きな参考となる.

以下に合成数 n と多項式 $F(x)$, $G(x)$ を示す. ここで n は懸賞問題 (RSA-1024) の対象となっている合成数である. また m は $F(m) \equiv G(m) \equiv 0 \pmod{n}$ を満たす共通解である. これら多項式は Franke と Kleinjung によって生成された [LTS+03].

$$
\begin{aligned}
n = {} & 135066410865995223349603216278805969938881475605667027524 48//\\
& 514385152651060485953383394028715057190944179820728216447 15//\\
& 513736804197039641917430464965892742562393410208643832021 10//\\
& 372958725762358509643110564073501508187510676594629205563 68//\\
& 552947521350085287941637732853390610975054433499981115005 69//\\
& 77236890927563\\
F(x) = {} & 171930489423634514340101141808 0x^5\\
& -699197348886660586107407418604363447 1x^4\\
& +270860304835695328940509742578513466495213 14x^3\\
& +469375840526685745028867918355365522774102423590 42x^2\\
& -101070294842572111371781458850696845877706899545394501384x\\
& -226669159394909405786175246770453711891289098997165603984341 36\\
G(x) = {} & 938772308370263069845713674770 27x\\
& -379348954964250275136910457556396371742114833244516283 65\\
m = {} & 262619868791250802778182368904158187239810594129624673885 00//\\
& 761036823061967405529250615451338729866356088714618385497 51//\\
& 981605022782432450670748205937110547238505700273957561400 11//\\
& 420203134807117903732061718812827366825166704434650128222 81//\\
& 608387169409282469138311259520392769843104985793744494821 43//\\
& 7272961970486
\end{aligned}
$$

2.5　詳細デザイン

本節では TWIRL の詳細なデザイン法について述べる. 動作原理については 2.3 節を参照されたい. 繰り返しになるが, 本節の内容は提案論文に基づくものであり, ことわりのない限り報告者の主張は含まれていない.

2.5.1　Delivery Line

Delivery Line は Buffer/Funnel/Emitter から受け取った Delivery Pair を指定の Bus Line まで運ぶ目的で, 全ての Station において利用される. 基本的な構造については 2.3.2 節で述べたが, 以下では効率的な実装法について説明する.

■Interleaving　Largish Station における Delivery Line 上の Cell は, 大半の時間はシフトレジスタとして機能しており, 加算が行われることは滅多にない. 従って Deliverry Line と Bus Line の全ての交点に加算器を配置するのは効率的でない. つまり interleaving (中抜き) を適用することで, 加算器を削減することが可能となる. Cell は $r \,《= 4》_{\mathrm{R}}$ 本の Bus Line をまたぐ構造とし, q 番目の Bus Line とだけ加算が計算できるようにしておく. ここで q は Delivery Line 上で一定であり, その値は Delivery Line に応じて周期的に増えていくとする ($1 \le q \le r$). このとき Delivery Line r 本を同時に処理するようにすれば, 各 Bus Line に加算器が 1 個ずつ配置されることになる. このように interleaving を用いることによって, Largish Station 全体での Cell の個数は $1/r$ に削減することができる. なお Smallish/Tiny Station では interleaving は使用しない.

Delivery Line を流れる Emission Pair は 1 clock cycle あたり r 本の Bus Line を横切ることになるので, Buffer から Bus Line への距離が遠ざかるに応じて, "age" $\lfloor a_i/s \rfloor$ は小さくなる. 正確に書くと, 時刻 t において, ふるい位置 a は $t - \lfloor a_i/s \rfloor - \lfloor (a \bmod s)/r \rfloor$ 本目の Delivery Line の $\lfloor (a \bmod s)/r \rfloor$ 番目の Cell の中の r 本の Bus Line のいずれかに (存在すれば) 位置している.

Largish Station では, Buffer は Delivery Pair を Delivery Line に適切な時刻に送信することが前提となっている. 従って Buffer を 2 個用意し, それぞれの Buffer が半数の Bus Line を担当するようにすれば, 約 2 倍の高速化が可能となる.

Bus Line にパイプライン状のレジスタを用いると, Buffer に接続された Delivery Line への配送は 1 本ごとに 1 cycle clock ずつ遅れるが, Buffer の最後にパイプラインを追加することによって対応可能である.

2.5.2　Emitter の重複

本節では Smallish/Tiny Station において使用する Emitter の実装法について述べる (それぞれの構造については 2.3.3 節, 2.3.4 節を参照). 1 個の Emitter は 1 個の Progression P_i を担当し, Bus Line を segment 分割して得られる集合 $G_1, G_2, \ldots, G_{n_i}$ の中のいずれかの line $\ell^{(i)} \in G_j$ に対して, Delivery Pair $(\lfloor \log p_i \rceil, \ell^{(i)})$ を発行する (n_i は segment 分割の個数). このとき $\ell^{(i)}$ の計算と $\ell^{(i)} \in G_j$ の判定が必要となるが, これらを効率的に処理するには工夫が必要となる. 始めに単純な実装法を用いた場合の問題点を述べ, 次にその改良として効率的な実装法を説明する.

■単純な実装　Emitter の最も単純な実装法は, $\lceil \log p_i \rceil$-bit のレジスタを用意して, 各 clock cycle 毎に $s \bmod p_i$ ずつ増加させていき, 桁上がりが生じる (つまり Delivery Pair を発行する) ときに $\ell^{(i)}$ を計算して $\ell^{(i)} \in G_j$ かどうかを調べる方法である. しかしレジスタは 1 clock cycle で更新されなければならないので, 高価なキャリー先読み機能付き加算器 (carry-lookahead

adder) が必要になってしまう. また s と $|G_j|$ の設定方法によっては $\ell^{(i)}$ の計算と $\ell^{(i)} \in G_j$ の判定は手間がかかってしまう.

別のアプローチとして考えられるのは, 2 つのレジスタを用意し, 1 つは (TWINKLE のように) 次の Emission までの時刻をカウントダウンで管理し, もう 1 つは $\ell^{(i)}$ を管理する方法である. これには 2 種類の具体的な実装法が考えられる. 次の Emission までの時刻を $\ell^{(i)}$ とは無関係に管理してしまうと, Emission の周期が短いため, レイテンシーが小さくなければならない上に, $p_i < s$ の場合を扱うことができない. そこで G_j への次の Emission までの時刻を管理することにすると, 別の問題が生じてしまう. つまり G_j への Emission の周期は不規則であり, 計算のコストが大きくなってしまうのである. この最後の問題を解消するのが次節の改良方法である.

■Line 番号のビット反転を用いた改良　ふるい位置の任意の Bus Line への割り当ては 1 clock cycle で可能である. しかし wire をグループに 1 clock cycle で分割するには物理的に接近していることが必要である. そこで以下のようなトリックを用いる. 適当な整数 α, β_i に対し $s = 2^\alpha$, $|G_j| = 2^{\beta_i}$ とする ($\alpha = \langle\!\langle 12 \rangle\!\rangle_{\mathrm{R}}, \langle\!\langle 15 \rangle\!\rangle_{\mathrm{A}}$). そして s で割った余りと Bus Line のビット反転 (bit-reversal), すなわちふるい位置 $w \bmod s$ と $\mathrm{rev}(w)$ 番目の Bus Line を対応づける. ここで $w = \sum_{i=0}^{\alpha-1} c_i 2^i$ $c_i \in \{0, 1\}$ のビット反転 $\mathrm{rev}(w)$ は

$$\mathrm{rev}(w) = \sum_{i=0}^{\alpha-1} c_{\alpha-1-i} 2^i$$

で与えられる. Progression P_i を担当する Emitter は n_i ($\approx 2^{\alpha-\beta_i}$) 個ある. そのうち j 番目の Emitter は, j 番目のグループ G_j へ送信するようにする ($j \in \{1, \ldots, 2^{\alpha-\beta_i}\}$). ただし G_j は 2^{β_i} 本の Bus Line を持つ. このとき次のような性質がある [ST03]:

> 素数 $p_i > 2$ に対応する Emitter は, s での剰余のために起こる 1 clocl cycle の遅れを除き, 一定の間隔 $T_i = \lfloor 2^{-\beta_i} p_i \rfloor$ で Delivery Pair を発行する.
>
> **証明**　ふるい位置 $a \in P_i$ に対応する j 番目のグループ G_j への *emission* を考える. このとき定義より $\lfloor \mathrm{rev}(a \bmod s)/2^{\beta_i} \rfloor = j$ すなわち $a \equiv \mathrm{rev}(j) \pmod{2^{\alpha-\beta_i}}$ を得る. 他方 $a \in P_i$ より $a \equiv r_i \pmod{p_i}$ となる r_i が存在し, p_i は $2^{\alpha-\beta_i}$ と互いに素であることから, 中国人剰余定理によって, このようなふるい位置の集合は, 適当な $c_{i,j}$ を用いて, $P_{i,j} = \{\, a \mid a \equiv c_{i,j} \pmod{2^{\alpha-\beta_i} p_i} \}$ と一致することがわかる. 連続するふるい位置

$a_1, a_2 \in P_{i,j}$ は $a_2 - a_1 = 2^{\alpha-\beta_i} p_i$ を満たすから, *emission* の間隔 Δ は

$$\Delta = \lfloor a_2/s \rfloor - \lfloor a_1/s \rfloor = \begin{cases} T_i & (a_2 \bmod s) > (a_1 \bmod s) \\ T_i + 1 & \textit{otherwise} \end{cases}$$

となる. □

なお β_i の選び方により $T_i = \lfloor \sqrt{p_i}/2 \rfloor_2$ となる.

■Smallish Station における Emitter の構造　Smallish Station では, 各 Emitter は以下のような 2 つのカウンタを持つ.

Counter A:　その Emitter が次に emission を発行するまでの時刻を保持する. 値は法 $T_i = \lfloor 2^{\alpha-\beta_i} p_i \rfloor$ で保持され, 典型的なサイズは $\langle\!\langle 7 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 5 \rangle\!\rangle_{\mathrm{A}}$ bit である. (ほぼ) 各 clock cycle 毎に 1 ずつ減少していく.

Counter B:　次の emission に対応するふるい位置番号 a に対し, $a \bmod s$ の上位 β_i-bit を保持する. 値は法 2^{β_i} で保持され, 典型的なサイズは $\langle\!\langle 10 \rangle\!\rangle_{\mathrm{R}}$, $\langle\!\langle 15 \rangle\!\rangle_{\mathrm{A}}$ bit である. Counter A に繰り下がりが生じたときには $2^{\alpha-\beta_i} p_i$ ずつ増加する. Counter B に繰り上がりが生じたときには, Counter A は 1 clock cycle の間サスペンドする (これで s による剰余の影響が消える).

Delivery Pair $(\lfloor \log p_i \rceil, \ell^{(i)})$ は Counter A に繰り下がりが起きたときに発行される. ここで $\lfloor \log p_i \rceil$ は各 Emitter ごとに固定の値である. このときターゲットとなる Bus Line 番号 $\ell^{(i)}$ は, Counter B の上位 β_i-bit によって定まる. この Emitter では残り $(\alpha - \beta_i)$-bit は固定値となるので, 他の値を求める必要はない.

Emitter は担当する Bus Line の物理的に近くに配置される必要がある. また Counter の値は初期化で適切に設定される必要がある.

■Tiny Station における Emitter の構造　Tiny Station でも Smallish Station と同様のデザインを用い, Bus Line は $a \bmod s$ のビット反転と対応づける. ここで β_i は, $|G_j| = 2^{\beta_i}$ が p_i 以下の最大 2 べきとなるように選ぶ. これによって $T_i = 1$ となるので, emission は 1 or 2 clock cycle ごとに発行される. よって Tiny Station における Emitter の構造は Smallish Station のときと同じであるが, Counter A のサイズは 0 であり, 実際にはサイズ β_i-bit の Counter B だけとなる.

2.5.3 Funnel

本節では Smallish Station において使用する Funnel の実装法について述べる. Funnel の役割は, Emitter から受け取った Delivery Pair を Delivery Line に送ることである (2.3.3 節参照). つまり Funnel は Largish Station における Buffer からソート機能を削除したユニットと考えることができる. Emitter から Funnel への入力情報には (Buffer と同様に) null 情報も含まれるため, これらを効率的に削除する必要がある. 以下では Funnel を 2 段に配置することで効率を向上させる.

n 列 m 行の行列状の Cell を持つ Funnel を n-to-m Funnel と呼ぶ ($n \gg m$). 各 Cell は 1 個の Progression Triplet を保持するレジスタを持つ. Funnel への入力は n 個の Progression Triplet からなる配列で, 各 clock cycle 毎に各列の先頭の行に 1 個ずつ格納され, t 番目に入力された配列の i 番目の要素は, 時刻 $t+i$ に i 番目の列に挿入される. そして全てのデータは毎 clock cycle ごとに 1 列右にシフトされる. また他の値を書き換えることのない限り 1 行下にシフトされる. t 番目に出力される配列は時刻 $t+n$ のときに右端から読み出される.

この Funnel でオーバーフロー (空のない列に挿入される) の確率を見積もる必要がある. $n > m$ として, 各入力が空でないという事象は確率 ν で互いに独立に起きるとする (実際 2.3.3 節の結果より $\nu \approx 2/\sqrt{p_i}$ となる). このときオーバーフローによって (空でない) データが失われる確率は

$$\sum_{k=m+1}^{n} \binom{n}{k} \nu^k (1-\nu)^{n-k} \frac{k-m}{k}$$

で与えられる. ここで $m = \langle\!\langle 5 \rangle\!\rangle_{\mathrm{R}}$, $n = \langle\!\langle 1/\nu \rangle\!\rangle_{\mathrm{R}}$ とすると, 上記の確率は高々 0.00011 となるので, これらパラメータを採用する (表 2.10 参照).

上記の値を使用した場合の Funnel の圧縮率は $n/m \langle\!\langle \approx 1/5\nu \rangle\!\rangle$ となるので, Funnel の出力が空にならない確率は $\nu' = \nu \times n/m \langle\!\langle \approx 1/5 \rangle\!\rangle_{\mathrm{R}}$ となるが, 十分な大きさではない. そこでこの出力を 2 つ目の Funnel に再度入力し, さらなる圧縮を行うことにする. $\langle\!\langle m' = 14,\ n' = 35 \rangle\!\rangle_{\mathrm{R}}$ と設定した場合に オーバーフローが起きる確率は高々 0.00016 なので, 十分小さいと見なすことができる. これら 2 つの Funnel の出力が空にならない確率は $\nu' \times n'/m' = 1/2$ となる. Delivery Line を最適値の 2 倍程度用意すれば, これら 2 段 Funnel との最適な接続が実現できる. このとき Largish Station で用いたような interleaving は使用しない.

表 2.10　Funnel の行数とオーバーフローの起きる確率

m	p_i					
	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
2	0.02445	0.02751	0.02895	0.02965	0.03000	0.03017
3	0.00301	0.00416	0.00476	0.00506	0.00521	0.00529
4	0.00026	0.00050	0.00065	0.00074	0.00078	0.00080
5	0.00001	0.00005	0.00008	0.00009	0.00010	0.00011
6	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

2.5.4　初期化

デバイスの初期化では, 全ての Station において, Progression 状態および初期値の読込やバスのバイパスへの命令読み込みが必要となる. 特にふるい計算の最中に Progression 状態は変化するため, b が異なるたびに再初期化が必要となるが, これら読込は時間がかかるため, あまり得策ではない (実際 TWINKLE ではボトルネックとなっていた [Sha99]).

ふるいの対象となる各 line は互いに関連がある [GS03]. したがって初期値 r_i をふるい毎に定数 $\tilde{r}_i$ ずつ増加させていけば, 再読込は不要となる. これら初期値は, Progression ごとに $\log_2 p_i$ bit 程度で DRAM に記録可能であり, 加算は専用のプロセッサを通じて行う. 加算が行われる間隔は R/s と長いので, 急いで計算する必要はない.

2.5.5　ふるい対象の削減

数体ふるい法のふるいステップで収集する組 (a', b) は, $\gcd(a', b) = 1$ を満たす必要があった (ただし $a' = a - R/2$). 従ってふるいの段階でこの条件を満たさない (a', b), つまり小さな整数 $c > 1$ に対して $c | \gcd(a', b)$ を満たす (a', b) を排除することによって, 効率を上げることが可能となる. 数体ふるい法のソフトウェア実装では, $c = 2$ の場合を考慮するのが普通で [ST03], この場合 25% の効率化が達成できる. TWIRL においても, 同様のテクニックを用いる. まず b が奇数となる line に対するふるい処理を普通に行う. 次に b が偶数となる line をふるうのだが, このときは奇数の a' だけを用いる. このとき $p_i > 2$ を満たす素数は p_i 個ずつのふるい位置に contribute すれば良いので, a' の初期値 a'_0 だけを変更すれば良い.

同様にして $c = 3$ の場合も扱うことにする. $c = 2$ の場合も同時に考慮すると, ふるいは 4 種類のタイプに分かれる: $b \equiv 1, 5 \pmod 6$ の場合は全領域を, $b \equiv 2, 4 \pmod 6$ の場合は半分の領域 ($a' \equiv 1 \pmod 2$) を, $b \equiv 3 \pmod 6$ の場合は 2/3 の領域 ($a' \equiv 1, 2 \pmod 3$) を, $b \equiv 0 \pmod 6$ の場合は 1/3 の領域 ($a' \equiv 1, 5 \pmod 6$) をふるうことになるので, 全体では 2/3 の領域をふるえば良いことになり, 33% の効率化が達成される. しかし $c > 3$ の場合を考慮した場合, 分類が複雑になるのに比べて効率化の度合いが小さいため, TWIRL では扱わないことにする.

2.5.6 ふるいのカスケード

数体ふるい法における関係式探索ステップでは, 有理数的ふるいと代数的ふるいを処理する必要があり, 両者のふるいに合格した (a, b) を集めるのが目的であった (2.2.2 節参照). しかし代数的ふるいの smoothness bound $B_{\mathrm{A}} \langle\!\langle = 2.6 \times 10^{10} \rangle\!\rangle_{\mathrm{A}}$ は 有理数的ふるいの smoothness bound $B_{\mathrm{R}} \langle\!\langle = 3.5 \times 10^{9} \rangle\!\rangle_{\mathrm{R}}$ に比べて大きいため, 各 Station において s が最適に設定された場合, 代数的ふるいのコストが全体の大部分を占めてしまう. さらに 2.4.1 節で示すパラメータを使用して TWIRL によって 1024-bit 合成数のふるい処理を行った場合, その回路面積が 1 枚のシリコンウェハーよりも大きくなってしまう可能性が生じる. 従って s_{A} の値をあまり大きくすることができない. そこで以下のようにカスケードを用いることによって, この問題を解決する.

s_{R}, s_{A} を Rational/Algebraic TWIRL の Bus Line の本数とする. 前述したように s_{A} の値はあまり大きくすることができない. しかし, 確かに Algebraic TWIRL では s_{A} 本ずつの Bus Line を処理するように設計されているが, Rational TWIRL の処理においてほとんどのふるい位置は棄却されてしまうため, Algebraic TWIRL の入力となるふるい位置は極めて少ない (提案者のデータによると, およそ $\langle\!\langle 1.7 \times 10^{-4} \rangle\!\rangle$ の割合でしかない). この性質を利用して, Algebraic TWIRL に対して以下のような改良を施す. Algebraic TWIRL では s_{A} 本という多数の Bus Line を用いる代わりに, $u \langle\!\langle = 32 \rangle\!\rangle_{\mathrm{A}}$ 本というごく少数の Bus Line を用意する[*16]. ここで各 Bus Line 上の Cell は (v, L) という情報を保持し, $L = a \bmod s_{\mathrm{A}}$ はふるい位置情報, v はこのふるい位置に対応する対数値の和 $\sum \lfloor \log p_i \rceil$ を表す. これらふるい位置は従来通りパイプライン状に処理され, その速度は 1 clock cycle あたり s_{A} 個とする. Delivery Pair の生成法は従来通りとする.

変更後の Delivery Line はとても短いので, 送信された Delivery Pair $(\lfloor \log p_i \rceil, \ell^{(i)}$ に対し, u 個の Cell が保持する L の値と $\ell^{(i)}$ の値を 1 cycle clock で一斉に比較することが可能である. 等しい場合には対応する v に $\lfloor \log p_i \rceil$ を加え, そうでない場合 (ほとんどがこのケースとなる) Delivery Pair を棄却する.

[*16] $s_A \times (1.7 \times 10^{-4}) = 5.44$ であるから, u の値としては 6 より大きい値が必要である.

Rational TWIRL に合格したふるい位置 a に対し, Bus Line へは $(0, a \bmod s_{\mathrm{A}})$ を入力する. このため Rational TWIRL の出力と Algebraic TWIRL の入力をつなぎ, 同時に処理することが必要となる. 両者のふるいの Bus Line の本数 s は異なるため, 1 個の Algebraic TWIRL に対し $s_{\mathrm{A}}/s_{\mathrm{R}}$ ⟪= 8⟫ 個の Rational TWIRL を接続する. 接続されたデバイスをまとめて TWIRL クラスタと呼ぶ. 1 組のクラスタは 1 本の sieve line を処理するが, Rational TWIRL は複数で処理するため, (2.5.2 節の bit 反転と同様の) ふるい位置の interleaving が必要となる. 各 Rational TWIRL から Algebraic TWIRL へ流れるデータは, 1 cycle clock あたり高々 $1 + \log_2 s_{\mathrm{R}}$ ⟪= 13⟫ bit となる (ここで先頭 1-bit はデータの有無を表す).

カスケードを用いることで, Algebraic TWIRL におけると Bus Line の専有面積が劇的に削減できる. また Smalish Station において Emitter を重複させる必要もなくなる (ただし $p_i < s_{\mathrm{A}}$ の場合は除く). このようにして Algebraic TWIRL では実質的に s_{A} 本ずつの並列処理が可能となり, そのコストは Rational TWIRL よりも小さくなる (2.4.1 節参照). またほとんどコストをかけずに B_{A} を大きくする, つまり H や R を小さくすることが可能となる.

■Rational TWIRL における閾値判定部　Rational TWIRL 内部の Bus Line の本数は s_R ⟪= $4,096\rangle\!\rangle_{\mathrm{R}}$ 本であるのに対し, カスケードを用いた場合, Rational TWIRL から Algebraic TWIRL に接続される Bus Line の本数は $u/(s_A/s_R)$ ⟪= 4⟫ 本となる. 閾値判定部は, 本来の機能である閾値判定を行うとともに, Bus Line を流れる不要なふるい位置を棄却して必要なふるい位置のみを圧縮する動作も必要がある.

2.5.7　関係の導出

有理数的ふるいと代数的ふるいの双方に残った (a, b) を候補と呼んだ. これら候補は対数の和の概算値によって判定されているため, 実際の f_{R}, f_{A} の値が smooth になっているかを調べる必要がある. 従って Algebraic TWIRL の後に, この処理を行う関係式導出部が必要となる (2.2.2 節参照) [*17].

■候補の特定　Rational/Algebraic TWIRL の Bus Line の最後に比較器を置き, ふるい位置 a に対応する対数値の和 v_a が閾値 T よりも大きいかどうかを判定する (閾値判定部). 基本的な TWIRL のデザインでは 2 つのふるいは同時に処理され, これらふるいの間で比較器をパスしたペアの情報が交換されることを通じて候補が定まる. 2.5.6 節のカスケードを用いた場合には, Algebraic TWIRL は, Rational TWIRL をパスしたペアしか処理しないため, Algebraic

[*17] しかし TWIRL として必須の機能ではない.

TWIRL の閾値判定をパスしたペアは 2 つの判定をパスしたことになる. 提案者の試算では, 候補となる確率は非常に小さい (⟪2.0×10^{-11}⟫).

■対応する Progression の決定　候補 (a, b) に対し, 次に $f_R(a, b)$, $f_A(a, b)$ の素因数分解情報が必要となる. 素因数が小さな場合は簡単なので, Largish Prime に関する分解情報が得られれば良い. これは Largish Station において Progression Triplet の情報を保存することで求められる.

実装に当たっては, Largish Station の全ての Processor を通過するパイプライン状の Channel を 2 つ追加する必要がある. Line Channel は $\log_s$ s-bit 幅であり, データは Bus Line とは逆方向に流れていく. Division Channel は $\log_2$ B-bit 幅であり, データは Bus Line と同じ方向に流れる. 各 Channele は Processor と Processor の間にレジスタを持ち, Rational/Algebraic TWIRL の出力に接続される. 各 Processor に対し, Diary と呼ぶデータを対応づける. ここで Diary は $\log_2$ B-bit のデータを巡回的にとる. 各 clock cycle に, Processor は自分の Diary にデータを保存する: この clock cycle に Emission Triplet $(\lfloor \log p_i \rceil, \ell^{(i)}, \tau^{(i)})$ を Buffer に送信したならば Triplet $(\log p_i, \ell^{(i)}, \tau^{(i)})$ を, そうでなければ null を保存する. Bus line ℓ において候補が見つかったとき, ℓ は Line Channel を通じて送信される. 各 Processor は ℓ の値を参照し, ちょうど z clock cycle 前に Bus Line ℓ に Emission を発行したかを自分の Diary から調べる. ここで z は Processor が Buffer へ出力してから, Triplet が該当する Bus Line に到着し, 閾値判定部まで至ってから Line Channel を通って戻るまでの距離である. 従って Diary の探索は z clock cycle 前に保存されたものから ⟪64⟫ 個の空でない要素を調べていけばよい. 該当する Diary が見つかったらば, Processor は Divisor Channel を通じて p_i を送る. 異なる候補に対応するデータが混ざる可能性は無視することができると提案者は主張している.

ふるいのカスケードを行う場合, Algebraic TWIRL は Delivery Line を流れるふるい位置に対応するもののみを保存する. Rational TWIRL の Diary は比較的大きくなる (⟪$13,530$⟫$_{\mathrm{R}}$). これはかなり以前のデータを保存する必要があるからである. 最悪の場合, z には全ての Delivery Line を通過する時刻が含まれる. しかし提案者は, DRAM を用いればこれら Diary を効率的に実装することが可能であるとしている.

■ミニ素因数分解　様々な概算値による計算やエラーの影響, さらには Large Prime の利用により, 前述のような素因数分解情報が与えられたとしても, 候補に対してさらなる計算が必要となる. 対応する f_{R}, f_{A} の値の計算, 小さな素数と Diary に登場する素数による除算, 余因子の大きさの判定と素数判定は, 専用の Processor を用いて効率的に実装することが可能である.

しかし余因子が比較的大きく, 合成数になっている場合には, この余因子を素因数分解する必要がある (ミニ素因数分解). ここでミニ素因数分解とは高々 ⟪1.0×10^{24}⟫ の合成数を分解する

ことで, ふるい対象の $\langle\!\langle 2.0 \times 10^{-11} \rangle\!\rangle$ の割合で必要となる[*18]. 専用の Processor を使用すればこの素因数分解は 0.05 秒以下で可能であり, ふるい処理に与える影響は小さいと提案者は主張している.

2.6 Largish Station における送信器の理論的サイズ

TWIRL の仕様によると, Largish Station においては C 個の送信器が必要であり, j 番目の送信器は d_j 個の Progression を保持する必要があったが (2.3.2 節参照), 詳細なパラメータは与えられていなかった. そこで本節では, 計算実験によって得られた C と d_j の具体的な数値結果 (抜粋) を示す. ここで j 番目の送信器は連続する d_j 個の素数 $\{p_{\min}, \ldots, p_{\max}\}$ を担当するものとする.

2.6.1 Rational TWIRL

j	$p_{\min}$	$p_{\max}$	d_j
1	520019	520411	32
100	564229	564653	35
200	612193	612751	38
300	665293	665843	41
400	723029	723473	45
500	785921	786661	49
600	854923	855667	53
700	929843	930637	57
800	1012619	1013563	62
900	1103171	1104179	68
1000	1201327	1202429	74
1100	1310123	1311229	81
1200	1429367	1430617	88
1300	1559209	1560547	96
1400	1702543	1704137	105
1500	1859911	1861567	114
1600	2032273	2033803	125
1700	2222621	2224661	136
1800	2431811	2434099	149
1900	2661371	2663867	163
2000	2914741	2917457	179
2100	3193763	3196709	196
2200	3501919	3504961	214
2300	3841027	3844433	235
2400	4215517	4219571	258
2500	4629617	4633999	283
2600	5086007	5090753	311
2700	5592227	5597563	342
2800	6151819	6157673	376
2900	6770921	6777521	414
3000	7455743	7463231	456
3100	8216297	8224421	502
3200	9059173	9067843	554
3300	9994871	10004947	611
3400	11032039	11042963	675
3500	12184489	12196759	745
3600	13467173	13480529	823
3700	14891713	14906989	910
3800	16479301	16495783	1007
3900	18244279	18262877	1115
4000	20213257	20234167	1235
4100	22409887	22433197	1370

[*18] $\lfloor \log p_i \rceil$ による誤差を考慮している.

4200	24857197	24883003	1519
4300	27591653	27620503	1686
4400	30643601	30674821	1873
4500	34059281	34095497	2082
4600	37882531	37922741	2315
4700	42154283	42199601	2576
4800	46938961	46989857	2869
4900	52306747	52362581	3196
5000	58324949	58388279	3564
5100	65078939	65151509	3977
5200	72658241	72738389	4440
5300	81180107	81270311	4961
5400	90767077	90868957	5547
5500	101549729	101665099	6206
5600	113688271	113817371	6947
5700	127373417	127518023	7784
5800	142803263	142965967	8726
5900	160212719	160397917	9790
6000	179859703	180068839	10991
6100	202082987	202319617	12349
6200	227201969	227468291	13884
6300	255621127	255922217	15621
6400	287814977	288156917	17588
6500	324289561	324678671	19817
6600	365658277	366098329	22345
6700	412601771	413099989	25214
6800	465914413	466482031	28472
6900	526528193	527175023	32177
7000	595444519	596174017	36388
7100	673899671	674739193	41183
7200	763288063	764246513	46646
7300	865168963	866259869	52873
7400	981421061	982663039	59977
7500	1114117409	1115536157	68087
7600	1265801609	1267420051	77358
7700	1439225197	1441074511	87957
7800	1637670719	1639798129	100086
7900	1865003033	1867439429	113980
8000	2125557289	2128349851	129905
8100	2424479921	2427673949	148174
8200	2767654801	2771325673	169149
8300	3161963159	3166182653	193249
8375	3496019321	3500716913	213667
			d_j の合計 = 167268917

2.6.2 Algebraic TWIRL

j	$p_{\min}$	$p_{\max}$	d_j
1	4200013	4200433	33
1000	4728211	4728799	37
2000	5326921	5327587	41
3000	6007091	6007643	46
4000	6778439	6779299	52
5000	7654259	7655201	59
6000	8650673	8651683	67
7000	9785707	9787067	75
8000	11078297	11079547	85
9000	12552079	12553487	96
10000	14236771	14238593	109
11000	16160189	16162249	124
12000	18357737	18360383	141
13000	20877271	20879707	160
14000	23764063	23767039	182
15000	27074329	27077587	207
16000	30872477	30876319	236
17000	35247323	35251763	269
18000	40271291	40276463	308
19000	46060379	46066219	352
20000	52737779	52744729	403
21000	60443023	60451861	462
22000	69344711	69353827	530
23000	79638347	79649371	608

24000	91556693	91569847	699	43000	1599934507	1600192019	12209
25000	105366263	105380689	804	44000	1881937133	1882245187	14361
26000	121395661	121413571	927	45000	2216386547	2216748029	16913
27000	140006491	140026591	1069	46000	2613498311	2613928991	19943
28000	161649307	161672933	1234	47000	3085711217	3086226977	23547
29000	186832981	186859573	1426	48000	3647856203	3648470279	27836
30000	216192953	216224249	1650	49000	4317975953	4318705987	32950
31000	250434131	250472279	1911	50000	5117764919	5118638549	39053
32000	290432393	290475371	2217	51000	6073584943	6074625449	46346
33000	337195613	337245973	2573	52000	7217395123	7218644689	55074
34000	391928701	391987793	2991	53000	8587910557	8589404393	65532
35000	456070961	456140161	3481	54000	10232332307	10234136329	78081
36000	531340517	531423433	4055	55000	12208030343	12210196409	93157
37000	619731727	619826491	4729	56000	14584862657	14587471207	111294
38000	723686279	723798839	5523	57000	17448199243	17451343451	133144
39000	846085543	846218603	6457	58000	20902366367	20906160493	159502
40000	990379277	990536219	7558	59000	25074995459	25079580571	191343
41000	1160668739	1160854171	8857	59198	25999481891	26004238873	198397
42000	1361874313	1362092183	10392	d_j の合計 = 1133598328			

第３章

詳細検討

本章では 1024-bit 合成数をターゲットとした TWIRL の回路設計を行うとともに、設計結果を元にして回路規模の見積もりを示す。まず 3.1 節で簡単に TWIRL の構造をまとめた後、Largish/Smallish/Tiny Station の具体的な回路設計を 3.3 節から 3.5 節で提示する。そしてこれらの設計結果を元にした TWIRL の回路規模見積もりを 3.6 節で算出する。

設計においては、可能な限り TWIRL の提案論文 [ST03]の記述を忠実に反映させ、記述が不十分である場合には適当な補完を行った。ただし提案論文の仕様を越えるような改良は一切行っていない。また実装に伴うさまざまな制約により、設計で用いるパラメータは、第 2 章で理論的に導出されたパラメータとは異なることもありえることに注意が必要である.

