マイクロプロセッサ特論　第12回
パイプラインハザード
テキスト9章　115~124

情報工学科
天野英晴

前回、パイプライン構造の基本を紹介しました。今回は、パイプライン設計における最大の壁となるパイプラインハザードを紹介します。

パイプラインハザードとは？

- パイプラインがうまく流れなくなる危険、障害のこと
  - 構造ハザード
    - 資源が競合して片方のステージしか使えない場合に生じる
  - データハザード
    - データの依存性により生じる
    - 先に進んだ命令の結果を後の命令が利用するため、その結果がレジスタに書かれるまで、読むことができない
  - コントロールハザード
    - 分岐命令が原因で、次に実行する命令の確定ができない
- パイプラインストール
  - ハザードが原因による性能の低下
  - パイプライン処理は理想的に動くとCPIが1
    - ストールによりCPIが大きくなってしまう

パイプラインは調子良く流れれば1クロックに1命令が終了します。しかし、場合によってはこれがうまく行かないことがあります。パイプラインがうまく流れなくなる危険、障害のことをパイプラインハザードと呼びます。ハザードには、資源の競合による構造ハザード、データの依存性により生じるデータハザード、分岐命令が原因のコントロールハザードの三つがあります。ハザードによりパイプラインがうまく流れなくなって性能が低下する現象をパイプラインストールと呼びます。

データハザード

- 直前の命令の結果がレジスタファイルに書き込まれないうちに、後続の命令が読み出しを行ってしまう
  - データの依存性により生じるハザード
- 一つ前、さらに一つ前まで問題に
- 複数命令を時間的に重ねて実行する場合には常に問題になる
  - Read After Write (RAW)ハザードと呼ばれる
  - Write After Read(WAR)はPOCOでは生じない
  - Write After Write(WAW)は通常あまり問題にならない
- 回避手法

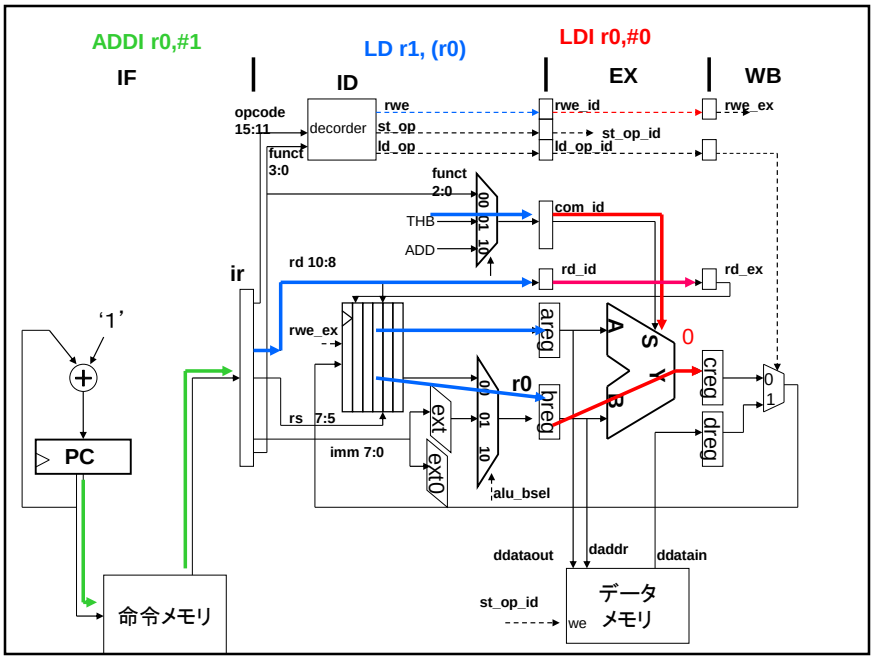
→ NOPを入れて命令の間隔を保持する

→ フォワーディング（Forwarding)

