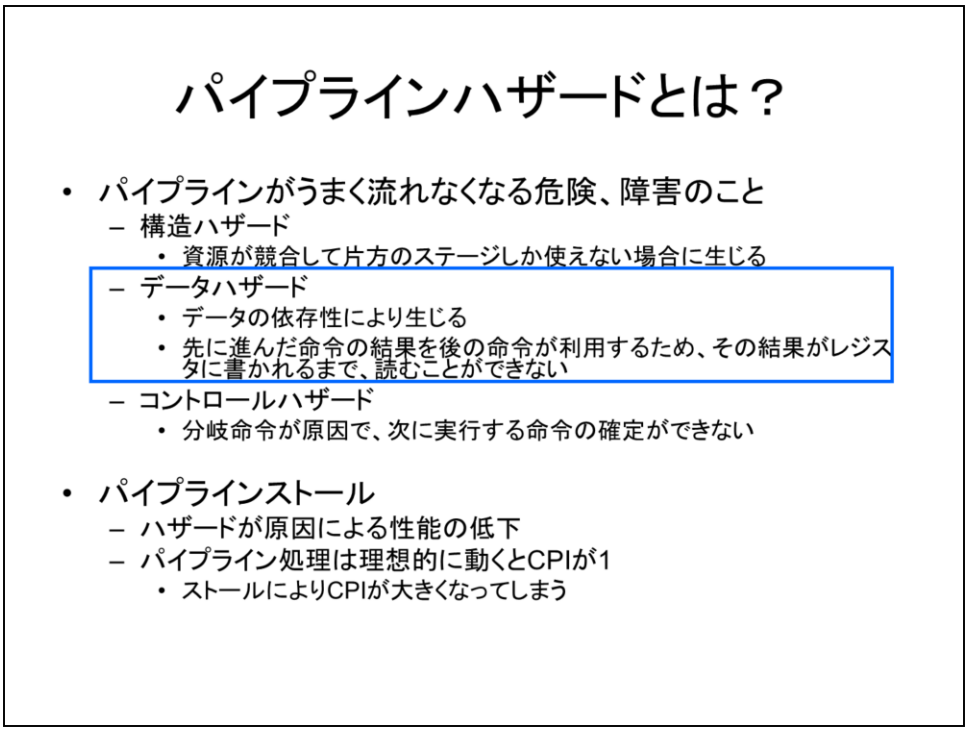
コンピュータアーキテクチャB
パイプラインハザード

天野　hunga@am.ics.keio.ac.jp

今回はパイプラインの動作を妨げるハザードとその対処法をやります。



前回紹介した構造ハザードは、資源の競合により起こるハザードで回避は簡単（というか複製しか手がない）でした。今回はハザードの中のハザード、データハザードを紹介します。

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MIPS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

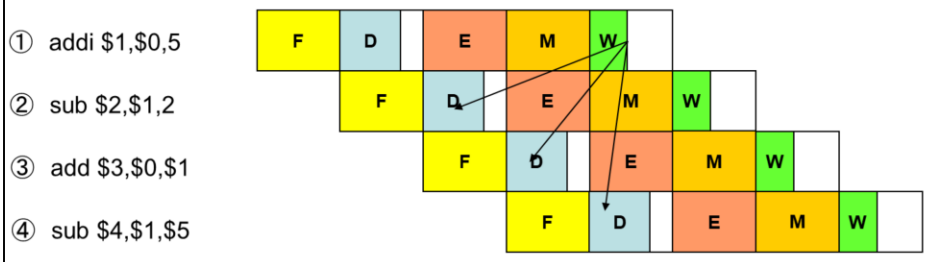
→ NOPを入れて命令の間隔を保持する

→ 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

パイプライン処理では、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うため、この命令間にデータの依存性があると、誤って更新前の値を読み出してしまいます。これを書き込む前に読んでしまうことからRAW(Read After Write)ハザードと呼ばれ、最も一般的なハザードです。他にもWARやWAWがあるのですが、MIPSではパイプラインの最後に結果を書き込むのでこれらは生じません。

RAWハザードを解決するには、命令間の間隔を保ってやれば良いのですが、これは本質的に性能を落とすことになります。もう一つ、最新の結果を横流しすることで、データハザードのロスを軽減することができます。

データハザード



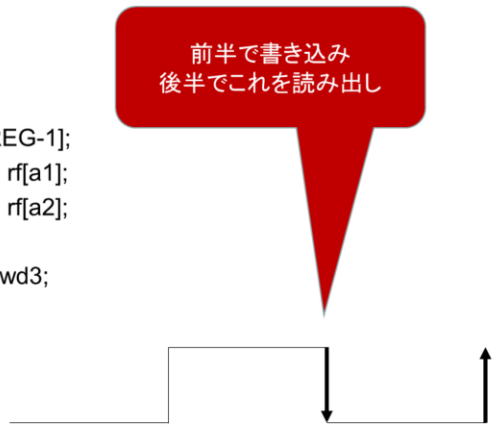
①の命令での書き込みの結果は②、③ではレジスタファイルから読み出せない過去の値を読んでしまう
④は工夫すれば読み出し可能

- レジスタファイルに書き込んだ値をスルーして出力
- レジスタファイルに前半で書き込み、後半で読み出す

データハザードの範囲を検討しましょう。Wステージで書き込みを行うので、②、③ではこの値が読めず、これ以前の値を読み出すことになります。④も書き込んだデータを読めるように工夫しなければ同様に以前の値を読んでしまうことになります。ここで、④は比較的容易に対処が可能です。レジスタファイルに書き込んだ値をそのまま読めれば良いので、書いた値をスルーして読めるようにするか、サイクルの前半で書いて、後半で読み出すようにするかを行います。

