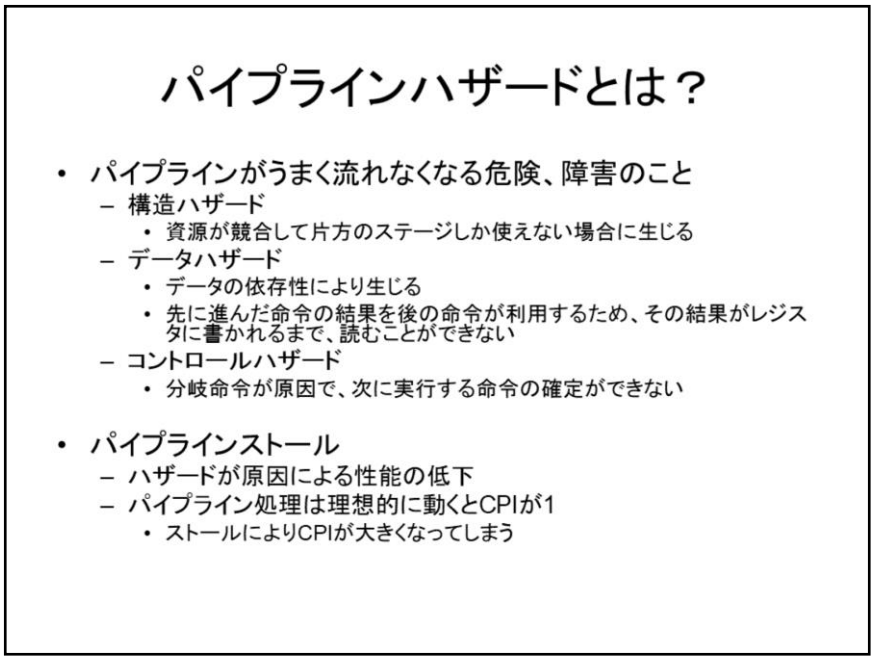
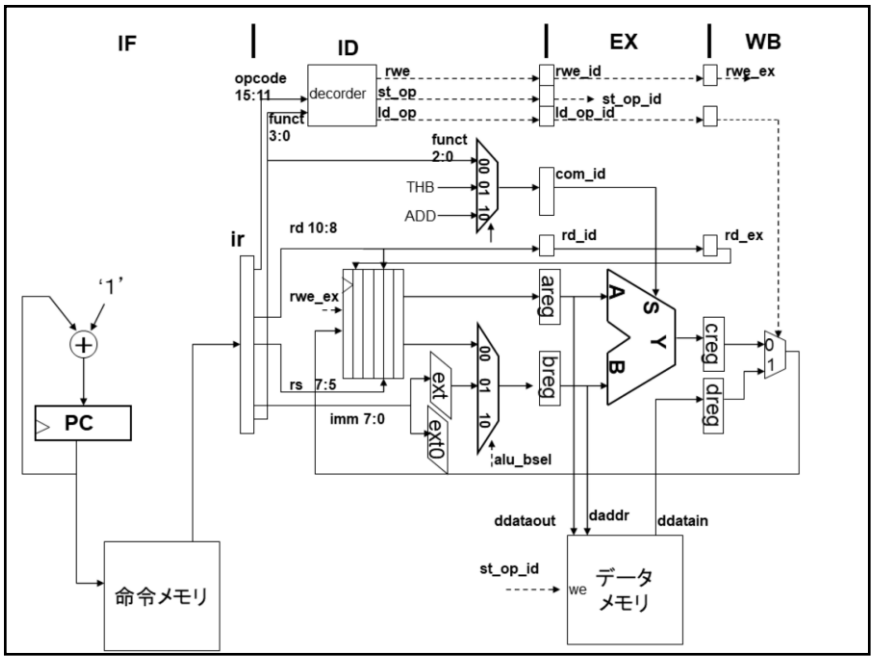
マイクロプロセッサ特論　第12回
パイプラインハザード
テキスト9章　115~124

