勉強会資料
プロセッサ アーキテクチャ

辰巳 尚吾

1995年5月11日

1 プロセッサの高速化手法

プロセッサの高速化の手法としては次のようなレベルが考えられ、それぞれは、どのようにしたら改善できるのかについて調べていく必要がある。

1. ハードウェア技術上の改善
    - 論理素子のスイッチング時間の短縮
    - メモリ素子のアクセス時間の短縮
    - 配線遅延時間の短縮
2. 論理方式上の改善

    いかにして命令パイプラインの乱れを最小限にし、かつ命令レベルでの並列性を出すかで改善できる。
3. 機械命令セットの改善

    代表的な2つの方式–RISC方式とCISC方式–では、システム性能にどのような違いを表すか、について比較検討していくことで、どのような方式がいいのかを見い出してく。

この資料では、「2. 論理方式上の改善」「3. 機械命令セットの改善」に関してまとめていきたいと思う。

2 論理方式上の改善

パイプライン方式を分類すると

1. 処理要求が入ってきた順番で処理され、その順に結果が出力される場合

    ···(パイプライン方式-単純な場合)

2. 命令処理順に飛び越しがあったり、複数の命令を同時にパイプラインに投入される場合 (乱発行、乱終了/VLIW、スーパスカラ) $\cdots$(パイプライン方式-応用)

がある。2. は 1. の応用であるので、1 から順にまとめていくことにする。

2.1 パイプライン方式-単純な場合

2.1.1 命令パイプラインの実行過程

パイプラインのステージ　命令の実行は、次の段階に分けることができる。

IF 命令フェッチ (読み出し)

D デコード (解読)

OA オペランドアドレス変換

OF オペランドフェッチ

EX 演算実行

S 結果格納

これらのステージを逐次的に進めることによって命令を実行していく。

2.1.2 パイプラインの乱れの要因

命令パイプライン方式では理想的な場合は 1 マシンサイクルごとに 1 命令の実行が可能となる。しかし、実際には命令パイプラインの実行を乱す要因がある。あるステージがある要因で次のステージに進めなくなっている状態をインターロック (interlock) またはハザードという。このために起こる待ち時間をバブル、パイプラインの乱れとよぶ。ハザードの要因を大別すると、データ依存、制御依存、資源競合によるものがある。

データ依存	先行命令の結果を後続命令が使用する場合に生じる
制御依存	先行命令が後続命令を実行するかどうかを決定する場合、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、後続命令は待たされる。
資源競合	ハードウェア資源は有限であるので、それらを取り合いする場合が生じる。取り合いは優先順位で処理される。資源を獲得できなかったものは資源獲得まで待たされる。

次に、具体的にどのようなときにハザードが生じるのかをあげる。

1. オペランドデータ待ち (データ依存) :命令パイプラインに入っている先行命令の演算結果を後続命令が必要とする場合、後続命令の OF ステージに待ちができる。

先行命令	IF	D	OA	OF	EX	S		
後続命令		IF	D	OA	-	-	OF	EX	S

2. オペランドアドレス待ち (データ依存):先行命令が後続命令のオペランドアドレスを決定する場合、OA ステージに待ちができる。(先行命令でインデックスレジスタの内容を更新し、後続命令では、インデックス修飾でオペランドアドレスを決める場合など)

先行命令	IF	D	OA	OF	EX	S			
後続命令		IF	D	-	-	-	OA	OF	EX	S

3. 条件分岐命令の実行 (制御依存) :条件分岐命令のテスト結果を計算している間に、後続命令がパイプラインに投入される。分岐不成立の場合は、ハザードは生じないが、分岐成立の場合は、パイプラインに投入された命令はすべて無効化し、分岐先アドレスから命令フェッチを開始するため、ハザードが生じる。

⇓この時点で分岐不成立がわかる

分岐命令	IF	D	OA	OF	EX	S						
後続命令		IF	D	OA	OF	EX	S					
			IF	D	OA	OF	EX	S				
				IF	D	OA	OF	EX	S			
					IF	D	OA	OF	EX	S		
						IF	D	OA	OF	EX	S	
分岐先の命令							IF	D	OA	OF	EX	S