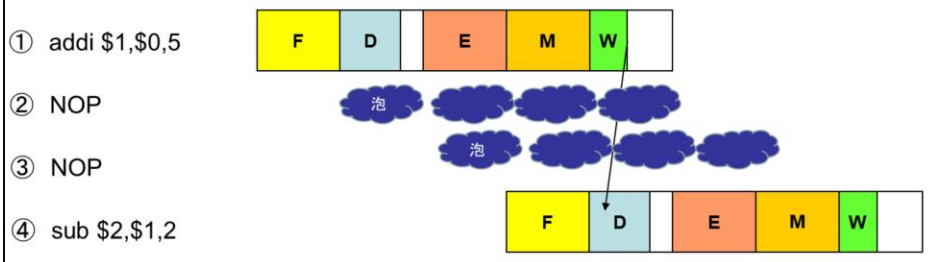
ネガティブエッヂを使うレジスタファイル

```
`include "def.h"
module rfile (
 input clk,
 input [`REG_W-1:0] a1, a2, a3,
 output [`DATA_W-1:0] rd1, rd2,
 input [`DATA_W-1:0] wd3,
 input we3);
        reg [`DATA_W-1:0] rf[0:`REG-1];
        assign rd1 = |a1 == 0 ? 0: rf[a1];
        assign rd2 = |a2 == 0 ? 0: rf[a2];
        always @(negedge clk)
                if(we3) rf[a3] <= wd3;
endmodule
```



この記述は後者のアプローチで、クロックが立ち下がった時にデータが格納されるようにします。この方法でクロックの前半で書き込み、後半で読み出しが行われます。後半の時間がクリティカルパスになり勝ちです。

データハザードの回避

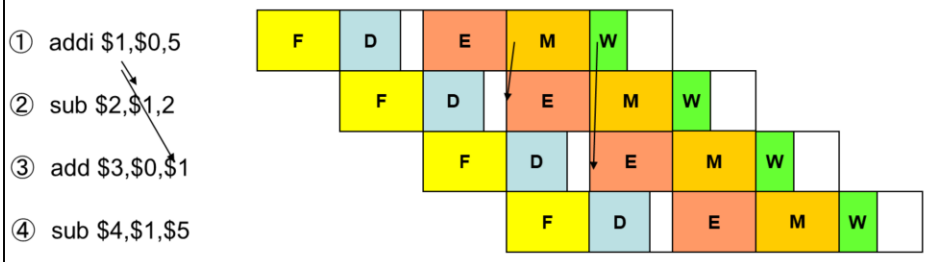


NOPを入れれば回避できる。しかし、これはパイプラインが激しくストールし
性能は大幅に低下する

→来週はフォワーディングを導入するが、結構面倒！
様々な命令のレジスタ間の組み合わせで生じるから…

⑤は回避できたので、それ以前の命令のデータハザードを回避するために、命令間の距離を取る方法を検討しましょう。この場合、二つNOPを入れれば回避できることが分かります。しかし、これはかなりの性能低下をもたらします。より現実的な方法は来週検討しましょう。

フォワーディングの原理



結果自体は、Eステージで計算が終わっている
この結果を横流して、Mステージの最初からEステージに横流しし、Dステージで読んだ古いレジスタと入れ替えれば良い。

注意：
- 命令はパイプを進んでいるので、Wステージの最初からのフォワーディングも必要
- rs, rtの両側に必要

さらに積極的にフォワーディングをするにはどうすれば良いでしょう？この例では①の命令の結果はEステージの終わりでは計算済です。これを次の命令のEステージの最初に送れば、計算可能になります。また、この命令がMステージを出た所で、次の次の命令のEステージに送ってやれば、③の命令も計算可能になります。

フォワーディング

- データの入れ替えはEステージのALUの前で行う
- Mステージの命令M（前の図の命令①）
- Eステージの命令E（前の図の命令②）
  - 命令Mの結果を書き込むレジスタ（rdかrt)が、命令Eのrt(rs)と一致
  - 命令Mが出力を書き込む命令である場合
    - swやbeqなどではない

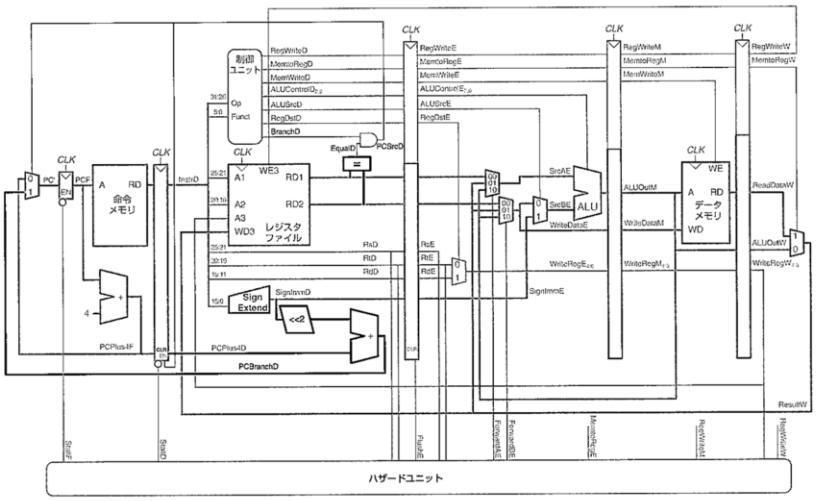
→　Mステージのパイプラインレジスタからのデータを Eステージの最初のデータと入れ替える

全く同じことをWステージの命令でも行う

ここでは、データの入れ替えは基本的にEステージのALUの直前で行います。これは、先行命令の結果を書き込むレジスタ（rdかrt)がEステージの命令のrt(rs)