情報工学科
天野英晴

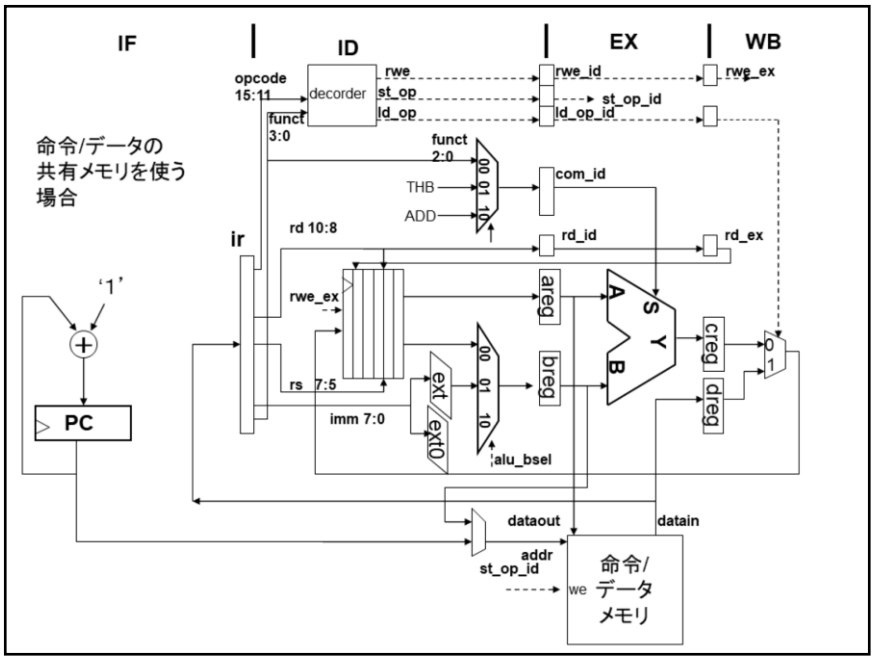
前回、パイプライン構造の基本を紹介しました。今回は、パイプライン設計における最大の壁となるパイプラインハザードを紹介します。



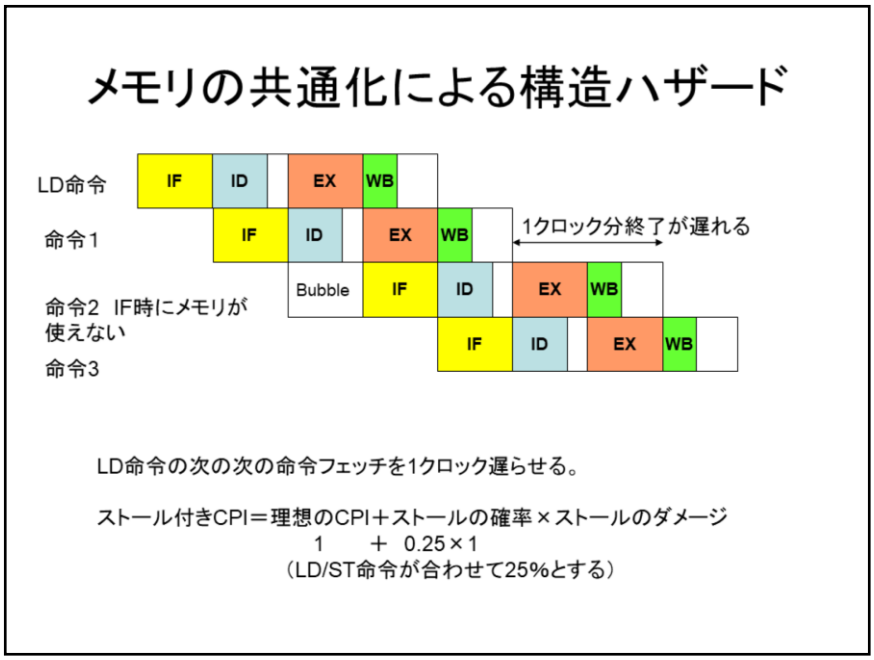
パイプラインは調子良く流れれば1クロックに1命令が終了します。しかし、場合によってはこれがうまく行かないことがあります。パイプラインがうまく流れなくなる危険、障害のことをパイプラインハザードと呼びます。ハザードには、資源の競合による構造ハザード、データの依存性により生じるデータハザード、分岐命令が原因のコントロールハザードの三つがあります。ハザードによりパイプラインがうまく流れなくなって性能が低下する現象をパイプラインストールと呼びます。



まず、最初に構造ハザードを紹介しましょう。構造ハザードは資源の共有によって生じます。今回のPOCOのパイプラインはそれぞれのステージがそれぞれの資源を占有しているために性能は低下しません。しかし、このため命令メモリとデータメモリは分離して持たなければなりません。



コストの関係上これが許されず、命令とデータを共用したメモリを用いる場合はどうでしょう？マルチサイクル版のようにアドレスにマルチプレクサを付けて共用メモリを実現することができます。