線ではさまれている部分が無効化され、分岐先の命令が実行される。

4. 割り込みの発生 (制御依存) :外部割り込みや演算例外は、S ステージでチェックされる。また、ページフォルトや、TLB ミスヒットは OF ステージで発生する。このように割り込みが発生すると、パイプラインを無効化し割り込み処理部分から命令を実行される。

5. 命令の書き換え (制御依存) :先行命令が、後続命令の内容を書き換えるときに起こる。

6. 制御レジスタの変更 (制御依存) :先行命令が、制御レジスタを書き換えるときに起こる。例えば Intel 社の 80486 では、リアルモード、プロテクトモード間を移行する時などに、パイプラインを無効化する必要がある。[1]

7. キャッシュミスヒット (資源競合) :キャッシュミスヒットが起こったときに実記憶装置からキャッシュにデータを転送する間、バブルが生じる。

---

[1] ただし、80486 は明示的に無効化する命令 (例えば JMP など) を挿入する必要がある。

8. キャッシュアクセス競合 (資源競合) :命令間で IF、OF、S ステージが重なる場合に待たされる。

IF	D	OA	OF	EX	S				
	IF	D	**OA**	OF	EX	S			
		IF	D	OA	OF	EX	-	**S**	
			-	**IF**	D	OA	**OF**	EX	S
					IF	D	OA	OF	EX	S

9. 複雑な演算実行 (資源競合) :機械命令の演算が、浮動小数点演算や OS 向き命令などのように複雑で、演算実行に複数のマシンサイクルを必要とする場合には、後続命令は演算実行に入れない。

先行命令	IF	D	OA	OF	EX	EX	EX	EX	S	
後続命令		IF	D	OA	OF	-	-	-	EX	S

2.1.3 各ステージでの高速化手法

パイプラインでの各ステージごとの高速化の工夫をあげていく。さらに、これらの工夫で改善されるパイプラインの乱れがあれば、それに関しても触れる。

1. 命令フェッチの高速化:
    - 命令用キャッシュメモリ:命令とデータ用のキャッシュメモリを分離して使用 (ハーバードアーキテクチャ)。
    - 命令バッファ:1 度に多くの命令をフェッチし、バッファに蓄えておく。

    これらの方式をとると、キャッシュメモリに対する資源競合を減らすことができる。

2. デコードの高速化: これは機械命令の複雑さによって大きく異なる。機械命令セットの改善で述べる。

3. オペランドアドレス待ちの回避:(2.1.2)2参照。先行命令がインデックスレジスタの更新などの後続命令のオペランドアドレス決定に関与するような演算をしている場合に、後続命令がオペランドアドレス待ちとなる。先行命令がレジスタ-レジスタ命令の場合、OA ステージでも利用できる演算器を用意しておくと、演算に必要なデータは揃っているので、先行命令は OF、EX ステージを待たずに結果を出すことができる。これを後続命令が利用することでオペランドアドレス待ちを回避できる。

4. 分岐命令の高速化:

無条件分岐　自己相対アドレッシングモードでは PC との変位の加算、インデックスアドレッシングモードでは 2 つの汎用レジスタとの変位の加算が必要になる。これらを OA ステージで行なうと 2 つのバブルが生じる。デコードが簡単な命令セットでは、これらを D ステージで済ませることができるので、バブルは 1 つで済む。さらに命令バッファを利用している時に分岐先を予測しておくとバブルはなくなる。

条件分岐

- 分岐条件の設定操作と分岐操作の分離: 比較命令と分岐命令が別々になっている命令セットでは、比較命令と分岐命令の間に命令を組み込むことで、パイプラインの乱れは防げる。ただし、比較結果を変更するような命令は組み込むことができない。これは、コンパイラの支援によって実現できる。
- 遅延分岐: 比較命令と分岐命令が一つの命令になっている命令セットでは、上で述べた方法は使えない。そこで、この命令が実行されてから実際に分岐をするまでにある一定の量だけ後続の命令を実行するという機能をつけ加える。コンパイラによって命令をうまく配置すれば、バブルをなくすことができる。
- 予測分岐: 条件が成り立つかどうかを過去のデータをもとに予測し、パイプラインに投入していく方式。
- 複数命令流: 命令パイプラインを複数個用意して条件成立側と不成立側の両方をパイプラインに投入する方式。

