修 士 論 文

マルチスレッド型プロセッサの
動画像処理への適用

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

宮武 克幸

2007年3月

修 士 論 文

マルチスレッド型プロセッサの動画像処理への適用

指導教官　日比野靖 教授

審査委員主査　日比野靖 教授
審査委員　田中清史 助教授
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

510100 宮武 克幸

提出年月: 2007 年 2 月

概 要

リアルタイムの動画像処理では高速にエンコードを行う必要がある。しかし、エンコード処理の大部分を占める動き補償処理には膨大な計算量を必要とする。そのため、本研究では、動き補償処理の計算量を削減し高速に処理を行うため、動き補償処理に特化した命令セットと処理の命令列を設計する。また、動画像データはデータに依存関係がなく、マルチスレッド化が容易であるので、マルチスレッド処理による処理速度の向上を図る。設計したアーキテクチャで動画像を処理したときの処理速度の評価をする。

目 次

第1章　序論

1.1　研究の背景

近年、計算機やネットワーク技術の発展により、大容量のマルチメディアデータが扱われるようになった。例えば、リアルタイムの動画像処理が必要なテレビ電話では、動画像の高圧縮が必要であるため、高度な画像の符号化（エンコード）を高速に行う必要がある。また、HDTV(High Difinition Television) では従来の方式よりも高精細な動画を扱うため、高画質のエンコードが求められる。

動画像のエンコード処理は、フレーム内符号化とフレーム間符号化に分けられるが、フレーム間符号化により高度な圧縮を行う MPEG 系のエンコーダでは、動画像の各フレーム間の動き補償処理に膨大な処理時間を必要とする。リアルタイムの動画像処理ではエンコードのため許される遅延が数フレーム時間に限定される。したがって、プロセッサでは大量の処理を高速に行う必要がある。

1.2　研究の目的

本研究では、エンコードの中でも動き補償処理に重点を置き、動き補償処理をパイプライン処理する専用プロセッサを用いることで処理速度の向上を図る。

まず、動き補償処理は計算量が膨大なため、エンコードの圧縮率を保ちつつもその計算量を削減する手法を適用する。

次に、動き補償処理に特化した専用の命令セットアーキテクチャのプロセッサを設計し、効率的に動き補償処理が行えるようにする。そして、この命令セットにより動き補償処理を行う命令列を設計し、パイプラインストールなどの無駄なクロックサイクルを省き、総クロックサイクル数を改善し、処理速度を向上させる。

さらに、設計した命令列に対してマルチスレッド処理を適用する。動画像データはデータに依存関係がないので、処理の各行程を複数の流れに分割し、マルチスレッド処理を行うことで、処理速度のさらなる向上を図る。これらマルチスレッドによる並列処理は動画像の GOP 単位またはマクロブロック単位で処理することで可能となる。マルチスレッド化により、パイプラインストールをさらに減らすことができ、CPI をより改善できる。これにより処理速度はさらに向上する。

1.3 本論文の構成

本論文の構成は以下の通りである。第 2 章で動き補償処理の原理について説明し、一般的な動き補償処理の計算量削減法について述べる。第 3 章では、専用プロセッサの設計を考慮し、さらに計算量を削減する手法を提案する。第 4 章では、設計した命令セットについて述べる。第 5 章では、設計した命令セットを用いた動きベクトル検索処理の命令列について述べる。第 6 章では、5 章で設計した命令コード列をマルチスレッド化し、XVGA 画質の動画像処理を可能にする性能が得られることを示す。第 7 章は本論文のまとめである。

第2章　動き補償処理

動き補償処理はMPEGのエンコード時に行われる処理であり、現在の処理対象となる対象フレーム(Current Frame: C Frame)から前の時刻の参照フレーム(Refference Frame: R Frame)で物体が移動するような動画像があった場合、画面内で物体が移動したことによって発生する領域の動きベクトル（水平移動量，垂直移動量）と移動した原画像との各画素の差分のみで動画像データを表現することによって、エンコードの圧縮率を高める処理方法である。図2.1に動き補償処理の例を示す。動きベクトルを求めるにはフレーム間で動きベクトル検索を行う。そして、動き補償処理では、この動きベクトル検索の処理に膨大な処理時間を要する。

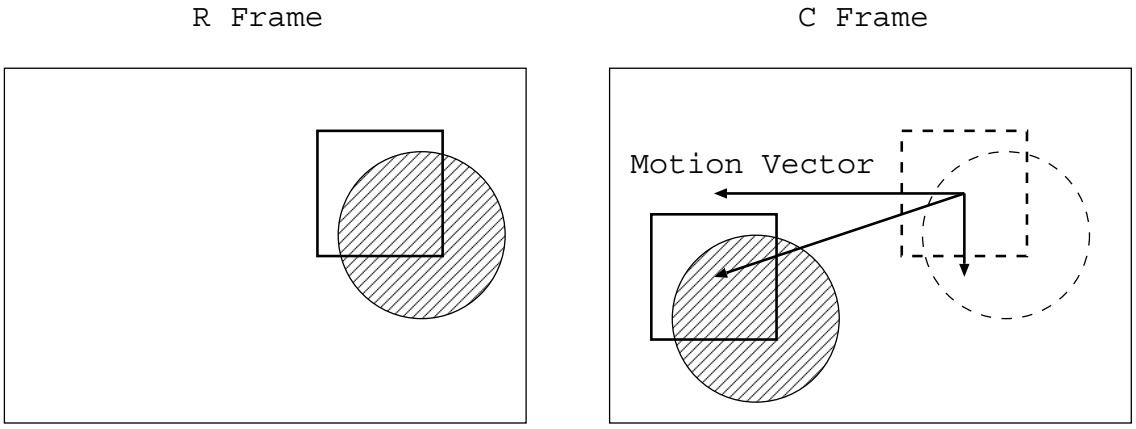


図 2.1: 動き補償処理の例

2.1　動きベクトル検索

図2.2に動きベクトル検索の例を示す。動きベクトルを導き出すには、動きベクトルを求める対象フレームの或るマクロブロック範囲（16 × 16画素）と、その対象フレームと比較する参照フレームのマクロブロック範囲の256画素の輝度値をそれぞれ比較して、差を求める。そしてそれら256個画素分の輝度値の差を全て足して、総和を求める。

この総和を、参照フレームではマクロブロック範囲を1画素づつずらしながら、比較計算を行い全フレーム領域分の総和を各マクロブロックごとに求める。そしてこの総和が最

も少ない参照フレームでのマクロブロック位置を、或る対象フレームのマクロブロックの動きベクトルとする。この処理を対象フレームの全マクロブロックで行い、すべての動きベクトルを導き出す。

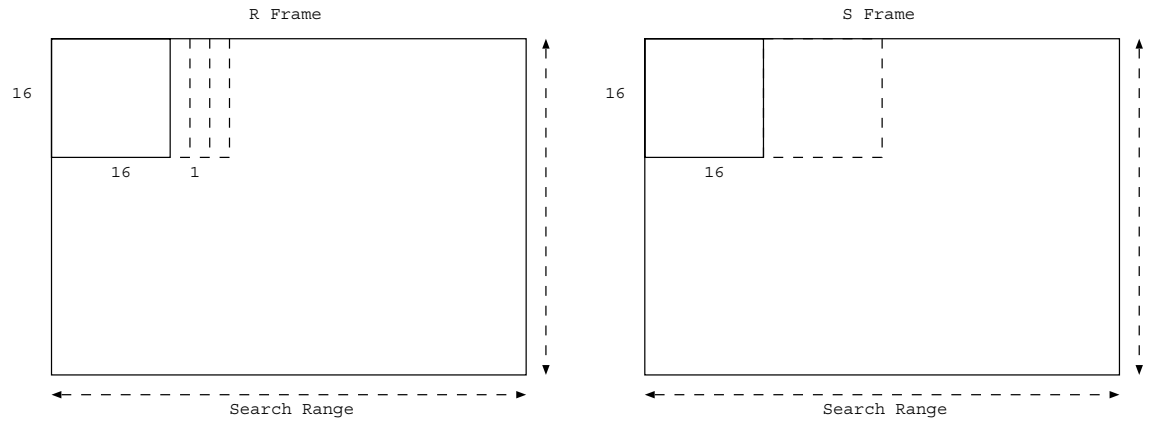


図 2.2: 動きベクトル検索の例

2.2 計算量削減法

動きベクトル検索はエンコード処理の大部分を占めるので、圧縮率を保ちつつも計算量ができるだけ少ない適切な手法を選ぶ必要がある。計算量を削減する一般的な手法として、階層的マッチング法と検索領域限定法がある。

2.2.1 階層的マッチング法

動きベクトル検索の計算量を削減する方法のひとつとして、階層的にブロックマッチングすることが考えられる。通常の動きベクトル検索では参照フレームの全領域を検索するが、この手法では図 2.3 のように、まず 1 段階目で縮小画像を用いて全領域を 1 画素ずつ検索し、荒い画像により大まかな探索を行う。次に 2 段階目では原画像を用いて、1 段階目で得られた輝度値の総和が最も小さい最適な位置を中心として、原画像により 1 画素づつのより詳細な探索を行う手法である。階層的に検索を行うことで、動きベクトル検索の計算量を削減できる。

例えば 4 画素毎に処理を行うとすれば、はじめの荒い画像によるおおまかな検索により、計算量を通常の処理の 16 分の 1 に削減できる。そして次に詳細な検索に移るが、検索領域としては 16×16 の範囲を 1 画素ずつずらしながら詳細に行うのが好ましい。これにより、計算量を見積もると、対象フレームが QVGA サイズの場合、比較回数は $(320 \times 240) \div 16 + 16 \times 16 = 5056$ となり、通常の動きベクトル検索処理のほぼ 16 分の 1 に計算

量を削減出来る。また、8 画素毎など間引いて検索する量を増やすことにより、さらに計算量を削減できる。しかし、そのぶん動きベクトルの精度が落ちるので、圧縮率が低下する。計算量と圧縮率のトレードオフを考える必要がある。