しかし、この場合、データメモリを読み書きするLDやST命令の実行時には、命令フェッチができなくなってしまいます。すなわち、2クロック後の命令が図のようにIFができないので、止めなければなりません。このような状態を水の流れの中にはいった泡に例えて、バブルと呼びます。バブルにより命令2の終了は通常よりも1クロック遅れることになります。これがストールです。

ストールの影響はCPIの増加となって表れます。これを見積もるには、理想のCPIにストールの確率×ストールのダメージを足してやります。ここではストールの確率はメモリアクセス命令の生じる確率で、ストールのダメージは1サイクルです。したがってこのストールによるCPIが25%増加することが分かります。

構造ハザード

- 資源の複製により解決可能
- コストと性能のトレードオフを考えて決める
  - メモリの共有化→コスト減を取るか？
  - CPI　1→1.25の性能低下を取るか？

構造ハザードは、資源を複製により解決可能です。すなわち、設計者がコストと性能のトレードオフを考えて決めます。この場合はメモリを共有化するコスト減を取るか、CPIが延びてしまうことを考えてやめるかを考える必要があるのです。

ちなみにメモリを複製するのは大変ですが、チップ内のキャッシュメモリを複製するのは楽なので、普通のプロセッサをキャッシュを分離して持たせることによりこの問題を回避しています。

