輪講資料
単位プロセッサの基礎

NGUYEN HAI HA
(グェン・ハイ・ハー)

発表日:平成９年５月２９日
提出日:平成９年７月１８日

1 プロセッサの基本構成

1.1 基本構成

プロセッサの基本構成は演算装置、制御装置と連絡バスからなる。プロセッサと主記憶や入出力装置をつながっているのがデータバス、アドレスバスと制御バス(外部バス)である。これらのバスを介して、制御信号、データ信号を送信たり受信したりする。

1.2 演算装置

通常１個のALUと１個のシフタからなる。ALUでは算術および論理演算がなされ、シフタではシフト演算がなされる。

計算の中間結果を保存するために高速な一時的記憶が用意され、これは汎用レジスタと呼ばれる。レジスタの個数は１６から６４までの程度で、一つのレジスタ・ファイルを構成する。

レジスタ・ファイル、ALUとシフタを連絡するのがデータバスで、バスの本数によって３バスと１バス構成の演算装置がある。３バス構成の演算装置では、入力が２本と出力１本のバスからなる。１バス構成の演算装置では、片方の入力バスの代わりにACC(累算器)を使用される。

1.3 制御装置

制御装置には演算装置制御部、主記憶制御部、入出力制御部などがある。各制御部は制御バスを介して、対応装置へ動作指定を発し、制御するのである。また割り込み処理ための専用回路も備えてある。

一般に制御信号は時間系列として生成され、制御を実現する回路は順序回路である。クロックによって信号が生成されると共に、回路は次状態に遷移する。初期状態は命令コードによる。

プロセッサの制御方式には、布線論理方式とマイクロプログラム制御方式がある。

- 布線論理方式:

  制御信号生成回路を論理ゲート、記憶素子から作る方式である。回路の簡単化などにより高速化できるが、一度実装した回路は修正や変更が困難である。この方式は、スーパーコンピュータやRISC型プロセッサに良く用いられる。

- マイクロプログラム制御方式:

  機械命令の各マイクロ操作に対応して、その操作をどこで、どのように実行するのかを指定するのがマイクロ命令である。マイクロプログラムはこれらのマイクロ命令の系列である。これらのマイクロ命令は制御記憶という特殊なメモリに格納される。

  複雑な命令セットをもつCISC型などのプロセッサには不可欠な制御方式とも言えよう。

  基本構成:

  マイクロ命令は制御信号指定部と順序制御指定部からなる。制御指定部はALUなどへ制御信号を発するに対して、順序制御指定部は次に実行されるマイクロ命

令のアドレスを決定する一つの要素である。次アドレスの決定する際、この順序指定部の他に機械命令コードや条件コード、割り込み、オーバフローなどといった入力変数がある。

マイクロ命令が格納される場所は制御記憶である。そこには、各機械命令に応じるμI の一連が連続領域に格納される。これらの領域は機械命令用のマイクロルーチンと呼ばれる。

動作 :

命令フェッチしデコードして得た命令コードに基づいて、制御は対応したマイクロルーチンに分岐する。マイクロルーチンの実行が終ったら命令フェッチマイクロルーチンに帰る。

2 高速化の手法

プロセッサの高速化を実現する手法として以下のように分類することができる。

2.1 物理的

これはハードウェア技術上の改善であり実質的である。

- 論理ゲートのスイッチング時間、記憶素子のアクセス時間の短縮 :

  論理ゲートやメモリ素子の設計を改善することによってスイッチング時間を短縮できる。例えば、蓄積時間をなくすために ECL (Emitter Couple Logic)、ショットキー TTL (Schottky TTL) の非飽和形回路を使用する。

  もう一つは、デバイスの改善による高速化手法がある。現在、ほとんど素子は単結晶のシリコンから作られる。単結晶シリコンには幾つかの利点はあるが、電子に対する易動度は高くない。それに対して、化合物を利用して化合物半導体の開発が行われ、デバイスの高速化が期待されている。例えば、GaAs (ガリウム・ヒ素) は電子に対する易動度は $4000cm^2/Vs$ で、単結晶シリコンの 6 倍である。

- 配線延長時間の短縮 :一つのチップにたくさんの素子を詰め込んで、配線遅延時間を短縮する。これが実現できたのは、パッケジー技術と冷却技術が格段に進歩したからである。

- キャッシュ・メモリ内蔵 :こうすることによって、演算などはプロセッサの内部クロックで実行できて、高速化できる。これが実現できるのも、やはりメモリとパッケージング技術が発達したからである。

