平成 30 年度

修士学位論文

微分可能ニューラルコンピュータ向き ハードウェアマルチコアアクセラレータの 検討

A Study on Hardware Multicore Accelerator for Differentiable Neural Computer

1215082　　齋藤 あかね

指導教員　　岩田 誠

2019 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

微分可能ニューラルコンピュータ向き
ハードウェアマルチコアアクセラレータの検討

齋藤 あかね

近年の DNN(Deep Neural Network) 技術の進展に伴って，IoT 機器へも AI 技術を導入する研究が活発になっている．中でも自動運転や AI アシスタント等への需要から，IoT エッジデバイスで DNN を低消費電力および高速に実現するためのアクセラレータの研究開発が活発化している．

現状では，画像処理向きの CNN に特化したアクセラレータが多く開発されている ([1], [2] など)．そこで本研究では Google Deep Mind 社より提案された微分可能ニューラルコンピュータ DNC(Differentiable Neural Computer) [3] に着目した．DNC は文章やグラフなど複雑なデータ構造の学習が可能な DNN の 1 つであるが，このアルゴリズムに対応したアクセラレータは現時点では見当たらない．

本研究では，DNC の計算に必要な LSTM 演算を行うための命令セットを検討し，パイプライン実行可能な回路構成をとった．更に，送受信命令および相互結合網を実装し，マルチコア構成による高速化を検討した．動作検証の結果，LSTM ネットワークを 16 コアで計算した場合の clock cycle 数が，1 コアで同様の LSTM ネットワークを計算する場合に要する clock cycle 数と比較して，34.4%減少したことを確認した．

キーワード　微分可能ニューラルコンピュータ (DNC)，長期短期記憶 (LSTM)，ハードウェアアクセラレータ，FPGA

Abstract

A Study on Hardware Multicore Accelerator for Differentiable Neural Computer

Akane SAITO

Along with the recent progress of DNN (Deep Neural Network) technology, research to introduce AI technology to IoT devices has become active. Among others, research and development of accelerators for realizing DNN at low power consumption and high speed with IoT edge devices has been actively pursued due to demand for automatic operation, AI assistant, etc.

Currently, many accelerators specialized for CNN oriented to image processing have been developed ([1], [2], etc.). Therefore, in this research, we focused on Differentiable Neural Computer (DNC) [3] proposed by Google Deep Mind.DNC is one of DNNs that can learn complicated data structures such as sentences and graphs, but accelerators corresponding to this algorithm can not be found at the present time.

In this research, we examined instruction set for LSTM calculation necessary for DNC calculation, and adopted a pipeline executable circuit configuration. Furthermore, we implemented transmission and reception instructions and interconnection network and investigated speedup by multicore configuration. When calculating the LSTM network for operation verification with 16 cores, it was confirmed that the number of clock cycles decreased by 34.4 % compared with the case of calculating the same LSTM network in single core.

key words DNC, LSTM, Hardwear Accerelator, FPGA

目次

図目次

表目次

第 1 章

序論

近年の DNN(Deep Neural Network) 技術の進展に伴って，IoT 機器への AI 技術導入の動きが活発になっている．例えば，高度な物体認識能力が必要とされる自動運転車や，音声認識能力を用いてユーザの発言を正確に解釈する AI アシスタントなど，AI を用いた高性能なエッジデバイスが研究・開発されている．エッジデバイス用途を意識した AI チップの例としては，Arm 社の Arm ML Processor [4] や, Intel 社の Myriad 2 [5] などが挙げられる．

一般的に DNN は画像認識や音声識別等に対して高い認識精度を得られる反面，その計算コストの高さが問題となっている．そのため，計算リソースの少ない IoT エッジデバイス上で DNN をより高速に計算するための研究は各地で成されている [6]， [7].

アルゴリズムレベルでのアプローチとしては，低ビット幅数でネットワーク計算を行う QNN [8] が提案されている．また，ニューラルネットワークの持つ重み行列をバイナリ化することによってパラメータや計算量を削減する工夫がなされた，バイナリニューラルネットワーク (BNN) [9]， [10] も提案されている．また，このバイナリニューラルネットワークによって CNN のような大規模ニューラルネットワークをバイナリ化し，ImageNet を使用した画像分類を行った事例も存在している [11].

ハードウェアレベルでのアプローチとしては，例えば，積和演算の高効率化を目的として SIMD 演算器と SIMD 演算命令を実装したもの [12] や，クロスバ構造メモリを値の格納だけでなく積和演算の実行にも使用することによって省電力・省スペースで DNN 演算を行うアクセラレータ ISAAC[13] が提案されている．その他，回路規模の縮小を目的として構成ニューロンを小型化したもの [14]，画像処理向けに性能を特化させたもの等 [1]， [2] が存在する．また，これらに挙げられうような DNN ハードウェアアクセラレータを，アルゴ

リズムおよび回路レベルで最適化を行うためのアプローチである Minerva が提案されている [15].

これらのアクセラレータの登場によって，ニューラルネットワークによる画像や音声認識，分類問題等を高速に実行できるようになった．しかしこれらのニューラルネットワークは，膨大な計算量以外にも，その不透明性や，階層構造を扱う方法が存在していない等といった，いくつかの問題が指摘されている [16].

現在活用されている一般的なニューラルネットワークは，その多くが「得られた入力に対する正解は何か」という事実を学習する，問題指向のアプローチを行うものである．そのため，あるタスクに対して「正解を得るにはどうすれば良いか」という目標指向のアプローチを行うことが難しい．ニューラルネットワークがこのような目標指向の方法で学習をするには，アルゴリズムの一般性にとって重要となる変数の概念や，学習情報を記憶しておくメモリが必要となる．通常のニューラルネットワークは，ニューロン間の重み情報のみが学習情報を記憶する資源であるため，一般性を得ることが難しい．

このような背景の元で，Google DeepMind 社から，新しいニューラルネットワークモデルとして微分可能ニューラルコンピュータ DNC(Differentiable Neural Computer) が提案された [3]．DNC は，長期短期記憶 LSTM(Long Short-term Memory) ニューラルネットワークと，メモリの役割を果たす，外部メモリ行列から構成されている．この外部メモリに新しい学習情報を書き込み，更に，メモリの内容を変数として扱うことを可能としている．DNC はネットワーク全体が微分可能であるため，勾配降下法で外部メモリの走査方法を学習することが可能となっている [3].

本研究ではこの DNC をエッジデバイス上で高速動作させることを検討する．前述の通り DNC は外部メモリを有しており，その走査方法を学習するアルゴリズムであるため，長文やグラフなどの複雑なデータ構造をもつタスクにも対応可能なニューラルネットワークである．このアルゴリズムに対応したアクセラレータは現時点では見当たらない．現行ニューラルネットワークでは困難であるタスクを学習可能な DNC をエッジデバイス上で実現可能にすることで，現行 IoT エッジ機器の更なる高性能化が期待できる．

本研究では DNC の Controller Network，外部メモリから値を読み出す Read head，外部メモリに値を書き込む Write head に着目し，これらの機構を高速に計算するマルチコアアクセラレータ回路を検討した．検討回路はパイプライン実行可能な積和演算や複数種類の非線形関数を含む命令セットに加え，時系列データを考慮したメモリアドレッシング方式を導入したシングルコアアーキテクチャを採用している．

本稿の第 2 章では，本研究で取り上げる DNC について解説し，DNC 向きハードウェアアクセラレータ回路のために必要な要件および設計方針をまとめる．

第 3 章では，設定した要件を元に試作したシングルコアアーキテクチャについて述べる．主に高速化および回路リソース節約のために検討した命令セット，アドレス計算方法について説明する．その後，試作シングルコアを用いたマルチコアアーキテクチャについて述べる．マルチコア化にあたっての要件定義を定めた後，定めた要件定義に基づいて設計したコア間結合回路の構造および，各コアで実行する命令のスケジューリング方法について説明する．また，命令スケジューラおよび試作した専用 HDL 高位合成ツールについても述べる．

第 4 章では提案回路を評価した結果について述べる．エッジデバイス上での運用を想定しているため，評価内容は処理に要する時間に加えて，回路規模についても評価を行う．最後に，本研究で得られた結果から，今後の課題および展望についても述べる．

第 2 章

微分可能ニューラルコンピュータ (DNC)

2.1 緒言

本章では，本研究で取り上げるニューラルネットワーク DNC について説明する．その後，DNC 演算を実行するために必要となる機能について述べ，要件定義を行う．

2.2 DNC(Differentiable Neural Computer)

DNC は Google Deep Mind 社によって考案されたニューラルネットワークモデルである [3].

CNN(Convolutional Neural Network) や RNN(Reccurent Neural Network) などの一般的なニューラルネットワークは画像認識や音声認識に対して高精度な結果を得ることが可能になっているが，文章やグラフの読解といった，複雑なデータ構造を扱う能力が不足している．文献 [3] では，例えば，図 2.1 のような無作為に作成されたグラフ中から欠落した繋がりを推定し，図 2.2 のような家系図や，交通網といったような特定のグラフに一般化させるといった学習を行っている．家族図を学習した DNC は,「Freya の母方の大おじは誰か？」といった入力を受けて「Fergus である」と答えることが可能になる [3].

DNC はこのような複雑なデータ構造を扱うことが可能なニューラルネットワークモデルとなっている．

構成としては，図 2.3 に示すような，長期短期記憶 LSTM(Long Short-term Memory) ニューラルネットワークに構造体データを記憶する外部メモリ (External Memory) が結合されたものである．ニューラルネットワークと外部メモリの間には，1 個の書き込みヘッド (Write head) と R 個の読み取りヘッド (Read head) が存在する．外部メモリは $W \times N$ の行列として扱う．

LSTM は DNC がメモリを操作するために必要なパラメータ群 (Interface vector)$\boldsymbol{\xi}_t$ の生成にも使用されており，コントローラネットワークとしての役割を持つ．$\boldsymbol{\xi}_t$ は 10 種類のパラメータを持つ，大きさ $(W \times R) + 3W + 5R + 3$ のベクトルである．

外部メモリへの書きこみ時は，コントローラネットワークで生成された $\boldsymbol{\xi}_t$ を元に書きこみヘッドで書き込み重み (Write weighting)$\mathbf{w}_t^w$ が生成される．この値と，1 時刻前のメモリの値 $\mathbf{M}_{t-1}$ から新しいメモリの値 $\mathbf{M}_t$ が計算され，更新されることで行われる．具体的な計算式は，式 2.10 に示す．

外部メモリからの読み出し時は，1 時刻前のメモリの値 $\mathbf{M}_{t-1}$ とメモリ操作パラメータ群 $\boldsymbol{\xi}_t$ を元に，読み取りヘッドで読み取り重み (Read weighting)$\mathbf{w}^r$ が生成される．この値と現時刻のメモリの値 $\mathbf{M}_t^T$ との積和を取ることで行われる．具体的な計算式は，式 2.9 に示す．こうして生成された読み取りベクトル (Read vector)$\mathbf{r}_{t-1}$ を，コントローラネットワークの

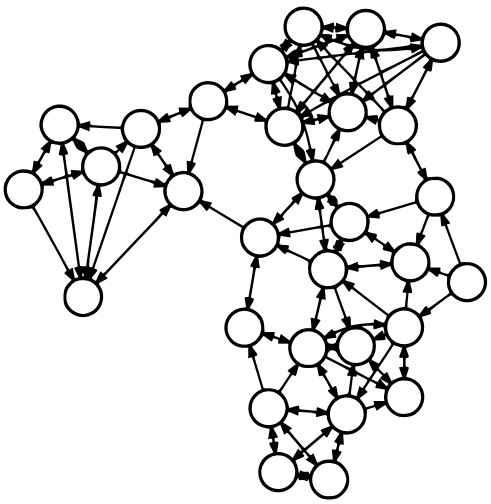


図 2.1　無作為なグラフ (文献 [3] より引用)

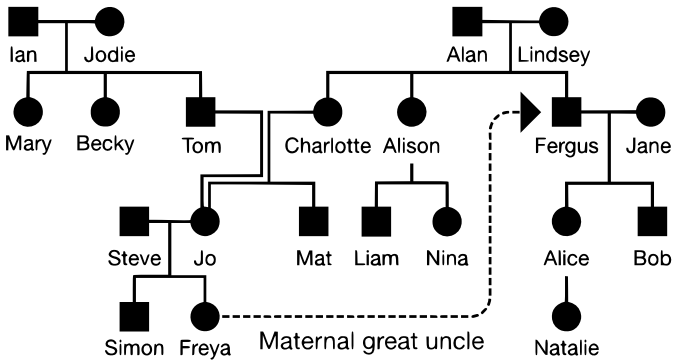


図 2.2　家族木 (文献 [3] より引用)

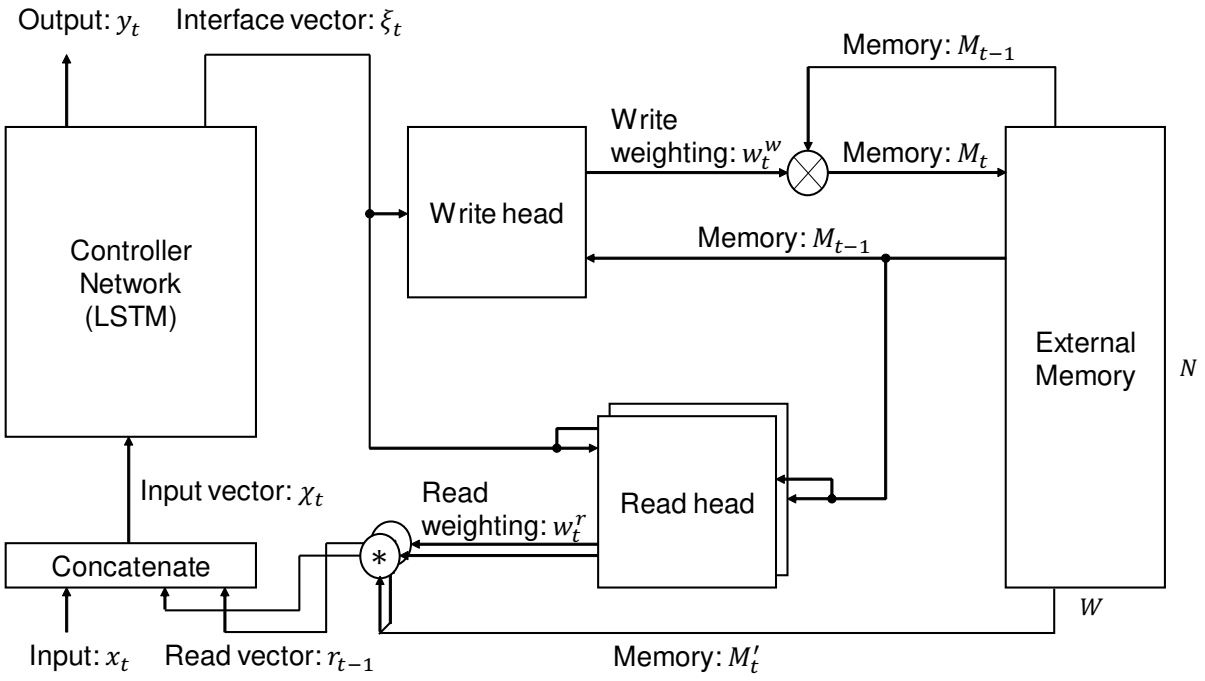


図 2.3　DNC の構成概略

新たな入力 ($\boldsymbol{\chi}_t$) の一部として扱う．よって，$\boldsymbol{\chi}_t$ は式 2.1 に表すようなベクトルとなる．

$$\boldsymbol{\chi}_t = [\mathbf{x}_t; \mathbf{r}_{t-1}^1; ...; \mathbf{r}_{t-1}^R] \tag{2.1}$$