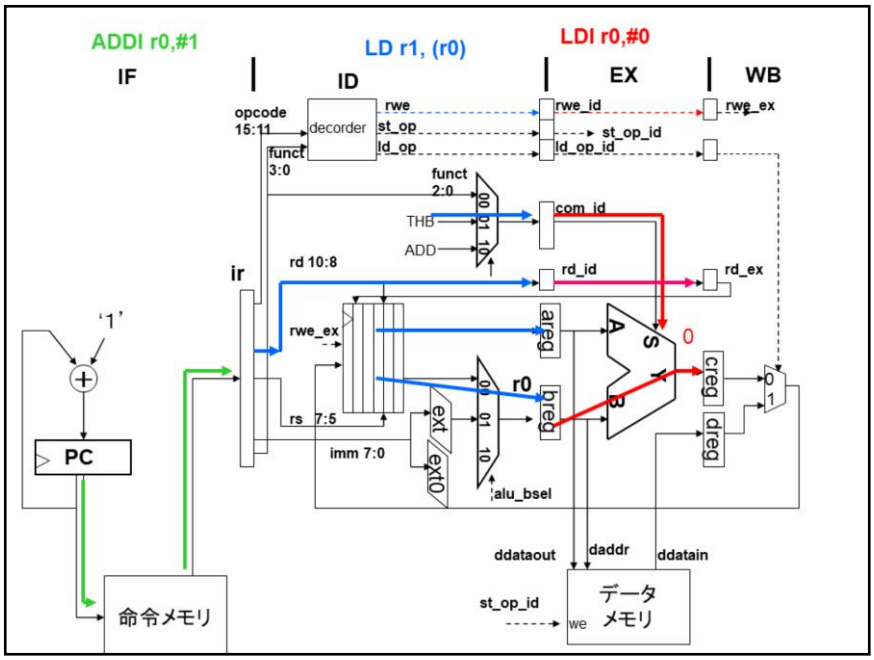
データハザード

- 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

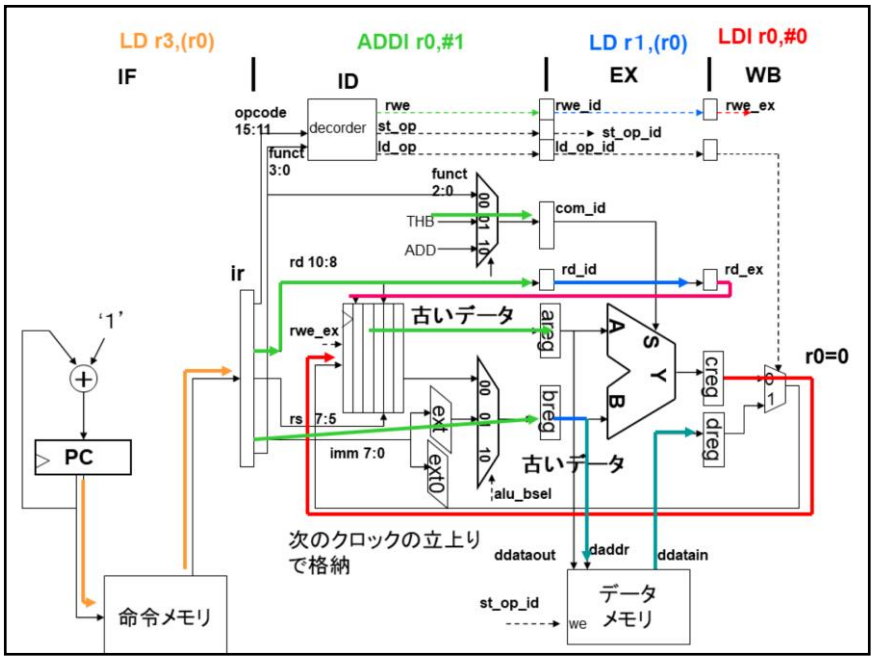
→ NOPを入れて命令の間隔を保持する

→ 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

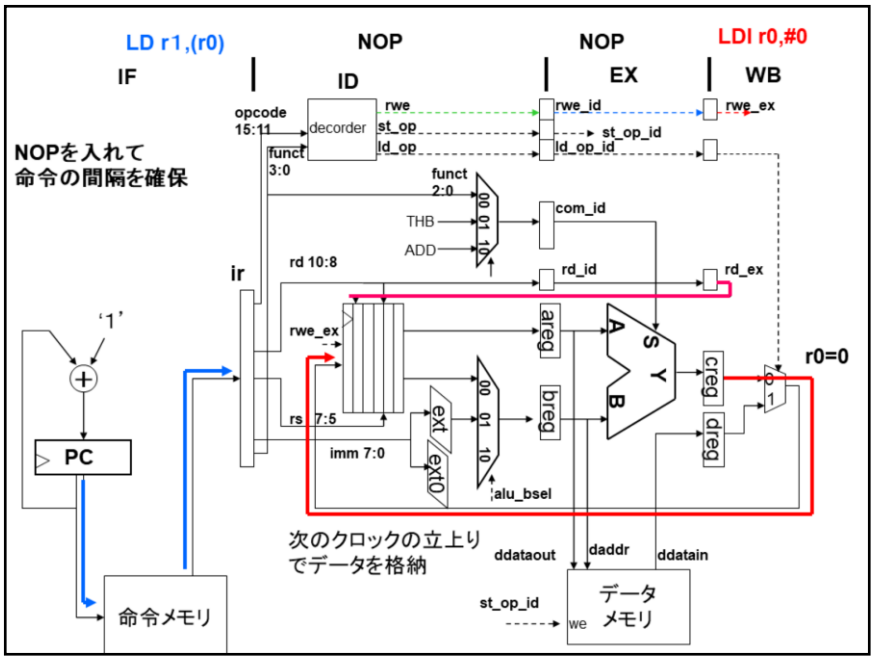
次にデータハザードを紹介します。データハザードは、直前の命令の結果がレジスタファイルに書き込まないうちに、後続の命令がこれを読みだすことにより起きます。相互に依存する命令を部分的に同時に実行しようとするために生じてしまい、パイプライン処理の本質的な問題点といえます。RAW、WAR、WAWの三つのハザードが考えられますが、POCOではRAWハザードだけが生じます。他のハザードについては後に説明します。



