FUCEプロセッサにおけるスレッド管理機構

松崎, 隆哲
九州大学大学院システム情報科学府

大庭, 直行
九州大学大学院システム情報科学研究院

雨宮, 真人
九州大学大学院システム情報科学研究院

https://doi.org/10.15017/4784013

出版情報：九州大学情報基盤センター年報. 3, pp.21-30, 2003-03. 九州大学情報基盤センター
バージョン：
権利関係：

FUCEプロセッサにおけるスレッド管理機構
A Thread Control Mechanism for the FUCE Processor

松崎 隆哲† 大庭 直行‡ 雨宮 真人‡
Takanori MATSUZAKI† Naoyuki OHBA‡ Makoto AMAMIYA‡

† 九州大学 大学院システム情報科学府
† Graduate School of Information Science and Electrical Engineering, Kyushu University
‡ 九州大学 大学院 システム情報科学研究院
‡ Graduate School of Information Science and Electrical Engineering, Kyushu University

要旨 本稿は、ハードウェアでスレッドの管理をおこなうスレッド管理機構について述べる。FUCEプロセッサは、プロセッサ内部にスレッドの実行制御をおこなうユニットを用意することで、細粒度スレッドを効率よく実行することができる。これにより、多数のスレッドを実行した際のスレッド実行制御を効果的におこなうことができ、細粒度マルチスレッド実行方式を用いる場合に問題となる、スレッド管理コストが解決できる。

Abstract In this paper, we describe a new thread control mechanism which manages threads by hardware. The innovative point of this mechanism is to integrate a thread control unit into a processor, thus the FUCE processor can execute fine-grain threads efficiently. In particular, this newly introduced mechanism is designed with the mind that the FUCE processor will perform many threads concurrently, and the management cost problem of the fine-grain multi-threading should be solved.

1 はじめに

我々は今後の集積回路技術の進歩を見据え、メモリとプロセッサを同一チップに搭載し、細粒度マルチスレッド実行方式を採用したFUCE(FUsion of Comunication and Execution) プロセッサを提案してきた[1] [2]。

FUCEプロセッサは、プロセッサ内部におけるすべての処理を走り切りスレッドとして扱い、スレッドを多数のスレッド実行ユニットで並列に実行する。また、割り込み処理もスレッドとして扱う。これにより、既存のシステムで問題となる、割り込み処理によって処理が中断されることによるシステム負荷の問題を解決することができる。オペレーティングシステムにおいては、割り込み処理による処理の中断が無くなるため、システム全体を軽くすることができる。また、予測不可能な長さの遅延を生じる可能性のある処理において、処理要求と結果受け取りを分離して扱うスプリットフェーズ方式と、細粒度マルチスレッド実行方式を組み合わせて利用することで、通信と処理とをオーバラップさせ、遅延を隠蔽することができる。以上のことより、FUCEプロセッサは、ノイマン型プロセッサでは、効率的に実行できない再帰処理を、効率よくおこなうことができる。また、ネットワークにおけるスイッチやルータなどの、通信割り込みが多数発生するシステムにおいても効果的に処理をおこなうことができる。

FUCEプロセッサは、細粒度スレッドを効率よく実行するために、プロセッサ内部にスレッド管理機構を持っている。これは、プロセッサ内部にハードウェアによって構成された同期カウンタ管理表を用意し、この同期カウンタ表を用いてスレッドの実行制御をおこなうことで実現される。これにより、FUCEプロセッ

サは、多数のスレッドを同時に実行した際にスレッド実行制御を効率的におこなうことができる。また、細粒度マルチスレッド実行方式において問題となる、スレッド管理コストの問題が解決できる。

以下本稿では、2章にてFUCEプロセッサについて概略を述べ、3章においてFUCEプロセッサの内部構造について述べる。4章では、スレッド管理機構について述べ、最後に5章で本稿のまとめと今後の予定について述べる。

2 FUCEプロセッサ

2.1 概要

FUCE プロセッサは、スレッド実行ユニットとメモリを同一チップ上に混載し、複数のスレッド実行ユニットにて、多数の細粒度スレッドを同時に実行するオンチップメモリ・マルチスレッドプロセッサである。

FUCEプロセッサは、従来のプロセスをスレッドと呼ぶ排他的に実行可能なプログラム断片に細分化し、これを多重化して並列に走行させる。また、割り込み処理を含めたすべての処理をスレッドとして扱い、中断無しに実行する。これによって、通信や入出力処理などの外部イベントの処理や内部計算処理を効率的におこなう。

FUCEプロセッサでは、すべてのプロセッサ内部の処理をスレッドとして扱うために、効率よくスレッドの管理をおこなうための機構をプロセッサ内部に持っている。FUCEプロセッサは、この機構を利用して、スレッドの同期処理やスケジューリングなどのスレッド実行制御をおこなう。