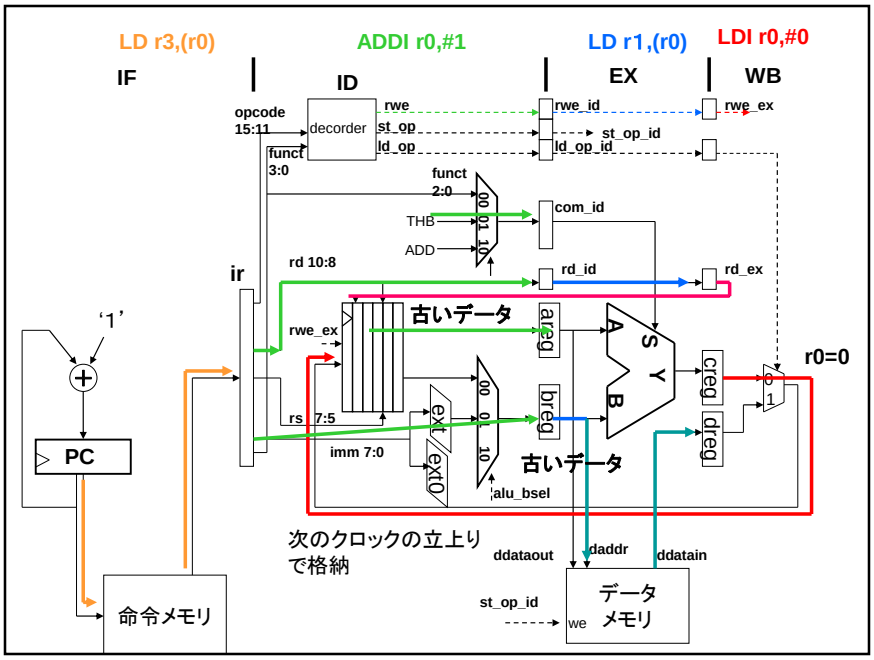
最新のデータを横流しにする

条件：1．後続の命令とレジスタ番号が一致　2．結果を書き込む命令

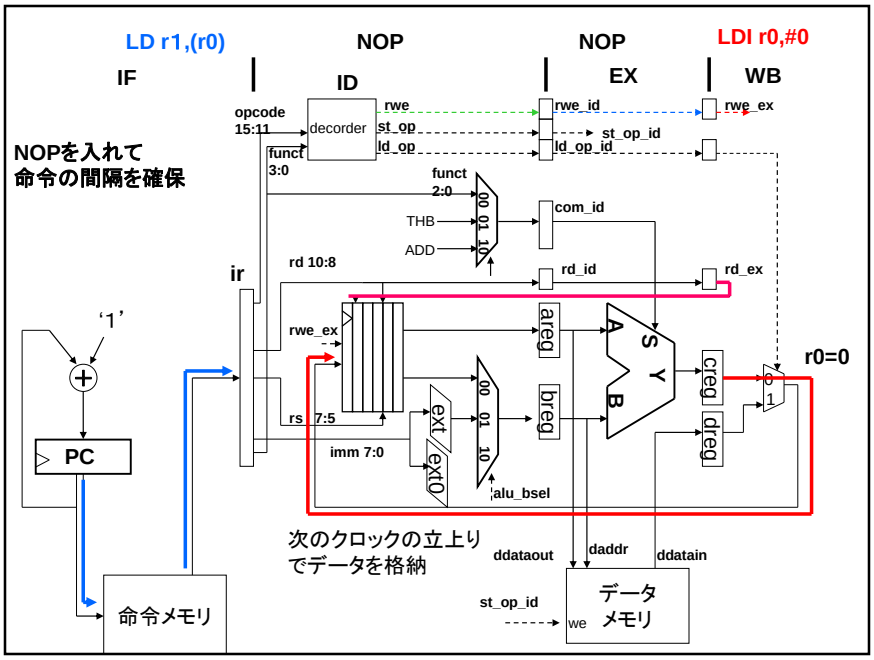
次にデータハザードを紹介します。データハザードは、直前の命令の結果がレジスタファイルに書き込まないうちに、後続の命令がこれを読みだすことにより起きます。相互に依存する命令を部分的に同時に実行しようとするために生じてしまい、パイプライン処理の本質的な問題点といえます。RAW、WAR、WAWの三つのハザードが考えられますが、POCOではRAWハザードだけが生じます。他のハザードについては後に説明します。



では、前回の演習のディレクトリでtest.asmのプログラムを実行してみましょう。このプログラムはLDI r0、＃0で、r0に0を入れて、これを使ってLD r1,(r0)でメモリの0番地を読み出そうとしています。ところが、LDI r0,#0がr0に0を書くのはWBステージの終わりです。しかしLD命令はLDIがまだEXステージに居るときにレジスタファイルを読み出してしまいます。ここで読まれるr0は古い値です。したがってレジスタがXになってしまいます。



同じようにLDIの次の次の命令であるADDIも、LDIがWBで書き込み終わる前に、読み出しを行うため、古いデータを読んでしまいます。



では、どのようにしてこの問題を回避できるでしょうか？先行した命令とこの結果を使う命令の間の間隔を広くしてやればいいのです。NOP、つまり何もやらない命令を二つ入れれば、LDI命令の結果が書き込まれてからLD命令が値を読み出すことができます。この図に対応するプログラムnop2.asmを実行してみましょう。きちんと動いていることはわかりますが、非常に時間が掛かります。