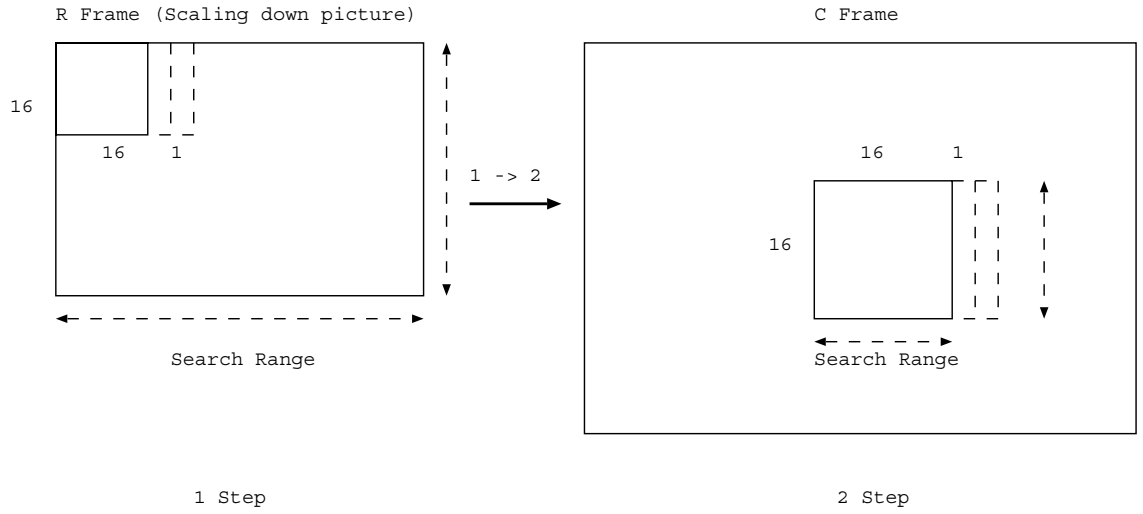


図 2.3: 階層的マッチング法

2.2.2 検索領域限定法

動きベクトル検索の計算量を削減する 2 つめの方法として、検索領域を限定することが考えられる。この手法は参照フレームの検索領域の全フレーム領域を比較するのではなく、普通は急激に物体が動くことはないと仮定して、フレームの検索領域を限定する手法である。

図 2.4 に検索領域限定法を示す。検索領域はフレームレートと物体のフレーム内での移動速度に依存する。例えば 30fps の動画像で、検索領域をフレームの 4 分の 1 の大きさに限定するということは、1 フレーム時間 (33.3msec) に画面内の半分の距離を移動する物体を追随できることに相等する。また計算量としては 4 分の 1 となる。

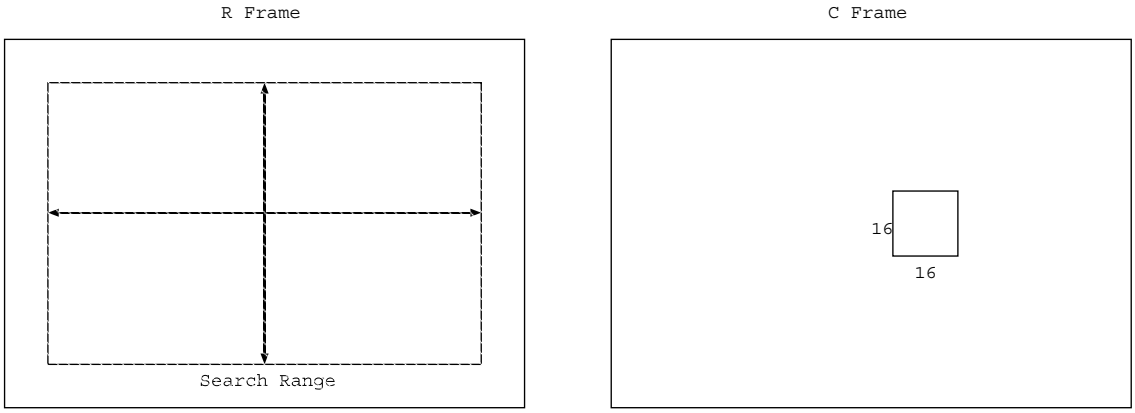


図 2.4: 検索領域限定法

第3章　提案手法

前章では、動きベクトル検索の原理と、一般的な計算量削減法について述べた。本章では、設計する専用プロセッサのアーキテクチャを考慮し、さらに計算量を削減し処理速度を向上させる方法について考える。

3.1　ロード回数削減法

動きベクトル検索では、対象フレームのある固定したマクロブロックに対して、参照フレームでは随時マクロブロック範囲をずらしながら輝度値の比較を行う。これより、対象フレームでは一定の期間は常に同じマクロブロックに対して処理を行うが、参照フレームでは毎回異なるマクロブロックに対して処理を行う。そのため、対象フレームでは画素のロード回数が参照フレームより少なくてすむ。メモリアクセス命令であるロードワード命令は、他の演算命令に比べメモリアクセスのために実行に長時間を要するので、ロード回数を少なくすることにより処理速度を向上させることが出来ると考えられる。

まず、動きベクトル検索においてフレーム間でマクロブロックの画素の差の総和を求める簡単なアセンブラレベルの命令列を表3.1に示す。この処理では、はじめに対象フレームと参照フレームのロードを行う。次に、それらの画素の差を求め、そして、この差を足し合わせる。次に、総和を求める。これをマクロブロック分の処理が行うまで繰り返す。

次に、表3.1に示した処理列にロード回数削減法を取り入れる。これを表3.2に示す。対象フレームのロード1回に対して、参照フレームでは異なるマクロブロックに対して画素の座標を変えながら、ロードを数回繰り返す。このロードの繰り返し回数により、ロー

表 3.1: 簡単な動きベクトル検索処理のアセンブラコード

```
Loop:  Load   R1   R5     C          :C Frame Load
       Load   R2   R5     R          :R Frame Load
       Diff   R3   R1     R2         :R3 ← [R1-R2]
       Add    R4   R4     R3         :R4 ← R4+R3
       Addi   R5   R5     1          :R5 ← R5+1
       Bne    R6   R5     Loop       :if(R6 ≠ R5) goto Loop
```

表 3.2: ロード回数削減法を取り入れたアセンブラコード（4 回ロード）

Loop:	Load	R1	R10	C	:C Frame Load
	Load	R2	R10	R	:R Frame Load
	Load	R3	R10	R+1	:R Frame Load
	Load	R4	R10	R+2	:R Frame Load
	Load	R5	R10	R+3	:R Frame Load
	Diff	R2	R1	R2	:R2 ← [R1-R2]
	Diff	R3	R1	R3	:R3 ← [R1-R3]
	Diff	R4	R1	R4	:R4 ← [R1-R4]
	Diff	R5	R1	R5	:R5 ← [R1-R5]
	Add	R6	R6	R2	:R6 ← R6+R2
	Add	R7	R7	R3	:R7 ← R7+R3
	Add	R8	R8	R4	:R8 ← R8+R4
	Add	R9	R9	R5	:R9 ← R9+R5
	Addi	R10	10	1	:R10 ← R10+1
	Bne	R11	R10	Loop	:if(R11 ≠ R10) goto Loop

ド命令の次の絶対値を求める命令が始まるころには、最初の対象フレームと 1 回目の参照フレームのロードを終えている。これにより、メモリアクセスレイテンシを多い隠すことができ、ストールなく動作させることができる。

またこれを実行するプロセッサを考慮すると、参照フレームのロード命令、絶対値計算、総和を求める計算では、それぞれレジスタ競合が起きないようにデスティネーションレジスタとソースレジスタは、それぞれの命令で異なるレジスタ番号に設定する必要がある。

このロード回数削減法を適用し、同じループでの参照フレームのロード回数を増やすことにより、ステップ数が減り処理速度が向上する。

しかし、このロード命令の繰り返し回数を増やすことにより、必要レジスタ数が増える。レジスタが多いアーキテクチャでは、命令セットでのオペランド指定のためのビット数が増え、命令の表現方法が減少する。また、ビット数が増えレジスタフェッチやレジスタデコードの際の処理量が増加し、その分処理時間も長くなる。そこで、ロード回数削減法により命令の総ステップ数を減らすことが出来るメリットと、レジスタ数が増えることによりプロセッサの処理速度が落ちるというデメリットについてのトレードオフについて考える必要がある。メリットを生かしつつ、デメリットを抑えるため、以下に最適なロード回数の繰り返し回数を決定をするための見積もりを行う。

3.2　メモリアクセスレイテンシの見積もり

まず、フレームのロード命令に要するメモリアクセス時間を見積もる。動きベクトル検索の処理対象となる動画像フレームはメモリに格納した状態で、そこからロードすることで各データの処理を行うわけだが、そのフレームデータをどのようなメモリに格納するかで、メモリアクセス時間は大きく異なる。

SRAM だとアクセス時間は速いが、容量は小さい。メモリアクセス時間は数ナノ秒程度である。同一面積での容量は DRAM に比べて 5 分の 1 程度である。そのため、DRAM で 256Mbyte のメモリの面積では、SRAM の容量は 50Mbyte 程度となる。

DRAM だとアクセス時間は遅い。だが容量は大きく最近主流の DRAM である DDR SDRAM でメモリアクセス時間を見積もったところ、性能や設定によりアクセス時間は異なるため、短くて 30ns から長くて 110ns であった。

見積もりの結果、SRAM にフレームデータを格納することにする。まず、アクセス時間も数ナノ秒で高速である。次に、動画像フレームを格納する容量としては、320 × 240 画素の QVGA 画質のフレームだと 1 フレームで 1Mbyte 程度であり、対象フレームと参照フレームの両方でも 2Mbyte である。また 1024 × 768 画素の XVGA 画質の場合だと、QVGA の 10 倍程度の容量が必要であるが、それでも対象フレームと参照フレームの両方で 20Mbyte である。これより、SRAM は容量的にも十分である。

3.3　具体的処理の見積もり

見積もったメモリアクセス時間より、具体的な命令処理コードにおいて、メモリアクセス時間がどのように影響するかについて考える。例えば、プロセッサのクロックサイクル時間を 1ns とする。ロード命令のステージ構成を IF,ID,EX,MEM,WB の 5 ステージ構成とし、MEM ステージ以外は 1 クロックかかるとする。SRAM のアクセス時間が長くても 10ns 程度であることから、MEM ステージには 12 クロックかかるものとする。上記のことからロード命令には 16 クロックかかる。また、ロード命令以外の命令は 4 クロックとする。

ここで新たに参照フレームのロードを 16 回繰り返した場合の命令列を表 3.3 に示す。ロード回数削減法で対象フレーム 1 回のロードに対して、参照フレームでは 1 ループに 16 回のロードを行うこととする。16 回にすることにより、プロセッサのクロック周波数が 1GHz とすると、1 クロックが 1ns なので、一番最初の対象フレームのロードがあり、次に参照フレームのロードが 16 回あり、この 16 回の命令が発行し終えると同時に、次の命令にパイプラインストールすることなしに、対象フレームと 1 回目の参照フレームの画素の絶対値計算である Diff 命令に移ることができる。絶対値計算の後は各異なる 16 個のマクロブロックの総和を求める処理を行う。