3.1. TWIRL を用いたふるい装置の概略

装置としての TWIRL は、TWIRL cluster と呼ばれる単位の装置を、複数用いて並列化することでふるい装置を実現する。Shamir-Tromer らは、TWIRL cluster の数が多いほどふるい処理を高速化できると主張している。

1024bit のふるいを 1 年で完了するためには。TWIRL cluster の数が多い場合、各 cluster が出力するデータを記録する装置(RAM,HDD)が必要となり、記録装置の種類によってふるい処理全体の性能に影響を与える[1]と考えられる。

Shamir-Tromer の論文では、TWIRL cluster の記録装置を含めたふるい装置全体の構成法を記していない。よって TWIRL を用いたふるい装置は本章の評価対象外とする。

3.1.1. TWIRL cluster

[1] 処理速度を重視するならば RAM、記憶容量を重視するなら HDD が適している

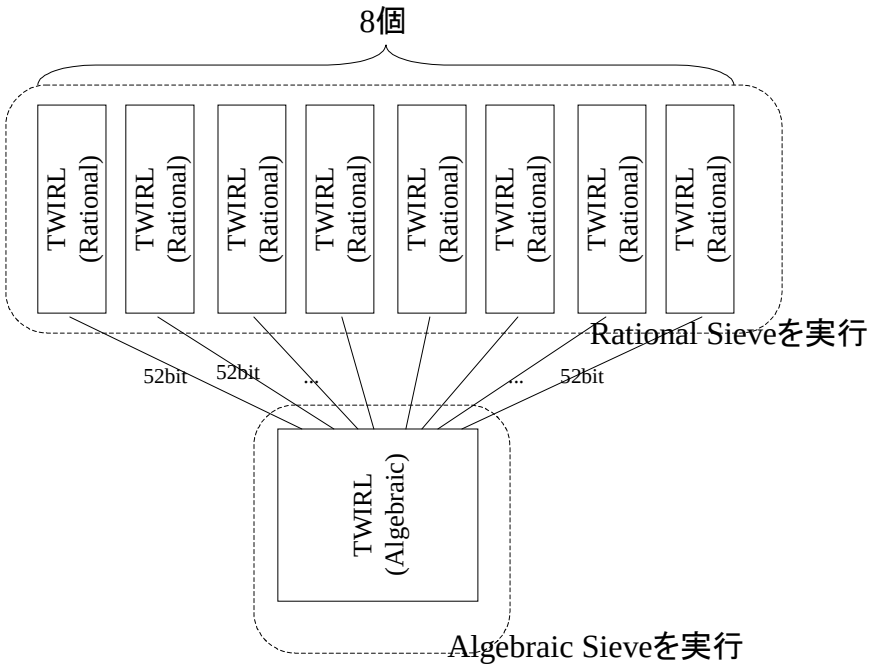


図 3 - 1 TWIRL cluster 概略図

図 3 - 1 に示す。1 個の TWIRL cluster は、Rational Sieve を行う 8 個の TWIRL と、Algebraic Sieve を行う 1 個の TWIRL から構成される[2]。

Rational から Algebraic の TWIRL への IO 数は、Algebraic の TWIRL 実装法によって(1)40,960bitΔ8(2)52bitΔ8 の 2 種類があるが、(1)は IO 数が多すぎるため実装不可能である。よって、現実的な TWIRL 実装を考える場合、IO 数を最適化した(2)の実装を検討する必要があるが、(2)における Algebraic の TWIRL 実装法に基本仕様は Raitonal TWIRL と同じであるが、詳細については、hamir-Tromer の論文の内容からは不明な点が多い。よって、本章では Rational TWIRL に限定して実装評価を行う。以下本章の"TWIRL"とは、Rational TWIRL を指すものとする。

[2] Shamir-Tromer は TWIRL を直径 300mm ウェハーから製造した 1 つの LSI で実装可能と主張しているが、欠損のない 300mm ウェハーは現実的に製造不可能である以上、複数の LSI を搭載したボードで実装する必要がある。

3.1.2. Rational TWIRL

図 3 - 2 TWIRL 概略図

図 3 - 2 に示す。ふるい制御部、Largish Station, Smallish Station, Tiny Station, 閾値検出部の 5 つから構成される。

ふるい制御部は、ふるい処理を行うのに必要となる *a,b,R,H,n,m* 多項式 *f* などの基本パラメータを決定する。本処理部の入力は、ふるいのパラメータによりデータのビット長が大きく変化するため、シリアル I/F を用いて行うのが最適と考えられる。

Largish/Smallish/Tiny Station は、ふるいのメイン処理を実行する。ふるい処理は、*f(a,b)*が p_i を素因数に持つと判定されるならば、$\lfloor log_2 p_i \rfloor$を *f(a,b)*の素因数の蓄積情報を保持する記憶領域 *FS(a,b)*に対し加算し、これを *0<a<R, 0<b<H* に属する全ての格子点*(a,b)*に対して繰り返すことで行う。この処理は、ふるい処理で用いられる素因数 p_i が *f(a,b)*の約数であるかどうかを、個々の p_i に関する周期情報を管理することで実現する。周期情報の管理を用いたふるい処理の基本アルゴリズムの詳細については、第 2 章を参照されたい。

全ての Station において、$\lfloor log_2 p_i \rfloor$の加算処理(10bit)は 4096 個の bus line 間で並列に実行される。これらの処理は Largish/Smallish/Tiny Station で共通して行われ、それぞれのブロックの違いはふるい処理で用いる素因数 p_i の大きさである。Rational TWIRL の場合 Largish Station では $5.2\Delta10^5<p_i<3.5\Delta10^9$ について、Smallish は $256< p_i <5.2\Delta10^5$ について、Tiny Station では $p_i<256$ についてそれぞれふるい処理を実行する。このようなデサインを用いることで、ふるい処理に必要となる素因数 p_i の周期情報の管理をそれぞれ Station に含まれる p_i の大きさに適した形で実現しており、p_i ごとにカウンタを用いる典型的な周期情報の管理の実現法

と比較して回路規模を大幅に削減することに成功している。

閾値検出部は、Largish/Smallish/Tiny Station の処理を終了した結果、$FS(a,b)=\Sigma\lfloor log_2 p_i \rfloor$の値が閾値を超えるかどうか判定することで、Candidate を選別する。結果、閾値を超えた*(a,b)* を Candidate として出力する。Candidate は、1cycle あたり最大 4 個[3]出力される。(13bitΔ4)

3.2 節以降では、我々が検討した Largish/Smallish/Tiny Station それぞれの機能仕様を記述する。なお、ふるい制御部、閾値検出部については、TWIRL 論文[ST03]からは実装法を特定できないため、本報告書の評価対象外とする。

3.2. 機能仕様について

3.3,3.4,3.5 節において、Rational TWIRL の Largish Station, Smallish Station, Tiny Station の機能仕様をそれぞれ述べる。各 Station を機能単位に分割し、それぞれをブロックと呼ぶ。ブロックは階層構造をもち、あるブロックの一つ下層のブロックをサブブロックと呼ぶ。 各ブロックに対し、サブブロックとモジュール（後述）を用いたブロック図、ブロックのインターフェース、機能、動作概要を記述することで機能仕様を与える。なお、ブロック名の命名は以下に従うものとする。

- TWIRL 論文で用いている機能単位名については、そのままブロック名として使用
- そうでない機能単位は「～処理部」という形の本報告書独自のブロック名を使用
- 同一の構造をもつ複数のブロックに対しては、「ブロック名#X」という形で命名する。

レジスタやカウンタなど、ブロックを構成する最小の回路機能単位をモジュールと呼ぶことにする。ブロックと異なり、モジュールに関してはその機能を記述するのみにとどめる。

本機能仕様は、TWIRL 提案者による回路デザインの内容を調査した上で、報告者が見直しが必要と判断した部分について、修正および追加を行っている。ただし「修正」とは、TWIRL 提案者のデザインを変更するものではなく、提案者のデザインに従いながら回路規模パラメータのみ見直しを行ったものである。また、「追加」とは、TWIRL を実現するのに必要な機能でありながら、TWIRL 論文には(必要最低限の機能要件を記しているが)実現法を記していない部分の回路について、報告者による回路デザインを追加したものである[4]。追加の回路デザインを行うにあたっては、TWIRL 論文の主張する要件に従った形で行った。これ

---

[3] TWIRL 論文によると Rational TWIRL において Candidate が発生する確率は $1.7\Delta10^{-4}$ であり、4096 個中最大 4 個出力できるようにしておけば、発生する Candidate の個数が出力可能な個数を超える確率は、無視できるほど小さいと提案者たちは主張している。

[4] 具体的には、Emission Triplet 発行処理部の空きアドレス検出処理部と、Buffer において *l* に関するソート処理を行う一連のブロックについて、TWIRL 論文の主張に従った形で報告者による回路デザインの追加を行った。

らの修正、追加の有無については、各セクションの「TWIRL 論文との違いについて」という項目に記している。

3.3. Largish Station

■ブロック図

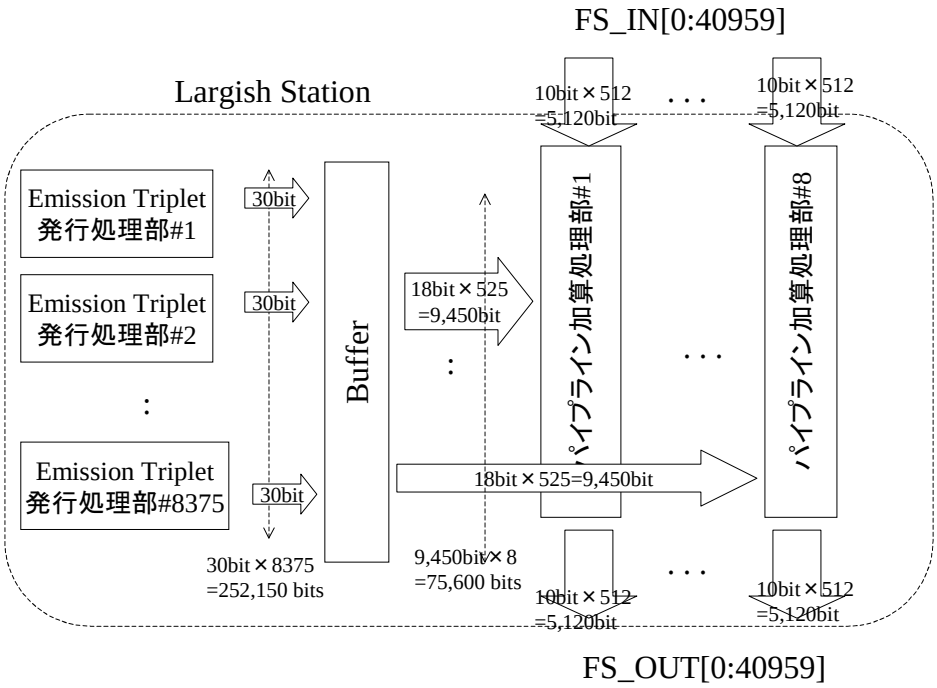


図 3 - 3 Largish Station のブロック図

■インターフェース

名称	I/O	ビット長	説明
FS_IN[0:40959]	I	40,960	10bit の *FS(a,b)*Δ4,096 個
FS_OUT[0:40959]	O	40,960	10bit の *FS(a,b)*Δ4,096 個

■機能

*f(a,b)*が p_i を素因数に持つならば、*FS(a,b)*に$\lfloor \log p_i \rfloor$を加算する処理を、*0ΩaΩR, 0<b<H* に含まれる全ての格子点*(a,b)*について行う。ただし、*5.2Δ10*^{5}*ΩpiΩ3.5Δ10*^{9}である。

■サブブロック

- Emission Triplet 発行処理部#1～#8375
- Buffer
- パイプライン加算処理部#1～#8

■モジュール

なし

■TWIRL 論文との違いについて

- Emission Triplet 発行処理部
  - 個数について見直しを行った。提案者が 8,490 個であると主張しているのに対し、TWIRL 論文の主張に従い必要な個数の見直しを行った結果、報告者は 8,375 個で良いと判断した。詳細は第 2 章を参照。
  - 空きアドレス検出処理部について、報告者による回路デザインの追加を行った。このブロックは、TWIRL 論文が主張している、ランダムに発生するメモリの空き状況に合わせて書き込み先を決定する機能を実現する。TWIRL は全体的にパイプライン的な高速処理を実現するのに対し、このブロックの機能はランダムに発生するメモリの空き状況に合わせて書き込み先を決定するという、パイプライン処理には向かないランダムなデータの処理である。TWIRL 実現のためには、この機能の適切なデザインが必要と考えられるが、TWIRL 論文には実現法を記していない。よって、報告者が回路デザインを行った。
  - DRAM,SRAM のサイズについて見直しを行った。提案者が主張するように、8,375 個の Emission Triplet ごとに DRAM,SRAM のサイズが異なる場合、全ての Emission Triplet をデザインするためコストが膨大となると予想される。よって報告者は、8,375 個を 12 グループに分類し、それぞれのグループごとに異なる DRAM,SRAM のサイズを設定した。報告者による見直し後の SRAM,DRAM サイズの詳細は表 3 - 1 を参照。

- Buffer
  - l に関するソートを行う一連のブロック（l-ソート処理部,l-重複削除処理部, Delivery Line 出力処理部）について、報告者によるデザイン追加を行った。これらは、TWIRL 論文が主張する、1 cycle あたり平均 2,100 個発生する Emission Triplet の Bus Line 番号 l_i に関するソート(l に関するソート)機能を実現するためのブロックである。しかし、l_i がランダムに 4,096 通りに変化する 2,100 個のデータが、1cycle ごと次々にパイプライン的に流れてくるという条件では、相当慎重なデザインを行わない

限り回路規模が極めて膨大になると考えられる[5]。この機能は提案者が実現法を示していないため、報告者によるデザインを行った。

- パイプライン加算部#
  - Delivery Line の本数と構成について見直しを行った。これは、Delvery Line の本数について Shamir-Tromer に確認した結果、Bus Line を左右 2,048 本ずつにわけ、それぞれに 2,100 本ずつの Delivery Line、合計 4,200 本[6](図 3 - 4)が必要であるとの回答によるものであり、本章のデザインもこれに従う。#

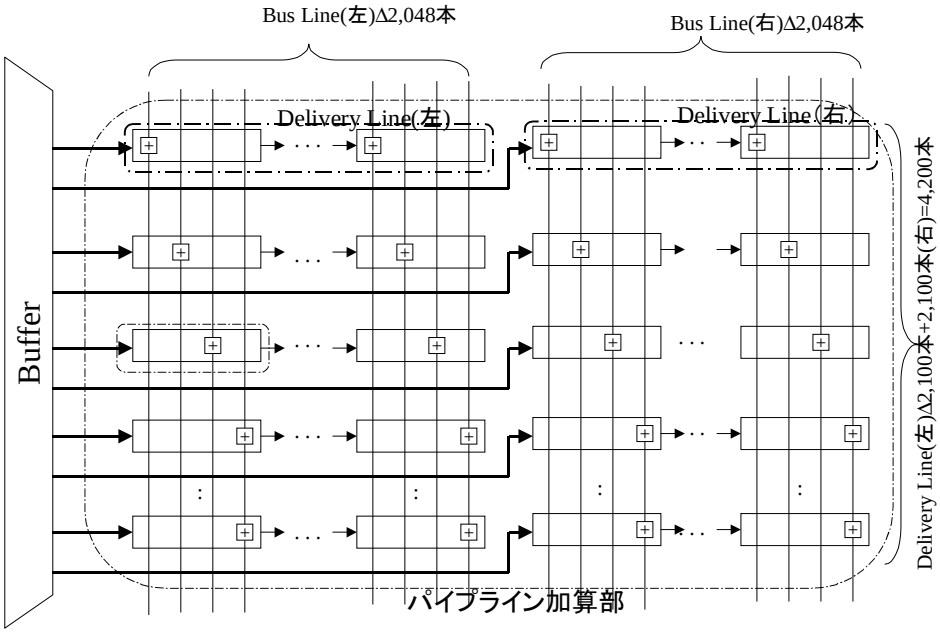


図 3 - 4 本章のデザインにおけるパイプライン加算部の構成

[5] バブルソートを用いた典型的な回路の場合、最低でも 5.6G gates 必要と予想している。
[6] 左右それぞれの 2,100 本の Delivery Line を、Interlevaling(第 2 章参照)構造を考慮してさらに 4 分割することで、525 本Δ4Δ2=4,200 本の Delivery Line とみなすことができる。本章におけるパイプライン加算部#1~#8 は、それぞれこの 525 本の Delivery Line を持つものとしてデザインしている。

■動作概要

概要を図 3 - 5 に示す。以下に示したサブブロック間の連携により、ふるい処理を実行する。

- Emission Triplet 発行処理部#1～#8375

  $f(a,b)$の p_i に関する周期情報を内部の DRAM へ read/write を繰り返しながら管理することで、Emission triplet($\lceil log_2 p_i \rfloor$, l_i, t)およびデータ有無を表す 1bit 値 e_i を Buffer へ出力する。

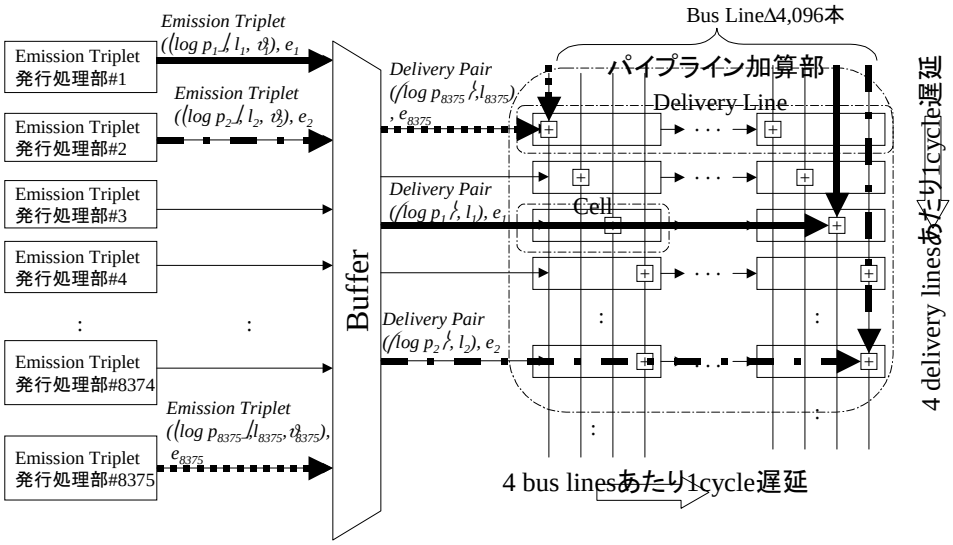
- Buffer

  各 Emission Triplet 発行部から出力された Emission Triplet を入力(30bitΔ8,375=252,150bit)として受け取り、それらを蓄積し、Delivery Pair($\lceil log_2 p_i \rfloor$, l_i), およびデータ有無を表す 1bit 値 e_i に変換した上でパイプライン加算処理部#1～#8 へ出力(9,450bitΔ8=75,600bit)する。

  Delivery Pair を出力する過程においては、時刻 t に Bus Line 番号 l_i で($\lceil log_2 p_i \rfloor$,の加算が行われるように、出力タイミングと出力先の Delivery Line(パイプライン加算処理部 8 個Δそれぞれ 525 本の Delivery Line=計 4,200 通り)について、複数の Emission Triplet 間で調整を行う。

- パイプライン加算部#1～#8

  Buffer から Delivery Pair($\lceil log_2 p_i \rfloor$, l_i), e_i を入力として受け取ることで、パイプライン加算処理部の内部に格子状に配置されている cell において$\lceil log_2 p_i \rfloor$の加算を行う。この処理は、(1)Delivery Pair の l_i が cell の番号と一致しないなら右の cell へ転送(2)l_i が cell 番号と一致するなら、cell 内部のレジスタに$\lceil log_2 p_i \rfloor$を加算、を各 cell が独立して繰り返すことで行う。(2)によって、$FS(a,b)$に$\lceil log_2 p_i \rfloor$を加算する処理が行われる。



• Emission Triplet発行処理部:Emission Triplet $((\lceil \log p_i \rfloor, l_i, v_i)$を発行
• Buffer:l_i番目のBus Lineで時刻 t_iに$\lceil \log p_i \rfloor$が加算されるように、Delivery pair $((\lceil \log p_i \rfloor, l_i)$を発行
• パイプライン加算部:目的のCellに着くまでDelivery Pairの内部転送を繰り返し、到着したら$\lceil \log p_i \rfloor$を加算

図 3 - 5 Largish Station の概要

3.3.1. Emission Triplet **発行処理部**

■ブロック図

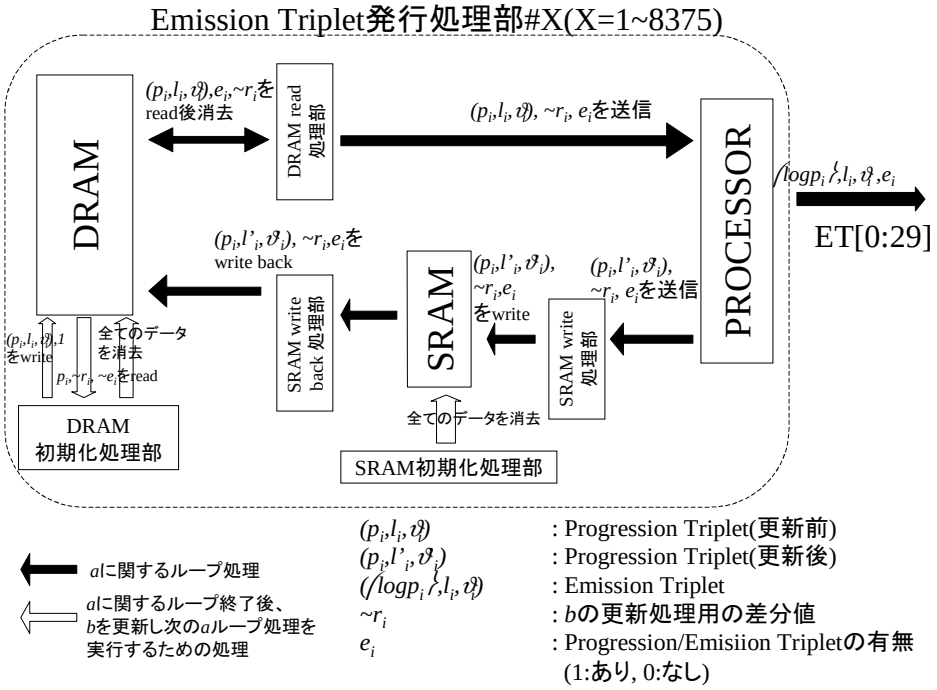


図 3 - 6 Emission Triplet 発行処理部のブロック図

■インターフェース

名称	I/O	ビット長	説明
ET[0:29]	O	30	Emission Triplet($(\lfloor log_2 p_i \rfloor, l_i, \vartheta_i)$, e_i を表すデータ。ただし、$(\lfloor log_2 p_i \rfloor, l_i, \vartheta_i)$, e_i はそれぞれ 6bit,12bit,11bit,1bit であり、e_i は Emission Triplet の有無を表す。(e_i= 1:あり、e_i= 0:なし)

■機能

$f(a,b)$の p_i に関する周期情報を、DRAM へ read/write を繰り返しながら管理することで、Emission triplet($(\lfloor log_2 p_i \rfloor, l_i, \vartheta_i)$を Buffer へ出力する。ただし、$a,b$ は $0<a<R$, $0<b<H$ に含まれる格子点(a,b)である。

■TWIRL 論文との違いについて

- 本ブロック数の個数について報告者による見直しを行った。提案者が 8,490 個であると主張しているのに対し、TWIRL 論文の主張に従い必要な個数の見直しを行った結果、報告者は 8,375 個で良いと判断した。詳細は第 2 章を参照。
- 本ブロックに含まれる DRAM,SRAM のサイズについて報告者による見直しを行った。提案者が主張するように、8,375 個の Emission Triplet ごとに DRAM,SRAM のサイズが異なる場合、全ての Emission Triplet をデザインするためコストが膨大となると予想される。よって報告者は、これらのサイズを、提案者が主張から算出される理論的な最適値以上である、最も低い 2 べき値へ丸め込むことで、8,375 個を 12 グループに分類し、それぞれのグループごとに DRAM,SRAM の設定するように修正した。これの修正結果は、本ブロック全体の回路規模を決定するパラメータである、SRAM_LEN, DRAM_LEN, SRAM_SIZE(=2^{SRAM_LEN}), DRAM_SIZE(=2^{DRAM_LEN})として与えられる。報告者が見直した結果の SRAM,DRAM サイズ、および前述の回路規模パラメータ値の詳細については表 3 - 1 を参照。

■サブブロック

- DRAM
- DRAM read 処理部
- PROCESSOR
- SRAM write 処理部
- SRAM
- SRAM write back 処理部
- DRAM 初期化部
- SRAM 初期化部

■モジュール

なし

■動作概要

0Ω≦a≦R, 0<b<H に含まれる格子点*(a,b)*の探索を、以下の(1)(2)の手順により実行する。(1)*b* を固定し、*a* をインクリメントしながら *0Ω≦a≦R* の範囲の探索を行う (2)(1)の終了後 *b* のインクリメント処理[7]を行い、*b* が *H* 未満であれば(1)を繰り返す。

なお、p_i, l_i, θ_i, e_i, $\sim r_i$, $\sim e_i$ の意味については、図 3 - 8 を参照のこと。

[7] *0Ω≦a≦R* に含まれるすべての *a* の格子点を探索する方法のほかに、*gcd(a,b)=1* となる *a* のみを探索することで *a* の範囲を小さくする高速化テクニックが使用可能である。どちらの方法についても本機能仕様の枠組みで実現可能であると考えられる。

(1)に示す *a* の探索処理を実行するための処理フローは図 3 - 6 における黒色の矢印で、(2)に示す *b* の変更処理のフローは図 3 - 6 における白色の矢印で記す。

(1)*a* の探索処理フロー

以下に示すようにサブブロック間で連携した処理を行うことで、Emission Triplet($\lceil log_2 p_i \rfloor, l_i, v_i$)を BUFFER へ出力する処理を実現する。各サブブロックは、2cycles に 1 回の割合でデータの Input/Output を繰り返す。（これら一連の処理概要は、図 3 - 7 も合わせて参照されたい）

以下の処理において、DRAM と SRAM は、read 処理を行った後、read されたアドレス上のデータを消去する必要がある。この read 後の消去操作処理は、偶数サイクルにおいて read を行い、奇数サイクルにおいて read されたアドレスに対して空データ(e_i=0)を write する処理を行うことで実現する。

- DRAM
  - 偶数サイクル時
    DRAM read 処理部が要求するアドレスの Progression Triplet $(p_i, l_i, v_i), e_i, \sim r_i, \sim e_i$ を read し PROCESSOR へ出力する。DRAM write back 処理部から write back される、更新後の Progression Triplet を格納する。
  - 奇数サイクル時
    偶数サイクル時に read されたアドレスに対し、e_i=0 であるデータの write 要求を、DRAM read 処理部から受け取ることで、偶数サイクル時に read されたアドレス上のデータの消去を行う。

- DRAM read 処理部
  - 偶数サイクル時
    DRAM に対し、指定したアドレスのデータを読み出し、これらのうち Progression Triplet($\lceil log_2 p_i \rfloor, l_i, v_i$), $e_i, \sim r_i$ を PROCESSOR に出力する。
  - 奇数サイクル時
    偶数サイクル時に read したアドレスに対し、e_i=0 であるデータの write 要求を DRAM に発行することで、read 後のデータ消去処理を行う。read されるアドレスカウンタをインクリメントし、DRAM の最大アドレス以上ならば先頭アドレスにリセットする。（周期的なアクセスによる read 処理）

- PROCESSOR
  - 2 cycles に 1 回

DRAM から出力された Progression Triplet を受け取ることで、Emission Triplet($\lceil log_2 p_i \rfloor, l_i, \vartheta$), e_i を BUFFER に出力すると同時に、Progression Triplet の更新処理(p_i, l_i, ϑ) → (p_i, l'_i, ϑ_i)を行い、更新された Progression Triplet(p_i, l'_i, ϑ_i), e_i を SRAM write 処理部へ出力する。

- SRAM write 処理部
  - 2cycles に 1 回
    PROCESSOR から入力された更新後の Progression Triplet(p_i, l'_i, ϑ_i),を SRAM に書き込む処理を行う。SRAM の書き込み先アドレスは、ϑ_i の値と SRAM のデータの空き状況に応じて変化する。（ランダムアクセスによる write 処理）

- SRAM
  - 偶数サイクル時
    SRAM write 処理部から指定されたアドレスに、更新後の Progression Triplet(p_i, l'_i, ϑ_i) e_i,, ~r_i を書き込む。DRAM write back 処理部から要求のあったアドレスより、更新後の Progression Triplet(p_i, l'_i, ϑ_i),e_i,, ~r_i を読み出す
  - 奇数サイクル時
    偶数サイクル時に read されたアドレスに対し、e_i=0 であるデータの write 要求を、DRAM write back 処理部から受け取ることで、偶数サイクル時に read されたアドレス上のデータの消去を行う。

- SRAM write back 処理部
  - 奇数サイクル時
    SRAM 上に書き込まれた更新後の Progression Triplet(p_i, l'_i, ϑ_i) e_i,, ~r_i のデータを、SRAM から DRAM に対し write back 処理を行う。
  - 偶数サイクル時
    一つ前の奇数サイクル時に read したアドレスに対し、e_i=0 であるデータを SRAM に write することでデータ消去処理を行う。write back 元の SRAM のアドレスと、write back 先の DRAM アドレスをインクリメントし、それぞれの最大アドレス以上となったら先頭アドレスにリセットする。（周期的なアクセスによる write back 処理）