2.2 論理的

これは処理のあり方、データ構造 (命令セット) の改善である。

2.2.1 並列処理による改善

実行が逐次的であったノイマン型のコンピュータを並列処理の導入によって大きくスピードアップできた。

- パイプライン処理 :

  命令を実行する時に機械は幾つかのステージにわけて実行する。各ステージをオーバラップさせて、複数の命令を同時に実行するのがパイプライン処理である。現在、これはプロセッサ高速化のための基本技術となっている。

- マルチプロセッサ :

  処理するジョブを分散して、複数のマイクロプロセッサで実行する方式を採用する。

  物理構成からみると、メモリを共有すると共有しない、即ち各プロセッサが専用のメモリをもつ、に分類できる。また、並列処理の形態を命令とデータの流れに注目すると、SIMD (Single Instruction Multiple Data stream) と MIMD (Multiple Instruction Multiple Data stream) に分類できる。SIMD では、各プロセッサが同一制御されるに対して、MIMD では各プロセッサに専用の制御部が割り当てられる。プロセッサ、共有記憶、全体制御部を連絡するのが通信ネットワークである。

  処理の効率を上げるためには、プロセッサにできるだけ均一な処理量を与えたり、プロセッ

サ間の通信オーバヘッドを最小限に押さえるなどが考えられる。

- 機械命令レベルにおける並列処理：

  同時に複数の命令を発行し、実行する機構である。これが実現できるのは、後述する分岐予測と投機実行技術やout-of-order技術などといったいろいろな技術を用い入れた成果である。しかし、これらはあくまでも実行スループットを向上させるだけで、命令の実行レイテンシを短縮するわけではない。

2.2.2 基本命令セットの改善

- 命令を単純化：

  ハードウェアを削減して動作周波数を上げるねらいである。また、並列制御が簡単化できるとともに、新しい処理ユニットを置けるようになる。

  しかし、一度命令セットを決めてしまったら、そう簡単に大きく変えられない。なぜなら、その命令セットで稼働するソフトウェアが大量に作られているからである。

- アプリケーション向けの命令の追加：

  代表的なのはMMX（Multi Media eXtension）命令セットである。この命令セットが追加されたことによって、画像や音声などといったマルチ・メディア・データは高速に処理できる。

3 並列処理

3.1 パイプライン処理

3.1.1 特徴

1. 命令ステージをオーバーラップさせ、複数の命令を（見掛け上）並列に実行するので、理想の場合はステージ数分だけ全体の実行が高速になる。

2. 命令実行そのものは速くならない（実際にパイプライン制御により若干実行時間が延びるのが普通）。しかし、命令ステージをオーバーラップさせたことで、プロセッサの命令スループット（単位時間当りの命令実行数）が向上する。従って、ジョブ全体の実行時間が短縮できる。

3. パイプライン段数を大きくした際に得られるクロック周波数に対して、制約がある。重要なのはラッチ遅延とクロック・スキューの複合効果である。

   パイプライン処理において、ステージ間にどうしても一時的記憶（パイプライン・レジスタと呼ばれる）を置かなければならない。そのため、ラッチ素子の遅延が無視できないのである。また、ラッチ素子の分布がクロック源からみると均等ではないから、クロック・スキューが避けられない。これらの時間の和はクロック・サイクルの下限のひとつである。

4. パイプライン処理効果は命令セットに依存する。命令長可変や命令の複雑さによる実行時間相違などがパイプライン・ステージ間のバランスを壊してしまう原因である。その結果他のステージが停滞し性能が低下ことがある。

3.1.2 パイプライン化にあたっての難問:パイプライン・ハザート

理想パイプラインでは、毎クロック・サイクルですべてのステージが実行されるので、パイプライン中にある命令は互いに独立である。しかし、実際では、命令間に依存関係があるので、なかなかうまくいかない。

依存関係は3つの種類があり、それぞれデータ依存（データ・ハザード）、制御依存（手続き依存、分岐ハザード）と競合依存（構造ハザードともいう）である。

データ依存：

先行命令の演算結果を後続命令に使用されることを意味する。回避方法として、次のようなことが挙げられる。

- バブル挿入やレジスタのリネーミングによる方法：

  先行と後続命令間にnop命令を入れる方法と、レジスタが複数ある場合はレジスタをリネーミングして、競合を回避する。

- バイパス法 forwarding, short-circuiting 法ともいう。先行命令演算結果などが後続命令で使用できるように前ステージのパイプライン・レジスタにバイパスする。
- 命令スケジューリング：

  依存関係のある命令間に存在する遅延スロットを無関係な命令で埋め込む。すなわち、あからじめ命令を並び替える手法である。

  この手法はデータ依存だけでなく、後述の制御、競合依存を消滅するためによく使われる手法である。(詳細は後述)