2.3　アクセラレータの要件定義

DNC 向きアクセラレータに必要な機能は，以下に挙げる演算を処理する機能である．

ニューロン： 通常ニューロン，LSTM ブロック

ネットワーク： 多層ネットワーク，リカレントネットワーク

外部メモリ： 読み取り，書き込み

ニューロンおよび外部メモリの項目に関しては，DNN 演算で多用される積和演算の高効率化に加え，LSTM ブロックおよび DNC における外部メモリへの読み書きに対応した命

令セットを定義する必要がある．また，ネットワークの項目に関しては，リカレントネットワークを扱うために時系列データを管理する必要がある．

以下より，DNC をコントローラネットワーク部分と外部メモリ (読み取り・書き込みヘッドを含む) 部分に分割し，それぞれに対する要件定義を行う．

2.3.1　コントローラネットワーク

DNC は，LSTM をコントローラネットワークとして使用している．LSTM は RNN(Reccurent Neural Network) の一種であり，RNN より長期的な時系列情報を扱うことが可能なニューラルネットワークである．入力層，中間層 (LSTM 層)，出力層から成り，中間層では LSTM Block がニューロンの役割を果たす．LSTM Block 内部の構造は図 2.4 に示す通りである．

LSTM への入力を式 2.2 に示す．また，式 2.3 は，DNC のコントローラネットワークとしての LSTM に対する入力である．

$$\mathbf{x}_{lstm} = [\mathbf{x}_t; \mathbf{y}_{t-1}^{l}; \mathbf{y}_t^{l-1}] \tag{2.2}$$

$$\mathbf{x}_{dnc} = [\chi_t; \mathbf{y}_{t-1}^{l}; \mathbf{y}_t^{l-1}] \tag{2.3}$$

LSTM Block 内の時刻 t，第 l 層における計算は式 2.4〜2.8 で表される．なお，図 2.4 左では， 式 2.7 における $\mathbf{s}_t^l$ の矢印は省略している．

$$\mathbf{i}_t^l = \sigma(\mathbf{W}_i \mathbf{in}_t + \mathbf{b}_i) \tag{2.4}$$

$$\mathbf{f}_t^l = \sigma(\mathbf{W}_f \mathbf{in}_t + \mathbf{b}_f) \tag{2.5}$$

$$\mathbf{o}_t^l = \sigma(\mathbf{W}_o \mathbf{in}_t + \mathbf{b}_o) \tag{2.6}$$

$$\mathbf{s}_t^l = \mathbf{f}_t \mathbf{s}_{t-1} + \mathbf{i}_t tanh(\mathbf{W}_s \mathbf{in}_t + \mathbf{b}_s) \tag{2.7}$$

$$\mathbf{h}_t^l = \mathbf{o}_t tanh(\mathbf{s}_t) \tag{2.8}$$

式中の σ はシグモイド関数を表す．

以上の内容から，コントローラネットワークの計算を実現するために必要な要件として

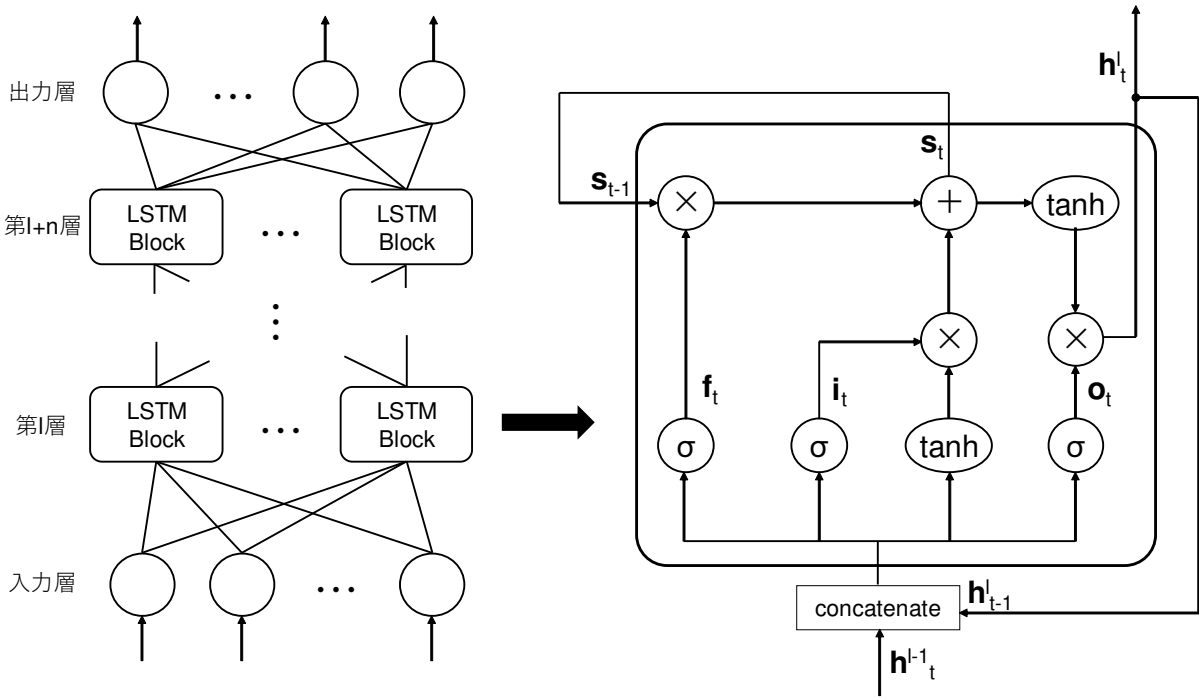


図 2.4　LSTM の全体構成と LSTM Block 内部の計算内容

- 積和演算命令の実装
- 時系列入出力データの管理機構
- 非線形関数への対応 (sigmoid, tanh)
- リカレント機構，メモリセル ($\mathbf{s}_t^l$) の実現

が挙げられる．

ニューラルネットワーク演算はその大部分が積和演算によって成り立つため，積和演算を高速化することは LSTM および DNC の高速演算化においても重要である．時系列入出力データの管理機構については，一般的なニューラルネットワークとは異なり，本研究で扱うネットワークが時系列データを対象としているために必要となる．アクセラレータが，現在計算している時刻を記憶するための機構を設けることで実現する．本研究で取り扱う非線形関数は sigmoid 関数および tanh であるが，これらの結果を計算によって求めると多大な計

算時間を要するため，精度は落ちるが LUT を使用して高速に結果を求める方法をとる．リカレント機構およびメモリセルの実現に関しては，ある時刻 t で算出された LSTM の出力およびメモリセルの値を次の時刻 t+1 でも使用するため，上書きされないように配慮した上でメモリマップの作製および，命令セットの検討を行う必要がある．

これらの点を踏まえて，シングルコアアーキテクチャの設計を行う．

2.3.2 外部メモリ

読み取りヘッドで読み取りベクトル r_t を生成する式は，式 2.9 で表される．T は転置行列を表す．書き込みヘッドで新たな外部メモリの値 M_t を計算する式は，式 2.10 で表される．○は要素ごとの積 (アダマール積) を表す．

$$\mathbf{r}_t^i = \mathbf{M}_t^T \times \ \mathbf{w}_t^{r,i} \tag{2.9}$$

$$\mathbf{M_t} = \mathbf{M_{t-1}} \bigcirc (E - \mathbf{w}_t^w \mathbf{e}_t^T) + \mathbf{w}_t^w \mathbf{v}_t^T \tag{2.10}$$

通常の四則演算および積和演算の機能を有したアクセラレータであれば上記の計算は可能であるが，転置行列を扱うため，転置の操作に要する時間を削減する工夫が必要となる．

以上の内容から，外部メモリに関する計算を実現するために必要な要件は，前節で述べたコントローラネットワークに対する要件定義と重複するものを除くと

- 転置行列の高速処理
- 単位行列とのベクトル計算

が挙げられる．

2.4 結言

本章では，本研究で取り上げる DNC について述べた．DNC を構成する要素をコントローラネットワークおよび，外部メモリ (読み取りヘッド，書き込みヘッド) に大別し，それぞれについて，演算を実現するための要件定義を行った．DNC の演算に関しては，コント

2.4 結言

ローラネットワークである LSTM および，外部メモリへの読み書き演算を高速化することを目的とする．

第 3 章

アクセラレータアーキテクチャ

3.1 緒言

本章では，前章で述べた要件を元に構成したアクセラレータのアーキテクチャについて述べる．シングルコアアーキテクチャおよび，マルチコアアーキテクチャに関する要素技術について述べる．シングルコアアーキテクチャについては，主にコア内メモリの構成や非線形関数の適用手法について述べる．マルチコアアーキテクチャについては，主にマルチコアでの処理で必要となるコア間通信方法や，命令スケジューリング手法について述べる．その後，これらのアーキテクチャで動作する命令セットについて述べる．

また，試作したマルチコア用スケジューラ，専用 HDL 高位合成ツールの概要についても述べる．

3.2 シングルコアアーキテクチャ

シングルコアは，図 3.1 のような 4 段のパイプライン構成とした．各ブロックの機能については，第 3.2.1 節で述べる．なお，試作コアの仕様は表 3.1 の通りである．

具体的な計算方法に関しては，第 3.6.1 節で説明する．積和演算命令の実行時には，演算対象ベクトルを DM_X と DM_W からそれぞれ連続的に読み出し，これらの乗算結果をアキュムレータ (acc) に順次積算する．よって，演算対象のベクトルサイズ n に対して，$n+3$ クロックサイクルで積和演算の実行が可能である．また，LSTM および DNC で扱うデータは時系列データである．そのため，時系列レジスタ (seq) を導入し，命令実行時にこの値を

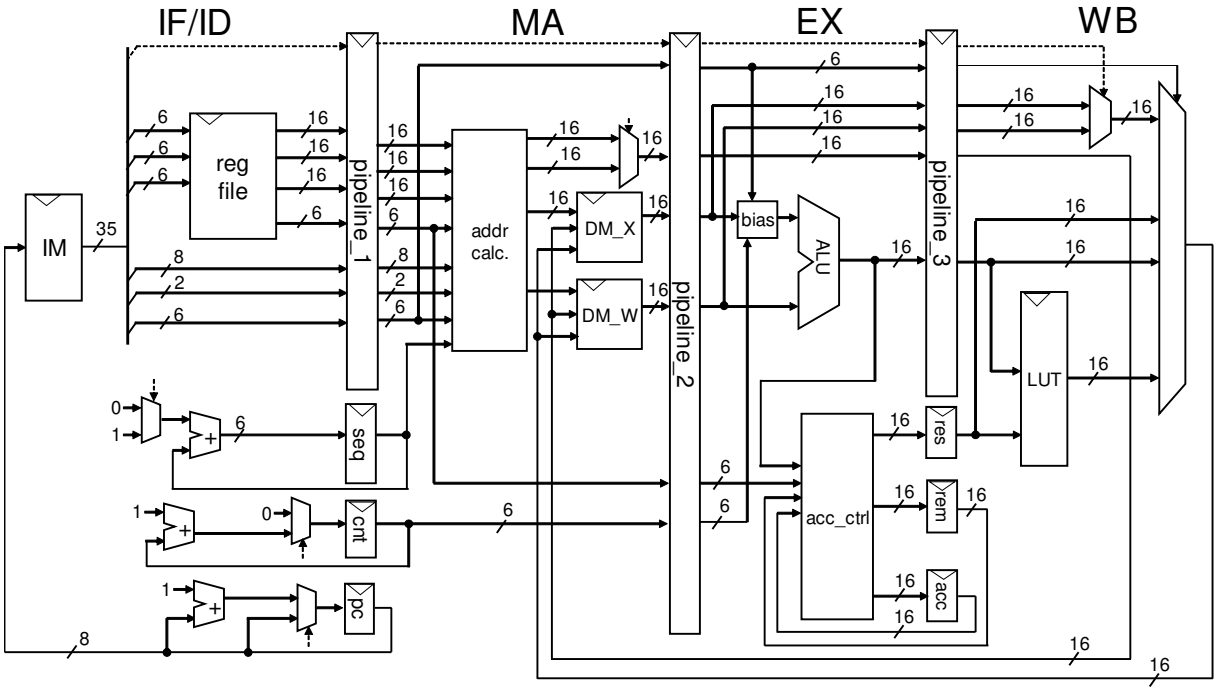


図 3.1　シングルコアアーキテクチャ

適宜更新して DM_X および DM_W 内のベクトルへアドレッシングする．

また，シグモイド関数や $tanh$ 等の非線形関数の適用には LUT を用いる．LUT サイズ削減のために，零近辺のみ ($|x| < 7$) を参照する方式とした．零近辺以外の値は適用する非線形関数によって異なるが，LUT を使用せず，0 または 1，-1 または 1 の値を，非線形関数の適用結果として出力する．

表 3.1　コアの設計仕様

データ形式	16bit fixed-point (Q6.10)
命令メモリ (IM)	35 bit × 256 words
データメモリ (DM_X, DM_W)	16 bit × 4,096 words × 2
LUT	16 bit × 256 words × 2