また、ロード回数削減法の 16 回という回数の別な利点としては、例えば 16 回だとマクロブロックの水平の 1 ラインの画素を一度に処理できることが挙げられる。このため丁度

表 3.3: ロード回数削減法を取り入れたアセンブラコード（16 回ロード）

```
Loop:  Load   R1   R10    C          :C Frame Load
       Load   R2   R10    R          :R Frame Load
       ...
       Load   R17  R10    R+16       :R Frame Load
       Diff   R2   R1     R2         :R2 ← [R1-R2]
       ...
       Diff   R17  R1     R17        :R17 ← [R1-R17]
       Add    R18  R18    R2         :R18 ← R18+R2
       ...
       Add    R33  R33    R17        :R33 ← R33+R17
       Addi   R34  R34    1          :R34 ← R34+1
       Bne    R35  R34    Loop       :if(R35 ≠ R34) goto Loop
```

よい回数であると言える。

3.4 レジスタ数の見積もり

メモリアクセス時間の決定、ロード回数削減法での参照フレームのロード回数の決定により、レジスタ数の見積もりを行うことができる。表 3.3 の 1 回のループでストールが起きないようにレジスタを用意し、レジスタ競合を避ける。

表 3.3 より、この命令列ではレジスタを 35 個用いていることがわかる。内訳は対象フレームのロードにレジスタ 1 を割り当て、参照フレームのロード 16 回にレジスタ 2 から 17 を割り当る。それぞれのマクロブロックの総和を求めるのにレジスタ 18 から 33 を割り当てる。また総和を求める命令はマクロブロック分処理する間ずっと加算していくため、一時的なレジスタを用いるというわけにはいかず、別にレジスタを用意する必要がある。表 3.4 に必要レジスタ数を示す。インクリメント値に用いているレジスタ 34 や、分岐の値に用いているレジスタ 35 を除いても、レジスタの総数は 33 以上必要だということが分かる。

表 3.4: レジスタ数の見積もり

処理命令	レジスタ数
対象フレームロード	1
参照フレームロード、絶対値計算	16
総和計算	16
合計	33

第4章　命令セットの設計

本章では、動きベクトル検索に特化した命令セットの設計を行う。

4.1　命令形式

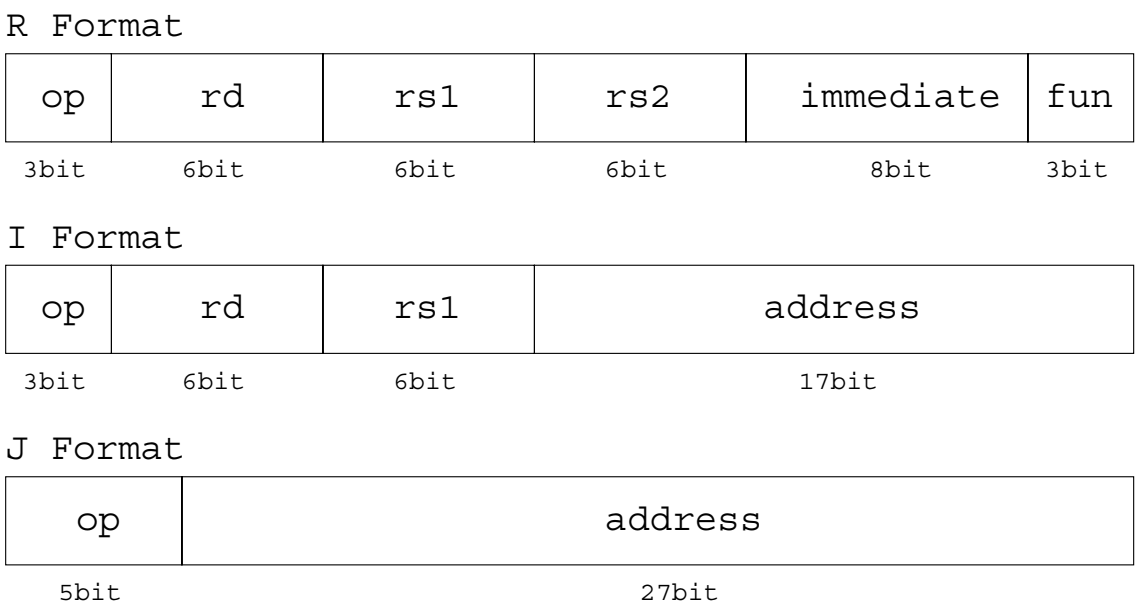


図 4.1: 命令形式

命令セットの命令形式を図4.1に示す。命令セット長はレジスタ長ともに32bit長とする。

まず、それぞれの形式について説明する。R形式は演算命令の形式であり、I形式はメモリアクセス命令と条件分岐命令である。J形式は無条件分岐命令である。

次に各フィールドについて説明をする。opはどのような処理の命令を行うかの命令操作である。rdは結果を納める先のデスティネーションオペランドレジスタである。rs1は第1のソースオペランドレジスタであり、rs2は第2のソースオペランドレジスタである。またこれらレジスタのフィールドは長さが6bitなので最大64個のレジスタを表現できる。immediateはイミディエイト値の設定フィールドである。funはR形式命令のopフィールドの種類を拡張するためのフィールドである。addressはメモリをアクセスするための

表 4.1: R 形式 (論理演算) op = 000, fun = xxx

ニーモニック	オペコード	意味
and	000	rd ← rs1 ∧ rs2
or	001	rd ← rs1 ∨ rs2
xor	010	rd ← rs1 xor rs2
mov	011	rd ← rs1
andi	100	rd ← rs1 ∧ immediate(14bit)
ori	101	rd ← rs1 ∨ immediate(14bit)
sl	110	rd ← rs1 ShiftLeft immediate(8bit)
sr	111	rd ← rs1 ShiftRight immediate(8bit)

アドレス値を設定するフィールドである。

4.2 R形式の命令

4.2.1 論理演算命令

表 4.1 に R 形式の論理演算命令のニーモニックとオペコードを示す。op=000 であり、fun=xxx で以下の命令を区別する。

andi、ori の 2 命令はイミディエイト値の指定には rs2 と immediate フィールドを繋げることによりイミディエイト値が 14bit まで指定できるようにする。イミディエイト値の指定には rs2 と immediate フィールドを繋げることによりイミディエイト値が 14bit まで指定できるようにする。

4.2.2 算術演算命令

表 4.2 に R 形式の算術演算命令のニーモニックとオペコードを示す。op=001 であり、fun=xxx で以下の命令を区別する。

cmp は比較命令であり、rs1 - rs2 を行うが rd には格納しない。フラグ状態だけを更新する命令である。

diff は本研究で特別に作った命令であり、絶対値計算の命令である。動きベクトル検索では画素と画素の差を求める計算が大量に行われるので、1 命令により絶対値を求められる diff 命令は大変都合が良い。

addi、subi、cmpi の各命令は先ほどの andi 命令、ori 命令と同様にイミディエイト値が 14bit まで指定できるようにする。

ldi は Load Immediate の事である。より大きいイミディエイト値をロードするために、

表 4.2: R 形式 (算術演算) op = 001, fun = xxx

ニーモニック	オペコード	意味
add	000	rd ← rs1 + rs2
sub	001	rd ← rs1 - rs2
cmp	010	rs1 - rs2
diff	011	rd ← | rs1 + rs2 |
addi	100	rd ← rs1 + immediate(14bit)
subi	101	rd ← rs1 - immediate(14bit)
cmpi	110	rs1 - immediate(14bit)
ldi	111	rd ← immediate(20bit)

表 4.3: I 形式 (メモリアクセス) op = 01x

ニーモニック	オペコード	意味
lw	0	rd ← memory[rs1 + address]
sw	1	memory[rs1 + address] ← rd

イミディエイト値の指定には rs1 と rs2 と immediate フィールドを繋げることによりイミディエイト値を 20bit まで拡張する。

4.3 I 形式の命令

4.3.1 メモリアクセス命令

表 4.3 に I 形式のメモリアクセス命令のニーモニックとオペコードを示す。op=01x であり、op の 3bit 目で以下の命令を区別する。

lw は Load Word 命令であり、メモリからレジスタへデータ転送を行う。sw は Store Word 命令であり、レジスタからメモリへデータ転送を行う。

メモリのアドレッシングについてはインデックスレジスタ方式を用い、memory[rs1 + address] により計算し、アクセスを行う。

4.3.2 条件分岐命令

表 4.4 に I 形式の条件分岐命令のニーモニックとオペコードを示す。op=10x であり、op の 3bit 目で以下の命令を区別する。

表 4.4: I 形式 (条件分岐) op = 10x

ニーモニック	オペコード	意味
beq	0	if(rd = rs1) goto address
bne	1	if(rd ≠ rs1) goto address

表 4.5: J 形式 (無条件分岐) op = 11xxx

ニーモニック	オペコード	意味
j	000	goto address
jal	001	r63 ← PC + 4; goto address
jr	010	goto r63
je	011	if(Z = 1) goto address
ja	100	if(Z = 0) ∧ (C = 1) goto address
jae	101	if(C = 1) goto address
jb	110	if(C = 0) goto address
jbe	111	if(Z = 1) ∨ (C = 0) goto address

beq は branch equal 命令であり、rd と rs1 が等しければ address へ分岐する。bne は branch not equal 命令であり、rd と rs1 が等しくなければ address へ分岐する。

4.4 J 形式の命令

表 4.5 に J 形式の無条件分岐命令のニーモニックとオペコードを示す。op=11xxx であり、op の 3-5bit 目で以下の命令を区別する。

jal 命令と jr 命令によりサブルーチン構造を可能にする。jal によりレジスタに戻り値を設定しつつサブルーチンコードへ分岐し、jr により戻り値のメインの処理へ分岐が行える。

je、jg、jge、jl、jle の各命令は、比較命令により更新されたフラグの状態によって分岐を行う命令である。状態を確認するフラグについては、Z(Zero Flag) は演算結果が 0 の時にセットされるフラグである。Z=1 のときは演算した rs1 と rs2 の値が等しいことを示す。C(Carry Flag) は演算結果の最上位ビットを示すフラグである。C=0 のときは rs1 のほうが rs2 より大きいことを示す。

第5章　動きベクトル検索処理を行う命令列の設計

本章では、2 章で挙げた計算量削減法と、3 章で提案したロード回数削減法を実現する動きベクトル検索を行う命令列を、4 章で設計した命令セットにより作成する。処理対象となる動画像のフレームサイズは 320 × 240 画素の QVGA とする。