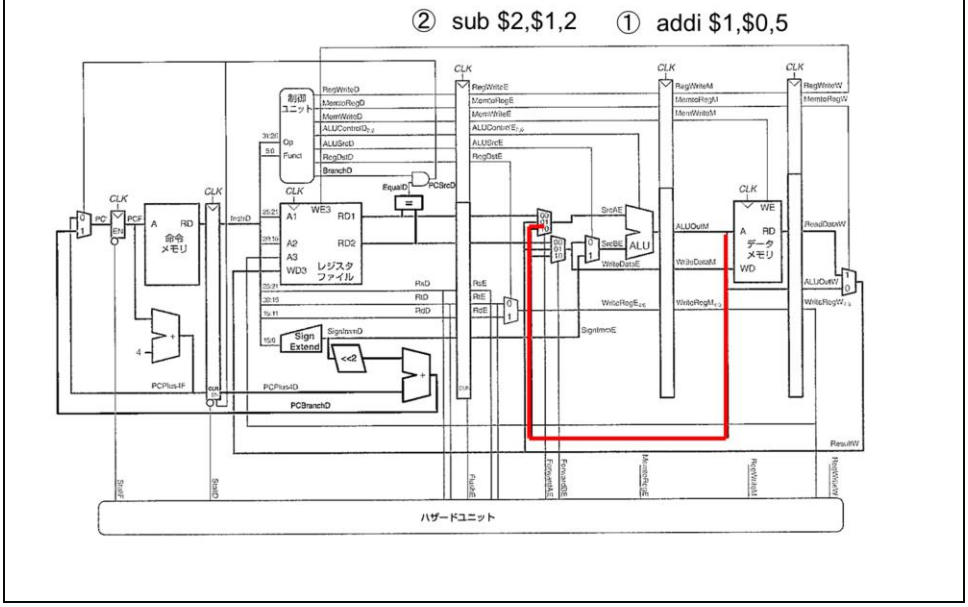
と一致することが必要で、かつ先行命令がレジスタファイルに書き込みを行う命令であることが必要です。

