修 士 論 文

# 積和エンジンを搭載した<br>高機能メモリコントローラに関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

今井 俊晴

2004 年 9 月

修 士 論 文

# 積和エンジンを搭載した<br>高機能メモリコントローラに関する研究


指導教員　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

210202 今井 俊晴



提出年月: 2004 年 8 月

## 概 要

近年, 計算機の性能向上に伴い音声や映像を扱うマルチメディアを対象としたアプリケーションが普及してきている. そして, 音声・映像の高品質化を目的とした DVD 等におけるハイビットレート・ハイサンプリングレートによるデータ量の増加に伴い, より高速な信号処理能力が必要とされている.

本論文では, メモリアクセス時間を最小限に抑え,CPU との処理速度差を軽減するメモリコントローラに,DSP の特徴である積和演算能力を設け, 応答時間の短縮と,CPU との負荷分散によるスループットの向上を実現する方式を提案する. また, 提案する機構を VHDL で設計し, シミュレーションにより性能の評価を行う.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

## 1.1　背景と目的

近年,計算機の性能向上に伴い音声や映像を扱うマルチメディアを対象としたアプリケーションが普及してきている．その理由として,CPU の高速化・計算機の性能向上に伴い,様々なリアルタイム処理が可能となってきていることがあげられる.

そのようなマルチメディアを対象としたアプリケーションの普及例として,図 1.1 に音楽プロダクションの例を示す.

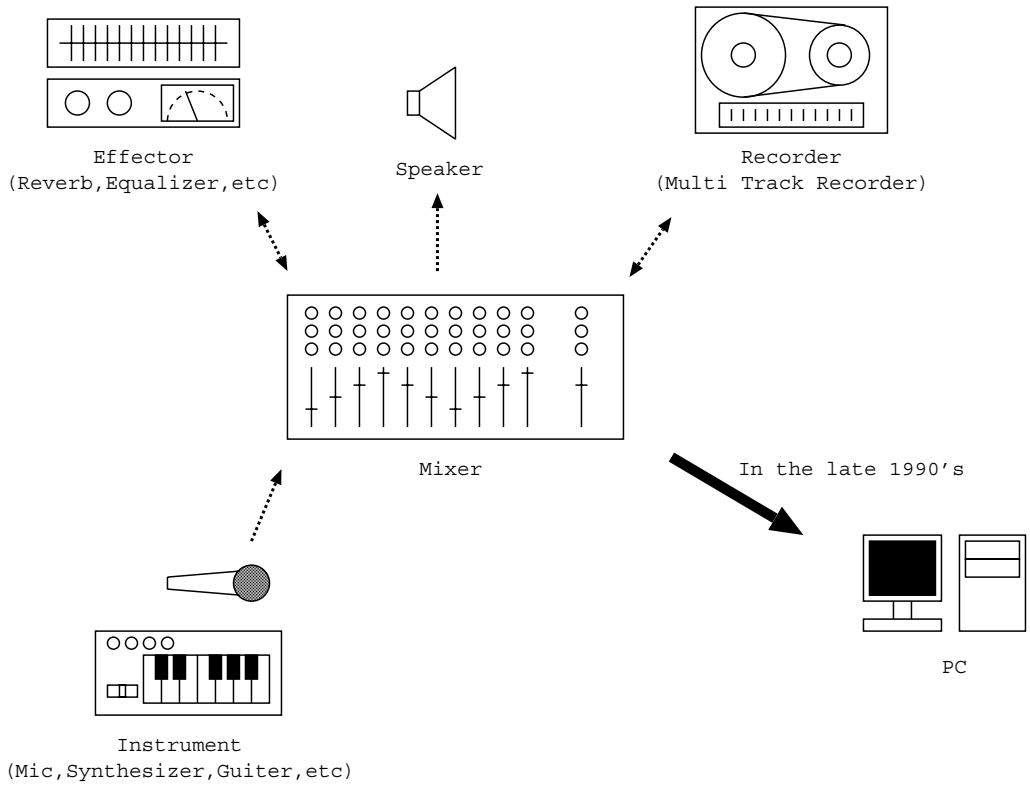


図 1.1: 音楽プロダクションの例

図 1.1 の左側に音楽プロダクションにおけるレコーディングシステムに必要な主な機材を示している. これらの機材は, 各々が専用のハードウェアで構成されており, コスト面・スペース面は共にかなり大規模なものとなる. また, 各々の機材の連係動作や設定の再現性等の様々な問題も存在する.

計算機の性能向上に伴い 1990 年代後半より, 図 1.1 の左側に示すような大規模なレコーディングシステムが, パーソナル・コンピュータ（以下 PC）のみで実現可能となってきている. 大規模なレコーディングシステムを PC のみで実現できることの利点として, 先ほど問題点としたコスト, スペース, 機材の連係動作, そして設定の再現性の改善があげられる. 他の利点としては, アナログ処理を介さず, すべての作業をディジタル処理できるため, 音質の劣化を伴わない編集作業が可能となる.

そのような利点があるため, 現在はパーソナルユース, プロユースと共に PC を利用した音楽プロダクションが普及している. しかし, 現在の計算機性能でもまだ処理能力が不足しているのが現状である.

### 1.1.1 メディアプロセッシングにおける要求

このようなアプリケーションを利用しメディアプロセッシングを行うプロダクションにおける要求として, 次のようなことががあげられる.

- 直感性

  ボリューム, イコライザー等の値を変化させたとき, すぐにその変化が音に反映されないと, どのくらい値を変化させたのかがわからない. つまり, 直感性が重要視されるメディアプロセッシングを行うプロダクションにおいては, 見聞きしながら編集作業を行うことが必要となるためにリアルタイム性が要求され, 高速な処理が求められている.

  現在でもリアルタイム処理ができないエフェクトとして, ノイズ・リダクション（雑音除去）等があげられる. そのような場合は, 非再生時のファイルに対して編集作業を行うファイルベースの処理を行うことになる. つまり, ファイルベースの処理は見聞きしながら編集作業が行えないため, 大変不便なもとなる.

  現在はリアルタイム処理が可能なボリューム, イコライザー等も, 計算機性能が十分でなかった頃のアプリケーションではファイルベースの処理を必要としたため, 音楽プロダクションに実用できるものではなかった.

- マルチ・トラック

  普段, 我々が耳にしている CD はステレオ (L,R) の 2 チャンネル,DVD は 5.1 チャンネル (Center,FrontL,FrontR,BackL,BackR,+woofer) である. しかし, ポップス, ロック等における録音・編集方法は, 個々の楽器を別々のトラックに録音し, その多数の

トラックを同時に再生し編集を行い, その後,2 チャンネルあるいは 5.1 チャンネルにまとめる , マルチ・トラック編集が現在の主流となっている.

必要とされるトラック数は 200 トラックを越える場合もある. そのような多数のトラックを同時に再生し, リアルタイムに編集作業を行うことを可能とする信号処理能力が要求されている.

- 高品質化

  音声・映像の高品質化を目的とした DVD 等におけるハイビットレート・ハイサンプリングレートによるデータ量の増加に伴い, より高速な信号処理能力が求められている.

### 1.1.2 メディアプロセッシングにおける問題点

DSP による CPU 負荷分散

このような要求に対し, 高速な信号処理を実現する一手法として, CPU と別に DSP（Digital Signal Processor）を設けて CPU 負荷を分散し処理の高速化を図る方法があるが, 次のような問題点があげられる.

- CPU と DSP の命令セットが異なるためプログラミングが困難である.
- CPU–DSP 間でのデータの受け渡しや, 処理の開始・終了の通知に要するオーバヘッドの発生が問題とされている.
- CPU と DSP の異なる 2 つのプロセッサを協調動作させるために設計・構造は複雑なものとなり, コスト及びハードウェア量が増大する.

メモリと CPU の処理速度差

一方,CPU の高速化は図られているが CPU 周辺装置との処理速度差がスループット向上の問題となっている. 例をあげると, 現在の CPU の動作周波数は GHz オーダーであるのに対し,CPU 周辺装置の一つであるメモリの動作周波数は 100MHz から 300MHz 程度である.

音楽プロダクションにおいて, そのようなメモリに対してのアクセス回数は増加する傾向にある.

- トラック数の増加

  先ほど述べたマルチ・トラック編集でのトラック数の増加に伴い, メモリアクセス回数は増加する.

- 発音数の増加

  PC で実現する仮想楽器としてソフトウェア・シンセサイザーが存在する．ソフトウェア・シンセサイザーはマルチトラック編集と併用されることが多い．現在主流となっているソフトウェア・シンセサイザーはPCM 方式を採用している．その方式では, メモリ上に様々な楽器の波形テーブルを配置し, その波形テーブルにアクセスすることにより発音を行う．発音数が多くなるほどメモリアクセス回数は増加する．

このように, メディアプロセッシングで扱うような大規模データに対する処理では, メモリのアクセススピード, アクセス回数が処理速度を制限する一つの要因となり, CPU 高速化によるスループット向上の妨げとなっている．

このような CPU とメモリの処理速度差を軽減するために, 効率の良いデータ転送によりメモリアクセス時間を抑える一手法として, DRAM の同一ページ内への連続アクセス方法（Stride Data Transfer:SDT）が提案されている [1].

### 1.1.3 本研究の目的

本論文では,SDT を利用しメモリアクセス時間を最小限に抑え,CPU との処理速度差を軽減するメモリコントローラ（以下 MC）に, DSP の特徴である積和演算能力を設けることにより, 高速なフィルター計算を実現する方式を提案する．

従来では, 一度のメモリアクセスのデータ転送量はキャッシュのブロック単位であったが, SDT を利用することにより, 一度のメモリアクセスで自由なデータ転送量を設定することが可能となる．また, 一度のメモリアクセスで転送可能な最大データ転送量は DRAM のページ単位となっている．

そして, メモリから直接データを受け取る MC 内に積和演算器を設けているため, メモリの読み出しスピードに合わせて連続した演算を行うことで, 最高速度のフィルター計算が達成可能となる．

本論文では, 提案する方式を実現する機構を組み込んだ MC の VHDL による設計および実装について述べ, RTL シミュレーションによりその有効性を示す．

## 1.2 本論文の構成

本論文の構成を以下に示す．

第 2 章

本研究で利用するデータ転送方式について説明する．

第３章

本研究で対象とするフィルタ計算の概要, および提案する機構の DSP に対する優位性を述べる.

第４章

提案する機構を実現する MC の仕様および動作ついて説明する.

第５章

提案機構の基本性能評価を示す.

第６章

本論文の関連研究を紹介する.

第７章

本論文のまとめ，及び今後の課題について述べる.

# 第2章　データ転送方式

本章では,DRAM の構造によるデータアクセスの問題点, そして, 本研究で利用するデータ転送方式（SDT 方式）と FIFO バッファについての概要を述べる.

## 2.1　DRAM

### 2.1.1　DRAM のアクセス方法

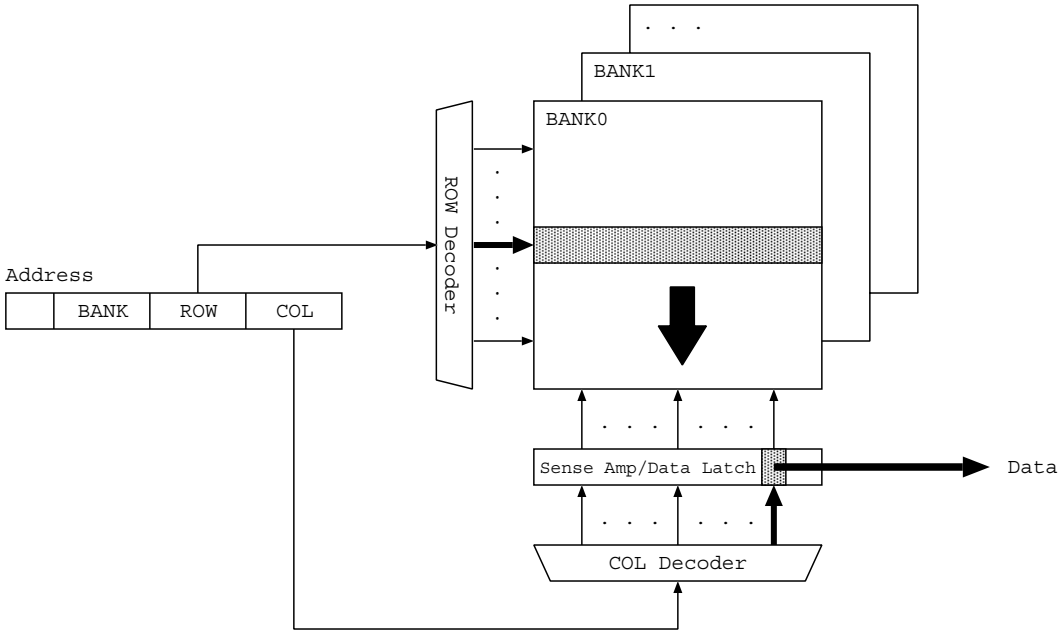


図 2.1: DRAM アクセス

計算機の主記憶装置として DRAM（Dynamic Random Access Memory）の使用が主流となっている.

現在の DRAM のメモリアレイは複数のバンクから構成されており, バンクごとに独立

した動作が可能になっている.

DRAM アクセスはメモリアドレスのバンク（Bank）アドレスで一つのバンクを選択し, そのバンクに対し行（Row）アドレスを与え, 該当する Row アドレスのデータ全体をセンス・アンプで増幅し, 一旦ラッチする. そしてラッチされたデータに対して列（Col）アドレスを与えることで CL（CAS Latency）後にデータが読み出される（図 2.1）. なお,Row,Col アドレスは信号線を共有しており, 時分割で Row アドレス,Col アドレスの順に与えられる.

### 2.1.2 DRAM アクセスの問題点

2.1.1 節で述べたように,DRAM のアクセス方法は,Bank,Row アドレスの指定後, 指定した Row 全体をセンス・アンプでラッチし, そのラッチされた Row に対して Col アドレスを与え, 該当するデータに対するアクセスを行うといったものである.

この DRAM アクセスの問題点は, 同一の Bank,Row 内に存在する複数のデータにアクセスする際にも, Bank,Row アドレスの毎回指定を行うためデータ転送までのレイテンシが増大してしまう.

府川ら [1] が着目している主記憶データベース（MMDB）においては, アクセスされるデータは, 同一の Bank,Row 内に一定間隔で不連続に複数個存在する（図 2.2）.

