

# はじめに

- 前回は、マシン命令の処理を高速化するために開発・導入された命令パイプライン制御方式について講義しました。
- しかし、もっと性能を上げたい/上げて欲しい、という欲求/要求は留まることはありません。
- さて、どうするか？ 今回は、さらにプロセッサの性能を向上させる方法について探って行きます。
- 教科書の対応範囲は、以下の通りです。
  - 5.2.2項(e) (p.139~142)

# 命令レベル並列処理

- マシン命令の逐次実行は、フォンノイマン型コンピュータの特徴でした。
- その処理を高速化するために命令パイプライン制御方式が開発されました。
  - 前回の講義では、**時間的多重化**と空間多重化の両方を行なっていると述べましたが、どちらかという、時間的多重化の特徴が大きいでしょう。
  - つまり、時間的に細分化することによって、見かけ上、複数のマシン命令を同時に処理しています。

- もっと性能を上げろ、ということになると、容易に考えられるのは、空間的多重化をもっと押し進めるという方向です。

－つまり、複数のマシン命令を、文字通り「同時に」処理しよう、というわけです。

- 大まかなイメージとしては、命令パイプラインを複数本用意するとか、命令パイプラインの太さを太くして、各ステージで複数の命令を同時に処理する、と捉えることができます。

注) 命令パイプライン制御を廃止するわけではありません。もはや命令パイプライン制御は当たり前の技術として使用します。その上で、複数の命令を同時に処理しよう、というのです。

- 例えば、命令パイプラインのEXステージでは、演算器は1個で十分でした。しかし、複数の命令を処理すると、その数に応じた演算器が必要になります。このように、物理的なハードウェア資源の投資が必要になるのです。

さて、具体的な命令処理方式について説明する前に、まず、一般的なレベルで、**並列処理 (parallel processing)** についてお話ししておきます。

# 並列処理(parallel processing)

- 複数個の処理単位を同時に処理することを、**並列処理(parallel processing)**と言います。
- ただ、なんでも並列に処理できるわけではありません。処理自体にそのような性質(以下の「並列性」)がなければなりません。
- その性質を意味する術語として以下の2つがあります。
  - **並列性(parallelism)**: **高速化を目的として**、実際に複数個の資源があれば、同時に処理を行うことができるという性質。

- 並行性(concurrency)：ソフトウェアの構造として、複数の処理単位が互いに通信や同期を行いながら、同時に動作するという性質。ソフトウェア上の仮定的な話であり、実際にそれら複数の処理単位を並列に処理するかどうかという話とは直接の関係はない。

注) かつてはそれらの処理単位を時間的多重化で(1台のプロセッサで、それらの処理単位を時間的に切り替えながら)処理している例が多かったので、それらの例を見て誤解した人達は、資源を共有するか否かで「並列」と「並行」を区別するようですが、もちろん、これは間違いなので注意が必要です。

- ちなみに、プロセッサの設計者は主に「並列性」に着目します。一方、プログラミング言語の設計者は主に「並行性」に着目します。

- 並列処理の単位

- 何を同時に処理するのか、その単位は色々考えられます。細かいものから粗いものの順に並べると、以下のような単位が考えられます。

- 演算
    - マシン命令
    - ...
    - 文
    - 関数、サブルーチン
    - ...
    - スレッド(軽量プロセス)
    - プロセス

細粒度 (fine grain)

粒度 (granularity)

粗粒度 (coarse grain)

- 処理単位の機能的な大きさを粒度 (granularity) といいます。
  - 「文」を含めて、それより下の処理を単位とした時の粒度の定義は曖昧です。例えば、プログラミング言語で“文”と言っても、「 $a=b+c;$ 」と「 $a=a+(b-c)*(d+e/f);$ 」とでは処理量は全く異なります。
- 通常、1個のマシン命令で 1個の演算を行いますので、処理単位をマシン命令としても演算としても大差はありません。
- そこで、マシン命令(または演算)を単位として並列処理を行う場合を、「命令レベル並列処理」と言います。他に、「低レベル並列処理」「細粒度並列処理」と言う場合もありますが、全て同じ意味です。



- ちなみに、「粗粒度並列処理」や「中粒度並列処理」などと言う場合がありますが、どこまでが中粒度でどこからが粗粒度なのか、厳密な定義はありません。

- 命令レベル並列処理マシン(プロセッサ)

- 1つのマシン命令で複数の演算を同時に行うプロセッサをVLIW(Very Long Instruction Word)マシンと言います。

- マシン命令中で、OPコードとそのオペランドのセットを複数個指定しますが、それによって命令語長が非常に長くなるのがVLIWの名前の由来です。

- 1つのマシン命令で1個の演算を指定する点はそのままで、複数のマシン命令を同時に実行するプロセッサをスーパースカラ(superscalar)プロセッサと言います。