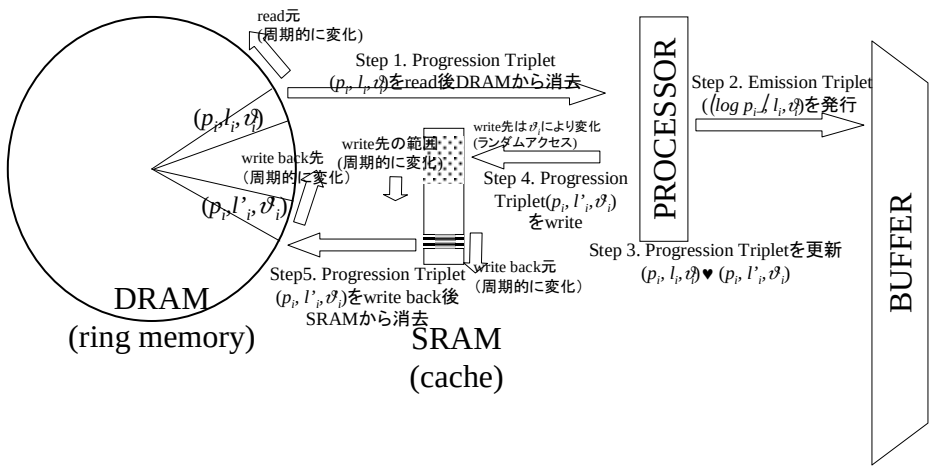


図 3 - 7 Emission Triplet 発行処理部における *a* の探索処理の概略図

(2)*b* の変更処理フロー

- DRAM 初期化部
  DRAM 上に配置されている Progression Triplet 情報の再配置を行う。

- SRAM 初期化部
  SRAM の全てのデータを消去する。

上記に記した処理のほかに、*b* の更新に伴い本ブロックに含まれる全てのカウンタ値の初期値設定処理が必要となる。しかしこの処理は、簡単な処理で実現できると思われるため、本章ではその実現法について省略する。

■備考
本ブロックで用いられる SRAM,DRAM の大きさを表すパラメータ(SRAM_LEN, DRAM_LEN)は、ブロック番号#1~8375 ごとに以下の表 3 - 1 によって与えられる。

表 3 - 1 Emission Triplet 発行処理部の SRAM, DRAM の規模パラメータ SRAMLEN,DRAM_LEN

Emission Triple 発行処理部番号(個数)	DRAM_LEN	SRAM_LEN	DRAM の大きさ	SRAM の大きさ
#1～#839(839)	0	7	0 bit	9,450,496 bit
#840～#1634(795)	8	7	18,113,280 bit	8,954,880 bit
#1635～#2393(759)	9	7	34,586,112 bit	8,549,376 bit
#2394～#3120(727)	10	7	66,255,872 bit	8,188,928 bit
#3121～#3816(696)	11	7	126,861,312 bit	7,839,744 bit
#3817～#4484(668)	12	7	243,515,392 bit	7,524,352 bit
#4485～#5126(642)	13	7	468,074,496 bit	7,231,488 bit
#5127～#5744(618)	14	7	901,152,768 bit	6,961,152 bit
#5745～#6340(596)	15	7	1,738,145,792 bit	6,713,344 bit
#6341～#6914(574)	16	7	3,347,972,096 bit	6,465,536 bit
#6915～#7469(555)	17	8	6,474,301,440 bit	12,503,040 bit
#7470～#8006(537)	18	9	12,528,648,192 bit	24,195,072 bit
#8007～#8375(369)	19	10	17,218,142,208 bit	33,251,328 bit
合計			43,165,768,960 bit	147,828,736 bit

(注 1) #1~#839 については、SRAM のみを用い、DRAM は用いない[8]。

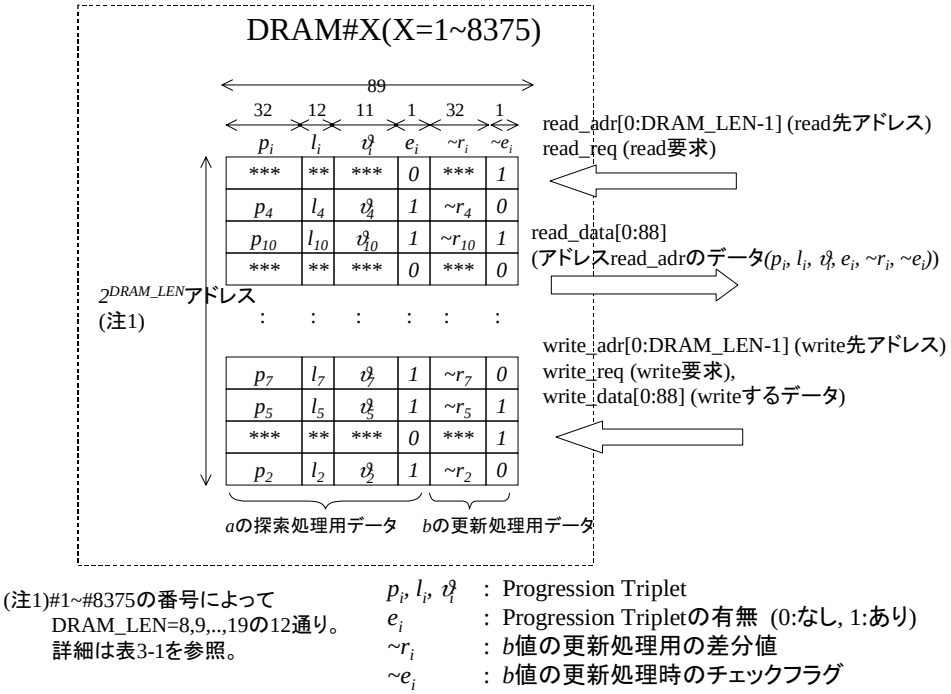
(注 2) DRAM の大きさは、$89\text{bit}\Delta 2^{\text{DRAM\_LEN}}\Delta$(Emission Triplet 発行処理部の個数)である。

(注 3) SRAM の大きさは、$88\text{bit}\Delta 2^{\text{SRAM\_LEN}}\Delta$(Emission Triplet 発行処理部の個数)である。

[8] TWIRL 提案者は、必要な DRAM サイズの理論値が小さい Emission Triplet 発行処理部については、DRAM を用いずに SRAM のみを用いることを推奨している。

3.3.2. DRAM

■ブロック図



(注1)#1~#8375の番号によって
DRAM_LEN=8,9,..,19の12通り。
詳細は表3-1を参照。

p_i, l_i, v_i : Progression Triplet
e_i : Progression Tripletの有無 (0:なし, 1:あり)
$\sim r_i$: b値の更新処理用の差分値
$\sim e_i$: b値の更新処理時のチェックフラグ

図 3 - 8 DRAM のブロック図

■インターフェース

名称	I/O	ビット長	説明
read_adr[0:DRAM_LEN-1]	I	DRAM_LEN (8～19)	read 先アドレス(0~2^{DRAM_LEN}-1)
read_req	I	1	read 要求
read_data[0:88]	O	89	read_adr に格納されているデータ (p_i, l_i, v_i, e_i, $\sim r_i$, $\sim e_i$)
write_adr[0:DRAM_LEN-1]	I	DRAM_LEN (8～19)	write 先アドレス(0~2^{DRAM_LEN}-1)
write_req	I	1	write 要求
write_data[0:88]	I	89	write_adr に書き込むデータ (p_i, l_i, v_i, e_i, $\sim r_i$, $\sim e_i$)

(注１) DRAM#1～#8375 ごとの DRAM_LEN の値については、表 3 - 1 を参照

(注２) p_i, l_i, v_i, e_i, $\sim r_i$, $\sim e_i$ については、図 3 - 8 を参照

■機能

外部ブロックからの read/write 要求と指定されたアドレスに応じて、DRAM 上のデータの read/write 処理を行う。read/write は同一サイクルにて実行(2 ポートによるアクセス)が可能である。DRAM の各アドレスには、*a* に関する探索処理を実行するためのデータ$(p_i, l_i, \vartheta, e_i)$と、*b* を更新するときに必要となるデータ$(\sim r_i, \sim e_i)$の 2 種類が格納されている。

各 DRAM のアドレス長とアドレス数は、DRAM_LEN-bit(=8,9,…,19 の 12 通り)と DRAM_SIZE=2^{DRAM_LEN} という定数で与えられ、DRAM#1～#8375 ごとに異なる。詳細は表 3 - 1 を参照のこと。

■サブブロック

なし

■モジュール

なし

■TWIRL 論文との違いについて

- DRAM のサイズについて報告者による見直しを行った。見直した後のサイズは、回路規模パラメータ DRAM_LEN, DRAM_SIZE(=2^{DRAM_LEN})として与えられる。見直しの理由については 3.3.1 節を、DRAM_LEN, DRAM_SIZE の詳細については表 3 - 1 を参照。

■動作概要

read,write はそれぞれ以下の処理を行う。これら read/write は同一サイクルにて実行(2 ポートアクセス)が可能。

なお、他ブロックから本ブロックに対するアクセスは、偶数サイクルと奇数サイクルで read/write を使い分けてくる場合がある（例えば偶数サイクルで read を行い、奇数サイクルで write を行うなど）。また DRAM はそのメカニズム上、アドレスを指定するため row/column アドレスを 2 cycles にわたって指定する必要がある。つまり、「偶数サイクルで read を行う」という表現だけでは、偶数サイクルにおいて row/column のアドレス指定を開始するのか、終了するのが不明であると思われる。よって、以下本ブロックに対し「X サイクルで read を行う」という表現は、X サイクルにおいて row/column のアドレス指定を開始すると定義する。

- read 処理

  read 要求(read_req)があった場合、アドレス read_adr から 89bit 情報を読み込み、read_data として出力する。

- write 処理
  write 要求(write_req)があった場合、アドレス wrire_adr に対し、89bit の write_data の書き込み処理を行う。

■備考

提案者が主張するように TWIRL を 1GHz で動作させるためには、DRAM も 1GHz で動作させる必要がある。この動作速度は 2004 年 1 月現在の DRAM のテクノロジーでは困難であるように思えるが、これについては(TWIRL 論文に記されていない）解決策が存在する。TWIRL における DRAM へのアクセスはランダムアクセスではなく、アドレスを周期的に変更しながら行われることを利用して、低い動作クロックの DRAM を複数用いて、サイクルごとにアクセスする DRAM をインターリーブ的に変更することで 1GHz のアクセスを実現できる。例えば 100MHz 動作の DRAM を 10 個用意し、クロック数(mod 10)の値に応じて使用する DRAM を切り替えることで、（見た目上）1GHz で動作する DRAM を構築可能である。

3.3.3. DRAM read 処理部

■ブロック図

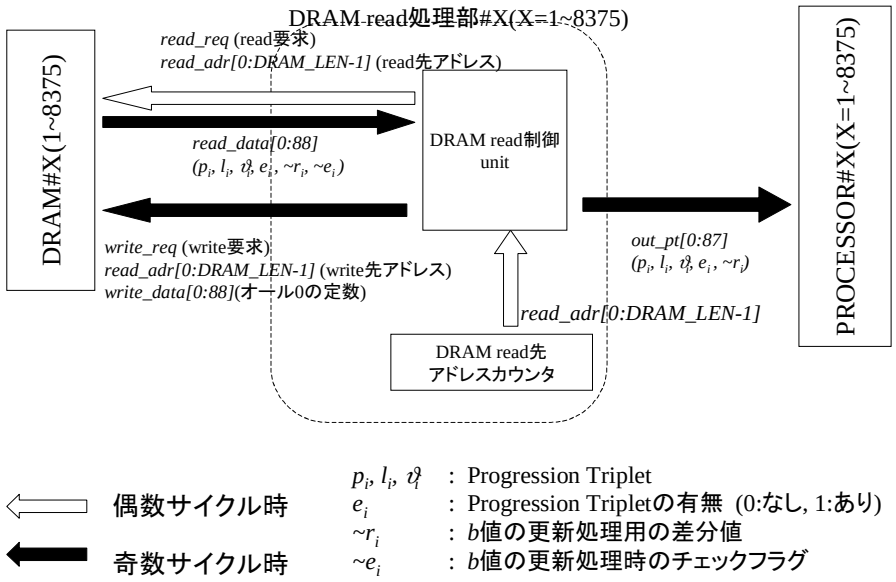


図 3 - 9 DRAM read 処理部のブロック図

■インターフェース

名称	I/O	ビット長	説明
read_adr[0:DRAM_LEN-1]	O	DRAM_LEN (8〜19)	DRAM への read 先アドレス(偶数サイクル時) もしくは write 先アドレス(奇数サイクル時)
read_req	O	1	DRAM への read 要求
read_data[0:88]	I	89	DRAM より read されたデータ $(p_i, l_i, v_i, e_i, \sim r_i, \sim e_i)$
write_req	O	1	DRAM への write 要求
write_data	O	89	00…0(データ消去用の 89bit 定数)
out_pt[0:87]	O	88	PROCESSOR へ送信するデータ $(p_i, l_i, v_i, e_i, \sim r_i)$

(注 1) #1〜#8375 ごとの DRAM_LEN の値については、表 3 - 1 を参照

(注 2) $p_i, l_i, v_i, e_i, \sim r_i, \sim e_i$ については、図 3 - 9 を参照

■機能

read 先アドレスを周期的に変化させながら、DRAM より Progression Triplet 情報の read 処理を行い、PROCESSOR に対して送信する。また、DRAM から read したアドレス上のデータ消去処理も行う。

■サブブロック

なし

■モジュール

- DRAM read 制御 unit
- DRAM read 先アドレスカウンタ(DRAM_LEN-bit バイナリカウンタ)

■TWIRL 論文との違いについて

- DRAM のサイズについて報告者による見直しを行った。見直した後のサイズは、回路規模パラメータ DRAM_LEN, DRAM_SIZE(=2^{DRAM_LEN})として与えられる。見直しの理由については 3.3.1 節を、DRAM_LEN, DRAM_SIZE の詳細については表 3 - 1 を参照。

■動作概要

以下の一連の動作を、2cycle 周期で繰り返す

- DRAM read 制御処理 unit
  - 偶数サイクル時
    read 要求(read_req)と read 先アドレス(read_adr)を DRAM に送信する。
  - 奇数サイクル時
    偶数サイクル時に read 要求したアドレス(read_adr)に対し、e_i=0 であるデータ(write_data)の write 要求(write_req)を奇数サイクル時に DRAM へ送信することで、read したデータの消去処理を行う。
    偶数サイクル時の read 要求により、DRAM から出力されたデータ(read_data)である Progression Triplet$(p_i, l_i, v_i), e_i, \sim r_i, \sim e_i$ のうち、$(p_i, l_i, v_i), e_i, \sim r_i$ で表されるデータを出力データ(out_pt)として PROCESSOR に送信する。

- DRAM read 先アドレスカウンタ
  - 偶数サイクル時
    DRAM read 制御処理部へ read 先アドレス値を送信する
  - 奇数サイクル時
    DRAM_LEN-bit バイナリカウンタのインクリメントを行う。（カウンタ値が 2^{DRAM_LEN} 以上となったら 0 にリセット）

3.3.4. PROCESSOR

■ブロック図

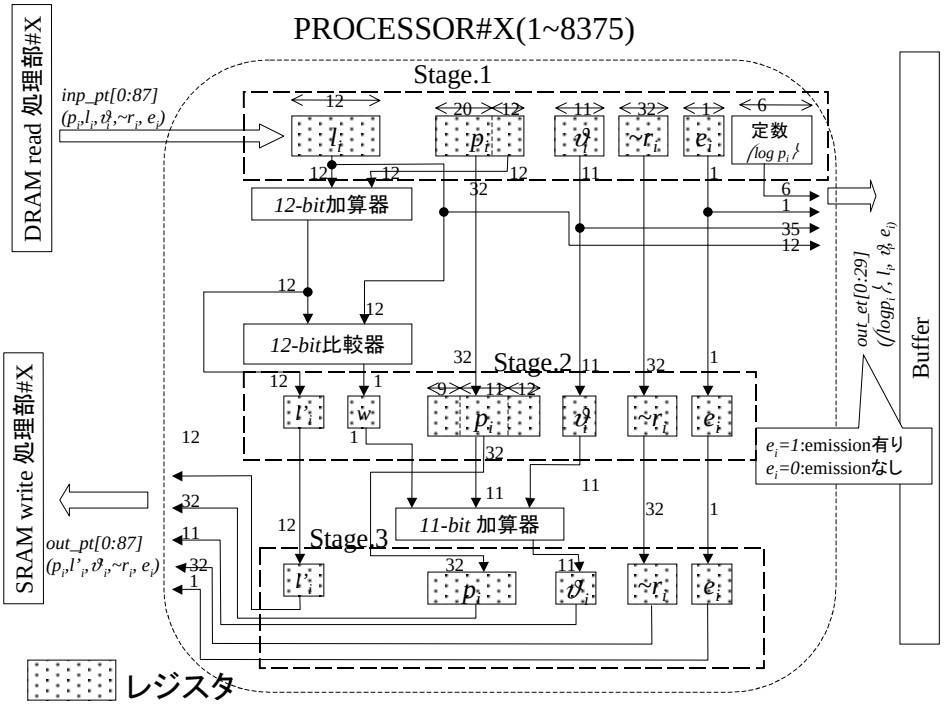


図 3 - 10 PROCESSOR のブロック図

■インターフェース

名称	I/O	ビット長	説明
inp_pt[0:87]	I	88	更新前の Progression Triplet を含んだデータ(p_i, l_i, ϑ_i, e_i, $\sim r_i$, $\sim e_i$), (e_i=*1*:データ有り, e_i=*0*:データなし)
out_et[0:29]	O	30	BUFFER へ送信する Emission Triplet ($\lceil log_2 p_i \rfloor$, l_i, ϑ_i, e_i)
out_pt[0:87]	O	88	更新後の Progression triplet を含んだデータ (p_i, l'_i, ϑ_i, e_i, $\sim r_i$, $\sim e_i$)

(注 1) p_i, l_i, ϑ_i, e_i, $\sim r_i$, $\sim e_i$ については、図 3 - 9 を参照

■機能

Progression Triplet(p_i, l_i, ϑ_i)データが入力された場合、Emission Triplet($\lceil log_2 p_i \rfloor$, l_i, ϑ_i)を生成し

Buffer に出力した上で、Progression Triplet の更新処理$(p_i, l_i, \vartheta_i) \rightarrow (p_i, l'_i, \vartheta_i)$を行う。

■サブブロック

なし

■モジュール

- 32bit レジスタ∆6 個
- 12bit レジスタ∆3 個
- 11bit レジスタ∆3 個
- 1bit レジスタ∆4 個
- 12bit 加算器∆1 個
- 11bit 加算器∆1 個
- 12bit 比較器∆1 個

■TWIRL 論文との違いについて

- 特に無し

■動作概要

以下の処理を 3 段のパイプラインステージで実行する(図 3 - 10)

- DRAMread 処理部→Stage1
  更新前の Progression Triplet を含むデータ$(p_i, l_i, \vartheta_i, e_i, \sim r_i, \sim e_i)$を入力(*inp_pt*)する。
- Stage1→2
  BUFFER へ$((\lfloor log_2 p_i \rfloor, l_i, \vartheta_i), e_i$ を出力(*out_et*)する。
  $l'_i=(l_i+p_i)\ (mod\ 4096))$, および $w(w=1\ if\ l'_i<l_i, w=0\ otherwise)$を計算する。
- Stage2→3
  $\vartheta_i=\vartheta_i+ p_i/4096+w$ を計算し、更新後の Progression Triplet を含むデータ$(p_i, l'_i, \vartheta_i, e_i, \sim r_i, \sim e_i)$
  を SRAM write 処理部へを出力(*out_pt*)する。

3.3.5. SRAM write **処理部**

■ブロック図

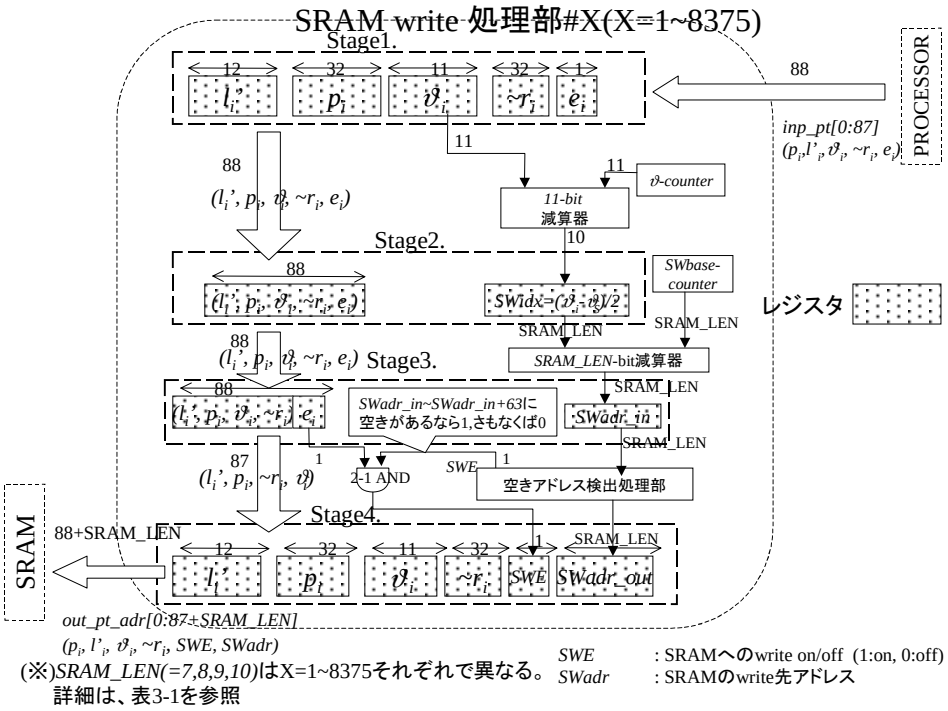


図 3 - 11 SRAM write 処理部のブロック図

■インターフェース

- Output

名称	I/O	ビット長	説明
inp_pt[0:87]	I	88	更新後の Progression triplet を含んだデータ $(p_i, l'_i, \vartheta_i, e_i, \sim r_i, \sim e_i)$
out_pt[0:87+SRAM_LEN]	O	88+SRAM_LEN	更新後の Progression Triplet の、SRAM の書き込み先情報を含んだデータ $(p_i, l'_i, \vartheta_i, e_i, \sim r_i, SWE, SWadr)$

(注 1) $p_i, l_i, \vartheta_i, e_i, \sim r_i, \sim e_i$ については、図 3 - 9 を、*SWE,SWadr* については図 3 - 11 を参照

(注 2)*SRAM_LEN*(=7~10)は *SRAM#X(X=1~8375)*のアドレスのビット長を表す定数であり、#X ごとに異なる。詳細は表 3 - 1 を参照。

■機能

更新された Progression Triplet(p_i, l'_i, ϑ_i, e_i, $\sim r_i$)を書き込む SRAM のアドレスを計算し、SRAM への write 処理を行う。

■サブブロック

空きアドレス検出処理部

■モジュール

- 88bit レジスタ∆2 個
- 32bit レジスタ∆4 個
- 12bit レジスタ∆2 個
- 11bit レジスタ∆2 個
- SRAM_LEN-bit レジスタ∆3 個
- 1bit レジスタ∆2 個
- ϑcounter∆1 個
- SWbase-counter∆1 個
- 11bit 減算器∆1 個
- SRAM_LEN-bit 減算器∆1 個
- 2-1 AND ∆1 個

■TWIRL 論文との違いについて

- SRAM のサイズについて報告者による見直しを行った。見直した後のサイズは、回路規模パラメータ SRAM_LEN, SRAM_SIZE(=2^{SRAM_LEN})として与えられる。見直しの理由については 3.3.1 節を、SRAM_LEN, SRAM_SIZE の詳細については表 3 - 1 を参照。
- サブブロックである空きアドレス検出処理部については、報告者による回路デザインの追加を行った。理由は、提案者がこのブロックの機能の必要性を主張しているにも関わらず、実現法を記していないからである。詳細は 3.3.8 節を参照。

■動作概要

以下の処理をパイプライン的に実行する(図 3 - 10)。Stage1~3 の 3 つのステージから構成され、各ステージの処理を 2 cycles で実行する。

- PROCESSOR→Stage1

  (p_i, l_i, ϑ_i, e_i, $\sim r_i$, $\sim e_i$)を入力(inp_pt)

- Stage1→2

  ϑ=(ϑ*counter の出力値*)より、*SWidx*=(ϑ_i - ϑ)/2 を計算する。

  - ϑ*counter*

    2cycle に 1 回のϑ=ϑ+1 を繰り返す 11bit バイナリカウンタ。

- Stage2→3

  SWbase=(*SWbase-counter の出力値*)より、*SWadr_in*=*SWbase-SWidx* を計算する。この計算は、*SRAM_LEN*-bit の減算器により実行される。

  - *SWbase-counter*

    2cycle に 1 回の *SWbase*=*SWbase+1* を繰り返す *SRAM_LEN*-bit バイナリカウンタ。*SRAM_LEN* は、SRAM#1~#8375 のアドレスのビット長を表す定数(*SRAM_LEN=7,8,9,10*)。詳細は表 3 - 1 を参照。

- Stage3→4

  SWadr_in+j(*j*=0,1,…,63)の中で、SRAM に書き込み可能(アドレスのデータが空)である最小の *j* を探索する処理を行う。この処理は、空きアドレス探索処理部(3.3.8 節参照)により実行される。結果、e_i=1 かつ空きアドレスが発見されたら *SWE*=1 となり、書き込み可能な SRAM のアドレス *SWadr_out*=*SWadr_in+j* が決定される。そうでなければ *SWE=0* となる。

- Stage4→SRAM

  (p_i, l'_i, ϑ_i, e_i, ~r_i, *SWE, SWadr_out*)を SRAM へ出力(*88+SRAM_LEN*-bit)する。*SWE*=1 ならば SRAM のアドレス *SWadr* へ更新された Progression Triplet 情報(p_i, l'_i, ϑ_i, e_i, ~r_i, e_i)の書きこみ処理が行われ、*SWE=0* ならば書き込み処理は行われない。

3.3.6. SRAM

■ブロック図

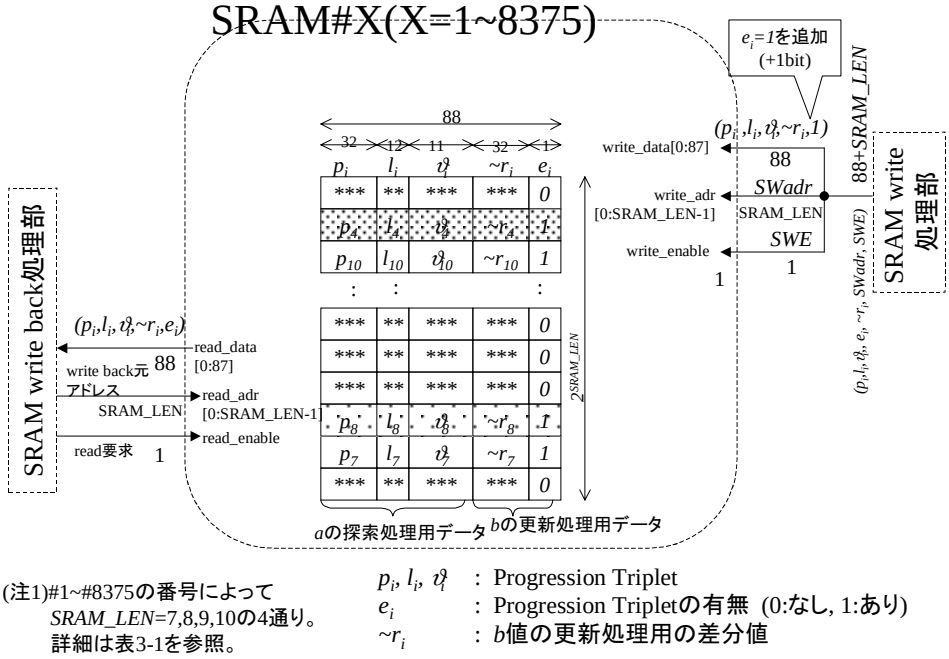


図 3 - 12 SRAM のブロック図

■インターフェース

名称	I/O	ビット長	説明
write_data [0:87]	I	88	アドレス write_adr へ write する、更新された Progression Triplet を含むデータ(p_i, l'_i, ϑ_i, ~r_i)
write_enable	I	1	SRAM への write 要求
write_adr[0:SRAM_LEN-1]	I	SRAM_LEN	SRAM への write 先アドレス
read_data[0:87]	O	88	アドレス read_adr 上の Progression Triplet を含むデータ(p_i, l_i, ϑ_i, ~r_i, e_i)
read_enable	I	1	SRAM への read 要求
read_adr[0:SRAM_LEN-1]	I	SRAM_LEN	SRAM の read 先アドレス

(注 1) p_i, l_i, ϑ_i, e_i, ~r_i については、図 3 - 12 を参照

(注 2)*SRAM_LEN*(=7~10)は SRAM#X(X=1~8375)のアドレスのビット長を表す定数であり、#X ごとに異なる。詳細は表 3 - 1 を参照。

■機能

更新された Progression Triplet 情報$(p_i, l'_i, \vartheta_i, \sim r_i, e_i)$に関する DRAM への書き込みキャッシュ機能を果たす。SRAM への書き込みはランダムアクセスであるが、SRAM から DRAM への write back は SRAM,DRAM ともにアドレスを周期的に変更しながら行う。

各 SRAM のアドレス長とアドレス数は、それぞれ SRAM_LEN-bit(=7,8,9,10 の 4 通り)と SAM_SIZE= 2^{SRAM_LEN} という定数で与えられ、SRAM#1～#8375 ごとに異なる。詳細は表 3 - 1 を参照のこと。

■サブブロック

なし

■モジュール

なし

■TWIRL 論文との違いについて

- SRAM のサイズについて報告者による見直しを行った。見直した後のサイズは、回路規模パラメータ SRAM_LEN, SRAM_SIZE(=2^{SRAM_LEN})として与えられる。見直しの理由については 3.3.1 を、SRAM_LEN, SRAM_SIZE の詳細については表 3 - 1 を参照。

■動作概要

read,write はそれぞれ以下の処理を行う。これら read/write は同一サイクルにて実行可能(2 ポートアクセス)が可能である。

- read 処理

  read 要求(read_req)があった場合、アドレス read_adr から 88bit 情報を読み込み、read_data として出力する。なお、read にかかるレイテンシは 1cycle という前提である。この前提を元に、SRAM,DRAM への write 要求が同一サイクルにて 2 つのブロックから重複しないよう Emission Triplet 発行処理部のデザインを行っている。write 要求の重複については、3.3.7 節を参照。