制御依存：

先行命令で後続命令を実行するかしないかの決定がなされる場合、あるいは実行環境を変更する場合には、先行命令の結果が判明するまで後続命令を待たされることを意味する。種類として、分岐命令を伴う依存、割り込み発生、命令の書換え、制御レジスタの変更によるものが挙げられる。中でも、分岐命令を伴う依存がパイプラインの性能を一番に低下させるのであろう。なぜなら、その出現頻度が他と比べて大きい(条件分岐では１１から１７％、無条件分岐では２から８％というデータがある)からである。

解決方法として、以下が挙げられる。

- 分岐予測方式：

  静的予測と動的予測の２種類がある。

  静的予測では、続行命令系列が分岐条件不成立時の系列(predict-not-taken 方式採用)か成立時の系列(predict-taken 方式採用)かでふたつ分れている。だいたいのマシンでは条件結果の判明が分岐先アドレス決定の前になされるので、分岐不成立方式を採用している。それに対して、高機能な分岐条件を用いた(即ち、分岐結果の判明が遅い)マシンでは分岐成立方式を採用している。

  動的予測では、過去の分岐履歴情報で分岐先を予測する方式である。この方式が広く用いられているのはループ制御、エラーチェック、プログラムなど分岐には偏りが存在し、当たる確率が高いからである。
- 分岐命令を条件判定操作と分岐操作に分離する方式：

  こうすることによって、条件判定操作と分岐操作の間に命令が埋め込め、高速化が図れる。
- 遅延分岐方式：

  分岐先が判明までの間(分岐遅延スロットと呼ぶ)に無関係な命令をパイプラインに投入することによって処理が高速化できる。
- 複数命令流方式：

  パイプラインを２重化し、条件不成立と成立したときの命令系列を同時に実行する方式である。完全に２重化するのがハードウェア量の増加(それを伴う動作周波数低下)を招くので、実際に、一方をパイプラインに投入し、他方をキャッシュにおいて置くことが普通である。

競合依存：

パイプライン処理では、同じクロック・サイクルで異なる命令が同じハードウェア資源を要求するときがある。資源競合によりある命令組み合わせがオーバラップ実行できない場合、競合依存があるという。

完全パイプライン化にあたっては、機能ユニットのパイプライン化、および任意の命令組み合わせのオーバラップ実行を可能にするために資源の多重化が必要である。しかし、実際にこのやり方はあまり使用されていない。理由が２つある。一つはコストがかかりすぎる。二つ目は機能ユニットを完全パイプライン化よりもそのレイテンシの方を選んだ方が効率がよいということである。

3.1.3 パイプライン実現上の課題

割り込み：

パイプライン化においては、割り込みが最大の障害物だと言っても過言ではない。割込み処理にはふたつのタイプがある。

- precise interrupt （正確な割込み）:

  割込み処理後、実行中のプロセスが正確に再開できるような割込み。

- imprecise interrupt （不正確な割込み）:

  文字通り、それ以外のもの。

高度パイプラインをもつプロセッサでは正確な割込みが要求される。以下で正確な割込みを実現する際に考慮しなければならない点を示す。

1. 割り込まれた命令の検出 :

   パイプラインでは、ステージをオーバーラップさせ実行するので、割り込まれた命令は簡単には判明できない。それは、異なるステージで同じクロック・サイクルで同一の割込みが起こり得るからである。例えば、ロード／ストア命令で、IF（命令フェッチ）とMEM（メモリ・アクセス）がページ・フォールト、メモリ・アクセスのミスアラインメントかメモリ保護違反を起こす可能性がある。

   割り込まれた命令を特定できなければ前後の命令の処理をすべて停止するしかない。

2. マシン状態の保存 :

   割込みが発生した時にそのマシン状態を保存しなければならない。問題なのは割込みされた命令が、割込みが発生する前に、マシン状態を変更してしまうことである。これは命令の設計段階で決められてしまって、ヒストリのレジスタなどを追加することで解決できる。また、バイパス技術を使う場合はマシン状態を厳重に管理しなければならない。なぜなら、バイパスでは、データ選択枝がいつの間に変わってしまう可能性があるからだ。

3. 多重割込み処理 :

   割込みの発生する時点から実際に割込み処理が行われるまでの間にまた別のところで割込みが起き得る。これは、interrupt status vectorの導入で解決可能である。

4. out-of-order 実行性能への影響 :

   先行命令の割込みステージより続行命令が先に終了してしまって、しかもマシン状態を変更する可能性がある。解決方法として、マシン状態が変更できるようなステージは、必ず命令の順序通りに実行するなどがある。