# 命令レベル並列性 (ILP: Instruction-Level Parallelism)

- 一般に、プログラム中の命令間には、以下のような依存関係があります。
  - データ依存 (data dependency):
    - フロー依存 (flow dependency)
      - 先行する命令が生成した値を、後続の命令が使用する場合の依存関係
      - 例)
        - 命令1:    add   r2, r3, r1                    ( $r1 \leftarrow r2 + r3$ )
        - 命令2:    mul   r1, r4, r5                    ( $r5 \leftarrow r1 \times r4$ )
        - » レジスタr1の使用に関して、命令2が命令1に依存

- 逆依存 (anti-dependency)

- 先行する命令が読み出したデータの格納場所に、後続の命令が書き込む場合の依存関係

- 例)

- 命令1:    add   r1, r2, r3                    ( $r3 \leftarrow r1 + r2$ )

- 命令2:    mul   r4, r5, r1                    ( $r1 \leftarrow r4 \times r5$ )

- » レジスタr1の使用に関して、命令2が命令1に依存

- 出力依存 (output dependency)

- 先行する命令がデータを書き込んだ場所に、後続の命令も書き込む場合の依存関係

- 例)

- 命令1:    add   r2, r3, r1                    ( $r1 \leftarrow r2 + r3$ )

- 命令2:    mul   r4, r5, r1                    ( $r1 \leftarrow r4 \times r5$ )


- » レジスタr1の使用に関して、命令2が命令1に依存

## – 制御依存(control dependency) :

- 条件分岐命令の条件が成立する(真)か否か(偽)で、実行する後続命令が変わる場合の依存関係

– 例)

命令1:    cmp   r1, r2                   (compare r1 with r2)  
命令2:    bne   ERROR                   (branch on not-equal)  
命令3:    add   r1, r2, r3 (r3 ← r1 + r2 )  
      .....  
          ERROR:  
命令4:    add   r4, r5, r3 (r3 ← r4 + r5 )  
      .....



» 命令3は、条件分岐命令である命令2に依存  
命令4も、命令2に依存

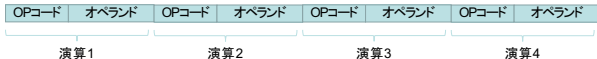
- 依存関係にある命令は、プログラム中での順番を守って実行しなければなりません。つまり、並列に(同時に)実行することはできません。

- 依存関係のない命令同士は、並列に(同時に)実行することが可能です。
- 命令間の依存関係は、命令レベル並列性を阻害する要因です。
  - 命令パイプラインの講義で、パイプラインハザードについて話をしましたが、論理ハザードを生じさせる原因が、命令間の依存関係です。
    - データ依存 → データハザード
      - フロー依存 → RAW(Read-After-Write)ハザード
      - 逆依存 → WAR(Write-After-Read)ハザード
      - 出力依存 → WAW(Write-After-Write)ハザード
    - 制御依存 → 制御ハザード
- ですから、プログラムから命令レベル並列性を抽出するためには、命令間の依存解析を行い、ハザードを回避する必要があります。

# VLIWマシン

- 1個のマシン命令で複数の演算を指定
  - 命令語長が長くなります。

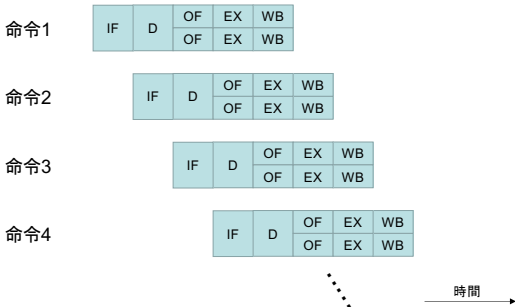
例) 並列度(演算数)が4の場合の命令語



- 従来の命令セットとは互換性がなくなります。
- 1個のマシン命令で指定された演算は同時に実行します。逆に、同時に実行可能な演算しか同じ命令内で指定してはいけません(依存関係にある演算は同じ命令内で指定できません)。

## 命令パイプライン

例) 並列度が2の場合のパイプライン実行の様子



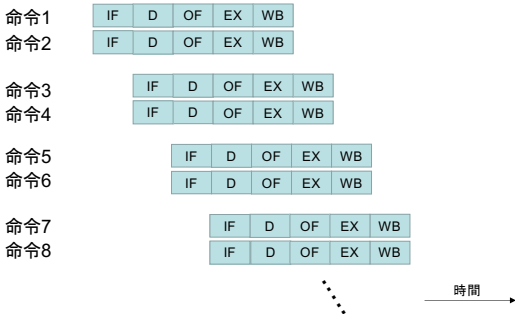
# スーパスカラプロセッサ

- 同時に複数個のマシン命令を処理
  - 従来の命令セットとは互換性が保てます。
    - 従来の命令セットを用いたプログラムをそのまま高速化できます。
  - 実行時に、ハードウェアで、命令間の依存解析を行います。



