■6 群（コンピュータ －基礎理論とハードウェア）－ 5 編（コンピュータアーキテクチャ（II）先進的）

1 章 命令レベル並列コンピュータ

（執筆者：佐藤寿倫）［2010 年 5 月 受領］

■概要■

単一プロセッサの性能向上には命令レベル並列性の抽出が必須である．本章では，成熟した技術である制御投機方式から話を始め，研究段階から実用化に移行しつつあるタイル型コンピュータに至るまで，幅広い話題を解説する．

【本章の構成】

命令レベル並列性の抽出にとって，投機方式はそのための重要な技術である．1-1 節では制御フローにおける投機方式を，1-2 節ではデータフローにおける投機方式を解説している．並列性拡大を大きく推し進めたものが 1-3 節で解説されるタイル型コンピュータである．これは命令レベル並列コンピュータと 2 章で解説されるスレッド並列コンピュータの境界に位置すると考えることができる．

■6 群 － 5 編 － 1 章

1-1 制御投機方式

（執筆者：安藤秀樹）［2008 年 9 月 受領］

1-1-1 制御投機

分岐命令があると，それに続いてどの命令を実行すべきかは，分岐命令の実行結果に依存する．このため，分岐命令が実行ステージに至るまでの間パイプラインはストールする．**図 1･1** のコードを例に説明する．

```
    i0: add r1, r2, r3
    i1: beq r4, r5, L
    i2: and r6, r7, r8
    i3: sub r9, r10, r11
    ...
L:  i9: or r12, r13, r14
```

図 1･1　コード例

このコードでは命令 `i1` の `beq` は分岐命令で，分岐が不成立なら命令 `i2` を次に実行し，成立なら命令 `i9` を実行する．

図 1･2 に実行タイミングを示す．分岐命令がなければ，命令 `i0` と `i1` のように，パイプラインはストールすることなく命令は滞りなく流れていく．しかし，分岐命令があると，それが実行ステージ（EX）に至るまで，次に実行すべき命令が定まらず，パイプラインはストールする．この例では，クロックサイクル 2 で次の命令はフェッチできず，クロックサイクル 3 での分岐命令 `i1` の実行結果が出力されるまでパイプラインはストールする．そして，この例では，分岐不成立で，クロックサイクル 4 に命令 `i2` がフェッチされている．

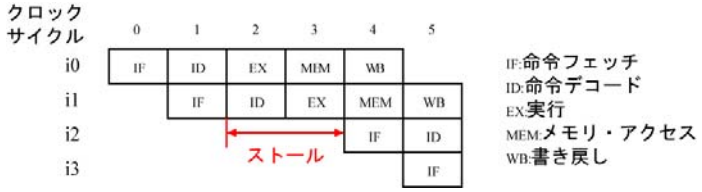
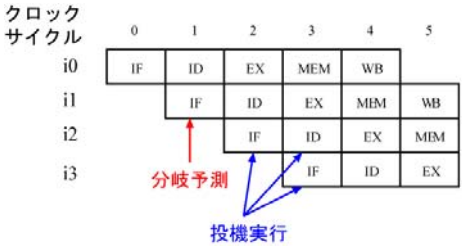


図 1･2　分岐によるパイプラインのストール

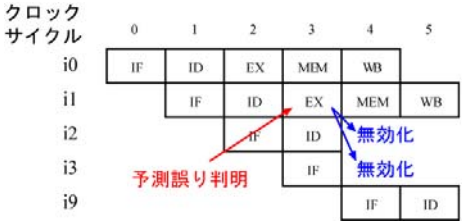
このストールをなくすため，一般的に採られている方策が制御投機である．制御投機とは，分岐命令の実行結果が判明する前に，それを予測し，それに基づいて分岐に続く命令の実行を開始することをいう．分岐の実行結果を予測することを分岐予測と呼ぶ．予測が正しければ，パイプラインは分岐がなかったかのごとく流れ，ストールを除去することができる．誤っていれば，予測に基づいてパイプラインに投入された命令を無効化し，予測とは反対の方向の命令を改めてパイプラインに投入する．この場合，制御投機を行わなかった場合と同様のパイプラインストールを実質的に生じることとなり，投機の利益は得られない．