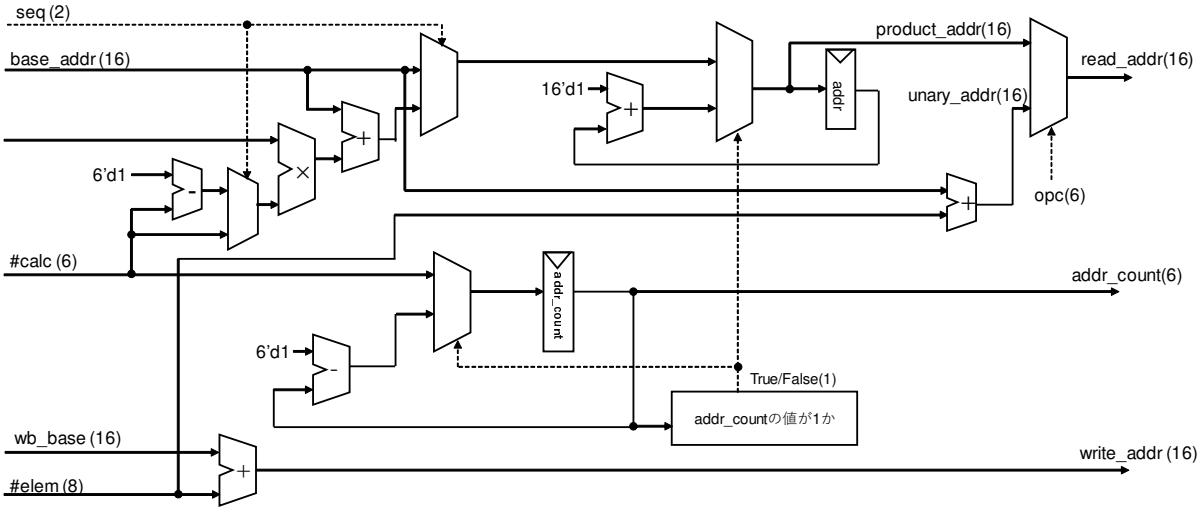


図 3.2　アドレス計算回路アーキテクチャ

3.2.1　アーキテクチャ詳細

メモリおよびレジスタファイルを除く，図 3.1 中の各ブロック (addr_calc，acc_ctrl，bias) の機能について述べる．メモリおよびレジスタファイルの詳細に関しては，第 3.3 節より述べる．

アドレス計算回路 (addr_calc)

データメモリ (DM_X，DM_W) から値を読み出すためのアドレス計算および，計算済みの値をデータメモリへ格納するためのアドレス計算を行う回路である．図 3.2 に，その構成を示す．

二項演算の場合，データメモリに対する読み取り用 (read_addr) および書き戻し用 (write_addr) アドレス計算式は以下の式に示す通りである．

$$read\_addr = (seqence \times \ arity) + base\_addr \tag{3.1}$$

$$write\_addr = (seqence \times \ arity) + base\_addr + \#elem \tag{3.2}$$

sequence は，時系列情報を表す．詳細は第節で述べる．arity は，積和演算における，現

在計算している項の番号を表す．そのため arity の値は 1 命令実行中に 0 から (#calc-1) まで変動する．

なお，図中の product_addr は二項演算時のアドレス，unary_addr は単項演算時のアドレスを表す．実行されている命令によって，どちらかが read_addr として選択される．

アキュムレート管理回路 (acc_ctrl)

積和演算に伴うアキュムレータ (acc) の値を管理するための回路である．アキュムレート中は accumurate(acc) レジスタに積和結果が蓄積される．remain(rem) レジスタで残りのアキュムレートする回数を管理し，最終結果は result(res) レジスタから出力する．

図 3.5 に構成を示す．レジスタファイルから読み出してきた計算回数 (#calc) をデクリメントし，その値をアキュムレートの制御に使用している．remain の値が 0 になると計算終了とみなし，新しい#calc を選択し，同様の操作を行う．

result レジスタは ALU の計算結果を出力するレジスタとしても機能しており，積和演算以外の命令が実行されている場合は ALU からの出力 (alu_out) が選択され，result レジスタに格納される．

acc レジスタは積和演算実行中は積和結果を格納するが，積和演算の実行が完了したことが remain レジスタの値により判明すると値を 0 にリセットする．

バイアス判定回路 (bias)

提案アーキテクチャでは，データメモリにバイアス項を格納し，他データと同様に連続的に読み出される．例えば図 3.5 に示す LSTM ネットワークにおいて，1 つの LSTM ブロック (LSTM_0) の忘却ゲート (f) について計算を行う場合は，図に示すように，各 LSTM ブロックに対する入力 (LSTM_IN[0], LSTM_IN[1]) と，各 LSTM ブロックの 1 時刻前の出力 (R[0], R[1])，および忘却ゲートに関するバイアス (l2/upward/b(0)) の 5 種類のデータを使用する．

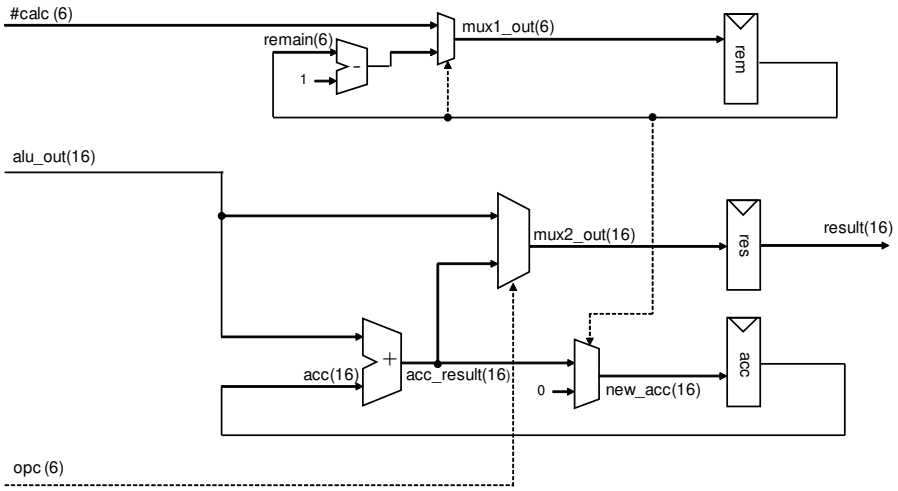


図 3.3　アキュムレート管理回路アーキテクチャ

よって，このとき，データメモリから連続的に読み出されるデータ数は 5 つである．具体的には，DM_X からは，図 3.9 の”LSTM_IN[0]” ～ ”f[0]”が，DM_W からは，図 3.10 の”l2/upward/W(0)” ～ ”l2/upward/b(0)”に相当するデータが読みだされ，バイアス判定回路が無い場合は以下の様に計算が行われる．

$$foget\ gate(LSTM\_0) = (LSTM\_IN[0] \times l2/upward/W(0)) + ... \\ ... + (R[1] \times l2/lateral/W(1)) + (f[0] \times l2/upward/b(0)) \tag{3.3}$$

このとき $f[0]$ は忘却ゲート計算に不要なデータであるため，得られる値が異なってしまう．そこで，図 3.4 に示すようなバイアス判定回路では，バイアス有り計算命令が実行されると，上記の式を，以下の様に計算するようデータを変更する．

$$forget\ gate(LSTM\_0) = (LSTM\_IN[0] \times l2/upward/W(0)) + ... \\ ... + (R[1] \times l2/lateral/W(1)) + (1 \times l2/upward/b(0)) \tag{3.4}$$

バイアス項を含めて連続的に積和演算データを読み出して計算することで，アドレッシングを複雑にすることなく，バイアス項を含んだ積和演算の計算を 1 命令で行うことを可能にしている．

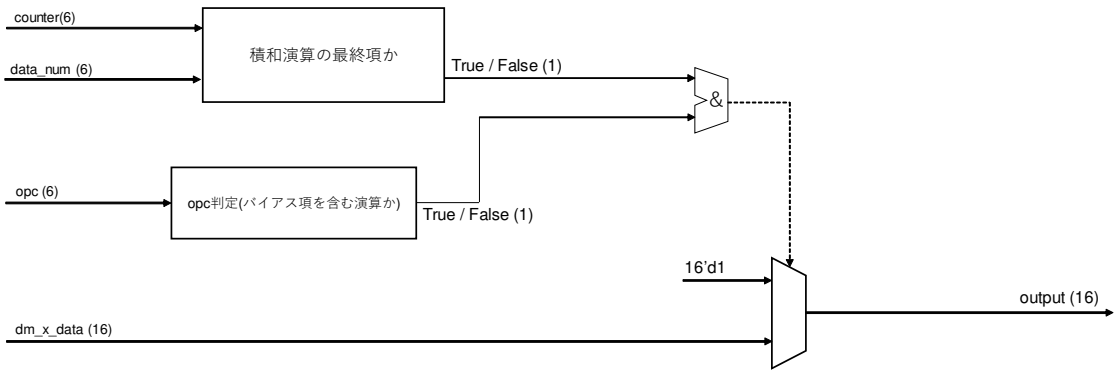


図 3.4　バイアス判定回路アーキテクチャ

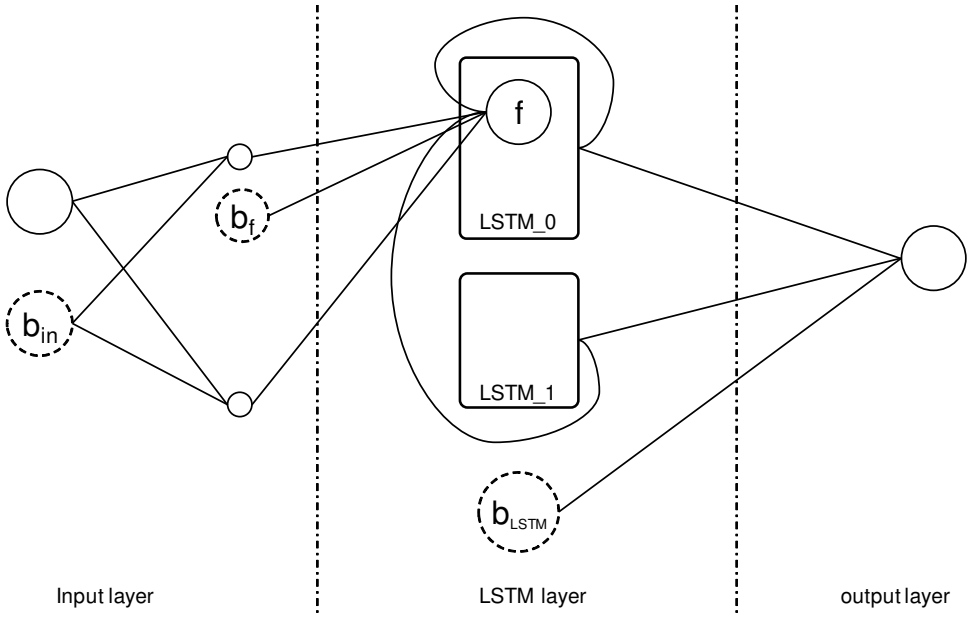


図 3.5　忘却ゲート計算時に使用されるパラメータ

3.3　レジスタファイル

提案アーキテクチャはオペランドを 2 つのデータメモリから同時に読み出すため，命令ビットを節約する必要がある．そのためベースアドレス指定によるアドレッシングを採用している．ベースレジスタファイル内部は図 3.6 のような構成とした．レジスタファイル内の要素は，データメモリ (DM_X，DM_W) のベースアドレス (16bit) と，そのデータに対応

したベクトル演算回数 (6bit) が結合されたビット列から成る．

レジスタファイルに格納されているベースアドレスは，ネットワークの各層の先頭ニューロンや，先頭ニューロンに対応する重みのアドレスである．例えば，図 3.7 のネットワーク構成であれば，ベースアドレスは，図 3.6 中レジスタファイルのレジスタ番号 1〜6 に対応する位置に格納されている．

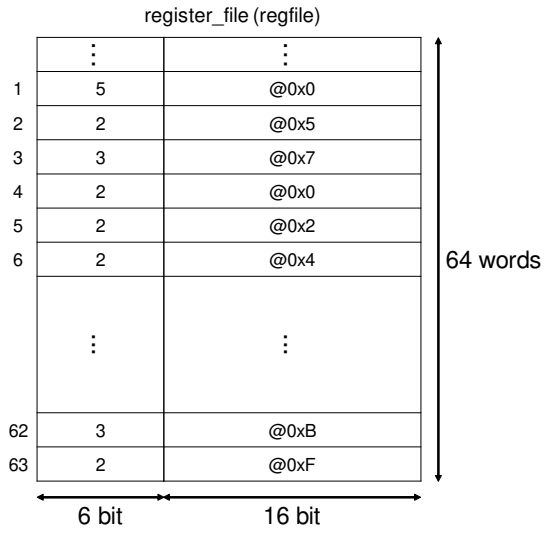


図 3.6　レジスタファイルの構成

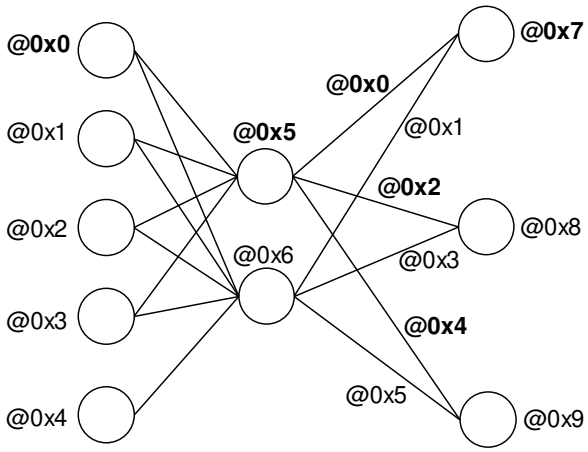


図 3.7　計算ネットワーク例

3.4　メモリ構成

使用メモリの構成について述べる．非同期読み出し･同期書き込みのデュアルポート RAM として，命令メモリ，データメモリおよび LUT に使用している．ただし，命令メモリと LUT に関しては，書き込みは発生しない．

3.4.1　命令メモリ

命令メモリの構成は図 3.8 左のようになっている．提案回路では命令メモリにおけるアドレスビット長を 8bit に指定しているため，命令メモリのサイズは 35 bit × 256 words となっている．PC(Program Counter) レジスタで指定したアドレスから命令を読み出した後

