前回:
CPUでの命令の実行

今回:
パイプライン

キーワード:
パイプラインハザード(構造ハザード, 制御ハザード, データハザード),
ストール, 分岐予測, 遅延分岐

第11回 パイプライン

6.5 パイプライン実行の障害 - パイプラインハザード
次のクロックサイクルで次の命令を実行できないケース
三種類ある
  構造ハザード (structural hazard)
  制御ハザード (control hazard)
  データハザード (data hazard)

基本的な対処: 後続命令を待たせる
  ストール(バブルともいう)

6.5.1 構造ハザード
複数の命令が使うハードウェアが競合

例1: 命令フェッチとデータアクセス(図7.7)
メモリとの通路がひとつだと発生
  命令メモリとデータメモリの分離で解決
  (ハーバードアーキテクチャと呼ぶ)
  本物のメモリの実体は一種類
    メモリとCPUの間で分離→後述

例2: 浮動小数点演算器
複数クロックサイクル占有される(時間がかかる)
連続した命令で使用できない
  複数化
  演算器をパイプライン化

例3: 記憶階層におけるミス発生 (後述)

6.5.2 制御ハザード
命令の実行に関する判断(実行するかしないか)が,
以前の命令の結果に基づく場合
  その「以前の命令」がまだ実行中だと発生

つまり(条件)分岐命令
  次に実行する命令の番地がなかなか定まらない

制御ハザードの回避法

6.5.2.1 パイプラインストール(図7.8)
後続の命令を遅らせる

6.5.2.2 分岐予測
簡単には「分岐は成立しない」と予測
  次の番地の命令をフェッチに行く
予測が外れて分岐が成立した場合
  あらためて飛び先の命令をフェッチ
  実行しかけた次の命令を無効化

低位の番地への分岐は成立, という予測法もあり
  ループを構成する分岐命令を想定

さらに, 動的な分岐予測
ハードウェアで分岐の回数を数える
  直前の分岐命令の結果で次回を予測
  直前ともう一つ前の結果, という方法もある

6.5.2.3 遅延分岐 delayed branch(図7.9)
分岐命令の直後のストール部分(分岐遅延スロット 遅延スロット)に
別の命令を挿入
「別の命令」は分岐してもしなくても実行

通常は nop を入れる - ストールと等価
  アセンブラが自動的にいれてくれる
  (だからこれまでのプログラムで意識しなくてすんだ)

コンパイラが挿入可能な命令を見つけてくれる(こともある)
例： 本来なら分岐の前に実行すべき命令を分岐後に置く
```
    add $t2, $t2, 1
    bne $t1, $0, loop
```
  この順序を入れ替えて
```
    bne $t1, $0, loop
    add $t2, $t2, 1
```
  条件分岐の成立にかかわらず add 命令は実行される
  （$t2に関する処理は分岐の条件に無関係だから）

パイプラインの深さが深くなると遅延スロットが複数必要
  埋めるのが困難に

実例
```
<C program>
  int sum = 0;
  int i;
  for(i=1; i<=10; i++){
    sum += i;
  }
  return sum;

<no-delayed-branch>
        li      $3,1    # i
        move    $2,$0   # sum
        li      $4,11
$L3:
        addu    $2,$2,$3        # sum += i
        addiu   $3,$3,1         # i++
        bne     $3,$4,$L3
        nop                     # delay slot

        j       $31
        nop

<delayed-branch>
        li      $3,1    # i
        move    $2,$0   # sum
        li      $4,11
        addu    $2,$2,$3
$L4:
        addiu   $3,$3,1         # i++
        bne     $3,$4,$L4
        addu    $2,$2,$3        # sum += i

        j       $31
        subu    $2,$2,$3
```

xspim: -delayed_branches をつけて起動すると、分岐命令の次の
命令も（条件成立の可否に関わらず）実行するようになる。

6.5.3 データハザード(図7.11)
まだパイプライン中にある先行命令の結果を
あとの命令が使おうとする

例:
```
    sub $t3, $t1, $t2   # t3 = t1 - t2
    sub $t5, $t3, $t4   # t5 = t3 - t4
```
  $t3 が書き込まれるまで後の命令で$t3が読み出せない

コンパイラによる命令の並べ替え
  あまり有効ではないケースもあり

ストール回避(図7.12)
  フォワーディング, バイパシング
  途中のステージから結果を後の命令に送る

6.6 まとめ - パイプラインによる性能向上の三つの制約
  データハザードによるストール発生
  制御ハザードにより分岐が遅くなる
  パイプラインレジスタのオーバヘッド(ラッチ遅延)