そのようなデータをアクセスする際, 従来の MC では各データを含むキャッシュブロック単位でのデータアクセスを行い, それぞれのブロックに対して Bank,Row アドレスを毎回指定するため効率が悪く, データ転送までのレイテンシが増大する.

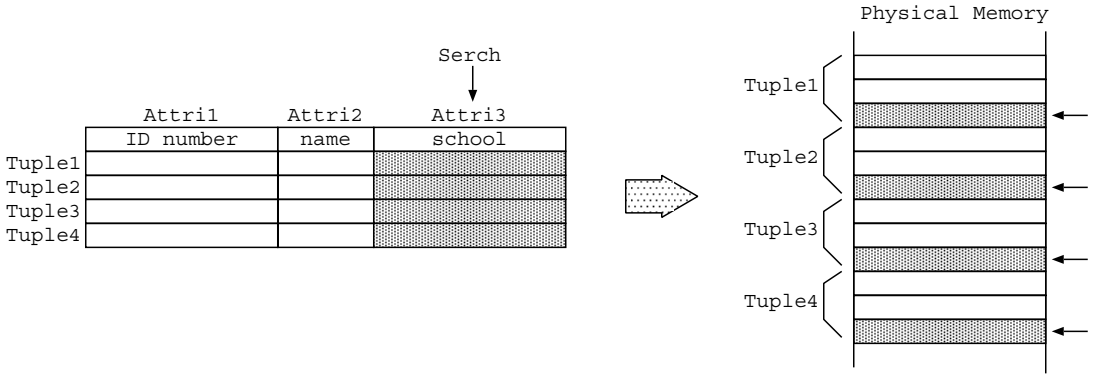


図 2.2: MMDB のメモリ配置

また, 本研究で着目するメディアプロセッシングにおいてアクセスされるデータは, 図 2.3 の下部 (Data) に示すように連続して存在している. （サウンドファイルとして主に使用される WAV ファイル,AIFF ファイルは共に図 2.3 のような構造となる.）そのため, アクセスされるデータは, 同一の Bank,Row 内に連続して存在する.

そのようなデータにアクセスする際も同様に, 従来の MC では, 各データを含むキャッシュブロック単位でのデータアクセスを行い, 必要な連続データ数が満たされるまで Bank,Row アドレスを毎回指定するため効率が悪く, データ転送までのレイテンシが増大する.

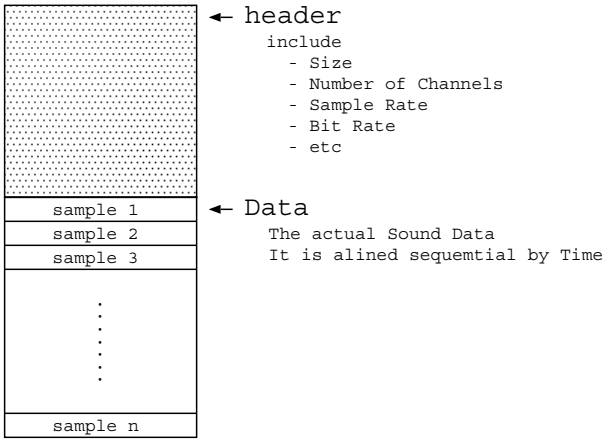


図 2.3: メディアプロセッシングで使用されるファイルフォーマット例（サウンドファイル）

## 2.2 Stride Data Transfer（SDT）方式 [1]

本節では, メモリアクセス時間を抑える一手法として提案されている, DRAM の同一ページ内への連続アクセス方法（Stride Data Transfer:SDT）について述べる.

SDT の実現方法については 4 章で述べる.

### 2.2.1 SDT の概要

Stride Data Transfer（SDT）方式はメモリアクセス時間の短縮のために DRAM のハードウェア特性を利用したデータ転送方式であり, DMA 方式のようにブロック毎のアドレス指定を削減し, 一回のアドレス指定で大量のデータを転送することを可能とする.

SDT の特徴は以下となる,

- 連続データ転送

  同一の Bank,Row 内に存在する複数のデータにアクセスする際,Bank,Row アドレスを指定した状態で, 複数の Col アドレスを連続して与えることにより該当するデータに連続してアクセスすることができる.

  この方法を用いると, 2.1.2 節で DRAM アクセスの問題点として述べたような Bank,Row アドレスの毎回指定を行う必要がなくなりデータ転送までのレイテンシが短縮される.

  DRAM がサポートする高速ページモード (First Page Mode:FP Mode) 等の連続データ転送（バースト転送）は, このような方法でデータ転送までのレイテンシを短縮している.

  SDT 方式は MC が同様の方法をサポートする. つまり, 同一の Bank,Row 内に存在する複数のデータにアクセスする際, Bank,Row アドレスを指定した状態で, MC が自動的に複数の Col アドレスを生成し, 複数の Col アドレスを連続して DRAM に与えることにより, 該当するデータに連続してアクセスする.

  SDT 方式は等間隔に並んだデータに対する連続データ転送方式であり, 現在の Col アドレスに, 次にアクセスするデータの間隔値を加算することにより,MC が自動的に次の Col アドレスを生成する.

  この間隔値の設定によって, 一定間隔に存在する連続・不連続なデータに対して連続したデータ転送を行うことができる.

- 自由度の高い連続転送データ数の設定

  通常, 連続してアクセスするデータサイズ（バースト長:Burst Length：BL）はキャッシュのブロックサイズとする. 例を挙げると, データバス:32bit, キャッシュのブロックサイズ:32Byte,1 ブロック:8 ワードといった構成の場合,DRAM の BL を 8 と設定しキャッシュのブロックサイズに合わせることで 1 回のメモリアクセスでブロック単位の転送を行う[1].

  Intel 社の定める SDRAM の仕様 [2] より, 通常サポートされている BL は BL=1,2,4 であり, 製品によっては BL=8, フルページをサポートするものも存在する.

  SDT 方式は, 自由度の高い連続転送データ数の設定を行うことができ,BL に限定されない連続データ転送が可能である. また, 最大でフルページ分の連続データ転送が可能である.

[1]ただし, メモリのバスクロックの 1 サイクル毎に BL=1(データバス幅) にあたるデータがキャッシュに格納されるため, BL=8 の転送には計 8 バスクロックサイクルが必要となる.

2.1.2 節で述べた MMDB おける SDT の利用方法は, 次のようなものになる.

同一の Bank,Row 内に一定間隔で存在する複数の不連続なデータに対して, 初回のみ Bank,Row アドレスを指定し, その後は MC が一定間隔値を加算し Col アドレスを連続して自動生成することにより, 同一 Bank,Row アドレスに存在する複数のデータに対して, BL に限定されないデータ数の連続データ転送が可能となる.

本研究で着目するメディアプロセッシングにおける SDT の利用方法は, 次のようなものになる.

MC が加算する一定間隔値を 1 と設定し, 同一の Bank,Row 内に存在する連続なデータに対して, 初回のみ Bank,Row アドレスの指定を行う. その後は MC によって連続して自動生成される Col アドレスは連続したものとなり, 同一 Bank,Row アドレスに存在する複数の連続したデータに対して, BL に限定されないデータ数の連続データ転送が可能となる.

### 2.2.2 従来の MC と SDT のデータ転送方式の比較

従来の MC と SDT のデータ転送方式の比較を図 2.4 に示す. 従来の MC では物理アドレスに一定間隔で存在する各データに対して, Bank,Row,Col アドレスを毎回指定するためデータ読み出しのレイテンシが発生するが, SDT 方式により,MC と DRAM 間での Bank,Row アドレスの再入力の除去によってデータの高速な連続読み出しが可能になる. また図 2.4 において, 従来ではプロセッサが四回のメモリリクエストを発行するのに対し, SDT 方式ではプロセッサと MC 間のメモリリクエストを一括することでデータ転送効率を向上させる.

SDT 方式の利点は以下となる.

- 必要なデータのみを選択し, 不必要なデータを含まない連続データ転送
- 一括化したメモリリクエストによるメモリアクセス時間の短縮
- 自由度の高い連続データ転送数設定

## 2.3 FIFO バッファ [3]

本節では,SDT 方式によってメモリからプロセッサに転送されるデータを格納する FIFO バッファについて述べる.

SDT 方式と FIFO バッファの協調動作については 4 章で述べる.

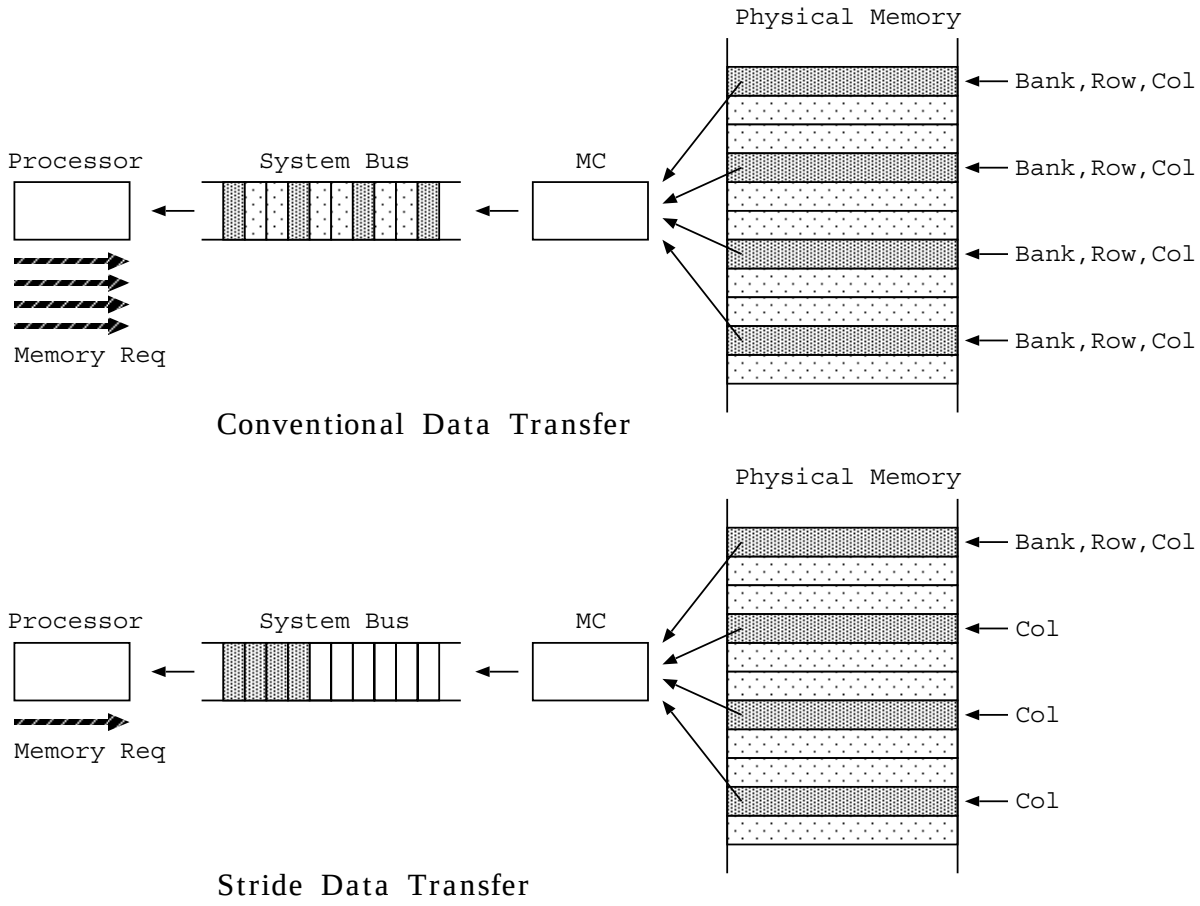


図 2.4: SDT 方式によるデータ転送

### 2.3.1 FIFO バッファの概要

通常のキャッシュは, 一般目的としたデータ処理において有効であるが, 時間的および空間的局所性が存在しない再使用されることが少いデータを扱う場合には, 有効に利用できないことがある.

そのような特徴をもつアプリケーションとしてデータベースやメディアプロセッシングなどがあげられる. これらのようなアプリケーションでは大規模なデータセットを扱い, そのデータは再使用されることが少いため時間的および空間的局所性が存在しない.

これらのようなアプリケーションに対して, キャッシュを複数のパーティションに分割し, 目的に合わせて利用するキャッシュを, 再構成可能なキャッシュ[4] と呼んでいる.

Khairuddin ら [3] は,2 ウェイセットアソシアティブキャッシュを基本として再構成可能なキャッシュを構築している. 再構成可能なキャッシュの構成を決定するために特殊なレジスタが用意されており, そのレジスタの値が 0 のときは通常の 2 ウェイセットアソシアティブキャッシュとして動作し, そのレジスタの値がセットされたときには 1 つのパーティションを通常のキャッシュ, もう一つのパーティションを FIFO バッファとして使用する. この FIFO バッファに, メモリから連続転送されるデータを格納する.

2 ウェイセットアソシアティブキャッシュのパーティション分割を図 2.5 に示す.

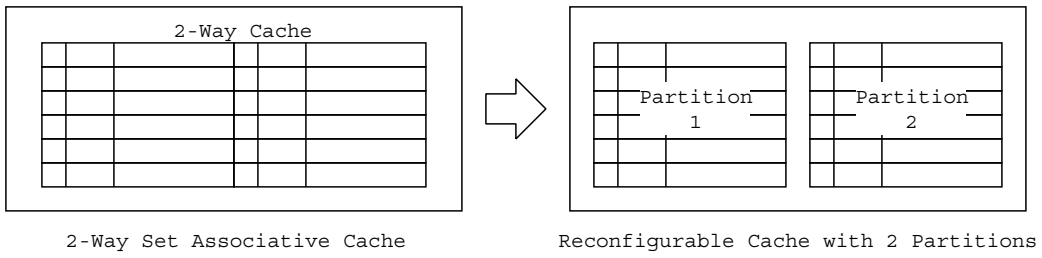


図 2.5: 2 ウェイキャッシュのパーティション分割

### 2.3.2 FIFO バッファの動作

FIFO バッファにアクセスする際には, リードカウンタとライトカウンタを使用する（図 2.6）. リードカウンタはデータの読み出し位置を示し, ライトカウンタはデータの書き込み位置を示す.

2.3.1 節で述べたように, キャッシュの構成を決定する特殊なレジスタがセットされ, FIFO バッファの使用を始める際, リードカウンタとライトカウンタは 0 にリセットされる.