5.1　ロードのインデックス指定

まず、ロード時のインデックス計算について考える。メモリに処理対象となるフレームが格納されているわけだが、フレームは画像であり 2 次元のデータである。そこで、1 次元の配列であるメモリへのデータの格納方法はインデックス計算がしやすいように工夫をする。例えば 320 × 240 画素のフレームをメモリへ格納するには、左上の画素から右下の画素まで水平の 1 ラインごとにメモリへ格納することを考える。水平の 1 ラインは 320 画素のため、2 進数でこのインデックスを指定しやすいように考えると、512 の区画ごとに水平の 1 ラインを格納していくと都合がよい。これより、データの 1 区画が 4byte と考えると、1 画素右へアクセスするには 1 × 4byte、1 画素下へアクセスするには 512 × 4byte である。フレームデータのメモリへの格納法を図 5.1 に示す。

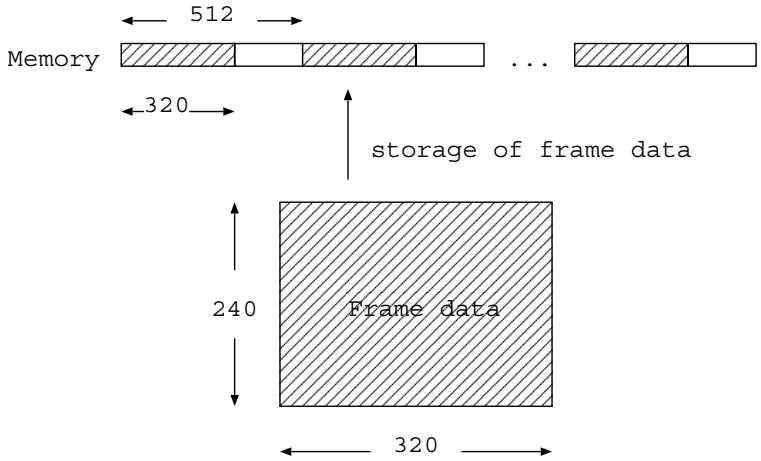


図 5.1: フレームデータのメモリへの格納法

次に、対象フレームの 1 回のロードに対して参照フレームでは 16 回のロードを行うわけだが、この 16 回のロードはそれぞれ異なるマクロブロックのロードである。これにより総和の計算時にはストールなく処理が行える。また 16 回ロードする各マクロブロックの位置としては、縦横 4 画素づつの 16 画素の範囲を一度に処理する。例えば階層検索で 4 画素毎の縮小画像に対してこれを行えば、これはちょうど 1 マクロブロック分の処理に相当する。また、このように縦横の処理量が等しいことにより、制御も行いやすい。

5.2 マクロブロックレベルの処理

5.2.1 水平方向

マクロブロックの水平方向の処理範囲を図 5.2 に示す。対象フレームの画素をロード、参照フレームの画素を 16 回ロードし、それらの絶対値を求め、それらの総和を求める。マクロブロックの水平の画素数は 16 なので、その分をインクリメントし、それを対象フレームと参照フレームのインデックス値に加算し、これを 16 回分行うまで繰り返す。

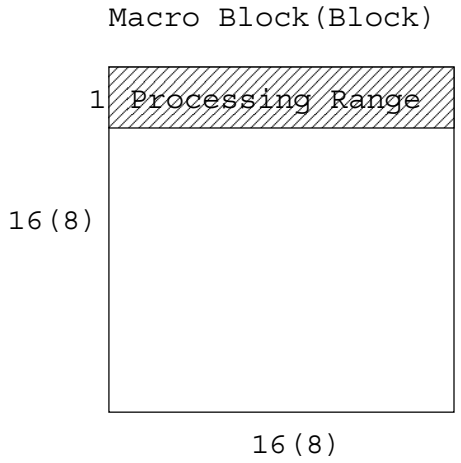


図 5.2: ＭＢ水平ループの処理範囲

5.2.2 垂直方向

マクロブロックの垂直方向の処理範囲を図 5.3 に示す。マクロブロックの水平処理のループを抜けると、次に 1 列下のラインへアクセスするため、垂直方向のループに入る。まず、マクロブロックの水平座標の変数をリセットし、次にインデックス値の水平座標部分のみをリセットする。そして、この処理を 16 回行うまで繰り返す。

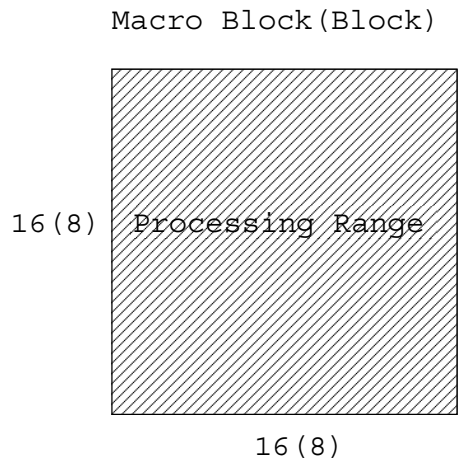


図 5.3: ＭＢ垂直ループの処理範囲

5.3 参照フレームレベルの処理

5.3.1 水平方向

参照フレームの水平方向の処理範囲を図 5.4 に示す。マクロブロックレベルの処理が終えた時点でマクロブロックの輝度値の差の総和が求まることになる。求まった 16 個の総和より、それらを比較して最小総和を求める必要がある。最小総和を求めるには各総和を cmp で比較して、小さい方の総和と、そのマクロブロックの参照フレームでの座標を保存する。16 個の中の最小総和が求まると、次に前回までの最小総和と比較して、常に現時点での最小総和を求める。次に、最小総和より動きベクトルである水平座標と垂直座標を求める。

またこのループの処理回数は、検索領域限定法と階層的マッチング法により決定される。またインクリメント値は、4 画素毎の階層的マッチング法の時一度に 1 マクロブロック分の処理を行うので、次の処理へ移るには 16 画素隣へ移動することになる。

5.3.2 垂直方向

参照フレームの垂直方向の処理範囲を図 5.5 に示す。このループの処理回数も、検索領域限定法と階層的マッチング法により決定し、インクリメント値も 4 画素毎処理時は垂直に 16 画素である。

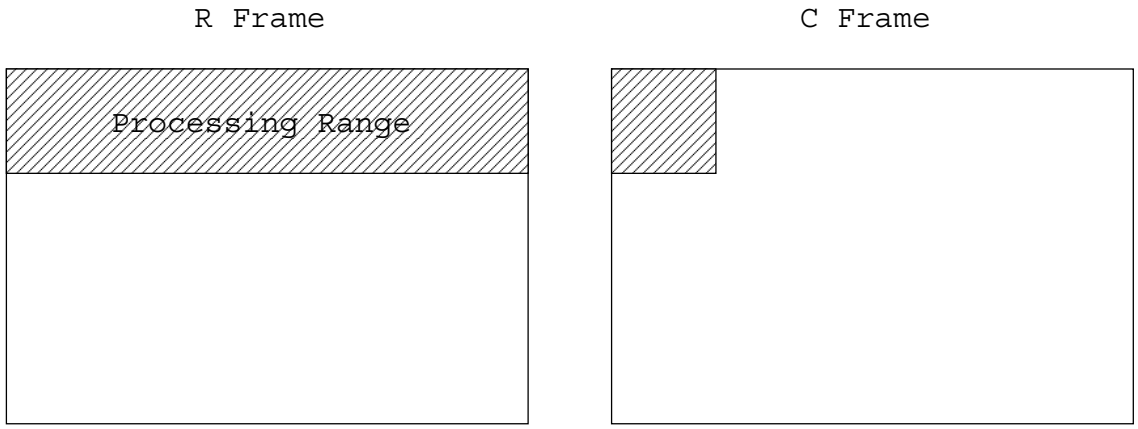


図 5.4: 参照フレームの水平ループの処理範囲

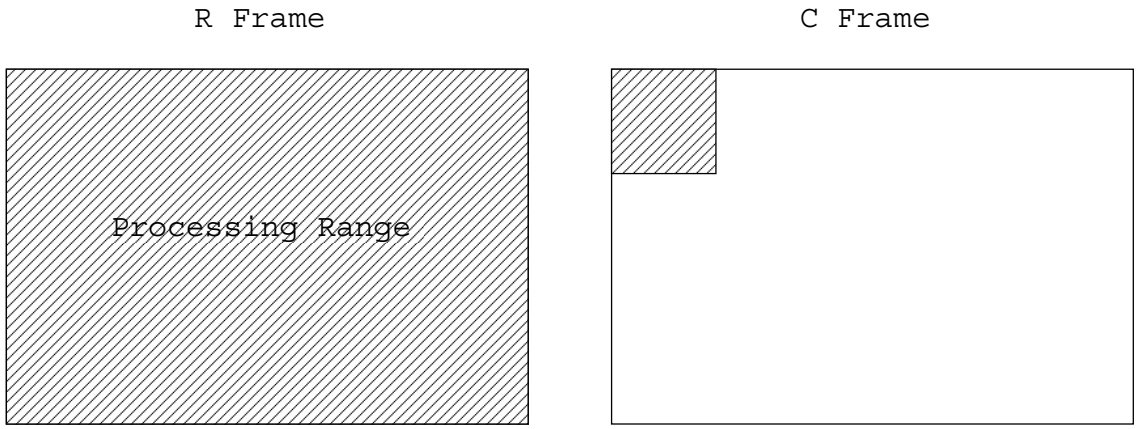


図 5.5: 参照フレームの垂直ループの処理範囲

5.4 対象フレームレベルの処理

5.4.1 水平方向

対象フレームの水平方向の処理範囲を図 5.6 に示す。参照フレームの全処理範囲のマクロブロックの輝度値の差の総和が求まった時点で、或る対象フレームのマクロブロックの動きベクトルが求まる。これより、動きベクトルをメモリに保存する。またこの処理のループ回数は、320 × 240 画素のフレームのため横のマクロブロック数である 20 回分の処理を行う。