図 1･1 のコード例で，制御投機を行った場合の実行タイミングを**図 1･3** に示す．この例では，分岐予測は，分岐 `i1` の結果をフェッチするタイミング（クロックサイクル 1）で行い，

不成立と予測している．予測に基づき，次のサイクルには，分岐不成立側の命令 i2 を，続いて命令 i3 もパイプラインに投入されている．



(a) 投機成功の場合



(b) 投機失敗の場合

図 1・3　制御投機実行

図 1･3(a) は，クロックサイクル 3 で分岐命令の結果（不成立）が判明し，予測が正しかったことがわかった場合の命令実行タイミングである．この場合，命令 i2, i3 の投機は成功し，ストールは生じない．

一方，図(b) は投機失敗の場合を示している．図(a) の場合と異なり，今度は分岐結果が成立で，予測誤りを起こしたとする．この場合，投機も失敗であり，命令 i2, i3 は無効化される．そして，分岐成立側の命令 i9 が改めてパイプラインに投入される．図からわかるように，分岐命令により 2 サイクルほどパイプラインがストールしたことと同じ結果になる．この実質的ストールのサイクル数を分岐予測ミスペナルティと呼ぶ．

図 1･3 では，インオーダ実行の例を示したが，アウトオブオーダ実行では，分岐命令 i1 が実行される前に，後続するより多くの命令が実行される可能性がある．また，アウトオブオーダ実行のプロセッサはインオーダ実行のプロセッサに比べ論理が複雑なので，クロック速度を高く保つために，一般にパイプラインは深くなっている．このため，分岐予測ミスペナルティも大きい．以上のことから，近年のアウトオブオーダ実行のスーパースカラプロセッサでは，分岐予測誤りは性能を大きく低下させる要因となっている．

1-1-2 分岐予測

前節で述べたとおり，制御投機は分岐予測に基づき行われるので，投機が成功し，いかにパイプラインストールを減少させることができるかは，分岐予測の精度に依存している．本節では，分岐予測について説明する．

分岐予測には，分岐方向の予測と，分岐成立の場合の分岐先予測がある．最初に分岐方向予測方式を，次に分岐先予測方式について述べる．

(1) 分岐方向の予測

分岐方向の予測方式（以下，短く分岐予測方式と呼ぶ）には種々のものがあるが，静的分岐予測と動的分岐予測に大別される．静的分岐予測とは，分岐一つひとつについて実行時に予測が変わることがない予測方式である．これに対して，動的分岐予測は変わり得る予測方式である．

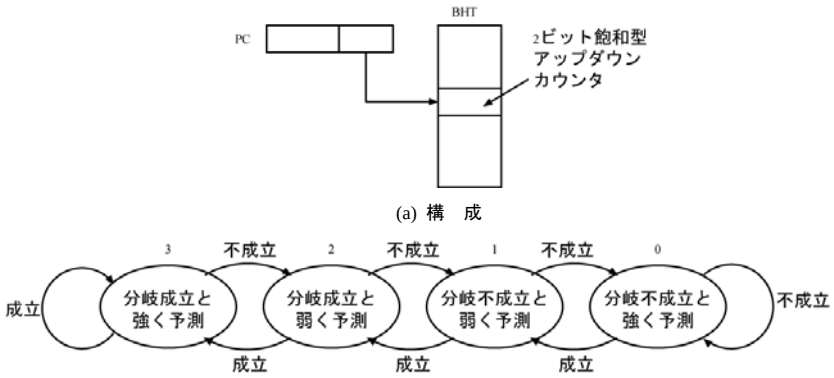
静的分岐予測方式には，例えば，次のものがある．

1. すべての分岐について，不成立と予測する．
2. 前方分岐は不成立と予測し，後方分岐は成立と予測する．
3. プログラムを実際に試し実行し，各分岐について，成立／不成立のどちらに偏っているかの統計を採取する．成立に偏っている分岐は成立と予測し，不成立に偏っている分岐は不成立と予測する．