フォワーディング用のマルチプレクサ



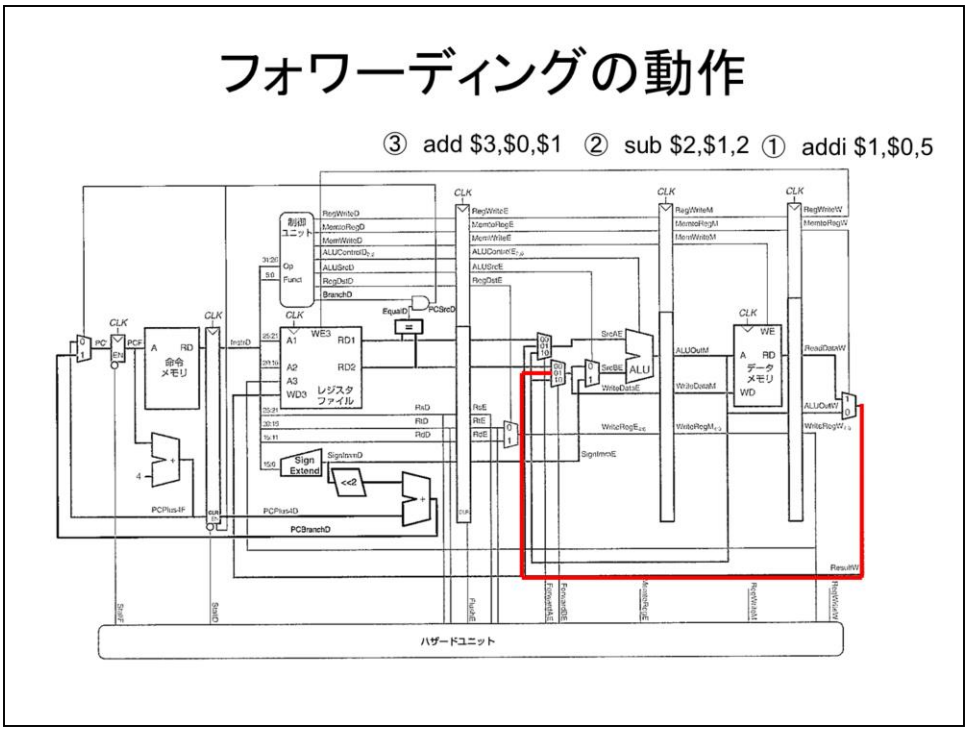
このためにALUの入力にフォワーディング用のマルチプレクサを付けます。

フォワーディングの動作



このマルチプレクサに対して条件が成立した場合の計算結果をフィードバックします。
これは、命令①から命令②へのフィードバックです。

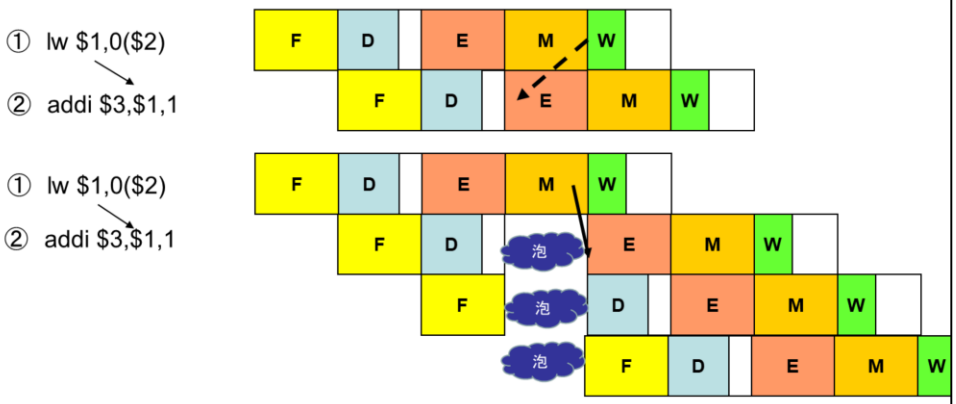
フォワーディングの動作



同様のフォワーディングはWステージからも行います。両方からのフィードバックが必要な場合、Mステージを優先します。

Load命令のデータハザード

- フォワーディングでは回避不能
  - まだメモリから取ってきていないデータをフォワーディングできない



通常の計算データはこの方法でフォワーディング可能です。しかし、Load系の命令lw, lb, lbuではこれだけでは十分でないです。この命令では、答がMステージの終了後でなければ得られないためです。このため次の命令でこの結果を利用する場合、どうしても1サイクル分のバブルを入れてパイプラインを待たせてやる必要があります。

パイプラインインターロック

- Dステージでチェックする
- EステージのLoad命令（前の図の命令①）
- Dステージの命令D（前の図の命令②）
- Load命令の出力レジスタrtが命令Dのrsまたはrtと同じ場合
  - MとWは実行を続行
  - F,D,Eは実行を停止

この待たせる操作をパイプラインインターロックと呼びます。これを実現するにはまず、Dステージでチェックをし、EステージのLoad命令の読んできた結果が、Dステージで利用される場合、MとWは実行を続け、F,D,Eは実行を停止します。これをパイプラインインターロックと呼びます。

コードスケジュール

- C=A+B, F=D+Eを実行する場合

```
lw $1,A($0)
lw $2,B($0)
add $3,$1,$2
sw $3,C($0)
lw $1,D($0)
lw $2,E($0)
add $3,$1,$2
sw $3,F($0)
```

ストール

ストール

2箇所ストールする

パイプラインインターロックは命令コードの実行順を入れ替えることで対処できます。例えば、例題のコードを実行する場合、普通にプログラミングすると2か所ストールしてしまいます。

コードスケジュールでストールを減らす

- C=A+B, F=D+Eを実行する場合

```
lw $1,A($0)
lw $2,B($0)
lw $4,D($0)
add $3,$1,$2
lw $5,E($0)
sw $3,C($0)
add $6,$4,$5
sw $6,F($0)
```

しかし、処理の順番を入れ替えることで、ストールは0にすることができます。これをコードスケジュールと呼びます。

フォワーディングのデータパスのVerilog記述

A入力のマルチプレクサ

```
assign srcaE =
 regwriteM & rsE!=0 & writeregM == rsE  ? aluoutM :
  regwriteW & rsE!=0 & writeregW == rsE ? resultW : rd1E;
```

B入力のマルチプレクサ

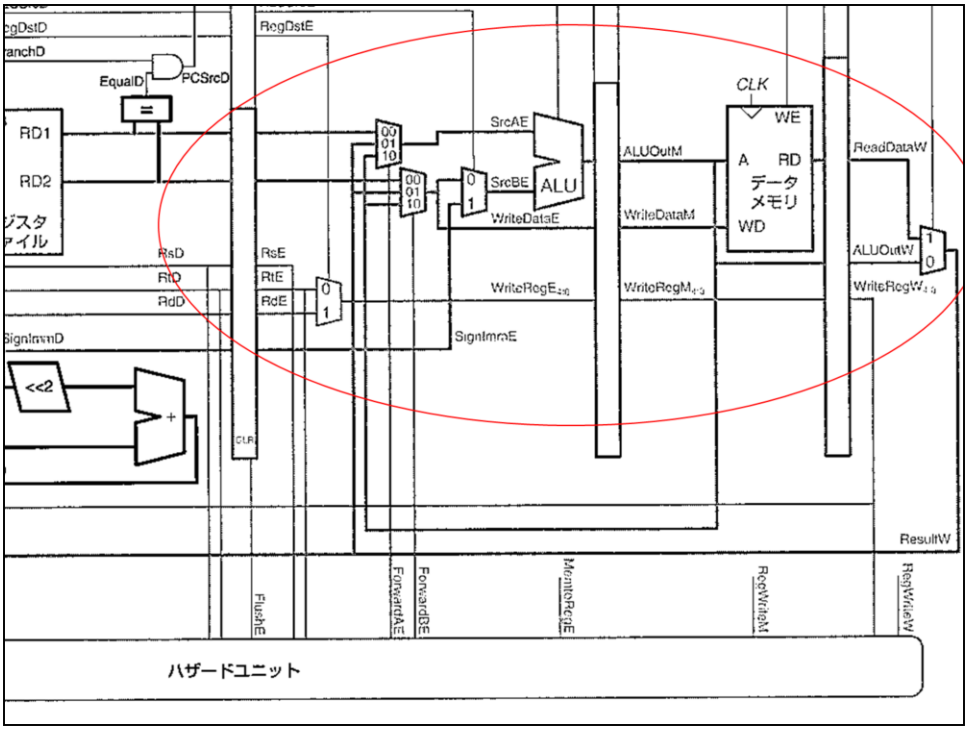
```
assign writedataE =
  regwriteM & rtE!=0 & writeregM == rtE ? aluoutM :
  regwriteW & rtE!=0 & writeregW == rtE ? resultW : rd2E;
assign srcbE = alusrcE ? signimmE : writedataE;

assign writeregE = regdstE ? rdE: rtE;

alu alu_1(.a(srcaE), .b(srcbE), .s(alucomE), .y(aluoutE));
```