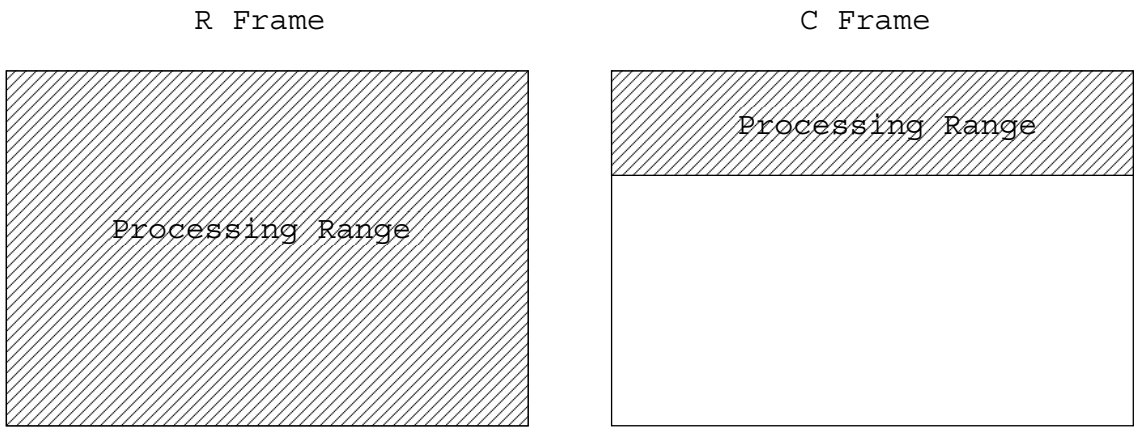


図 5.6: 対象フレームの水平ループの処理範囲

5.4.2 垂直方向

対象フレームの垂直方向の処理範囲を図 5.7 に示す。垂直方向のループも同様に、処理回数は縦のマクロブロック数である 15 回分の処理を行う。

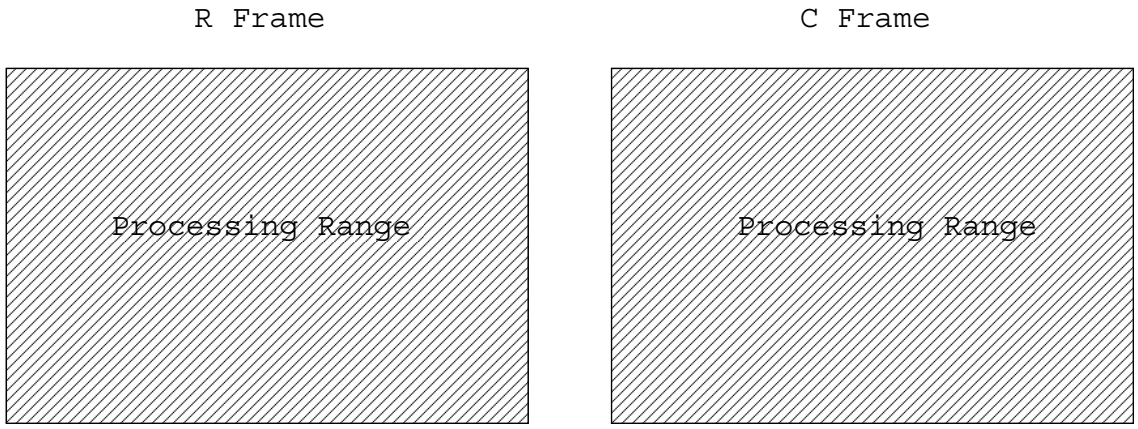


図 5.7: 対象フレームの垂直ループの処理範囲

5.5 階層的マッチング法の処理

階層的マッチング法では、3 段階の階層により動きベクトル検索処理を行うことにする。1 段階目では 4 画素毎の縮小画像である参照フレームの全領域 80 × 60 画素に対して 8 × 8 画素のブロックで 1 画素毎に検索を行う。これは 32 × 32 画素の範囲のブロックで検索することに相当する。2 段階目では 2 画素毎の縮小画像である参照フレームにおいて、1 段

階目の処理で得られた最適位置を中心として、4 × 4 画素の範囲を 16 × 16 画素のマクロブロックで 1 画素毎に検索を行う。3 段階目では縮小していない原画像より、2 段階目の処理で得られた最適位置を中心として、4 × 4 画素の範囲を 16 × 16 画素のマクロブロックで 1 画素毎に検索を行うものとする。

5.6 検索領域限定法の処理

検索領域限定法の検索範囲は、QVGA サイズの場合、水平± 100 画素、垂直± 50 画素で行うのが好ましいとされている [1]。本手法ではマクロブロックごとに検索するため水平± 96 画素、垂直± 48 画素の範囲に対して検索を行うものとする。また、参照フレームでの検索領域の位置は、対象フレームのマクロブロック位置により決まる。そのため、対象フレームのマクロブロック位置が端のほうだと検索領域は少なくてすむ。この検索領域の決定を検索領域限定法の命令列で行う。

5.7　命令列の見積もり

5.7.1　総ステップ数の見積もり

各ループのステップ数から総ステップ数の見積もりを行う。まず、マクロブロックの処理ステップ数から見積もる。その詳細を表 5.1 に示す。また、階層的マッチング法により 1 段階目は 8 × 8 画素の範囲で検索を行う。

本手法では 16 個ごとにマクロブロックを処理するため、表 5.1 の結果は 16 個分のステップ数である。次に、対象フレームの 1 つのマクロブロックに対して動きベクトルを求める。検索領域限定法と階層的マッチング法の適用を考慮しつつ、先ほどの 16 個分のマクロブロックの処理を何回行うか見積もる。まず、階層的マッチング法により 1MB 分ごとに一度に処理が行える。次に、検索領域限定法では水平± 6MB，垂直± 3MB 分の処理を行う。対象フレームのマクロブロック位置が端の位置だと、参照フレームの検索領域は少ない範囲であり、対象フレームのマクロブロック位置が中央の位置だと、参照フレームの検索領域は広い範囲を検索する。これより、最小で 6 × 3=18、最大で 12 × 6=72 マクロブロック分の処理を行う。対象フレームの位置対しての参照フレームの処理ループ回数を見積もる。水平方向のループ回数の見積もりを表 5.2 に示す。また、垂直方向のループ回数の見積もりを表 5.3 に示す。表 5.2、5.3 より、平均処理回数は 9.9 × 5.2=51.5 である。以上のことより、対象フレームの 1 マクロブロックに対して、16 個分の処理を行うステップ数は表 5.4 のようになる。また、対象フレームの全てのマクロブロックを処理した時のステップ数を表 5.5 に示す。

次に、階層的マッチング法の 2 段階目および 3 段階目の処理のステップ数を見積もる。まず、マクロブロックループでの処理は、1 段階目と異なり 16 × 16 画素の範囲で処理を行う。これを表 5.6 に示す。次に、階層階層的マッチング法の 2 段階目および 3 段階目の処理では、1 段階目と異なりフレームの全領域を調べるわけではなく、最適位置からの詳細検索であるため、参照フレームループの回数は 1 回である。詳細検索時の対象フレームの 1MB 分の処理ステップ数を表 5.7 に示す。

次に、対象フレームのすべてのマクロブロックを処理した時のステップ数を表 5.8 に示す。

最後に、動きベクトル検索すべての処理の総ステップ数を見積もる。これを表 5.9 に示す。2 段階目および 3 段階目の処理は処理ステップが同じであるので、ループ回数を 2 とする。これより総ステップ数は 64,549,986 となる。

表 5.1: 16 個のマクロブロックの処理ステップ数

	ステップ数	ループ回数	合計
マクロブロック　水平	53	64	3392
マクロブロック　垂直	9	8	72
マクロブロック　合計			3464

表 5.2: 参照フレーム水平ループのループ回数

														平均
T frame の MB 位置	1	2	3	4	5	6	7-14	15	16	17	18	19	20	
処理 MB 数	6	7	8	9	10	11	12	11	10	9	8	7	6	9.9

表 5.3: 参照フレーム垂直ループのループ回数

								平均
T frame の MB 位置	1	2	3	4-12	13	14	15	
処理 MB 数	3	4	5	6	5	4	3	5.2

表 5.4: 対象フレームの 1MB 分の処理ステップ数

	ステップ数	ループ回数	合計
マクロブロック　合計	3464	51.5	178326.7
参照フレーム　水平	105	51.5	5405.4
参照フレーム　垂直	9	5.2	46.8
検索領域限定法	31	1	31
合計			183809.9

表 5.5: 対象フレームのすべての MB 分の処理ステップ数

	ステップ数	ループ回数	合計
参照フレーム　合計	183809.9	300	55142976
対象フレーム　水平	15	300	4500
対象フレーム　垂直	9	15	135
合計			55147611

表 5.6: 16 個のマクロブロックの処理ステップ数（Step2,3）

	ステップ数　ループ回数	合計
マクロブロック　水平	53　256	13568
マクロブロック　垂直	9　16	144
マクロブロック　合計		13712

表 5.7: 対象フレームの 1MB 分の処理ステップ数（Step2,3）

	ステップ数　ループ回数	合計
マクロブロック　合計	13712　1	13712
参照フレーム　水平	104　1	104
参照フレーム　垂直	8　1	8
階層的マッチング法	37　1	37
合計		13861

表 5.8: 対象フレームのすべての MB 分の処理ステップ数 (Step2,3)

	ステップ数　ループ回数	合計
参照フレーム　合計	13861　300	4158300
対象フレーム　水平	17　300	5100
対象フレーム　垂直	11　15	165
合計		4163565

表 5.9: 動きベクトル検索処理の総ステップ数

	ステップ数　ループ回数	合計
階層処理　1 段階目	55147611　1	55147611
階層処理　2、3 段階目	4163565　2	8327130
縮小画像作成処理	1075200　1	1075200
Start 処理	8　1	8
End 処理	37　1	37
合計		64549968

表 5.10: 命令によるステージ構成

		ステージ数
lw	IF ID EX MEM × 12 WB	16
sw	IF ID EX MEM × 12	15
それ以外の命令	IF ID EX WB	4

表 5.11: 各ループごとのサイクル数

	ステップ数	サイクル数	ストール数
マクロブロック　水平	53	56	3
マクロブロック　垂直	9	13	4
参照フレーム　水平	105 (104)	199 (195)	94 (90)
参照フレーム　垂直	9 (8)	18 (14)	9 (6)
対象フレーム　水平	15 (17)	23 (30)	8 (13)
対象フレーム　垂直	9 (11)	17 (24)	8 (13)
検索領域限定法	31	53	22
階層的マッチング法	37	76	39
Start	8	8	0
End	11 (13)	19 (26)	8 (13)

5.7.2　CPI の見積もり

総ステップ数が分かったので、次に CPI の見積もりを行う。作成した動きベクトル検索処理を行う命令列に対して、パイプラインハザードの回避を行うために、ストールサイクルを挿入する。また、さらにコードスケジューリングを行うことで最適化を行う。これには各命令のステージ数が関係する。命令ごとのステージ数を表 5.10 に示す。メモリアクセス命令はステージ数が長く、lw 命令が 16 ステージで sw 命令が 15 ステージとする。それ以外の命令は全て 4 ステージとする。