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。

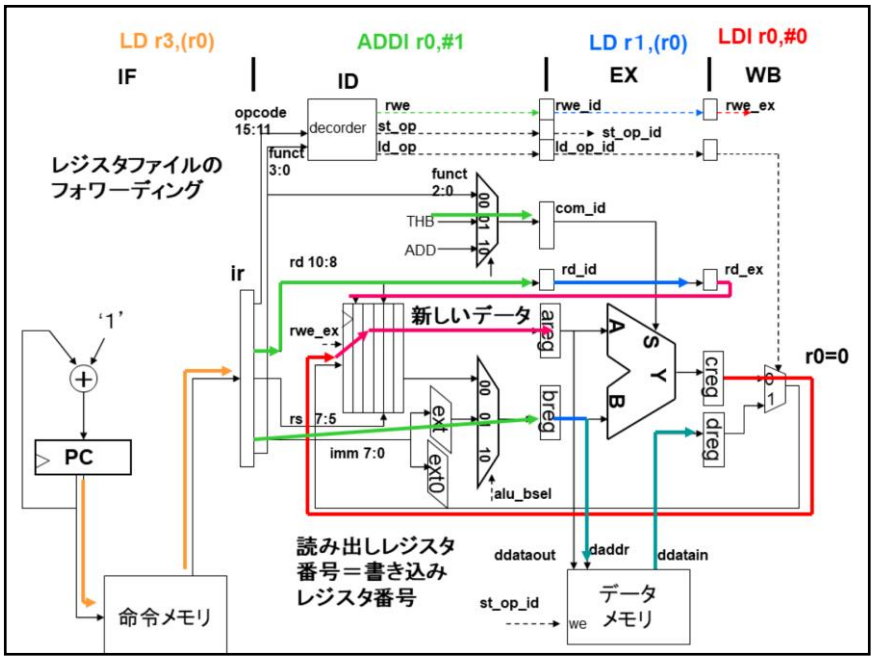
NOPを入れる方法

- 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。



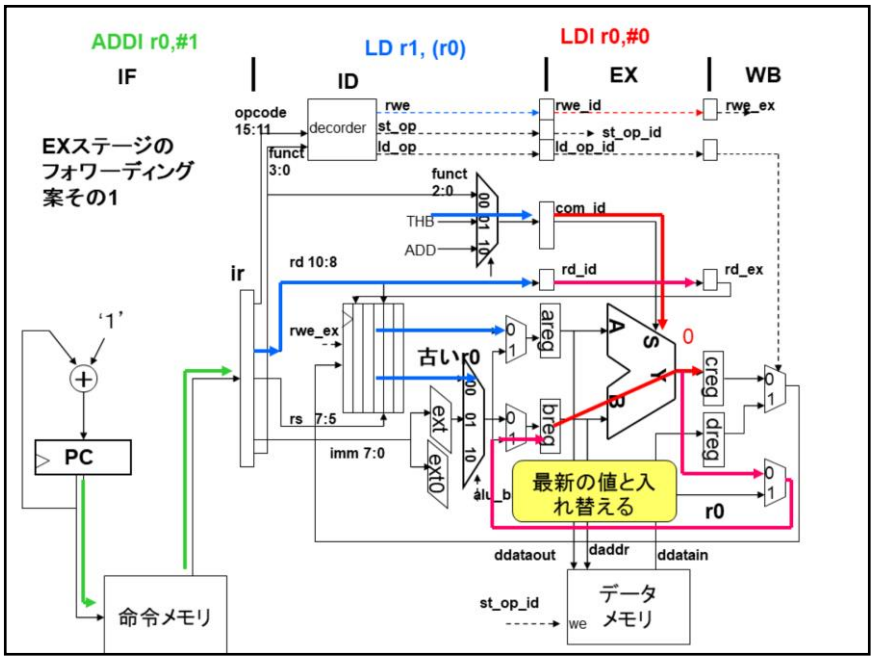
NOPを入れずにデータハザードを解決するためには、レジスタに書き込む前でも、用意のできた値を横流ししてやれば良いです。これをフォワーディングと呼びます。この例は、レジスタファイルにフォワーディングを付ける方法を示しています。この方法では、先行命令が、今書き込みつつある値を、IDステージにある命令が使う場合、つまり書き込むレジスタ番号と読み出すレジスタ番号が一致して書き込み信号rwe_ex=1になっている場合は、書き込む値をそのまま読み出すデータと入れ替えてareg, bregに入れてしまいます。この例ではaregの値が入れ替わります。これはWBステージからIDステージへのフォワーディングです。