命令セット :

多くの命令セットでは、演算の結果で暗黙的に条件コードやフラグがセットされる。だいたいの命令セットは、パイプラインの分岐遅延を短縮することや命令を節約する（条件照合命令を共通に）のために条件照合命令と実際に分岐する命令を区別する。条件分岐命令は条件照合命令でセットした条件コードをチェックするだけで分岐先がわかるなどという利点がある。

しかし、パイプラインでは、スケジューリングによって分岐遅延スロットに埋められた命令がこの条件コードを変更してしまうことがあるから、これを防止するために余分のハードウェアを追加しなければならない。

マルチサイクル実行 :

命令実行のなかで、あるステージが余計に長くなるとパイプラインの性能が低下してしまう。初期のCISC型プロセッサでは、デバイス技術がまだ発展していなかったためIFステージに時間がかなりかかった。それをカバーするよう、命令の機能を増やしてバランスを図っていた。

その後、デバイス技術が画期的に進歩して、IFを1クロック・サイクルで実現できるようになった。焦点は演算ステージに移っていった。演算ステージでは、短いものは1サイクルで済むが、長いものは数サイクル程度（命令セットによって異なり、極端の場合は数十、数百までのぼる）を要する。

パイプライン性能の低減を避けるために次のような対策が考えられる。

- 機能分割 :

演算器を機能ユニット（例えば、整数演算ユニット、浮動点数演算ユニットなど）に分ける。すると、正数演算の場合は、専用の演算ユニットがあるから1サイクルで演算を済ませて、パイプラインのバブルをなくすことができる。

- 演算器をパイプライン化：

  実現にはデータの供給バスや制御回路など大量のハードウェアが増設され、その分クロックが遅くなる。レイテンシを重視するアーキテクチャでは、この方式を採用しない。

- 命令スケジューリング

  命令スケジューリングで、遅延スロットに無関係な命令を埋める。

3.2 機械命令レベルにおける並列処理

3.2.1 諸制約

パイプパインと同様に機械命令レベル並列処理において3つの制約がある。それらは、データ依存、制御依存、資源競合である。

資源依存の解消方法として、機能ユニットを多重するか、パイプライン化する方法が挙げられる。それぞれの特徴、課題は前に述べた。

データ依存には、さらに3つの種類がある。それらは、

- フロー依存（RAW=Read-After-Write hazard）：

  先行命令がオペランドを書き換える前に後続の命令がそのオペランドをフェッチしてはいけない。

- 逆依存（WAR hazard）：

  先行命令がオペランドを読む前に後続の命令がそのオペランドを書き換えてはならない。

- 出力依存（WAW hazard）：

  後続命令があるレジスタに計算結果を書き込んだあと、先行の命令がそのレジスタを書き換えてはいけない。

これらの依存は、命令スケジューリングを行うことによって解消できるか、並列処理性能への影響を最小限に押さえることができる。

制御依存では、分岐命令の結果がわかるまでに分岐先（take/not taken）の命令系列を原則的には実行できなく、命令の並列度をさげる。しかし、分岐予測（by software/hardware）と投機実行の導入でこの問題点を解決している。

3.2.2 投機的実行

投機的実行は、分岐予測によって予測された分岐先命令を分岐条件が判明するまえにその命令（系列）を実行してしまう方式である。そのため、以下の2つの問題を持っている。

- プログラムの意味の維持：

  投機的実行が生きているデータを破壊する場合に生じる。即ち、分岐を越えて命令移動を行う場合、その命令の書き込みが、他の分岐方向で使用されるデータを破壊しないようにしなければならない。

- 投機命令の起こした例外（割込み）の処理：

  割込みが発生したときに、強制終了すべきか、割込み処理に分岐すべきかは分岐条件が判明したあとで決定してよい。

4 命令スケジューリング

4.1 静的スケジューリング

あからじめコンパイラが命令を並替えて、ハードウェアはそのスケジューリングに従って単純に実行する方式である。

スケジューリングにおいては、コンパイラは、まず与えられたウィンドウの中の命令の依存関係を調べる。それから、依存制約と資源制約を満足し、もっとも実行サイクル数が短くなるために、その命令順序を決定する。