各ループごとのステップ数とコードスケジューリング後のサイクル数からストール数を見積もった。これを表 5.11 に示す。また () 内の値は階層的マッチング法の 2 段階目、3 段階目の処理を示す。

表 5.11 より、総ステップ数の見積もり方と同じように、各ループでのサイクル数より計算を行い、総サイクル数を求めると、1 スレッドでのサイクル数は 70,147,162 である。以上のことから CPI は

$$CPI = 70147162 \div 64549986 = 1.087 \tag{5.1}$$

表 5.12: QVGA 動画処理の見積もり

クロック周波数 (GHz)	1.0	2.0	2.2
CPU 実行時間 (秒)	0.070	0.035	0.032
30frame の処理時間 (秒)	2.10	1.05	0.96

である。

5.7.3 QVGA 動画処理の見積もり

次に、サイクル数が分かったことにより、クロック周波数を仮定し、フレームサイズが QVGA でフレームレートが 30fps の動画像を 1 秒以内に処理できるかを見積もる。クロック周波数毎の、CPU 実行時間と 30 フレームの処理に要する処理時間を見積もった結果を表 5.12 に示す。

CPU 実行時間は以下の式で求める。

$$CPU \text{ 実行時間} = \text{サイクル数}/\text{クロック周波数} \tag{5.2}$$

また、CPU 実行時間は QVGA の 1 フレームの処理時間であるので、30 フレームを処理する処理時間は以下の式で求めることができる。

$$30 \text{ フレームの処理時間} = CPU \text{ 実行時間} \times 30 \tag{5.3}$$

表 5.12 より、クロック周波数を 2.2GHz とすると、30 フレームの処理に要する処理時間が 1 秒以内に収まり、QVGA 動画が時間内に処理可能であることを示している。

5.7.4 XVGA 動画処理への適用

次に、フレームサイズが XVGA(1024×768) の動画を処理する場合について考える。XVGA はフレームサイズが QVGA(320×240) の 10 倍ほどの大きさのため、QVGA の見積もりから、単純に見積もると、クロック周波数は 10 倍程度必要になると考えられる。これより、XVGA 動画を規定時間内に処理することができるようにするために以下の事を行う。

- ステップ数の削減

  XVGA は QVGA よりもフレームサイズが大きいため階層的マッチング法において、よりおおまかに検索することができる。さらに小さい縮小画像で検索を行うことにより、処理量であるステップ数の削減ができる。

表 5.13: XVGA 動画処理のための必要クロック周波数

クロック周波数 (GHz)	1.0	2.0	3.0	4.0
CPU 実行時間 (秒)	0.125	0.062	0.041	0.031
30frame の処理時間 (秒)	3.75	1.88	1.25	0.94

- パイプラインステージの細分化

  数 GHz のクロック周波数を達成するには、各ステージのクロック間隔時間が非常に短くなくてはならない。そのためには、パイプラインステージを細分化し、1 ステージの論理段数を短くする必要がある。これにより、高周波数を達成することができる。

- マルチスレッド処理

  パイプラインステージを細分化することにより、パイプラインが長くなる。パイプラインハザードを回避するためには、命令列に対してその分ストールサイクルを挿入する必要がある。これにより、サイクル数の増加により CPI が悪化する。サイクル数の増加を抑える方法として、マルチスレッド処理によりストールを別のスレッドの命令で埋めることが考えられる。

ここで、階層的マッチング法をおおまかに行うことにより、ステップ数を削減した場合の、XVGA 動画像を時間内に処理するための必要クロック周波数を見積もる。ステップ数を見積もった結果、ステップ数としては 115,187,682 となる。仮に CPI が先ほどと同じ 1.087、サイクル数は 125,181,200 として、XVGA 動画像を時間内に処理するための必要クロック周波数を見積もった結果を表 5.13 に示す。表の結果より、クロック周波数が 4GHz の時に 30 フレームの処理に要する処理時間が 1 秒以内に収まっている。またこの結果は CPI が 1.087 の時の結果であり、XVGA への適用によりパイプラインを細分化し、パイプラインのステージ数の増加により CPI が悪化するが、マルチスレッド処理により CPI を下げることができれば、今の 4GHz より小さいクロック周波数で処理時間以内に処理が可能であると想定できる。

第6章　マルチスレッド型プロセッサ処理の適用

本章ではXVGA動画を時間内に処理するためのパイプラインステージの細分化とマルチスレッド処理について述べる。

6.1　マルチスレッド型プロセッサ

マルチスレッド処理は、パイプライン処理で複数の独立な命令流 (スレッド) を並列に実行することにより、パイプラインハザードを回避し、パイプラインのスループットを向上させる方法である。マルチスレッド型プロセッサのパイプライン処理の概要を図6.1に示す。マルチスレッド処理では、サイクル毎に異なるスレッドから命令を発行する。これにより、各スレッドの命令には依存関係が存在しないため、パイプラインの各ステージをすべて有効な命令で埋めることができる。また、パイプラインステージ数分のスレッドとレジスタセットを用意することで完全にパイプラインハザードを回避することができる。

本研究では処理対象とする動画像データは、データに依存関係がないため、各スレッドを依存関係がない独立なスレッドに分割可能である。また、このような並列処理に適したデータを処理する場合のマルチスレッド処理方式の特徴について伊藤らが示している[2]。

マルチスレッド処理では、命令レベルの並列処理とマルチスレッド処理を組み合わせることでより大きな並列処理を可能とする。命令レベルの並列処理とマルチスレッド処理を組み合わせたプロセッサ方式として、マルチスレッド型スーパーパイプライン・プロセッサ・アーキテクチャがある。この方式はパイプラインの処理単位を細分化し、動作クロックを高める方式である。パイプラインの処理段数の増加に伴って、パイプラインハザードによるペナルティが大きくなるが、マルチスレッド処理によりスループットの向上が可能である。また、パイプライン段数の細分化により多くのスレッドを必要とするが、処理単位となるデータが並列処理に適している動画像のためこれに適している。

動きベクトル検索ではマクロブロックごとの処理に対しては依存関係がない。そのため、多数の並列実行可能なスレッドを供給することが可能である。一方、マルチスレッド型プロセッサは、プロセッサの性能を引き出すために複数のスレッドを必要とする。これより、マルチスレッド型プロセッサに対して多数のスレッドを供給することによって、動作周波数の高速化による高スループット化が実現可能である。

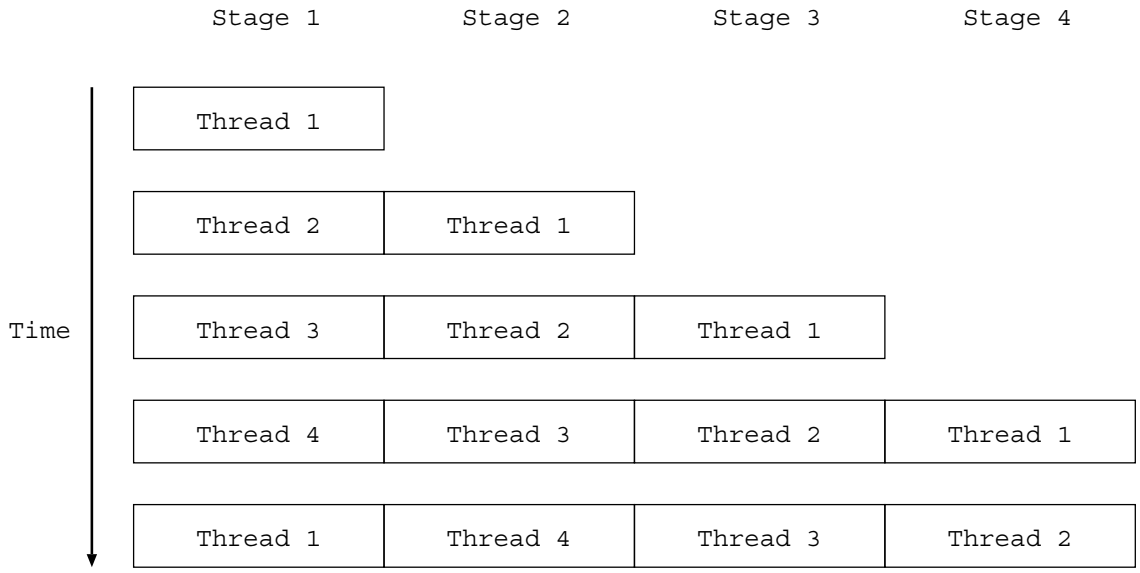


図 6.1: マルチスレッド型プロセッサのパイプライン処理

6.2 MUP

ここで、伊藤らが設計した Multithreaded Ultrapipeline Processor(MUP) と言うマルチスレッド型プロセッサについて説明する。

MUP はパイプラインステージ数が 17 段と長く、各ステージのクロック間隔時間を短くすることで、高周波数動作を実現可能としている。スレッド数とレジスタセットはステージ数に等しく 17 であり、スレッドとレジスタセットをステージ数分用意することで、パイプラインハザードを完全に回避する。

6.2.1 MUP のステージ構成

MUP のパイプラインステージの構成は以下の通りである。

1. TS1: スレッド番号の選択
2. TS2: スレッドのプログラムカウンタとレジスタセットを読み出す
3. IF1: 命令キャッシュのアドレスをデコード
4. IF2: 命令キャッシュのメモリセルを読み出す
5. IF3: 命令キャッシュのタグを読み出す
6. IF4: 命令キャッシュアクセスの終了、キャッシュミスの場合はスレッドを待避
7. RF1: 命令をデコード、レジスタファイルへのアクセス開始
8. RF2: レジスタオペランドのフェッチが完了
9. EX1: ALU の第 1 ステージ

10. EX2:　ALU の第 2 ステージ
11. EX3:　ALU の第 3 ステージ
12. DF1:　データキャッシュのアドレスデコード
13. DF2:　データキャッシュのメモリセルの読み出しや書き込み
14. DF3:　データキャッシュのタグを読み出す
15. DF4:　データキャッシュのアクセス終了
16. WB1:　レジスタファイルへの書き込み開始
17. WB2:　レジスタファイルへの書き込み終了

6.3　MUP によるステージの細分化の検討

本研究では高周波数動作とマルチスレッド処理を実現するために、MUP を参考にして、ステージ分割を行う。そして、XVGA を時間内に処理するための動作周波数 4GHz 程度を達成するために、必要なパイプラインステージ構成などを見積もる。また、MUP はその原理上パイプラインをどれだけ細分化してもかまわない。その際キャッシュメモリやレジスタファイルが動作周波数のボトルネックにならないように、これらもパイプライン化される。また、キャッシュメモリについては、鵜飼 [3]、相原 [4] によれば、キャッシュのパイプライン化により 100nm テクノロジで 7.7GHz の動作周波数を実現している。