- write 処理

  write 要求(write_req)があった場合、アドレス write_adr に対し、88bit の write_data の書き込み処理を行う。

3.3.7. SRAM write back **処理部**

■ブロック図

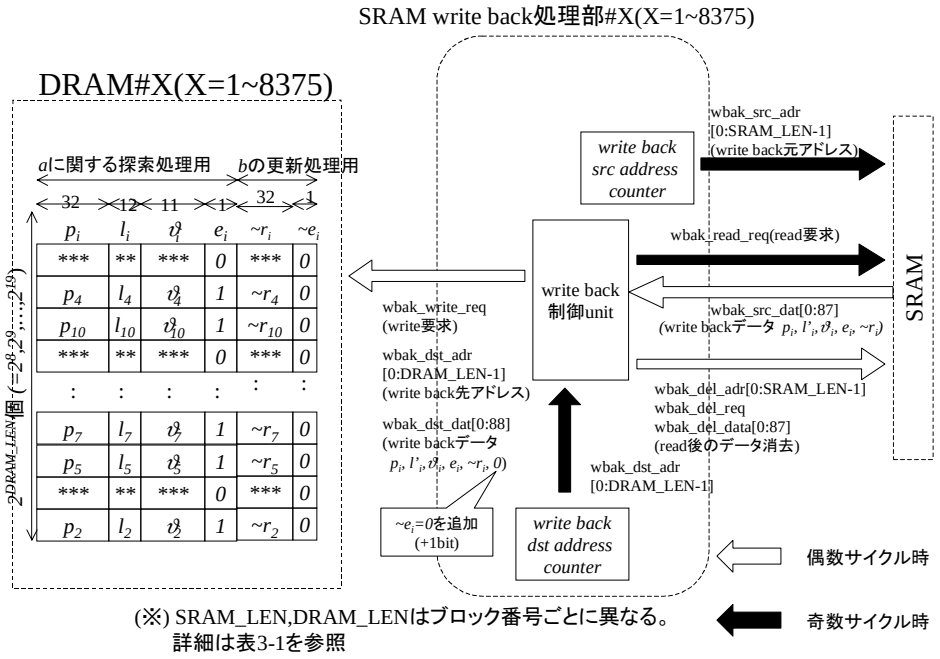


図 3 - 13 SRAM write back 処理部

■インターフェース

名称	I/O	ビット長	説明
wbak_src_adr[0:SRAM_LEN-1]	O	SRAM_LEN	SRAM から read するアドレス
wbak_read_req	O	1	SRAM への read 要求
wbak_src_dat[0:87]	I	88	アドレス read_adr から read された Progression Triplet データ$(p_i, l'_i, v_i, \sim r_i, e_i)$
wbak_dst_adr[0:SRAM_LEN-1]	I	SRAM_LEN	DRAM への write 先アドレス
wbak_write_req	O	1	DRAM への write 要求
wbak_dst_dat[0:87]	O	89	DRAM へ書き込む、Progression Triplet データ$(p_i, l'_i, v_i, \sim r_i, e_i, \sim e_i)$
wbak_del_adr[0:SRAM_LEN-1]	O	SRAM_LEN	read 後、データ消去対象となる SRAM のアドレス
wbak_del_req	O	1	SRAM への write 要求

wbak_del_dat[0:87]	O	88	オール 0 である定数。($e_i=0$)

■機能

SRAM に格納されている Progression Triplet 情報$(p_i, l'_i, v'_i, \sim r_i, e_i)$を read し、DRAM へ write することで、キャッシュデータの write back 処理を実現する。この write back 処理は、SRAM,DRAM ともにアドレスを周期的に変更しながら行う。write back 時に read された SRAM のデータは、read 後に消去する。

■サブブロック

なし

■モジュール

- write back src address counter
- write back dst address counter
- write back 制御 unit

■TWIRL 論文との違いについて

- DRAM, SRAM のサイズについて報告者による見直しを行った。見直した後のサイズは、回路規模パラメータ DRAM_LEN, DRAM_SIZE(=2^{DRAM_LEN}), SRAM_LEN, SRAM_SIZE(=2^{SRAM_LEN})として与えられる。見直しの理由については 3.3.1 を、DRAM_LEN, DRAM_SIZE, SRAM_LEN, SRAM_SIZE の詳細については表 3 - 1 を参照。

■動作概要

以下に従い、SRAM からの read 処理と DRAM への write 処理を行う。SRAM から read 後のデータ消去操作処理は、奇数サイクルにおいて SRAM への read を行い、次の偶数サイクルにおいて read されたアドレスに対して空データ($e_i=0$)を SRAM へ write することで実現する。DRAM read 処理部と SRAM write 処理部は偶数→奇数のサイクルの繰り返しであるのに対し、本ブロックは奇数→偶数のサイクルの繰り返しとなることに注意。これは DRAM を 2 ポートアクセスで実現する(奇数サイクルにて、DRAM/SRAM への write 要求が DRAM read 処理部/SRAM write 処理部と重複しない）ために必要である。

- write back 制御 unit
  - 奇数サイクル時

    SRAM に対して、read 要求(*wbak_read_req*)と read 先のアドレス(*wbak_read_adr*)を入力することで、Progression Triplet を含んだ 88bit データ$(p_i, l_i, v'_i, \sim r_i, e_i)$を SRAM より read する。*wbak_read_adr* は write back src address coutner によって保持される

✧ write back src address counter

2cycles に 1 回インクリメントする *SRAM_LEN*-bit バイナリカウンタ。

➢ 偶数サイクル時

DRAM に対して、write 要求(*wbak_write_req*)、write 先のアドレス(*wbak_read_adr*)を与え、1 つ前の奇数サイクル時に SRAM から read したデータを write する処理を行う。

また、1 つ前の奇数サイクル時に read したアドレスを write 先アドレス(*wbak_del_adr*)とし、$e_i=0$ としたデータ(*wbak_del_dat*)の write 要求(*wbak_del_req*)を SRAM に対して発行することで、read 後の SRAM のデータ消去処理を行う。

● DRAM への write 処理

DRAM に対して、write 要求と write back dst address counter が示す DRAM のアドレスを入力し、SRAM から read した 88bit データ$(p_i, l_i, v_i, \sim r_i, e_i)$を write する。

➢ write back dst address counter

2cycles に 1 回インクリメントする *DRAM_LEN*-bit バイナリカウンタ

3.3.8. 空きアドレス検出処理部

■ブロック図

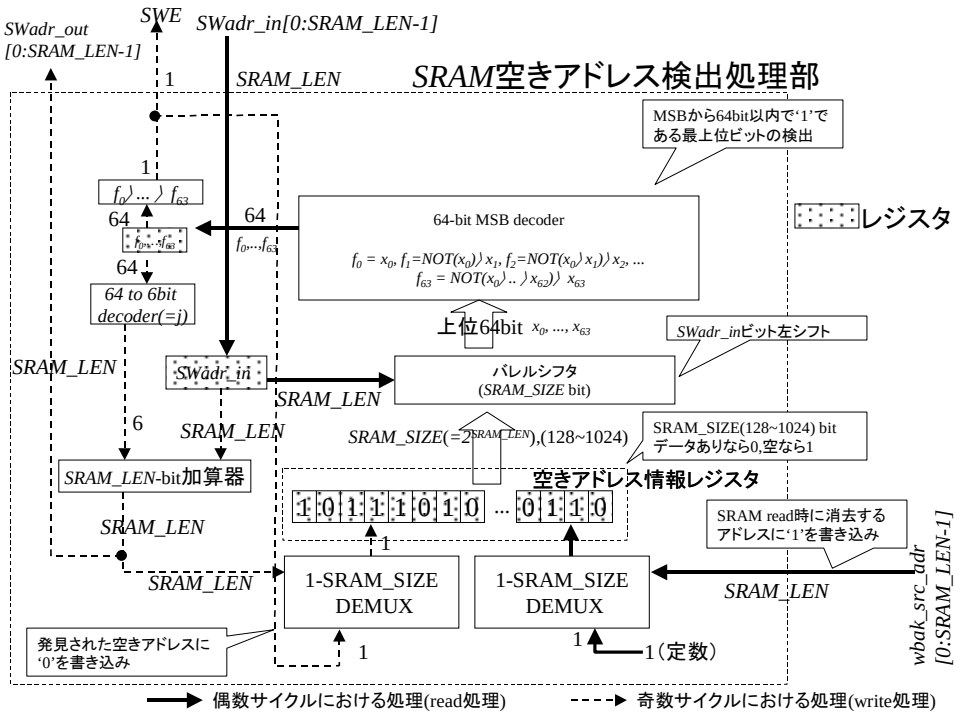


図 3 - 14 空きアドレス検出処理部のブロック図

■インターフェース

名称	I/O	ビット長	説明
SWadr_in[0:SRAM_LEN-1]	I	10	書き込み先候補の最下位アドレス
SWE	O	1	*SDadr_in~SDadr_in+63* の範囲での空きアドレスの有無の判定結果。(0:なし, 1:あり)
SWadr_out[0:SRAM_LEN-1]	O	10	書き込み先アドレス
wbak_src_adr	I	10	SRAM read に伴い、データを消去するアドレス

(注 1) *SRAM_LEN(=7, … ,10), SRAM_SIZE(=2^{SRAM_LEN})*の値はそれぞれ SRAM のアドレスのビット長を表し、X= 1~8375 ごとに異なる。詳細は表 3 - 1 を参照。

■機能

更新された Progression Triplet データ*(p_i, l'_i, v_i, e_i,~r_i)*を書き込む、SRAM のアドレス値を決

定する処理を行う。

■サブブロック

なし

■モジュール

- *64*-bit レジスタ
- *SRAM_LEN*-bit レジスタ
- *SRAM_SIZE*-bit レジスタ
- *1bit, 1-SRAM_SIZE* デマルチプレクサ2 個
- *SRAM_LEN*-bit 加算器
- 1bit, 64-1 OR gate
- 64-bit MSB decoder
- 64bit-6bit decoder
- *SRAM_SIZE*-bit バレルシフタ

■TWIRL 論文との違いについて

- 空きアドレス検出処理部について、報告者による回路デザインの追加を行った。このブロックは、TWIRL 論文が主張している、ランダムに発生するメモリの空き状況に合わせて書き込み先を決定する機能を実現する。TWIRL は全体的にパイプライン的な高速処理を実現するのに対し、このブロックの機能はランダムに発生するメモリの空き状況に合わせて書き込み先を決定するという、パイプライン処理には向かないランダムなデータの処理である。TWIRL 実現のためには、この機能の適切なデザインが必要と考えられるが、TWIRL 論文には実現法を記していない。よって、報告者による独自のデザインを行った。

■動作概略

偶数サイクルと奇数サイクルで、それぞれ以下の処理を交互に実行することで、アドレス *SWadr_in+j (mod SRAM_SIZE) (j*=0,1,..,63*)*のデータが空である最小の *j* を発見する処理を 2cycles 単位で行う。もし、空であるアドレスが発見されたならば *SWE*=1 と *SWadr_out=SWadr_in+j* を出力し、そうでなければ *SWE*=0 を出力する。また、SRAM への read に伴うデータ消去の結果を、内部の空きアドレス情報レジスタに対し反映する。

- 偶数サイクル
  - バレルシフタを用いて、空きアドレス情報レジスタを *SWadr_in* ビット左シフトする。なお、空きアドレス情報レジスタは、SRAM の各アドレスの空き状態をビッ

ト列として表したも *SRAM_LEN* ビットレジスタであり、'1'なら対応するアドレスの SRAM のデータが空、'0'なら対応するアドレスの SRAM データが存在することを意味する。本シフト処理の結果、SRAM の空き状態を表すビット列が *SWadr_in* を MSB とした形に並べ替えられる。

- 左シフトした結果の上位 64bit(x_0,…,x_{63})を 64-bit MSB decoder にかけることで、MSB から 64bit 以内で MSB に最も近く、かつ'1'であるビット位置を検出する。結果は、64bit 値 f_0,…f_{63} としてレジスタに格納され、MSB から j(=0,..,63)番目のビットが'1'なら $f_j=1$ であり、それ以外は'0'となる。すべて 0（空きアドレスがない）なら f_x は全て'0'である。
- 外部から *wbak_src_adr* を受け取り、空きアドレス情報レジスタの *wbak_src_adr* 番目のレジスタ値を'1'に書き込む。

- 奇数サイクル
  - f_0,…,f_{63} を 64-bit to 6 bit decoder にかけ、f_j=1 を満たす j を 6 ビット値にデコードすることで、*SWadr_in* から数えた空きアドレスの位置情報 j を得る。
  - f_0,…,f_{63} の各ビットを OR した結果を *SWE* として出力する。f_0,..,f_{63} がオールゼロならば *SWE=0*(空きアドレス無し)であり、そうでなければ *SWE*=1(空きアドレスあり)である。
  - j から *SWadr_out = SWadr_in+j* を計算、*SWadr_out* を出力する。
  - 空きアドレス情報レジスタの *SWadr_out* 番目に'0'を書き込む。

なお、上記の 64bit MSB decoder と 64bit to 6bit decoder は、以下によって与えられるデコーダである。

- 64-bit MSB decoder

  x_i*(i=0,1,…,63)*のうち、x_y*=1* となる最小の *i=y* について f_i=1 とし、*i=y* 以外は f_i=0 として出力する。すべての *i* について x_i=0 なら、f_i をオール 0 として出力する。

  input : x_i *(*x_i*:1bit, i*=0,1….,63*)*

  output : f_i *(*f_i*:1bit, i*=0,1,…,63*)*

  $f_0=x_0$

  $f_1=NOT(x_0)\rangle x_1$

  $f_2=NOT(x_0\rangle x_1)\rangle x_2$

  $f_3=NOT(x_0\rangle x_1\rangle x_2)\rangle x_3$

  :

  $f_{63}=NOT(x_0\rangle x_1+\ldots\rangle x_{62})\rangle x_{63}$

- 64-bit to 6bit decoder

  f_i(i=0,1,…,63)のうち、f_j=1 となる最小の j の値を、6bit 値の y_i として出力する。

  input : f_i (f_i:1bit, i=0,1,….,63)

  output : y_i (y_i:1bit, i=0,1,….,5)

  $y_0 = f_{32}+f_{33}+...+f_{63}$

  $y_1 = f_{16}+...+f_{31}+... + f_{48} ... + f_{63}$

  $y_2 = f_8+...+f_{15}+... + f_{56} + ... + f_{63}$

  $y_3 = f_4+...+f_7+... + f_{60} + ...+ f_{63}$

  $y_4 = f_2 + f_3 + f_6 + f_7 + ... + f_{62} + f_{63}$

  $y_5 = f_1 + f_3 + ...+ f_{63}$

3.3.9. DRAM **初期化部**

■ブロック図

省略

■インターフェース

名称	I/O	ビット長	説明
PT_read_req	O	1	DRAM への read 要求
PT_read_adr[0:DRAM_LEN-1]	O	DRAM_LEN	DRAM から read するアドレス
PT_read_dat[0:88]	I	89	DRAM への read された Progression Triplet$(p_i, l_i, v_i), e_i, \sim r_i, \sim e_i$
PT_write_req	O	1	DRAM への write 要求
PT_write_adr[0:DRAM_LEN-1]	O	DRAM_LEN	DRAM から write するアドレス
PT_write_dat[0:88]	O	89	DRAM への write される、更新後の Progression Triplet$(p_i, l'_i, v'_i), e_i, \sim r_i, \sim e_i$

(注 1)$p_i, l_i, v_i, e_i, \sim r_i, \sim e_i$ の定義は図 3 - 8 を参照

(注 2)*DRAM_LEN*(=8~19)は DRAM#X(X=1~8375)のアドレスのビット長を表す定数であり、#X ごとに異なる。詳細は表 3 - 1 を参照。

■機能

b のインクリメントに伴う、DRAM 上の Progression Triplet 情報の更新処理を行う。

■モジュール

- 1bit レジスタ∆2 個
- 12-bit レジスタ∆2 個
- 20-bit レジスタ∆1 個
- 32-bit レジスタ∆2 個
- DRAM_LEN-bit バイナリカウンタ∆1 個
- 12-bit 加算器
- 12-bit 比較器
- 20-bit 加算器

■TWIRL 論文との違いについて

- DRAM のサイズについて報告者による見直しを行った。見直した後のサイズは、回路規模パラメータ DRAM_LEN, DRAM_SIZE(=2^{DRAM_LEN})として与えられる。見直しの理由については 3.3.1 を、DRAM_LEN, DRAM_SIZE の詳細については表 3 - 1 を参照。

■動作概略

以下のアルゴリズムに従い、DRAM 上の(p_i, l_i, v_i), e_i, $\sim r_i$, $\sim e_i$ の再配置を行うことで、b に関する更新処理を行う。基本的には、e_i=1(Progression Triplet データが有効)かつ$\sim e_i$=0 (更新前)であるデータを DRAM から read し、Progression Triplet データを更新したあと、更新した Progression Triplet データを、適切な DRAM のアドレスに対して write(再配置)を行う。

a に関するループ処理と違い、高度なパイプライン的な並列動作を実施する必要はない。よって簡単な処理であり、いろいろな方法で実現可能であるため、本報告書では詳細な実装を特定しない。以下のアルゴリズムを実施するのに必要最低限のモジュールを列挙するにとどめる。

1. DRAM 上の$\sim e_i$を‘0’にセットすることを、DRAM 上の全てのアドレスに対して行う。
（各アドレスに対し、データを read し、$\sim e_i$のみを'0'として write することを繰り返す）
全てのアドレスに対してセットを完了したら、2.の処理を実行する。

（$\sim e_i=0$:「更新前」を意味するフラグ値, $\sim e_i$=1:「更新後」を意味するフラグ値）

2.以下の 2.1~2.7 に従い、全ての DRAM アドレスの Progerssion Triplet 情報を更新する。

2.1 $x=0$ とする。

2.2 DRAM アドレス x から $(p_i, l_i, v_i, e_i, \sim r_i, \sim e_i)$を read。

2.3 2.1.で read したデータが e_i=0 もしくは$\sim e_i$=1 ならば x をインクリメントし、xが2^{DRAM_LEN} となったら終了。さもなくば 2.2 へ戻る。

2.4 アドレス x に、e_i=0 としたデータを write することでデータ消去

2.5 $l'_i=l_i+\sim r_i \ (mod\ s)$, $v_i=\sim r_i/s+w$, $(w=1 \text{ if } l'_i<l_i, w=0 \text{ otherwise})$, $y=v_i/2$ を計算$(s=4096)$

2.6 DRAM アドレス y の e_i が‘0’となるまで y をデクリメント(空き領域探索)

2.7 DRAM アドレス y に対し、$e_i=\sim e_i$=1 とした上で $(p_i, l'_i, v_i, e_i, \sim r_i, \sim e_i)$を write
x をインクリメントし、xが2^{DRAM_LEN} となったら終了。さもなくば 2.2 へ戻る。

1.では、DRAM のアドレス上の更新前のデータと更新後のデータを区別するフラグ$\sim e_i$を、全てのアドレスに対して$\sim e_i=0$(更新前)に初期設定する。

2.2~2.3 では、更新前($\sim e_i=0$)かつ空でない($e_i=1$)データを DRAM の上位アドレスから下位アドレスの順にスキャンする。条件を満たしたデータが発見された場合、データを消去(2.4)し、$l_i \rightarrow l'_i$, , $v_i \rightarrow v_i$ の更新処理,更新したデータ$(p_i, l'_i, v_i, e_i, \sim r_i, \sim e_i)$の書き込み先候補の先頭アドレス y の計算を行う(2.5)。アドレス y のデータが空でなければ、y をデクリメントすることを繰り返すことで更新したデータ$(p_i, l'_i, v_i, e_i, \sim r_i, \sim e_i)$を書き込み可能な空きアドレスを探

索し(2.6)、発見されたら書き込みを行う(2.7).

3.3.10. SRAM **初期化部**

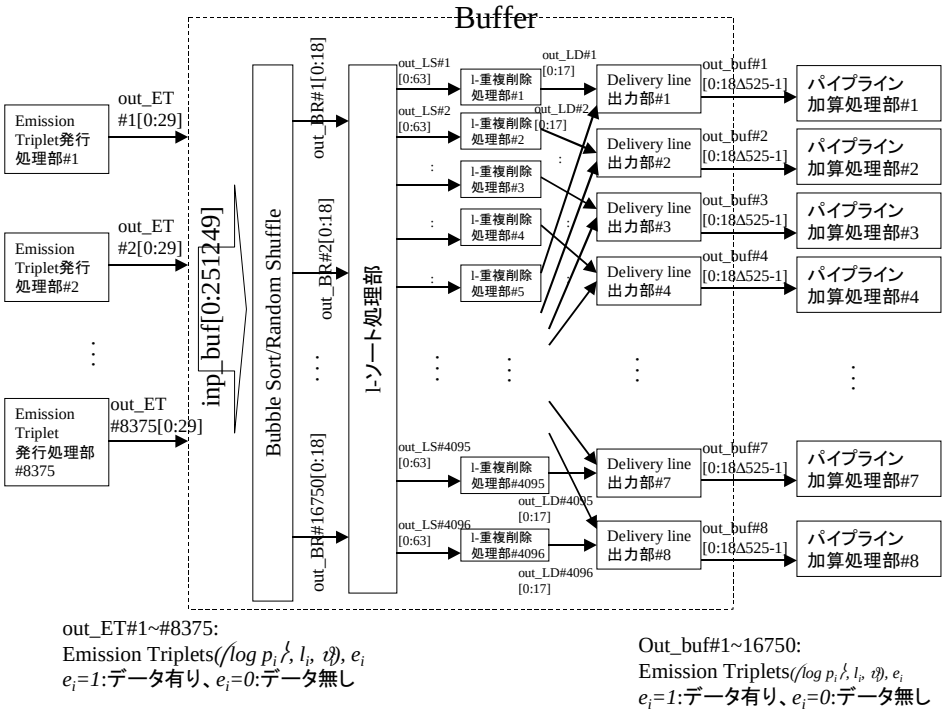
■機能

SRAM の全てのアドレスの e_i を 0 にセットすることで、SRAM をすべてクリアする。SRAM 空きアドレス検出処理部の空きアドレス情報レジスタの全ての値を'1'にセットすることで、空きアドレス情報をクリアする。

本ブロックの実現は容易であると考えられるので、機能のみの記述にとどめる。

3.3.11. Buffer

■ブロック図

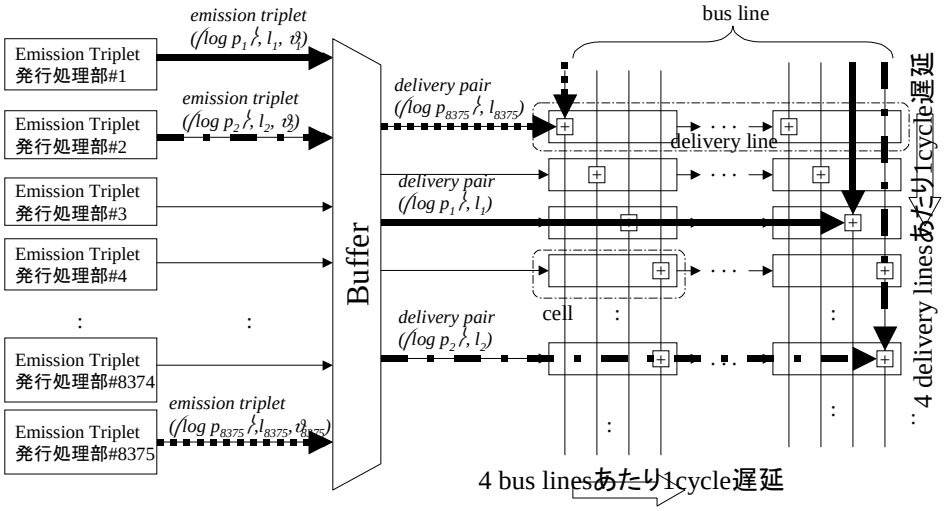


■インターフェース

名称	I/O	ビット長	説明
inp_buf[0:251249]	I	251,250 (30Δ8375)	Emission Triplet 発行部から入力される、Emission Triplet を含むデータ($(\lfloor log_2 p_i \rfloor, l_i, t_i), e_i$. 30bit データΔ8375 個=251250bit
out_buf#X[0:9449] (X=1~8)	O	9,450 (18Δ525)	パイプライン加算処理部#X へ出力する、Delivery Pair を含むデータ。$((\lfloor log_2 p_i \rfloor, l_i), e_i$ 18bit データΔ525 個=9,450bit

■機能

図 3 - 15 に概要を示す。Emission Triplet 発行処理部#1~#8375 から入力される、複数の Emission Triplet を内部に一時的に溜め込み、溜め込んだ複数の Emission Triplet 間で、出力タイミングと出力先（パイプライン加算処理部番号#1~8、各パイプライン加算部の 525 個の Delivery line で計 4,200 通りの組み合わせ）の調整を行い、Delivery Pair に変換した形で出力する。



複数のEmission Triplet発行処理部が生成したemission triplet *(⌊log p_i⌋, l_i, v_i)* を溜め込み*⌊log p_i⌋*がl_i番目のbus lineに対し時刻v_iにおいて加算されるように、delivery pair *(⌊log p_i⌋, l_i)*をタイミングよく発行する。

図 3 - 15 Buffer の機能の概要

■サブブロック

- Bubble Sort/Random Shuffle
- *l*-ソート処理部
- *l*-重複削除処理部#1~#8
- Delivery line 出力部#1~#8

■サブモジュール

なし

■TWIRL 論文との違いについて

- *l*-ソート処理部,*l*-重複削除処理部,Delivery Line 出力部(k ビット値 M-4Δ2 分配処理部)

上に示した一連のサブブロックについて、報告者による追加デザインを行った。これは、提案者が Buffer の機能として、*l* に関するソートを行った上で Delievry Line の出力する機能が必要であると主張しているにも関わらず、この機能の実現法を記していないことである。TWIRL 論文の主張の範囲では、これらのブロックに相当する箇所は「*l* に関するソート処理、データのマージを実行するパイプライン的なネットワークによる処理を実行し、適切なタイミングで Delivery Line へデータ出力する」

と主張しているので、この主張に沿った形で、報告者による回路デザインを行った。

■動作概要

以下に示すようにサブブロック間で連携した処理を行うことで、Emission Triplet 間の調整を行いながら Delivery Pair に変換して出力する処理を実現する。各サブブロックは 1cycle に 1 回の割合でデータの Input/Output を繰り返す。

- Bubble Sort/Random Shuffle

  各 Emission Triplet 発行部から入力される複数の Emission Triplet($(\lfloor log_2 p_i \rfloor, l_i, \theta)$, e_i(30bit)の間で、θ値をそろえて出力する処理を実行する。

  これは、内部でθに関するソートとランダムなシャッフル処理を繰り返すことで、本ブロック内部のθのカウンタ値とθの値が同一である Emission Triplet を選別し、Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i)$, e_i (19bit)に変換したうえで l-ソート処理部に出力することで実現する。

- *l*-ソート処理部

  Bubble sort/Random shuffle から入力された複数の Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i)$, e_i(19bit)の間で、l_i に関するソート処理を行う。この処理の結果、Delivery Pair が 4096 通りの l_i の値に応じて分類され、それぞれ *l*-重複削除処理部#1~#4096 へ出力される。

  出力時に、それぞれの Delivery Pair は、l_i の値の最上位 1 ビット、および最下位 2 ビットの合計 3 ビットが削除された 9 ビットの l_i による Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i)$, e_i (16bit)に変換される。*l*-重複削除処理部#1~#4096 に対しては、l_i ごとに最大 4 個の Delivery Pair が重複した 64bit データとして出力する。

- *l*-重複削除処理部#1~#4096

  l-ソート処理部から入力された複数の Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i)$, e_i(19bit)について、l_i の値が重複（最大 4 重複）する複数の Delivery Pair($(\lfloor log_2 p_1 \rfloor, l)$, … , $(\lfloor log_2 p_4 \rfloor, l)$の$\lfloor log_2 p_i \rfloor$,$e_i$ に対し$\lfloor log_2 p' \rfloor = \lfloor log_2 p_1 \rfloor + \ldots + \lfloor log_2 p_4 \rfloor$, $e' = e_1 \rangle \ldots \rangle e_4$ の処理を行うことで、1 つの Delivery Pair($(\lfloor log_2 p' \rfloor, l)$, e'(18bit)にまとめて、Delivery Line 出力部へ出力する処理を行う。

  まとめられた Delivery Pair は、ブロック番号#1~#4096 に対応する *l* の値の最上位 1 ビット、および最下位 2 ビットの合計 3 ビット値に応じて、8 通りの *l*-重複削除処理部#1~#8 へ出力される。

- Delivery line 出力部#1~#8

  l-重複削除処理から入力された複数の Delivery Pair の間で、出力として接続されている 525 個の Delivery Line、および出力タイミングを調整した上で、Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i)$, e_i(18bit)をパイプライン加算処理部内部の Delivery Line に対して出力する。

3.3.12. Bubble Sort/Random Shuffle

■ブロック図

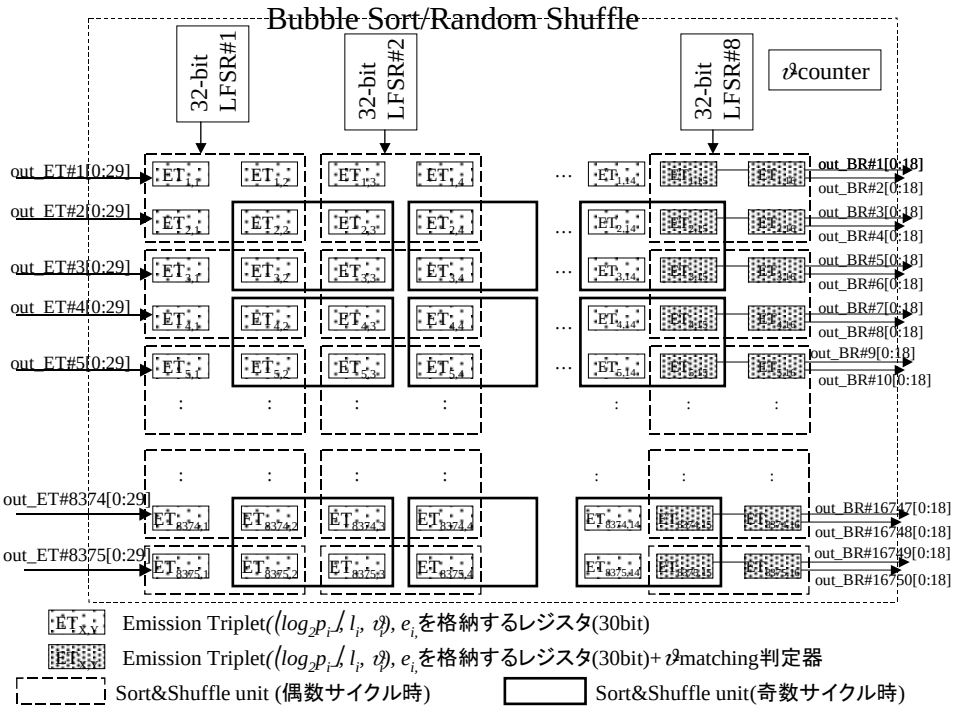


図 3 - 16 Bubble Sort/Random Shuffle のブロック図

■インターフェース

名称	I/O	ビット長	説明
out_ET#X *(X=1~8375)*	I	30Δ8,375 =251,250	Emission Triplet 発行部#1~8375 から入力された、Emission Triplet を含む 30bit データ(($\lceil log_2 p_i \rceil$, l_i, v_1), e_i, (e_i=1:Emission Triplet 有り,e_i=0:Emission Triplet 無し)
out_BR#X *(X=1~16750)*	O	19Δ16,750 =318,250	v_1に関する選別が行われた、Delivery Pair を含む 19bit データ(($\lceil log_2 p_i \rceil$, l_i), e_i (e_i=1:Delivery Pair 有り,e_i=0: Delivery Pair 無し)

■機能

複数の Emission Triplet(($\lceil log_2 p_i \rceil$, l_i, v_1)の間で、v_1に関するソート処理とランダムシャッフル処理を行うことで、v_1の値が本ブロックのv_8のカウンタ値と一致するものを選別し、出力する。

■サブブロック

なし

■モジュール

- 32-bit LFSR #1~#8
- Sort&Shuffle unitΔ62,813 個
- ϑcounterΔ1 個
- ϑmatching 判定器Δ16,750 個
- 30bit レジスタΔ134,000 個

■TWIRL 論文との違いについて

- 基本的に提案者の主張する通りのデザインである。ただし、内部の行数と列数、および ϑmatching 判定を行う列数については提案者が数値を示していなかった（提案者は、適切な大きさに設定すれば、問題ないとのみ主張している）ので、それぞれ報告者が 8,375 行Δ16 列、2 列と設定した。本パラメータ設定の根拠は特にないが、おそらく提案者の主張する「適切な大きさ」の範囲であると予想している。