4.1.1 特徴

1. ハードウェア量を削減できる。

2. マシンのアーキテクチャに強く依存する。そのため、汎用性が低い。実際、静的スケジューリングはコンパイラの一つのモジュールとして設計されている。

3. スケジューリングの限界がある。なぜなら、実行時にしか対処できないことが存在するからである。

4. 命令の投機的移動が弱い。場合によって、命令の投機的移動は逆効果をもたらすこともある。

静的スケジューリングには局所的スケジューリングと広域的スケジューリングがある。

4.1.2 局所的スケジューリング

基本ブロック内でのみ命令移動を許す方式で、ブロック間の命令移動は行わない。基本ブロックとは、一連の連続した命令ブロックで、実行時に必ずその順序で実行される。別の言い方で、そのブロックの出口を除いてすべての命令は分岐命令ではなく、また入り口の命令のみは外から分岐されてきた命令である。

この方法の所長は、コードのサイズを増やさないことである。しかし、基本ブロックは、通常小さい（分岐命令の出現確率は２０％前後からだ）のであまり効果をもたらさない。

スケジューリング手順は以下に示す。

- 制御グラフの作成：

  基本ブロック内に分岐命令が存在しないので、分岐依存はない。しかし、データ依存と資源競合はまだ残っている。データの制約を解析するために制御グラフが用いられる。

  このグラフでは、親と子を結ぶエッジはデータのフロー依存をあらわす。

- プライオリティ・セット（クリティカル・パス決定）：

  各ノードに実行優先順位をつける。クリティカル・パスとは、グラフ上において、仮ルート（親のないノードの意味で、トリー構造のルートと少し違う）からそこに到達するのにもっとも時間のかかるパスである。クリティカル・パスの長さに基づいて、順位を決定する。

- リスト・スケジューリング：

  目的は、制御グラフに矛盾しなく、かつ並列命令を抽出し出力することである。抽出順序によって、以下の２通りに分れる。

  - 順順序：

    候補集合（初期は仮ルートの集合である）から、並列実行可能な命令群を優先度に基づいて抽出し出力する。抽出された仮ルートを候補集合および制御グラフから削除する。新しいグラフから、その時点ですべての親が削除されていたノードを候補集合に追加する。これを候補集合は空になるまでに繰り返す。（詳しくは、[13] を参照）

  - 逆順序（タイム・スロットを利用）:

    アルゴリズム自体は順順序とほぼ同じが、抽出方向は終端ノードから仮ルートへである。抽出された命令群を一時にタイム・スロットに入れて、出力時にタイム・スロットの逆順に行う方法である。（詳しくは、[1] を参照）

4.1.3 広域スケジューリング

基本ブロック間の命令移動を行って、スケジューリング効果を向上させる方式である。一般のプログラムに対応する方式と特定のプログラム構造に対応する方式の２通りがる。

トレース・スケジューリング：

概要：

この方式は、一般の科学技術の計算プログラムのような、分岐に非常に偏りのあるプログラムに適用される。

手順は以下に示す。

- 広域制御グラフ（基本ブロックを単位としたグラフ）において、実行時に一番高い確率で通るトレースをサンプル・プログラムで抽出する。
- 抽出されたトレース内でリスト・スケジューリングを行う。また、プログラムの

意味を維持するために、トレース外の修正を行う。

- この手順を繰り返すと、高度なスケジュールが得られるのである。詳しくは、[1] または [13] を参照）

特徴：

- コードの修正が必要なので、コードのサイズが大きくなる。
- ブロック間の命令移動を許すので、かなりよい効果をもたらす。

パーコレーション・スケジューリング：

概要：

パーコレーション・スケジューリングでは、制御グラフにおいて、できるだけコードを下位のブロックから上位のブロックへ移動させる。

移動は以下の４つのルールに従って行われる。

- 削除（Delete Transformation）命令の移動で空になった基本ブロックや制御グラフにおいて最上位のブロックを除いてそこへ分岐のない基本ブロックを削除する。
- 一般命令移動（Move-op Trans.）：分岐命令でない一般命令を、そのブロック内の前方の命令に依存関係が存在しなければ、つながっているパスを辿って、上位ブロックに移動させる。プログラムの意味を維持するために、コードの修正を行う。（イメージは水道のメカニズムと同じ）
- 分岐命令移動（Move-cj Trans.）:分岐命令を複製して、上位ブロックへ移動させる。プログラムのセマンティックを維持するためにコードの修正が必要である。
- 単一化（Unification Trans.）：ある分岐ブロックの後方ブロックに同一命令があり、かつ、その分岐ブロックに移動可能な時は、後方からそれらの命令を持ってきて、同一化する。このように、単一化の目的はコピーをできるだけ抑えることである。