6.3.1　分割ステージ数の検討

MUP では高周波数で動作させるために、各ステージの論理段数を小さく抑えている。これにより 100nm CMOS テクノロジにより 1GHz のクロック周波数動作を実現可能としている。各ステージの論理段数を表 6.1 に示す。表より、各ステージの論理段数は 3 から 5 で構成されている。高周波数動作を可能とするために、ステージを分割して各ステージの論理段数を小さくすることを考える。すべてのステージの論理段数を 3 に抑えるようにステージを分割した。分割した後のステージ構成と論理段数を表 6.2 に示す。この表のステージ構成はデータメモリのメモリアクセスステージである DF ステージを必要としない命令のステージ構成である。表 6.2 より、メモリアクセスステージがない lw と sw 以外の命令のステージ数は、TS が 3、IF が 6、RF が 3、EX が 4、WB が 3 のため総ステージ数は 19 である。

6.3.2　動作可能周波数の検討

次に、ステージの最大論理段数が 5 から 3 に変わったことによる動作可能周波数の向上率を見積もる。

表 6.1: MUP の各ステージの論理段数

ステージ	段数	ステージ	段数
TS1	3	EX1	4
TS2	4	EX2	5
IF1	5	EX3	3
IF2	-	DF1	5
IF3	4	DF2	-
IF4	5	DF3	4
RF1	4	DF4	5
RF2	5	WB1	4
-	-	WB2	5

表 6.2: XVGA 処理用 CPU の各ステージの論理段数 (ステージ分割後)

ステージ	段数	ステージ	段数
TS1	3	RF1	3
TS2	2	RF2	3
TS3	2	RF3	3
IF1	3	EX1	3
IF2	2	EX2	3
IF3	-	EX3	3
IF4	3	EX4	3
IF5	3	WB1	3
IF6	3	WB2	3
-	-	WB2	3

$$\text{動作可能周波数の向上率} = 5/3 = 1.666 \tag{6.1}$$

式 6.1 より動作可能周波数 1.666 倍となる。しかしこれではまだ、XVGA が処理可能なクロック周波数にはみたない。そこで、最先端の CMOS テクノロジを採用することで速度向上を図る。CMOS テクノロジとしては 45nm テクノロジを採用する。素子の動作周波数は素子寸法に比例するとして、100nm と比べて動作可能周波数の向上率を見積もる。

$$\text{動作可能周波数の向上率} = 100/45 = 2.222 \tag{6.2}$$

式 6.2 より動作可能周波数 2.222 倍となる。これにより、ステージ分割と最先端の CMOS テクノロジの採用を合わせた全体の動作可能周波数の向上率を見積もる。

$$\text{動作可能周波数の向上率} = 1.666 \times 2.222 = 3.701 \tag{6.3}$$

式 6.3 より動作可能周波数 3.701 倍となる。MUP の動作周波数が 1GHz であるため、向上率による動作可能周波数を見積もる。

$$\text{動作可能周波数} = 3.701 \times 1GHz = 3.701GHz \tag{6.4}$$

式 6.4 より動作可能周波数はほぼ 3.7GHz である。

6.3.3 メモリアクセスステージ数の検討

次にデータメモリのメモリアクセスステージである DF ステージのステージ構成の見積もりを行う。ロード回数削減法により見積もったメモリアクセスステージ数は 12 であった。しかし、この数値は動作可能周波数が 1GHz 程度の場合のステージ数である。そのため、動作周波数が 3.7GHz でもメモリアクセス時間を覆い隠せるほどのステージ数を見積もる。

まず、DDR SDRAM では速くて 30ns 程度のメモリアクセス時間が実現できる。これにより SRAM では DRAM の 4 倍程度と仮定すると、

$$SRAM \text{ のメモリアクセス時間} = 30/4 = 7.5ns \tag{6.5}$$

式 6.5 より SRAM のメモリアクセス時間は 7.5ns 程度である。次に、動作周波数が 3.7GHz の場合クロック間隔時間は 0.27ns である。これにより、SRAM のメモリアクセス時間 7.5ns とクロック間隔時間 0.27ns より必要ステージ数を見積もる。

$$\text{必要ステージ数} = 7.5/0.27 = 27.7 \tag{6.6}$$

式 6.6 より、必要ステージ数としては 28 あれば、メモリアクセスステージに SRAM のメモリアクセス時間が収まる。これより、DF ステージは 28 ステージとする。DF ステージ数が分かったので、6.3.1 で検討したメモリアクセスステージがない命令の総ステージ数である 19 ステージと足して、メモリアクセス命令である lw と sw の総ステージ数を見積もる。

$$\text{総ステージ数} = 19 + 28 = 47 \tag{6.7}$$

式 6.7 より、lw と sw のステージ数は 47 ステージとする。

6.4 スレッドの分割法

動画像データはデータに依存関係がないため、処理の各行程を複数のスレッドに分割するのが容易であり、マルチスレッド化に適している。分割するスレッドとしては、各スレッド同士はお互い依存関係がないようにする必要がある。さらに、無駄なく処理するためには各スレッドの処理サイクルが等しい方がよい。

スレッドの分割方法としては、対象フレームのマクロブロックごとにスレッド分割を行うものとする。対象フレームでは、各マクロブロックごとに動きベクトルを求める処理を行うため、各マクロブロックごとの処理の工程はほぼ同じである。ただ異なる点としては対象フレームでは検索領域限定法のため、対象フレームのマクロブロックの位置によって参照フレームで検索するマクロブロックの数が異なる。対象フレームのマクロブロック位置が端に近づくほど検索領域が少なく、中央に近づくほど検索領域は多い。これでは、処理回数が異なるため、スレッドを複数に上手く分割できない。これにより、対象フレームのマクロブロック位置に関係なく検索領域限定法での検索領域は同じとする。スレッド分割数ごとの各スレッドの処理範囲を図 6.2 に示す。

2 スレッド、4 スレッド、8 スレッド、16 スレッドでのマルチスレッド処理を示す。また、対象フレームではマクロブロックごとの処理のため、分割した境界にまたがるような処理はないため、処理をスレッドごとに分割できる。しかし、マルチスレッド化には動画像のフレームサイズが関係し、フレームの全マクロブロック数がスレッド数の公約数でないと上手くマクロブロック単位に処理が割り当てられないことを示している。本手法によりフレームサイズによって可能なスレッド数について調べた結果を表 6.3 に示す。

表 6.3 より、QVGA では 4 スレッドまでの処理が可能であり、VGA では 8 スレッドまでの処理が可能である。また、XVGA では 16 スレッドまでの全マルチスレッド処理が可能である。フレームサイズが大きくなるにつれ、スレッド数が増やすことができているため、フレームサイズが大きくなることにより、処理量が増加しても上手くマルチスレッド処理によりその処理量の増加にも対応できるのではないかと考えられる。

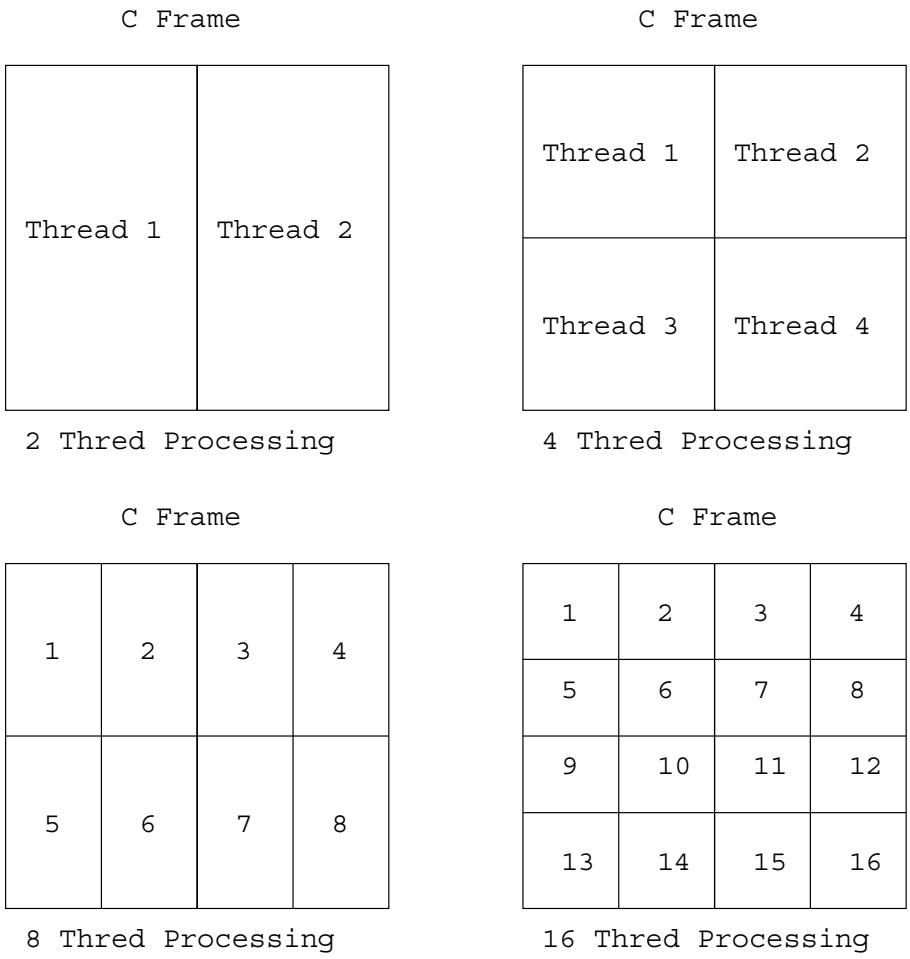


図 6.2: スレッド数による処理範囲

6.5 マルチスレッド処理による CPI の見積もり

サイクル数を見積もった結果、パイプラインステージ数が増え、メモリアクセスステージがない命令では 19 ステージであり、lw と sw 命令は 47 ステージであった。パイプラインハザードが発生する命令列には、パイプラインハザードを回避するために、ストールサイクルを挿入する必要がある。ストールサイクルを挿入する必要があるケースについて示す。