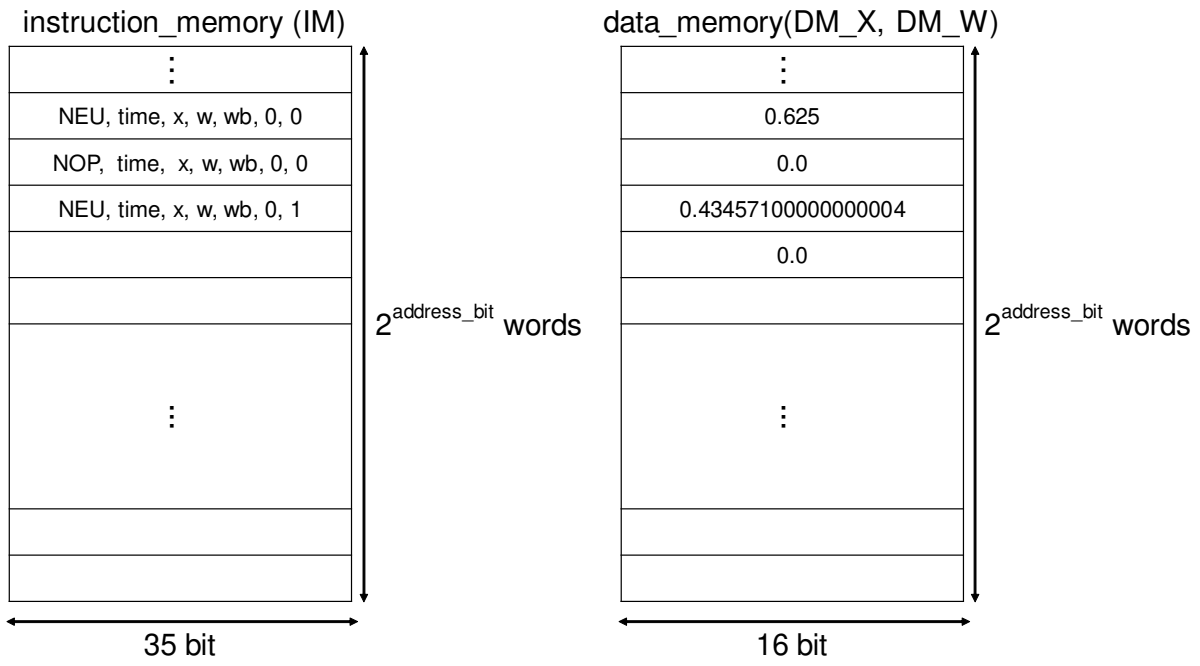


図 3.8　命令メモリ，データメモリの構成

は次のアドレスの命令を指定するが，積和演算命令の場合は，レジスタファイルに格納されているベクトル演算回数の間だけ PC の更新が停止する．

3.4.2　データメモリ

データメモリの構成は図 3.8 右のようになっている．提案回路ではデータメモリにおけるアドレスビット長を 12bit に指定しているため，メモリのサイズは DM_X，DM_W 共に 12 bit × 4096 words となっている．二項演算に必要なオペランドを各メモリから 1 つずつ同時に読み出す．

提案回路での動作確認用 LSTM(図 4.1) における，データメモリのメモリマップを図 3.9，3.10，3.11 に示す．

図 3.9 は DM_X，図 3.10，図 3.11 は DM_W のメモリマップである．DM_X には主に，LSTM ネットワークへの入力値，各ゲートの値，LSTM ネットワークからの出力値を格納

する．DM_W には重みとバイアス，LSTM ブロック内のメモリセルの値を格納する．

DM_X における各データの名称は，”データ名 [LSTM ブロック No.]”としている．よって，LSTM ブロック (LSTM_0) 内の忘却ゲートの値であれば，”f[0]”のように表される．以下にデータ名の説明を示す．

LSTM_IN　LSTM ブロックへの入力値

R　LSTM ブロックからのリカレント値

f　忘却ゲート

i　入力ゲート

o　出力ゲート

LSTM_OUT　LSTM ブロックからの出力

X　LSTM ネットワークへの入力値

Y　LSTM ネットワークからの出力値 (出力層からの出力)

重みデータの名称は，” レイヤ No. / 重み (W) かバイアス (b) か [LSTM ブロック No] ”または，” レイヤ No. / 現時刻の情報 (upward) か前時刻の情報 (lateral) か / 重み (W) かバイアス (b) か (通し番号) ”としている．例えば，入力層 (第 1 層) から LSTM 層の LSTM ブロック (LSTM_0) にかかる重みの場合，データ名称は”l1/W [0]”となり，LSTM 層 (第 2 層) 内の LSTM ブロック (LSTM_0) の忘却ゲートにかかる重みの場合，データ名称は”l2/upward/W (0)”および，”l2/upward/W (1)”となる．

3.4.3　LUT

シグモイド関数 (図 3.12) や $tanh$(図 3.13) といった非線形関数は式 3.5，3.6 で求められるが，これらの式は除算や自然対数といった，比較的計算時間を要する計算が含まれるため，LUT を用いて近似解を求めることで非線形関数計算を実現する．

3.4 メモリ構成

address	データ名称	説明
0x0	LSTM_IN[0]	input for lstm block0
0x1	LSTM_IN[1]	input for lstm block1
0x2	R[0]	recurrent from LSTM 0
0x3	R[1]	recurrent from LSTM 1
0x4	f [0]	forget gate (LSTM 0)
0x5	f [1]	forget gate (LSTM 1)
0x6	i [0]	input gate (LSTM 0)
0x7	i [1]	input gate (LSTM 1)
0x8	o [0]	output gate (LSTM 0)
0x9	o [1]	output gate (LSTM 1)
0xA	LSTM_OUT [0]	output from LSTM block 0
0xB	LSTM_OUT [1]	output from LSTM block 1
0xC	X[0] (t=0)	input (t=0)
…	…	…
0x14	X[0] (t=8)	input (t=8)
0x15	X[0] (t=9)	input (t=9)
0x16	Y[0] (t=0)	output (t=0)
…	…	…
0x1F	Y[0] (t=9)	output (t=9)

図 3.9　DM_X メモリマップ

$$sigmoid(x) = \frac{1}{1+e^{x}} \tag{3.5}$$

$$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{3.6}$$

address	データ名称	説明
0x0	l1/W [0]	input layer
0x1	l1/b [0]	input layer
0x2	l1/W [1]	input layer
0x3	l1/b [1]	input layer
0x4	l2/upward/W (0)	foget gate [0] (from LSTM_IN[0])
0x5	l2/upward/W (1)	forget gate [0] (from LSTM_IN[1])
0x6	l2/lateral/W (0)	reccurent_f [0] (from LSTM[0])
0x7	l2/lateral/W (1)	reccurent_f [0] (from LSTM[1])
0x8	l2/upward/b (0)	forget gate [0]
0x9	l2/upward/W (2)	mem. cell [0] (LSTM_IN[0])
0xA	l2/upward/W (3)	mem. cell [0] (LSTM_IN[1])
0xB	l2/lateral/W (2)	reccurent_s [0] (LSTM[0])
0xC	l2/lateral/W (3)	reccurent_s [0] (LSTM[1])
0xD	l2/upward/b (1)	mem. cell [0]
0xE	l2/upward/W (4)	input gate [0] (LSTM_IN[0])
0xF	l2/upward/W (5)	input gate [0] (LSTM_IN[1])
0x10	l2/lateral/W (4)	reccurent_i [0] (LSTM[0])
0x11	l2/lateral/W (5)	reccurent_i [0] (LSTM[1])
0x12	l2/upward/b (2)	input gate [0]

図 3.10　DM_W メモリマップ (1/2)

0x13	l2/upward/W (6)	output gate [0] (LSTM_IN[0])
0x14	l2/upward/W (7)	output gate [0] (LSTM_IN[1])
0x15	l2/lateral/W (6)	reccurent_o [0] (LSTM[0])
0x16	l2/lateral/W (7)	reccurent_o [0] (LSTM[1])
0x17	l2/upward/b (3)	output gate [0]
0x18	l2/upward/W (8)	foget gate [1](LSTM_IN[0])
0x19	l2/upward/W (9)	forget gate [1](LSTM_IN[1])
0x1A	l2/lateral/W (8)	reccurent_f [1] (from LSTM[0])
0x1B	l2/lateral/W (9)	reccurent_f [1] (from LSTM[1])
0x1C	l2/upward/b (4)	forget gate [1]
0x1D	l2/upward/W (10)	mem. cell [1] (LSTM_IN[0])
0x1E	l2/upward/W (11)	mem. cell [1] (LSTM_IN[1])
0x1F	l2/lateral/W (10)	reccurent_s [1] (LSTM[0])
0x20	l2/lateral/W (11)	reccurent_s [1] (LSTM[1])
0x21	l2/upward/b (5)	mem. cell [1]
0x22	l2/upward/W (12)	input gate [1] (LSTM_IN[0])
0x23	l2/upward/W (13)	input gate [1] (LSTM_IN[1])
0x24	l2/lateral/W (12)	reccurent_i [1] (LSTM[0])
0x25	l2/lateral/W (13)	reccurent_i [1] (LSTM[1])
0x26	l2/upward/b (6)	input gate [1]
0x27	l2/upward/W (14)	output gate [1] (LSTM_IN[0])
0x28	l2/upward/W (15)	output gate [1] (LSTM_IN[1])
0x29	l2/lateral/W (14)	reccurent_o [1] (LSTM[0])
0x2A	l2/lateral/W (15)	reccurent_o [1] (LSTM[1])
0x2B	l2/upward/b (7)	output gate [1]
0x2C	l3/W [0]	output layer
0x2D	l3/W [1]	output layer
0x2E	l3/b	output layer
0x2F	s [0]	mem.cell (LSTM 0)
0x30	s [1]	mem.cell (LSTM 1)

図 3.11　DM_W メモリマップ (2/2)

3.4 メモリ構成

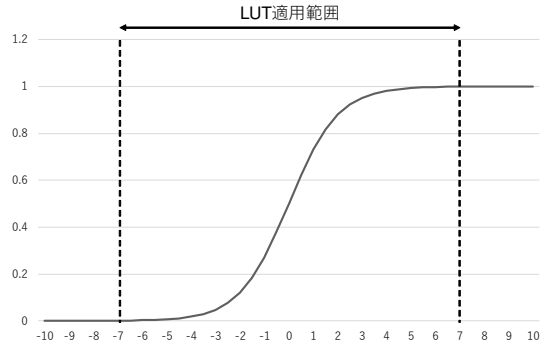


図 3.12　シグモイド関数のルックアップ

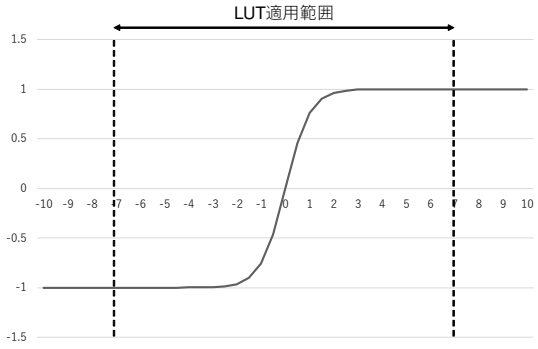


図 3.13　$tanh$ のルックアップ

address		LUT(sigmoid)	
@00001010	(0.625)	0000001010011011	(0.651354865)
@00001011	(0.6875)	0000001010101001	(0.665410559)
@00001100	(0.75)	0000001010110111	(0.679178699)
@00001101	(0.8125)	0000001011000101	(0.692641983)
@00001110	(0.875)	0000001011010011	(0.705785028)
@00001111	(0.9375)	0000001011100000	(0.718594393)
@00010000	(1)	0000001011101101	(0.731058579)
@00010001	(1.0625)	0000001011111001	(0.743168009)
@00010010	(1.125)	0000001100000101	(0.754914987)
@00010011	(1.1875)	0000001100010001	(0.766293643)
@00010100	(1.25)	0000001100011100	(0.777299861)

図 3.14　LUT(シグモイド関数)

LUT は図 3.1 中の WB ステージに設置したメモリである．メモリマップの一部を図 3.14 に示す．

値の変動が大きい箇所に絞ってルックアップを行うことで LUT に使用するメモリサイズを削減している．シグモイド関数および $tanh$ の場合は $|x| < 7$ でのルックアップを行う．

LUT にアクセスする際のアドレスは，非線形関数の被適用データである．この 16bit データの内，整数部の 4bit と小数部の 4bit を用いた 8bit の値をアドレスとして使用する．

この方法でルックアップを行った場合のシグモイド関数と $tanh$ と，32bit 浮動小数点数で式 3.5，3.6 を用いて計算した場合の値を比較した結果を表 3.2 に示す．

表 3.2　LUT 使用時における非線形関数適用値の誤差

	シグモイド関数	$tanh$
平均相対誤差 (%)	1.77	0.06

3.5　マルチコアアーキテクチャ

本提案では，マルチコア回路を C 個のコアによるクラスタで構成する．クラスタ内の回路構成を図 3.15 に示す．また，マルチコアに対応するために I/O 機構を拡張したシングルコアアーキテクチャを図 3.20 に示す．

3.5.1　設計方針

マルチコアアーキテクチャの設計にあたって，以下の項目に着目した．

- プログラムおよびデータの割当て・実行スケジューリング方式
- コア間通信同期オーバーヘッドの最小化
- メモリ使用量の最小化

動的スケジューリングを行う回路を実装するとその分回路規模が必要以上に大きくなるため，実行スケジューリング方式は，静的スケジューリングとする．DNC の計算に特化したアクセラレータであるので，ネットワーク構成や外部メモリ規模が決まれば，その構成に合わせて命令メモリの内容 (プログラム) および，データメモリの内容 (データ) も決定する．よって提案回路は静的スケジューリングとの相性が良いと言える．

マルチコア化に伴い，コア間で計算結果を送受信する必要がある．本提案では各コアへの命令およびデータの割り当て方法として，データ並列負荷分散方法 [17] を採用する．この方法による，通信オーバーヘッドを考慮しない場合のニューロン数毎のスケーラビリティは図 3.16 のように表すことができる．ニューロン数が増えるほど，スケーラビリティは向上し，マルチコア化による並列演算が効果的に行われることがわかる．よって，コア間の通信