特徴：

- トレース・スケジューリングと同様に、コードのコピーがなされるので、コードのサイズが大きくなる。しかし、この方式では、コードの単一化が行われるため、トレース・スケジューリングと比べてサイズが小さい。
- 下位ブロックの命令を積極的に上位ブロックに移動させる、即ち、オペランドの共有する命令が近くに配置されるようになるので、データ・フロー型のマシンに大きな効果をもたらす。また、実行時の通信と同期化のコストが削減できる。

ループ・アンローリング：

概要：

ループ構造に適用される方式である。通常のループの何個分を展開して、新しいループを生成するのである。実際に、２か３個分のループを展開の対象になっている。それ以上個数をあげる価値がないと言われている。

プログラムの意味を維持するため、制御回路が増設され、元の１反復の実行スループットはすこし長くなるが、全体のスループットは大幅に短縮できる。

特徴：

- VLIW 型プロセッサでは、命令が長くて、ビットの使用効率が高くないと批判されているが、この方式を利用すると、ビットの使用効率を改良することができる。即ち、多重の機能ユニットが効率よく使用できるのである。
- ループを展開したことで、制御変数の変更回数、アドレス計算の回数を減らすことができる。しかし、その反面、プログラムの意味を維持するのに資源（レジスタ）を増設しなければならない。
- ループの回数が動的に決定される場合は、補正の必要がある。また、コードのサイズが大きくなる。

ソフトウェア・パイプライン：

概要：

ループ・アンローリングのように、ソフトウェア・パイプラインもループ構造に大いに適用される方式である。

この方式では、命令を仮想パイプライン・ステージと見て、その入力を反復番号とする。

特徴：

- 反復間の依存（ループ運搬依存）がなければ効果が最大となる。この依存があるとき、投入する間隔を拡げる。
- 効果を引き出すためにレジスタのリネーミングを積極的にする。レジスタの数をたくさん用意しなければならない。

4.2 動的スケジューリング

目的は静的スケジューリングと同じだが、スケジューラはハードウェアである。

4.2.1 特徴

1. コンパイル時に不明な依存関係に対して対処可能であり、コンパイラを簡単にする。また、ある特定のパイプライン向きにコンパイルされたオブジェクト・コードを別のパイプラインでも効率よく実行できる。
2. 動的スケジューリングでは、キャッシュ・ミスヒットのときなどはそのペナルティによる性能低下を防ぐことができる。
3. 制御用の専用回路を設けるので、ハードウェア量が大きく増加してしまう。

4.2.2 幾つかの方式

スコアボーディング：

概要：

スコアボードという制御ネットワークを用いて命令の発行、実行を管理する方式。すべての命令はこのスコアボードを通る。そこで、データ依存関係の解析がなされれ、命令を発行するかどうかを判断する。

特徴：

- レジスタファイルと機能ユニットの間に、複数のバスを介してオペランド、結果を転送する。結果は必ずレジスタに格納され、もし後続の命令が使うならまたそこから読み出される。
- ハザード・チェックとオペランド読みのふたつのパイプライン・ピッチを命令デコードの中に隠す。
- スコアボーディングは、WAW、WAR 依存を検出したときに、その命令と後続の命令の命令発行を停止させてしまう。RAW 依存は動的に処理し、命令を out-of-order で発行可能である。

Tomasulo 方式：

概要：

各機能ユニットの前にリザベーション・ステーションを設け、中に機能ユニットの実行状況（空いているかどうか）とオペランドに関する情報が含まれる。また、Common Results Bus（CRB）で計算結果をバイパスすることによってオペランド待ち時間を短縮できる。

特徴：

- リザベーション・ステーションで、依存関係で待ち合わせている命令は、その数が複数であっても、実行結果が放送されるために同時に実行できる。即ち、「可能限り、早期に実行を開始する」という目的に達している。
- リザベーション・ステーションに命令を登録される際、利用できるオペランドはリザベーション・ステーションにコピーされるので、WAR ハザード（逆依存）が解消される。
- 依存データは、レジスタ番号ではなくタグを用いて識別されるため、WAW ハザード（出力依存）が生じない。
- タグ付のデータ放送の方式を採用しているので、リザベーション・ステーションが重く、高速化の限界をみずから作ってしまう。

- リザベーション・ステーションにおけるオーバヘッドをパイプライン・ピッチとしてデコード・ステージに隠す手法はあるが、分岐予測ミスに伴うペナルティが大きい。

5 幾つかのアーキテクチャの紹介

5.1 スーパスカラ

5.1.1 概要

パイプラインの副を拡げて、1クロック・サイクルで複数の命令を同時に実行する方式である。機能ユニットが空いていれば、動的に命令を投入し実行する。

5.1.2 特徴

- スーパスカラでは制御ネットワークが複雑なので、クロックの周波数を上げにくい。
- 以前の命令セット(x86)のアーキテクチャの変更は一切しない。そのため、オブジェクト・コードレベルで完全な互換性がある。
- 動的に積極的にスケジューリングするので、キャッシュ・ミスヒットと伴うメモリ・アクセス・レイテンシをうまく隠蔽できる。