2 番目の方法で，前方分岐とは，分岐成立時の分岐先が分岐よりプログラム上で前方にある分岐のことで，後方分岐とは後方にある分岐のことをいう．前方分岐は if-then-else に対応していることが多く，後方分岐はループの制御分岐に対応していることが多い．ループは一般に繰り返し回数が多いから，後方分岐を成立と予測するものである．

3 番目の方法は，プロファイリングと呼ばれる．この方法による分岐予測は，プログラムの入力にかかわらず，分岐の振る舞い（成立か不成立か）がほぼ一定していることを利用するものである．

動的予測方式には非常に多くの方式が提案されているが，どの方式も，過去の振る舞いから将来の振る舞いを予測するものである．



(a) 構　成

(b) 状態遷移

図 1・4　2 ビットカウンタ分岐予測方式

動的分岐予測の中でも基本的なものとして，2 ビットカウンタ分岐予測方式 [1] がある．これは，過去不成立が多く生じた分岐は，将来も不成立を生じ，その逆もいえるという性質を

利用するものである．予測器の構成は，**図 1･4**(a) に示すように，分岐命令の PC でインデクスされ，各エントリに 2 ビットの飽和型アップ／ダウンカウンタをもつ分岐履歴表（BHT：Branch Hitory Table）をもつものである．カウンタの状態遷移を図 1･4(b) に示す．分岐実行の結果，成立ならば，BHT の対応するエントリのカウンタを 1 増加させ，不成立ならば 1 減少させる．カウンタ値が 3, 0 のときに，それぞれ，分岐成立，不成立ならその値を維持する．このようにして，分岐の過去の振る舞いをとらえる．

予測は次のように行う．予測を行おうとしている分岐の PC で BHT を参照し，対応するエントリを参照する．得られたカウンタ値が 1 以下の場合，不成立と予測し，2 以上の場合，成立と予測する．

2 ビットカウンタ分岐予測方式より高い予測精度を達成する方式として， 2 レベル適応型分岐予測方式 [3] がある．この方式は，分岐が同一のパターンを繰り返すことを利用するものや，ある分岐の振る舞いが他の分岐の振る舞いに相関があることを利用するものである．前者をローカル履歴 2 レベル適応型分岐予測方式と呼び，後者をグローバル履歴 2 レベル適応型分岐予測方式と呼ぶ．

(2) 分岐先の予測

分岐先予測は，通常，BTB（Branch Target Buffer）で行う．BTB は，分岐命令の PC をインデクスとし，各エントリが対応する分岐の分岐先を保持する表である．エントリにはこのほか，キャッシュのようにタグをもっている．

一度実行された分岐の分岐先を BTB に記憶しておき，次に同じ分岐が実行される場合，記憶した分岐先を予測とする．BTB による分岐先予測の精度は非常に高いが，これは，多くの分岐の分岐先は PC 相対であり，動的に変化することがないためである．

1-1-3 制御投機と例外処理の関係

一般に，投機を行わないプロセッサでは，命令の実行中に例外が生じると，すぐにその処理を行う．しかし，制御投機を行うプロセッサではそれは性能上よくない．なぜなら，投機実行された命令が本当に実行されるべきであったかどうかは，依存する分岐の予測の正誤によるからである．分岐予測が誤りであったにもかかわらず投機実行された命令の起こした例外を処理すると，その例外処理時間が無駄に消費されたこととなる．したがって，例外処理はそれを起こした命令が依存する分岐の予測が正しかったことを確認した後まで延期されなければならない．

この処理延期は通常，リオーダバッファ [2] によって行われる．実行中に例外が生じたら，例外処理は行わず，その命令が対応しているリオーダバッファのエントリに例外が生じたことを記録する．リオーダバッファから命令はプログラム順にコミットされるが，その際に例外の記録があれば，その時点ではじめて例外が処理される．もし，例外を起こした命令がコミットされる前に依存する分岐の予測誤りがわかれば，その分岐より後方の命令は無効化されるので，その例外記録も消滅し，処理は行われない．