動作例を示すと, まずプロセッサがメモリにアクセスし FIFO バッファにデータが書き込まれライトカウンタが自動的にインクリメントしていく. その後,FIFO バッファのデータ

を読み出すとリードカウンタが自動的にインクリメントしていき, やがてリードカウンタがライトカウンタと同じ値を指す. このとき更に読み出しを要求した場合,FIFO バッファに読み出すべき有効なデータは存在しないことを表し, ミスシグナルがアサートされる.

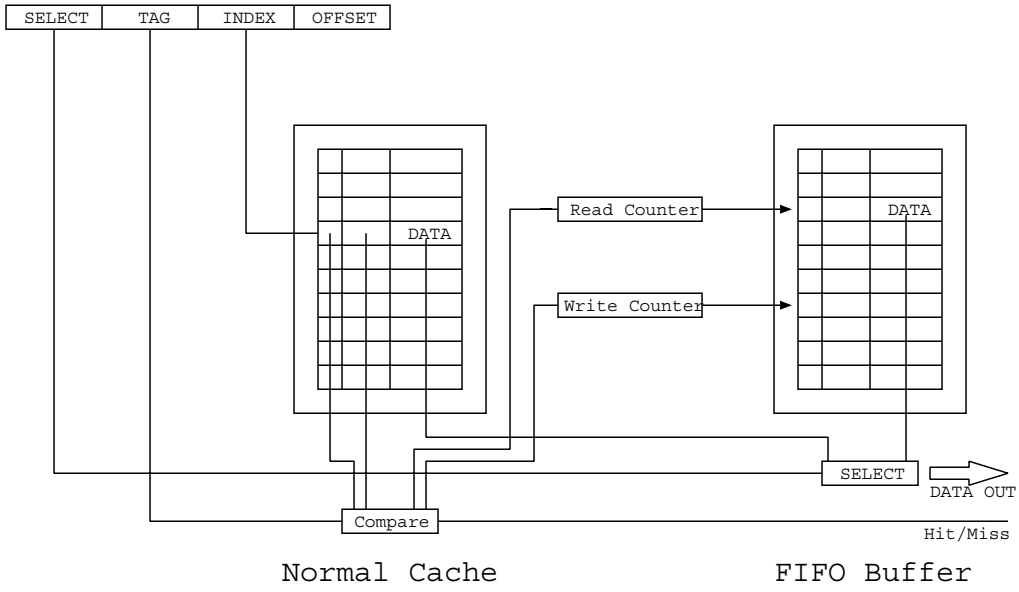


図 2.6: 再構成可能なキャッシュの FIFO バッファとしての利用

### 2.3.3 通常キャッシュと FIFO バッファを使用したキャッシュの比較

図 2.7 に通常キャッシュと,SDT 方式と協調動作を行った場合の FIFO バッファを使用したキャッシュの比較を示す.

2.3.1 節でも述べたが, 通常のキャッシュは, 一般目的としたデータ処理において有効であるが, 再使用されることが少いために時間的および空間的局所性が存在しないデータを扱う場合には, 有効に利用できないことがある.

図 2.7 の上部は, 通常のキャッシュを使用した際の,MMDB でのメモリアクセス後のキャッシュの状態を示している. 2.1.2 節で述べたように,MMDB のメモリ上でのデータ配置は, シーケンシャルではあるが等間隔の隙間が存在する.

問題点としては, 通常, キャッシュ–メモリ間のデータの受け渡しはキャッシュブロックサイズで行われるが, MMDB のように必要なデータが不連続に存在する場合, データの受け渡しを行うブロック内のすべてのデータが必要なデータとなることはなく, 不必要なデータも含まれる. そのため, 必要なデータが断片的な状態でキャッシュに配置されることに

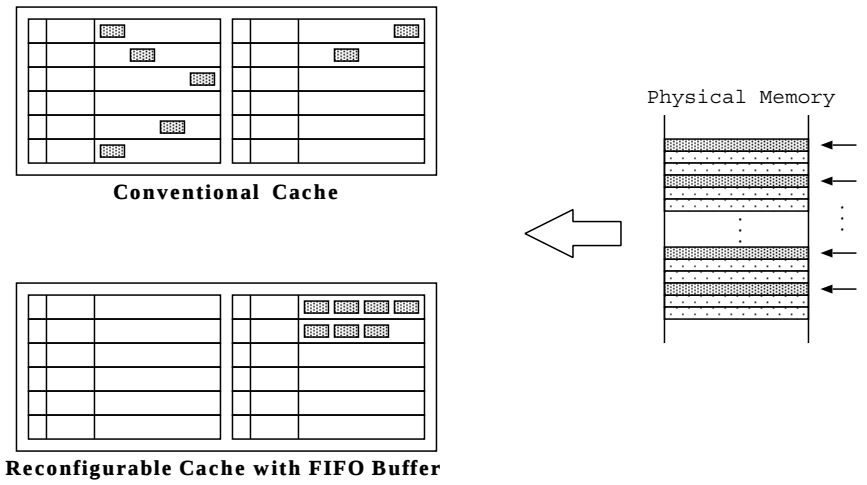


図 2.7: SDT 方式による FIFO バッファの使用

なる.

このように不必要なデータがキャッシュに配置されることにより, 限られたキュッシュスペースを有効に利用できないばかりでなく, 不必要なデータの転送に要する転送時間も無駄なものとなる.

また,2.1.2 節で述べたように, 本研究で着目するメディアプロセッシングでは, 必要なデータは連続してメモリ上に配置されている.

この場合の問題点としては, キャッシュ–メモリ間のデータの受け渡しを行うブロック内のデータに不必要なデータが含まれることは少なくなるが, 必要なデータがすべてキャッシュに取り込まれるまで, ブロックサイズでのデータ転送を繰り返すことになる. また, キャッシュのタグ, インデックスが一致するデータのリプレースに対するデータ転送も発生することになる.

図 2.7 の下部は,SDT 方式と協調動作を行った場合の FIFO バッファの様子を示している.

FIFO の First–In,First–Out という特徴は, 再使用されることが少く時間的および空間的局所性が存在しないシーケンシャルなアクセスを行うデータに対して適したものである.

SDT 方式でのデータアクセスを行うことにより, 必要なデータのみを一括して連続転送することが可能である. また, メモリから転送されるそのような連続データを FIFO バッファで受け取ることにより, 先ほど問題としていた, キャッシュスペースの有効利用, データ転送回数を改善することができる.

# 第3章　ディジタルフィルタ

本章では, ディジタル信号処理の基本演算, 基本的なディジタルフィルタの例として FIR ディジタルフィルタの構成, そして本研究の特徴となる MC に積和演算能力を持たせることの優位性について述べる.

## 3.1　畳み込み

### 3.1.1　畳み込みとは

どのようなフィルタでも, 入力信号とそのフィルタの持つインパルス応答（Impulse Response:IR）[1] を畳み込むことによって, フィルタされた出力信号を生成している. つまり, 畳み込み（convolution）はディジタルオーディオ信号処理の基本演算である [5],[6].

ある音を任意の IR で畳み込むことにより, さまざまな音楽的効果を作り出すことが可能である. 例をあげると, ある空間の IR を獲得し, その IR と任意の入力信号とを畳み込むことによって, 一種の複雑なフィルタであるリバーブ[2] を作りだすことができる. 畳み込まれた音はもとの音と混ぜ合わせることにより, 入力信号をその空間で演奏したような音が得られる.

リバーブエフェクトだけでなく, すべてのオーディオプロセッサでは, 着目するシステムの IR をオーディオ信号と畳み込むことにより, システムが持つ性質をその信号に与えることができる.

図 3.1 に畳み込みの典型的な例を示す.

1. 単位インパルスによる入力信号の畳み込みは恒等演算となる.
2. 0.5 倍にスケールされた単位インパルスによる畳み込みは入力信号を 0.5 倍にスケールする.
3. 時間シフト（遅延）された単位インパルスによる畳み込みは入力信号を時間シフトする.

[1] 単一サンプルのパルス（単位インパルス:unit impulse）をフィルタに与えることにより生成される出力信号を, フィルタのインパルス応答という. インパルス応答はフィルタ（あるシステム, あるいは, ある空間等）の持つ特徴であり, 畳み込みを行う際のフィルタ係数に相当する.

[2] 風呂場やコンサートホールなどで顕著に得られる音の残響.

4. 広く離れた二つのインパルスによる畳み込み.

2. は入力信号に対するヴォリュームの変化に相当する. 4. はエコー（やまびこ）の効果を作り出す. 遅延した IR を複数回設け, 徐々にスケールダウンさせるとリバーブの効果を得ることができる.

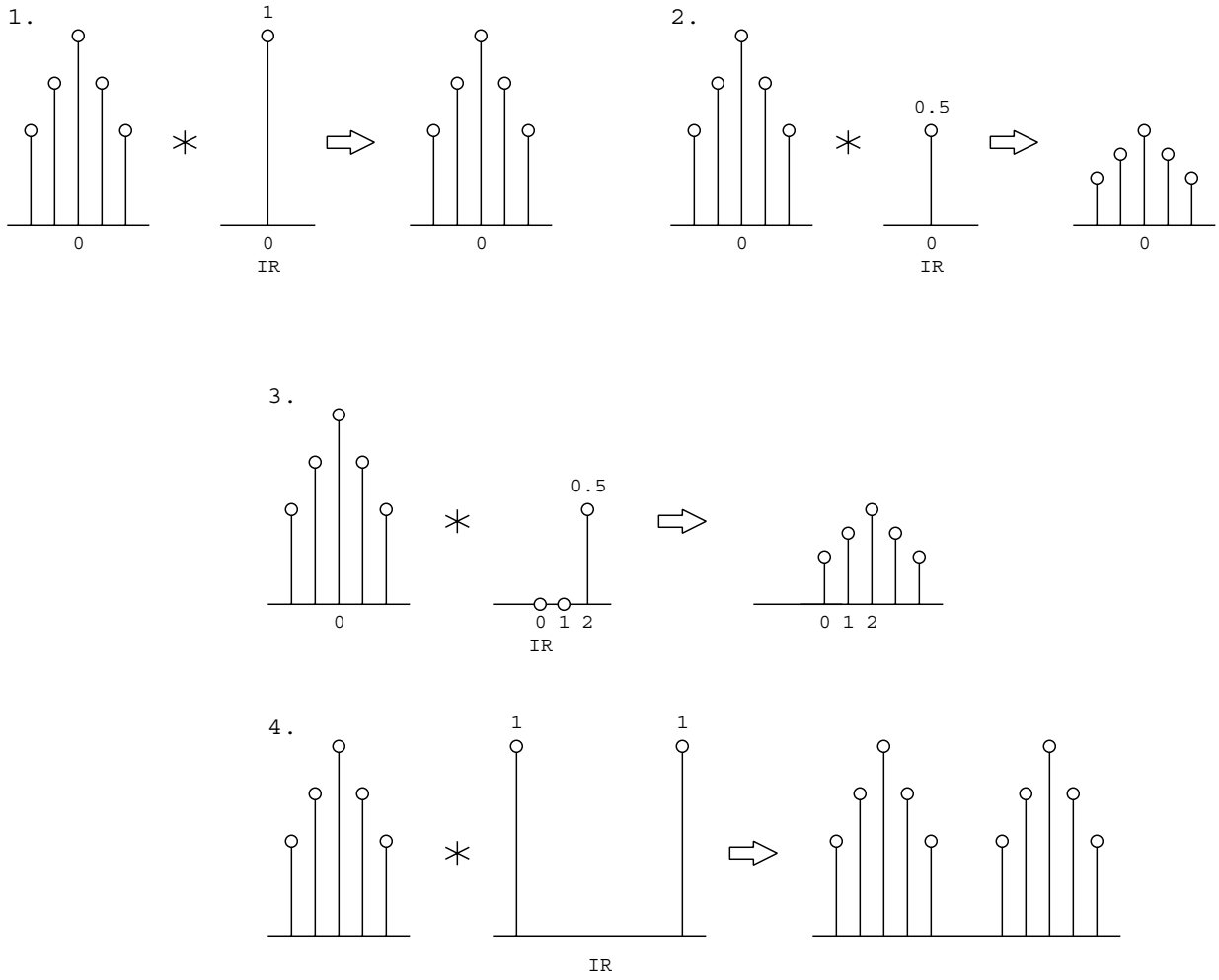


図 3.1: 畳み込みの典型的な例

### 3.1.2　畳み込みとフィルタリングの関係

二つの有限サンプル列に対する畳み込みの数学的定義を式 3.1 に示す.

$$a[N] * b[k] = output[k] = \sum_{i=0}^{N-1} a[i] \times b[k-i] \tag{3.1}$$

次に,3.2 節で述べる一般的な FIR フィルタの方程式を式 3.2 示す.

$$y[k] = \sum_{i=0}^{N-1} a[i] \times x[k-i] \tag{3.2}$$

$N$ は列 $a$ のサンプル長（フィルタのタップ長:フィルタ係数の個数）であり, 係数 $a[i]$ はインパルス応答（あるシステム, 空間等の持つフィルタの特性）に相当する. 列 $x$ は入力信号であり, 列 $y$ は列 $x$ が列 $a$ によってフィルタリングされた出力となる.

式 3.1 の畳み込みと, 式 3.2 の FIR フィルタは数学的に同じ式となっており, このことは, 畳み込みとフィルタリングは直接関係があると言うことができる. また, どのような FIR フィルタも畳み込みとして表現することができ, 逆もまた成り立つ.

上記の二つの式 3.1, 式 3.2 は共に, 列 $a$ の長さの連続した積和演算を行っている. この積和演算が畳み込みの基本となる.

## 3.2　一般的な FIR フィルタの構成

図 3.2 に FIR フィルタのブロック図を示す.