比較命令	IF	D	OA	OF	EX	S							
埋め込まれた命令		IF	D	OA	OF	EX	S						
埋め込まれた命令			IF	D	OA	OF	EX	S					
埋め込まれた命令				IF	D	OA	OF	EX	S				
分岐命令					IF	D	OA	OF	EX	S			
分岐不成立側命令						IF	D	無効化					
分岐不成立側命令							IF	無効化					
分岐成立側命令								IF	D	OA	OF	EX	S

分岐条件の設定操作と分岐操作の分離の例

比較+分岐命令	IF	D	OA	OF	EX	S						
埋め込まれた命令		IF	D	OA	OF	EX	S					
埋め込まれた命令			IF	D	OA	OF	EX	S				
埋め込まれた命令				IF	D	OA	OF	EX	S			
埋め込まれた命令					IF	D	OA	OF	EX	S		
埋め込まれた命令						IF	D	OA	OF	EX	S	
分岐先命令							IF	D	OA	OF	EX	S

遅延分岐 (比較+分岐命令) の例

5. データバイパス機能: データバイパス機能とは、通常のデータの転送路以外に脇道を作っておき、特定の場合にその脇道を利用して高速にデータを供給するする方式である。

   - 演算結果バイパス機能: 演算結果を汎用レジスタを介することなく得られるようにするバイパス。オペランドデータ待ちによるバブルはこれによってなくせる。

先行命令	IF	D	OA	OF	EX	S			
後続命令		IF	D	OA	-	-	OF	EX	S

     ⇓

先行命令	IF	D	OA	OF	EX	S
後続命令		IF	D	OA	OF	EX	S

   - ロードストアバイパス機能:

     ```
     LOAD   R1         M(RB+D1)
     MULI   R1         R2
     STORE  M(Rb+D2)   R1
     ```

     の命令系列の場合、第一命令でメモリの内容をレジスタに代入し終るまで第二命令は実行できない。しかし、ロードストアバイパスを利用すると第一命令でオペランドフェッチが済んだ段階で第二命令が実行できる。

6. 演算装置のパイプライン化: 演算の種類によって、必要な時間が変わってくる。先行命令で演算時間の長い命令を実行していると、後続命令はこの終了を待たなければならず、バブルが生じる。これを解消するために、演算装置もパイプライン化すると、1 マシンサイクルごとに演算器に投入しける。ただし、同一演算を繰り返して結果を求める方法をとっているもの (Newton-Raphson 法) は、パイプライン化できない。

7. 内蔵型アレイプロセッサ: 汎用大型コンピュータの演算装置にスーパーコンピュータの考え方を導入して、ベクトルレジスタや、ベクトルアドレス加算器などの最小限の付加装置によって、科学技術計算におけるベクトルや配列処理の高速化をはかったものが内蔵型アレイプロセッサである。

8. マイクロプログラムの分散化: マイクロプログラムを格納する制御記憶を集中管理させるのではなく、各演算装置に物理的に近い位置に分散配置する。このことで、マイクロ命令の伝搬遅延時間を減少させ、かつ演算装置を専用のマイクロプログラムによってきめ細かく高速制御できる。

9. 大容量レジスタ: レジスタを積極的に利用することによって、

   - レジスタ-レジスタ演算をベースとした命令パイプライン構成とすることによってマシンサイクルの短縮化をはかれる。
   - 命令長が短くなり、記憶用量が小さくてすむ。

- 主記憶への参照頻度が減少し、高速化がはかれる。

という効果がある。大容量のレジスタを用意することで、レジスタを積極的に利用することができる。また、手続きの引数受渡しにレジスタを利用することで効率良い手続き呼びだしが行なえる。