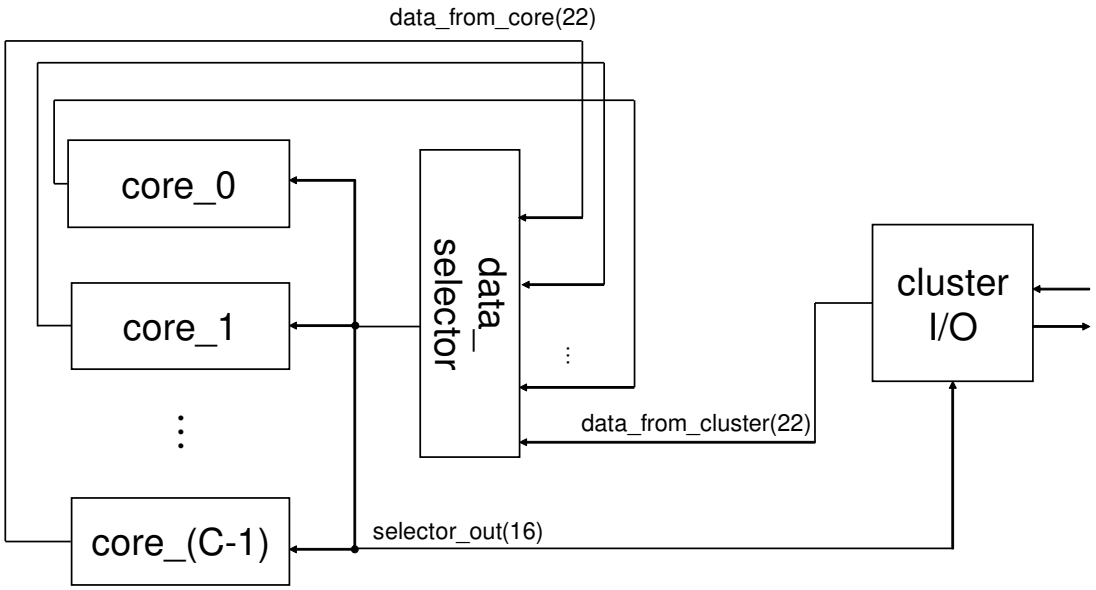


図 3.15　クラスタ構成

オーバーヘッドを最小限に保つことで，総計算時間図 3.16 で示すような理想値に近づける必要がある．

データ並列負荷分散方法を適用した場合，ネットワーク構成の分割方法と，各コアへの命令およびデータ割り当てのイメージは図 3.17 のようになる．図中に示す通り，ネットワーク内のニューロンを各層で分割し，各コアで計算を行う．

以上の方針に従って，マルチコアアーキテクチャの設計を行った．以下より，負荷分散方法，通信用コア間結合回路の仕様および，静的に行う命令スケジューリングについての詳細を述べる．

3.5.2　負荷分散方法

各コアへの負荷分散方法は，図 3.18 のように，演算対象ベクトルを分割し各コアに分配する方法 (データ並列負荷分散方法) を採用する．よって演算対象ベクトルの大きさを n とすると，コア間通信コストを考慮しない場合，演算に要する clock cycle 数は $n+3$ から $(n+3)/C$ へ短縮される (C:コア数)．LSTM Block の場合は図 3.19 のように LSTM Block

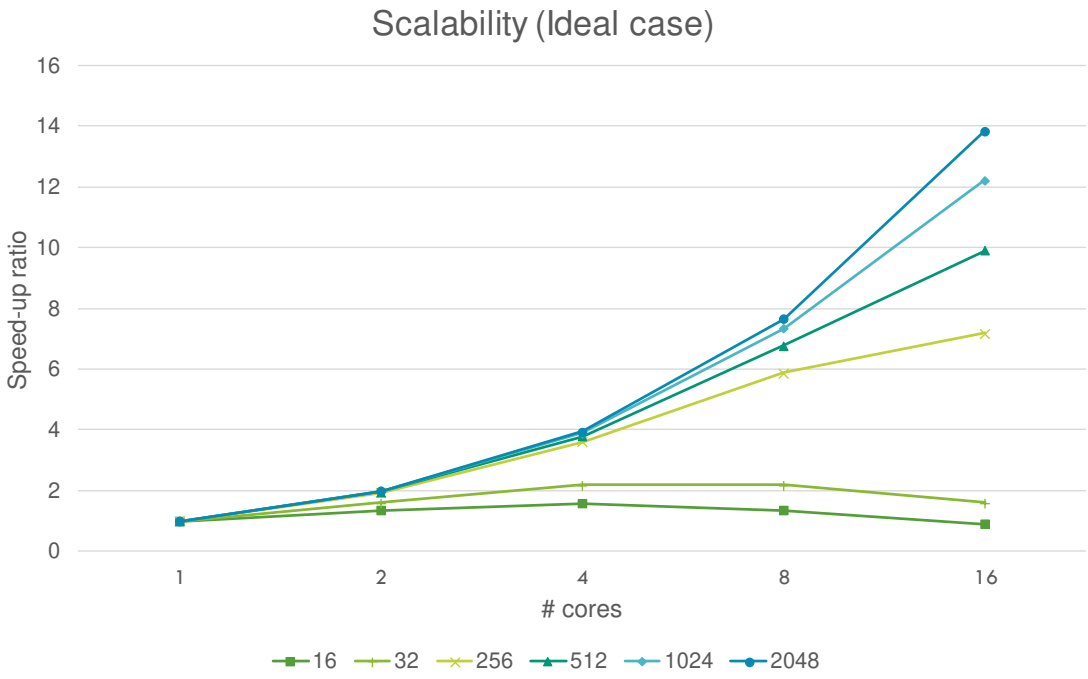


図 3.16　データ並列負荷分散方法におけるニューロン数毎のスケーラビリティ(理想値)

単位でコアへ分割する．実際にはコア間通信コストとして，SEND-RECEIVE 命令の実行に要する時間や NOP 命令を用いた送受信同期待ちが発生するため，理想値と比較してスケーラビリティは低下する．

3.5.3　コア間結合回路

コア間を結合している回路 (図 3.15 の data_selector) は，各コアから送られてきたデータの中から，SEND 命令によって送信されているデータを選択する回路である．各コアから結合回路に向けて送られているデータは先頭 6bit に命令ビットを持つため，この命令ビットで判断を行う．

data_selector 回路の実態は，入力されてくるデータの先頭 6bit の情報を選択信号として扱うマルチプレクサである．各コアから出力されている 22bit データ (data_from_core) は，送信対象データ (16bit) の先頭に命令ビット (6bit) を付与したものである．data_selector 回

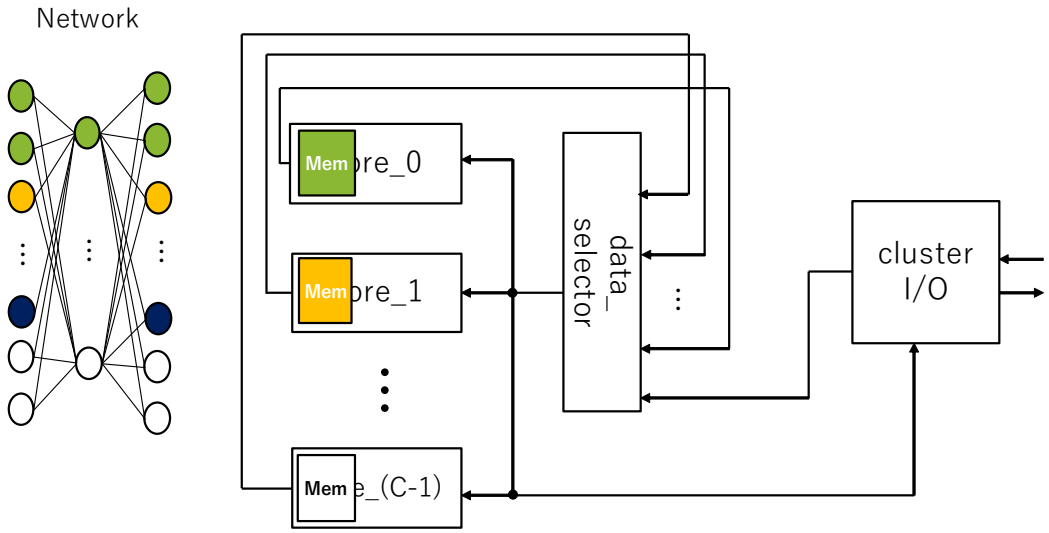


図 3.17　マルチコアアーキテクチャレベルでのデータ並列負荷分散

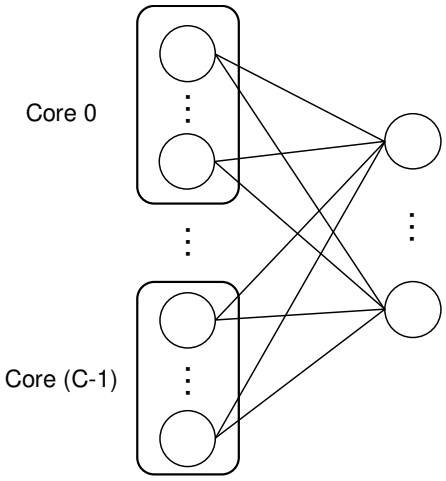


図 3.18　通常ニューロン負荷分散法 (コア数:C)

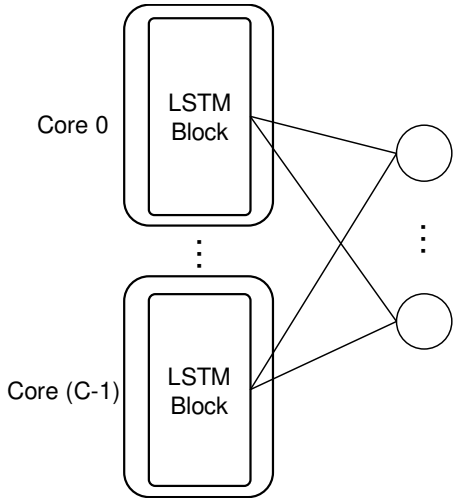


図 3.19　LSTM Block 負荷分散法 (コア数:C)

路では，このデータの先頭 6bit が SEND 命令を表す”000010”である入力データを選択し，後半 16bit のみを出力している．

data_selector 回路で選択されたデータは全コアに送られるが，この時 RECEIVE 命令を実行しているコアが選択データを自コア内のデータメモリに書きこむことで，1 対 1 および 1 対複数コアの通信を各コアが任意に行うことが可能である．よって 1 対 1 通信でも，ブロードキャスト通信を含む 1 対複数通信の場合でも，送受信に要する時間は変わらない．

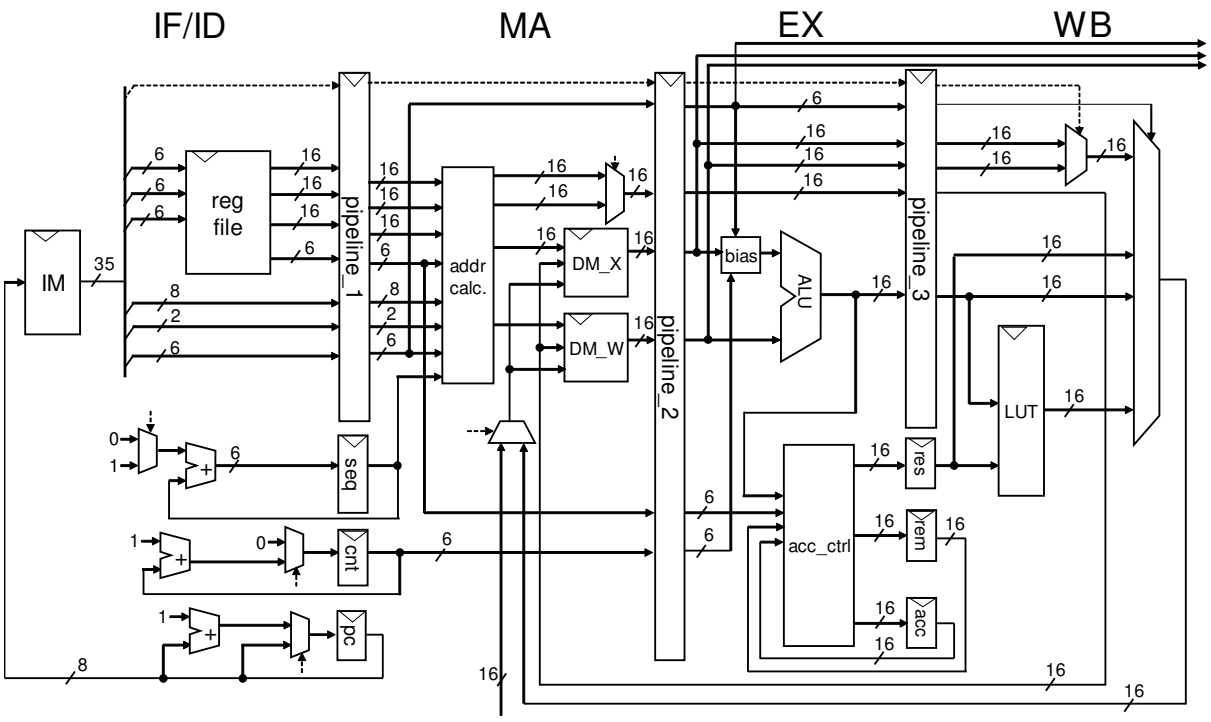


図 3.20　マルチコア用に拡張したシングルコアアーキテクチャ

3.5.4　命令スケジューリング

事前に学習済みの重みおよびバイアスを使用して計算を行う場合，あらかじめ実行するネットワーク構成は事前の学習に使用した構成と同じものである必要があるため，アクセラレータで実行する命令はあらかじめ決定され，動的にスケジューリングを行う必要がない．

よってネットワーク演算実行およびコア間データ送受信は，命令レベルで静的にスケジューリングを行うことにより，動的スケジューリングを行う場合に要する回路コストや演算コストを削減する．図 3.21 に，2 コア間で送受信を行う場合の命令発行タイミングを示す．

図 3.21 に示す通り，core 0 から送信されたデータを core 1 で受信する場合，core 0 で SEND 命令が実行された次の clock cycle で data_selector 回路に送信データが到着し，選択されるため，このタイミングで core 1 は RECEIVE 命令を実行する必要がある．これにより通信に要する時間は 2 clock cycle となる．

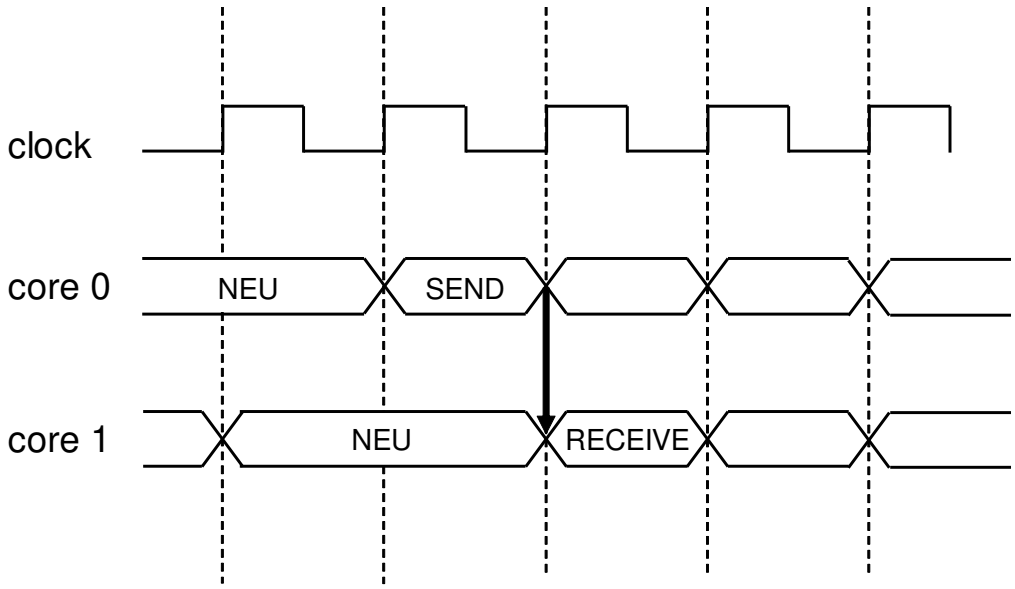


図 3.21　core0 から core1 へデータ送信を行う場合の送受信命令タイミング

スケジューリングを行う前の各コアの命令実行タイミングの様子を図 3.22 に示す．受信コア (コア 1) の計算中に送信コア (コア 0) が SEND 命令を実行しているため，コア 1 はデータを受信することが不可能である．そのため，図 3.23 に示すように NOP 命令を挿入することでタイミングを調整する必要がある．よってこの場合，コア 0 の計算完了後からコア 1 にデータが送信されるまで 4 clock cycle を要することになる．ただし，実行可能なその他の命令を NOP 命令の代わりに実行することで全体のスケーラビリティを向上させることも可能である．