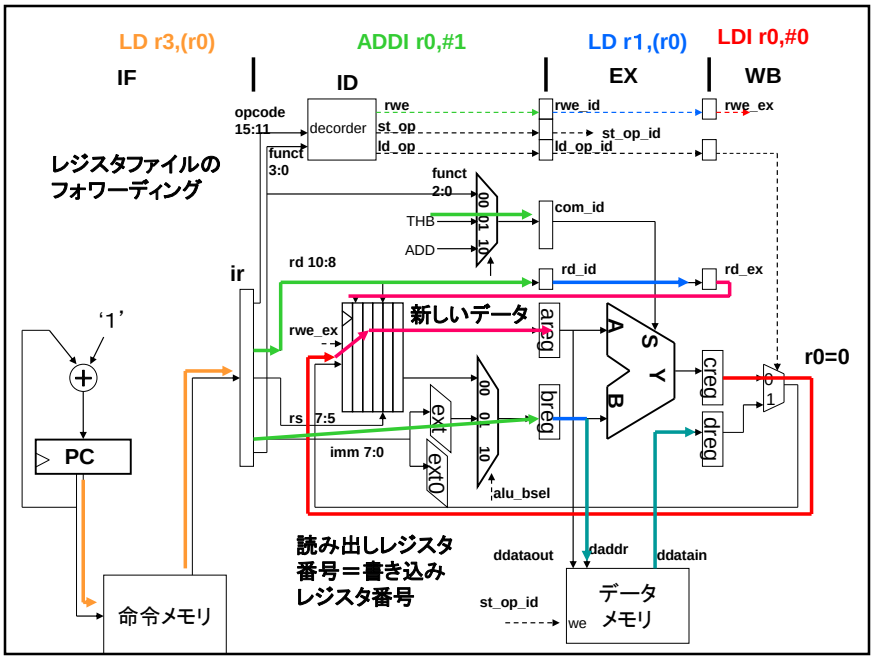
NOPを入れる方法

- NOPを2つ入れて命令間隔を確保

ハザード付きCPI＝
理想のCPI＋ストールの確率×ストールのダメージ

1 ＋ 後続の命令が利用する確率×２＝
1+ 0.8×2くらいはあるか？？
2.6は悪化のしすぎ

では先の構造ハザードと同じく、CPIがどの程度伸びるかを見積もりましょう。理想のCPIを1とします。ストールの確率は、ある命令の結果を後続の命令が利用する確率ですが、これは結構高いです。というのは多くの場合、ある命令は、次の命令で使うデータを作るために実行されるからです。ここでは0.8とします。ストールのダメージはNOP2個分なんで、CPIは2.6になってしまうことになります。これではパイプライン処理の性能向上が台無しです。



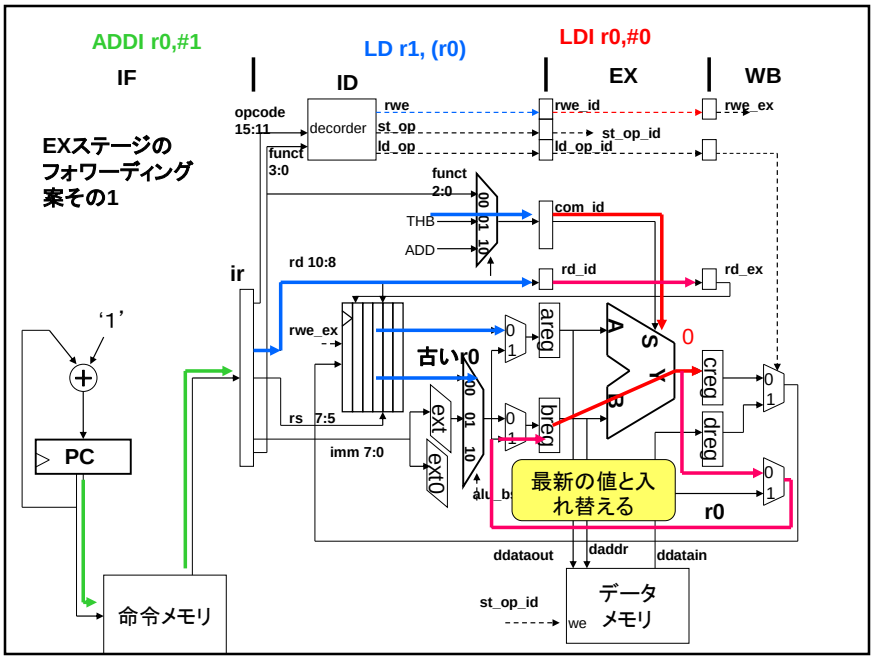
NOPを入れずにデータハザードを解決するためには、レジスタに書き込む前でも、用意のできた値を横流ししてやれば良いです。これをフォワーディングと呼びます。この例は、レジスタファイルにフォワーディングを付ける方法を示しています。この方法では、先行命令が、今書き込みつつある値を、IDステージにある命令が使う場合、つまり書き込むレジスタ番号と読み出すレジスタ番号が一致して書き込み信号rwe_ex=1になっている場合は、書き込む値をそのまま読み出すデータと入れ替えてareg, bregに入れてしまいます。この例ではaregの値が入れ替わります。これはWBステージからIDステージへのフォワーディングです。