2.2 パイプライン方式-応用

2.2.1 命令依存関係

命令間の依存関係は、データ依存、制御依存、資源競合があるが、データ依存はさらに次の3つに分類される。

フロー依存 先行命令の結果を後続命令が使用するので、後続命令は先行命令の終了を待つ必要がある。

逆依存 先行命令が使うレジスタやメモリに後続命令がデータを書き込むとき、先行命令がデータを読み出した後に書き込まなければならない。

出力依存 書き込みの際は後続命令の結果を残す必要がある。

2.2.2 命令レベルでの並列処理の分類

命令レベルでの並列処理は以下の項目で分類できる。

- ハザードの検出方法
  - 静的ハザード検出···コンパイル時にハザードを検出し、「パイプライン方式-単純な場合」で説明したように解消できるハザードを解消する。特徴は以下の通り。
    - ハードウェアが単純
    - NOPが多用され、プログラムサイズが増大し、キャッシュメモリとの親和性が悪くなる。
    - 後続機種との目的コードレベルの互換性がなくなる。
  - 動的ハザード検出···ハードウェアに命令の依存解析をする能力があるときに対応し、依存関係が生じたとき、ハードウェアでインターロックをかける方式である。ハードウェアが非常に複雑になるという特徴がある。
- 命令の発行順序

  命令の発行とは、命令パイプラインのデコードステージから演算ステージに命令実行ステージを移すことを指す。

  - 順発行···プログラムのフェッチされた順に命令を発行する。

  - 乱発行···依存関係にある命令はリザベーションステーションに格納され、その命令の後続命令が発行される。

- 命令の終了順序

  命令の終了とは、S ステージの実行終了を指す。

  - 順終了···命令のフェッチされた順に終了する。
  - 乱終了···演算命令が終了し次第 S ステージに入る。

- 同時命令の発行方法

  - スカラ方式···一度に一つずつ命令を発行する。
  - スーパスカラ方式···どのような命令の組み合わせでも演算装置があいていれば複数個の命令を同時に発行できる。デコード出力と演算装置の間には配線量の多い命令発行ネットワークが必要である。
  - VLIW 方式···各演算装置に対応するフィールドからなる長命令を持つ。従って、同時に発行できる命令の組み合わせには限りがある。スーパースカラ方式に比べて命令発行ネットワークの複雑さは軽減されるが、長命令内に指定すべき命令が存在しない場合は、NOP が挿入されビット使用率が悪くなる。

以上の分類項目によって論理的には 24 通りの方式が考えられる。(中には、矛盾する組合せもある。) そのうち、動的順発行乱終了スカラ方式、動的乱発行乱終了スカラ方式、動的順発行乱終了スーパスカラ方式、静的順発行乱終了 VLIW 方式について説明する。

2.2.3 動的順発行乱終了スカラ方式

一般的に乱終了すると、

- レジスタの同時書き込みが生じるので、マルチポートのレジスタが必要となる。
- EX、S ステージでの制御が複雑になる。
- 分岐予測をした際、分岐命令の S ステージが終了する前に予測された側の命令系列を終了させてはいけない。予測失敗した場合にレジスタの復旧ができないため。

2.2.4 動的乱発行乱終了スカラ方式

この方式の例として、IBM360/91 や CDC6600 などがある。

IBM360/91での依存関係解消　Tomasuloアルゴリズムを利用している。

逆依存　先行命令は必ずレジスタを後続命令より先に読み出し別にとっておくことで解消。

フロー依存　演算の開始、終了時に、レジスタの値が有効であるか無効であるかを示すビジービットを、必要であれば更新して、後続命令が演算ステージに入れないようにする(リザベーションステーションで待つ。)。

CDC6600での依存関係解消　IBM360/91では出力依存が検出されると以後の命令の発行を停止する。レジスタへは、直接レジスタ番号でアクセスでき、連想アクセスするTomasulo方式よりもハードウェアを簡略化できる。

また、出力依存はコンパイル段階でのレジスタリネーミングで対処できる。

2.2.5　スーパスカラ方式

これらの特徴は以下の通り。

- 実行時での並列性検出
- 目的コードレベルでの互換性
- 機械命令のビット使用率向上

乱発行乱終了方式は、

- ハードウェアが極めて複雑になる
- 最適化コンパイラの支援によって同等な高速化が可能なことが多い
- 浮動小数点演算装置のパイプラインステージ数が短く設計できるようになり、命令の乱発行の効果が少なくなった。