図 3.24 は，図 3.18 のような通常ニューロン 32 個から成るネットワークに対して，同時刻に 8 個のコアで積和演算を始めた場合の，各コアでの命令スケジューリング例である．時刻 t に，NEU 命令が完了したと仮定する．NEU 命令の括弧内の数字は，そのコアが処理したベクトル演算の長さを表している．この場合は 32 個のニューロンを 8 個のコアで計算するので，1 コアあたり 4 個のニューロンを担当することとする．RECEIVE 命令の括弧内の数字は，どのコアから送られてきたデータを受信しているかを表している．

先に述べた通り送受信データの競合が発生するため，SEND-RECEIVE 命令は同じ時刻

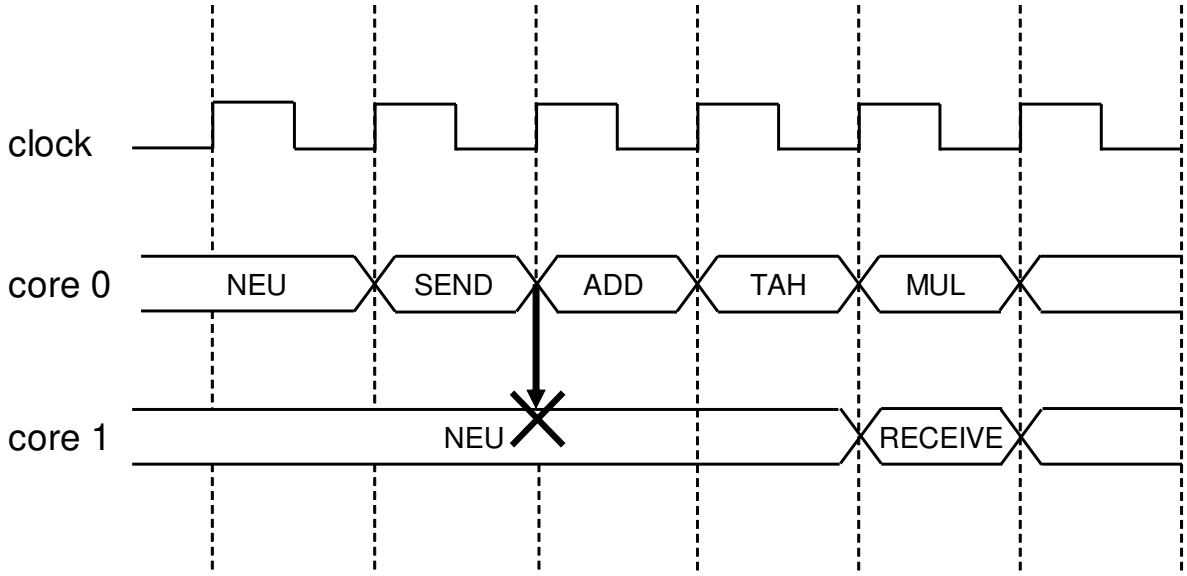


図 3.22　送受信命令タイミング (スケジューリング前)

に実行することができない．そのため，SEND-RECEIVE 命令の同期待ち時間が発生している．この影響で，積和演算開始から結果が得られるまでに非線形関数の適用も含めて，27 clock cycle を要することになるが，同じ演算内容をシングルコアで実行した場合，35 clock cycle を要することになる．よってこの場合コア間での送受信は，シングルコア内で 32 ニューロンの計算を 1 命令で行う時間よりも短く済んでいることが分かる．

実際は送受信同期待ち時間の間にも，SEND 命令，RECEIVE 命令以外の命令は実行可能であるので，より高効率に計算が可能になる見込みである．

3.6　命令セット

要件定義を元に，提案回路で実行する命令を表 3.3, 表 3.4 のように定義した．これらの命令セットのフィールド内訳は図 3.25 に示す．

命令は 35bit の bit 列で構成されている．命令 bit 列節約のため，アドレスの指定方法はレジスタファイルを用いた，ベース + オフセット形式である．各フィールドはそれぞれ，以下の役割を持つ．

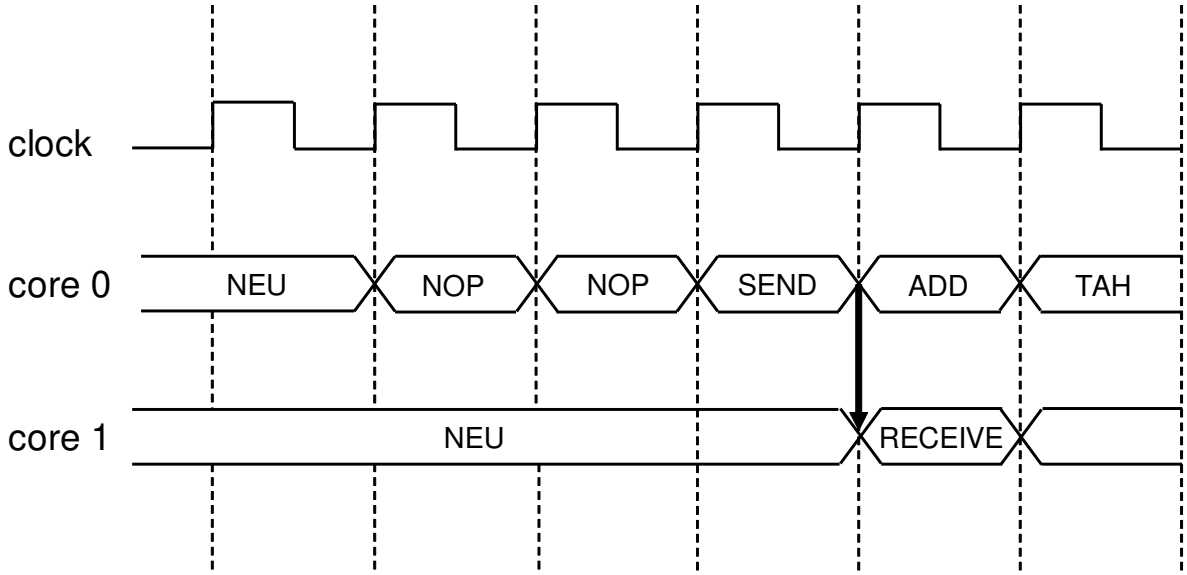


図 3.23　送受信命令タイミング (スケジューリング後)

opc　命令

seq　時系列情報 (sequence)

reg:base_x　データメモリ (DM_X) へのアドレスと計算回数を格納したレジスタの番号

reg:base_w　データメモリ (DM_W) へのアドレスと計算回数を格納したレジスタの番号

reg:base_wb　データメモリへの書き戻し用アドレスを格納したレジスタの番号

sel　書き戻すメモリの指定 (二項演算，単項演算)，読み出すメモリの指定 (単項演算)

#elem　計算対象のニューロンを識別する番号 (neuron ID)

reg:base_x，reg:base_w，reg:base_wb はレジスタ番号を表しており，レジスタファイルから読み出されるものは，16bit のベースアドレス (base_addr) と，6bit の計算回数 (#calc) が結合された 22bit のデータである．これを元に，データメモリへのアドレスを計算を行っている．

提案回路で動作する命令について，以下より解説する．

clock	core0	core1	core2	core3	core4	core5	core6	core7
t	NEU(4)	NEU(4)	NEU(4)	NEU(4)	NEU(4)	NEU(4)	NEU(4)	NEU(4)
t+1	SEND							
t+2		RECEIVE(0)						
t+3		ADD(0+1)=α	SEND					
t+4				RECEIVE(2)				
t+5				ADD(2+3)=β	SEND			
t+6						RECEIVE(4)		
t+7						ADD(4+5)=γ		
t+8							SEND	
t+9								RECEIVE(6)
t+10		SEND						ADD(6+7)=δ
t+11				RECEIVE(1)				
t+12				ADD($\alpha+\beta$)=ε		SEND		
t+13								RECEIVE(5)
t+14				SEND				ADD($\gamma+\delta$)=ζ
t+15	RECEIVE(3)							
t+16								SEND
t+17	RECEIVE(7)							
t+18	ADD($\varepsilon+\zeta$)							

図 3.24　8 コアにおける送受信命令スケジューリング例

3.6.1　二項演算命令 (積和演算命令)

この命令は，図 3.27 に示すような機構で計算を実行する．積和演算を行う場合の命令は以下の様に記述する．PSM 命令以外の命令に関しては，データメモリ容量の節約およびアドレッシングの単純化の為，バイアス項を考慮した積和演算命令としている．例えば，図 3.26 のようなバイアス項を持つニューロンを計算する場合の挙動は，図 3.28 のようになる (図 3.26 中の b はバイアス項を指す)．図に示す通り，二項演算の場合はオペランドを 2 つのデータメモリから同時に取得している．

NEU seq, base_x , base_w, base_wb, sel, elem

提案回路で用いるアドレッシングモードはベース＋オフセット形式である．積和演算系命令では，base_x フィールド, base_w フィールドで指定されているレジスタ番号を用いて，レジスタファイルからそれぞれのベースアドレスを読み出す．このとき，実行命令での計算回数 (calc_num)，つまりアキュムレートを行う回数も同じレジスタファイルから読み出している．calc_num で指定された回数だけオフセットをインクリメントすることで，1 命令の

表 3.3　命令一覧 (ベクトル演算系)

命令	opecode	説明
NEU	001011	バイアス有り積和演算
NEU_S	000111	積和 + sigmoid
NEU_T	001000	積和 + tanh
NEU_R	001010	積和 + ReLU
PSM	001001	バイアス無し積和演算

表 3.4　命令一覧 (スカラ演算系，その他)

命令	opecode	説明
NOP	111111	何もしない
CP	010011	コピー
ADD	100000	和
SUB	100001	差
MUL	100011	積
TAH	100100	tanh
ONE	100110	oneplus
RELU	100111	ReLU
SIG	101000	sigmoid
SEND	000010	送信
RECEIVE	000001	受信

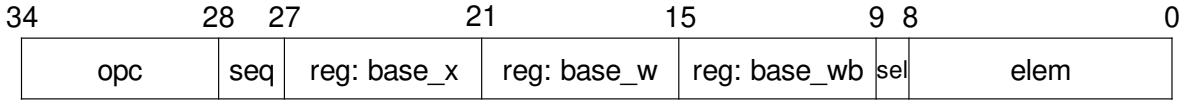
実行中に，データメモリに連続してアクセスする．

seq は，時系列データを扱う際に使用する 2bit の情報である．2bit の値のうち MSB を sequence_flg，LSB を usage_flg としている．sequence_flg が有効である場合は，回路内の seq レジスタに 1 を加算する．usage_flg が有効である場合は，命令実行時，回路内の seq レジスタの値に，”現在の seq レジスタの値 × calc_num(または calc_num-1)”を加算することで，現在計算している時刻に対応した入力データにアクセスする．

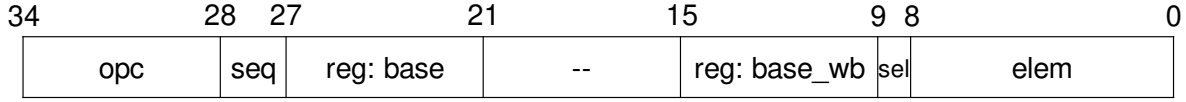
また，NEU_S 命令および NEU_T 命令は，積和演算結果に非線形関数を同時適用する命令である．これにより LSTM Block 内の各ゲートの値を 1 命令で算出可能にしている．

ADD 命令などの単純な二項演算命令も同様の記述方法である．ただし，これらの計算を行う場合は，base_x フィールド, base_w で得られたベースアドレスに，オフセットを表す elem の値を加算した値がデータメモリへのアドレスとなる．

二項演算命令



単項演算命令 (非線形関数，コピー)



送受信命令

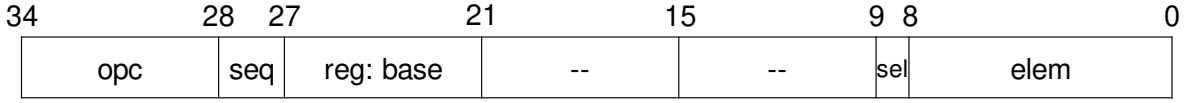


図 3.25　命令フィールド内訳

計算完了後，データメモリに結果を書き戻す際のアドレスは，base_wb フィールドで指定されているレジスタから取得したベースアドレスに elem を加算した値となる．また，書き戻すデータメモリは sel フィールドで指定する．

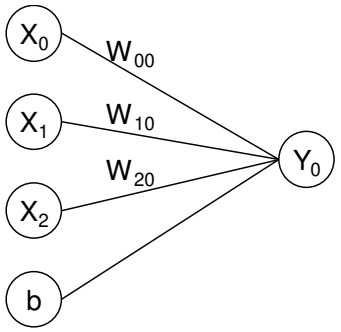


図 3.26　演算対象ニューロンの例

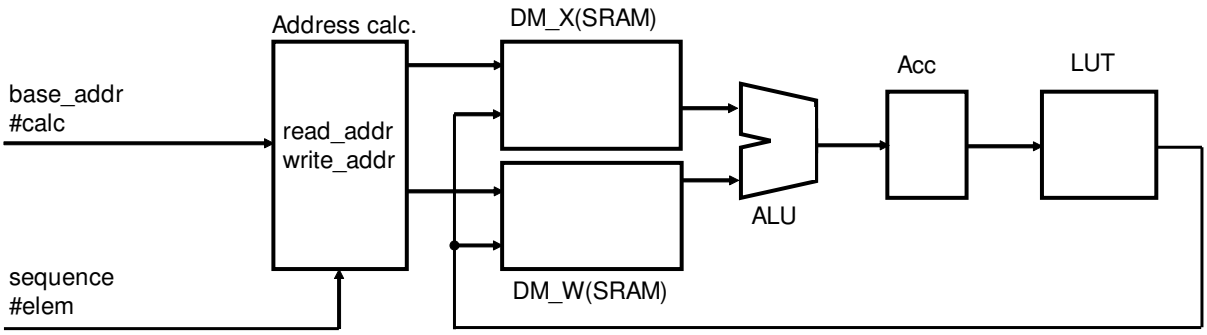


図 3.27　積和演算+LUT 計算部分のアーキテクチャ概略

3.6.2　単項演算命令

ある値に非線形関数を適用する場合や同データメモリ内への値のコピー等，単項演算を行う場合の命令は以下の様に記述する．

```
CP seq, base, base_wb, sel, elem
```