図を対応させて理解しよう

ではフォワーディングのVerilog記述を紹介します。ALUのA,Bそれぞれのマルチプレクサを拡張します。図と対応させて理解しましょう。



やや拡大した図です。Verilog記述と対応させてください。

パイプラインインターロックのVerilog記述

Dステージで判定を行う

```
assign lwstall = ( (rsD == rtE) | (rtD == rtE) ) & MtoregE ;

ifステージはこの信号でパイプラインを止める
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) instrD <= 0;
  else if(!lwstall) instrD <= instr;
end
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if(!lwstall)    pc <= pc+4;
end
```

次にパイプラインインターロックのVerilog記述を紹介します。Dステージで判定を行い、Fステージはこの信号lwstallでパイプラインを止めます。

Dステージのパイプラインレジスタも止めてやる

```
always @(posedge clk) begin
  if(!lwstall) begin
        rd1E <= rd1D;
        rd2E <= rd2D;
        alucomE <= alucomD;
        alusrcE <= ~alu_opD;
        regdstE <= alu_opD;  end
end
```

以下、同様にsignimmE、制御信号もDステージのパイプライン
レジスタへの記憶は全てストップする。

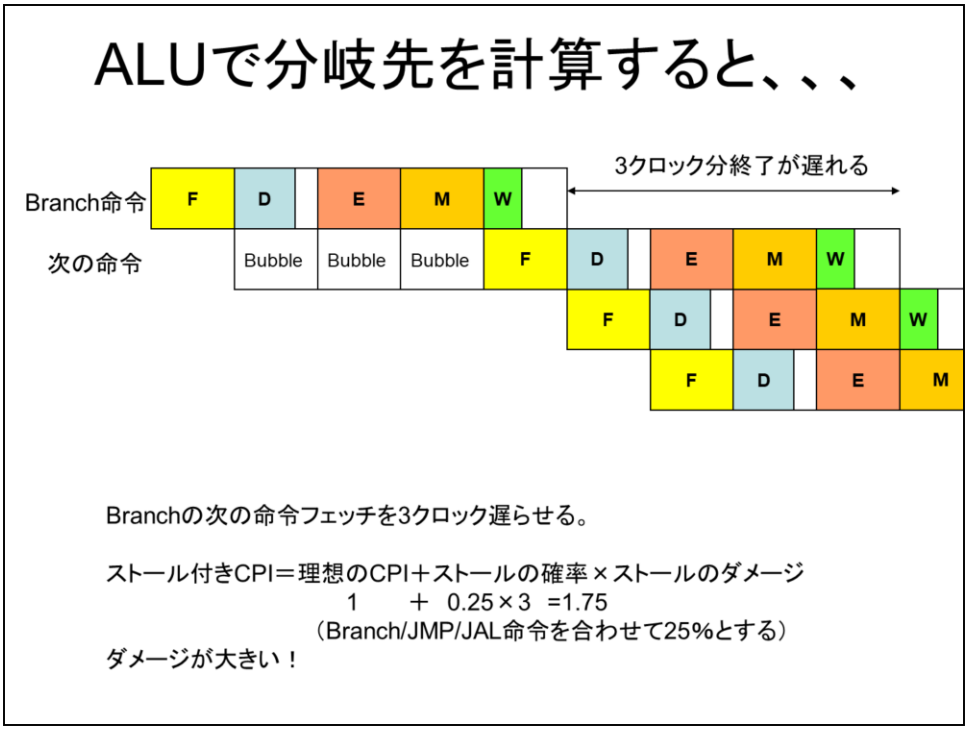
Eステージ以降はインターロックさせない

Dステージも同様にしてパイプラインを止めます。一方、Eステージ以降はこのような
インターロックをさせません。

パイプラインハザードとは？

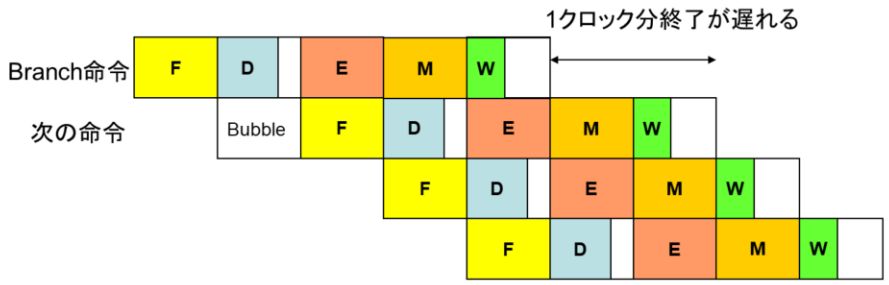
- パイプラインがうまく流れなくなる危険、障害のこと
  - 構造ハザード
    - 資源が競合して片方のステージしか使えない場合に生じる
  - データハザード
    - データの依存性により生じる
    - 先に進んだ命令の結果を後の命令が利用するため、その結果がレジスタに書かれるまで、読むことができない
  - コントロールハザード
    - 分岐命令が原因で、次に実行する命令の確定ができない
- パイプラインストール
  - ハザードが原因による性能の低下
  - パイプライン処理は理想的に動くとCPIが1
    - ストールによりCPIが大きくなってしまう

最後のハザードがコントロール（制御）ハザードです。これは分岐命令が原因で次に実行する命令の確定ができないことから生じます。



ALUで分岐先を計算させるとしましょう。Eステージの後のMステージでPCが更新され、次のクロックからそれに従ってフェッチされます。これだ3クロック分次の命令の始まりが遅れ、パイプラインの性能計算の式に基づくと、分岐系の命令が合わせて25%と仮定すると、CPI=1が1.75になってしまいます。これはちょっとダメージが大きいです。