5.1.3 実現上の課題

実際に、命令間に依存関係が存在するから理想な形にもつことができない。できるだけその形にもっていくためには、

- 並列にパイプラインが動き得るように、十分な数の機能ユニットを用意しなくてはならない。
- データ依存を伴う競合/ハザードが避けられるようなプログラムである必要がある。
- 制御ハザードが仮にあったとしても、大きなストールが生じないようなハードウェア構成上の工夫が必要である。
- 正確割込みの対応はどうすればよいのか
- レジスタ群は、同時に複数の命令が読み・書きできるためにマルチ・ポットのレジスタとなっている必要がある。
- MS(メイン・メモリ)がネックとならないように、十分なスループットを用意しなければいけないし、またキャッシュの強化も逃せない。

5.2 VLIW

5.2.1 概要

目的はスーパスカラ方式と同様、VLIWは1クロック・サイクルで複数の命令を同時に実行させる方式である。しかし、スーパスカラでは並列実行可能な命令群の発行をハードウェアで実行するのに対して、VLIWではコンパイラにまかせる。

5.2.2 特徴

- VLIWコンパイラでは、並列実行可能な命令ブロックを水平に並べて、1つの長い命令にフォーマットし、ハードウェアはその長い命令を実行する方式である。これは、水平型マイクロ命令と似ている。
- 並列命令の検出はソフトウェアに任せたので、ハードウェア量を削減できクロック周波数を改良することができる。しかし、その反面では、命令セットのアーキテクチャが変わったので、再コンパイルしなくては既存プログラムが動かないのである。実行効率はコンパイラに強く依存する。
- 静的スケジューリングなので、実行時における割込みの対処が弱い。たとえば、キャッシュ・ミスヒットの時に命令パイプラインを完全に停止させてしまう。

5.2.3 実現上の課題

- 制限 制約:依存関係の解消をコンパイラに完全に任せきれない。なぜなら、実行時にしか対処できないことが存在するからである。その例は以下に示す。
  1. bank conflict:命令が長いので、キャッシュやメモリのアクセス時にバンク衝突

    が起こりやすい。従って、平均アクセス・サイクル数が大きくなる。

  2. 分岐のtaken/not taken 結果が判明するまでに分岐先がわからない。たとえば、データによりアドレスが決まる場合など。
  3. NOPの命令が多大である。

- 処理結果が保証されない危険性：

  上記のような場合で、VLIWプログラムは制限・制約に違反したら、処理結果が保証されないか、またはハードウェアがこれを検出して、プログラム・エラーの割込みを起こすことになる。

  処理結果を保証するためには、依存関係を動的に検出するハードウェアを設け、割込みで報告する手段を採るか、または、コンパイラができる範囲で最適化して、それ以上はnopを使用し、命令をフォーマットする。しかし、後者では、nopの率が多く、効率の悪いプログラムにならざるを得ないことになる。

5.3 スーパパイプライン

5.3.1 概要

パイプラインの段数を増やし、全体処理のスループットの向上を狙った方式である。制約がなければ、パイプ・ピッチを小さくして、パイプラインの深さをおおきくすれば実現できる。

5.3.2 特徴

- パイプラインの詳細化のため、制御ネットワークが複雑で、クロックの周波数を上げにくい。
- 分岐予測のミスのときなどはそのペナルティが大きい。

5.3.3 実現上の課題

- パイプラインの深さが大きいと、データ・ハザードによるストールや分岐予測がはずれたときのペナルティが大きくなる。
- スーパスカラ方式と同様な課題をもつ。とくに、データ供給の問題でキャッシュメモリへのアクセスもパイプライン化する必要がある。

5.4 マルチスレッディング

5.4.1 概要

全く独立な複数個のプログラム・ストリームを用意して、これらを並行して実行させる方式である。

クロックごとにストリームを切換えて、命令をパイプラインに投入する方法で、実行を進めていく。

5.4.2 特徴

- この方式で、パイプライン中にある命令の間に関係がないので、パイプラインの効率はほぼ１００％に達する。
- 命令ストリームが複数あるので、キャッシュ・ヒット率が低下してしまう。

