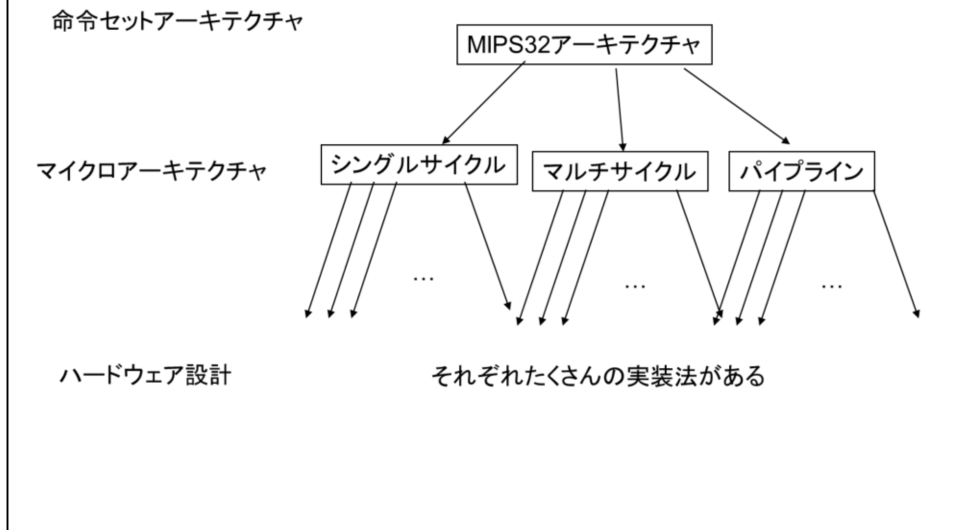
コンピュータアーキテクチャB
MIPSeのパイプライン化

天野英晴
hunga@am.ics.keio.ac.jp



今までシングルサイクル、マルチサイクルマイクロアーキテクチャを紹介してきました。

シングルサイクル マイクロアーキテクチャ

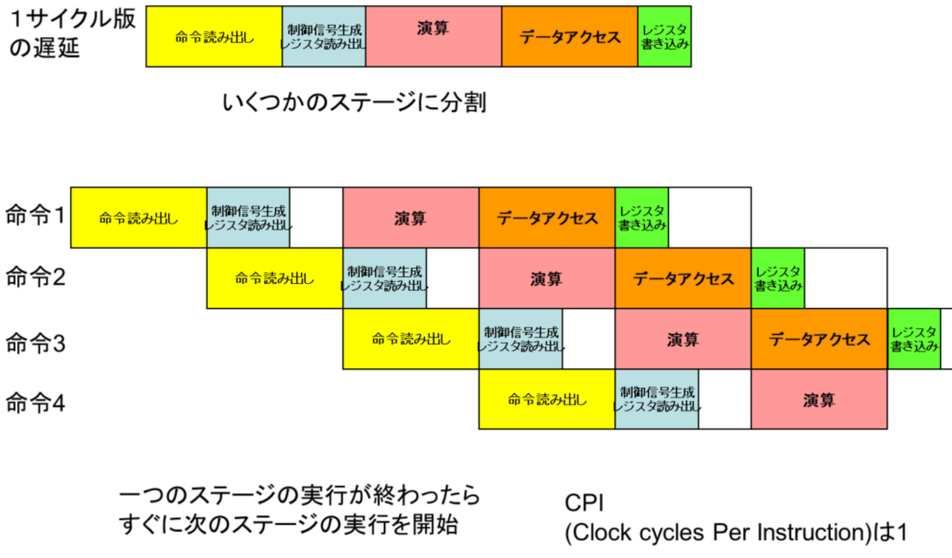
- 最も単純な構成
- 全ての命令は1サイクルで実行される
  - CPI（Clock cycles Per Instruction)=1
- 資源の共有が不可能
  - 特に命令メモリとデータメモリの分離はコスト高につながる
- クリティカルパスが長い
  - しかしCPI=1なので案外速い

シングルサイクルマイクロアーキテクチャは最も単純な構成を持っています。全ての命令は1サイクルで実行され、クリティカルパスが長く、動作周波数は低くなります。しかし、全ての命令は1サイクルで実行されるので案外速いです。一方で資源の共有は不可能で、特に命令メモリとデータメモリの分離はコスト高につながります。