投機実行やリオーダバッファの詳細については，例えば文献 4) を参照されたい．

■参考文献

1) J. K. F. Lee and A. J. Smith, “Branch prediction strategies and branch target buffer design,” IEEE Computer, vol.17, no.1, pp.6-22, Jan. 1984.
2) J. E. Smith and A. R. Pleszkun, “Implementation of precise interrupts in pipelined processors,” In Proc. 12th Int. Symp. on Computer Architecture, pp.36-44, Jun. 1985.
3) T-Y. Yeh and Y. Patt, “Two-level adaptive branch prediction,” In Proc. 24th Int. Symp. and Workshop on Microarchitecture, pp.55-61, Nov. 1991.
4) 安藤秀樹, “命令レベル並列処理－プロセッサアーキテクチャとコンパイラー,” コロナ社, 2005.

■6 群 － 5 編 － 1 章

1-2 データ投機方式

（執筆者：佐藤寿倫）［2008 年 11 月 受領］

命令レベルの高い並列性を抽出するためには，投機実行は必要不可欠な技術である．現在のマイクロプロセッサは，ほぼ例外なく分岐予測に基づいた投機実行を行っている（5 編 1 章 2-1 節参照）．それは命令間の制御依存関係を投機的に解消する方式であるが，もう一つの依存関係であるデータ依存を投機的に解消する方式がある．それがデータ投機方式であり，2000 年前後に積極的に研究された．

データ依存は真の依存，逆依存，出力依存に分類される．逆依存と出力依存はレジスタリネーミングにより解消可能である．データ投機方式が対象としているのは真の依存である．名前からわかるとおり，真の依存関係はいかなる方式を用いても解消不可能である，と従来は考えられていた．投機の考え方を導入することで，その解消が可能になった．

データ投機実行は大きく二つに分類可能である．一つはレジスタを介するデータ依存に関わる方式であり，もう一つはメモリを介するデータ依存に関わる方式である．前者はレジスタの内容を予測する方式である．後者には，メモリの内容を予測する方式と，メモリを介する依存が存在するか否かを予測する方式とが含まれる．以下で，これら三つの方式を順に説明する．

1-2-1 レジスタデータ値予測方式

代表的な算術論理演算の入力オペランドまたは出力オペランドを予測する方式である．いずれの命令もその入力オペランドが揃っていなければ演算を開始することは不可能である．先行する命令とデータ依存の関係にある場合には，その完了を待って演算を開始することになる．しかし，もし未決定の入力オペランドを予測することができれば，それを用いて投機的に演算を開始することができる．一方，出力オペランドを予測する方式では，実際に演算結果が得られる前にその結果を予測する．したがって，後続の命令が，その予測値を利用して投機的に実行されることになる．以上からわかるように，入力オペランドの予測方式も出力オペランドの予測方式も本質的には同じである．以下では，出力オペランド，すなわち演算結果を予測する方式を説明する．

演算結果を予測するために，様々な方式が提案されてきた．最終値予測方式，ストライド予測方式，2 レベル予測方式，コンテキスト予測方式などである [1]．これらはデータを正しく予測することを第一の目的とした方式であるが，その後，消費電力やハードウェア規模を考慮した方式も多数提案されている [2]．

図 1･5 に示す最終値予測方式では，当該命令が前回実行されたときの演算結果を今回の演算結果の予測値として利用する．ストライド予測方式では，当該命令の過去 2 回分の結果から計算される差分（ストライド）を利用する．前回の実行結果にこの差分を加算し，今回の演算結果の予測値とする．2 レベル予測方式やコンテキスト予測方式では，過去複数回分の演算結果を保持しておき，その中から最もふさわしいと思われるものを選択して予測値とする．