図1に、FUCEプロセッサの概要図を示し、以下に特徴を述べる。

- 同一チップ上にスレッド実行ユニット、メモリ、通信ユニットを搭載
- スレッド実行ユニット、メモリ、通信ユニット間は高バンド幅の内部バス [3] で接続
- スレッドはプログラムの逐次実行の基本単位で、実行中のスレッドは、他のスレッドにより中断されない
- すべてのプロセッサ内部の処理は、スレッドとしてスレッド実行ユニットで実行される
- スレッド実行ユニットは、一つのスレッドのみを排他的に実行し、スレッドの中断や再開を行なう機能を持たない
- スレッド実行ユニットは大容量のレジスタファイルを持つ
- 効率よくスレッドを実行するために、スレッドコンテキストの先読み [4] をおこなう

これらの特徴については、後で詳細を述べる。
FUCEプロセッサの設計方針を以下に示す。

方針 (1) スレッドを基本としたプログラム実行モデル

FUCE プロセッサのすべての処理は、スレッドとして扱う。スレッドの実行は、スレッド実行ユニットにて排他的におこなう。

方針 (2) 割り込み処理もスレッドとして扱う

FUCE プロセッサは、割り込み処理によりスレッド実行の中断が発生しないことで、効率の良い細粒度スレッド実行方式を実現する。

方針 (3) 単純な構造の命令実行の多重化

命令実行ユニットの高機能化を行なわず、単純なハードウェア構成にする。命令実行ユニットの多重化を行なうことでスレッドレベルの並列度の確保を図る。

方針 (4) メモリ性能の強化

オンチップメモリと高バンド幅内部バスを用い、低レイテンシで高バンド幅なメモリシステムの実現を図る。また大容量のレジスタファイルを用意する。

方針 (5) 大容量レジスタの実現

スレッド実行ユニットは、大容量レジスタファイルを持ち、スレッドはレジスタ間のみで演算をおこなう。

方針 (6) スレッド実行の効率化

多数の細粒度スレッドを実行するため、効率のよいスレッド実行の実現を図る。

方針 (1) の「スレッドを基本としたプログラム実行モデル」に基づき、効率よくスレッドの管理をおこなうための機構を実現する。方針 (2) の「割り込み処理もスレッドとして扱う」に基づき、効率の良いスレッ

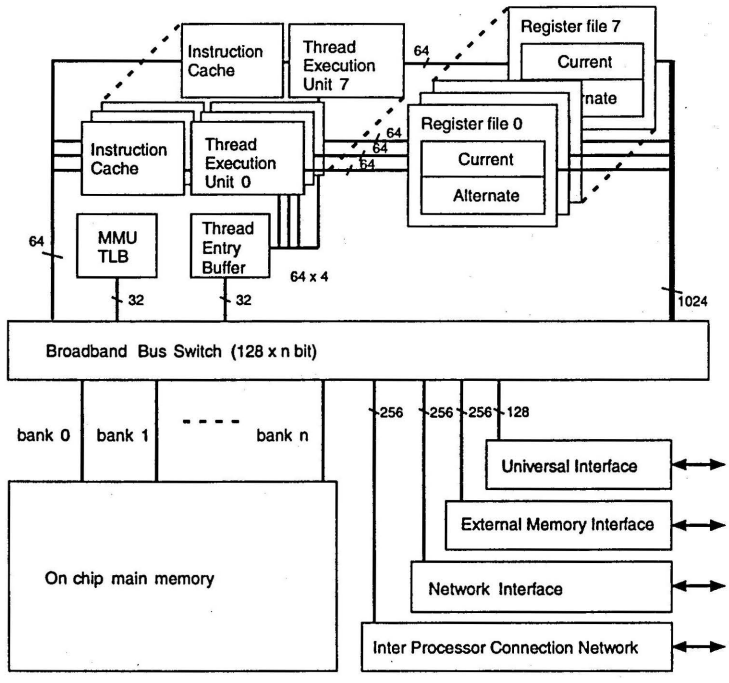

図 1: FUCE プロセッサ構成図

ド実行環境を実現する。方針 (3) の「単純な構造の命令実行ユニットの多重化」に基づき、多重化した命令実行ユニットで、複数スレッドを並列実行する。方針 (4) の「メモリ性能強化」に基づき、メモリの低レイテンシ (4 サイクル程度) と高バンド幅 (512 G byte/sec) を実現する。方針 (5) の「大容量レジスタの実現」に基づき、スレッドはレジスタ内のデータだけで実行される。方針 (6) の「スレッド実行の効率化」に基づき、スレッドコンテキストの先読みを実現する。これらの詳細については後述する。