この記述で実行される命令は，一部の二項演算命令と同様にデータメモリに対する読み出しアドレス，書き戻しアドレス共に elem で指定した値を，ベースアドレスに対するオフセットとして扱う．データメモリの指定は sel で行う．

3.6.3　送受信命令

コア間送受信を行う場合の命令は以下の様に記述する．この命令は，マルチコア化の際にコア間でデータをやり取りする必要がある場合に使用する．

```
SEND seq, base, sel, elem
```

SEND 命令の場合は base で指定したレジスタファイルから読み出したベースアドレスに elem を加算してアドレスを求め，sel で指定したデータメモリから値を読み出す．その後，

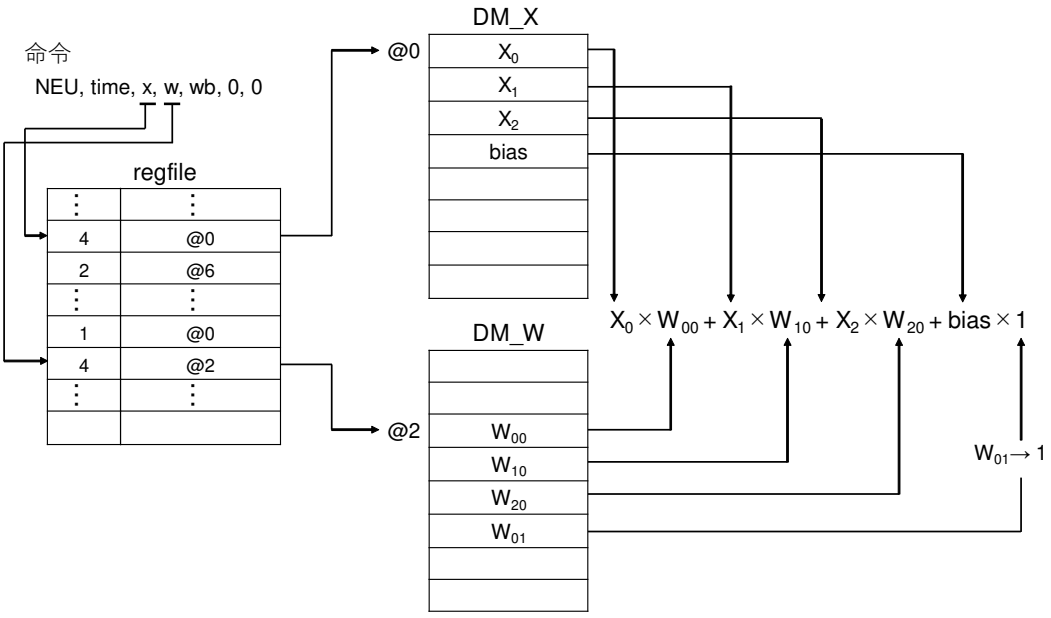


図 3.28　積和演算系命令の挙動 (NEU 命令の場合)

読み出した値はコア間結合回路へ送信される．

RECEIVE 命令の場合は，SEND 命令と同様の方法で計算したアドレスを書き戻しアドレスとして扱う．他コアが実行した SEND 命令によって送信されてきたデータを，自コアで実行した RECEIVE 命令で受信することで，コア間通信を実現している．

3.7　マルチコア用スケジューラ

各コアの命令メモリに命令を格納するため，コア間通信命令を含めた LSTM 実行用命令を各コアで実行する際のスケジューリングを事前に行う必要がある．

本研究でのスケジューリングアルゴリズムは，最も単純な方式を採用した．これは，図 3.29 に示すような方式である．各コアが一斉に LSTM 演算を開始し，各コアでの LSTM 演算が終了した後，一斉に通信を開始する方式をとった．通信が完了したら再び一斉に次の LSTM 演算を開始し，以下、この繰り返しによって演算を進めていく．

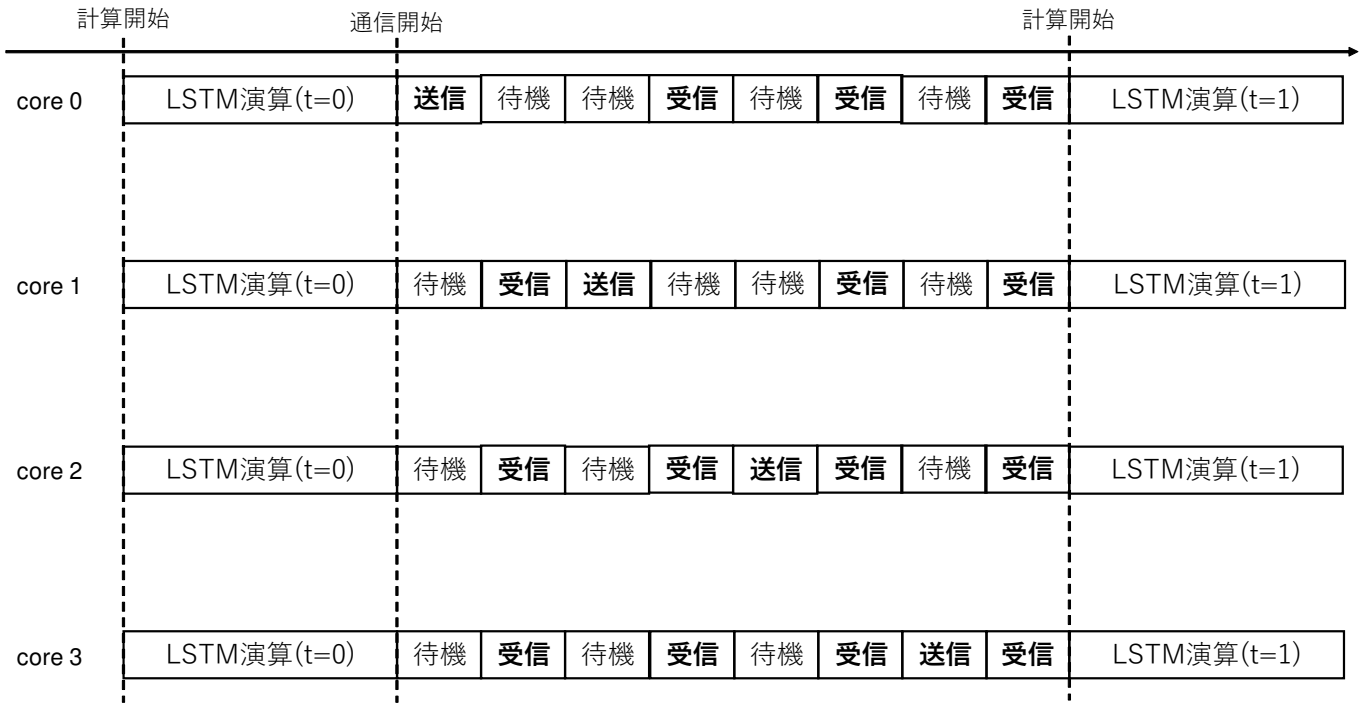


図 3.29　採用スケジューリングアルゴリズム (4 コアの場合)

スケジューラは，通信命令のみに注目し，以下の状況が発生していないかを判定する．

- 同時刻に，複数コアが送信命令を実行している
- 他のコアが受信命令を実行中に送信命令を実行している
- 実行している送信命令の送信内容と，次時刻に実行される受信命令の内容が一致していない
- 送信命令の直後に正しい受信命令が実行されていない

同時刻に送受信命令が複数コアで実行されていると，コア間通信回路でデータの衝突が発生し，正常なデータのやり取りが不可能になる．また，コア間通信回路にデータを保持しておく機能が無いため，送信命令が実行された場合，次の時刻に実行する受信命令は必ず，前時刻で実行された送信命令で送信されたデータを受信するための命令を実行する必要がある．よってスケジューラでは，これらの観点について確認し，必要に応じて命令を入れ替え

たり NOP 命令を挿入する必要がある．

3.8 専用 HDL 高位合成ツール

設計したシングルコアをマルチコアに拡張する際に，十数コア規模以上になると，その分のマルチコア回路およびコア間接続回路を HDL によって記述する必要がある．

そこで，マルチコア化するコア数を指定して実行することで，指定したコア数に応じたマルチコア回路の Verilog-HDL 記述を生成する，専用の HDL 高位合成ツールを python で試作した．以降の評価では，このツールを用いて生成および合成したマルチコア回路での評価を行う．

例えば，4 コアのマルチコア構成の HDL 記述 (ファイル名 : core4.v) を得る場合は，以下のように使用する．

```
python connect_generator.py 4 > core4.v
```

上記の実行により得られた HDL 記述のうち，コア間合成回路に相当する箇所を以下に示す．

```
function [`DATA-1:0] data_select;
  input [`SEND_DATA-1:0] data_to_0;
  input [`SEND_DATA-1:0] data_to_1;
  input [`SEND_DATA-1:0] data_to_2;
  input [`SEND_DATA-1:0] data_to_3;
  begin
    if(data_to_0[`SEND_DATA-1:`DATA] == `SEND)
      data_select = data_to_0[`DATA-1:0];
    else if (data_to_1[`SEND_DATA-1:`DATA] == `SEND)
      data_select = data_to_1[`DATA-1:0];
    else if (data_to_2[`SEND_DATA-1:`DATA] == `SEND)
      data_select = data_to_2[`DATA-1:0];
    else if (data_to_3[`SEND_DATA-1:`DATA] == `SEND)
      data_select = data_to_3[`DATA-1:0];
    else
```

```
        data_select = 16'd0;
    end
  endfunction

assign selector_out = data_select(data_to_core_0,
  data_to_core_1, data_to_core_2, data_to_core_3);
```

3.9 結言

本章では，提案アクセラレータのシングルコアアーキテクチャと，マルチコアに対する負荷分散方法，その実現のための命令スケジューリングについて述べた．1 つのコアはパイプライン構造をとり，2 つのデータメモリからオペランドを同時に読み出して計算を行う．マルチコアでは，コア間通信を制御するための方法として命令レベルでのスケジューリングの最適化を行うことで通信コストを下げることを目的とした．

また，コア間通信用スケジューラおよび，試作した専用 HDL 高位合成ツールについても述べた．以降の評価ではこれらを使用して設計したマルチコアに関して評価を行う．

第 4 章

評価

4.1 緒言

本章では，提案マルチコアアクセラータの性能および回路規模に関する評価を行った結果を述べる．本研究では多様なエッジデバイス用途に適用可能な FPGA 実装を想定している．Intel 社製 FPGA Cyclne IV を想定し，開発ツール Quartus Prime を用いた論理合成により，回路規模（LE 数）を評価した．また，提案マルチコア構成用命令スケジューラ，および，アセンブラを開発し，任意のコア数で任意の規模の LSTM ネットワークが実行可能な開発環境を試作している．

4.2 回路規模

表 4.1 に，コア数毎の回路規模を示す．想定 FPGA の総 LE 数は 114,480 個，総メモリ量は 592,896 bits である．

マルチコアにおける使用メモリ bit 数は，シングルコアにおける使用メモリ bit 数 × コア数となった．コア数 1〜8 の場合は，平均して最大約 73.00MHz で動作することが判明したが，16 コアになると最大動作周波数は 58.6MHz まで下がった．これは，クラスタ内のマルチプレクサ (図 3.15 中の data_selector) の回路規模の拡大に伴って LE 間の配線距離が長くなってしまうためであると考えられる．LE 間の配線距離が長くなることで，配線上での情報伝達に遅延が発生する．

表 4.1 試作マルチコア構成の回路規模 (型番 EP4CE6U14l7)

コア数	LE 数 [個]	メモリ使用量 [bits]	最大動作周波数 [MHz]
1	779 (1%)	148K (4%)	77.07
2	1,549 (1%)	296K (7%)	72.54
4	3,080 (3%)	592K (15%)	71.1
8	6,111 (5%)	1,185K (30%)	71.31
16	12,224 (11%)	2,371K (60%)	58.6

4.3 実行時間

図 4.1 と同様の構成の LSTM ネットワーク実行に要した clock cycle 数を表 4.2 に示す. 動作検証用プログラムでは，コア数が 1 から 2 になると，実行 clock cycle 数は 18 短縮されたが，これは，積和演算命令を並列に実行することで短縮された clock cycle 数と，各コアで行う送受信命令を実行することで発生する clock cycle 数の差分となる．この差分は命令スケジューリングを最適化して並列度を上げることで，改善される可能性がある.