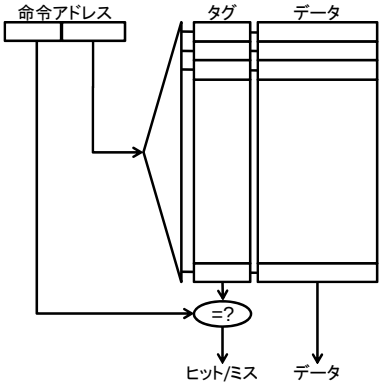


図 1・5　最終値予測器

いずれの方式でも，過去の演算結果などを記憶するために，キャッシュに似たハードウェアとして実現されるテーブルを用いる．デーブルのアクセスには命令アドレス（プログラムカウンタ）を用いる．より詳細な履歴を記憶する必要のある方式となるに従って，必要なハードウェア規模は増大し，また消費電力も増加する．現実的なハードウェアを仮定するとき，データ値予測によって獲得可能な性能向上は 10 %程度であることが多くの研究で示されている．近年の省電力指向のもとでは，10 %程度の性能向上では，必要とされるハードウェアや消費電力の増加を正当化できないと考えられる．また，予測に失敗した場合に無駄に消費される電力と，失敗から回復する際に要する電力も無視できない．予測に失敗した場合には，間違ったオペランドを用いて後続の命令が実行されているため，これらの命令実行は本来必要のない無駄な処理であるうえ，無駄に実行された命令が実行される前の状態にプロセッサの内部状態を戻さなくてはならない．以上の理由から，急激に研究が衰退した．

1-2-2 メモリデータ値予測方式

メモリデータ値予測方式は，メモリ（実際的にはデータキャッシュ）からリードされるデータ値を予測することで，出力オペランドを予測するレジスタデータ値予測方式と同様に，後続の命令が投機的に実行される．主記憶アクセスレイテンシが甚だしく大きくなってしまった現在，データキャッシュアクセスにミスヒットし，主記憶アクセスを生じる場合には，メモリデータ値の予測に基づく投機的実行は非常に有効であると期待される．

データ値の予測には，レジスタデータ値予測方式と同様の機構を利用可能である．ハードウェアとして実現されるテーブルに，過去メモリからリードされた値を保存しておけばよい．データキャッシュと異なるのは，キャッシュのアクセスにはデータアドレスが用いられるのに対し，データ値予測のためのテーブルにアクセスする際には命令アドレスが使用される点である．

ロード命令によりメモリからリードされるデータ値は，その獲得操作の違いからレジスタデータ値とは異なる方法で予測することができる．その方法には二つある．一つ目はデータそのものではなくアドレスを予測する方法であり，もう一つはデータ依存関係にあるストア

命令のもつ情報を利用する方法である．

ロード命令の操作は，データアドレスの計算とメモリアクセスに分割できる．予測されたアドレスを用いてメモリにアクセスすることで,本来よりも早期にデータを獲得可能である．配列アクセスなど，データ値そのものよりもデータアドレスにはより規則性が観察されるので，データ値予測に比べて予測精度が高いと期待される．無駄な投機や投機失敗からの回復に要する電力を削減できる．一方で，正しくアドレスを予測できてもキャッシュのミスヒットを生じるとデータ投機による恩恵は縮小する．メモリアクセスレイテンシと比べて，アドレス生成に要する演算レイテンシは無視できるほどに小さいからである．

ロード命令によりメモリからリードされるデータのほとんどは，先行するストア命令がメモリにライトしたものである．したがって，ストア命令がライトするデータをロード命令にバイパスできれば，メモリアクセスを省略可能である．依存関係にあるストア命令とロード命令を常に正しく獲得することに拘らなければ，非常に精度の高いメモリデータ値予測方式として利用できる．前述のアドレス予測に基づく方式とは異なり，キャッシュのミスヒットに関係なくメモリアクセスレイテンシを隠ぺいできる点で優れている．

レジスタデータ値予測と同様にメモリデータ値予測方式でも，予測失敗時には消費電力的に不利となる．ロード命令は実行レイテンシが長いため投機失敗中に実行される命令数が多くなりがちで，その意味で予測精度を高くすることが重要である．レジスタデータ値予測方式の研究が衰退した同じ理由から（本章 1-2-1 項参照），メモリデータ値予測方式の研究も近年は活気を失っている．