2.2 細粒度マルチスレッド

細粒度マルチスレッド方式は細粒度スレッドレベルの動的スケジューリングにより、静的には半順序関係しか決定できない処理の実行が可能、細粒度スレッドレベルでの並列性が活用できる、といった特徴を持つ。また、予測不可能な長さの遅延を生じる可能性のある処理において、処理要求と処理結果の受け取りを分離して扱うスプリットフェーズ方式と、細粒度マルチスレッド実行方式を組み合わせることで通信と処理とをオーバーラップさせ、遅延を隠蔽することができる。

細粒度マルチスレッド方式を用いると、スレッドの個数は Pthreads[5] といった今日よく使われているスレッドと比べ多くなる。そこで、FUCE プロセッサは、多数のスレッドを効率よく処理するため、多数の細粒度スレッドを並列に実行する。そのため、スレッド実行ユニットを多重化し、スレッドをそれぞれ一つのスレッド実行ユニットに割り付け並列に実行する。また、細粒度実行方式を利用することでスレッドの粒度が小さくなる。これにより、スレッド切り替えが多くなり、命令実行の効率が低下する。そこで FUCE プロセッサではスレッド先読み機構を用い、スレッド切り替えの高速化を実現する。

2.3 FUCE スレッドとプロセス

FUCE プロセッサにおける、プロセスとスレッドの関係を図 2 に示す。プロセスは一つ以上のスレッドで構成される。ここで、プロセスはファイルといった資源操作の制御単位であり、スレッドはプロセッサ割り当ての単位である。同一プロセスに属するスレッドは、プロセスの環境を共有する。

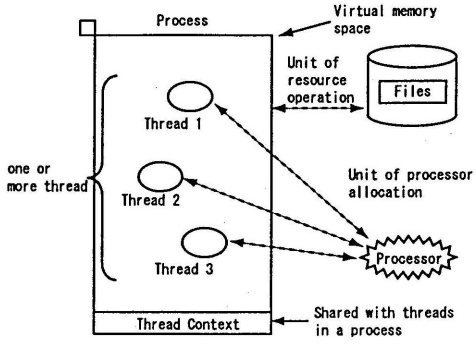

図 2: プロセスとスレッドの関係

2.3.1 FUCE スレッド

FUCE プロセッサにおけるスレッドの定義を以下に述べる。

- 実行終了命令を実行するまで中断することなく排他的に走りきる
- 他のスレッドとの同期は生成時に解決されており、実行の際に他のスレッドと同期を取る必要はない

FUCE スレッドは、細粒度マルチスレッド方式に基づいた細粒度スレッドである。

FUCE スレッドは、細粒度スレッドレベルの動的スケジューリングにより、以下の特徴を持つ。

- 静的には半順序関係しか決定できない処理の実行が可能
- 細粒度スレッドレベルでの並列性が活用できる

また予測不可能な長さの遅延を生じる可能性のある処理において、処理要求と処理結果の受取りを分離するスプリットフェーズ方式と、細粒度マルチスレッド方式を組み合わせることで通信と処理をオーバーラップさせ、遅延を隠蔽する。

2.3.2 スレッド実行の流れ

FUCE プロセッサにおけるある一つのスレッド実行ユニットでのスレッド実行の流れを図 3 に示す。図 3 では、スレッドを実行している部分 (Thread Execution) と次のスレッドコンテキスト (スレッド開始時にレジスタへ用意しておくデータ) 先読みを行なっている部分 (Pre-loading the next Thread Context) を示している。スレッドコンテキスト先読みの詳細については、後述する。

まず 1 番目のスレッドの実行 (Thread 1) が開始される。この時、2 番目のスレッド (Thread 2) のスレッドコンテキストについて先読みを並列に行なっていることが分かる。そして、1 番目のスレッド実行が終了すると、2 番目のスレッドの実行が開始される。3 番目以降のスレッド (Thread 3、4、5) は 2 番目のスレッドと同様に、1 つ前のスレッドが実行されている間に、次スレッドのスレッドコンテキストについて先読みを並列におこなう。これにより、 前スレッドの終了後、直ちにスレッドの切り替えをおこない、次スレッドの実行を開始する。

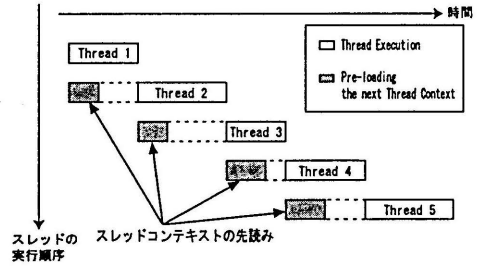

図 3: スレッド実行の流れ

2.4 FUCE プロセッサの実行モデル

FUCE プロセッサは、すべての処理をスレッドとして扱う。そのため、FUCE プロセッサはスレッドの管理を効率よくおこなうために、スレッド管理機構を持っている。スレッド管理機構は、多数のスレッドの管理情報 (同期情報) を保持している。FUCE プロセッサは、スレッド管理機構に対してスレッド情報の登録や同期命令を発行をする。命令により、スレッド管理機構はスレッドの登録処理や同期処理をおこない、実行可能となったスレッドの開始処理をおこなう。スレッド管理機構の詳細については後述する。