などの理由で、順発行乱終了スーパスカラ方式が主流になっている。

2.2.6　静的順発行乱終了VLIW方式

通常静的順発行VLIW方式を単にVLIW方式と呼ことが多い。

VLIW方式の特徴

- VLIW方式では、並列性の検出がすべてコンパイル時になされる。
- 上の理由から、ハードウェア構成は簡潔化でき、高速化につながる。
- 並列操作の埋め込みが少ないと通常の機械命令に比べてビット使用率が低くなる。
- 目的コードレベルでの互換性がない。

マルチスレッド VLIW 方式　パイプラインは相互に依存関係のない命令が投入される時最大性能を発揮するので、別の命令系列をサイクリックにパイプラインに投入することを考える。一つのプログラム内に存在する同時実行可能な命令系列をスレッドと呼ぶ。プログラムカウンタやレジスタはスレッド分だけ必要となる。

スレッドの切替えを 1 マシンサイクルごとに行なう方式では、パイプラインのステージ数分のスレッドが必要であり、処理の優先順位を考慮できないなどの欠点がある。

そこで、同一スレッドに依存関係が生じた段階でスレッドを切替える方式を考える。

データ依存や資源競合などコンパイル時に検出できるパイプラインのハザードの解消のために、有効な命令を挿入できなかった場合は、通常ならば NOP 命令を挿入することになる。しかし、命令内にそのような依存関係が生じることを明示的に表せるフィールドを付け加えることによって、スレッド切替を実行させせることができ、有効でない命令を埋め込まなくても良い。

このようなことから、VLIW 方式とマルチスレッド方式の親和性が良いといえる。

3　機械命令セットの改善

3.1　CISC と RISC の命令セット

CISC の代表例である VAX と、RISC の代表例である R3000 を比較する。

3.1.1　命令形式とアドレッシングモード

VAX 命令形式は直交型である。アドレッシングモードも多数ある。

汎用レジスタを利用したもの…即値/レジスタ/レジスタ間接/自動デクリメント/ 自動インクリメント/自動インクリメント間接/バイト、ワード、ロングディスプレースメント/バイト、ワード、ロングディスプレースメント間接/インデックス/

プログラムカウンタを利用したもの…即値/相対/相対間接

バイト、ワード、ロングディスプレースメント (間接) などのモードがあるため、命令長は可変長になる。

R3000 命令形式は 32 ビット固定長であり、非直交型である。アドレッシングモードはディスプレースメント M(R+D) のみである。D は符合つき 2 バイト長である。

3.1.2　命令の複雑さ

VAX 単純な命令から複雑な命令までさまざまな命令がある。複雑な命令は以下のようなものがある。これらの複雑な命令のため、マイクロプログラム制御方式をとっている。

多項式演算/キュー操作/ビット操作/文字列操作/10 進演算/手続き呼び出しと復帰

R3000 四則演算、論理演算、シフト演算など単純な命令ばかりである。このため、制御方式として、布線論理方式で対応できる。

3.2 マイクロプログラム制御方式について

機械命令の実行は、命令フェッチ、デコード、オペランドアドレス変換、オペランドフェッチ、演算実行、結果格納などの処理を経てなされる。制御装置は各処理過程で算術論理演算装置、レジスタ、主記憶、バスなどの装置に対して適切な制御信号を与えるように構成される。CISCのように命令形式や機能が複雑であると、制御信号は制御装置内で複雑な状態遷移によって生成される。これを布線論理で構成することは困難である。そこで、マイクロプログラム制御方式が導入された。

マイクロプログラム制御方式とは、制御記憶に格納されたマイクロ命令とシステム内の状態にしたがって制御装置の状態を遷移させ、各機能装置に制御信号を送る。マイクロ命令は、機能装置を制御する制御信号指定部とマイクロ命令の順序制御指定部から構成される。

マイクロ命令の形式には次の2つがある。

水平型マイクロ命令形式 非常に長い形式であり、1マイクロ命令で、制御信号指定部としてALU演算指定、メモリアクセス指定など多数の機能装置を同時に制御できる。高速。マイクロ命令のビット使用効率悪くなりやすい。