■動作概要

16Δ8375 の行列状に Emission Triplet($\langle log_2 p_i \rfloor, l_i, \vartheta \rangle, e_i$ を格納した 30bit レジスタが $ET_{X,Y}$($1 \le X \le 8375, 1 \le Y \le 16$)が配置されている。各行に対し ϑ の値に関する Bubble Sort が、各列に対してランダムシャッフルがそれぞれ実行される。これを実現するために、1cycle ごとに以下の 3 つの処理を並列に実行する。

- データ入力

  Emission Triplet($\langle log_2 p_i \rfloor, l_i, \vartheta \rangle, e_i$ を最も左の列のレジスタ($ET_{X,1}$)に書き込むことで入力する。書き込む先に有効なデータがある場合入力データを上書きする[9]。

- ϑ に関するソート処理、Random Shuffle 処理

  Sort & Shuffle unit という、$ET_{X,Y}$ を 2Δ2 単位に区切り、それぞれにおいて、Bubble Sort と Random Shuffle を同時に実行する処理が行われる[10]。Sort&Shuffle unit の処理対象となる $ET_{X,Y}$ の範囲は、偶数サイクルと奇数サイクルでそれぞれ切り替わり、偶数サイク

[9]本ブロックの行数と列数を考慮すると、データ上書きが発生する確率は非常に低いと予想している

[10] 行数が奇数(8375)であるため、最下行($ET_{8375,*}$)のみ 1Δ2 に区切る必要があるが、本章では簡略化して、これらも 2Δ2 の Sort&Shuffle unit により扱われるものとみなす。

ルにおいては($ET_{2x+1,2y+1}$, $ET_{2x+1,2y+1}$, $ET_{2x+2,2y+2}$, $ET_{2x+2,2y+2}$)が(x=0,1,...,4186, y=0,1,...,7)、奇数サイクルにおいては($ET_{2x+2,2y+2}$, $ET_{2x+2,2y+2}$, $ET_{2x+3,2y+3}$, $ET_{2x+3,2y+3}$)が(x=0,1,...,4186, y=0,1,...,6)がそれぞれ対象となる(4,188Δ8+4187Δ7)=62,813 単位)(図 3 - 16)。Random Shuffle を実行するための乱数は、線形シフトレジスタ(図 3 - 16 の LFSR#1~#8)により生成される。

Sort & Shuffle unit の実現法を図 3 - 17 に示す。左右のレジスタでϑの値による比較を行った結果Δ2=2bit と、線形シフトレジスタが生成する 1bit 乱数の合計 3bit の値に応じて、それぞれのレジスタ値のデータ転送先をデマルチプレクサ(d_1~d_4)により選択することで実現する

左右のレジスタをϑの値を比較する際に用いる 11bit 特殊比較器は、図 3 - 18 に示す構成で実現する。このような特殊な比較器を用いる理由は、本来の時刻情報は 35-bit の情報量を持つのに対し、ϑの持つ時刻情報を 11bit に短縮していることによるものである。比較される 2 つの値 x,y が完全にランダムではなく、x と y の差分の最大値が小さい($|x-y|<128$)という性質を利用することで、通常の 11bit 比較器(x,y の値をそのまま比較)に例外処理(x の最上位 4bit=1111 かつ y の最上位 4 ビット=0000 ならば、$x<y$ と判定)を追加することで、時刻情報の比較を実現することができる。

- データ出力

右 2 列($ET_{15,Y}$, $ET_{16,Y}$)のレジスタに含まれる Emission Triplet のうち、ϑの値がϑcounter と一致するものを選別し、19bit の Delivery Pair($\lfloor log_2 p_i \rfloor, l_i$), e_i として l-ソート処理部へ出力する。また、出力したデータは e_i=0 とすることで消去する。

この処理は、上記で述べた「ϑに関するソート処理、Random Shuffle 処理」と同じ処理回路構成に、各 30bit レジスタのϑの値がϑcounter の値と一致するものだけを判別し出力する、ϑmatching 判定器を追加することで実現できる。それぞれのϑmatching 判定器は、30bit レジスタのϑの値と Buffer 全体のϑcounter を比較し、一致するならばレジスタのデータを消去し l-ソート処理部へ出力する。一致しない場合は、「ϑに関するソート処理、Random Shuffle 処理」と同一の処理を実行する。

なお、16,750 個のϑmatching 判定器を一つのϑcounter で担当するのが回路実現上困難である場合、複数のϑcounter を用いる必要がある。

- ϑmatching 判定器

  11bit 比較器と、30-bit 1-2 デマルチプレクサにより実現する。

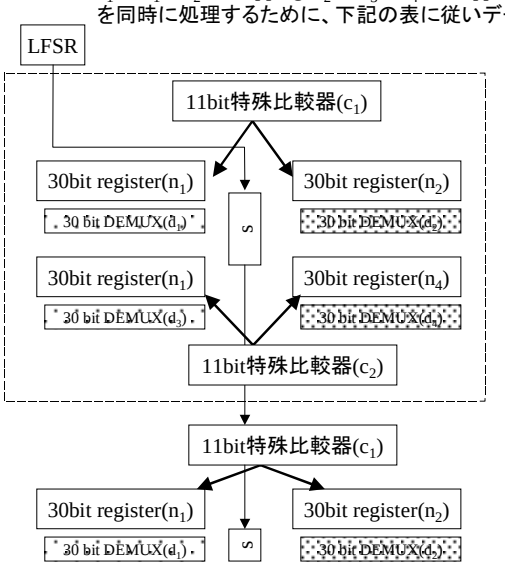
- ϑcounter

  1cycle ごとに+1 増加する 11bit バイナリカウンタにより実現する。

  b の更新処理時に初期化する。

・c_1 = 1 if (n_1のϑ)<(n_2のϑ), otherwise 0,　c_2 = 1 if (n_3のϑ)<(n_4のϑ), otherwise 0
s = (LFSRから転送される1bit乱数)とする。

・c_1=1:n_1⇐n_2のswapping, c_2=1:n_3⇐n_4のswapping, s=1:n_1⇐n_3のswapping
を同時に処理するために、下記の表に従いデータ転送先を選択するデマルチプレクサを使用



値			転送先			
c_1	c_2	s	d_1	d_2	d_3	d_4
0	0	0	n_1	n_2	n_3	n_4
0	0	1	n_3	n_2	n_1	n_4
0	1	0	n_1	n_2	n_4	n_3
0	1	1	n_4	n_2	n_1	n_3
1	0	0	n_2	n_1	n_3	n_4
1	0	1	n_3	n_1	n_2	n_4
1	1	0	n_2	n_1	n_4	n_3
1	1	1	n_4	n_1	n_2	n_3

図 3 - 17　Sort&Shuffle unit の実現法(*ϑ*matching 判定器を含まない列)

ϑ: 35bit →11 bitの短縮に伴い、以下の特殊な比較処理を用いる必要あり

x∅2^{11} – 128かつy<128の場合　.... x<yと判定（1を返す)
それ以外の場合　.... x<yの大小関係を判定する(1 or 0)

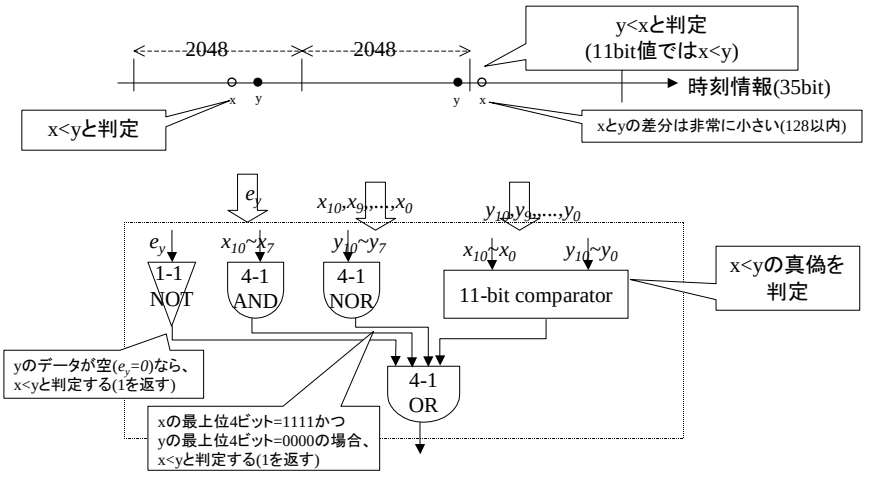


図 3 - 18 11bit 特殊比較器について

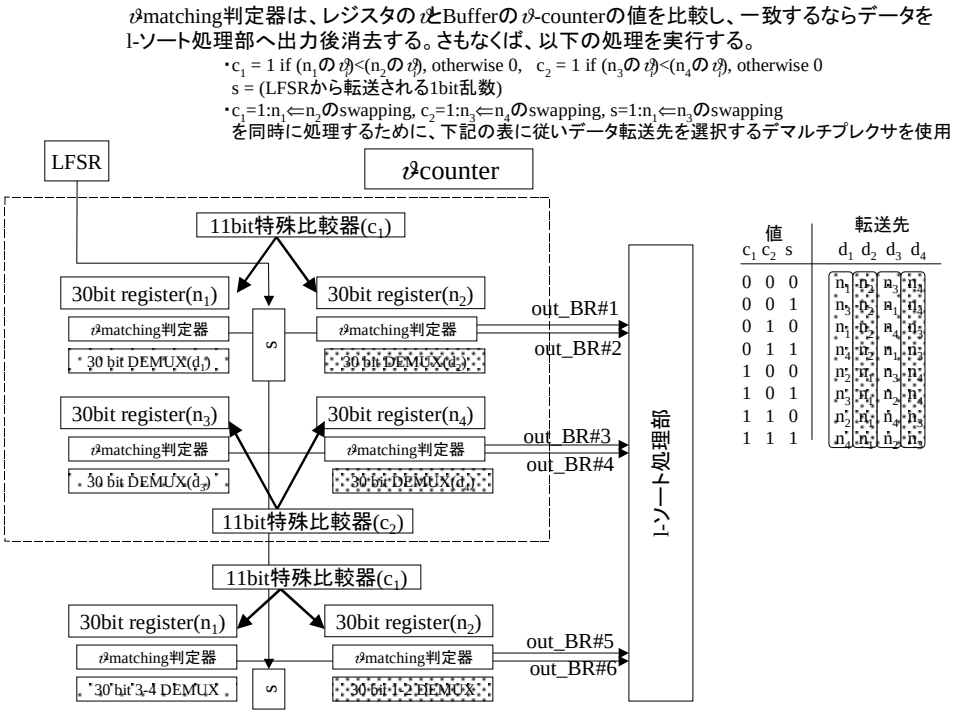


図 3 - 19 Sort&Shuffle unit の実現法(ϑmatching 判定器を含む列)

3.3.13. *l*-ソート処理部

■ブロック図

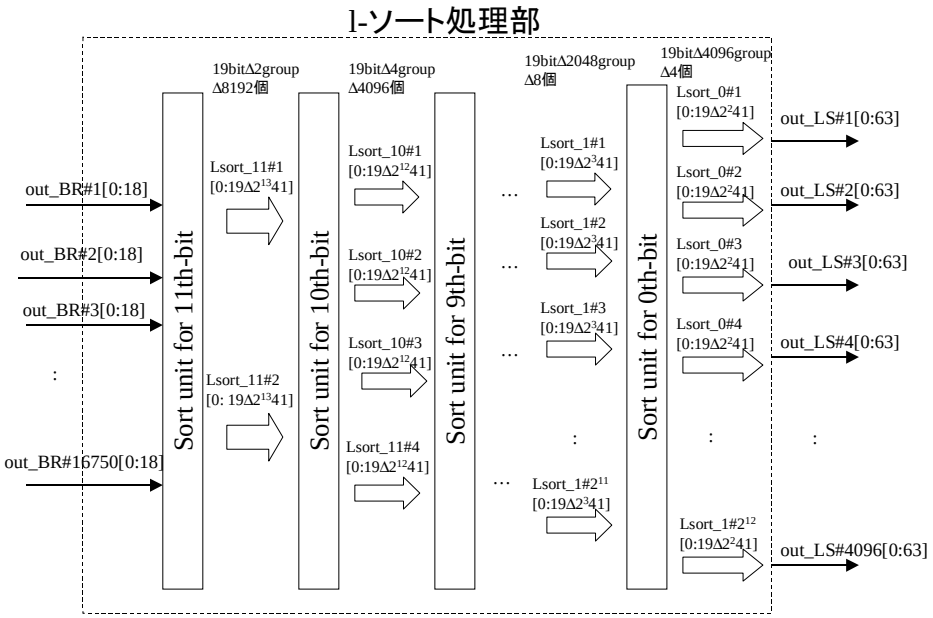


図 3 - 20 l-ソート処理部のブロック図

■インターフェース

名称	I/O	ビット長	説明
out_BR#X[0:18] (X=1~16750)	I	318,250	Bubble Sort/Random Shuffle から入力される, Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i), e_i$ で表される 19bit 値 Δ16,750 ($e_{i,}$=1:Delivery Pair 有り,$e_{i,}$=0: Delivery Pair 無し)
out_LS#X[0:63] (X=1~4096)	O	262,144bit	l_i 値で 4096 通りにグループ化された Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i), e_i$ で表される 16bit 値 Δ4 ($e_{i,}$=1:Delivery Pair 有り,$e_{i,}$=0: Delivery Pair 無し) l_i=X41 のデータは out_LS#X から出力される。

■機能

Bubble Sort/Random Shuffle から入力される、最大 16750 個の Delivery Pair を含むデータに対して、それぞれの Delivery Pair 中の l_i の値(bus line 番号)に応じたソートを行った結果を出力する。出力される Delivery Pair は、4096 通りの l_i のそれぞれを 3bit を削除(最上位ビットと最下位の 2 ビット)した 16bit データ$((\lceil log_2 p_i \rfloor, l_i), e_i$ であり(6bit($\lceil log_2 p_i \rfloor$)+9bit(l_i)+1bit(e_i)=16bit)、同一の l_i について最大 4 重複する(16bitΔ4=64bit)形で行われる。

■サブブロック

- 以下から構成される Sort unit for k-bit (k=11,10, …, 0)
  - 11 ビット値 10-4Δ2 分配処理部Δ183 個
  - k ビット値 8-4Δ2 分配処理部Δ24,393 個(k=11,10,…,0)

■モジュール

なし

■TWIRL 論文との違いについて

- 本ブロックは、提案者が主張するにも関わらず実現法が記されていない、l に関するソート機能に必要なブロックの一つとして、報告者によるデザイン追加を行ったブロックである。これらは、TWIRL 論文が主張する、l_i に関するソート機能を実現するためのブロックである。しかし、l_i がランダムに 4,096 通りに変化する 2,100 個のデータが、1cycle ごと次々にパイプライン的に流れてくるという条件では、相当慎重なデザインを行わない限り回路規模が極めて膨大になると考えられる。

  例えば、バブルソートを用いた典型的なソート処理回路を用いた場合、最低でも 5.6G gates 必要と見積もられる。これは、1 cycle に 1 回ずつ新しいソート対象データ列が流れてくるのを、各パイプラインステージにバブルソート回路を備えた 4,096 段のパイプラインでソートすることを仮定した場合の回路規模である。(バブルソートの場合、ソート終了まで 4,096 サイクル必要)

  この問題を解決すべく、回路規模を押さえつつパイプライン的なソートを実行する回路を報告者が検討した結果が本ブロックの回路デザインである。
  このデザインを用いることで、回路規模を小さく(329M gates)押さえたソート処理を実現する。ただし、本回路は入力データのごく一部を失う可能性がある。これは、入力データの偏りが生じることで、中間データの転送先がデータ領域の特定範囲内に収まらなかった場合、そのデータを破棄するからである。しかしデータを破棄する確率は非常に小さく（本ブロック全体で 0.000737)、問題ない範囲と考えられる。本回路によるソート処理の基本構造とデータ破棄確率の詳細については、3.3.14 節を参照のこと。

■動作概要

Sort unit for k-bit に入力される全ての Delivery Pair($(\lfloor log_2 p_i \rfloor, l_i), e_i$ に対し、l_i の k 番目のビット値に応じて 2 つにグループ分けして出力することを k=11,10,…,0 について 12 回繰り返す(図 3 - 20)。これにより、Delivery Pair が l_i の値に応じて 4096 通りにグループ分けされる。

このグループ分けは、約 16,000 個[11]の Delivery Pair 全てを一つの回路で処理するのではなく、これらの Delivery Pair を小さい個数(8 or 10 個)ずつの単位にわけて、それぞれの単位に対するグループ分け処理を一つの回路で実行することで行う。この単位ごとにグループ分け処理を実行する回路のブロックを、"k-ビット値 M-4Δ2 分類処理部"と呼ぶ(M=8 or 10)。このブロックは、8 個(M=8)もしくは 10 個(M=10)の Delivery Pair を入力として受け取り、それぞれを l_i の k 番目のビット値=0,1 に応じて分類して出力する。分類された Delivery Pair は、ビット値=0,1 それぞれについて最大 4 個出力する。

Sort unit for k-bit で使用される k-ビット値 M-4Δ2 分類処理部の個数を、k=11,10,…,1,0 ごとにまとめた結果を表 3 - 2 に示す。

表 3 - 2 Sort unit for k-bit の回路規模

k	入力される Delivery Pair 数	出力される Delivery Pair 数	k ビット値 8-4Δ2 分類処理部の個数	k ビット値 10-4Δ2 分類処理部の個数
11	16,750	16,384	1865	183
10	16,384	16,384	2048	0
9	16,384	16,384	2048	0
8	16,384	16,384	2048	0
7	16,384	16,384	2048	0
6	16,384	16,384	2048	0
5	16,384	16,384	2048	0
4	16,384	16,384	2048	0
3	16,384	16,384	2048	0
2	16,384	16,384	2048	0
1	16,384	16,384	2048	0
0	16,384	16,384	2048	0

[11] これらのうち、e_i=1 である有効なデータは平均 2,100 個であり、残りの 14,000 個は e_i=0 である空のデータである。

3.3.14. k ビット値 M-4Δ2 分配処理部

■ブロック図

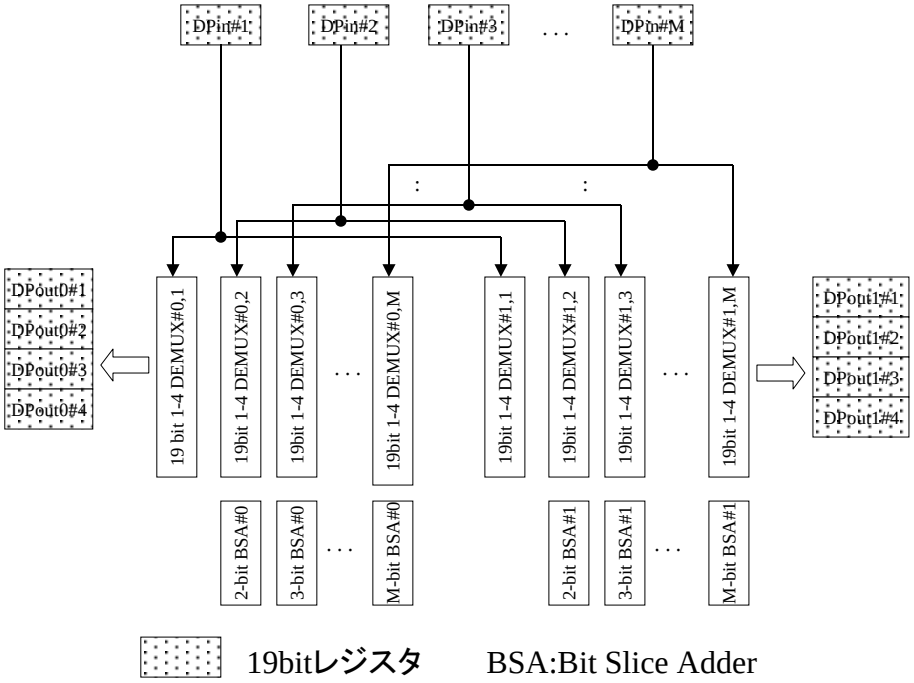


図 3 - 21　k ビット値 M-4Δ2 分類処理部のブロック図

■インターフェース

名称	I/O	ビット長	説明
DPin#X[0:18] (X=1~M, M=8 or 10)	I	19ΔM (M=8or10)	19bit で表現された M 個の Delivery Pair データ $((\lceil log_2 p_i \rceil, l_i), e_i$ (e_i=1:Delivery Pair 有り,e_i=0: Delivery Pair 無し)
DPout0#X[0:18] (X=1~4)	O	76	入力された M 個の 19bit Delivery Pair データ $((\lceil log_2 p_i \rceil, l_i), e_i$ のうち、l_i の k 番目のビット値が 0 であるもの。 (e_i=1:Delivery Pair 有り,e_i=0: Delivery Pair 無し)
DPout1#X[0:18] (X=1~4)	O	76	入力された M 個の 19bit Delivery Pair データ $((\lceil log_2 p_i \rceil, l_i), e_i$ のうち、l_i の k 番目のビット値が 1 であるもの。 (e_i=1:Delivery Pair 有り,e_i=0: Delivery Pair 無し)

■機能

入力された M 個(M=8 or 10)の 19bit Delivery Pair $((\lfloor log_2 p_i \rfloor, l_i), e_i$ を、l_i の k 番目のビット値に応じて 2 種類の 19bit レジスタΔ4 へ出力する。

■サブブロック

なし

■サブユニット

- 2~M Bit Slice Adder(BSA)#0~1
- 19bit registerΔ(M+8)個
- 9-bit 1-4 DEMUX#0,1~M
- 19-bit 1-4 DEMUX#1,1~M

■TWIRL 論文との違いについて

- 本ブロックは、提案者が主張するにも関わらず実現法が記されていない、l に関するソート機能に必要なブロックの一つとして、報告者によるデザイン追加を行ったブロックである。ハードウェアでは典型的であるバブルソートを用いずに、2 分木（のような）ソートを 12 回繰り返す処理を実行する。ただし、8 入力を(k 番目のビット値に応じて)8-8 のデータに分類するソートを繰り返した場合、12 段の処理後の出力レジスタの大きさが入力前の 4,096 倍となり回路規模が大きくなりすぎるため、8 入力を 4-4 に分類するソートとした。このデザインでは、偏った入力データが与えられ出力データが 4 個を超える場合がある。結果、4 個を超えた場合にデータを破棄する必要がある。本ブロックでこのデータ破棄が発生する確率を評価した結果、0.000737 と非常に低い値になった。（これは、入力データが有効である確率がおよそ 1/8 であることによるものである。）これは現実的に問題ないほど低いと考えられる。本ブロックのデータ破棄発生確率の評価式については、次項の動作概要を参照されたい。

■動作概要

動作原理を図 3 - 22 に示す。この動作原理を実現するために、以下の動作を 1 cycle 以内に実行し、M 個のデータを Delivery Pair $((\lfloor log_2 p_i \rfloor, l_i), l_i$ の k 番目のビット値に応じて分類する。なお、Bit Slice Adder(BSA)とは、複数のビット値を並列的に加算する回路である。文献[Kob95][Kob98]によると、n ビット入力, m ビット出力の Bit Slice Adder は、(n4m)個の Full Adder 用いて構成できる。

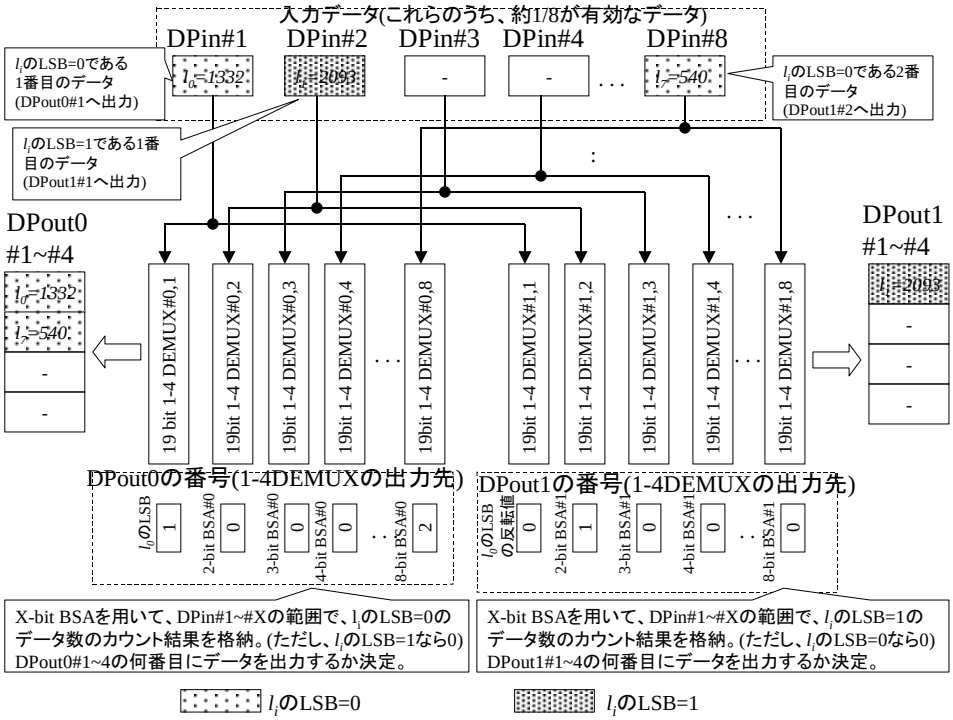


図 3 - 22 k ビット値 M-4Δ2 分類処理部の動作原理(M=8,k=0 の例)

- X-Bit Slice Adder#1 (X=2~10)

  それぞれの X-Bit Slice Adder は、DPin#1~#X における l_i の k 番目のビット値 $B_{k,1},\ldots,B_{k,X}$ に対して、$Y_X = (B_{k,1}+..+B_{k,X}) \Delta B_{k,X}$ を計算する。この計算により与えられる Y_X の意味は以下の通りである。

  Y_X= 0　　:　　DPin#X の k 番目のビット値=0 である。

  Y_X≠1　　:　　DPin#X の k 番目のビット値=1 であり、Y_X には DPin#1~X の範囲で k 番目のビット値=1 であるデータの個数が与えられる。

- X-Bit Slice Adder#0 (X=2~10)

  X Bit slice adder #1 と同様の計算を、DPin#1~#X における l_i の k 番目のビットの反転値 $\sim B_{k,1},\ldots,\sim B_{k,X}$ に対して行う。つまり、$\sim Y_X = (\sim B_{k,1}+..+\sim B_{k,X}) \Delta \sim B_{k,X}$ を計算する。この計算によって与えられる $\sim Y_X$ の意味は以下の通り。

  $\sim Y_X$= 0　　:　　DPin#X の k 番目のビット値=1 である。

  $\sim Y_X$≠1　　:　　DPin#X の k 番目のビット値=0 であり、$\sim Y_X$ には DPin#1~X の範囲で k 番目のビット値=0 であるデータの個数が与えられる。

- 1-4 DEMUX #0.X, #1,X
  1-4 DEMUX#0,X : ~Y_X∅1 なら、DPin#X→DPout0#~Y_X へのレジスタ転送処理を実行。
  1-4 DEMUX#1,X : Y_X∅1 なら、DPin#X→DPout1#Y_X へのレジスタ転送処理を実行

以上に従い~Y_X, Y_X を算出し、これらの値に応じたレジスタ転送を行うことで、他の DPin からのレジスタ転送先が重複することを避けながら、各 DPin#X の k 番目のビット値に応じて DPout0 と DPout1 に分類して出力する処理を実現する。

なお、入力データに偏りがある場合、DPout0 もしくは DPout1 へ転送される対象となるデータの個数が 4 個を超え、データが損失される可能性がある。その確率は、

$$O_{k=5,\ldots,M}\left({}_MC_k \Delta q^k \Delta (1-q)^{M-k}\right)$$

により与えられる。ただし、q は、有効なデータが入力される確率 q'を用いて $q=q'/2$ で与えられる値である。k=11 の場合 q=0.0625 であり、M=8 ならば確率は 0.0000455、M=10 ならば確率は 0.000184 である。k=10,9,…,0 の場合 q=0.064, M=8 から、確率は 0.0000514 である。これらより k=11,10,…0 全体でデータを失う確率は 0.000737 と評価される。この確率は十分小さく、ふるいを行う上で問題ない範囲であると考えられる。

3.3.15. *l*-重複削除処理部

■ブロック図

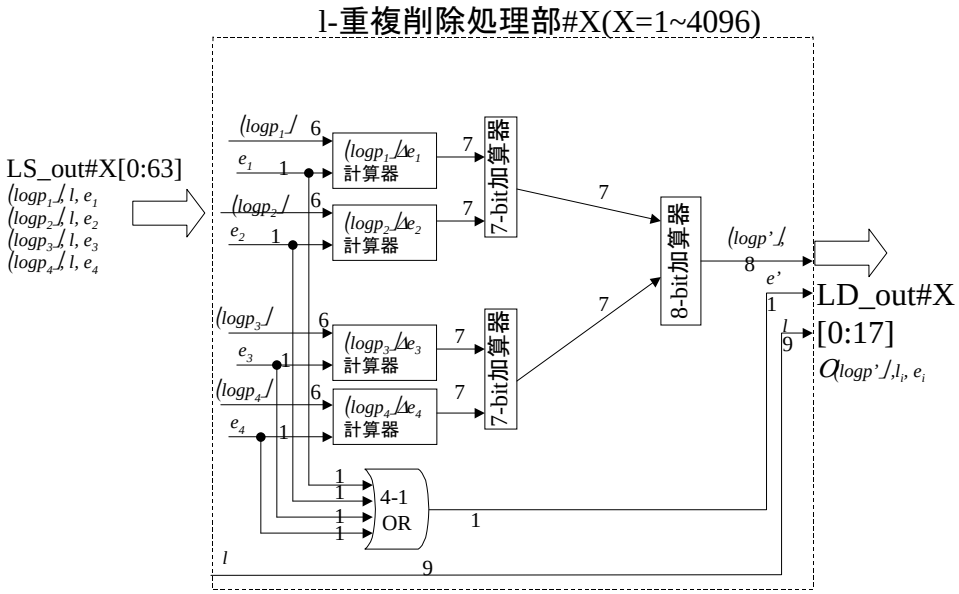


図 3 - 23 l-削除処理部#X(1~4096)のブロック図

■インターフェース

名称	I/O	ビット長	説明
LS_out#X[0:63] (X=1~4096)	I	63	16bit で表現された 4 個の Delivery Pair データ *(⌈log2pi⌋, l, ei,* ただし、*l=X41*。 *(ei*=1:Delivery Pair 有り, *ei*=0: Delivery Pair 無し)
LD_out#X (X=1~4096)	O	18	18bit で表現された 1 個の Delivery Pair データ *(⌈log2p'⌋, l, e'),* ただし、*l=X41*。 *(ei*=1:Delivery Pair 有り, *ei*=0: Delivery Pair 無し)

■サブブロック

なし

■モジュール

- 7-bit 加算器Δ2 個
- 8-bit 加算器Δ1 個

- 1-bit 4-1 OR ゲート×1 個
- 以下から構成される$\lfloor log_2 p_i \rfloor \times e_i$計算器×4 個
  - 6-bit 2-1 and

■機能

l_iの値が同一である複数の Delivery Pair 同士（最大４つ）の$\lfloor log_2 p_i \rfloor$の値を加算することで、一つの Delivery Pair にまとめる。