FUCE プロセッサの実行モデルを示すために、以下の再帰関数におけるスレッド実行モデルを図 4 に示し、FUCE アセンブラコードを図 5 に示す。

$$f(n) = \begin{cases} f(n-1)+n & n \geq 1 \\ 0 & n = 0 \end{cases}$$

図 4 で、左肩に四角がついた一番外の四角がプロセス f(n) を表し、プロセス内にて四角で表された各ノー

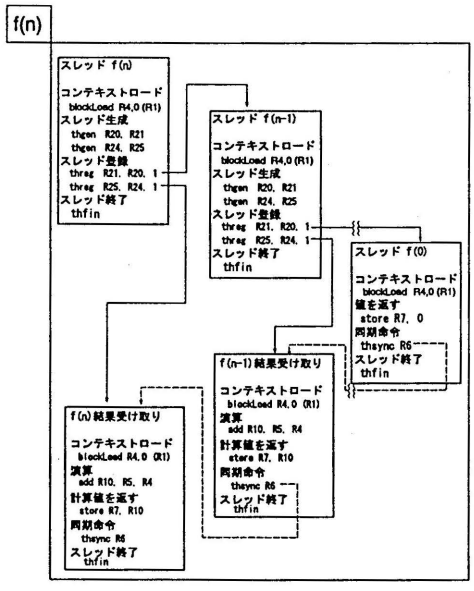

図 4: スレッド実行モデル

```
スレッド f(n)
        blockLoad   R4,0 (R1)       # context load
        bnez        R4, F_N         # if (R4 != 0) jmp F_N, R4 = n
        store       R7, 0           # store 0 to R7 (return result)
        thsync      R6              # thread sync, R6 is thread ID
        thfin
F_N:
        thgen       R20, R21        # gen thd context, f(n-1)
        thgen       R24, R25        # gen thd context, 結果受け取りスレッド
        subi        R12, R4, 1      # R12 = R4 -1, R4 = n
        move        R14, R25
        move        R15, 4(R24)
        move        R16, R4
        move        R18, R6
        move        R19, R7
        blockStore  R12, 0(R20)     # set thread context area f(n-1)
        blockStore  R16, 0(R24)     # set thread context area 結果受け取りスレッド
        threg       R21, R20, 1     # f(n-1)
        threg       R25, R24, 1     # 結果受け取りスレッド
        thfin

f(n) 結果受け取りスレッド
        blockLoad   R4,0 (R1)       # context load
        add         R10, R5, R4     # R10 = f(n-1) + n, R5 = f(n-1), R4 = n
        store       R7, R10         # store R10 to R7 (return result)
        thsync      R6              # thread sync, R6 is thread ID
        thfin
```

図 5: サンプル FUCE アセンブラコード

ドがスレッドを表している。スレッド間の実線の矢印がスレッドの生成を表し、点線の矢印が同期関係を表している。

スレッド f(n-1) は、スレッド f(n) によって生成される。ここで、スレッドの生成とは、スレッドコンテキストの確保 (スレッド実行に利用するレジスタ値の用意) とスレッド管理機構への登録である。スレッド f(n) は、スレッド f(n-1) とスレッド f(n) の結果受け取りスレッドを生成する。結果受け取りスレッドが別途生成される理由は、結果受け取りまでに時間がかかるため、別スレッドとして扱うことで他のスレッドを実行できるようにするためである。

スレッド f(0) と結果受け取りスレッドは、それぞれの終了時に同期命令を発行する。これにより、それぞれのスレッドが返す値を受け取るスレッドを開始状態とする。同期命令により、結果を受け取るスレッドの処理が開始される。演算の結果、再び値を返す必要がある場合は、スレッドの同期命令を発行して、スレッドを終了する。

このように、FUCE プロセッサでは、再帰関数においては演算要求と結果受け取りのスレッド分割をおこなう。

3 FUCE プロセッサの内部構造

3.1 スレッド実行ユニット

FUCE プロセッサは、複数スレッドを同時に効率よく実行するため、スレッド実行ユニットを複数持つ。

スレッド実行ユニットは、一つの細粒度スレッドを実行開始から実行終了まで中断無しに実行する。スレッド実行ユニットは、プログラムカウンタを利用しスレッドの実行を制御する。また、2 命令同時発行、Block Load/Store 命令、Non Blocking Load 命令、交替レジスタファイル、スレッドエントリバッファ などの特徴を持つ。

スレッド実行ユニットの特徴を以下に示す。

- 2 命令同時発行
  命令の発行を 2 命令ずつおこなう。ただし、同時発行する命令間に依存関係はない。
- Block Load/Store 命令
  レジスタを複数本まとめブロック単位で転送をおこなう。
- Non Blocking Load 命令
  データをメモリからレジスタへと読み込む際に、パイプラインを止めずにデータの読み込みをおこなう。また、データの読み込みをおこなうときの