Dステージで分岐先を計算すると、、、

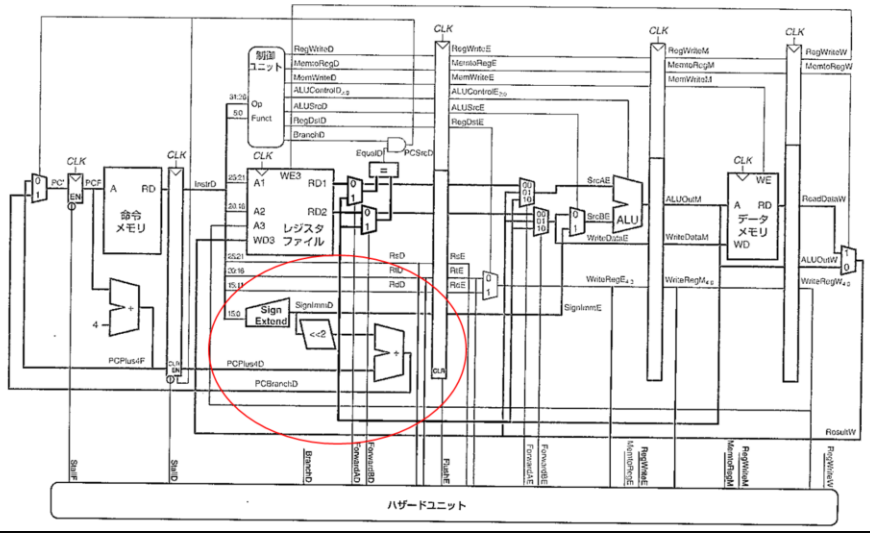


Branchの次の命令フェッチを1クロック遅らせる。

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

Fステージではそもそも命令をまだ取って来てないので、最速で分岐先を計算するのは、Dステージで計算および判断をやって、次のステージに分岐後の命令を取ってくることです。この方法ではALUが使えないので、専用の加算器が必要ですがダメージが1サイクルになります。分岐命令と分かったら次に命令を取ってくるのを止めて、1クロック待って（バブルが入る）、次のクロックに正しい命令を取ってきます。この場合、1クロックのダメージがあるので、分岐命令の確率を25%とすると、CPIは1から1.25になります。

Dステージでの分岐アドレスの計算と分岐の判定



では、このための仕組みを考えます。Dステージに飛び先計算と、飛ぶかどうかを判定するハードウェアを入れてやります。飛び先の計算は加算器に入れる前にシフトが必要です。分岐の判定はレジスタ同士が等しいかどうかをしれべれば良いので簡単です。

分岐判定用レジスタのデータハザード

ケース①

addi $1,$1,-1

add $2,$3,$4

beq $0,$1,loop Mステージからのフォワーディングが必要

ケース②

addi $1,$1,-1

beq $0,$1,loop

- ALU計算直後からのフォワーディングが必要
  - この方法はクリティカルパスを伸ばしてしまうため、インターロックする

問題は、分岐の判定を早い時期に持ってきたことで、判定するレジスタに対してデータハザードが生じてしまうことです。これはMステージからとEステージからの二つを考慮する必要があります。両方ともレジスタ番号が一致して先行命令がレジスタに書き込む命令で、後続命令が分岐命令の時フォワーディングが必要になりますが、直前からフォワーディングをすると、クリティカルパスが延びてしまうので、ここではインターロックをすることにします。

分岐判定用レジスタのデータハザード

ケース③

lw $1,$1,4($0)