レジスタファイルのフォワーディング Verilog記述

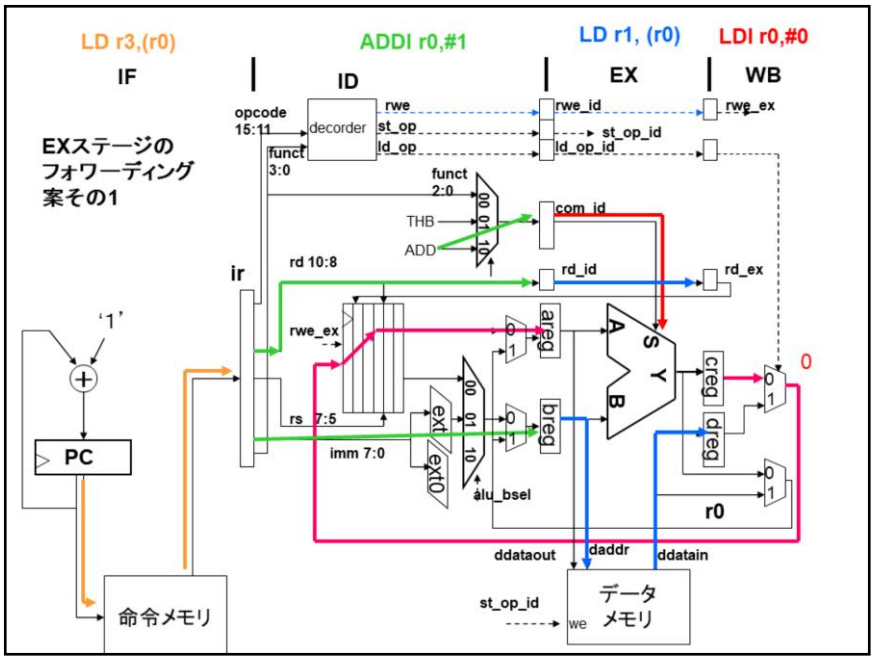
```
assign a =
        (aadr == cadr)&we ? c:
        aadr == 0 ? r0:
        aadr == 1 ? r1:
        aadr == 2 ? r2:
        aadr == 3 ? r3:
        aadr == 4 ? r4:
        aadr == 5 ? r5:
        aadr == 6 ? r6: r7;
assign b =
        (badr == cadr)&we ? c:
        badr == 0 ? r0:
        badr == 1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

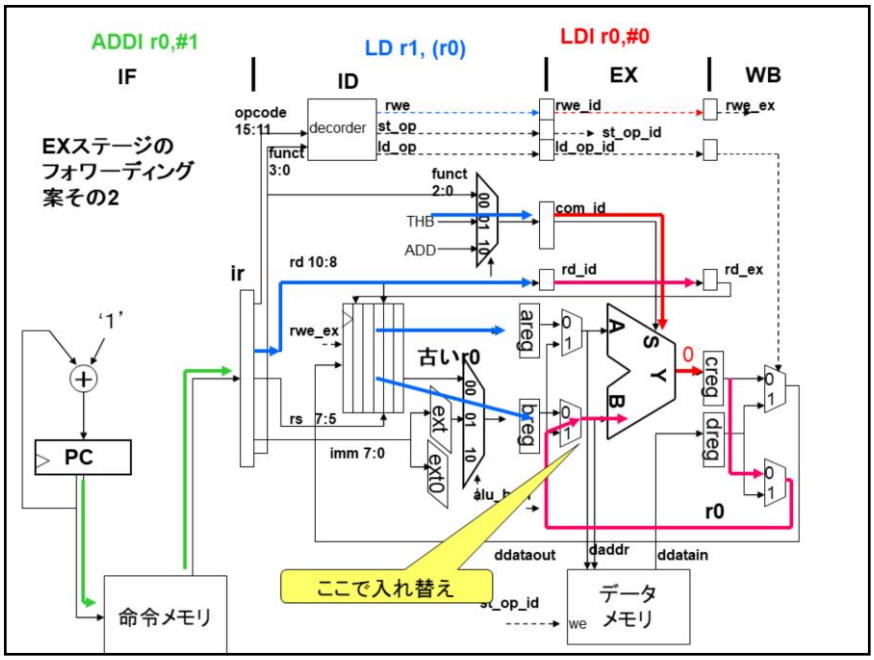
レジスタファイルのフォワーディングの記述は非常に簡単です。読み出しレジスタ番号と書き込みレジスタ番号が一致しており、かつwe=1(つまり書き込みが起きる)となっていれば、今書き込みつつあるc入力を直接a,bに横流しします。



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。

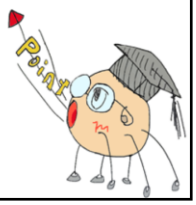
一般的なデータハザード

- 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。

本日のまとめ

- 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。

演習

フォワーディング付きパイプライン化POCOに
LDHI命令を付加せよ。

enshu.asmを実行してr0が0x5555になっていればOK

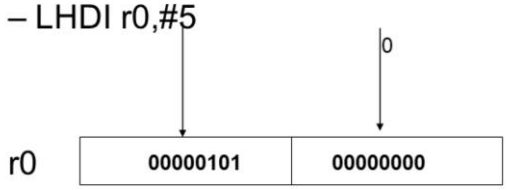
今回の演習は以前パイプライン化していないPOCOで実装した命令です。

LDHI（Load High Immediate)

大きな値をレジスタに直値で入れる命令

LDHI（opcode: 01010)

- 上位8ビットにImmediateの数字を入れ下位を0にする
- LHDI r0,#5



以前のスライドです。opcodeを01010にします。デコード部分の記述はもう付けてあります。この命令はI型です。この命令はセコイ感じもしますが、便利なので、全てのRISCが持っています。