レジスタファイルのフォワーディング Verilog記述

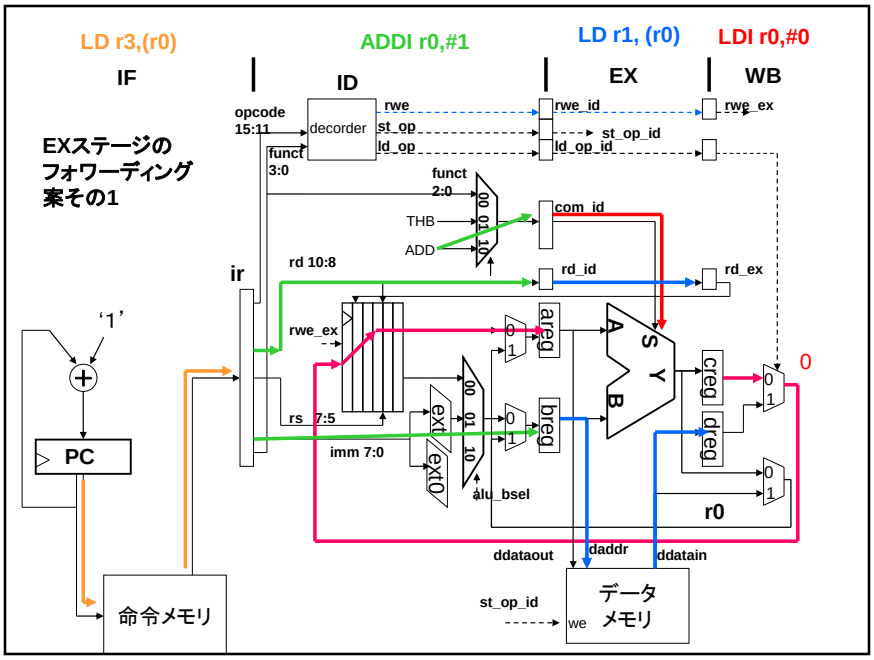
```
assign a =
        (aadr == cadr)&we ? c:
        aadr == 0 ? r0:
        aadr == 1 ? r1:
        aadr == 2 ? r2:
        aadr == 3 ? r3:
        aadr == 4 ? r4:
        aadr == 5 ? r5:
        aadr == 6 ? r6: r7;
assign b =
        (badr == cadr)&we ? c:
        badr == 0 ? r0:
        badr == 1 ? r1:
        badr == 2 ? r2:
        badr == 3 ? r3:
        badr == 4 ? r4:
        badr == 5 ? r5:
        badr == 6 ? r6: r7;
```

一致していれば書き込みデータを出力に直結

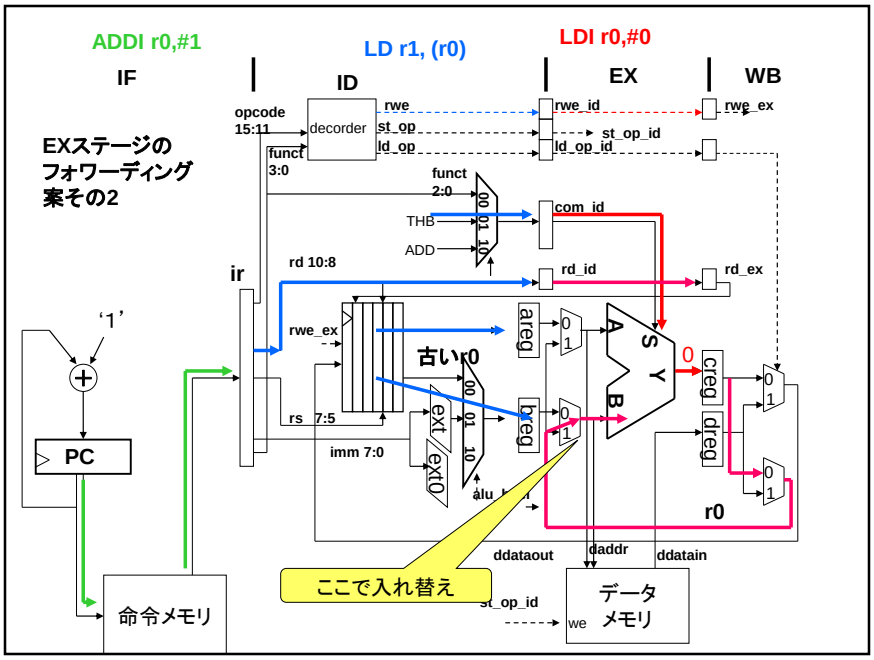
レジスタファイルのフォワーディングの記述は非常に簡単です。読み出しレジスタ番号と書き込みレジスタ番号が一致しており、かつwe=1(つまり書き込みが起きる)となっていれば、今書き込みつつあるc入力を直接a,bに横流しします。



次は同じ方法をEXスレー時とIDステージ間でも行ってみましょう。EXステージで計算の結果は出ているので、この計算したてのデータを、レジスタファイルから読み出した値と入れ替えて、areg, bregに入れてやります。条件はやはりレジスタ番号が一致して、rwe_1=1になることです。このためにareg, bregの入力にマルチプレクサをつけます。p2kaiの下のpocop.vはこの図と同じフォワーディング回路を使っています。p1kaiの下では動かなかったtest.asmが動作することを確認しましょう。