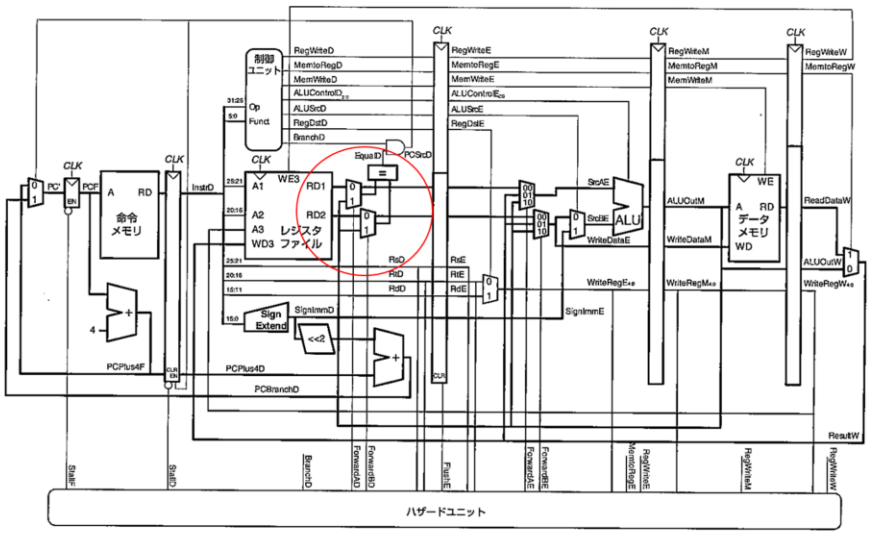
add $2,$3,$4

beq $0,$1,loop

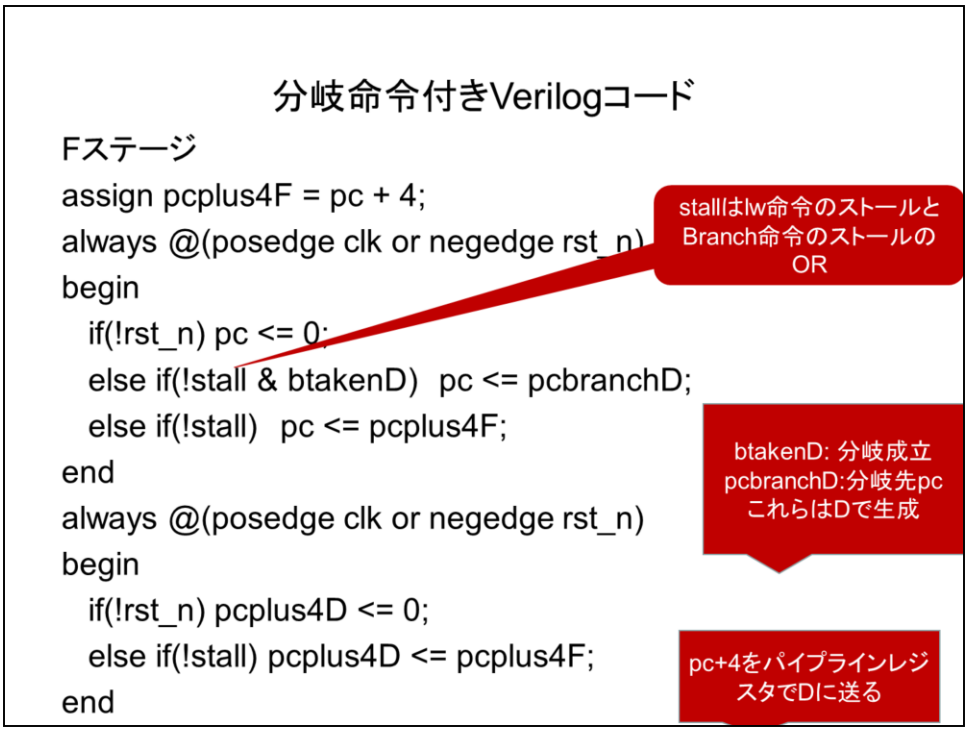
Mステージからのフォワーディングでは間に合わない（読み出されたデータが使えるのはWステージの最初）ので、インターロックの必要がある。

また、lw命令は結果が使えるのはMステージの後なので、これもインターロックの必要があります。

Mステージからのフォワーディング



Mステージからのフォワーディングを行うためにマルチプレクサをレジスタファイルの出力に付けてやります。



それではVerilogコードを見てやりましょう。パイプラインハザードの対処はステージ間をまたがるので、慎重に考えて信号名を間違えないようにしましょう。ストールしない場合で、分岐が成立すれば、pcに飛び先をセットし、そうでなければpc+4をpcにセットします。これとは別にpc+4は次のステージに送ってやる必要があります。

Dステージでの処理

```
// 成立の判定と飛び先の計算、フォワーディング後のレジスタを使うのに注意！
assign btakenD =
    beq_opD & (rd1fD == rd2fD) | bne_opD & (rd1fD != rd2fD);
assign pcbranchD = pcplus4D + {signimmD[29:0],2'b00};
// Mステージからのフォワーディング
assign rd1fD =
  (rsD !=0) & (rsD == writeregM) & regwriteM  ? aluoutM: rd1D ;
assign rd2fD =
  (rtD !=0) & (rtD == writeregM) & regwriteM  ? aluoutM: rd2D ;
```

ではDステージでの処理です。分岐命令が成立するかどうかはフォワーディングのマルチプレクサを含めての記述です。条件が少しややっこしいです。分岐の飛び先はFステージからのPC+4に飛び先をシフトした値を足します。ここで専用の加算器を使います。

ここで使うレジスタには、Mステージからのフォワーディングを行う必要があります。

パイプラインインターロック

```
// lw命令用
assign lwstall = ( (rsD == rtE) | (rtD == rtE)) & MtoregE;
// Eステージとのデータハザード
assign branchstall = (branchD & regwriteE &
                (writeregE==rsD | writeregE==rdD)) |
// Mステージのlw命令とのデータハザード
(branchD & MtoregM &
                (writeregM==rsD) | (writeregM==rtD));
// どちらもストール
assign stall = lwstall | branchstall;
```

次はパイプラインインターロックの説明です。lw命令の次の命令がそれを使う時、これがデータハザードによるインターロックでlwstallという信号名を使っています。分岐命令の方はbranchstallという名前になっていて、Eステージの命令の結果が次の分岐命令の判断に使う時、Mステージのレジスタを分岐命令で使う時に、パイプラインを止めています。これらのインターロックは、命令スケジューリングによって回避できます。