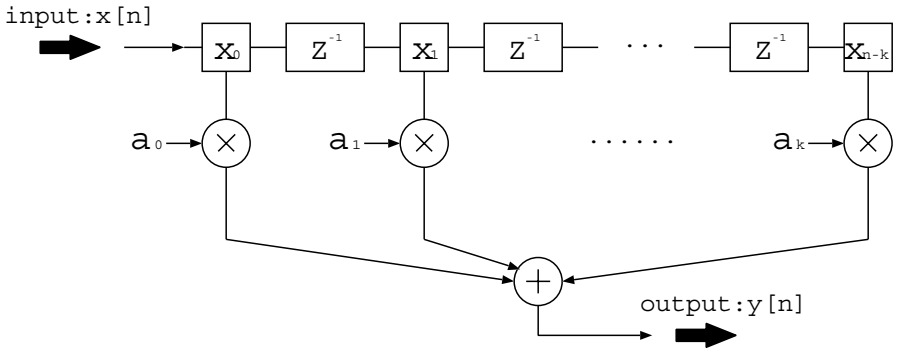


図 3.2: FIR フィルタのブロック図

図 3.2 中の $Z^{-1}$ は単位遅延回路を表し, その前後を 1 タイミングずつずらす（遅らせて伝える）役割を果たしている.

つまり, 入力データは 1 サンプル毎に $X_0$ に取り込まれるので, 次に新しいデータが $X_0$ に取り込まれる前にそれまでの $X_0$ の値を $X_1$ に移動させている. その結果,$X_1$ には常に 1 タイミング遅れて $X_0$ のデータが取り込まれることになる. 一般的に表すと,$X_{n-1}$ と $X_n$ の間にある $Z^{-1}$ の役割も同様に, この単位遅延の働きにより $X_n$ には常に 1 タイミング遅れて $X_{n-1}$ のデータが取り込まれることになる.

図 3.2 の動作を全体でみると, このフィルタは, 新しいデータが $X_0$ に流れ込んでくるたびに, 式 3.3 を計算して出力する働きをしている.

$$output = a_0 \times X_0 + a_1 \times X_1 + \ldots + a_k \times X_{n-k} \tag{3.3}$$

つまり, 入力データ $X$ の 1 サンプル毎にフィルタ係数 $a$ の個数回の連続した積和演算を行っている. そして, 入力データ $X$ も通常は連続したもの（2 章の図 2.3 の下部（Data））となる. そのため, 入力データ $X$ のサンプル数を $x$ 個, フィルタ係数 $a$ の個数を $A$ 個とすると, 入力データ $X$ の 1 サンプル毎に, フィルタ係数 $a$ のフィルタ個数回の積和演算を行うので, 総乗算回数は $x \times A$ 回となる.

この出力式 3.3 は 3.1.2 節で述べた一般的な FIR フィルタの方程式 3.2 を展開した形となっている. そのため, 式 3.1, 式 3.2, 式 3.3 の 3 式は同等の意味を持ち, このシステムは畳み込みの効果を得ることができる FIR フィルタを構成していることがわかる.

$X_0$ に入力されたデータ（インパルス）は, 遅延回路によって右側へ流れていき, 最終的にはこのシステムから消えていく. そのため, このシステムは FIR（Finite–Impulse–Response: 有限インパルス応答）フィルタと呼ばれている.

## 3.3 Digital Signal Processor（DSP）

本節では DSP（Digital Signal Processor）の概要と, その例について述べる.

### 3.3.1 DSP の概要

DSP（Digital Signal Processor）とは, ディジタル信号処理向けのプロセッサである.

1970 代後半から 1980 代初頭に第一世代の DSP が誕生した. 当時の汎用プロセッサは動作クロック周波数が 10MHz 程度と低く, 集積度も満足いくものではなかった. そのような技術制約の中で, 算術演算性能向上のために乗算器（積和演算器）を汎用プロセッサに組み込むことは不可能であった. そのため, 機能を絞った特定用途向けの専用プロセッサとして DSP が誕生した. 当時は音声信号のリアルタイム処理などが DSP の応用される分野であった.

現在では, 高度なディジタル信号処理が要求されるところに DSP は必要不可欠となり様々な用途で使用されている.

DSP の応用例としてあげられる携帯電話は, ディジタル信号処理技術なしには実現不可能である. 音声劣化の少ない音声データ圧縮・伸長, ノイズを低減するノイズキャンセラー, エコーを防止するためのエコーキャンセラーなど, 多くの処理が同時進行でリアルタイムに行われている.

他に例をあげると,HDD,DVD–ROM などのモータ制御, コピー機の画像処理,TV ゲームなどの民生家電製品, 自動車分野ではカーナビゲーション,GPS 装置, サスペンション制御, パワーステアリング制御など DSP の応用例は多数存在する.

3.1 節では, ディジタル音声信号処理を例に畳み込みについて述べた. しかし, 畳み込み, そしてその基本となる積和演算はディジタル音声信号処理に限らず, ここで例をあげた全てのディジタル信号処理の基本演算となる.

ディジタル信号処理向けのプロセッサである DSP は, ディジタル信号処理の基本となる積和演算をリアルタイムに効率良く処理することを目的として設計されている. そのため DSP は, 次のような独特のアーキテクチャを持っている.

- ハードウェア積和演算器

  乗算と加算を 1 マシンサイクルで実行するハードウェア積和演算器（Multiplier accumulator 以下 MAC）を内蔵している. また、汎用 DSP の多くのチップは、積和演算器とは別に加減算や論理演算を行う ALU を備えている.

- ハーバード・アーキテクチャ

  汎用プロセッサはメモリとのデータバスを 1 本だけ持つが、DSP はメモリをデータ用とプログラム用に分けてバスも分離している. このように, データ・メモリ・バスとプログラム・メモリ・バスを分ける構造を, ハーバード・アーキテクチャという. ハーバード・アーキテクチャは, データのアクセスとプログラムのアクセスを同時に行えるので高速処理が可能となる（日立社の SH 系の RISC チップように, 汎用プロセッサでもスピードを追求するために, ハーバード・アーキテクチャを採用しているものも存在する）.

さらに, 処理性能向上のために DSP には, 複数の演算ユニットの搭載, パイプライン処理などの汎用プロセッサの技術が多数使われている.

このように DSP が高機能・高性能化するにしたがって, そのアーキテクチャはより複雑なものとなっている. そして, 汎用プロセッサが高機能・高性能化するにしたがって DSP の専用性が低下してきている. 例をあげると, 汎用プロセッサも乗算器を搭載しているものが増えてきている. 現在 PC に搭載されている x86 系プロセッサも乗算器を内蔵しており DSP でなければできなかった音声データ圧縮・伸長（IP 電話）などの処理が可能となっている.

しかし, 一般的に汎用のプロセッサの入力となるのは複数の周期・非周期のイベントであり, それらのイベントに応じた処理を行うのが汎用プロセッサの役割となる.

それに対して DSP の入力となるのは連続したデータストリームであり, DSP はその入力に対してリアルタイムに処理を行うことを目的としている.

やはり, 汎用プロセッサは様々な処理を柔軟に行うプロセッサであり, 信号処理に関しては, 信号処理に特化したアーキテクチャを持つ DSP に劣るものとなる. また, その逆も言え, 信号処理に特化したアーキテクチャを持つ DSP は汎用的な用途を得意としない.

### 3.3.2 DSP の例:TMS320C5000 DSP

TMS320C5000 DSP（以下 C5000）は TI 社の DSP であり, 高性能、低消費電力を考慮して設計されたデバイスである. 命令を効率よく実行できるように, 以下のような計 4 本のバスを持つ.

- 命令フェッチのためのプログラム・バス：1 本
- 演算に必要な 2 つの引数を同時に読み出すためのデータ・バス：2 本
- 演算結果を書き戻すための書き込み専用データ・バス：1 本

これによってプログラムのフェッチ・サイクル時にプログラム・バスで命令をフェッチし, それと同時に 2 本のデータ・バスデータで 2 つのデータ・アドレスから 2 つのデータを取り込むため, データ処理のスループットが向上する.

C5000 シリーズはディジタル携帯電話, ネットワーク関連の製品やモデムなど幅広く使用されている. 動作クロック周波数は 100MHz 程度で 100MIPS 程度の処理が可能である.

## 3.4 MC に MAC を設けることの優位性

DSP は汎用のプロセッサと同様に, データを処理する演算ユニットとメモリから命令（プログラム）をフェッチ・デコードし DSP 自身を制御する制御ユニットを持つ.

本研究では,DSP の特徴となる演算ユニット, つまり MAC を MC に組み込むことを特徴としている.

1.1 節でも述べたが, PC 向けのディジタル信号処理の高速化の手法として, CPU とは別に DSP を設けて CPU の負荷を分散し処理の高速化を図る方法があるが,CPU と DSP の命令セットが異なるためプログラミングが困難であることや, CPU–DSP 間でのデータの受け渡しや, 処理の開始・終了の通知に要するオーバヘッドの発生が問題とされている.

また, 日立社の SH 系 CPU では,CPU に DSP を組み込むという方法も存在する. この方法では, 先ほど述べた問題点はある程度解消されるが, CPU の構造・設計が複雑なものとなる.

MC に MAC を組み込むことは,CPU との並列性を持つこととなる. このことは, 先に述べた 2 つの方法と同様に,CPU の負荷を分散しスループットを向上させる.

以下に,DSP を用いた手法に対する,MC に MAC を設けることの優位性を示す.

- DSP の持つ制御ユニットを持たず, 演算ユニット（MAC）のみを持ち連続した積和演算を行う. MC にフィルタ計算能力のみを持たせ, 制御ユニットを削除することによる構造の簡単化・ハードウェア量の削減を図る.
- メモリのバスクロックに合わせて演算を行うことで最高速度のフィルタ計算が可能となるため, 特別に高速な動作クロック周波数を必要としない.
- CPU の高速化・DSP を用いる手法と比べ, 構造・設計が単純なものとなる.

本研究ではメディアプロセッシングに着目しているが, 本章で述べたようにディジタル信号処理の応用分野は多岐にわたる. また, 大多数の CPU は,MC を用いてメモリアクセスを行う. その MC に MAC を設ける方式は, 演算能力の劣る組み込み向けの CPU や, 信号処理を得意としない汎用の CPU 等の様々な CPU を支援できるという可能性をもつ.

# 第4章　積和エンジン搭載高機能メモリコントローラ

本章では高速フィルタリングを行う積和エンジンを提案し, それを実現する MC と CPU の協調動作について述べる.

## 4.1　メモリコントローラ

本節では,4.2 節で述べる積和エンジンを組み込むメモリコントローラについて述べる. 図 4.1 に設計した MC のブロック図を示す.

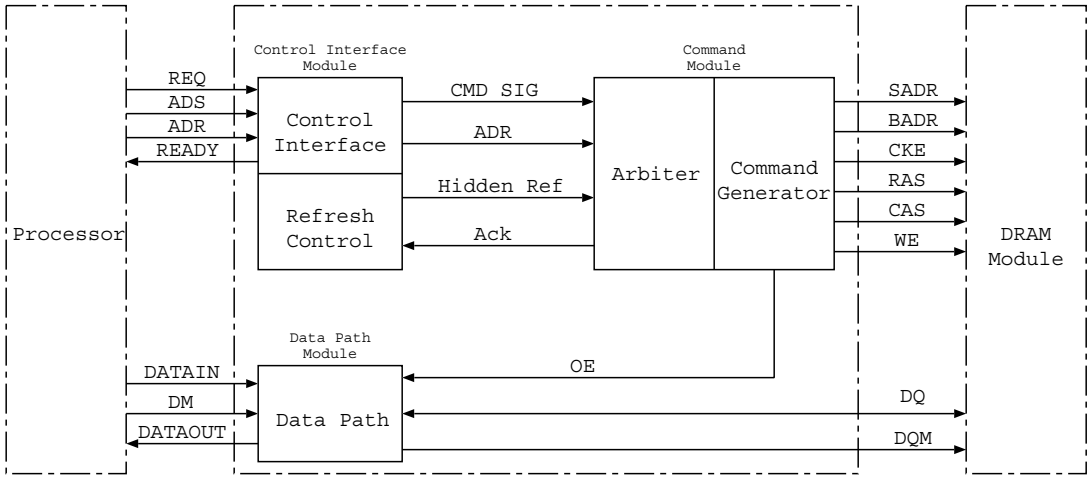


図 4.1: メモリコントローラのブロック図

- Control Interface Module

  Command Interface 部は, プロセッサから Read,Write などのメモリアクセス要求（REQ）と, アドレスストローブ信号（ADS）およびメモリアドレス（ADR）を受取

り,DRAM への動作要求とアクセスモード（4.3.1 節）を決定し,Command Module に伝える.

Refresh Control 部は,DRAM のデータ保持に必要なリフレッシュ動作を定期的に行うためのプログラマブルなダウンカウンタを持っている. そのカウンタにリフレッシュ期間をセットする. MC が動作中はバスクロック毎にカウンタ値はデクリメントし, カウンタ値が 0 になると Command Module に Hidden Ref（内部リフレッシュ要求）を自動的に伝える. Command Module から Ack を受け取ると Hidden Ref の送信を止め, カウンタ値はリセットされる.MC が動作していれば, 再びダウンカウントを始める.

- Command Module

  Arbiter 部は,Control Interface Module から送られてきたコマンドのデコードを行う. また, リフレッシュ要求と他の要求とのアービトレーションを行う. 他のコマンドの実行中にリフレッシュ要求を受けると, 実行中のコマンドが終了次第リフレッシュコマンドを DRAM Module に送る. 他のコマンド要求およびリフレッシュコマンドの実行中にリフレッシュ要求を受けると, リフレッシュ要求を優先させる. 他のコマンド要求はリフレッシュコマンドの実行終了まで待たされ, その後に実行される.

  Command Generator 部は,Arbiter 部からはデコードされたコマンド,Control Interface Module からはアクセスモードを受取り, DRAM Module への制御信号を生成し適切なタイミングで DRAM に伝える. メモリアドレス（SADR）は Row アドレスと Col アドレスを共有するため, それらのアドレスは RAS 信号と CAS 信号で時分割に区別される.

- Data Path Module

  プロセッサと DRAM Module 間のデータ転送を行う.OE（Output Enable）信号によってデータの流れる方向を決定する.

MC の基本仕様

- バースト長:BL（メモリがサポートする連続して入出力可能なデータ数）･･･1,2,4,8 に設定可能
- CAS レイテンシ:CL（Read 動作時に Col アドレス入力後, 実際にデータがメモリから読み出されるまでの必要バスクロックサイクル数）･･･2,3 バスクロックサイクルに設定可能
- 40 ビット・アドレス・バス（上位 8bit はロードとストアの代替命令で提供されるアドレス空間識別しとして使用（4.3.1 節））
- 64 ビット・データ・バス

## 4.2 積和エンジン

本節では,4.1 節で述べた MC に組み込む積和エンジンの構成について述べる.