■TWIRL 論文との違いについて

- 本ブロックは、提案者が主張する機能である、lに関するソート処理機能の一部を実現する。ただし提案者は実現法を示していないため、報告者が実現法検討を行った。本ブロックの前処理であるl-ソート処理部によって、データを 4,096 通りにソートされて出力されるが、この出力データはl_iごとに最大 4 重複している状態である 4,096×4=16,384 個のデータ領域に対し、約 2,100 個の有効なデータが発生していることを考えると、l-ソート処理部の出力はデータ領域全体の約 1/8 のみを使用している疎なデータ列である。このデータ列を密にするのが本ブロックの目的である。

  重複している Delivery Pair 同士は$\lfloor log_2 p_i \rfloor$, l_iの値がともに同一であることを考慮すると、4 個の Delivery Pair の$\lfloor log_2 p_i \rfloor$の値を加算することで 1 つの Delivery Pair にまとめられ、疎なデータ列を密にすることができる。本ブロックの処理により、データ領域の使用率を高め、効率的な処理を実現できる。

■動作概要

以下に従い、4 つの Delivery Pair$(\lfloor log_2 p_1 \rfloor, l, e_1)$, $(\lfloor log_2 p_2 \rfloor, l, e_2)$, $(\lfloor log_2 p_3 \rfloor, l, e_3)$, $(\lfloor log_2 p_4 \rfloor, l, e_4)$(各 16bit)を一つの Delivery pair $(log_2 p', l, e')$ (18bit)にまとめる。 $\lfloor log_2 p_i \rfloor \times e_i$の計算は、$\lfloor log_2 p_i \rfloor \times e_i$計算器が行い、これは$e_i$を 6 並列に並べた値と$\lfloor log_2 p_i \rfloor$の 6 ビット値との and をとることで実現する。

$$log_2 p' = \lfloor log_2 p_1 \rfloor \times e_1 + \lfloor log_2 p_2 \rfloor \times e_2 + \lfloor log_2 p_3 \rfloor \times e_3 + \lfloor log_2 p_4 \rfloor \times e_4$$

$$e' = e_1 \rangle e_2 \rangle e_3 \rangle e_4$$

3.3.16. Delivery line **出力処理部**

■ブロック図

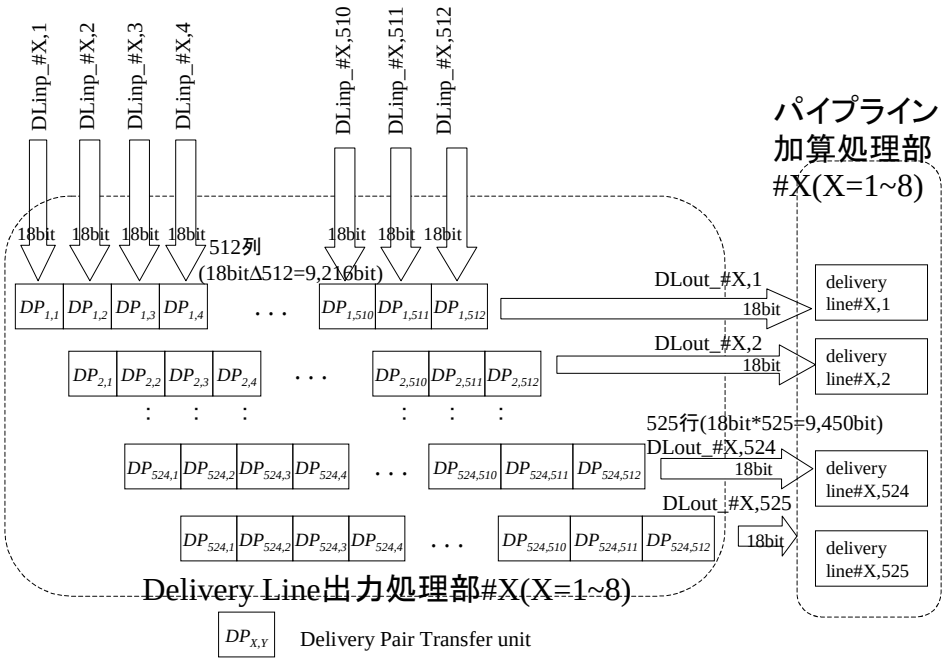


図 3 - 24 Delivery line 出力処理部のブロック図

■インターフェース

名称	I/O	ビット長	説明
DL_inp#X,1~512[0:17] (X=1~8)	I	9,216 (18bitΔ512)	512 個の 18bit Delivery Pair データ($\lfloor log_2 p'_i \rfloor$, l_i, e'_i) (e'_i=1:Delivery Pair 有り, e'_i=0: Delivery Pair 無し)
DL_out#X,1~525[0:17] (X=1~8)	O	9,450 (18bitΔ525)	525 個の 18bit Delivery Pair データ($\lfloor log_2 p'_i \rfloor$, l_i, e'_i), (e'_i=1:Delivery Pair 有り, e'_i=0: Delivery Pair 無し)

(注 1) DL_inp#X,Y と Bus Line 番号 l_i の関係は、X=1+ $(l_i \bmod 4)$, Y=1+$\lfloor l_i/4 \rfloor$(X=1~4 の場合),
X= 1+ $(l_i - 2048) \bmod 4$, Y=1+$\lfloor (l_i - 2048)/4 \rfloor$(X=5~8 の場合)である。

(注 2) *DL_out#X,Y* はパイプライン加算部#X の Delivery Line 番号 Y へ出力

■サブブロック

なし

■モジュール

- Delivery Pair Transter unitΔ268,800 個

■機能

入力された複数の Delivery pair の間で、データを出力する Delivery Line 番号と出力時刻の両方について調整を行った上で、Delivery Pair を出力する。

■TWIRL 論文との違いについて

- 本ブロックは、提案者が主張する、適切なタイミングによる Delivery Line へのデータ出力処理を行う機能を実現する。提案者によって回路実現法が記されてないため、報告者によるデザイン追加を行った。デザインにあたっては、パイプライン加算部の長所（Bus line 間のデータ転送遅延と Delivery Line 間のデータ転送遅延の差分が同一の範囲で、出力タイミングと Delivery Line 番号を選択可能)を生かしながら最適な調整を行える方法を、TWIRL 提案者による Smallish Station の Funnel の実現法をもとに検討した。本デザインの詳細については動作概要を参照されたい。

■動作概要

動作原理を図 3 - 25 に示す。図に示すとおり、Delivery Pair Transfer unit を基本単位としたパイプライン的な構造をもつ。この基本単位を、$DP_{X,Y}$(1ΩXΩ25, 1ΩYΩ12)と記述する。Delivery Pair Transfer unit には 18bit の Delivery Pair のデータが格納されており、それぞれの Delivery Pair Transfer Unit は、1 cycle ごとに以下の処理を実行する。

1. 一つ下($DP_{X,Y}$から見た $DP_{X-1,Y+1}$)の Delivery Pair Transfer unit データが空である場合、そこに Delivery Pait データを転送する。
2. 全ての Delivery Pair Transfer unit の一つ右($DP_{X,Y}$から見た $DP_{X+1,Y}$)に、Deliovery Pair データを転送する。結果、各行の最も右列のデータは Delivery Line に対して出力される。

1.により、Delivery Line 番号と出力タイミングの調整が実現される。1.により調整が可能な理由は、パイプライン加算部における Bus line 間のデータ転送遅延(($\lceil log_2 p_i \rceil$)の転送遅延)と、Delivery Line 間の転送遅延((($\lceil log_2 p_i \rceil$)が加算される *FS(a,b)*の転送遅延)の差分が変化しないことが調整のための条件であり、1.のデータ転送はこの条件を満たしているからである。

1.に示すように、調整可能な範囲でデータをなるべく下方向に移動することで、入力時には疎であるデータ列を圧縮し密なデータ列にして出力することが可能となる[12]。これにより、

[12] 疎な入力データ列を圧縮して密な出力データ列することが可能である以上、Delivery Line の本数を(Shamir-Tromer の回答である)525 本より減らすことができると考えられる(入力の 512 本より多く、明らかに冗長)が、この課題については本章の検討範囲外とする。

複数の Delivery Pair 間で、出力先と出力タイミングの最適な調整を実現できると考えられる。

これらの調整と同時に、2.により全体のデータを一つ右にシフトする処理を行う。この時、最も右の列のデータが Delivery Line に出力される。

以下のメカニズムにより、Delivery Line番号と出力時刻の調整を行う
1. 1 cycleごとに全体を一つ右にシフト(最も右のデータはDelivery Lineへ出力)
2. ただし、一つ下の行のデータが空であれば、データを移動する。
   →パイプライン加算部で発生する、縦方向(Delivery line番号)と横方向(Bus line番号)の2つの遅延時間の差分が、移動前後で同一。

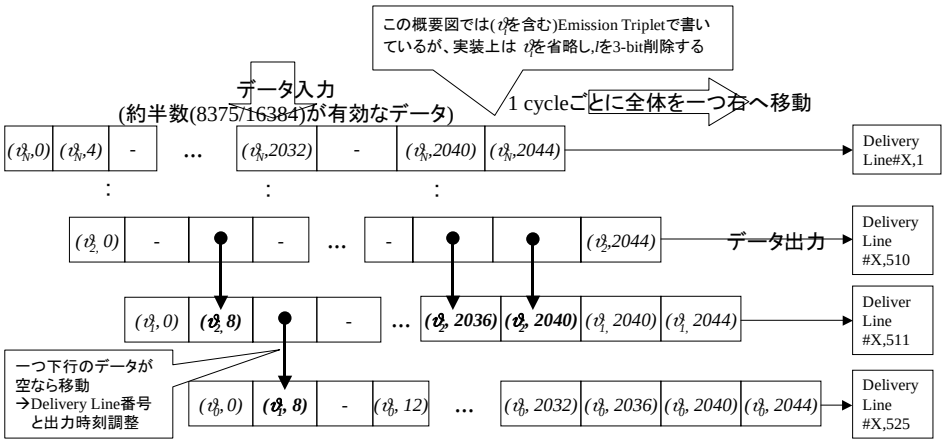


図 3 - 25 Delivery Line 番号と出力時刻調整の原理

以上における 1.2.を 1cycle で同時に処理するための、Delivery Pair Transfer Unit を実現した回路を図 3 - 26 に示す。16-1 AND gate を用いて、下 16 個以内にデータの空きがあるかどうかを検出し、空きがあるならデータを右下に、さもなくば右に転送する。データの空き判定を下 1 個に対してのみ行うのではなく、下 16 個以内全てについて実行することで、図 3 - 26 の下に示す通り複数のデータを同時に移動することが可能となり、効率的な圧縮を実現することができる。

$DP_{X,Y}$の下16個以内に、空であるデータが存在するなら移動
→複数のデータをまとめて移動することが可能

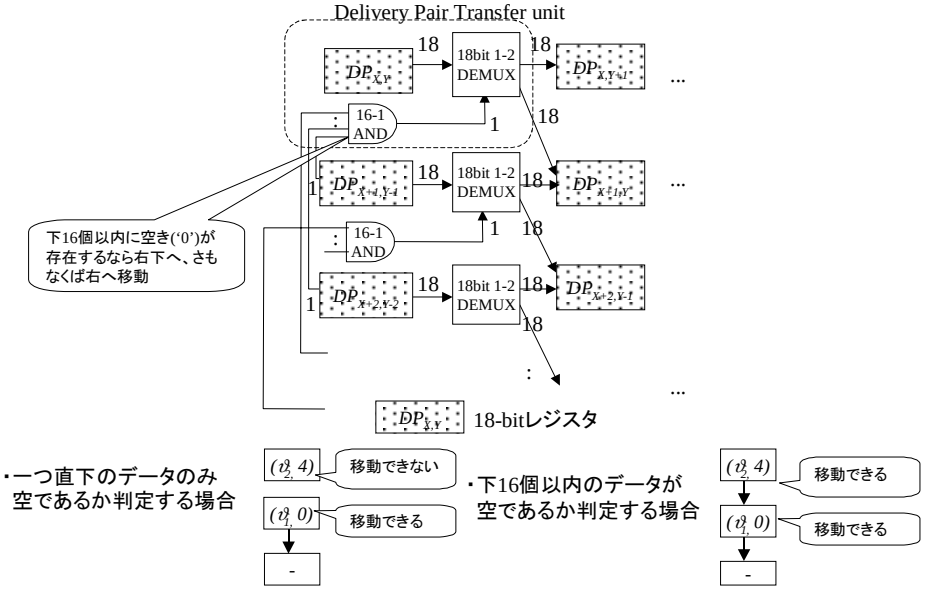


図 3 - 26 Delivery Pair Transfer Unit の回路構成

3.3.17. **パイプライン加算処理部**

■ブロック図

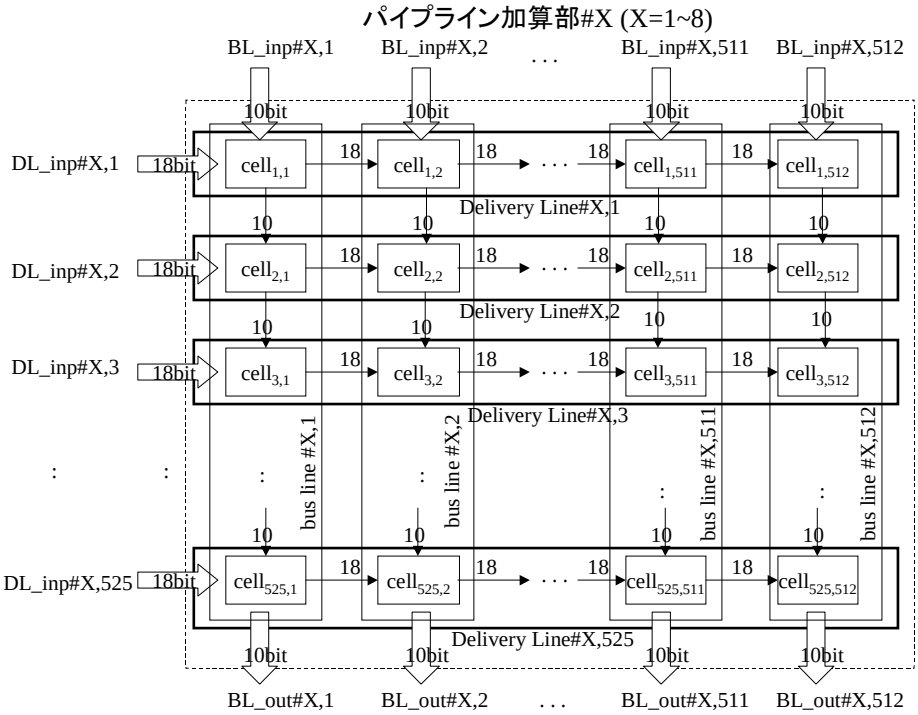


図 3 - 27 パイプライン加算部のブロック図

■インターフェース

名称	I/O	ビット長	説明
BL_inp#X,1~512[0:9] (X=1~8)	I	5,120 (10bitΔ512)	512 個の 10bit データ $FS(a,b)=\lceil log_2 p_i \rceil$
DL_inp#X,1~525[0:17] (X=1~8)	I	9,450 (18bitΔ525)	525 個の 18bit Delivery Pair データ($(\lceil log_2 p_i \rceil, l_i, e_i)$, ($e_i$=1:Delivery Pair 有り, e_i=0: Delivery Pair 無し)
BL_out#X,1~512[0:9] (X=1~8)	O	5,120 (10bitΔ512)	512 個の 10bit データ $FS(a,b)=\lceil log_2 p_i \rceil$

(注 1) BL_inp/out#X,1~512 と Bus Line 番号の対応は、表 3 - 3 に従う。

■サブブロック

なし

■モジュール

- cell×268,800 個

■機能

Buffer より入力された Delivery Pair に含まれる素因数の対数値($\lfloor log_2 p_i \rfloor$)を、素因数情報を蓄積する記憶領域 *FS(a,b)*に対して加算する処理を行う。この加算処理は、内部に行列状に配置されている cell と呼ばれる各ユニットが、それぞれ独立してデータ加算、縦方向のデータ転送、横方向のデータ転送を繰り返すことで行う。

表 3 - 3 パイプライン加算部への入出力と Bus Line 番号の対応

BL_inp/out#X,Y		対応する Bus Line 番号(Y の順)
X	Y	
1	1,2,…,512	0,4,…,2044
2	1,2,…,512	1,5,…,2045
3	1,2,…,512	2,6,…,2046
4	1,2,…,512	3,7,…,2047
5	1,2,…,512	2048,2052,…,4092
6	1,2,…,512	2049,2053,…,4093
7	1,2,…,512	2050,2054,…,4094
8	1,2,…,512	2051,2055,…,4095

■TWIRL 論文との違いについて

- Delivery Line の本数と構成について、報告者による見直しを行った。これは、Delvery Line の本数が論文の内容からは不自然(210 万本)であったと判断したので、Shamir-Tromer に確認した結果、Bus Line を左右 2,048 本ずつにわけ、それぞれに 2,100 本ずつの Delivery Line、合計 4,200 本[13](図 3 - 4)が必要であるとの回答によるものであり、本章のデザインもこれに従う。#

■動作概要

図 3 - 27 に示すように、cell と呼ばれるユニットが 525×512 の行列状に配列された構造を持つ。各 cell を $cell_{x,y}$ と表記する($1 \le x \le 525,\ 1 \le y \le 512$)列方向にならんだ一連の cell を Bus Line

[13] 左右それぞれの 2,100 本の Delivery Line を、Interlevaling(2 章参照)構造を考慮してさらに 4 分割することで、525 本×4×2=4,200 本の Delivery Line とみなすことができる。本章におけるパイプライン加算部#1~#8 は、それぞれこの 525 本の Delivery Line を持つものとしてデザインしている。

と呼び、行方向に並んだ一連の cell を Delivery Line と呼ぶ。パイプライン加算部あたりに Bus line は 512 本、Delivery line は 525 本存在する。本ブロック入出力 BL_inp/out#X,1~512 と 512 本の Bus Line 番号の対応は表 3 - 3 に従う。525 本の Delivery Line は、図 3 - 4 に示した左右 2,100 本の Delivery Line について、Interleaving 構造を考慮しさらに 4 分割したものに対応する。

この構造を用いた加算処理の基本原理を図 3 - 28 に示す。Bus Line 方向には *FS(a,b)*のデータ転送が常に行われており、1 cycle ごとに一つ下の cell へ *FS(a,b)*のデータが転送される。Delivery Line 方向は、Delivery Pair($\lceil log_2 p_i \rfloor, l_i$)のデータ転送が行われている。Delivery Line の最も左の Cell より入力された Delivery Pair は、l_i が目的の Bus Line 番号に到着するまで、1 cycle ごとに一つ右の Cell に転送されることを繰り返す。目的の Bus Line 番号に到着した Delivery Pair は、Cell 内部の加算器で$\lceil log_2 p_i \rfloor$を *FS(a,b)*に加算する処理が行われる。また、Cell は図 3 - 29 に示した回路で実現できる。

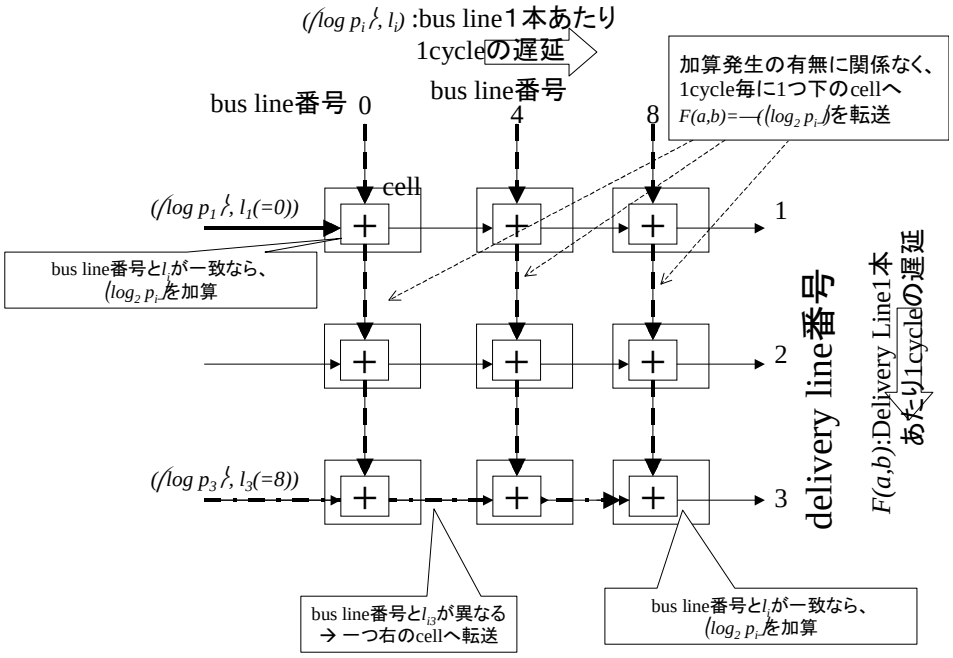


図 3 - 28 パイプライン加算部の基本動作原理

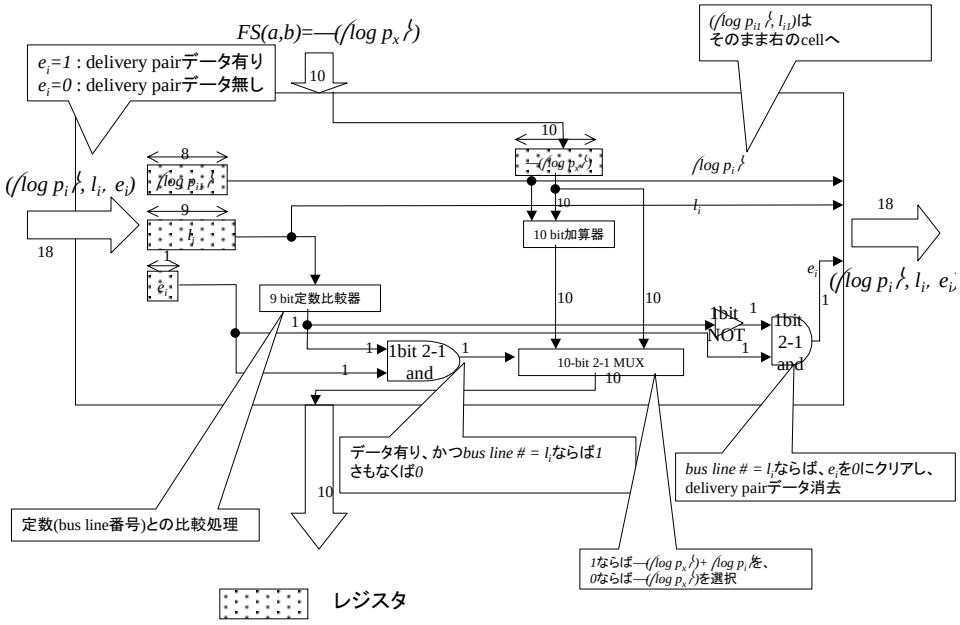


図 3 - 29 Largish Station の cell の回路構成

3.4. Smallish Station

■ブロック図

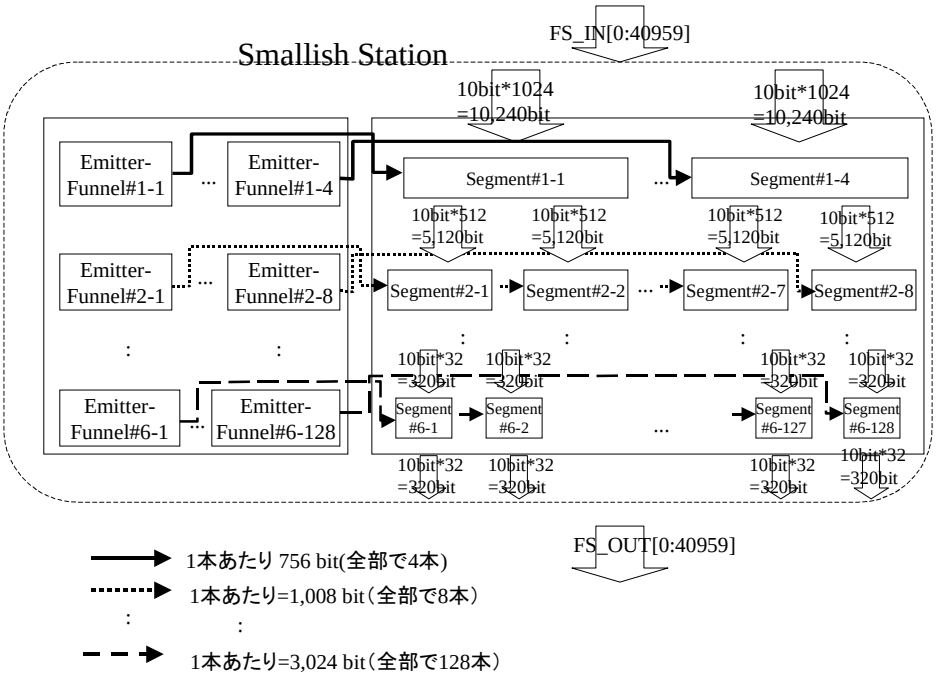


図 3 - 30 Smallish Station のブロック図

■インターフェース

名称	I/O	ビット長	説明
FS_IN[0:40959]	I	40,960	10bit の *FS(a,b)*Δ4,096 個
FS_OUT[0:40959]	O	40,960	10bit の *FS(a,b)*Δ4,096 個

■サブブロック

- Emitter-Funnel#X-Y(X=1,2,…,6, Y=1,2,…,2^{X+1})
- segment#X-Y(X=1,2,…,6, Y=1,2,…,2^{X+1})

■モジュール

なし

■機能

$f(a,b)$が p_i を素因数に持つならば、$FS(a,b)$に$\lfloor log p_i \rfloor$を加算する処理を、$0 \le a \le R$, $0 < b < H$ に含まれる全ての格子点(a,b)について行う。ただし、$256 \le p_i \le 3.2 \times 10^5$ である。

■TWIRL 論文との違いについて

- 基本的な回路構成は提案者の通りである。ただし、p_i の大きさに対応する Segment の数(4,本の Bus Line を G_i 個の Segment に分割した時の G_i の値)については、報告者による見直しを行い、提案者の推奨するより若干大きな G_i の値を設定した。

  提案者の主張する値では G_i の最小値は 2 であるが、この場合 2,048 本の Bus Line を一つの Emitter-Funnel が担当することになる。しかしこれは、Smallish Statiion の狙いの一つである、Emitter あたりの Bus Line 本数が少なくし回路実現を容易にするという目標を実現していないと報告者は考える。また全体の回路規模も G_i に反比例する[14]。これら 2 つの点から、G_i を提案者の値より大きくした方が回路的に有利と報告者は判断し、見直しを行った。G_i を含む見直した結果のパラメータについては、表 3 - 6 に示す。

■動作概要

図 3 - 30 に示す通り、Emitter-Funnel は Semgment に 1 対 1 で接続されている。それぞれの接続において、1cycle に 1 回の割合で以下に示す連携した処理を繰り返すことで、素因数の対数値加算処理を実現する。

- Emitter-Funnel

  Largish Station における Emisison Triplet 発行処理部と Buffer に相当するブロックであり、Delivery Pair$(\lfloor log_2 p_i \rfloor, l_i)$を Segment に発行する処理を実行する。
- Segment

  Largish Statio におけるパイプライン加算部に相当するブロックであり、Emitter-Funnel から受け取った Delivery Pair$(\lfloor log_2 p_i \rfloor, l_i)$のうち、$l_i$ の値と一致する bus line 番号にて$\lfloor log_2 p_i \rfloor$の加算処理を行う。

Emitter-Funnel と Segment の IO 長は、表 3 - 4 に従う。Y の最大値は Segment 数 G_i であり、また Segment あたりに含まれる Bus Line 数η_iは G_i より $\eta_i = 4096/G_i$ で与えられる・

[14] 例えば G_i を 2 倍すると、Funnel の数が半減し、回路規模が減ったように見えるが、代わりに Funnel への入力列の長さが 2 倍になる。Funnel の回路規模は入力列の 2 乗に比例するため 4 倍となり、トータルでは 2 倍の回路規模増となる。

表 3 - 4 Emitter-Funnel と Segment の IO 長

X	Y (1~G_i)	Emitter-Funnel から Segment への IO 長	Emitter-Funnel の 個数	合計 IO 長
1	1~4	672 bit	4	2,688 bit
2	1~8	896 bit	8	7,168 bit
3	1~16	1,344 bit	16	21,504 bit
4	1~32	2,464bit	32	78,848 bit
5	1~64	4,256 bit	64	272,384 bit
6	1~128	2,688 bit	128	344,064 bit

3.4.1. Emitter-Funnel

■ブロック図

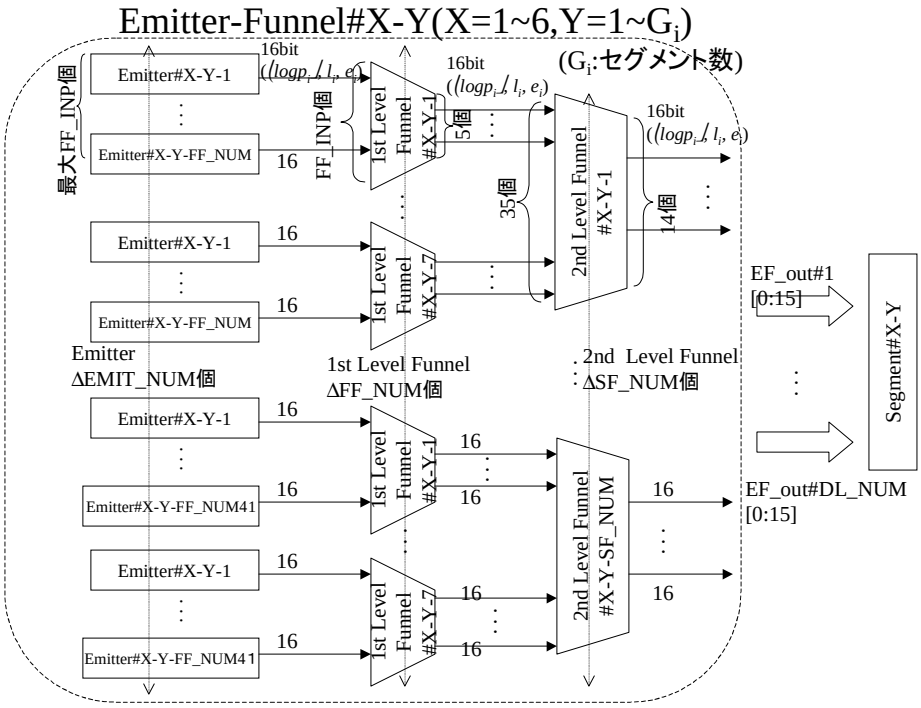


図 3 - 31 Emitter-Funnel のブロック図

■インターフェース

名称	I/O	ビット長	説明
EF_out#1~DL_NUM [0:15]	O	16ΔDL_NUM	16bit の Delivery Pair($(log_2 p_i, l_i)$, e_i ($e_i=0$:Delivery Pair なし, $e_i=1$:Delivery Pair あり)

(注 1) EMIT_NUM, FF_INPNUM, FF_NUM, SF_NUM, DL_NUM については表 3 - 5,表 3 - 6 を参照。

■サブブロック

- Emitter#X-Y-*, EMIT_NUM 個
  (X=1,…,6, Y=1,…,G_i, G_i:セグメント数)
- 1st Level Funnel#X-Y-*,　FF_NUM 個
  (X=1,…,6, Y=1,…,G_i, G_i:セグメント数)

- 2nd Level Funnel#X-Y-*, SF_NUM 個
  (X=1,…,6, Y=1,…,G_i, G_i:セグメント数)

■モジュール

なし

■機能

接続された Segment に対して 16bit の Delivery Pair($\lceil log_2 p_i \rfloor$, l_i), e_i を発行する。1cycle あたり、最大 DL_NUM 個のデータを並列に出力する。