以下は動作検証用の LSTM ネットワークのアセンブリ記述 (シングルコア用) である. ネットワーク構成は図 4.1 の通りである．なお視認性優先のため，NOP 命令は除いている.

```
NEU,1,x,l1_0,lstm_in,0,0
NEU,1,x,l1_1,lstm_in,0,1
NEU_T,0,lstm_in,w_memcell_0,memcell,1,0
NEU_S,0,lstm_in,w_input gate_0,input gate,0,0
NEU_S,0,lstm_in,w_forget gate_0,forget gate,0,0
MUL,0,input gate,memcell,memcell,1,0
NEU_S,0,lstm_in,w_output gate_0,output gate,0,0
ADD,0,forget gate,memcell,memcell,1,0
NEU_S,0,lstm_in,w_forget gate_1,forget gate,0,1
TANH,0,,memcell,memcell,1,0
NEU_T,0,lstm_in,w_memcell_1,memcell,1,1
MUL,0,output gate,memcell,lstm_out,0,0
NEU_S,0,lstm_in,w_input gate_1,input gate,0,1
```

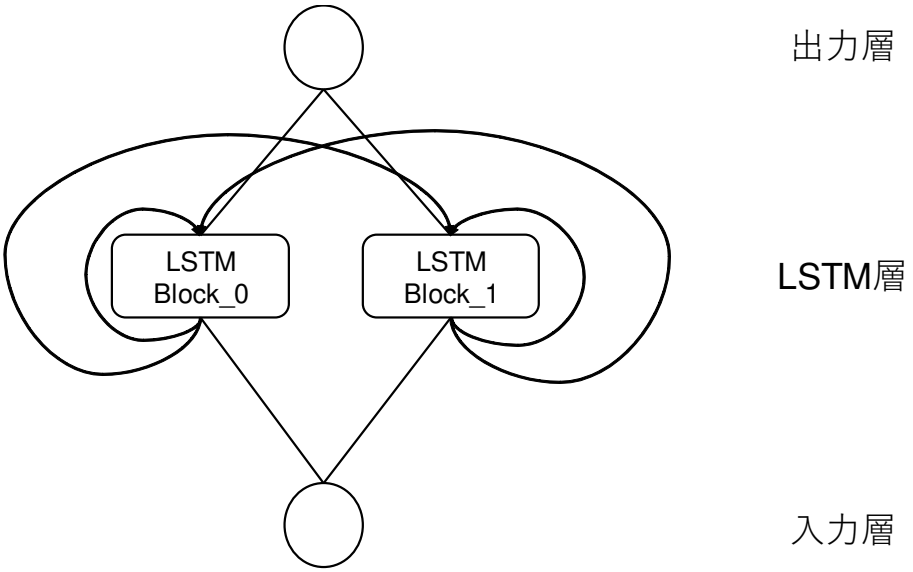


図 4.1 動作検証用 LSTM ネットワーク構成

表 4.2 コア数毎の計算時間比較 (予備評価)

コア数	clock cycle 数
1	69
2	51

```
WRITE,0,lstm_out,0,lstm_in,0,2
MUL,0,input gate,memcell,memcell,1,1
NEU_S,0,lstm_in,w_output gate_1,output gate,0,1
ADD,0,forget gate,memcell,memcell,1,1
TANH,0,,memcell,memcell,1,1
MUL,0,output gate,memcell,lstm_out,0,1
WRITE,0,lstm_out,1,lstm_in,0,3
NEU,3,lstm_out,l3,y,0,0
```

更に LSTM ネットワークの規模を拡大した場合における，計算時間と通信時間を概算した．概算において，各コアで実行する命令のスケジューリングを行った．スケジューリングアルゴリズムとしては，各コアで LSTM 層/コア数 個の LSTM Block を計算し，全コア

表 4.3　コア数毎の実行 clock cylcle 数と実行命令数

コア数	1 コアあたり実行時間	計算時間	通信時間	実行命令数
1	2039	2039	0	612
2	780	768	12	373
4	362	326	36	431
8	240	156	84	751
16	269	89	180	1493

で LSTM Block の計算が完了したら順に送受信を開始するという最もシンプルなアルゴリズムである．概算に使用したネットワーク構成は図 4.1 の各層を以下の様に改変したものである．

入力層　16

LSTM 層　32

出力層　16

結果は表 4.3 の様になった．表中の計算時間および通信時間の単位は [clock cycle 数](1 コアあたり実行時間，計算時間，通信時間) および，[個](実行命令数) である．

表 4.3 における計算時間と通信時間の比率を求めた結果が図 4.2 である．処理するニューロン数および LSTM Block 数と，並列コア数が同程度になると通信時間が計算時間を上回るため，通信オーバーヘッドが総処理時間の大半を占めてしまうことになる．

4.4　結言

本章では，試作マルチコアアクセラレータに関して，回路規模の計測や，動作検証用プログラムの実行に要する時間を評価した．動作検証用の LSTM ネットワークを 2 コアで計算した場合，1 コアで同様の LSTM ネットワークを計算する場合と比較して 18%の clock

4.4 結言

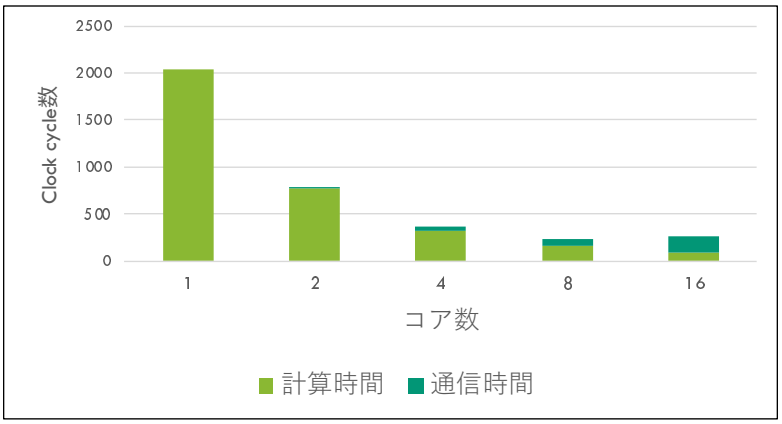


図 4.2 コア数毎の計算 clock cycle 数と通信 clock cycle 数の関係

cycle 数の減少が確認できた.

第 5 章

結論

本研究では，DNC(Differentiable Neural Computer) の高速化を目的とした，ハードウェアマルチコアアクセラレータの検討を行った．要件定義に沿って試作回路を設計し，その性能を最大動作周波数および，実行に要する総 clock cycle 数，マルチコアで実行した場合におけるコア間通信コスト関して検証を行った．

DNC は長文やグラフなど，複雑なデータ構造の学習が可能な DNN アルゴリズムの 1 つである．コントローラネットワーク (LSTM) に，学習結果を保存しておく外部メモリを結合させた構造をしている．そのため DNC は，LSTM の内部状態がより複雑になったリカレントニューラルネットワークととらえることも可能である．そこで本研究ではまずコントローラネットワークの高速化の実現を目指した．

コントローラネットワークは LSTM ネットワークを使用しているため，時系列データを管理する機構や，ハイパパラメータ群の効率的な格納および読み出しを工夫する必要がある．ニューロンの計算方法としては，通常ニューロン，LSTM ブロックの 2 通りの計算が行えること，ネットワーク構成としては多層ネットワークかつ，リカレントネットワークが計算可能であることを要件として定義した．更に DNC の特徴である，外部メモリ行列に対する読み取りと書き込みが行える機能も必要となる．

以上の要件を受けて，提案回路は，ニューラルネットワークにおける積和演算と非線形関数の適用を 1 命令で実行する 4 段パイプライン構造を持つ構造とした．更にこの回路をマルチコア化することで，性能の向上を図った．また本研究では，命令を静的にスケジューリングするためのマルチコアスケジューラや，コア数に応じてマルチコア構成の HDL(Verilog-HDL) 記述を生成するための専用 HDL 高位合成ツールの作成を行った．

本論文では，DNC 向きハードウェアマルチコアアクセラレータを設計するにあたっての要件定義を行い，提案したアーキテクチャ，メモリマップ，命令セットについて重点的に述べた．設計後の評価として，DNC のコントローラネットワーク部分である LSTM を実行する命令を作成し，計算に要する時間を確かめた他，計算時間を算出するシミュレータを作成してより大規模な LSTM の場合における計算時間の概算を行った．また，2 コア〜16 コアで合成を行った場合の回路リソース (LE，メモリ bit) 使用率や，各コア数における最大動作周波数の確認を行った．

2 章では，本研究で取り上げる DNC について述べ，これらの演算を実現するアクセラレータの要件定義を行った．DNC の演算に関しては，コントローラネットワークである LSTM および，外部メモリへの読み書き演算を高速化することを目的とした．DNC の計算に必要な LSTM 演算を行うための命令セットを検討し，パイプライン実行可能な回路構成を採用した．

3 章では，提案アクセラレータのシングルコアアーキテクチャと，マルチコアに対する負荷分散方法，その実現のための命令スケジューリングについて述べた．シングルコアとしての主な動作は，2 つのデータメモリからオペランドを同時に読み出して計算を行い，積和演算の場合はパイプラインを停止させてアキュムレートを行うものである．

シングルコアの試作回路完成後はマルチコアに向けて，コア間通信を制御するための方法を検討し，命令レベルでのスケジューリングの最適化を行うことで通信コストを下げることを目的とした．

4 章では，試作マルチコアアクセラレータに関して，回路規模の計測や，動作検証用プログラムの実行に要する時間を評価した．動作検証用の LSTM ネットワークを 2 コアで計算した場合，1 コアで同様の LSTM ネットワークを計算する場合と比較して実行 clock cycle 数が減少していることを確認し，また，命令スケジューリングの最適化を更に行う必要があることが判明した．

今後の課題は以下の通りである．

- 提案回路のみで DNC 演算を完了できるように機能拡張
  - 除算やソートの高速化方法を検討
  - Softmax 関数や oneplus 関数などの未実装非線形関数の実装方法検討
- コア間通信におけるバースト送受信の実現
- 命令メモリ・データメモリ階層化

現時点では，DNC の計算の一部に対応できていないため，上記のような計算が可能な回路として機能拡張を行う必要がある．除算はメモリ操作の際のコンテンツルックアップ計算やコサイン類似度を求める際に必要になる．ソートはメモリへの値の割り当て量を計算する際に必要となる．

DNC の Interface Vector を生成する際に必要となる Softmax 関数，および oneplus 関数は，LUT による近似が使用できない．そのため他の高速実現方法を提案するか，計算によって求める必要がある．計算によって求める場合は，除算や自然対数の計算を実装する必要がある．除算に関しては，そのまま除算器を実装してしまうと計算に時間がかかるため，短時間で除算を実行できるように工夫する必要がある．

現在のコア間通信方法は 1 命令につき 1 データを送信する方法である．そのため通信が多発するとその分命令を発行する必要があるため命令メモリを必要以上に消費してしまう．よって 1 命令で多くのデータを送受信するためにバースト送受信方法を検討する必要がある．

また，現在の提案回路で想定している DNC は比較的小規模である．実用的な規模でのネットワーク構成で演算が実行できるようにメモリを拡張する必要がある．まずは現行アーキテクチャのまま，各メモリのサイズを拡張するという方法が考えられるが，16 コア並列の段階で全体の 6 割のメモリ bit 数を使用している状況であるため (Cyclone IV E EP4CE6U14l7 の場合)，より大規模なニューロン数およびコア数で処理を行う場合に対応することが難しいと考えられる．そこでメモリ階層化を行い，大容量メモリに必要なデータを保持させておき，各コアに必要なデータをそれぞれ分配することで 1 コアあたりのメモリ

容量を抑えつつ総保持データ容量を拡張することでより大規模なネットワーク構成にも対応できると考えられる.

謝辞

本研究を遂行するにあたり，多くの皆様にご指導・ご協力をいただきました．特に指導教員である岩田誠教授はお忙しい中でも時間を割いてくださり，非常に多くのご指導・ご助言を賜りました．心より感謝申し上げます．

また，本研究の論文の副査をお引き受けいただきました福本昌弘教授，吉田真一准教授にも，中間発表ならびに，修士論文発表会での質問やコメントを通して様々なご助言を頂き，本研究をより良いものとすることができました．心より感謝申し上げます．

研究室の同輩である福田 和馬氏には，回路設計に関する相談によく乗っていただきました．また，唯一の同輩としてお互い励ましあうことで，ここまで乗り越えてこられました．ありがとうございました．

修士課程 1 年の和田 悠伸氏，学士課程 4 年の楠田 健太氏，柳田 海志氏，汐見 興明氏，長野 寛治氏，下出 千晴氏には，主に研究室ミーティングの際に有益な質問やコメントなどしていただきました．専門外であるからこその率直な意見に本研究における課題を見出したこともあり，非常に助けになりました．楠田氏，汐見氏，長野氏におかれましては，修士課程に進まれても頑張ってください．

学士課程 3 年の井上 聡氏および西岡 周真氏にも，普段からのご支援ご協力を感謝いたします．4 年生へと進み，研究室の中核を担う立場として益々の活躍を期待しています．

参考文献

[1] Amir Yazdanbakhsh, Hajar Falahati, Philip J. Wolfe, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. "GANAX: A unified MIMD-SIMD acceleration for generative adversarial networks,". *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 650–661, 2018.

[2] Lin Bai, Yiming Zhao, and Xinming Huang. "A CNN Accelerator on FPGA Using Depthwise Separable Convolution,". *IEEE Transactions on Circuits and Systems II: Express Briefs*, 08 2018.

[3] Alex Graves, Greg Wayne, Malcom Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adria Puigdomenech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. "Hybrid computing using a neural network with dynamic external memory,". *NATURE*, Vol. 538, pp. 471–476, 2016.

[4] Arm Limited. "Arm Machine Learning Processor,". `https://developer.arm.com/products/processors/machine-learning/arm-ml-processor`.

[5] Intel Corporation. "Intel Movidius Myriad VPU 2: A Class-Defining Processor,". `https://www.movidius.com/myriad2`.

[6] Guo Kaiyuan, Zeng Shulin, Yu Jincheng, Wang Yu, and Yang Huazhong. "A Survey of FPGA Based Neural Network Accelerator,". *CoRR*, Vol. abs/1712.08934, , 2017.

[7] V. Sze, Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A tutorial and Survey,". *Proceedings of the IEEE*, Vol. 105, No. 12, pp. 2295–2329, Dec 2017.

[8] Shu-Chang Zhou, Yu-Zhi Wang, He Wen, Qin-Yao He, and Yu-Heng Zou. “Balanced Quantization: An Effective and Efficient Approach to Quantized Neural Networks,”. *Journal of Computer Science and Technology*, Vol. 32, No. 4, pp. 667–682, Jul 2017.

[9] Courbariaux Matthieu, Bengio Yoshua, and David Jean-Pierre. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations,”. *CoRR*, Vol. abs/1511.00363, , 2015.

[10] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. “Binarized Neural Networks,”. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pp. 4107–4115. Curran Associates, Inc., 2016.

[11] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. “XNOR-Net: Imagenet Classification Using Binary Convolutional Neural Networks,”. *CoRR*, Vol. abs/1603.05279, , 2016.

[12] 林遼, 森下真幸, 高田遼, 坂本龍一, 近藤正章, 中村宏. “ディープラーニング向けアクセラレータアーキテクチャの FPGA 実装, ”. Technical Report 12, 東京大学, sep 2016.

[13] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars,”. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 14–26, June 2016.

[14] 田中愛久, 黒柳奨, 岩田彰. “FPGA のためのニューラルネットワークのハードウェア化手法,”. 電子情報通信学会技術研究報告. NC, ニューロコンピューティング, Vol. 100, No. 688, pp. 175–182, mar 2001.

[15] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei, and David Brooks.

"Minerva: Enabling low-power, highly-accurate deep neural network accelerators,". *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, 2016.

[16] Gary Marcus. "Deep Learning: A Critical Appraisal,". 01 2018.

[17] 山崎尚之. "LSTM-RNN 用マルチコアアクセラレータの負荷割当方法の検討,". 高知工科大学 学士学位論文, 2018.