フォワーディングはrd,rsの両方に対して行う必要があります。これは、aポート側にフォワーディングを行った例です。



もう一つ、ALUの入力にマルチプレクサを付けた例を示します。どちらの方法でもフォワーディングを行うことができます。

フォワーディングの記述

案1を利用している

```
assign fwddata = (ld_op_id) ? ddatain: alu_y;
assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
               (addiu_op | ldiu_op) ? {8'b0,imm} :
              ((rd_id == rs)& rwe_id ) ? fwddata :rf_b;

assign fwda = ((rd_id == rd)& rwe_id) ? fwddata: rf_a;
…
always @(posedge clk or negedge rst_n ) begin
  if(!rst_n)
     rwe_id <= `DISABLE;
  else begin
       st_op_id <=  st_op; ld_op_id <=  ld_op; rwe_id <= rwe;
       com_id <= com; rd_id <= rd;
       areg <= fwda; breg <= alu_b;
   end
end
```

rega, regb両方にフォワーディングする必要がある

では、フォワーディングはどのようにVerilogで書けば良いでしょうか？まずフォワーディングのデータfwddataを作ってやります。LD命令で取ってきた値もフォワーディングする必要があるので、この点をきちんと書きます。次に、B入力に対するフォワーディングは、先行命令の書き込みレジスタ番号（rd_id)と現在IDステージにある命令のソースレジスタ番号（rs)が一致して、rwe_idが1の時に行います。これがそのまま論理式として書かれています。Aポートは、今までレジスタファイルからの信号がaregに直結されていたので、フォワーディングを行うために、新しくfwdaという信号を用意してやります。今度の条件は先行命令の書き込みレジスタ番号（rd_id)と現在IDステージにある命令のディスティネーション番号（rd)の一致を調べます。

後は、パイプラインレジスタを介して次のステージにデータを送ってやります。

一般的なデータハザード

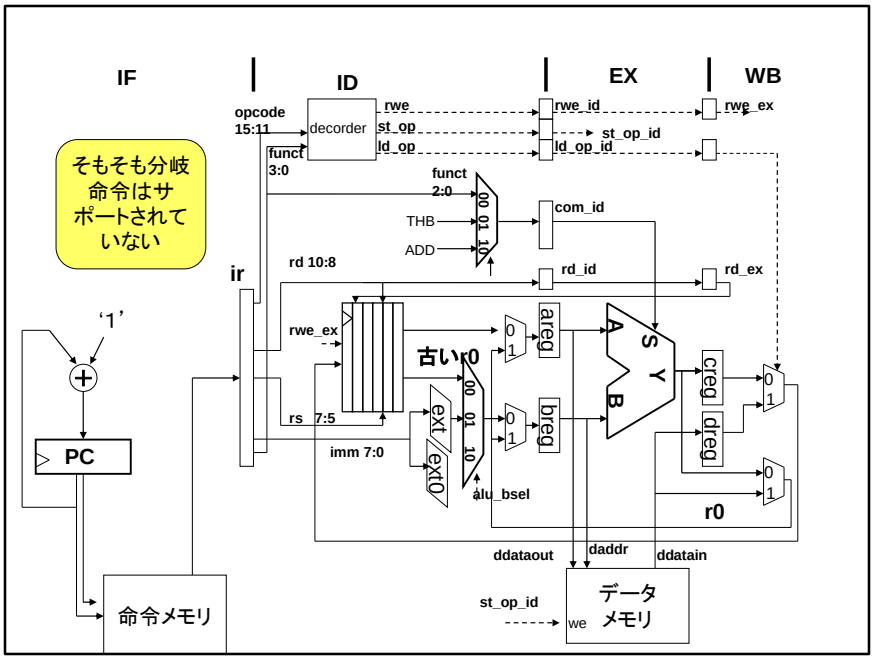
- RAWハザード：Read After Writeハザード
  - 書く前に読んでしまう。
  - 今回扱った最も一般的なハザード
- WARハザード：Write After Readハザード
  - 読む前に書いてしまう（書き潰し）
  - POCOではWBステージが最後なので起きない
- WAWハザード：Write After Writeハザード
  - これも書き潰しでPOCOでは起きない

今回取り扱ったハザードは、先行する命令が答えを書く前に読んでしまうことからRead After Write（書いた後に読む）のがうまく行ってないという意味でRAWハザードと呼びます。RAWハザードは命令間に普通の依存性があるときに生じるため最も本質的なハザードです。これに対してWARハザードは読む前に書いてしまうことによるハザードで、POCOではWBが最終ステージなので起きることはありませんが、早い段階で書くパイプラインでは発生します。このハザードは、実はレジスタの名前を変える（リネーミング）によって解決が付きます。WAWハザードはレジスタに関してはあまり一般的なハザードではなく、もちろんPOCOでは発生しません。

パイプラインハザードとは？

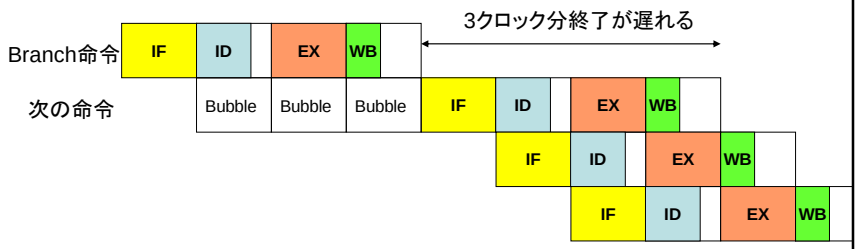
- パイプラインがうまく流れなくなる危険、障害のこと
  - 構造ハザード
    - 資源が競合して片方のステージしか使えない場合に生じる
  - データハザード
    - データの依存性により生じる
    - 先に進んだ命令の結果を後の命令が利用するため、その結果がレジスタに書かれるまで、読むことができない
  - コントロールハザード
    - 分岐命令が原因で、次に実行する命令の確定ができない
- パイプラインストール
  - ハザードが原因による性能の低下
  - パイプライン処理は理想的に動くとCPIが1
    - ストールによりCPIが大きくなってしまう