レジスタ競合の確認はスコアボードを用いておこなう。後続命令がレジスタを利用する場合に、はじめてパイプラインを停止する。

- 交替レジスタファイル
  スレッド実行ユニットは、レジスタファイルを2個ずつ持っている。一方は、Current Register File (以下、CRF) と呼ばれ、スレッド実行に用いられる。他方は、Alternate Register File (以下、ARF) と呼ばれる。後述するスレッドコンテキスト先読みに用いられ、それぞれの役目は、スレッド切り替え時に交替する。

- スレッドエントリバッファ
  スレッドエントリバッファは、ハードウェアで管理するスレッドの情報を保持するバッファである。

FUCE プロセッサは、Non Blocking Load 命令 を用いることで、ロード命令発行時にパイプラインを停止せずに命令実行を続ける。そのため、ロード命令を先行して発行することで、データハザードによるパイプラインストールを避けることができる [6]。

3.1.1 スレッドコンテキスト先読み

細粒度マルチスレッド実行ではコンテキスト切り替えが頻繁に発生し、それに伴うオーバーヘッドが重大な問題となる。そこで、FUCE プロセッサでは、コンテキスト切り替えに伴うオーバーヘッドを削減するために、スレッドコンテキストの先読みをおこなう。スレッドコンテキスト先読みをおこなうために、FUCE プロセッサは交替レジスタファイル (CRF と ARF) とスレッドエントリバッファ(Thread Entry Buffer) を持つ。スレッドコンテキストの先読みをおこなうことで、FUCE プロセッサは高速なスレッドコンテキスト切り替えを実現する。

図 6 に、スレッドコンテキスト先読みの概要を示す。

FUCE プロセッサは、プリロードユニット (Pre-load Unit) を利用して次スレッドのスレッドコンテキストを先読みする。プリロードユニットは、スレッドコンテキスト先読みをおこなうためのユニットで、スレッド実行ユニット (Thread Execution Unit) に接続されている。

単一のスレッド実行ユニットによるスレッドコンテキストの先読みの手順を以下に述べる。

1. 実行可能なスレッドを空いているスレッド実行ユニットに割り当てる
2. 割り当てられたスレッドは CRF で実行する
3. スレッド実行ユニットが CRF のスレッドを利用して実行開始すると、次のスレッドをこのスレッド実行ユニットに接続されているプリロードユニットに割り当てる
4. スレッドを CRF で実行している間に次のスレッドのスレッドコンテキストを ARF に読み込む
5. スレッドの実行が終了したら CRF と ARF を切り替え、次のスレッドの実行を開始する

上記 (1) はスレッド実行ユニットの初期状態、すなわちスレッド実行ユニットが待ち状態のときに行なう。そして、(2) から (5) は、(1) が行なわれた後に繰り返し実行される。

プリロードユニットは、スレッドコンテキストの先読み処理として、次のスレッド命令の命令コードの先頭部分に置かれている先読み命令コードを実行することで、次のスレッドコンテキストの先読みを行なう。

図 7 に示す C コードの FUCE アセンブラコードを図 8 に示す。図 8 における先頭の命令コード (blockload R4,.., blockload R16) が、スレッドコンテキスト先読みを指示する命令コードである。プリロードユニットは、この命令コードを実行することで、スレッドコンテキストの先読みをおこなう。

```
for (i = 0; i < N; i++) {
    d = d + a[i] * b[i];
}
```

図 7: サンプル C コード

これにより、先行スレッド終了後に直ちに後続スレッドの実行を開始することができ、スレッド切り替えを高速化できる。

3.2 オンチップメモリ

今後の半導体技術の進歩と単一チップ上にメモリを搭載するオンチップメモリ技術によって、大容量メモリをプロセッサと同一チップに搭載することが可能になると予測する。そこで、FUCE プロセッサは、チップ上にメモリを搭載する。

既存のメモリを外部に持つプロセッサは、ピン数がボトルネックとなり、メモリのバス幅を広げることができない。しかし、メモリを混載することによって、プロセッサのピンボトルネックが生じないため、メモ