■TWIRL 論文との違いについて

- 基本的な回路構成は提案者の通りである。Segment の数を表すパラメータ G_i およびη_i について報告者による見直しを行い、提案者の推奨するより若干大きな G_i(小さなη_i)に設定した。見直した理由は Smallish Station の章の最初において述べた通りである。これらを基準に他の回路規模パラメータである EMIT_NUM, FF_INPNUM, FF_NUM, SF_NUM, DL_NUM を設定した。これらの回路規模パラメータの名称と意味については、表 3 - 5 を参照されたい。また見直した結果であるこれらの回路規模パラメータの値については表 3 - 6 を参照されたい。

■動作概要

Emitter, 1st Level Funnel, 2nd-Level-Funnel の間で連携した処理を、1 cycles に 1 回繰り返すことで実行する。

- Emitter
  Delivery Pair($\lceil log_2 p_i \rfloor$, l_i), e_i(16bit)を発行する処理を行う。最大 FF_INPNUM 個の Emitter が一つの 1st Level Funnel に並列に接続されている。

- 1st Level Funnel
  最大 FF_INPNUM の並列接続された Emitter から受け取った Delivery Pair($\lceil log_2 p_i \rfloor$, l_i) , e_i（データ有り／無し両方）のうち、有効なデータを選別し 5 並列にデータ圧縮した形で出力する。

- 2nd Level Funnel
  7 個の並列接続された 1st-Level Funnel から受け取った 35 個の Delivery Pair($\lceil log_2 p_i \rfloor$, l_i) , e_i（データ有り／無し両方）のうち、有効なデータを選別し 14 並列にデータ圧縮した形で出力する。

表 3 - 5 Smallish Station の回路規模パラメータ名称と意味

パラメータ名	意味
G_i	Smallish Station において、4,096 本の Bus Line を複数に分割した単位を Segment と呼ぶ。 G_i は 4,096 本を分割した結果の Segment 数を表す。
η_i	一つの Segment に含まれる Bus Line 本数=$2^{\eta i}$ を決定するパラメータ
EMIT_NUM	一つの Segment に含まれる Emitter の個数。
FF_INPNUM	一つの Segment に含まれる 1st-Level Funnel の入力数。
FF_NUM	一つの Segment に含まれる 1st-Level Funnel の個数。
SF_NUM	一つの Segment に含まれる 2nd-Level Funnel の個数。
DL_NUM	一つの Segment に含まれる Delivery Line の本数。 Segment 内の Delivery Line については図 3 - 39 を参照。

表 3 - 6 Smallish Station の回路規模パラメータ

X	1	2	3	4	5	6
η_i	10	9	8	7	6	5
Segment 数 G_i (Y=1~G_i)	4	8	16	32	64	128
Segment に含まれる Bus Line 数 (=$2^{\eta i}$)	1,024	512	256	128	64	32
Segment ごとの Emitter 数 EMIT_NUM (p_i の範囲に含まれる素数の個数に一致。)	20,061	16,458	4,642	1,336	392	118
1st-Level-Funnel の入力数 FF_INPNUM	256	128	64	32	16	8
Segment ごとの 1st-Level-Funnel 数 FF_NUM (=⌈EMIT_NUM/FF_INP_NUM⌋)	79	129	73	42	25	15
Segment ごとの 2nd-Level-Funnel 数 SF_NUM (=⌈FF_NUM/7⌋)	12	19	11	6	4	3
Delivery Line 本数 DL_NUM (=SF_NUMΔ14)	168	266	154	84	56	42
Emitter の総数 (=EMIT_NUMΔG_i)	80,244	131,664	74,272	42,752	25,088	15,104
1st-Level-Funnel の総数 (=FF_NUMΔG_i)	316	1,032	1,168	1,344	1,600	1,920
2nd-Level-Funnel の総数 (=SF_NUMΔG_i)	48	152	176	192	256	384
p_i の範囲	$2^{18}<p_i<5.2\Delta10^5$	$2^{16}<p_i<2^{18}$	$2^{14}<p_i<2^{16}$	$2^{12}<p_i<2^{14}$	$2^{10}<p_i<2^{12}$	$2^8<p_i<2^{10}$

3.4.2. Emitter

■ブロック図

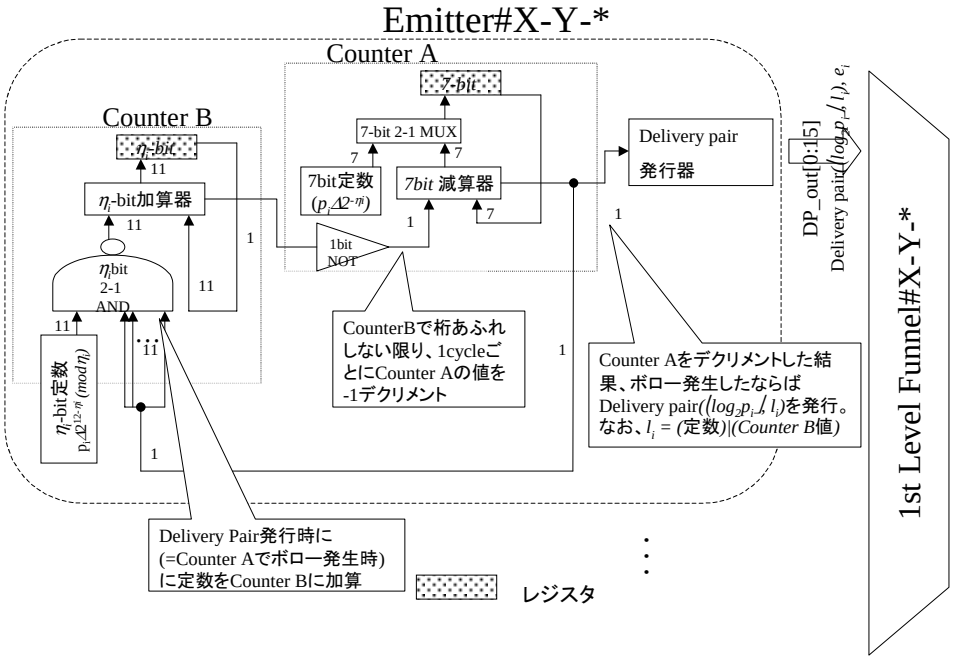


図 3 - 32 Emitter のブロック図

■インターフェース

名称	I/O	ビット長	説明
DP_out [0:15]	O	16	16bit の Delivery Pair($\lceil log_2 p_i \rceil$, l_i), e_i (e_i=0 :Delivery Pair なし, e_i=1 :Delivery Pair あり)

■サブブロック

なし

■モジュール

■機能

Emitter が担当する 1 つの素数 p_i に関する周期情報の管理を行うことで、Segment に対し

Delivery Pair を発行する処理を行う。

■TWIRL 論文との違いについて

特に無し

■動作概要

TWIRL 提案者による、"Count-down method"をそのまま実装している。この処理においては、CounterA, CounterB, Delivery Pair 発行器が以下のように連携した処理を実行することで、Pi の周期情報を管理し、Delivery Pair$((\lceil log_2 p_i \rfloor, l_i), e_i$ を発行する。

- Counter A

  7bit のカウンタであり、基本的に 1 cycle ごとに CounterA の値を-1 デクリメントする処理を行う。ただし、(1cycles 前における)CounterB での加算処理の結果、桁あふれが生じた場合は、デクリメントを 1cycle の間停止する。デクリメントの結果ボローが発生したら、初期値に戻す。この初期値は、$\lfloor p_i \Delta 2^{-\eta_i} \ (mod\ 2^{\eta_i}) \rfloor$により与えられる。

- Counter B

  η_i -bit のカウンタであり、Counter A でボローが発生した場合に定数を加算する処理を行う。この定数は、$2^{\zeta-\eta_i} \Delta p_i \ \ (mod\ 2^{\eta_i}), (\zeta=12)$により与えられる。

- Delivery Pair 発行器

  CounterA のデクリメントの結果ボローが発生した場合、Delivery Pair を発行する。
  発行する Delivery Pair は(定数)|(CounterB 値)で与えられる。この定数は、Emitter#X-Y-* の X,Y から、(定数)=$2^{X+1}\Delta$(Y41)により与えられる。

3.4.3.1st-Level/2nd -Level Funnel(N-M Funnel)

■ブロック図

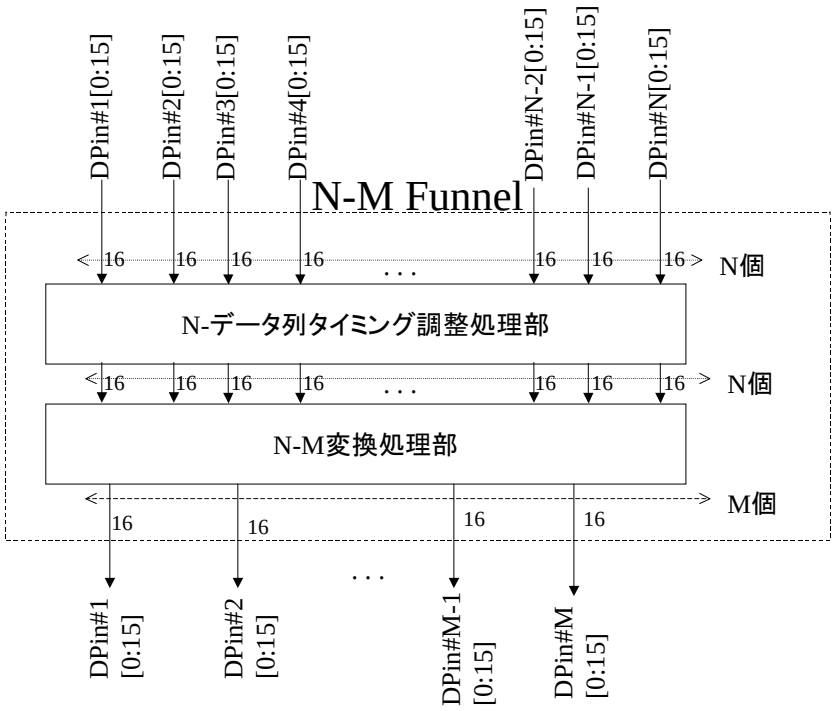


図 3 - 33 N-M Funnel のブロック図

■インターフェース

名称	I/O	ビット長	説明
DP_out#X [0:15] (X=1~N)	I	16	16bit の Delivery Pair($(\lceil log_2 p_i \rceil, l_i), e_i$ (e_i=0 :Delivery Pair なし, e_i=1 :Delivery Pair あり)
DP_in#X [0:15] (X=1~M)	O	16	16bit の Delivery Pair($(\lceil log_2 p_i \rceil, l_i), e_i$ (e_i=0 :Delivery Pair なし, e_i=1 :Delivery Pair あり)

■サブブロック

- N-データ列タイミング調整処理部(1stLevel/2nd-Level Funnel で同一)

- N-M 変換処理部(1stLevel/2nd-Level Funnel で異なる)

■モジュール

なし

■機能

1st/2nd-Level Funnel の機能を実現する。これらに共通することは、有効もしくは無効なデータである N 個の Delivery Pair を並列に入力することで、有効なデータを選別し M 個のデータ列に圧縮したデータを出力する処理を行う。この N 個→M 個の圧縮処理を行うブロックを N-M Funnel と呼び、本章は N-M Funnel の機能仕様を与える。

■TWIRL 論文との違いについて

- 基本的な回路構成は提案者の通りである。Segment の数を表すパラメータ G_i およびη_l に関する報告者による見直しに伴い、1st-Funnel の入力列数の見直しを行った。見直した理由は Smallish Station の章の最初に述べた通りである。見直した結果のパラメータについては、表 3 - 6 を参照。

■動作概要

以下の動作を 1cycle 周期で繰り返す。

- N-データ列タイミング調整処理部
  並列入力された N 列のデータ入力(DPin#1~DP#N)に対し、左から t 番目(DPin#t)のデータに対して、データが入力されてから t cycles の間溜め込む処理を t=1~N 全てに行う。
  1st/2nd-Level Funnel で同一の構造をもつ。
- N-M 変換処理部
  N 個の入力データを M 個に圧縮するメイン処理を実行、M 個のデータを出力する。
  1st/2nd-Level Funnel で異なる構造をもつ。

3.4.4. N-データ列タイミング調整処理部

■ブロック図

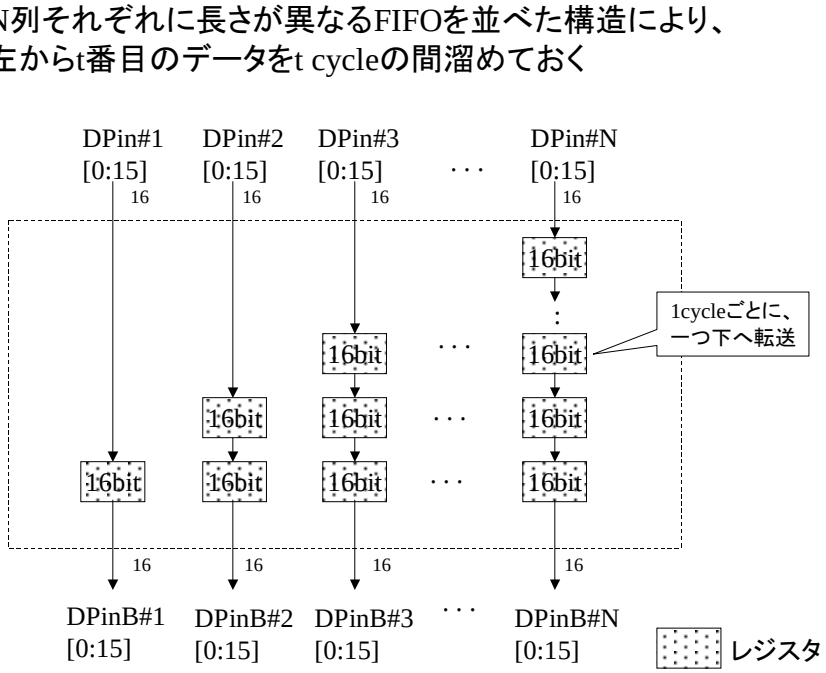


図 3 - 34 N-データ列タイミング調整処理部のブロック図(レジスタによる実装)

■インターフェース

名称	I/O	ビット長	説明
DP_in#X [0:15] (X=1~N)	I	16	16bit の Delivery Pair($(\lceil log_2 p_i \rceil, l_i), e_i$ (e_i=0 :Delivery Pair なし, e_i=1 :Delivery Pair あり)
DP_inB#X [0:15] (X=1~M)	O	16	16bit の Delivery Pair($(\lceil log_2 p_i \rceil, l_i), e_i$ (e_i=0 :Delivery Pair なし, e_i=1 :Delivery Pair あり)

■機能

入力された N 列の Delivery Pair データの各列に対し、*t* 番目の列のデータが *t* cycles 後に出力されるように一時的にデータを溜め込んでおく処理を行う。

■TWIRL 論文との違いについて

TWIRL 論文と同一のデザインであるが､本デザインで用いられる FIFO の構造については、図 3 - 35 に示したような工夫が必要と考えられる。

■動作概要

図 3 - 34 に示す通り、N 列それぞれに長さの異なる FIFO を並べた構造により処理する。t 列めの FIFO は長さ t であり、各 FIFO は 1cycle に 1 回データを 1 つ下方向にシフトする。これにより、入力されてからデータが出力されるまで t cycles を要する。なお、この構造では、N の 2 乗に比例したレジスタ数を必要とし、N=256(FF_INPNUM の最大値)の場合非常に巨大な回路規模[15]となる。図 3 - 34 の代わりに、図 3 - 35 に示した形の回路を用いることで、レジスタを SRAM に置き換えて回路規模を縮小することができる。3.6.1 に示した回路規模見積もりは、図 3 - 35 の構成を用いた場合を前提として算出している。

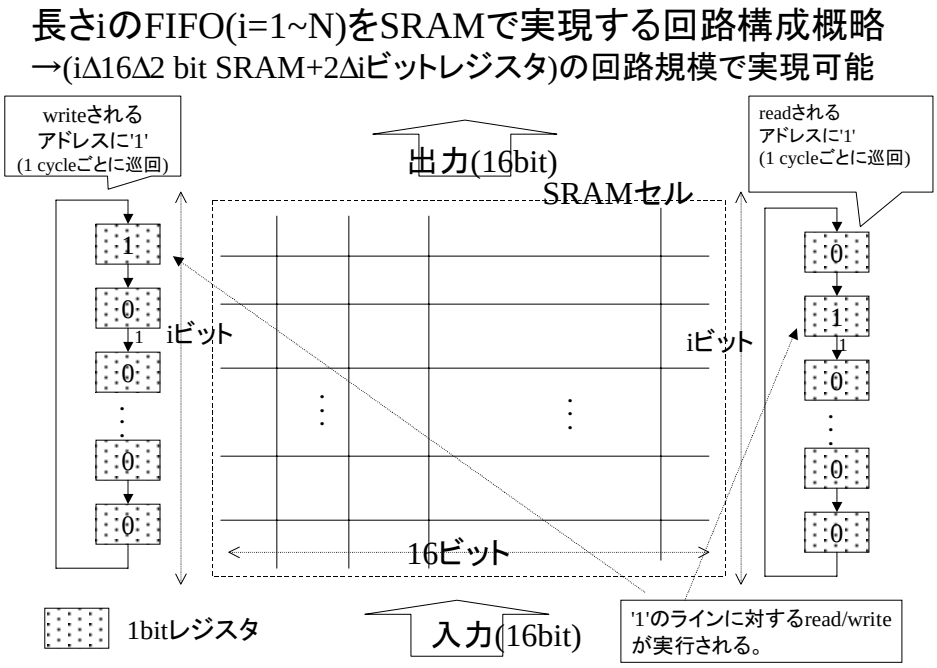


図 3 - 35 N-データ列タイミング調整処理部のブロック図(SRAM による実装)

[15] Smallish Station 全体で 6.52G gates 必要と換算される。

3.4.5. N-M 変換処理部

■ブロック図

台形(上辺N個, 下辺(N*4*M)個, 高さM個)でDelivery pairデータを保持
*N*個のDelivery pair 列を、*M*個のDelivery pair列に圧縮して出力する

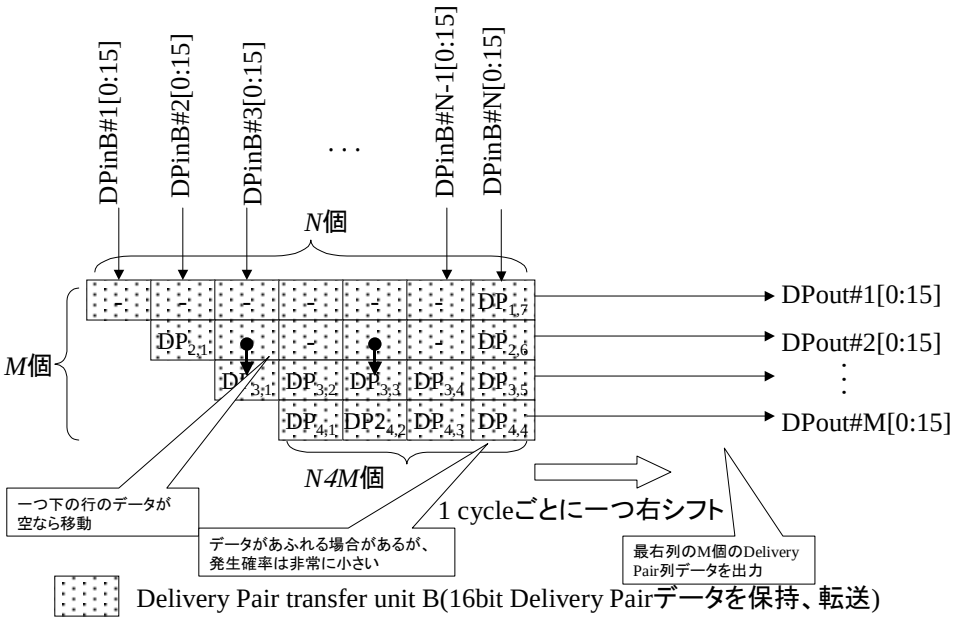


図 3 - 36 N-M 変換処理部(1st-Level Funnel)

平行四辺形(上辺,下辺N個, 高さM個)の形でDelivery pairデータを保持
*N*個のDelivery pair 列を、*M*個のDelivery pair列に圧縮して出力する

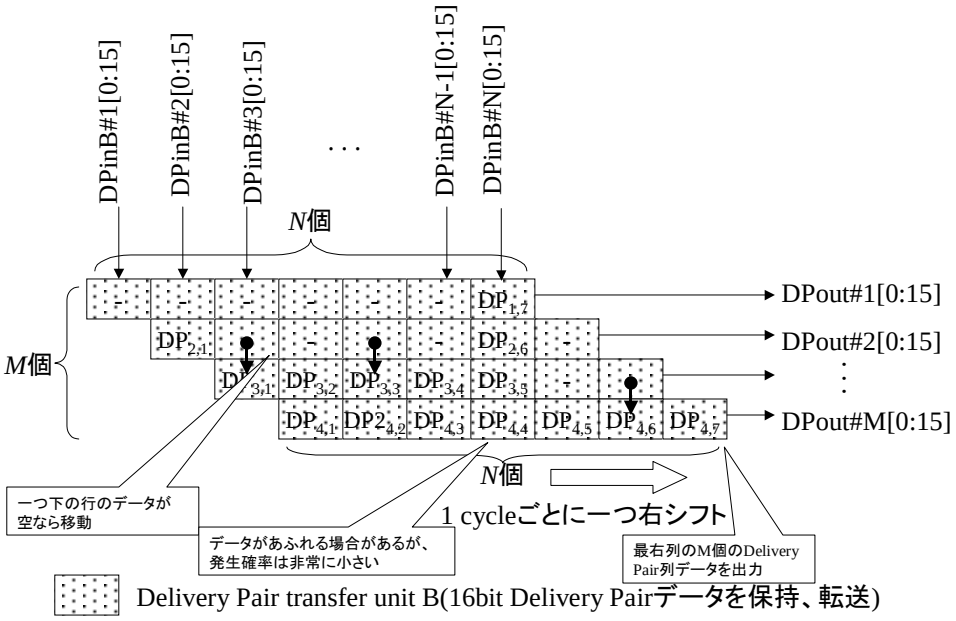


図 3 - 37 N-M 変換処理部(2nd-Level Funnel)

■インターフェース

名称	I/O	ビット長	説明
DP_inB#X [0:15] (X=1~N)	I	16	16bit の Delivery Pair($\lceil log_2 p_i \rfloor, l_i$), e_i ($e_i=0$:Delivery Pair なし, $e_i=1$:Delivery Pair あり)
DP_out#X [0:15] (X=1~M)	O	16	16bit の Delivery Pair($\lceil log_2 p_i \rfloor, l_i$), e_i ($e_i=0$:Delivery Pair なし, $e_i=1$:Delivery Pair あり)

■サブブロック

なし

■モジュール

- 以下を含む Delivery Pair Transfer unitBΔNΔM 個
  - 16-bit レジスタ
  - 16-bit 1-2 DEMUX
  - 1bit X-1 and (X=2,3,...,M, M=14 or 5)

■機能

入力された N 列の有効／無効な Delivery Pair データ列から、有効なデータを選別し M 列に変換する処理を行う。

■TWIRL 論文との違いについて

特に無し

■動作概要

Delivery Pair データを２次元的に配置する。配置の仕方は 1st と 2nd で異なり、1st-Level Funnel においては図 3 - 36,に示す通り、上辺 N 個、下辺 N-M 個、高さ M 個の台形に、2nd-Level-Funnel においては、図 3 - 37 に示す通り、上辺 N 個、下辺 N 個、高さ M 個の平行四辺形に Delivery Pair データを配置する[16]。

行列状に配置されたデータの、X 行,Y 列にある Delivery Pair データを $DP_{X,Y}(1 \le X \le M, 1 \le Y \le N)$と記す。各 Delivery Pair データに対して、以下を 1cycle に 1 回繰り返す処理を実行する。

1. 自分より一つ下($DP_{X,Y}$ に対する $DP_{X+1,Y-1}$)の Delivery Pair データが空($e_i=0$)ならば、そこに移動する。
2. すべての Delivery データ全体を一つ右へ移動する。このとき、最も右列のデータが出力される。

上記 1.2.を 1cycle で実行するために、Delivery Pair Transfer unit B というモジュールを用いる。、Delivery Pair Transfer unit B の回路構成を図 3 - 38 に示す。これは、図 3 - 26 の Delivery Pair Tarnsfer Unit とほぼ同様の処理を実行する。つまり、下 M 個以内に空きががあるかどうかを調べ、空きがあるならばデータを右下へ、さもなくば一つ右へデータを転送する処理を繰り返す。Funnel においては、M=5 or 14 と小さいため、下 M 個全ての空き状況を調べる回路は容易に実現できる。

[16] 2nd-Level Funnel の出力先である Delivery Line 間の転送遅延を考慮すると、2nd-Level Funnel は台形の構造をとる必要がある。

下M個以内に、空であるデータが存在するなら移動

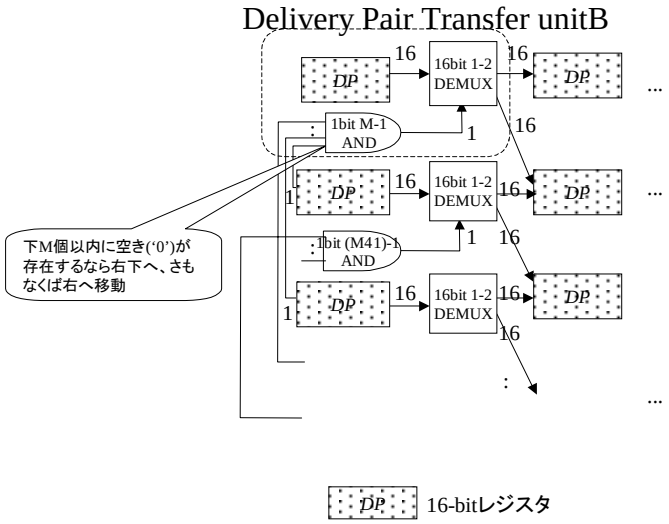


図 3 - 38 Delivery Pair Transfer Unit B の回路構成図

3.4.6. Segment

■ブロック図

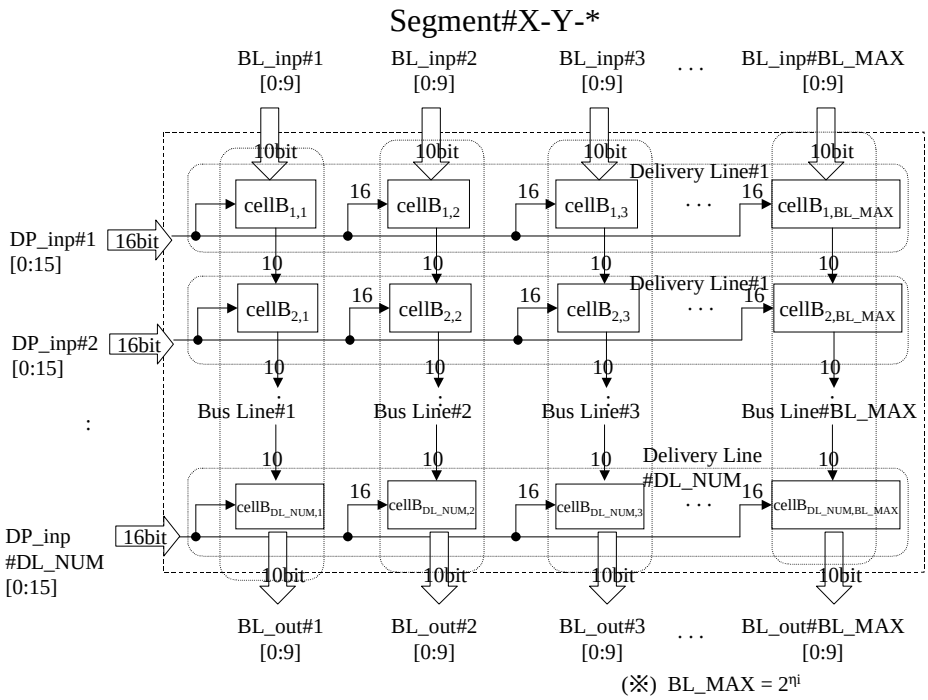


図 3 - 39 Segment のブロック図

■インターフェース

名称	I/O	ビット長	説明
BL_inp#X [0:9] (X=1~$2^{\eta i}$)	I	$10\Delta 2^{\eta i}$	$2^{\eta i}$ 個の 10bit データ $FS(a,b)=O(\log_2 p_i)$
DL_inp#X[0:17] (X=1~DL_NUM)	I	16ΔDL_NUM	DL_NUM 個の 16bit Delivery Pair データ $((\log_2 p_i), l_i, e_i)$, (e_i=1:Delivery Pair 有り, e_i=0: Delivery Pair 無し)
BL_out#X [0:9] (X=1~$2^{\eta i}$)	O	$10\Delta 2^{\eta i}$	$2^{\eta i}$ 個の 10bit データ $FS(a,b)=O(\log_2 p_i)$

(注 1)

DL_NUM は、Delivery Line の本数を表す定数。値については表 3 - 6 を参照。

■サブブロック

なし

■モジュール

- cellB$\Delta 2^{\eta i}$ ΔDL_NUM 個

■機能

Emitter より入力された Delivery Pair に含まれる素因数の対数値$(log_2 p_i)$を、素因数情報を蓄積する記憶領域 *FS(a,b)*に対して加算する処理を行う。この加算処理は、内部に列状に配置されている cellB と呼ばれる各ユニットが、それぞれ独立してデータ加算、縦方向のデータ転送を繰り返すことで行う。Largish Station の Cell と異なり、横方向のデータ転送は行われない。

■TWIRL 論文との違いについて

基本的な回路構成は提案者の通りである。Segment の数を表すパラメータ G_i およびη_iについて報告者による見直しを行い、提案者の推奨するより若干大きな G_i(小さなη_i)に設定した。見直した理由は 3.4 に述べた通りである。G_i を含む見直した結果のパラメータについては、表 3 - 6 に示す。

■動作概要

図 3 - 39 に示すように、cellB と呼ばれるモジュールが行列状に配列された構造を持つ。各 cell を $cellB_{x,y}$ と表記する(1≦x≦DL_NUM, 1≦y≦$2^{\eta i}$)。列方向にならんだ一連の cellB を Bus Line と呼ぶ。行方向に並んだ一連の cellB を、便宜的に Delivery Line と呼ぶことにする。("便宜的"というのは、Largish Station と異なり、Delivery Line に沿った横方向のデータ転送は行われないからである。)

この構造を用いた加算処理の基本原理を図 3 - 40 に示す。Bus Line 方向には *FS(a,b)*のデータ転送が常に行われており、1 cycle ごとに一つ下の cellB へ *FS(a,b)*のデータが転送される。

Emitter から入力された Delivery Pair$((log_2 p_i), l_i)$は、Delivery Line に含まれる全ての CellB に対して転送が行われる。各 CellB においては、受け取った Delivery Pair$((log_2 p_i), l_i)$の l_i と自身に含まれる bus line 番号を比較し、一致した場合に$(log_2 p_i)$を加算する処理を行う。

また、cellB は図 3 - 41 に示した回路で実現できる。これは図 3 - 29 に示した、Largish Station の cell の回路から、横方向へ転送する機能を削除したものである。