前回まで構造ハザード、データハザードを紹介しました。三つ目がコントロールハザードです。これは分岐命令が原因です。元々パイプラインは次に実行する命令が分かっているから流れるのであって、分岐命令があって次に実行する命令がどちらになるかが分からない場合にストールが起きるのは当然と言えます。



実を言うと、今まで分岐命令をパイプラインに組み込んでいませんでした。PCは毎クロック1づつ増やしていったため、パイプラインはスムーズに流れたわけです。しかし、実際にはそうは行きません。ではALUで飛び先を計算するとどうなるでしょう？この場合、飛び先が決まるのには3クロック掛かってしまいます。

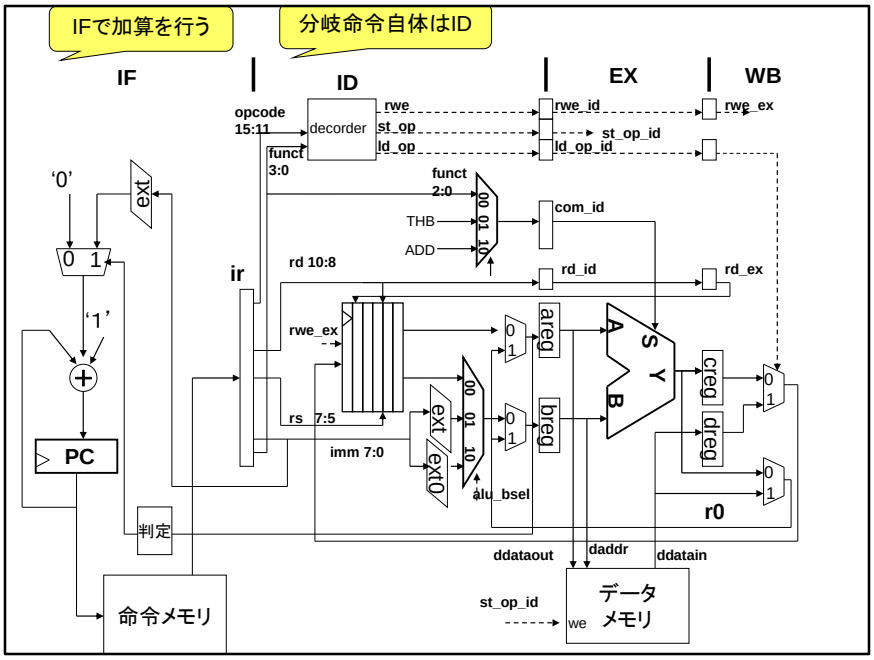
ALUで分岐先を計算すると、、、



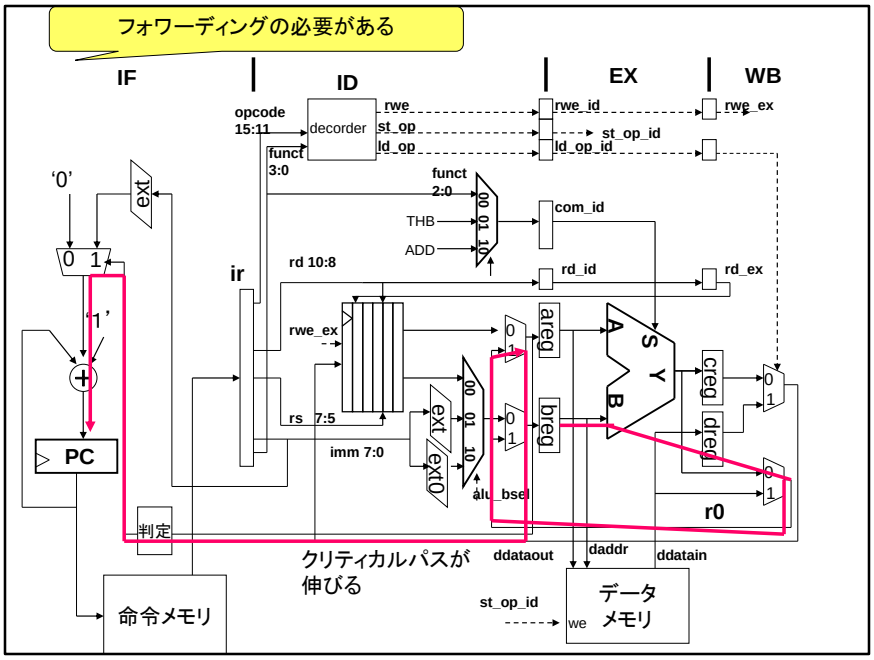
Branchの次の命令フェッチを3クロック遅らせる。

ストール付きCPI＝理想のCPI＋ストールの確率×ストールのダメージ
1　＋　0.25×3　=1.75
（Branch/JMP/JAL命令を合わせて25%とする）
ダメージが大きい！