図 4.2 に乗算器が 2 つの場合の積和エンジンのブロック図を示す.

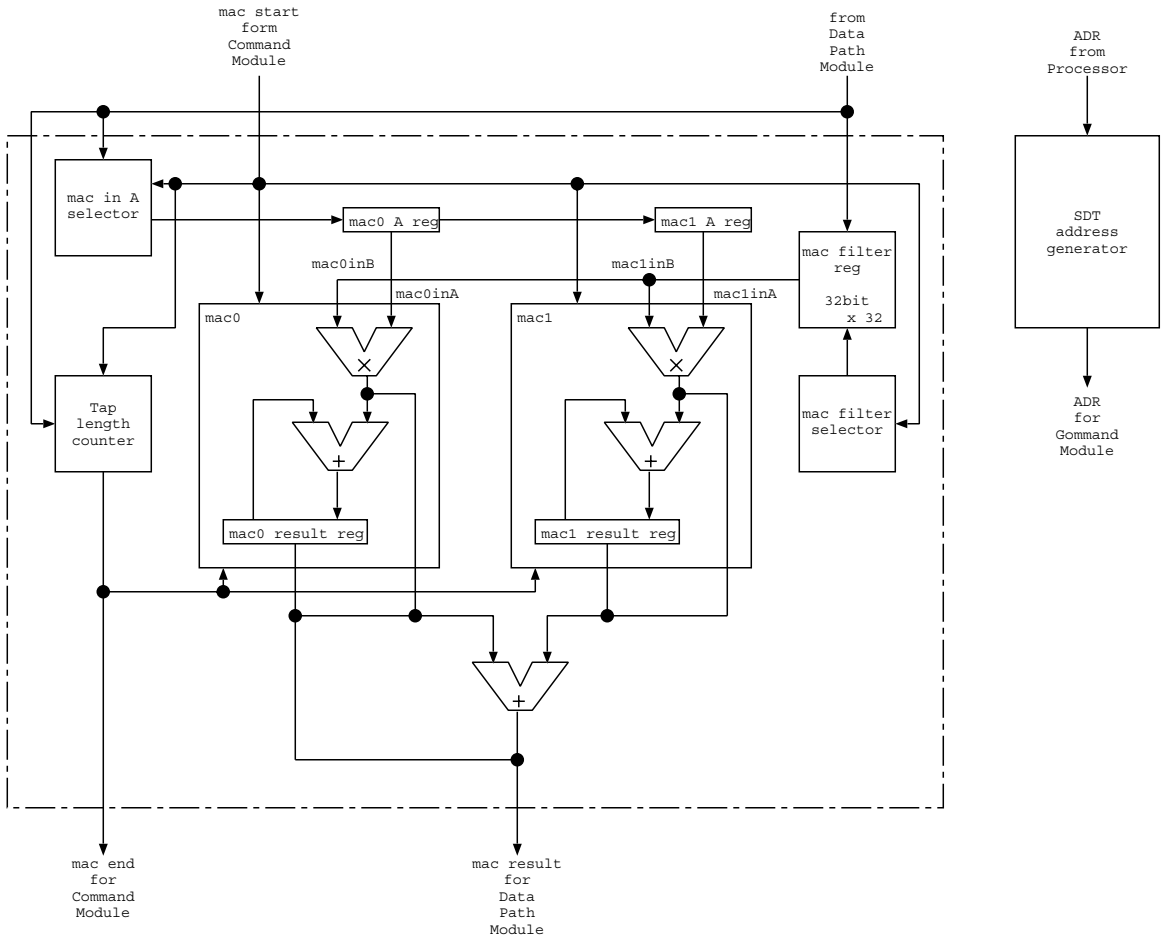


図 4.2: 積和エンジンのブロック図（乗算器が 2 つの場合）

積和エンジンとは,“大規模な連続した入力データに対し, 連続した積和演算を行うもの”とし, DSP の特徴である積和演算器を持ち, 連続した積和演算だけを専用に行う.

構造の簡単化・ハードウェア量の削減を目的とし,DSP のように命令のフェッチ・デコードは行わない.

- mac filter reg

  mac filter 番号を 0 から 31 とする 32 ビットのレジスタを 32 本備えている. 事前に, 積和演算を行う際のフィルタ係数を格納しておく. キャッシュスルー命令でフィルタ係数のセットを行う（4.3.1 節）.

- mac filter selector

  積和演算実行の際に mac filter reg に格納されたフィルタ係数の選択を行う.

- mac in A selector

  4.1 節で述べた MC は 64 ビット・データ・バスであるため, 64 ビット・データ・バスのデータの内, どのビットを mac に与えるかを選択する. mac に与えるデータのビット幅は, 事前にキャッシュスルー命令でセットしておく（4.3.1 節）.

- Tap length counter

  フィルタ係数の数（タップ長）を, 事前にキャッシュスルー命令でセットしておく（4.3.1 節）. mac 動作中にデクリメントしタップ長分の積和が終了すれば,mac end を発行する.

- mac

  積和演算器である. この mac は入力を受け取ってから 2 クロックで結果を返す設計となっている. 連続した入力を与えた場合, 最初の結果を返すのに 2 クロック必要となるが, それ以降はパイプライン処理を行い, 見掛け上 1 クロック毎に連続した結果を得ることができる.

  ここでは mac が 2 つの場合についての例を述べていくが, 用途やハードウェア量の制約に合わせて,mac 数を決定し設計するものとする.

- SDT address generator

  このモジュールは積和エンジンとは異なるものだが,4.1 節で述べた MC の追加機能となるのでここで説明する.

  SDT 動作時にアドレスの加算を行い, 一定間隔値を加算したアドレスを Command Module へ送る.

本設計は畳み込み演算を専用に行うものとし,mac への二つの入力は互いに逆行し合うものとしている. mac filter reg からは,mac filter 番号 0 からインクリメント方向にレジスタが選択され, 格納された値が連続して mac in B に入力されていく. DRAM からは, 指定されたアドレスから SDT を利用してデクリメント方向に Col アドレスを指定し, 読み出された値が mac in A に入力されていく.

4.1 節で述べた MC に, 本節で述べた追加機能を組み込んだ, 提案する高機能メモリコントローラを図 4.3 に示す.

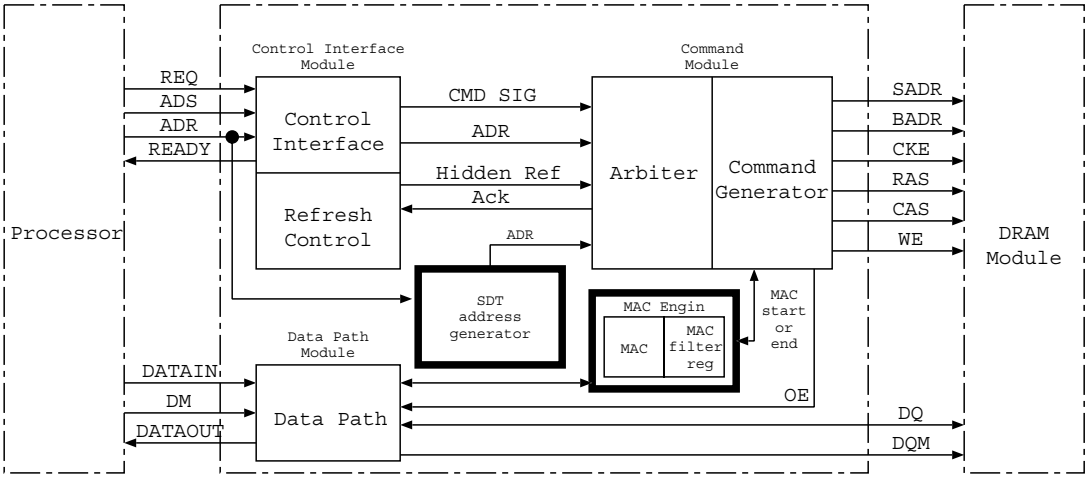


図 4.3: 提案する高機能メモリコントローラのブロック図

## 4.3 プロセッサとMCの協調動作

### 4.3.1 メモリアドレス形式

SDT および FIFO バッファを使用する MAC 動作を行うメモリアクセス要求を検出するために, MC は物理アドレスのフィールドを使用する. 通常, メモリアドレスのフィールドは Byte Offset を除いた下位ビットから Col,Row,Bank アドレスで構成され, 残りの上位ビットはシステム依存となっている.

ここでは,40 ビット・アドレス・バスの上位から 9 ビット目のビットをメモリアクセスモードの指定フィールド（AMODE）とし, AMODE が 0 のときは通常のメモリアクセス, AMODE が 1 のときは SDT および FIFO バッファを使用するメモリアクセスを行うものとする. 図 4.4 にメモリアドレス形式を示す.

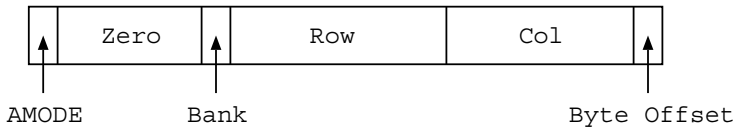


図 4.4: メモリアドレス形式

表 4.1: ロードとストアの代替命令

| 上位 8 ビット | 動作 |
|---|---|
| 0x48 | SDT データ転送数（タップ長）を MC のレジスタにセット |
| 0x49 | SDT 方式のストライド幅（データ間隔値）を MC のレジスタにセット |
| 0x4A | SDT 方式で扱うデータサイズ（32 ビットのみサポート）を MC のレジスタにセット |
| 0x4B | mac filter reg の値を MC のレジスタにセット |
| 0x4F | 提案方式のデータ転送を実行（現在は 4.3.4 節の動作のみサポート） |

また,40 ビット・アドレス・バスの上位 8 ビットは, ロードとストアの代替命令で提供されるアドレス空間識別しとして使用している. 表 4.1 に本提案での使用方法をまとめる.

MC は AMODE フィールド以上の上位のビットを除いた Col,Row,Bank アドレスで DRAM アクセスを行う.

### 4.3.2 通常のリードリクエスト動作

通常, プロセッサのリードリクエストは命令実行中に命令キャッシュミス, ロード・ストア命令によるデータキャッシュミスによってメモリアクセスが発生する. この時のプロセッサのリクエストからデータを読み出すまでのタイミング波形を図 4.5 に示す.

リードタイミングの波形

1. プロセッサからアドレスストローブ信号（ADS）, メモリアドレス（ADR）およびリードリクエスト（REQ）がシステムバスを通して MC に送られる.

2. MC がこれらの信号をデコードし,Bank アドレス（BADR）,Row アドレス（SADR）, および RAS 信号を生成し DRAM にアクセスする. DRAM 内の制御信号によって,DRAM 内のメモリアレイの該当する Row データがラッチされる.

3. CAS 信号と共に Col アドレス（SADR）を与える.

4. Col アドレスを与えてから CL 後に,DRAM から該当するデータが DQ を通して読み出され,MC に転送される.

5. MC は読み出された 4 つ分のデータを DATAOUT を通してプロセッサに転送し, その後処理を終了する. READY 信号により, プロセッサにデータ転送を通知する.

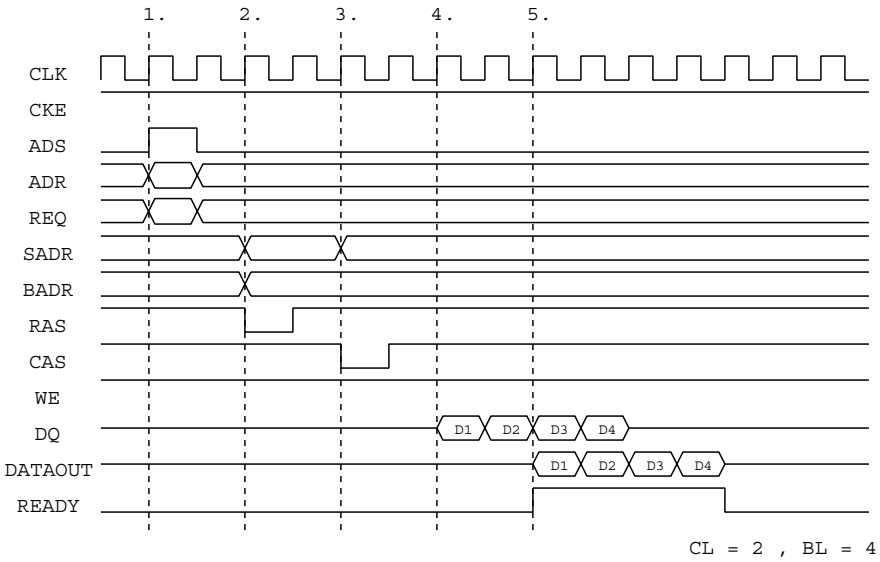


図 4.5: 通常のリードタイミング波形

### 4.3.3 SDT 方式と FIFO バッファの協調動作

SDT 転送は一定間隔に存在するデータを連続転送するため,SDT address generator でメモリアドレスを自動生成し DRAM アクセスを行う（2.2 節）. このときのプロセッサと MC の動作を述べ, そのタイミング波形を図 4.6 に示す.

SDT の実現方法は以下となる,

1. プロセッサはキャッシュスルー命令で MC のレジスタに,転送するデータサイズ[1], データ数およびデータ間隔値を格納する.

2. プロセッサは AMODE フィールド（4.3.1 節）を “1” とするアドレスによりメモリリクエストを発行する.

3. MC は受け取ったアドレスの AMODE フィールドの判断別後, SDT の開始アドレスとして受け取ったアドレスを MC 内のレジスタに記憶し, 開始アドレスのデータに対し通常にメモリアクセスを行う.

4. MC は記憶したアドレスにデータ間隔値を加算し, 次アドレスを生成する.

[1]データサイズを格納することにより任意のデータサイズの SDT が可能となるが, 本設計ではこれを簡単化のため省略し, SDT でアクセスするデータサイズを 32 ビットに限定する.

5. MC は Col アドレスを指定し, 初回のメモリアクセスでラッチした Row に対し一定間隔毎のデータ読み出しを行い, Data Path を通してプロセッサにデータを連続して転送する. プロセッサ内の再構成可能なキャッシュの一部を FIFO バッファとして利用し, 転送されたデータが順次格納される（2.3 節）.