垂直型マイクロ命令形式 非常に短い形式で、1マイクロ命令でわずかな機能装置しか制御できない。高速化できない。ビット使用率は向上する。

マイクロプログラム制御方式の特徴は以下のようなものである。

1. 複雑な制御装置の設計を組織的に行うことができ、設計、製作が用意になる。
2. 設計の際、マイクロプログラムの作成と論理回路の設計に分離され、並列して作業が進められる。
3. 機械命令レベルで同一のアーキテクチャを有し、さまざまな性能価格比をもつコンピュータ郡を設計することができる。
4. 診断用マイクロプログラムを実行することで、ハードウエアの故障情報を収集できる。
5. 異なるコンピュータの目的コードをそのままの形でマイクロプログラムで高速に実行できる。

3.3 RISC圧巻の理由

現在、ワークステーション分野ではRISC方式が主流になっている。この理由に関して見ていく。

3.3.1 VLSIの制約

1. 経済的に実現できるチップ面積には限りがあり、限られたゲート数内で論理設計を行う必要がある。

2. 内部結線が多くならないようにPLAやROMなどをうまく利用して規則性のある回路を構成する必要がある。
3. チップから取り出せるピンの数には限りがある。

3.3.2 VLSI制約下でのRISCのアプローチ

VLSIの諸制約の下でRISCは次のような方針をとった。

1. コンパイラが利用しないような複雑な機械命令セットを排除して簡単な命令セットを採用。
2. 簡単な命令セットにして余ったチップ面積を内部レジスタスペースなどにあて、高速処理を達成する。
3. 制御装置を簡潔化してチップ設計時間を大幅に短縮させる。

この結果RISCプロセッサは??で説明したような特徴を持っている。

3.3.3 VLSI制約下でのCISCのアプローチ

1. 高機能命令の採用により、プログラムサイズを小さくし、さらにキャッシュミスヒットを小さくする。
2. 高機能命令を多数のハードウェアの導入や並列処理によって高速化できる。
3. OSの命令など、割り込みや他のプロセッサの影響を排して排他的なメモリにアクセスに向いたプロセッサにする。

3.3.4 CISCアプローチの問題点

以下にあげるようなCISC問題点のため、ワークステーション分野ではRISCが優勢になっている。

1. チップ内で占める制御スペース:CISC方式では、これまでに述べた通り、マイクロプログラム制御方式に頼らざるをえない。制御記憶自身の容量が大きくなり1チップで作成する時には、多くの面積を占有する。

   演算系速度と機械命令供給速度の関係によって、高速化の手法を考えると以下のように分析できる。その際のマイクロプログラム方式の位置付けがどのようになるかも考えていくこととする。

   - 演算速度$\gg$機械命令供給速度のとき: プログラムサイズを小さくし、できるだけ主記憶へのアクセスを減らし、1命令で多くの機能を果たしていく必要がある。これはCISCに対応する。マイクロプログラム制御記憶と主記憶とのアクセス速度の差が非常に大きい時は、マイクロプログラムの存在意義は大きい。

- 演算速度$\sim$機械命令供給速度のとき: キャッシュメモリ、制御記憶両方がチップ内に収まる時は、水平型のマイクロプログラム命令を用いると並列処理の可能性のある高機能の命令を高速実行できる。ただし、演算装置やキャッシュメモリが多数装備される必要がある。チップ内にどちらか一方しか収まらない時は次のようになる。
  - キャッシュメモリを選択: CISC 方式は実現できない。
  - 制御記憶を選択: 命令フェッチに時間がかかる分、1CISC 命令で多くの処理を実行する必要がある。実際には、命令解釈実行でのオーバーヘッドがあったり、単純命令が多く現れることが多く、RISC よりも性能が低下する。
- 演算速度$\ll$機械命令供給速度のとき: 演算系を多重化してバランスをとる必要がある。

現在では、「演算速度$\sim$機械命令供給速度」の場合に相当し、チップ面積に関してはキャッシュメモリ、制御記憶の両方を収容できるほど大きくないので、RISC 方式の方が有利である。

2. 命令デコードの複雑さ

3. 一般性重視による無駄な操作:一般性を持たせて設計されているので実際の場合には必要のない操作も命令実行で一緒に行なわれてしまう場合があり、無駄がある。

4. **CISC** 命令間でのコードの最適化の困難さ

参考文献

[1] 富田眞治 コンピューターアーキテクチャI 5-7 章,1994.

[2] 蒲地輝尚 はじめて読む 486.