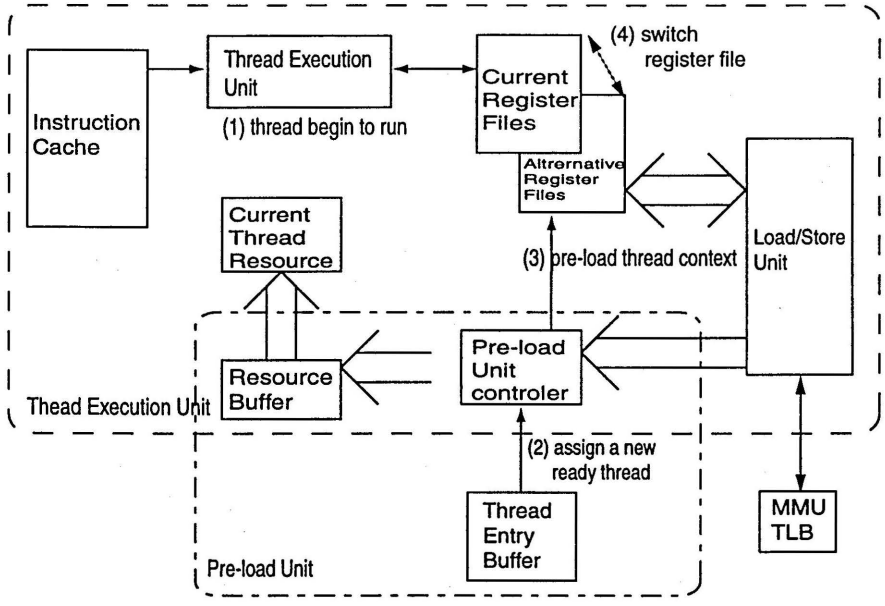

図 6: スレッドコンテキスト先読みの概要

```
        blockLoad  R4, 0(R1)        # pre-load
        blockLoad  R8, 0(R2)          the thread
        blockLoad  R12, 16(R1)        context
        blockLoad  R16, 16(R2)
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
loop:   mul        R20, R4, R8
        mul        R21, R5, R9
        mul        R22, R6, R10
        mul        R23, R7, R11
        add        R24, R12, R13
        add        R25, R14, R15
        blockLoad  R4, 32(R1)        # Non Blocking
        blockLoad  R8, 32(R2)          load
        mul        R20, R12, R16
        mul        R21, R13, R17
        mul        R22, R14, R18
        mul        R23, R15, R19
        add        R26, R21, R22
        add        R27, R23, R24
        add        R28, R24, R25
        blockLoad  R12, 48(R1)       # Non Blocking
        blockLoad  R16, 48(R2)         load
        beq        R1, R29, End
        addi       R1, R1, 32
        addi       R2, R2, 32
        b          loop
End:       .
           .
           .
        blockStore R30, 0(R3)        # store the result
```

図 8: サンプル FUCE アセンブラコード

リのバス幅を大幅に拡張できる。また、オンチップメモリを採用することによって、低レイテンシメモリを実現することができる。つまり、FUCE プロセッサはメモリに関して以下の特徴を持つ。

- 低レイテンシ (4〜6 サイクル程度)
- 高バンド幅 (128 GB/sec)

3.3 通信ユニット

FUCE プロセッサでは、プロセッサ外部とのデータのやりとりをおこなう処理を通信処理と定義する。また通信処理をおこなうユニットを通信ユニットと呼ぶ。通信処理の項目を以下に示す。

- プロセッサ間ネットワーク
  同一計算機内の他のプロセッサとの通信処理
- ネットワークで接続されたメモリコントローラとメモリ
  他の計算機との通信処理
- I/O 処理
  外部記憶装置に関する処理

- 外部ネットワーク
  他の計算機との通信処理

通信ユニットを同一チップ上に搭載することで、命令実行と通信制御を同様に扱えるようにし、通信制御と命令実行の間の処理時間の差を縮めることを目指す。通信ユニットは、チップ内部で内部バスを介してオンチップメモリに結合する。そのため、通信制御を行う際は、命令実行ユニットを介さずに、独自に通信制御を行うことができる。これにより、通信ユニットは通信制御を行う命令実行ユニットの一種として考える。

3.4 高バンド幅内部バス

FUCE プロセッサは、スレッド実行ユニット、通信ユニットとオンチップメモリを、1024bit 幅の内部バスで接続する。内部バスは多段スイッチと多数のバンクによって構成され、以下のような特徴を持つ。

1. 高バンド幅 (128 G Byte/sec)
2. アクセス要求に遅延なく対応
3. 複数のユニットからの同時アクセスに対応
4. 連続したアドレスへの読み/書きに適した構造

上記項目 (2) の「アクセス要求に遅延なく対応」により、内部バスで遅延が生じることなくメモリへアクセスすることができる。また、上記項目 (3) の「複数のユニットからの同時アクセスに対応」により、複数スレッド実行ユニットからのロード/ストア命令に対応できる。ただし、同じバンクに対して要求があった場合は、一方の要求が終わるまで他方の要求が待つ。上記項目 (4) の「連続したアドレス読み/書きに適した構造」により、内部バスは DRAM の特性を生かす構造を持ち、ブロックロード/ストア実行時に内部バスを効果的に利用することができる。