6. MC は以下の条件でデータ転送を終了する.

   - セットしたデータ転送数を越えた場合
     正常終了
   - データ間隔値の加算によって Bank,Row アドレスが変化した場合
     SDT は同一 Bank,Row 内の連続転送に限るため, どちらかのアドレスが変化した場合は, Bank,Row アドレスを与え直すことで SDT を再開する
   - 仮想記憶のページ境界を越えた場合
     仮想アドレス空間に対する物理アドレスの連続性が保証されないため, この場合もアドレスを与え直すことで SDT を再開する

次に, 図 4.6 について述べていく. MC のレジスタ内に, 転送データ数およびデータ間隔値を格納する部分の波形は省略した. 転送データ数を 5, データ間隔値を 10 として既にセットされているものとする.

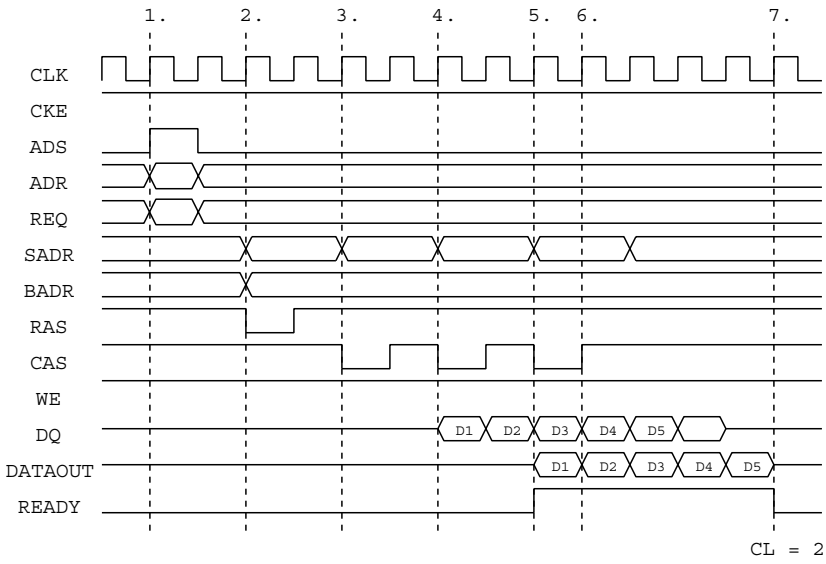


図 4.6: SDT 方式のタイミング波形

SDT 方式のタイミング

1. プロセッサからアドレスストローブ信号（ADS）,AMODE フィールドを “1” とするメモリアドレス（ADR）およびリードリクエスト（REQ）がシステムバスを通して MC に送られる.

2. MC がこれらの信号をデコードし,Bank アドレス（BADR）,Row アドレス（SADR）, および RAS 信号を生成し DRAM にアクセスする. DRAM 内の制御信号によって,DRAM 内のメモリアレイの該当する Row データがラッチされる.

3. 2 バスクロックサイクル毎[2] に CAS 信号と共に, セットされている間隔値（ここでは 10）を加算された Col アドレス（SADR）の指定によって DRAM アクセスを行う.

4. Col アドレスを与えてから CL 後に,DRAM から該当するデータが DQ を通して読み出され,MC に転送される.

5. MC は読み出された 5 つ分のデータを DATAOUT を通してプロセッサに転送する.

6. MC にセットされている転送データ数に達したので Col アドレス指定を終了する.

7. MC にセットされている転送データ数（ここでは 5）個のデータを転送後に終了する. READY 信号により, プロセッサにデータ転送を通知する.

次に SDT 転送中に, 命令あるいはデータキャッシュミスによるメモリアクセスが発生し, SDT を一時中止してから再開するまでのタイミング波形にを図 4.7 示す.

MC のレジスタ内に, 転送データ数およびデータ間隔値を格納する部分の波形は省略した. 1. から 5. までの SDT 開始までの波形は図 4.6 と同様となる.

SDT 方式の一時停止

6. では, プロセッサは SDT の初回データ D1 を使用して命令実行を開始する. その実行途中に命令あるいはデータキャッシュミスによるリード命令が発生したため, アドレスストローブ信号（ADS）,AMODE フィールドを “0” とするメモリアドレス（ADR）およびリードリクエスト（REQ）を MC に送る. このとき MC 内部では,SDT による Col アドレスの連続入力により DRAM からデータが読み出されているが, データ D4,D5 はプロセッサには転送せず SDT を一時停止する.

[2]設計した MC のデータ・バス幅 64 ビットとなっている. 本設計では SDT で扱うデータサイズを 32 ビットと限定している. 1 回の Col 指定で DRAM から出力される 64 ビットであり,32 ビットのデータを 2 個ずつ取り出すことができる.

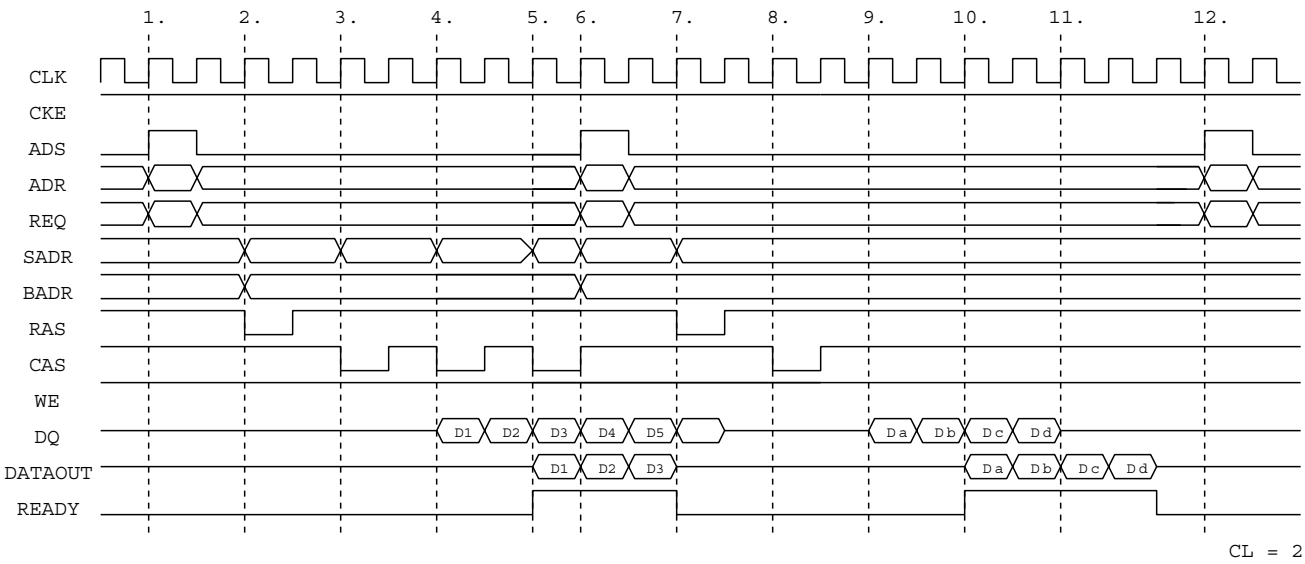


図 4.7: SDT 方式のタイミング波形（SDT の一時中断）

7. から 9. では,6. の時点で発生したキャッシュミスによるリード命令を実行している. この間, プロセッサはストールしている. MC がリード命令に対するアドレス（SADR,BADR）と共に RAS 信号あるいは CAS 信号を DRAM に与えて, キャッシュの 1 ブロック数分のデータがバスクロックの 1 サイクル毎に読み出される.

10. では, 先ほどのリード命令でリードしたデータ $D_a$ が MC からプロセッサに読み出される.

11. では, そのデータ $D_a$ を使用してプロセッサのストールを解消し, 再び命令実行を開始する. この時点で,FIFO 内にはデータ D2,D3 が格納されており, プロセッサの命令実行で FIFO 内が空になるまでそれらのデータを消費する.

12. では,FIFO が空になったため, プロセッサは D4 に対する再リクエスト（ADS,ADR,REQ）を発行して SDT を再開する.

### 4.3.4 積和エンジンを含めた協調動作

本節では, 本研究の提案手法である SDT を用いた積和エンジンの動作について述べる.

#### 積和エンジンの基本動作

基本動作を行う場合の積和エンジンの構成は, 図 4.8 の実線部となる.

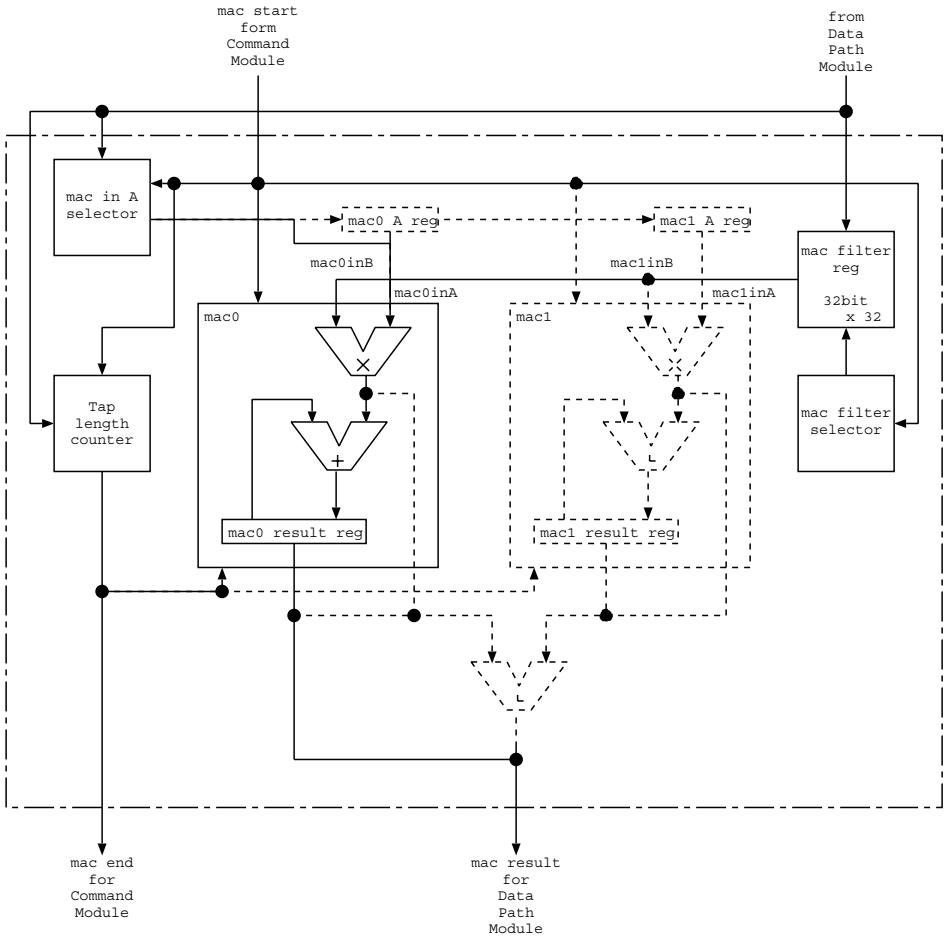


図 4.8: 積和エンジンの基本動作

基本動作時のタイミング波形を図 4.9 に示す.

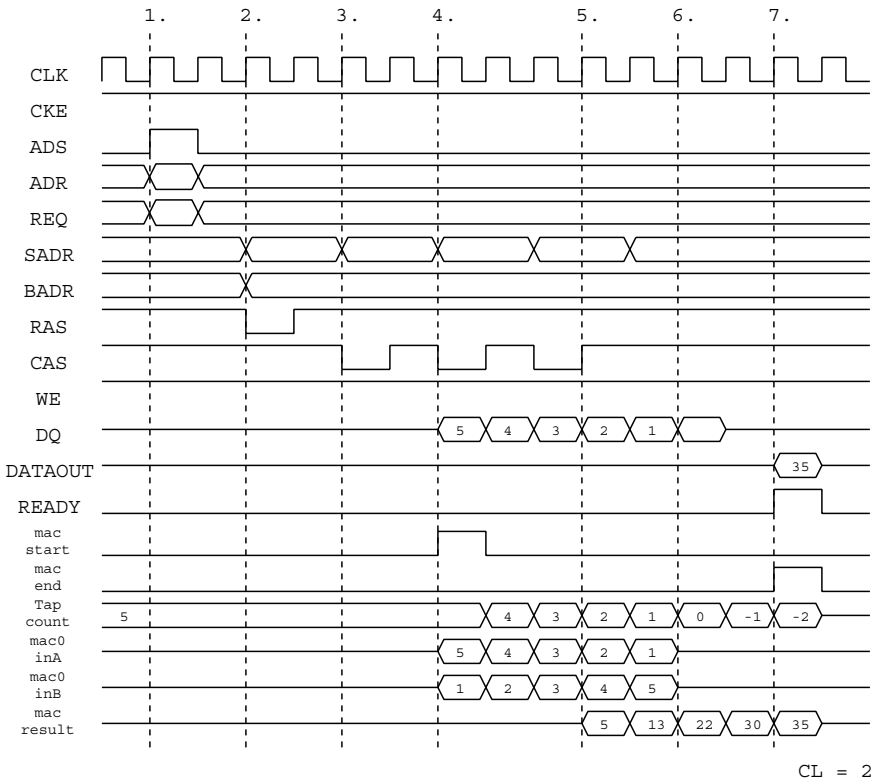


図 4.9: 積和エンジン基本動作時のタイミング波形

基本動作

MC のレジスタ内に, 転送データ数およびデータ間隔値を格納する部分の波形は省略した. 転送データ数を 5, データ間隔値を 1 として既にセットされているものとする. また,DRAM と mac filter reg に表 4.2, 表 4.3 の値がセットされているものとする.

基本動作時のタイミング波形（図 4.9）について述べていく.

1. プロセッサからアドレスストローブ信号（ADS）, 上位 8 ビットを “0x4F” 及び AMODE フィールドを “1” とするとしたメモリアドレス（ADR）14 へのアドレス指定, およびがリードリクエスト（REQ）がシステムバスを通して MC に送られる.

2. MC がこれらの信号をデコードし,Bank アドレス（BADR）,Row アドレス（SADR）, および RAS 信号を生成し DRAM にアクセスする. DRAM 内の制御信号によって,DRAM 内のメモリアレイの該当する Row データがラッチされる.