1-2-3 メモリ依存予測方式

メモリアクセス命令でデータ投機実行を行うためには，必ずしもリードされるデータがわからなくても構わない．ストア命令とロード命令との間，そして二つのストア命令間での曖昧なメモリ依存関係のために，命令レベルの並列性を抽出することが困難になる．この場合のデータ依存には，真の依存，逆依存，出力依存のすべてが含まれる．レジスタを介するデータ依存とは異なり，メモリを介するデータ依存ではデータアドレスが定まらない限りデータ依存が存在するか否かを判定できない．したがって，データアドレスがわかってはじめて，メモリアクセス命令の順序を入れ替えることが可能になる．逆に考えると，何らかの方法でメモリアクセス命令間の曖昧なデータ依存を解消できれば，より大きな命令レベル並列性を抽出することができる．データ依存の存在を予測し，もし無いと予測されると投機的に命令の実行順序を入れ替えるわけである．

メモリを介するデータ依存が存在するか否かを予測するには，メモリデータ値予測方式（本章 1-2-2 項参照）と同じ観察に基づく，アドレス予測方式 [3] とストア命令情報を利用する方式 [4] が用いられる．

近年はメモリ依存予測に依存するのではなく，ロードストアキューと命令スケジューリングの工夫により，曖昧なメモリ依存関係を解消して命令レベル並列性を抽出する研究が盛んである．

■参考文献

1) B. Calder and G. Reinman: “A comparative survey of load speculation architectures,” Journal of Instruction

Level Parallelism, **2,** May 2000.
2) 神代剛典, 佐藤寿倫, “低消費電力指向マルチスレッドプロセッサのための低コスト値予測機構の検討,” 情処論, vol.45, SIG 1(ACS 4), pp.43-53, Jan. 2004.
3) 佐藤寿倫, “命令再発行機構によるデータアドレス予測に基づく投機実行の効果改善,” 情処論, vol.40, no.5, pp.2093-2108, May 1999.
4) G. Z. Chrysos and J. S. Emer, “Memory dependence prediction using store sets,” ACM SIGARCH Computer Architecture News, vol.26, no.3, pp.142-153, Jun. 1998.

■6 群 － 5 編 － 1 章

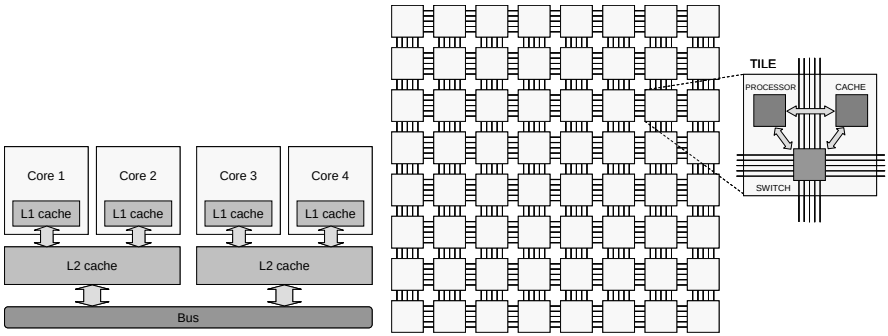
1-3 タイル型コンピュータ

（執筆者：吉瀬謙二）［2009 年 6 月 受領］

動作周波数を向上させることで達成してきたコンピュータの性能向上維持が困難となり，スレッドレベルの並列性を活用するプロセッサアーキテクチャが採用されはじめている．マルチコアプロセッサあるいはメニーコアプロセッサと呼ばれるアーキテクチャである．メニーコアプロセッサは，チップに多数のコアを集積してスレッドレベルの並列性を利用することで高性能かつ低消費電力を狙うが，その実現方法については幾つかのアプローチがある．タイル型コンピュータもその一つである．

タイルと呼ばれる小さいサイズの機能ブロックを規則的に敷きつめることで高速なプロセッサを構成する方式をタイルアーキテクチャと呼ぶ．それを採用するコンピュータがタイル型コンピュータである．