4 スレッド管理機構

これまでの一般のプロセッサでは、スレッドの実行制御はオペレーティングシステム (ソフトウェア) がおこなってきた。この場合、スレッド管理の自由度は大きいが、スレッド管理コストが高いという問題があった。そこで、我々はスレッド管理の一部をハードウェアでおこなうことにした。これにより、多数のスレッドを同時に実行したときのスレッド実行制御を効果的におこなうことができ、細粒度マルチスレッド実行方式を用いる場合に問題となる、スレッド管理コストが解決できる。

また、スレッド管理機構を利用することで、プロセッサ外部からの通信割り込み処理などの処理結果を受け取る任意のスレッドをハードウェアで起動することが可能となる。これにより、プロセッサ外部からの処理結果を受け取る場合に、現在走行しているスレッドを一度中断させ、結果を受け取るスレッドを走らせた後に、中断したスレッドを再開するという必要が無くなる。このことにより、スレッド切り替えの回数が減り、効率の良いスレッド実行を実現する。

4.1 スレッド管理機構の概要

FUCE プロセッサは、スレッド管理機構を用いてスレッドスケジューリングをおこなう。スレッド管理機構について、図 9 に概要を示す。FUCE プロセッサでは、スレッド管理機構は同期カウンタ表にて構成される。

同期カウンタ表は、FUCE プロセッサ内部に構成されたスレッドの情報を保持するテーブルである。FUCE プロセッサでは、スレッド実行ユニットでスレッドの登録命令、スレッドの同期命令を発行することで、同期カウンタ表へスレッドの登録やスレッドの同期をおこなう。スレッド実行ユニットは、スレッドの情報を受け取り、その情報を基にしてスレッドの実行をおこなう。スレッドエントリバッファは、多数の同期が完了したスレッドを保持している。スレッドエントリバッファが持つスレッドは、スレッドの実行が終了し、新しいスレッド実行開始されるときにプリロードユニットが空いたスレッド実行ユニットに移される。

同期カウンタ表の制御は、スレッド実行ユニットにおける命令によっておこなわれる。スレッド登録処理およびスレッド同期処理実行時に、同期カウンタ表を操作する。

それぞれの制御について、以下で詳細を示す。

- スレッド登録処理命令

  命令ニーモニック *threg*

  引数 同期カウンタ表に登録する情報

  動作 引数として渡されたスレッドの情報を同期カウンタ表へ登録する。対象スレッドが同期済み状態として登録された場合は、ただちにスレッドエントリをスレッドエントリバッファへと移す。

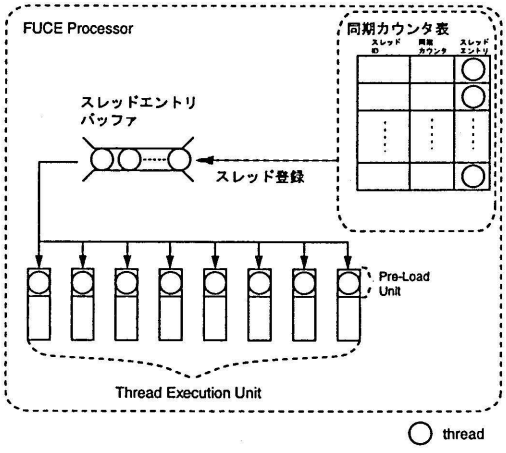

図 9: スレッド管理機構の概要

- スレッド同期処理命令

  命令ニーモニック *thsync*

  引数 スレッド ID (同期をとる対象のスレッド ID)

  動作 引数として渡されたスレッド ID をもつスレッドに対して同期処理をおこなう。もし、対象スレッドの同期が終了した場合は、スレッドエントリをスレッドエントリバッファへと移す。

4.1.1 同期カウンタ表

同期カウンタ表はスレッドの情報 (スレッド ID、同期カウンタ、スレッドエントリ) にて構成されている。

同期カウンタ表を構成するスレッドの情報について詳細を以下に示す。

- スレッド ID
  一意に決まっているシリアル値であり、この ID を基にして同期をおこなうスレッドを決定する

- 同期カウンタ
  スレッドの同期情報を持つ

- スレッドエントリ
  スレッドを特定するための情報を持つ。スレッド実行ユニットはスレッドエントリを基にして、スレッドの実行をおこなう。スレッドエントリは以下の項目を持つ。