5.4.3 実現上の課題

- 資源（レジスタ群、機能ユニット、キャッシュ容量など）の提供が十分でないといけない。
- ストリームが複数あるという前提はいつでも成立するわけではない。また、ストリームの長さ（実行時間）が異なったときにどのように効率に対応するかが問題点である。

5.4.4 幾つかの提案

- CPC（cyclic pipeline）方式：

  スーパパイプライン的な短ピッチのパイプラインで、パイプライ化されたMIMD（multi-instruction multi-data stream）を実現しようというもので、キャッシュメモリなどもパイプライン化するところに特徴がある。

- PPC（packed pipeline control）方式：

  ストリーム間の切換は強制ではなく、キャッシュ・ミスヒットまどで、パイプラインに比較的に長いストールが生じたときのみ、実現される方式である。

6 プロセッサの将来展望

6.1 何が要されるのか

- 命令レベルでの並列化は限界に達しているのか：

マルチスレッディングによって、パイプラインをほぼ１００％で有効に使えるが、いつでも複数のストリームがあるわけではない。そのため、命令レベルの並列化の研究がまだ続けられるであろう。ただ、命令の並列化をコンパイラで行い、VLIW にフォーマットし、計算機に引き渡す方式は一般になるであろう。

- アプリケーション指向への分岐 ：

  今までは、計算機が速ければよいだという考え方は一般で、ハードウェアを作って、その上にアプリケーションを作るのであった。しかし、これからはアプリケーションのトレンドを予測しながら、プロセッサのアーキテクチャを設計していくことも必要である。

  そういう意味で、今は、プロセッサの歴史において、分岐点にあるかもしれない。即ち、どんな AP に対しても効率よく高速な性能をもつ枝とある種の AP に対して強いが、それ以外の AP に対しては使用効率が低下する枝のふたつである。

- デバイス技術をどこまで活かせるか ：

  プロセッサのゲート規模に関しては、Moore の法則 （3 年で４倍）よがく知られている。また、クロック速度は年約２２％、ピン数は約１７％の割合で増加しているというデータがある。

  これらをどのように使って、プロセッサの性能を向上させるかにかかる。選択枝として以下に幾つかの機構があげられる。

  - DRAM 混載
  - 大量レジスタ
  - 大量機能ユニット

  また、アーキテクチャの柔軟性も要求されるのであろう。即ち、設計したアーキテクチャはすぐに新しいデバイス技術に対応できる必要がある。

6.2 注目される幾つかのアーキテクチャ

- マルチスレッディング :OLTP (on-line transaction processing) のように、入力データによって、処理流れが変化する非決定的問題に対してはこの方式が有効である。
- On-chip Multiprocessor :デバイス技術の進歩で、将来複数のプロセッサを１つのチップに収納できるようになる。
- System On-chip :チップに１つの計算機システムを載せられる時代もやってくる。

7 感想

この資料には単位プロセッサに関する一部の基礎的な知識しか書いていない。プロセッサの研究に興味ある人に是非抑えてほしい内容である。

この資料に詳しいことは書いていないが、単位プロセッサの性能限界を征服しようとマルチプロセッサが多く開発されている。

ここで、紹介したプロセッサはノイマン方式のもので、非ノイマン方式のプロセッサがたくさんある。しかし、いずれもまだノイマン方式プロセッサほど汎用性をもっていないので、ノイマン方式プロセッサがまだ長生きするそうである。

参考文献

[1] 富田眞治 :コンピュータ ·アーキテクチャ、丸善 （１９９４）

[2] John L.H, David A.P. :コンピュータ ·アーキテクチャ (設計 ·実現 ·評価の定量的アプローチ)、富田眞治、村上和彰、にいみ治男 共訳、経 BP 社 （１９９２）

[3] John L.H, David A.P. :コンピュータの構成と設計 （下)、成田光彰 訳、日経 BP 社 （１９９６）

[4] 中澤喜三郎 :計算機アーキテクチャを構成方式、朝倉書店 （１９９５）

[5] Mike Johnson :スーパースカラ ·プロセッサ、村上和彰 監訳

[6] 國領、富田眞治 :マルチスレッド処理をサポートするVLIWプロセッサ・アーキテクチャ, 情報処理学会第４７回 (平成５年後期) 全国大会

[7] 安藤秀樹 :投機的実行を行うマイクロ・プロセッサに関する研究、論文（１９９６）

[8] 坂井修一 :オンチップ・マルチプロセッシングに関する初期検討

[9] Design Wave

[10] 日経エレクトロニクス

[11] 日経パソコン

[12] 五島正裕先生のプリント (プロセッサの技術動向、平成９年度研究テーマの紹介)

[13] J.Fisher: Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. Computer, C-30, 7, pp. 478-594.