タイルアーキテクチャは，タイルとして実現される小さい機能ユニット（プロセッサコア）を多数集積するという構成から，バスやリングで接続されたマルチコアプロセッサと共通点がある．しかしながら，その設計思想は大きく異なっている．**図 1･6** に，Intel Core 2 Extreme QX 9650 プロセッサ（図(a)）とタイルアーキテクチャを採用する Tilera TILE 64 プロセッサ（図(b)）の構成を示す．ここでは，相違を明確にするために，メモリコントローラ及び I/O コントローラを省略して描いている．Intel Core 2 は 4 コア構成のプロセッサで，それぞれのコアは占有する L1 キャッシュをもつ．L2 キャッシュは 2 個のコアによる共有キャッシュとなっており，それらがバスを介して接続される．タイルアーキテクチャを採用する TILE 64 では，64 個のタイルが格子状に整然と敷き詰められている．また，それぞれのタイル（TILE）は同一の設計の機能ブロックを複製したもので，プロセッサコア（PROCESSOR），キャッシュ（CACHE），タイル間を接続する配線及びスイッチ（SWITCH）で構成される．



(a) Intel Core 2 Extreme　　(b) タイルアーキテクチャを採用する Tilera TILE 64 プロセッサ

図 1･6　Intel Core 2 Extreme とタイルアーキテクチャを採用する Tilera TILE 64 プロセッサ

Intel Core 2 に代表される汎用のチップマルチプロセッサでは，従来のプロセッサの設計を再利用しながら，キャッシュやネットワークの構成を最適化することで，高い性能を狙うものが多い．一方，タイルアーキテクチャでは，従来のプロセッサ構成にとらわれることなく

タイルの内部構成を工夫するものがある．ただし，チップ上に，同じ設計あるいは少ない種類のタイルを複製して配置することで設計の再利用性を高め，また，検証などの作業の軽減をねらう．タイルアーキテクチャでは，タイルの数を増やしたときに動作周波数が低下しないようにメッシュ接続に代表される近接接続のネットワークを採用することが多い．

1-3-1 MIT Raw プロセッサと Tilera TILE 64

マサチューセッツ工科大学（MIT）で開発された Raw プロセッサが，タイルアーキテクチャのさきがけである．豊富なハードウェア資源の活用，配線遅延の克服，限られたピンの有効利用を目指して，チップ試作とシステムレベルの評価が行われている．

Raw プロセッサは，16 個の同じ設計のタイルを敷き詰める構成をとる．それぞれのタイルは，MIPS プロセッサに近い単命令発行でインオーダ処理の計算パイプライン（プロセッサコア）をもつ．加えて，キャッシュ，コンパイル時にルートが決められるスタティックネットワーク，割込みやメインメモリを参照するためなどに利用される動的なネットワークにより構成される．それぞれのタイルは独自のプログラムカウンタをもつプロセッサとして動作し，命令やデータキャッシュにミスした場合にはチップの外に配置されるメインメモリからデータを取得する．

タイル間のデータの授受には必ずレジスタが介在し，すべての配線長はタイルの一辺の長さより短くなるように設計されている．このため，アプリケーションからの性能要求や，利用できるトランジスタ数の増加に応じてチップに集積するタイルの数を増やしたとしても，動作周波数が低下することはない．一方で，タイルを経由するたびに 1 サイクルの通信遅延が生じるため，例えば左上のタイルが生成したデータを右下のタイルが利用する場合には長い通信時間を必要とする．

一つのタイルが有するプロセッサコアは 8 ステージの命令パイプラインを採用する．個々の計算パイプラインは単命令発行の単純な構成であるため，一つのタイルではサイクル当たり高々1 命令しか処理することができない．しかしながら，16 個のタイルがすべて同時に計算を行うことで，チップとしてサイクル当たり 16 命令という高いピーク性能を達成できる．