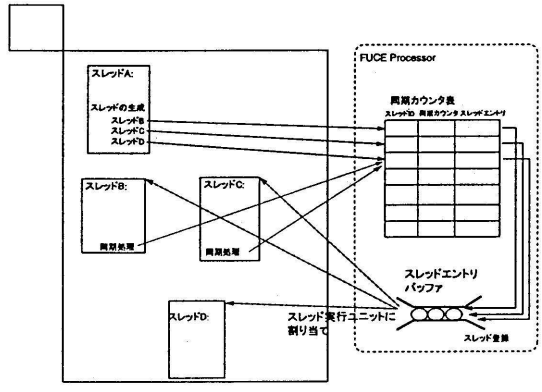

図 10: 同期処理の流れ

  - プロセス ID
  - スレッドコンテキストを保持している空間の先頭アドレス
  - スレッドの命令コードの先頭アドレス

同期が終了したスレッドは、スレッドエントリバッファへと移され、スレッド実行ユニットに割り当てられ実行される。

4.2 スレッド同期処理の流れ

図 10 に、スレッド同期処理の流れを示す。FUCE プロセッサではスレッドの制御はスレッド管理機構 (同期カウンタ表) を用いておこなわれる。

図 10 において、スレッド A がスレッド B、スレッド C、スレッド D を生成している。スレッド B とスレッド C については、同期済み状態として登録されることで、直ちにスレッドエントリバッファへと移される。スレッド D については、スレッド B とスレッド C の同期待ちであるので、同期カウンタ表に登録され、同期待ち状態となる。

スレッド B とスレッド C の同期処理の命令発行により、スレッド D の同期が終了すると、スレッド D がスレッドエントリバッファへと移され、スレッド実行ユニットに割り当てられるのを待つ。

4.3 割り込み処理の扱い

FUCE プロセッサは、すべての処理をスレッドとして扱う。そのため、割り込み処理についてもスレッドとして扱う。

FUCEプロセッサでは、割り込み処理をおこなうスレッドが、初期値として同期カウンタ表へと登録されていて、すべての割り込み処理は固有のスレッドIDを保持している。割り込み処理が発生した場合は、その割り込み処理が持つスレッドIDを引数としてスレッド同期処理命令を発行する。これにより、割り込み処理をおこなうスレッドが、同期カウンタ表からスレッドエントリバッファへと移され、スレッド実行ユニットにスレッドが割り当てられることで、割り込み処理が開始される。

5 おわりに

本稿では、FUCEプロセッサの概要とスレッド管理機構について述べた。

FUCEプロセッサは、すべてのプロセッサ内部の処理をスレッドとして扱う。そのため、FUCEプロセッサはスレッドの管理を効率よくおこなうために、スレッド管理機構を持っている。FUCEプロセッサは、スレッド管理機構を利用することで、ハードウェアでスレッドの登録処理やスレッドの同期処理のようなスレッドの実行制御をおこなうことができる。

ハードウェアにてスレッド実行制御をおこなうことによって、多数のスレッドを同時に実行する際に問題となる、スレッドの管理コストを削減することができる。また、割り込み処理が発生した場合も、現在実行中のスレッドを中断することなしに、割り込み処理をおこなうスレッドを用意することができる。これにより、スレッドを中断することによる、実行効率の低下を防ぐことが可能となり、多数のスレッドを実行したときのスレッド実行を効果的におこなうことができる。

現在、プロセッサ構造の詳細を検討すると共に、FUCEプロセッサのシミュレーション環境の開発をおこなっている。今後、シミュレーションにより、多数のスレッドを実行する条件下での、本アーキテクチャの性能評価をおこない、FUCEプロセッサ実行モデルの有効性を確認する予定である。

本研究は、通信・放送機構の創造的情報通信技術研究開発推進制度に係わる研究開発課題「次世代型インテリジェント・マルチメディア情報通信網の基盤技術に関する研究」による。

参考文献

[1] 松崎 隆哲, 富安 洋史, 大庭 直行, 雨宮 真人, 「通信と処理の融合を行なうFUCEプロセッサの提案」, 信学技報, CPSY2000-52, Vol.100, No.249, pp.1-7(2000)

[2] Makoto Amamiya, Hideo Taniguchi, Takanori Matsuzaki, " An Architecture of Fusing Communication and Execution for Global Distributed Processing," Parallel Processing Letters, Vol.11, No.1, pp.7-24, 2001

[3] 松崎 隆哲, 富安 洋史, 大庭 直行, 雨宮 真人, 「高バンド幅内部バス構造のオンチップメモリを持つFUCEプロセッサ」, 情処研報, 2000-HPC-83, Vol.2000, No.93, pp.7-12 (2000).

[4] Takanori Matsuzaki, Hiroshi Tomiyasu, Makoto Amamiya,. "Basic Mechanisms of ThreadControl for On-Chip-Memory Multi-threading Processor," In Proceedings of the Fifth Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC-5), pages 4350, 12 2001.

[5] B. Nichols, D. Buttlar, and J. P. Farrell,"Pthreads Programming," O'Reilly, 1996, 邦訳 榊 正憲:Pthreads プログラミング, オライリー・ジャパン (1998).

[6] 松崎隆哲, 富安洋史, 大庭直行, 雨宮真人, 「マルチスレッドプロセッサにおけるメモリアクセスレイテンシ隠蔽の一手法」, 情処研報, HOKKE2001,Vol.2001,No.22,pp67-72(2001)