次の命令を取ってこれるのは、飛び先のアドレスが決まった次のサイクルからになってしまいます。この場合は3クロック分バブルが入ってしまい、ストールのダメージは3クロックになります。分岐命令がフェッチされる確率はプログラムに大きく依存しますが、それなりの割合になることが多いです。Branch,JMP,JALの割合を合わせて25%と考えると、理想CPIの1が1．75になります。これは相当大きなダメージであると言えます。

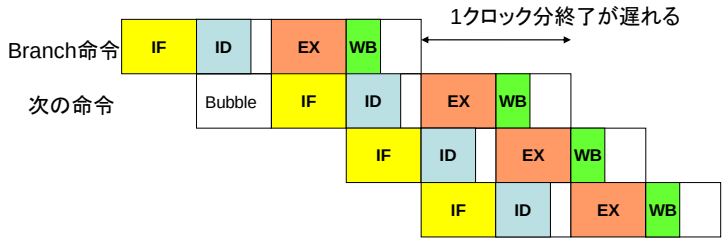


ダメージを減らすためには、分岐命令が成立するかどうか、成立する場合どこに飛ぶかを早めに判別する必要があります。命令がirに入った後、すなわちIDステージで分岐命令かどうかは分かります。このステージでレジスタファイルからレジスタが読み出され、分岐の条件がチェックされます。ir中にある飛び先をIFステージに送って加算を行い、分岐が成立する場合にはpcの内容を更新します。



分岐命令は、レジスタの内容で分岐するかどうかを判定します。このレジスタは直前の命令で更新される可能性があります。データハザードを避けるためには、直前の演算結果をフォワーディングする部分について判定が必要になります。この判定の結果で飛び先をpcに書き込むかどうかが決まるので、かなり長大なパスができてしまいます。これが、この方法の問題点です。

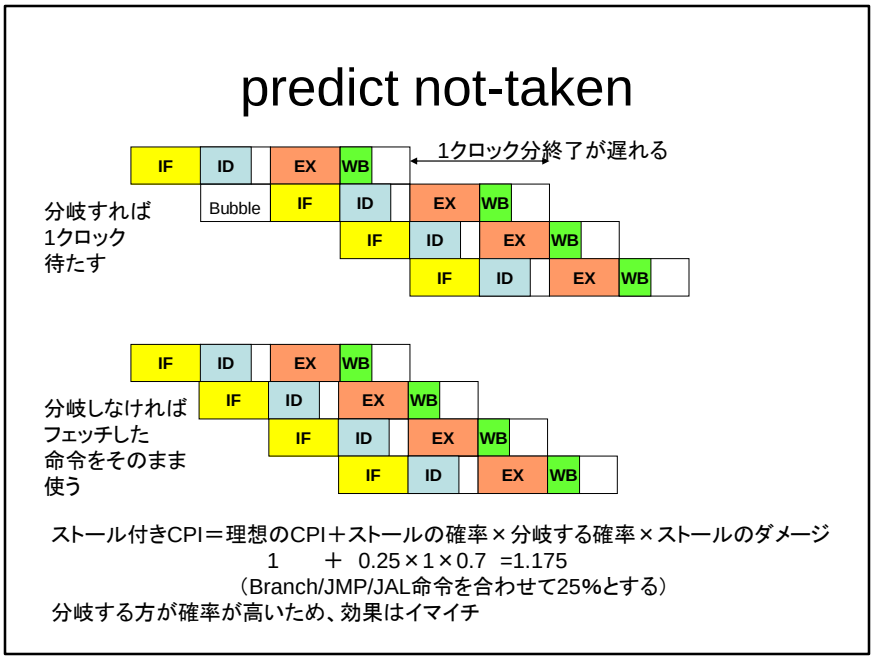
IDステージで分岐先を計算すると、、、



Branchの次の命令フェッチを1クロック遅らせる。

ストール付きCPI＝理想のCPI＋ストールの確率×ストールのダメージ
1　＋　0.25×1　=　1.25
（Branch/JMP/JAL命令を合わせて25%とする）
これ以上はどうにもならない

先のスライドの方法を使っても、分岐命令はIDステージで実行されることになるため、分岐命令の直後の命令は実行していいのかどうか分かりません。真面目にこれに対処するためには分岐命令の後に1クロックストールさせて、次の命令を1クロック遅らせてフェッチする必要があります。このようにすると、CPIが1から1．25に増えます。分岐命令の判定をこれ以上早く行なうことができないことを考えると、この性能低下は避けられないように思います。しかし、単純な方法が二つ考えられます。



一つの方法はpredict not-takenと呼び、分岐命令が成立した場合は1クロック待たす一方、分岐しなければフェッチした命令を捨てないで流してやる方法です。この方法を使うと、ストールするのは分岐命令が成立する場合のみになります。では、分岐が成立する確率はどの程度になるでしょう？実はこれは結構高く、ここでは7割と見積もりました。なぜ高いかと言うと、コンピュータはループを繰り返し実行しますが、ループにつき1回かならずアドレスが後方に戻っているはずだからです。したがってこの方法の効果はイマイチと言えます。