表 4.2: DRAM にセットされている値（積和エンジン基本動作）

| アドレス | 値 |
|---|---|
| 10 | 1 |
| 11 | 2 |
| 12 | 3 |
| 13 | 4 |
| 14 | 5 |

表 4.3: mac filter reg にセットされている値（積和エンジン基本動作）

| mac filter reg 番号 | 値 |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |

3. SDT 機能により,2 バスクロックサイクル毎に CAS 信号と共に, セットされている間隔値（ここでは 1）を加算された Col アドレス（SADR）の指定によって DRAM アクセスを行う.

4. Col アドレスを与えてから CL 後に,DRAM から該当するデータが DQ を通し連続して読み出され,MC に転送される. そのとき mac start 信号がアサートされ,DRAM に入力値としてセットされていた値が Data Path から mac0inA へ, フィルタ係数としてセットされていた値が mac filter reg から mac0inB にデータが入力される. ここでは,mac0 で畳み込み演算を行うものとする.mac filter reg はインクリメント, DRAM アドレスはデクリメントして得られた値が mac0 に入力される. そして,Tap length countor（データ転送数がセットされている）もデクリメントを始める.

5. MC にセットされている転送データ数に達したので Col アドレス指定を終了する. また,mac0 の最初の入力に対する結果が得られる.mac は連続した入力を受取りパイプライン処理を行うため, 連続した結果出力を行う.

6. Tap count が 0 となり, タップ長分の入力が終了したため,mac0 の入力を終了.

7. mac0 の最後の入力までの畳み込みの結果が得られる. READY 信号により, プロセッサにデータ転送を通知し, 結果を転送する.

ここでは,READY 信号により, プロセッサにデータ転送を通知し, 一つの畳み込み結果を転送する方法を取っているが, タップ長分の積和演算結果ができ次第, 再構成可能なキャッシュの FIFO バッファに畳込み結果を書き込む方法もある.

そして, 連続ではないが複数の畳み込み結果を得る方法も存在する. 図 4.9 の動作は 1. での 1 つのメモリアドレス指定に対する, SDT を利用した mac による畳み込みを行い,1 つの畳み込み結果を返していた. それに対して,SDT 機能を改良し 2 重ループの SDT を行う方法を用いると,1. でのメモリアドレス指定に対しても SDT 方式でのメモリアクセスを行い, 上記の動作と同様に SDT を利用した mac による畳み込みを行う. これにより 1. での一度のアドレス指定で, 連続ではないが複数の畳み込み結果を得ることができる.

### タップ長が乗算器数以内の動作

タップ長が乗算器数以内の動作を行う場合の積和エンジンの構成は, 図 4.10 の実線部となる.

図 4.10 は, 図 3.2 で示した FIR フィルタのブロックと同様の構成をとり, 同様の動作を行う. 各々mac の in B はフィルタ係数を固定し,mac in A は SDT による連続データ転送により DRAM から連続入力データストリームを受取る. 連続出力となるデータストリームは, 再構成可能なキャッシュの FIFO バッファに書き込んでいく.

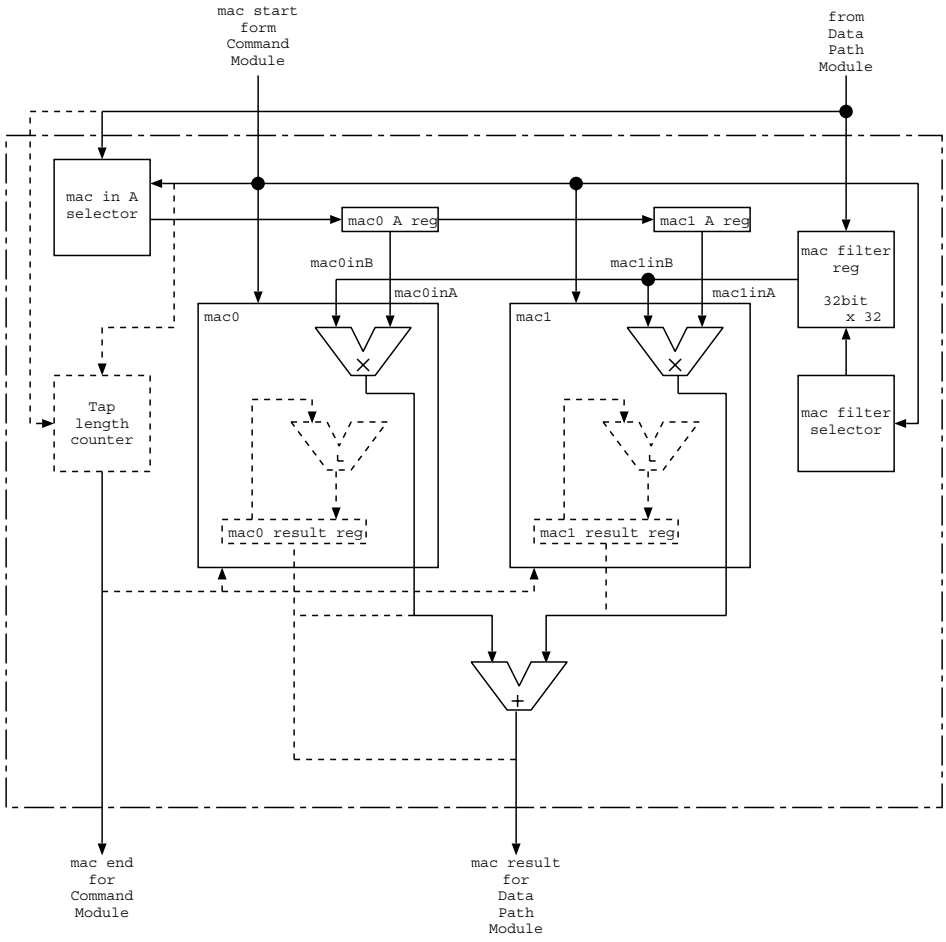


図 4.10: 積和エンジンのタップ長が乗算器数以内の動作

# 第5章　性能評価

本章では,VHDL で設計したメモリコントローラの基本性能評価を RTL シミュレーションにより行う. また, 論理合成を行うことにより, 設計した回路のハードウェア量の見積もりを行う.

## 5.1　性能評価

### 5.1.1　シミュレーション環境

プロセッサには *PRESTOR–1*[7] の RTL 記述を用いる. *PRESTOR–1* の特徴および構成は以下の通りとなる.

- 10 ステージ実行パイプライン
- 命令/データキャッシュサイズは 16KB/16KB
- キャッシュブロックサイズは 32B
- キャッシュの連想度は 4 ウェイセットアソシアティブ
- 外部入出力バスは 64 ビット

また,MC に関して以下の設定を使用した.

- バスの動作周波数はプロセッサの 1/10 倍（プロセッサ：100MHz,MC：10MHz）
- MAC の乗算器にはプロセッサと同性能のものを使用（動作周波数に 10 倍の差があるため, 乗算結果を得るのにプロセッサの 10 倍の時間を必要とする）
- プロセッサからのメモリリクエストのストローブタイミングに対して, 第一データの応答は 10 バスクロックサイクル

なお, シミュレーションには ModelTechnology 社の ModelSim を使用した.
評価の方法は 5.1.3 節に示すプログラムの実行速度を比較することにより行う.

### 5.1.2 評価対象

比較対象は以下とする

- プロセッサのみでの演算

  提案機構を持たない, 従来の MC を使用した際の実行速度（演算を行うのはプロセッサのみとなる）

- SDT 方式を用いた MC 内の mac での演算（提案方式）

  提案機構を持った,MC を使用した際の実行速度（ここで行う評価は基本性能評価のみとし,4.3.4 節で示した, 積和エンジンの基本動作を行った際の実行速度を示す.）

### 5.1.3 評価プログラム

評価に用いた基本的な C プログラムを下記に示す. フィルタ長 32 の畳み込み演算を行い, 演算結果を一つの返すプログラムとなっている.

```
int main(void)
{
  int i,j,k;
  int f[32]; /* 畳み込み演算のフィルタ係数をセット */
  int m[32]; /* 入力信号の値をセット */
  int result = 0; /* 畳み込み結果 */

  for ( i = 0 ; i < 32 ; i++ )  /*フィルタ係数として,1 から 32 の値をセット */
    {
      f[i] = i + 1;
    }

  for ( j = 0 ; j < 32 ; j++ )  /* 入力信号の値として 1 から 32 をセット */
    {
      m[j] = j + 1;
    }

  for ( k=0 ; k<31 ; k++ ) /* 畳み込み */
    {
      result += f[k] * m[31-k];
    }
```

```
  return result;
}
```

- 1 つめの引数として, メモリ上に入力データを配置し, メモリからその値を読み込み, 演算を行う.
- 2 つめの引数として, フィルタ係数を配置し, その値を読み込み, 演算を行う.
  - プロセッサのみでの演算 : フィルタ係数もメモリ上に配置
  - 提案方法 : フィルタ係数を MC 内のレジスタに配置

実際に使用したアセンブリプログラムを巻末の付録としている. 基本的には上記の C プログラムをアセンブリ化したものであるが, 若干の修正を加えている. 修正個所は以下となる.

- プロセッサのみでの演算

  データアクセスの際のキャッシュミスを防ぐために, ポインタによりメモリ上のデータを指し示すようにしている.

- 提案方式での演算

  提案方式を実現するために, 演算を行う前処理として, 転送データ数, データサイズを MC 内のレジスタに格納する命令を追加している.

  アセンブリコードの.LL8 以降で畳み込み演算を行っているが, 提案方式では.LL8 内の 1 回の lda 命令で連続した積和演算を行うため, 比較対象であるプロセッサのみの演算の比べ命令数が減少している.

### 5.1.4 実行結果

アセンブリコードのラベル.LL8 以降で畳み込み演算を行っている. ラベル.LL8 を計算の開始とし, 計算の終了までに各々の評価対象が要したクロック数は（プロセッサの動作周波数を基準とする）, 表 5.1 となる.

表 5.1: タップ長 32 の実行結果

| 評価対象 | 計算に要したクロック数 |
|---|---|
| プロセッサのみ | 3739 clock |
| 提案する方式 | 631 clock |

提案方式による性能向上は

$3739clock \div 631clock =$ 約 5.93 倍
計算に要したクロック数の差は
$(3739clock - 631clock) = 3108clock$
となる.

### 5.1.5 実行結果の考察

本節では 5.1.4 の実行結果に対して, キャッシュミスペナルティ及び命令実行数の減少についての考察を行う. プロセッサの動作周波数を基準として考察を行う.

#### キャッシュミスペナルティ

プロセッサのみ
プロセッサのみの計算によるキャッシュミスペナルティは, 表 5.2 となる.

表 5.2: プロセッサのみの計算によるキャッシュミスペナルティ

| ミス分類 | 時間 |
|---|---|
| 命令キャッシュミス | 306500 – 307600 ns |
| 命令キャッシュミス | 308000 – 309100 ns |
| 命令キャッシュミス | 309500 – 310600 ns |
| データキャッシュミス | 311000 – 312100 ns |
| データキャッシュミス | 312500 – 313600 ns |
| 命令キャッシュミス | 314000 – 315100 ns |
| データキャッシュミス | 319800 – 320900 ns |
| データキャッシュミス | 321300 – 322400 ns |
| データキャッシュミス | 327400 – 328500 ns |
| データキャッシュミス | 328900 – 330000 ns |
| データキャッシュミス | 335000 – 336100 ns |
| データキャッシュミス | 336500 – 337600 ns |
| 命令キャッシュミス | 342500 – 343600 ns |

ミスペナルティとして必要なクロック数は表 5.2 より,110clock となっている.
プロセッサのみの計算によるキャッシュミスペナルティは,
(命令キャッシュミス回数 + データキャッシュミス回数) $\times 110clock = (5+8) \times 110clock =$
$1430clock$
となる. 計算のために必要なデータが CPU キャッシュになくなる度に, メモリアクセスを行い CPU キャッシュに必要なデータを取り込むためにキャッシュミスが発生する. 必要

となるデータが多くなるほど, キャッシュミスの回数は増加する.

提案手法

提案手法での計算によるキャッシュミスペナルティは, 表 5.3 となる.

表 5.3: 提案手法での計算によるキャッシュミスペナルティ

| ミス分類 | 時間 |
|---|---|
| 命令キャッシュミス | 302410 – 303510 ns |
| データキャッシュミス | 303630 – 304730 ns |

ミスペナルティとして必要なクロック数は表 5.2 より, （プロセッサのみの場合と同様に）110clock となっている.

提案手法の計算によるキャッシュミスペナルティは,

(命令キャッシュミス回数 + データキャッシュミス回数) $\times 110clock = (1+1) \times 110clock = 220clock$

となる. 提案手法では MC 内の MAC で積和演算を行うため, 演算のためのデータを CPU キャッシュに取り込む必要がないために, CPU のみの計算に対してキャッシュミス回数が減少する.

キャッシュミスペナルティ改善による性能向上

ここまでで示したように, プロセッサのみでの計算に対して提案手法での計算は

$(1430clock - 220clock) = 1210clock$

のミスペナルティを削減している.

表 5.2 に示すように, アクセスするデータが増加するほどキャッシュミス回数は増加している.

### 命令実行数の減少

プロセッサのみでの計算を行う場合, アセンブリコードのラベル.LL11 – .LL14 – .LL13 – .LL11 のループを 32 回行っている.

そのループでキャッシュミスを含まないループは 60clock で 1 ループ分の処理を行っている.

その 60clock の内,10clock で ld – ld – mul – ld – add の積和演算に相当する演算を行っている. 提案手法でも 1 つの積和演算に 1 バス・クロック・サイクル（=10CPU・クロック・サイクル）必要となるため, 1 つの積和演算を行うのに必要なクロック数は同等となっている.

プロセッサのみでの計算を行う場合, 残りの 50clock がループ等の制御に必要となっている. 提案手法では, この 50clock 分を必要としないため,

$(50clock \times 32 \text{ 回}) = 1600clock$

のクロック数を削減している事になる.

**まとめ**

- 本シミュレーション環境で, プロセッサが一度に読み込めるデータ量は, キャッシュのブロックサイズの 32B である. 評価プログラムでメモリにセットした値は 1 データあたり 4B（int）であり, プロセッサは一回のメモリアクセスで最大でも 8 個のデータまでしか読み出せない. そのために 8 個以上のデータの扱おうとする際に, キャッシュミスが発生している. （プロセッサのみの計算では, 入力値 32 個+フィルタ係数 32 個の合計 64 個分のデータをメモリから取得している.）

  プロセッサは必要なデータがキャッシュになくなるたびに, メモリアクセスを行い, キャッシュミスによるペナルティーが発生する. このような状況は, メディアプロセッシングで扱うような時間的, 空間的局所性のない連続したデータにアクセスする際に発生しやすい.