タイル間の通信遅延を小さくするために，計算パイプラインのデータパスに通信のための機構が組み込まれている．具体的には，特定のレジスタが，通信の入出力バッファに割り当てられており，特別な命令を必要とすることなくタイル間のデータ授受を実現する．例えば，レジスタ 24 番からの読み出しは，通信バッファからの値の読み込み（データ受信）となる．また，レジスタ 24 番への書き込みは通信バッファへの書き込み，すなわち，他のタイルへのデータ送信となる．

Raw プロセッサの性能は文献 1) にまとめられている．Raw プロセッサに適したアプリケーションにおいては，それらの実行に適したスーパーコンピュータの SX-7 や，ビット演算に強い FPGA などに匹敵する性能を達成できることが示されている．

Raw プロセッサの開発で得られた成果は，図 1･6 に示した 64 個のタイルを搭載する Tilera TILE 64 プロセッサを製品化する際のベースとなっている．

1-3-2 TRIPS プロセッサ

テキサス大学で開発された挑戦的なタイルアーキテクチャのプロジェクトが TRIPS である．

TRIPS プロセッサでは，単一発行の単純な整数演算ユニット，浮動小数点演算ユニット，命令バッファ，オペランドバッファ，オペランドルータから構成される計算ノード（タイル）を格子状に配置する．そこに，ブロックと呼ばれるコンパイラが生成する TRIPS プロセッサの実行に適する複数の命令のかたまりを割り当て，必要とするデータが揃った命令から処理を開始させる．このようなマクロデータフロー方式の実行モデルを採用することが TRIPS プロセッサの特徴である．

計算ノードを格子状に配置して，近傍とのノード間のみでデータを受け渡すことにより配線遅延の問題を緩和する点は Raw プロセッサと同様である．一方で，同じ設計のタイルを敷き詰める Raw プロセッサとは異なり，TRIPS プロセッサは役割の異なる数種類のタイルを活用する．計算ノードとして動作するタイルを格子状に配置し，これを囲む形で，命令キャッシュ，データキャッシュ，レジスタファイルとしての機能を提供するタイルを配置する．このように，個々の計算ノードがキャッシュをもたない構成を採るため，TRIPS プロセッサでは，必要とする命令やデータをフェッチするために数サイクルの遅延が発生する．例えば，TRIPS プロセッサでベクトル加算を計算する例では，あるブロックのすべての命令をフェッチするだけでも 10 サイクル程度が必要となる．また，一つのブロックの処理が始まってから完了するまでに 80 サイクルの時間が必要となることがある．これらブロックを処理するための長い遅延を隠蔽するために，最大で 8 個までのブロックの処理をオーバラップできるように工夫されている．

計算ノードでは，自律的にデータの到着を検出して，計算に必要となるデータが揃った命令から実行を開始する．これにより，スーパースカラがもつ発行ウィンドウやデータフォワーディングといった高速化が困難となる回路を利用することなく，大規模なアウトオブオーダ実行と同様の仕組みを実現する．TRIPS プロセッサの性能は文献 2）にまとめられている．

1-3-3 その他のタイルアーキテクチャ

Intel は 80 個のコアを搭載するタイルアーキテクチャの研究チップ（Teraflops Research Chip）を試作している．このチップは，Tilera TILE 64 と同様に，スイッチを介してメッシュ状にタイルを接続する．

タイルアーキテクチャは汎用プロセッサ以外にも利用されている．例えば，タイルアーキテクチャを採用するキャッシュとして NUCA（Non-Uniform Cache Architecture）の研究開発 [3] が進められている．また，近年注目を集めている再構成可能プロセッサ（リコンフィギャラブルプロセッサ）の分野においても，タイルアーキテクチャが採用されることがある．

■参考文献

1) Michael Taylor et al., “Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams,” The 31st Annual International Symposium on Computer Architecture, pp.2-13, 2004.
2) Karthikeyan Sankaralingam et al., “Distributed Microarchitectural Protocols in the TRIPS Prototype Processor,” 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp.480-491, 2006.
3) Changkyu Kim et al., “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” 10th International Conference on Architectural Support For Programming Languages and Operating Systems, pp.211-222, 2002.