遅延分岐

- 分岐命令の次の命令（遅延スロット）をパイプラインに入れてしまう。
  - 遅延スロットの命令は必ず実行される
    →POCOの場合は遅延スロットは1
  - つまり、遅延の効き目が遅い
  - 有効な命令を入れてやる必要がある
  - → パイプラインスケジューリング
  - どうしても埋まらなければNOPを埋める

もう一つの方法は遅延分岐と言って、成立しようがしまいが、次の命令をパイプラインに入れて実行してしまう方法です。すなわち、分岐命令の直後の命令（これを分岐スロット内の命令と呼びます）は必ず実行されます。このことは言葉を変えると、分岐は1命令分効き目が遅いと考えることができます。このためこの方法を遅延分岐と呼びます。分岐スロットに有効な命令を入れてやることで、オーバーヘッドはなくなります。命令の順番を入れ替えて遅延スロットに有効な命令を入れることをパイプラインスケジューリングと呼びます。どうしても埋まらない場合はNOP命令で埋めてやります。

パイプラインスケジューリング

```
      LDIU r0,#2
      LD r1,(r0)
      LDIU r0,#3
      LD r2,(r0)
      LDIU r3,#0
loop: ADD r3,r1
      ADDI r2,#-1
      BNZ r2,loop
      NOP
      LDIU r0,#0
      ST r3,(r0)
end:  BEZ r2, end
      NOP
```

```
      LDIU r0,#2
      LD r1,(r0)
      LDIU r0,#3
      LD r2,(r0)
      LDIU r3,#0
loop: ADDI r2,#-1
      BNZ r2,loop
      ADD r3,r1
      LDIU r0,#0
      ST r3,(r0)
end:  BEZ r2, end
      NOP
```

2番地の内容と3番地の内容の掛け算を行なうプログラムで遅延スロットの埋め方を説明しましょう。元のプログラムは左のようにBNZの後にはNOPを入れます。NOPはバブルと同じなので、このプログラムはループ1回につき1クロック分オーバーヘッドが掛かることが分かります。ところがここにADD r3,r1を移動したらどうなるでしょう。一見右のプログラムにおいてADD r3,r1はループの外に出ているようですが、BNZは遅延分岐なので、直後のADD命令は必ず実行されます。結果として右と左のプログラムは同じ結果を出力します。しかし、ループ内にNOPが入らない右のプログラムの方がループを回る回数に相当するクロック数分速いことになります。p3kaiの下の例題プログラムを使ってちゃんと答えが出ることを確認しましょう。

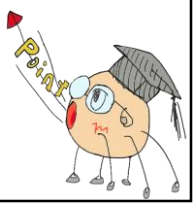
遅延分岐のPOCOのVerilog記述

```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if(pcsel)
    pc <= pc + {{8{imm[7]}},imm};
  else
    pc <= pc + 1;
end
…
assign pcsel = (bez_op & fwda == 16'b0) | (bnz_op &
   fwda != 16'b0) ;
```

では、このためのVerilog記述を紹介します。irにとってこられた命令が分岐命令かどうか、これが成立するかを調べます。成立すればpcselが1になります。条件を調べるレジスタはフォワードされた結果を使わなければならないのでfwdaを使うことに注意してください。後はpcselを1の時だけpcを飛び先に設定します。この記述はJMP、JR、JALを含んでいませんが、これは演習に譲ることにします。

本日のまとめ1

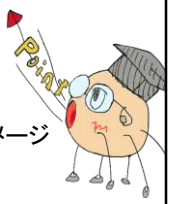
- パイプラインハザードは、パイプラインがうまく流れなくなる危険のこと
- 構造ハザードは資源の競合によっておきる
  - 資源の複製によって解決可能、コストとのトレードオフを考えて決める
- データハザードはデータの依存性によっておきる
  - 計算したばかりの結果を早いステージで横流しするフォワーディングで解決可能（POCOでは完全に解決できる）



インフォ丸が教えてくれる今日のまとめです。

本日のまとめ2

- コントロールハザードは、分岐命令によっておきる
- 分岐命令をIDステージで終わらせてしまう
  - IFステージに専用加算器が必要
  - クリティカルパスが延びる
- それでも1命令分バブルが生じる
  - predict not-taken
    - 分岐が成立したときだけストールさせる
  - 遅延分岐
    - 分岐命令直後の命令をパイプラインに入れて実行してしまう
    - 効き目が遅い分岐と考える
    - 直後の命令に有効な命令を埋められればオーバーヘッドはなくなる
    - うまく埋められなければNOPを埋める



理想CPI＋分岐命令の確率×遅延スロットが埋まらない確率×ダメージ

1＋　0．25×0．2×1＝1．05

たいした性能低下にはならない

インフォ丸が教えてくれる今日のまとめです。

演習

sum.asmをスケジュールし、遅延スロットを埋めて実行せよ。 提出物　sum.asm

p3kai.tarを利用のこと