- 提案方式は, 直接メモリから連続したデータを取り込み演算を行うことができる. プロセッサのキャッシュのブロックサイズより, 連続してより多くのデータを取り込むことができる. 本シミュレーション環境では、プロセッサが一度に読み込める最大データ量 32B なのに対し, 本提案では 最大タップ長 $\times$ $SDT$ データサイズ, つまり $32 \times 4B = 128B$ の連続した入力に対する積和演算が行える. つまり, 本評価の対象となった積和エンジンの基本動作においては, タップ長が長いほど効果が得られると言える.

- その他, 制御等に要したクロック数の削減も図れている. 提案方式による積和エンジンの起動方法により命令数が減少している.

## 5.2 ハードウェア量

Synopsys 社の Design Compiler を使用して論理合成を行った. 表 5.4 にその結果を示す.

表 5.4: ハードウェア量

| モジュール | ゲート数 |
|---|---|
| メモリコントローラのみ | 8,836 ゲート |
| 積和演算器（1 つあたり） | 14,547 ゲート |
| その他の追加機能 | 40,290 ゲート |

追加機能の合計をメモリコントローラのみのハードウェア量と比較すると,7 倍近くにもなっている. しかし, 現在の LSI の集積度を考慮すると, 合計で 6 万ゲートいうハードウェ

ア量は十分に実現可能と言える.

また,DSP は複雑な機構を持つためにハードウェア量も大きくなってしまう. 組み込みシステム分野で注目されているシステムオンチップ（SoC）への利用を考慮すると,DSP はハードウェア量が大きく Soc 向けではないが, 本提案方式はハードウェア量が小さいため SoC にも利用価値があると言える.

# 第6章　関連研究

本章では，主記憶内のストリーミングデータをプロセッサへ転送する際の，プロセッサ側の受信バッファ方式，およびメモリコントローラによる連続データ読み出し方式についての関連研究，および本論文で使用した方式との比較を述べる．

## 6.1　データ受信バッファ

Stream buffer [8, 9] はプロセッサがストリーミングデータを受け取る機構である．Stream buffer はデータキャッシュとは分離したメモリであり，したがってストリーミングデータを扱わないアプリケーションの実行ではそのバッファのためのメモリ資源が有効利用されず，ハードウェア効率が悪い．さらに，プログラムがストライドデータ列を参照する場合，この機構はそのアクセスがストライドアクセスであることを認識可能であるが，バッファへの挿入はキャッシュブロックサイズが基本単位であるため，バッファ内で実際に参照されないデータが存在するため，やはりメモリ資源が有効利用されない．

一方，本研究で使用する再構成可能キャッシュメモリによる FIFO バッファは，ソフトウェアによる再構成制御が可能であり，ストリーミングデータを扱わないアプリケーションの実行ではメモリ資源が全て通常のデータキャッシュとして使用されるため，メモリ資源の浪費が無い．また，バッファへの挿入はデータ単位であるため，実際に参照されるデータのみがメモリ資源を使用することになる．これにより，メモリ資源の有効利用が達成されることが特徴である．

## 6.2　連続データ転送方式

主記憶内のストリーミングデータに対して，メモリコントローラが特別な機能を持つことにより高速な連続読み出しを行う機構として Impulse [10] and SMC [11] がある．

ストライドデータ列のような空間的局所性を持たないデータ列に対して，Impulse ではストライドデータ列をパッキングして連続アドレスで参照可能とするエイリアス配列を実行プログラム中で定義することを仮定している，このエイリアス配列のアドレスがメモリコントローラ内で変換され，実際のデータの物理アドレスで主記憶がアクセスされる．この方法はプログラマに負担が大きく，また通常よりアドレス変換の階層が多いためオーバヘッドが大きい．また読み出されたデータ列はデータキャッシュに配置されるため，時

間的局所性が無いデータがキャッシュ容量を浪費する可能性がある．

SMC は DRAM の同一行内のデータに対して，列アドレスを連続して与えることによりデータ列をアクセスする方法であり，この点では本研究で使用した SDT 方式と同じ手法である．しかし SMC では読み出されたデータ列がメモリコントローラ内でバッファリングされるため，プロセッサ（プログラム）がデータを取得するためには，プロセッサとメモリコントローラ間でのノンキャッシャブルなアクセスが必要となり，大きなオーバヘッドとなる．

以上の方式に対し，本研究で使用する SDT と再構成 FIFO バッファの組み合わせは，上記の問題点を解決し，バッファへのプリフェッチ効果により小さいコストでのデータ取得を可能としている．また，本論文で提案した SDT，FIFO バッファおよび MAC を組み合わせた方式は，ストリーミングデータに対してフィルタ計算してから結果をプロセッサへ転送することで高速化を達成するが，時間的局所性の無いデータによるプロセッサ内のメモリ資源浪費を可能な限り削減する特徴も併せ持つ．

# 第7章　おわりに

本論文では，高速データ転送を可能とするSDTとFIFOバッファを使用し，さらにMCがMAC機能を内蔵し，ストリームデータに対してフィルタ計算を行うことにより，高速なフィルタリング処理を達成する方式を提案し，提案する機構をVHDLで設計した．

RTLシミュレーションにより，プロセッサのみによる積和計算よりも大幅に高速化が達成されることが示された．

今後の課題としては，FIFO機能を持つプロセッサを使用し，実機による有効性の評価を行う．

# 謝辞

本研究を遂行するにあたり, 終始熱心に, かつ懇切丁寧にご指導を賜りました, 田中清史助教授に心から深く感謝するとともに, ここに御礼申し上げます.

貴重なご助言をしていただきました日々野靖教授, 井口寧助教授に深く感謝致します.

その他, 貴重な御意見, 御討論をいただきました日々野研究室, 田中研究室の皆様をはじめとする多くの方々の御助言に対して厚く御礼申し上げます.

最後に, 日頃から暖かく支援して下さいました, 仲間たち, 親族の皆様, 姉そして両親に深く感謝致します.

# 付録

5 章で述べた評価プログラムのアセンブリコードを付録とする.

プロセッサのみでの演算

```
        !#PROLOGUE# 0
        !# vars= 24, regs= 1/0, args= 0, extra= 84
        add     %sp, -112, %sp
        st      %i7, [%sp+68]
        sub     %sp, -112, %i7  !# set up frame pointer ! 0x402
        !#PROLOGUE# 1
        st      %g0, [%i7-40]
        sethi   %hi(4259840), %o0
        st      %o0, [%i7-32]
        sethi   %hi(4263936), %o0
        st      %o0, [%i7-36]
        st      %g0, [%i7-20]
.LL3:                           ! f[ ]
        ld      [%i7-20], %o0
        cmp     %o0, 31
        ble     .LL6
        nop
        b       .LL4
         nop
.LL6:
        ld      [%i7-20], %o0
        mov     %o0, %o1
        sll     %o1, 2, %o0
        ld      [%i7-32], %o1
        add     %o0, %o1, %o0
        ld      [%i7-20], %o1
        add     %o1, 1, %o2
        sta     %o2, [%o0] 0x40 ! Set Value f[i] on Memory
.LL5:
        ld      [%i7-20], %o0
        add     %o0, 1, %o1
        st      %o1, [%i7-20]
        b       .LL3
         nop
.LL4:                           ! m[ ]
        nop
        st      %g0, [%i7-24]
```

```
.LL7:
        ld      [%i7-24], %o0
        cmp     %o0, 31
        ble     .LL10
        nop
        b       .LL8
         nop
.LL10:
        ld      [%i7-24], %o0
        mov     %o0, %o1
        sll     %o1, 2, %o0
        ld      [%i7-36], %o1
        add     %o0, %o1, %o0
        ld      [%i7-24], %o1
        add     %o1, 1, %o2
        sta     %o2, [%o0] 0x40 ! Set Value m[i] on Memory
.LL9:
        ld      [%i7-24], %o0
        add     %o0, 1, %o1
        st      %o1, [%i7-24]
        b       .LL7
         nop
.LL8:                           ! Convolution
        nop
        st
.LL11:
        ld      [%i7-28], %o0
        cmp     %o0, 31
        ble     .LL14
        nop
        b       .LL12
         nop
.LL14:
        ld      [%i7-28], %o0
        mov     %o0, %o1
        sll     %o1, 2, %o0
        ld      [%i7-32], %o1
        add     %o0, %o1, %o0
```

```
        mov     31, %o1
        ld      [%i7-28], %o2
        sub     %o1, %o2, %o1
        mov     %o1, %o2
        sll     %o2, 2, %o1
        ld      [%i7-36], %o2
        add     %o1, %o2, %o1
        ld      [%o0], %o0
        ld      [%o1], %o1
        smul    %o0, %o1, %o0
        ld      [%i7-40], %o1
        add     %o1, %o0, %o0
        st      %o0, [%i7-40]
.LL13:
        ld      [%i7-28], %o0
        add     %o0, 1, %o1
        st      %o1, [%i7-28]
        b       .LL11
         nop
.LL12:
        ld      [%i7-40], %o1
        mov     %o1, %o0
        b       .LL2
         nop
.LL2:
        !#EPILOGUE#
        sub     %i7, 112, %sp           !# sp not trusted here
        ld      [%sp+68], %i7
        retl
        add     %sp, 112, %sp
.LLfe1:
        .size    main,.LLfe1-main
        .ident   "GCC: (GNU) 2.95.3 20010315 (release)"
```

提案方式での演算

```
        !#PROLOGUE# 0
        !# vars= 288, regs= 1/0, args= 0, extra= 84
        add     %sp, -376, %sp
        st      %i7, [%sp+68]
        sub     %sp, -376, %i7  !# set up frame pointer  ! 0x402
        !#PROLOGUE# 1
        mov     31, %o0
        sta     %o0, [%i7] 0x48
        mov     2, %o0          ! Stride Data Size = 4Byte <- Fix
        sta     %o0, [%i7] 0x4A
        st      %g0, [%i7-300]
        st      %g0, [%i7-20]
.LL3:                           ! f[ ]
        ld      [%i7-20], %o0
        cmp     %o0, 31
        ble     .LL6
        nop
        b       .LL4
         nop
.LL6:
        ld      [%i7-20], %o0
        mov     %o0, %o1
        sll     %o1, 2, %o0
        ld      [%i7-20], %o2
        add     %o2, 1, %o3
        sta     %o3, [%o0] 0x4B ! Set Value f[i] on MAC_Filter
.LL5:
        ld      [%i7-20], %o0
        add     %o0, 1, %o1
        st      %o1, [%i7-20]
        b       .LL3
         nop
.LL4:                           ! m[ ]
        nop
        st      %g0, [%i7-24]
.LL7:
        ld      [%i7-24], %o0
```

```
        cmp     %o0, 31
        ble     .LL10
        nop
        b       .LL8
         nop
.LL10:
        ld      [%i7-24], %o0
        mov     %o0, %o1
        sll     %o1, 2, %o0
        add     %i7, -296, %o1
        ld      [%i7-24], %o2
        add     %o2, 1, %o3
        sta     %o3, [%o1+%o0] 0x40  ! Set Value m[i] on Memory
.LL9:
        ld      [%i7-24], %o0
        add     %o0, 1, %o1
        st      %o1, [%i7-24]
b       .LL7
         nop
.LL8:
        nop
        mov     (-296)+31*4,%g5         ! m[0]+n*4=(-296)+31*4
        lda     [%i7+%g5] 0x4F,%o0      ! Convolution starts m[n-1]
        st      %o0, [%i7-300]          ! [%i7-300]=result
        ld      [%i7-300], %o1
        mov     %o1, %o0
        b       .LL2
         nop
.LL2:
        !#EPILOGUE#
        sub     %i7, 376, %sp           !# sp not trusted here
        ld      [%sp+68], %i7
        retl
        add     %sp, 376, %sp
.LLfe1:
         .size   main,.LLfe1-main
         .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

# 参考文献


[1] Tomoharu Fukawa, Kiyohumi Tanaka and Jun Miyazaki, "The Highly Functional Memory Controllre for Main Memory Databese." ,IPSJ SIG Notes,ARC,Vol2002,No112,pp.77-82,2002.

[2] intel,Inc, "PC SDRAM Specification Revision1.63",October 1998.

[3] Khairuddin bin Khalid and Kiyohumi Tanaka, "Implementation of FIFO Buffer Using Cache Memory." ,IPSJ SIG Notes,ARC,Vol2002,No112,pp.83-88,2002.

[4] Parthasarathy Ranganathan,Sarita Adve and Norman P.Jouppi, "Reconfigurable Caches and Their Application to Media Processing." ,Proc.of ISCA,pp.214–224,2000.

[5] Rabiner,L and B,Gold, "Theory and Applications of Digital Signal Processing." ,Enflewood Cloffs,Prentice–Hall.

[6] Oppenheim,A and R,Schafer, "Digital Signal Processing." ,Enflewood Cloffs,Prentice–Hall.

[7] K.Tanaka, "Fast Context Switching by Hierarchical Task Allocation", Proc. of IWIA, pp.20–29, 2003.

[8] N.P.Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", Proc. of the 17th ISCA, pp. 364–373, 1990.

[9] Farkas, N.P.Jouppi and P.Chow, "How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?", Proc. of the 1st HPCA, pp. 78–89, 1994.

[10] "J.B.Carter, W.C.Hsieh, L.B.Stoller, M.R.Swanson, L.Zhang, E.L.Brunvand, A.Davis, C.-C.Kuo, R.Kuramkote, M.A.Parker, L.Schaelicke, and T.Tateyama, "Impulse: Building a Smarter Memory Controller", Proc. of 5th HPCA, pp.70–79, 1999.

[11] S.A.McKee, A.Aluwihare, B.H.Clark, R.H.Klenke, T.C.Landon, C.W.Oliver, M.H.Salinas, A.E.Szymkowiak, K.L.Wright, W.A.Wulf, J.H.Aylor, "Design and Evaluation of Dynamic Access Ordering Hardware", Proc. of 10th ICS, pp.125–132, 1996.

[12] Curtis Roads, “The Computer Music Tutorial”,Massachusetts Institute of Technology,1996.

[13] Xilinx,Inc http://www.xilinx.com.

[14] Altera,Inc “SDR SDRAM Controller White Pager”,1999.

[15] Elpida Memory,Inc “ ユーザーズマニュアル SDRAM の使い方”,1999.