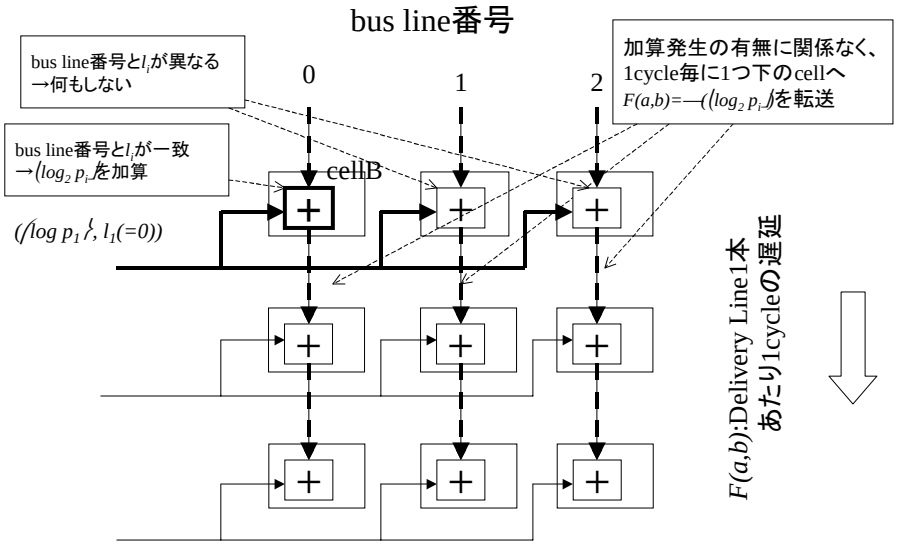


図 3 - 40 Smallish におけるパイプライン加算部の原理

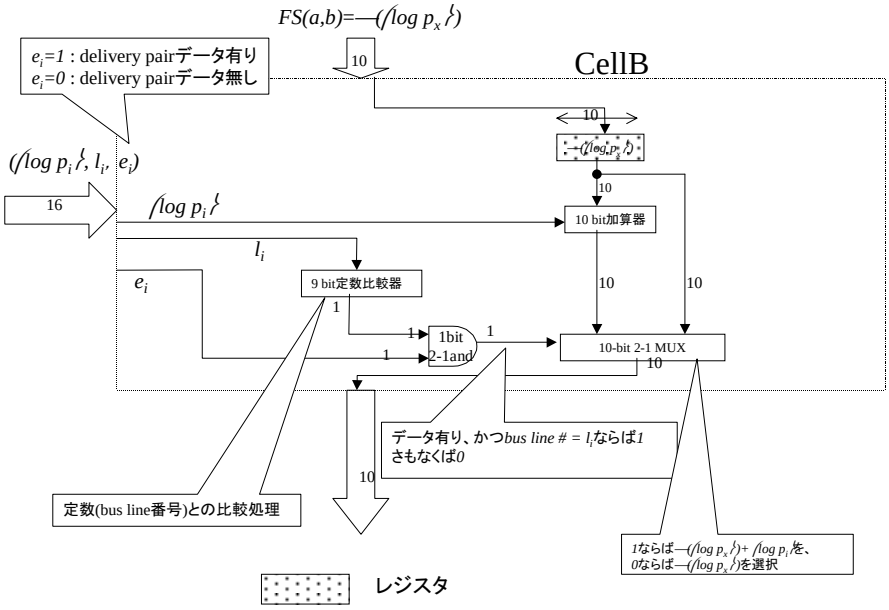


図 3 - 41 Smallish Station における CellB

3.5. Tiny Station

■ブロック図

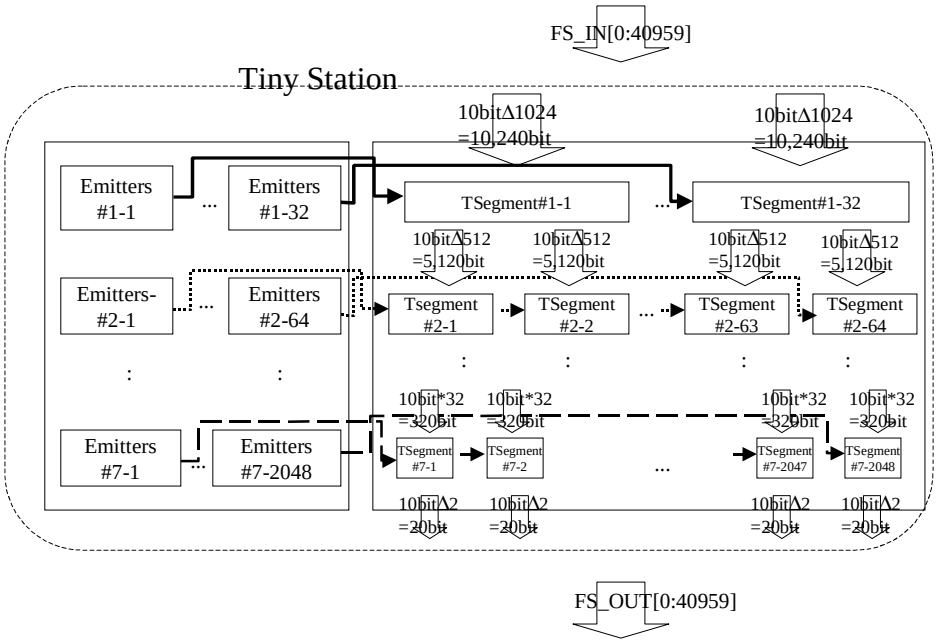


図 3 - 42 Tiny Station のブロック図

■インターフェース

名称	I/O	ビット長	説明
FS_IN[0:40959]	I	40,960	10bit の *FS(a,b)*Δ4,096 個
FS_OUT[0:40959]	O	40,960	10bit の *FS(a,b)*Δ4,096 個

■サブブロック

- Emitters#X-Y(X=1,2,…,7, Y=1,2,…,2^{X+4})
- TSegment#X-Y(X=1,2,…,7, Y=1,2,…,2^{X+4})

■モジュール

なし

■機能

*f(a,b)*が p_i を素因数に持つならば、*FS(a,b)*に$\lfloor log p_i \rfloor$を加算する処理を、*0ΩaΩR, 0<b<H* に含ま

れる全ての格子点(a,b)について行う。ただし、$2\Omega_i<256$ である。

■TWIRL 論文との違いについて

特になし

■動作概要

Emitters と TSegment を 1 対 1 で接続することで実現する。Smallish Station と異なり、Funnel を用いない

- Emitters
  Delivery Pair$(\lceil log_2 p_i \rfloor, l_i)$を Segment に発行する。
- TSegment
  TSegment 内に含まれる全ての bus line 番号のうち、Emitters から受け取った Delivery Pair$(\lceil log_2 p_i \rfloor, l_i)$のうち、$l_i$ の値と一致する bus line にて$\lceil log_2 p_i \rfloor$の加算処理を行う。

Emitters と TSegment の IO 長は、表 3 - 7 に従う。

表 3 - 7 Emitters と TSegment の IO 長

X	Y	Emitters から TSegment への IO 長	Emitters の個数	合計 IO 長
1	1~32	368 bit	32	10,304 bit
2	1~64	208 bit	64	13,312 bit
3	1~128	112bit	128	14,336 bit
4	1~256	80 bit	256	20,480 bit
5	1~512	32 bit	512	16,384bit
6	1~1024	32 bit	1024	32,768 bit
7	1~2048	32 bit	2048	65,536 bit

Emitters-TSegment の構造を図 3 - 43 に示す。基本的に、Smallish Station における Emitter と Segment を直結したものを並べただけである。DL_NUM 個の Emitter が、TSegment それぞれの Delivery Line に入力されている。TSegment は、基本的には図 3 - 40 に示した Segment と同一の構造であり、含まれる Bus Line 数と Delivery Line 数が異なるだけである。X,Y ごとのこれらの値を表 3 - 8 に示す。

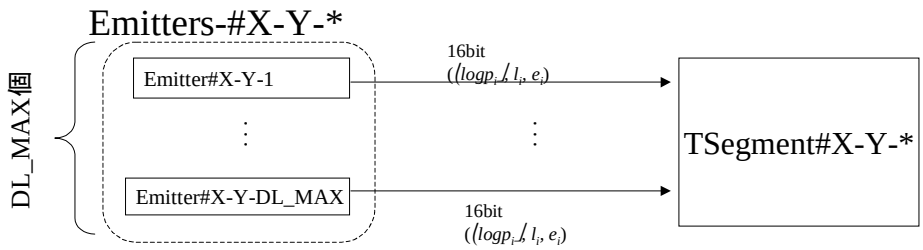


図 3 - 43　Emitters-TSegment の構造

表 3 - 8 TSegment ごとに含まれる Bus Line と Delivery Line の本数

X	Y	Bus Line 本数	Delivery Line 本数 (DL_NUM)
1	1~32	128	23
2	1~64	64	13
3	1~128	32	7
4	1~256	16	5
5	1~512	8	2
6	1~1024	4	2
7	1~2048	2	2

3.6. 詳細検討結果

3.6.1. 回路規模見積もり詳細

報告者の回路設計案から Rational TWIRL の回路規模と回路面積を評価した結果を表 3 - 9 に示す。条件は以下の通り。

- 0.09σm テクノロジーを使用し、集積度が以下である場合を前提とし、回路規模から回路面積を算出。また、1st/2nd Level Funnel の FIFO については、図 3 - 35 に示した SRAM を用いた回路による規模算出を行った。なお、配線に伴う面積は考慮していない。
  - 論理回路: $4.0\Delta10^{-6}mm^2$/gate
  - SRAM : $1.8\Delta10^{-6}mm^2$/bit
  - DRAM : $0.09\Delta10^{-6}mm^2$/bit

表 3 - 9 Rational TWIRL の回路規模評価結果

ブロック名-1	ブロック名-2	ブロック名-3	論理回路 (M gates)	SRAM (bit)	DRAM (bit)	回路面積 (90nm 時の mm²)
Largish Station	Emission Triplet 発行処理部	-	223	147,828,736	43,165,768,960	5043
	Buffer	Bubble Sort/ Random Shuffle	291	なし	なし	1164
		l-ソート処理部+ l-重複削除処理部	329	なし	なし	1316
		Delelery Line 出力処理部	727	なし	なし	2908
	パイプライン加算処理部	-	1,126	なし	なし	4504
	小計	-	2,696	147,828,736	43,165,768,960	14,935
Smallish Station	Emitter	-	147	なし	なし	588
	1st/2nd-Level Funnel	-	1280	1,503,936	なし	5120
	Segment(CellB)		757	なし	なし	3031
	小計	-	2,184	1,503,936	なし	8,739
Tiny Station	Emitter	-	1.6	なし	なし	6.4
	Segment(CellB)		53	なし	なし	212
	小計	-	54.6			218.4
合計			4,935	149,332,672	43,165,768,960	23,892

提案者は、Ratinal TWIRL に必要な回路面積は 0.13σm プロセステクノロジー使用時に 15960mm^2 であると主張している。テクノロジーの差を考慮すれば、提案者の回路面積見積もりは報告者の 3.12 倍と報告者は結論づける[17]。

[17] 提案者が実現法を示していない Buffer の l-ソート処理について、報告者の方法ではなく、典型的なバブルソートを用いた実装の場合、さらに 5.6Ggates の論理回路が必要となる。この場合の全体の回路面積を表 3 - 9 と同じ条件で評価すると、46,292mm² 必要と評価できる。この実装法を用いた場合、提案者の見積もりの 6.05 倍もの回路面積が必要と評価できる。

3.6.2. 複数の LSI を用いた TWIRL 実現に関する詳細検討

第 2 章で説明したように、TWIRL を 1 個の LSI として製造するためには、90nm プロセスを用いた場合でも直径 100mm 以上のウェハーを必要とするが、そのように巨大なウェハーを製造することは現在の技術では不可能である。この問題を回避するために、現在の技術で製造可能な LSI を複数用いて、Rational TWIRL を実現するという前提のもとに、必要な LSI 数について検討を行った。検討にあたっては、各 LSI はハード規模と IO ピン数の両方について、以下の制約条件を守るものとして、90nm プロセスを用いた場合に必要となる LSI 数の見積もりを行った。

- 回路規模に関する制約条件
  最大 1,600mm^2/LSI (400Mgates/LSI)
- IO ピン数に関する制約条件
  最大 3,000pins/LSI(1,500pins を IO として使用可能。残り 1,500pins は Vcc/Gnd 用)

結論から述べると、回路規模に関する制約条件を用いると LSI を 15 個で実装可能であるが、IO ピン数に関する制約条件も考慮すると、3,362 個必要である。つまり、IO 数のボトルネックが原因で膨大な LSI が必要となる。また、これだけの個数の 40mm 角 LSI を 1 枚のボードに搭載するためには、58Δ58 個に並べる必要があり、LSI 間の配線領域に 10mm 必要と仮定するとおよそ 3mΔ3m の巨大なボードが必要となる。よって、2004 年 1 月現在で製造可能な複数の LSI を 1 枚のボードに搭載することで TWIRL 実現を検討した場合、LSI 数が多すぎて製造不可能であり、この原因は IO 数のボトルネックによるものであると報告者は結論付ける。以下では、個々のブロックの分割法の詳細について説明する。

■回路規模に関する制約条件を考慮した場合
表 3 - 9 の回路規模見積より、Largish Station 全体で 23,892mm^2/1,600mm^2 = 15 個と評価できる。

■IO ピン数に関する制約条件を考慮した場合。
Largish Station 全体のうち、Emission Triplets 発行処理部に 168 個、Buffer に 218 個、パイプライン加算部に 2,912 個、Smallish Station と TinyStation 全体で 64 個それぞれ必要であり、合計 3,262 個必要と評価できる。

詳細を説明するために、Largish Station の Emission Triplet 発行処理部と Buffer の分割案を図 3 - 44 に、パイプライン加算部の分割案を図 3 - 45 に、Smallish/Tiny Station の分割案を図

3 - 46 にそれぞれ示す。参考のために、図 3 - 44,図 3 - 45,図 3 - 46 には分割前の IO 数と回路規模もあわせて示す。それぞれのブロックの分割方法について以下に説明する。

■Emission Triplet 発行処理部

複数の Emission Triplet 発行処理部を、IO 数が 1,500bit 以内となる単位で一つの LSI にまとめる(図 3 - 44)。つまり、50 個の Emission Triplet 発行処理部を一つの LSI として実装する。よって、必要な LSI 数は 8,375/50=168 個と評価できる。また、分割前の回路規模が 223M gates であるので、回路規模に関する制約条件を明らかに満たしている。。

なお、Emission Triplet 発行処理部全体では、30bitΔ8,375=251,250bit もの出力であり、IO 長が非常に多いにも関わらずこの部分を LSI の単位として区切る理由は、ロジック混載型 DRAM の LSI における論理回路部の回路規模を小さく抑えることを前提としているから[18]である。

■Buffer

入力(Emission Triplet 発行処理部)から出力(Delivery Line 出力処理部)の方向に沿って、IO 数が 1,500bit となるように分割する(図 3 - 44)。この場合、必要な LSI 数は(30Δ8,375+75,600)/1500=218 個と評価できる。

なお、Buffer に対してこのような分割を行った場合、n 分割することにより回路規模が 1/n なるブロックとならないブロックがある。つまり、分割数 n が特定値 c までなら、分割後の回路規模は 1/n となるが、n が c を超えた場合、分割後の回路規模は 1/n 以下にならないブロックが存在する。ブロックごとの n と c の詳細を以下に示す。これに基づくと、分割後の LSI の規模は 291Mgates/218 + 329M gates/4 + 727Mgates/8 = 174.5M gates となり、回路規模の制約条件である 400Mgates 以内である。

- Bubble Sort/Random Shuffle
  内部結合は密であるが分割は容易であるので、n 分割することで回路規模は 1/n となる。

- l-sort 処理部+l-重複削除処理部
  nΩ4 ならば、n 分割することで回路規模は 1/n となる。しかし、n>4 の場合、n 分割によっても回路規模は 1/n とならない。よって、回路規模を分割前の 1/4 以下にすることはできない(c=4)。なぜなら、このブロックは 16,384 個→8,192 個+8,192 個のデータ分類を行うことを繰り返す構造であり、n 分割することで 16,384/n 個→8,192/n 個+8,192/n 個という構造にすることができると考えられるが、本ブロックの機能はデータを 4,096 通りに分類するという性質上、n>4 にできないからである。

[18] DRAM には仕組み上リフレッシュ機構が必要であるため、一般的には論理回路用の LSI は別の LSI として製造されるが、もしロジック混載型 DRAM を製造する場合は、論理回路部の回路規模が小さいほど製造が容易であると報告者は考える。

- DL 出力部#1~#8

  n≧8 ならば、n 分割することで回路規模は 1/n となる。しかし、n>8 の場合、n 分割によっても回路規模は 1/n とならないと予想する(c=8)。これは、Delivery Line 出力部の内部結合は非常に密なので、ブロック番号ごと分割(最大 8 分割)は非常に簡単であるが、それ以上の分割は困難であると予想されるからである。

■パイプライン加算部

図 3 - 45 に示すように、4 方向に 375bit の IO を配置した LSI を複数用いて実現するのが最も効率的であると考えられる。これは、パイプライン加算部を一つの長方形とみなし、これを複数の LSI により実現する場合、正方形の LSI を用いるのが最も LSI 数が少なく効率的であるからである。この場合、必要な LSI 数は、それぞれのパイプライン加算部を実現するのに⌈5,120/375⌉×⌈9,450/375⌉=14×26=364 個必要であり、8 つのパイプライン加算部を実現するためには 364×8=2,912 個必要である。この分割案では n 分割することで回路規模が 1/n であり個々の LSI は 1,122Mgates/2,912=0.385Mgates であり回路規模の制約条件を満たす。

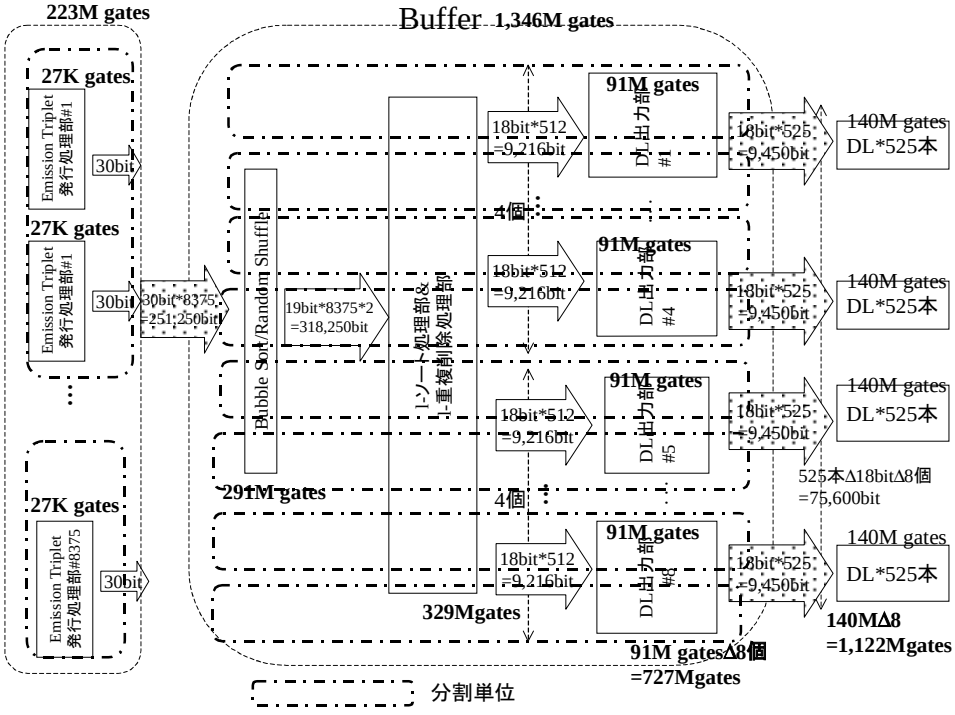


図 3 - 44　IO ピン数を基準とした Emission Triplet 発行処理部+Buffer の分割案

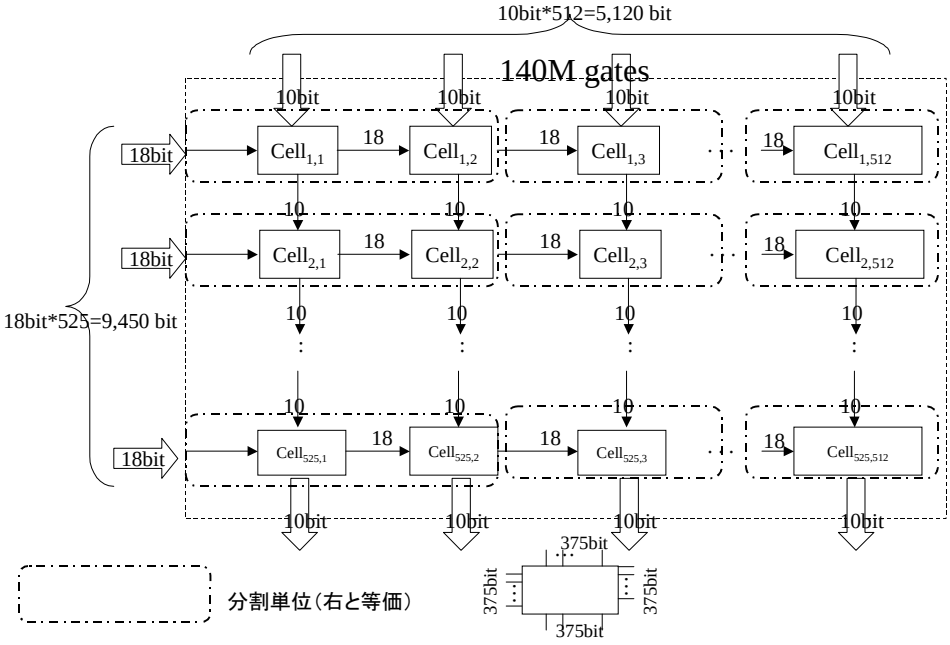


図 3 - 45　IO ピン数を基準としたパイプライン加算処理部の分割案

■Smallish/Tiny Station

64 個必要であると見積もられる。以下に理由を説明する。

まず、図 3 - 46 より、Emitter-Funnel と Segment 間は IO の結合が密であり、Emitter-Funnel と Segment を別々の LSI として実現するのは不利であると考えられる。よって、図 3 - 46 の破線で示すように、Emitter-Funnel を一つの LSI に収めながら、Bus Line に沿った形で分割するのが最適であると考えられる。 また、分割にあたっては、Emitter と Segment の性質から分割数を 2 べきに設定するのが最適であり、64 分割ならば、(40,960bit(input)+40,960bit(output))/64=1,280bit となり、1,500bit 以内に収まる。

分割後の回路規模については、Segment は 1/64 になるが、G_i<64 である一部の Emitter-Funnel については、分割後の回路規模が $1/G_i$ 以下にならない。この性質を考慮した場合、分割後の各 LSI の大きさは 757M/64 +404M/4+440M/8+194M/16+103M/32+71M/64+68M/128=184.7 M gates と評価でき、回路規模に関する制約条件を満たす。

さらに上で述べた分割は、Smallish+Tiny Station という一括した形で行うことができ、Tiny Station の回路規模は小さく(53M gates)無視できるので、分割後の LSI は Smallish と Tiny Station あわせて 64 個であると結論付けられる。

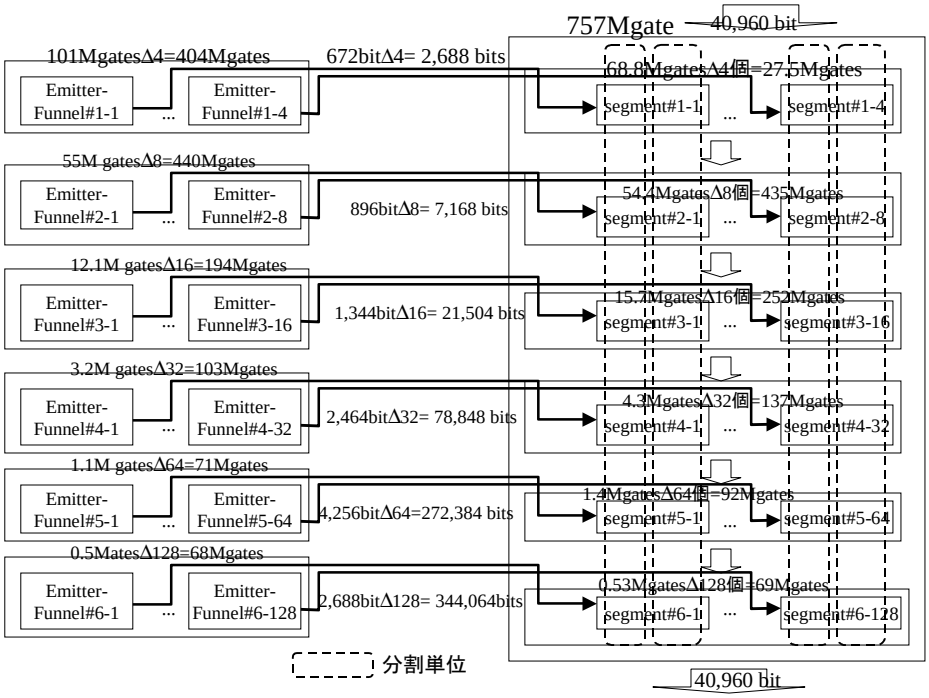


図 3 - 46 IO ピン数を基準とした Smallish Station の分割案

3.6.3. 複数のボードを用いた TWIRL 実現に関する詳細検討

3.6.2 節において、2004 年 1 月の時点で製造可能な LSI を複数用いて Ratonal TWIRL を実現する場合、3,362 個もの LSI が必要であり、LSI 数が多すぎて 1 枚のボードでは実現不可能であるとの結論を報告者は得た。よって、さらなる TWIRL 実現可能性の検討として、複数のボードを用いた場合の TWIRL の実現可能性について考察する。
本検討による実現可能性検討方針は以下の通りである。

1. 1 ボードあたり 100 個の LSI を搭載可能という前提条件
2. 分割にあたっては、ボード間の IO 数をなるべく少なくする方法を採用する。
3. 2.の結果、ボード間の IO 数が最大となる箇所により実現可能性を判断する。
4. ボード間の IO 数が多いほど IO の高速化が難しくなることを考慮すると、TWIRL 提案者の主張する動作周波数(1GHz)を前提の場合、ボード間 IO が 400bit を超えるなら、現在の技術では実現不可能と報告者は考えるからである。

結論から述べると、パイプライン加算部が原因で、ボード間 IO を 1,500bit 以下にするのは不可能であるため、実現不可能であると考えられる。

本評価については、概要のみ説明する。

■Emission Triplet 発行処理部+Buffer(図 3 - 44 に示す通り、Emission Triplet 発行処理と Buffer の間の IO 数が多く、Buffer からパイプライン加算処理部の間の IO 数が少ない。よって、Emission Triplet 発行処理部の LSI と Buffer の LSI を同一のボードで搭載する方法が最適であると考える。
必要なボードの枚数 n 枚とすると、最小の枚数は 5 枚であり、Buffer からパイプライン加算処理部の 75,600bit を n 枚で実現すると、ボード間 IO は 75,600/n と評価できる。n が小さいほどボード枚数は小さくなるが、ボード間 IO は多くなる。逆に、ボード枚数を増やすことで、ボードごとの IO 数は小さくすることができる。IO 数を優先した場合、n=168+218 の時に、75,600/(168+218)=196bit と評価できる。
■パイプライン加算部
パイプライン加算部 1 個あたり、長方形に配置された 26Δ14=364 個の LSI を必要とする。ボードに 100 個の LSI 搭載する場合、ボード間 IO 数最小で実現するためには、10Δ10 の LSI のボードを用いるのが最適であり、ボード枚数は⌈26/10⌉Δ⌈14/10⌉Δ8=48 枚であり、ボード間 IO 数は 1,500bitΔ10=15,000bit と評価できる。ただし、Emission Triplet 発行処理部と同じように、ボードあたりの LSI 数を小さくすることで、ボード枚数を多くする代わりに IO 数を減少させることができるが、パイプライン加算部の格子状の配置されるという性質上、LSI あたりの IO 数である 1,500bit に未満するのは不可能である。
■Smallish/Tiny Station
パイプライン加算部と同じ理由により、ボード間 IO を 1,500bit 以下にするのは不可能である。最小ボード枚数は、64 個 LSI を一つのボードに搭載した場合の 1 枚となる。

なお、ボード間の IO 数が問題となっている以上、ボードの枚数に関する議論は重要でないかもしれないが、上記の最小枚数を合計すると、5+48+1=54 枚と評価できる。

参考文献

[Ber01] Daniel J. Bernstein, “Circuits for integer factorization: a proposal”, preprint, 2001. Availabe at http://cr.yp.to/papers/nfscircuit.pdf

[CRY02] 情報処理振興事業協会 (IPA), 通信・放送機構 (TAO), “暗号技術評価報告書 (2002 年度版)”, March, 2002. Availabe at http://www.shiba.tao.go.jp/kenkyu/CRYPTREC/PDF/c02_report.pdf

[Cha94] Chin-Chieh Chao, “Multiport Memory Design for An MCM Coprocessor”, Technical Report No.ICL94-037, pp.95-113, Department of Electrical Engineering Stanford Univ., December 1994.

[F+03] J. Franke and others, “RSA-576”, Email announcement, December 2003. Available at http://www.crypto-world.com/announcements/rsa576.txt

[GS03] Willi Geiselmann and Rainer Steinwandt, “A dedicated sieving hardware”, *PKC 2003*, LNCS 2567, pp.254-266, Springer-Verlag, 2003.

[GS04] Willi Geiselmann and Rainer Steinwandt, “Yet another sieveing device”, *CT-RSA 2004*, LNCS 2964, pp.278-291, Springer-Verlag, 2004. (Preliminary version is included in IACR ePrint Archive, 2003/202, 2003. Available at http://eprint.iacr.org/2003/202.pdf)

[IK03] 伊豆 哲也, 木田 祐司, “素因数分解の現状について”, 日本応用数理学会論文誌 Vol. 13, No. 2, pp.289-304, 2003.

[Kob95] 小林 芳直, “定本 ASIC のシステム設計—論理回路の記述からステート・マシンの設計まで”, CQ 出版, 1995.

[Kob98] 小林 芳直, “定本 ASIC の論理回路設計—高速・高信頼ディジタル・システムのための設計ノウハウ”, CQ 出版, 1998.

[LL93] Arjen K. Lenstra and H.W. Lenstra (eds.), “The development of the number field sieve”, Vol. 1554 in Lecture Notes in Mathematics (LNM), Springer-Verlag, 1993.

[LS00] Arjen K. Lenstra and Adi Shamir, “Analysis and optimization of the TWINKLE factoring device”, *EUROCRYPT 2000*, LNCS 1807, pp.35-52, Springer-Verlag, 2000.

[LSTT02] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson and Eran Tromer, “Analysis of Bernstein’s circuit”, *ASIACRYPT 2002*, LNCS 2501, pp.1-26, Springer-Verlag, 2002.

[LTS+03] Arjen K. Lenstra, Eran Tromer, Adi Shamir, Wil Kortsmit, Bruce Dodson, James Hughes and Paul Leyland, “Factoring estimates for a 1024-bit RSA modulus”, *ASIACRYPT 2003*, LNCS 2894, pp.55-74, Springer-Verlag, 2003.

[RSA03] RSA Security, “TWIRL and RSA Key Size”, May 2003. Available at `http://www.rsasecurity.com/rsalabs/technotes/twirl.html`

[Sha99] Adi Shamir, “Factoring large numbers with the TWINKLE device (extended abstract)”, *CHES 1999*, LNCS 1717, pp.2-12, Springer-Verlag, 1999.

[ST03] Adi Shamir and Eran Tromer, “Factoring large numbers with the TWIRL device”, *CRYPTO 2003*, LNCS 2729, pp.1-26, Springer-Verlag, 2003.