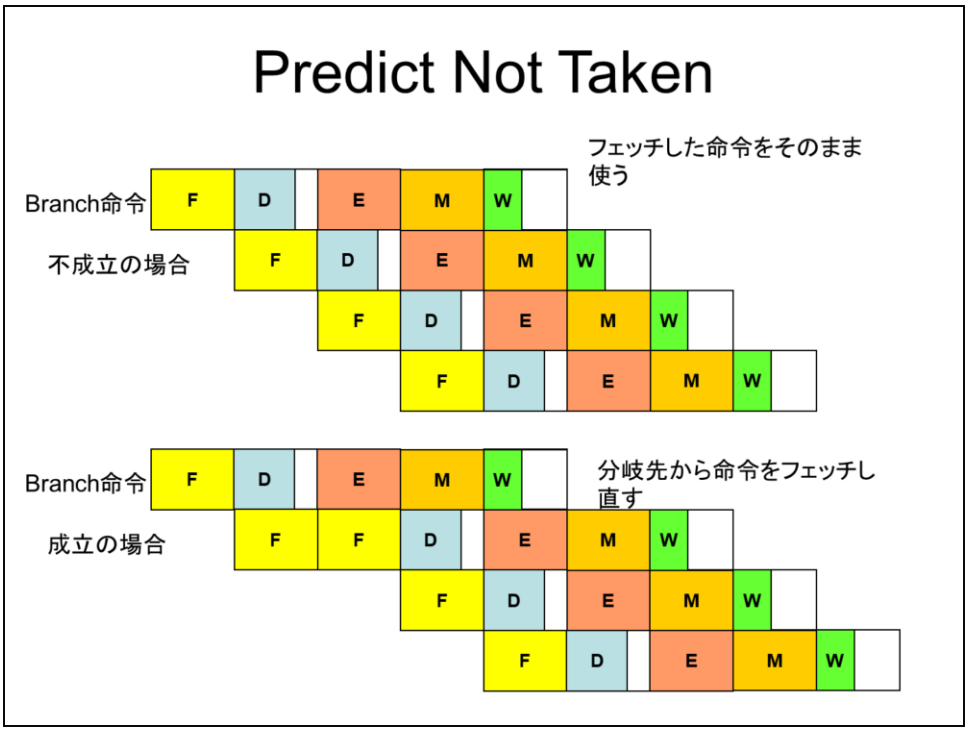
分岐命令のストール回避のアイディア

- 分岐命令の次の命令は捨てなければならない
- 1クロックのストールが生じる
- 簡単な対処法を2つ紹介
  - Predict Not Taken　成立しないと予測する
    - 最も簡単な分岐予測
    - 分岐が不成立ならばそのまま実行する。そうでなければNOPに変更
  - 遅延分岐
    - 何もしないで、分岐の効き目が遅いことにする
    - コードスケジュールと組み合わせる

このパイプラインでは、分岐命令の次の命令はフェッチしてきても捨てなければならず、1クロックのストールが必ず生じます。これを低減するための簡単な方法を二つ紹介します。

一つは、Predict Not Takenという方法で、「分岐命令が常に分岐しない」と予想する一種の分岐予測です。予測がはずれて分岐が成立すると分岐命令をNOPに変更してパイプラインに流します。これはバブルとなってダメージとなりますが、分岐が不成立ならば、フェッチしてきた命令をそのまま使うことができてロスが生じません。この方法は簡単な付加ハードウェアで性能が向上しますが、不幸なことに分岐命令は成立する場合の方が多いので、思ったより効果が得られません。

もう一つの方法は、遅延分岐（Delayed Branch)といって、ハードウェアは何も変更せずに、取ってきた命令をパイプラインに流してしまいます。そして「この分岐命令は一命令分効き目が遅いんだ」と解釈します。このパイプラインに流してしまう命令の場所を遅延スロットと呼びます。



この図はPredict Not Takenを示しています。成立の場合のみ命令をフェッチしなおします。

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
    →MIPSe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
- 今回のMIPSeはこの方法を採用している

遅延分岐は、分岐命令の次の命令をパイプラインに入れてしまい、必ず実行する方法です。すなわち分岐命令の効き目が遅いと考えるのです。パイプラインスケジュールによって、有効な命令を入れてやることができれば、この命令は無駄にはならないです。どうしても有効な命令が入れられない場合、NOP命令を入れておきます。これはロスになってしまいます。

掛け算のプログラムの例：mult.asm

```
        lw $1,0($0)
        lw $2,4($0)
        add $3,$0,$0
loop:   add $3,$3,$2
        addi $1,$1,-1   // ここはデータハザードでストールする
        bne $1,$0,loop
        add $0,$0,$0    // NOP   ここを埋めてみるには？
        sw $3,8($0)
end:    beq $0,$0,end
        add $0,$0,$0    // NOP ここはダイナミックストップなの
で気にしないでよい
```

mult.asmの例を考えましょう。この分岐は遅延分岐で、NOPが入って正常に動いています。では、このNOPを有効な命令で埋めるにはどうすれば良いでしょうか？

コードスケジューリング後 mult2.asm

```
        lw $1,0($0)
        lw $2,4($0)
        add $3,$0,$0
loop:   addi $1,$1,-1   // ここはデータハザードでストールする
        bne $1,$0,loop
        add $3,$3,$2    // 遅延スロット
        sw $3,8($0)
end:    beq $0,$0,end
        add $0,$0,$0    // NOP ここはダイナミックストップなの
で気にしないでよい
```

add命令を持ってきた例です。このコードは一見ものすごく変に見えますが、bneが遅延分岐ならばちゃんと動きます。

コードスケジューリング後:mult3.asm

```
        lw $1,0($0)
        lw $2,4($0)
        add $3,$0,$0
        addi $1,$1,-1   //あらかじめ引いておく
loop:   add $3,$3,$2
        bne $1,$0,loop
        addi $1,$1,-1  //遅延スロット
        sw $3,8($0)
end:    beq $0,$0,end
        add $0,$0,$0     // NOP ここはダイナミックストップなの
で気にしないでよい
```

もう一つ、制御変数の$1をカウントダウンする命令を使う方法もあります。この場合は、インターロックを減らす効力もあります。しかし、命令の実行順は変わらないため、あらかじめ一つ引いて置く工夫が必要になります。

演習6

- addarray.asmは0番地から並んだ8つの数の総和を求めるプログラムである
- パイプラインスケジューリングを行い、lw命令の後のストール、遅延分岐によるストールをなるべく小さくするようにスケジューリングせよ
- 提出物：改造後のaddarray.asm

パイプラインのまとめ

- 遅延分岐のスロットが埋まらない確率を20%
- ロード命令の後にこれを使わない命令をスケジュールできない可能性が20%とすると、

CPI＝1＋25%×20%＋12%×20%=1.074

クリティカルパスが伸びないとすれば、他の方法よりも圧倒的に有利

- 実際、組み込み用5段パイプラインはうまく行く場合が多い

では、インフォ丸にMIPS5段パイプラインをまとめてもらいましょう。実際、このパイプラインは良くできていて、単純な32ビットプロセッサはおおむねこれに類似した5段パイプラインを持っています。