## 命令パイプライン

例) 並列度が2の場合のパイプライン実行の様子

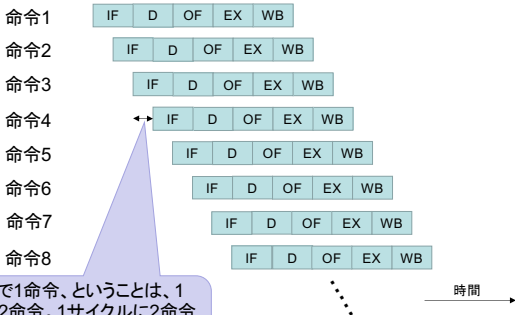


# スーパーパイプライン制御方式

- 命令パイプラインのステージを細分化して、パイプラインピッチを小さく、パイプライン段数を深くします。
  - どれくらいの段数以上ならスーパーパイプラインと呼べるのか、という明確な定義はありません。
- 従来の命令セットとは互換性が保てます。

## 命令パイプライン

例) パイプラインピッチを半分にした場合のパイプライン実行の様子



1/2サイクルで1命令、ということは、1  
サイクルで2命令。1サイクルに2命令  
(演算)を処理するスーパースカラや  
VLIWと理論的な性能は等価です。

# 命令レベル並列性の抽出

- 命令間の依存解析

- いつ、誰が依存解析を行う？

- 静的(実行前)に、ソフトウェアで(コンパイラが)やる:

- 解析に要する時間も、メモリ量についても、あまり気にすることなく依存解析を行うことができます。したがって、プログラム中の比較的広い範囲から、並列に実行可能な命令の組を見つけ出すことができます。

- 実行前の時点で、確実にわかっている事柄に対してのみ対応できます。実行中に変化する要因については、対応することができません。

- » 例えば、データの値によって演算時間が異なるような命令に対しては、最悪のケースを想定して並列を抽出するより仕方ありません。

- 動的(実行時)に、ハードウェアでやる：
  - 並列に実行できる命令の組み合わせを、フェッチした命令の中からしか探すことができません。つまり、狭く限られた範囲でしか解析できません。また、解析自体に時間をかけることも許されず、ハードウェア(論理回路)で処理することも考慮すると、ソフトウェアで行うような複雑な解析手法は採用できません。その結果、並列性抽出の可能性は制限されてしまいます。
  - 動的に変化する要因についても対応することができます。

- 各プロセッサはどちらを使う？

- VLIW、スーパパイプライン：
  - コンパイラで並列性の抽出を行います。
  - ハードウェアは、コンパイラが生成したコード(命令語)をその順番通りに実行するだけです。
    - もちろん、命令パイプラインのインタロック制御に必要な程度の依存解析を行います。

## – スーパスカラ:

- ハードウェアで依存解析を行います。
  - 実行時に命令レベルの並列性を抽出します。
  - ただし、レジスタの参照に関してのみ、依存解析を行います。メモリアクセスに関する依存解析はしません(できません)。
  - 後でもう少し詳しく説明しますが、場合によっては、プログラムに記述された命令の順序とは異なる順序で実行する場合もあります。
- もちろん、コンパイラでも並列性の抽出は行います。
  - ハードウェアだけでも並列性の抽出は可能ですが、それだけでは最高の性能は得られません。
  - コンパイラによって広い範囲から並列性を抽出し、さらに、動的な要因に対してハードウェアで修正する、というのが狙いです。

# スーパスカラの命令発行制御

- 処理する命令の順序
  - in-order = プログラム上で並んでいる通りの順に
  - out-of-order = in-order でない
- 命令フェッチ、命令解読は、in-orderで処理しますが、命令発行(演算実行の開始)は、in-orderで行う、out-of-orderで行うの2通りがあります。
  - もちろんout-of-order発行の方が高い性能が得られますが、複雑な依存解析機構が必要になります。
  - 次ページの例で、in-order発行とout-of-order発行の違いを説明します。

## 例) 並列度が2のスーパースカラで考えます。

命令1	div r2, r3, r1	(r1 ← r2 / r3)
命令2	add r1, r5, r4	(r4 ← r1 + r5)
命令3	mul r2, r3, r6	(r6 ← r2 * r3)

- 命令1の実行(除算)時間は長くなります。
- 命令2は、レジスタr1に関して命令1に依存しています。
  - 命令2の発行は、命令1の除算が終わるまで長く待たなければなりません。
- 命令1と命令2は同時に発行できません。in-order発行の場合は、命令2が発行できないのに、その後の命令3が発行できるはずがありません。
- 命令3は、命令1や命令2と依存関係がありませんので、out-of-order発行の場合は、命令1と命令3の演算を同時に開始できます。
  - このように、命令の実行順序を変更することで、並列実行の機会が増え、高速化できる可能性も高くなります。



# おわりに

- 現在の主要なマイクロプロセッサでは、並列度が4~6程度のスーパースカラプロセッサに、さらにスーパーパイプラインを組み合わせています。
- 命令レベルの並列処理はRISC向きです。
  - CISCの場合は依存解析が複雑になりすぎて、また、命令機能も高いので、実際には並列性があまり抽出できません。
  - x86命令セットはCISCの命令セットですが、Intel社やAMD社のプロセッサ(コア)は、内部でRISC命令に変換し、それを並列実行するスーパースカラプロセッサとして構成されています。