まず、メモリアクセスステージがない命令間でデータハザードによるケースである。これを表 6.4 に示す。このケースは SUB 命令は ADD 命令の結果に依存している。これにより、SUB では RF ステージの前に ADD の WB ステージで書き込みが終えていればよい。この表 6.4 より SUB の前に挿入するストールサイクル数としては、ステージが重なる TS と IF ステージ数と ADD 命令の 1 サイクルを総ステージ数よりひくことで求まる。

表 6.3: フレームサイズによるマルチスレッド化

名称	水平画素 (MB)	垂直画素 (MB)	2 Thread	4 Thread	8 Thread	16 Thread
QVGA	320(20)	240(15)	○	○	×	×
VGA	640(40)	480(30)	○	○	○	×
XVGA	1024(76)	768(48)	○	○	○	○

表 6.4: データハザードを回避するためのストールサイクル数

stage(ADD)	stage(SUB)	Inst	rd	rs1	rs2
TS		ADD	r1	r2	r3
IF		stall			
RF		stall			
EX	TS	SUB	r4	r1	r5
WB	IF				
	RF				

$$\text{ストールサイクル数} = 19 - (3 + 6 + 1) = 9 \tag{6.8}$$

次のケースとしては制御ハザードによるケースが考えられる。これを表 6.5 に示す。条件分岐命令の後は次にどの命令を実行するかわからないため、WB ステージの後に次の ADD 命令の IF ステージがあればよい。そのため、スレッドの決定を行う TS ステージは前の命令と重ねることができる。これにより、挿入するストールサイクル数は以下のようになる。

$$\text{ストールサイクル数} = 19 - (3 + 1) = 15 \tag{6.9}$$

次のケースではメモリアクセスステージのある lw 命令によりデータハザードが発生するケースについて考える。このケースでは、最初のメモリアクセスステージがない命令でのデータハザードのケースに比べてメモリアクセスステージ分ステージ数が長くなるので、挿入するストールサイクル数は以下のようになる。

$$\text{ストールサイクル数} = 9 + 28 = 37 \tag{6.10}$$

表 6.5: 制御ハザードを回避するためのストールサイクル数

stage(Bne)	stage(ADD)	Inst	rd	rs1	rs2
TS		Bne	r1	r2	Loop
IF		stall			
RF		stall			
EX		stall			
WB	TS	ADD	r3	r4	r5
	IF				

表 6.6: 各ループによるサイクル数の見積もり（マルチスレッド）

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
マクロブロック　水平	89	76	62	57	53
マクロブロック　垂直	38	21	14	11	9
参照フレーム　水平	532	302	207	144	105
参照フレーム　垂直	63	33	22	12	9
対象フレーム　水平	75	40	26	18	15
対象フレーム　垂直	53	28	16	12	9
検索領域限定法	150	82	57	39	31
階層的マッチング法	214	129	76	52	37
Start	10	8	8	8	8
End	44	29	21	17	14

それぞれのケースにより、最大で 37 のストールサイクルを挿入する必要がある。この場合、マルチスレッド化により 16 スレッドで実行しても、ストールサイクルに命令を埋めきれない。しかし、コードスケジューリングにより依存関係のない命令を埋めることによりこれを回避することが可能である。これよりパイプラインハザードによりストールサイクルを挿入する必要がある数を 16 以下にすれば、16 スレッドのマルチスレッド処理を行うことにより、パイプラインハザードを完全に回避することができる（付録参照)。

スレッドを複数にしたときの各ループでのサイクル数の見積もりを表 6.6 に示す。表 6.6 よりパイプラインステージ数が大幅に長くなったため、サイクル数は大幅に増大している。しかし、マルチスレッドによりスレッド数が増えるごとにサイクル数を抑えることができている。

これより、スレッド数でのサイクル数と CPI を見積もった。これを表 6.7 に示す。表 6.7 より、サイクル数の増加により大幅に CPI は悪化し 1 スレッドでは 1.69 である。またコー

表 6.7: スレッド数によるサイクル数と CPI

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
サイクル数	194,635,469	162,413,697	128,327,517	117,030,753	115,187,682
CPI	1.69	1.41	1.11	1.02	1.00

表 6.8: スレッド数による処理時間

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
CPU 実行時間 (秒)	0.053	0.044	0.035	0.032	0.031
30frame の処理時間 (秒)	1.58	1.32	1.04	0.95	0.93

ドスケジューリングにより最適化しているため、4 スレッドでもかなり CPI は改善されている。8 スレッドではほぼ 1 であり、16 スレッドでは CPI は 1 であり完全にパイプラインハザードを回避できている。

6.6 評価

前節までで見積もった XVGA 動画の動きベクトル検索処理のサイクル数を、ステージ分割を行い、クロック周波数が 3.7GHz で動作可能な MUP により処理した場合の処理時間を見積もった。CPU 実行時間と 30 フレームの処理時間を表 6.8 に示す。表 6.8 より、8 スレッドと 16 スレッドでは 30 フレームの処理時間が 1 秒以内に収まっている。また、1 スレッドの 1.58 秒と比べると、8 スレッドと 16 スレッドでは 1 秒以下のため 3 分の 2 程度となっており、マルチスレッド処理による効果が十分に現れていると言える。

第7章　結論

本論文では、動画像のエンコードの処理速度を速めるために、動き補償処理のための専用プロセッサを設計した。

はじめに、一般的な計算量削減法である階層的マッチング法と検索領域限定法を適用し、計算量の削減を試みた。さらに、プロセッサのアーキテクチャを考慮し、命令数に対してのサイクル数を減らすロード回数削減法を提案した。

また、動き補償処理に特化した命令セットを設計にすることにより、さらに処理を効率的に行えるようにした。

次に、上記の動きベクトル検索の計算量削減法とロード回数削減法を適用した動きベクトル検索処理のための命令列を作成した。

QVGA サイズの動画像処理を見積もった結果、CPI は 1.08 を達成できた。30fps を処理するためには動作周波数が 2.2GHz 必要であることが分かった。

次に、動画像データはデータに依存関係がないため、マルチスレッド化について検討した。動作可能周波数の向上のために、パイプラインステージの細分化を行った。パイプラインステージは演算命令で 19 段、ロード/ストア命令で 47 段とした。パイプラインの論理段数は 3 段に抑え、45nm テクノロジで 3.7GHz を見込んだ。

パイプラインハザード回避のためのストールサイクルの挿入によりサイクル数が増大し、CPI が悪化したが、マルチスレッド処理によりストールサイクルに別のスレッドの命令を埋めることでパイプラインハザードを完全に回避し、CPI の悪化を防ぐことができることを示した。

マルチスレッド処理を取り入れ、XVGA サイズの動画像処理を見積もった結果、CPI は 1 スレッドで 1.69、16 スレッドで 1 を達成できた。また、30fps の処理を動作周波数 3.7GHz、16 スレッドで達成できた。

マルチスレッド処理により CPI の改善が得られた。最終的にハイビジョンクラスの動画像である XVGA でフレームレートが 30fps の動画像の動きベクトル検索処理を 1 秒以内に行うことができることを示した。

謝辞

本研究を進めるにあたり熱心に御指導をしてくださった日比野靖教授に深く感謝いたします。

また、貴重な御意見をくださった本学の田中清史助教授、井口寧助教授に深く感謝いたします。

そして、日頃から御相談に受けて下さった菅原英子助手に深く感謝いたします。

その他、貴重な御意見、御討論を頂きました日比野研究室の皆様をはじめ、多くの方々の御助言に対し厚く御礼申し上げます。

参考文献

[1] 大塚竜元, MPEG-2 ビデオエンコーダ LSI の開発とその応用, FUJITSU, 50, 2, pp104-108, (03, 1999)

[2] 伊藤英治, 関数型プログラムの実行に適したマルチスレッド型プロセッサ・アーキテクチャに関する研究, 北陸先端科学技術大学院大学修士論文, 1997

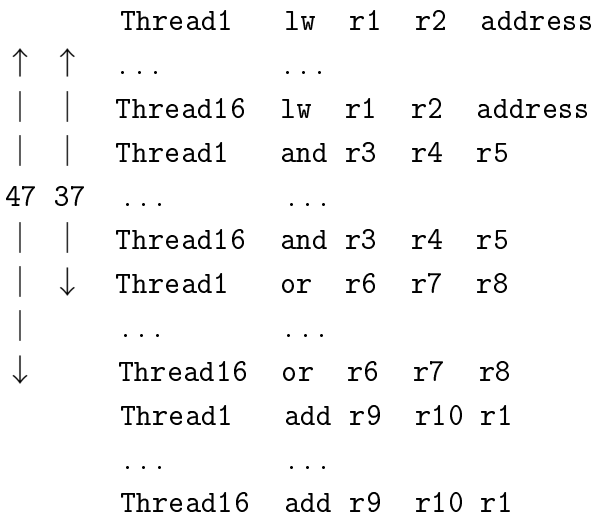
[3] 鵜飼和歳, パイプライン化によるキャッシュの高周波動作の可能性, 北陸先端科学技術大学院大学修士論文, 1999

[4] 相原孝一, マルチスレッド型プロセッサ向きのキャッシュ機構のパイプライン化に関する研究, 北陸先端科学技術大学院大学修士論文, 1997

付録

パイプラインハザードが発生する命令列には、パイプラインハザードを回避するためにストールサイクルを挿入する必要がある。6.5 章で述べたようにlw 命令のデータハザードにより、37 サイクルのストールを挿入する必要がある。この場合、マルチスレッド化により 16 スレッドで実行してもストールサイクルを埋めきれない。コードスケジューリングを行い、この 37 サイクル中に依存関係のない命令を挿入し、16 スレッドのマルチスレッド処理により、パイプラインハザードを完全に回避することが可能である。

lw 命令のデータハザードが発生している命令列に対して、コードスケジューリングを行った命令列を以下に示す。この命令列ではlw 命令の結果である r1 をadd 命令でソースとして使用しているため、lw 命令とadd 命令の間に依存関係がある。このため、lw 命令とadd 命令の間にデータハザード分の 37 サイクルを空ける必要がある。コードスケジューリングにより依存関係のない命令を挿入することにより、ストールサイクルを挿入することなく、全て有効な命令で埋めることができる。また、16 スレッドのマルチスレッド処理を行うことで、以下の命令列では、and 命令とor 命令の 2 命令だけをコードスケジューリングで挿入することにより、lw 命令とadd 命令の間に 47 サイクル分空けることができ、ハザードを回避することができる。


Thread1 lw r1 r2 address
...
Thread16 lw r1 r2 address
Thread1 and r3 r4 r5
47 37
...
Thread16 and r3 r4 r5
Thread1 or r6 r7 r8
...
Thread16 or r6 r7 r8
Thread1 add r9 r10 r1
...
Thread16 add r9 r10 r1