マルチサイクル
マイクロアーキテクチャ

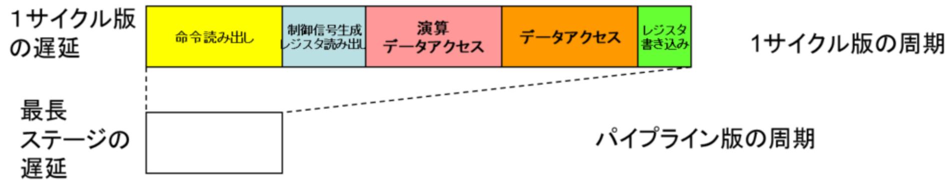
- 資源の再利用は可能
  - 命令・データメモリは兼用
  - PC演算用、分岐演算用の加算器が不要になる
  - しかしレジスタ分の資源は増加する
- クリティカルパスは短くなるが、4分の1にはならない
- 一方CPIは4を越すため、動作速度はシングルサイクルに比べて不利

一方、マルチサイクル版は、資源の再利用は可能で、命令とデータメモリを兼用にできます。ALUを使いまわせるので、PC演算用、分岐演算用の加算器は不要です。しかし、レジスタやマルチプレクサ分のハードウェア資源が増えます。クリティカルパスは短くなるので動作周波数は高くなりますが、4倍にはならないです。一方でCPIは4を超すので、動作速度はシングルサイクルに比べて不利になります。



性能改善のもっとも簡単な方法はパイプライン処理です。1サイクル版のクリティカルパスをなるべく等しく分割します。これはマルチサイクル版の状態遷移が参考になります。状態の進み方を5つに切り、それぞれをステージと呼びます。ここでは命令呼び出し(F)、制御信号の生成およびレジスタの読み出し(D)、演算(E)、データメモリアクセス(M)、レジスタ書き戻し(W)の5ステージに切ります。命令1がFを終え、Dを実行中に次の命令2はFを開始します。命令1がDを終え、Eを実行している間に命令2はDに入り、命令3がFをスタートします。このように5つの命令が一度に動作します。この方法は工場の流れ作業と同じで、パイプライン処理と呼ばれます。パイプラインは途切れなく流れれば、各命令は1クロックに1命令終了します。すなわちCPIは1になります。

パイプライン処理の原則



1. なるべく均等に分割
2. なるべくたくさんのステージに分割

理想の場合、ステージ数（深さ）がdならば性能はd倍

しかし、、、

そうは言っても均等には切れない
ステージ間の受け渡しのための損失がある

パイプライン処理でステージを分ける際には、なるべく均等に、なるべく数多く分割するのが原則です。まったく均等にdステージに分割することができれば、クリティカルパスは1／dになり、CPIは1のままですので、性能はd倍になります。しかし、そうは言っても均等には切れないですし、ステージ間の受け渡しのロスがあります。均等に切れない場合、全体の動作周波数は最も遅いステージに律速されます。最も遅いステージが分割できるならば、できる限りこれをやった方が性能が上がります。

パイプライン処理と並列処理

パイプライン処理はdステージあれば理想はd倍

並列処理はnプロセッサあれば理想はn倍

しかし、

- パイプライン処理は各ステージが自分に必要な資源のみを持てば良い（命令メモリ、ALU、データメモリetc.）
- 並列処理は全プロセッサが全資源を持つ必要がある

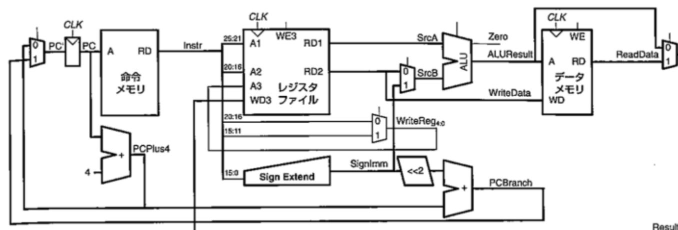
- パイプライン処理の方が簡単に性能が向上する

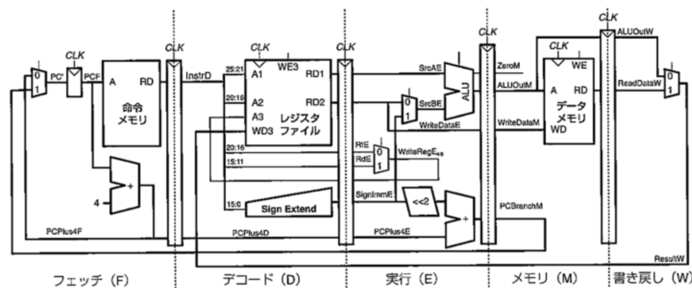
例）工場ではまずパイプライン処理（流れ作業）

- 生産が追いつかなくなると、ラインを複数にする→並列化

ではここでパイプライン処理と並列処理を比較しましょう。パイプライン処理はd個の命令をずらして実行することで、理想的にはd倍の性能向上を得られます。一方、プロセッサをn台同時に実行して性能を上げる並列処理（パイプライン処理も一種の並列処理と言えますが、、）もn個プロセッサがあれば、理想的にはn倍の性能向上が得られます。しかし、パイプライン処理では、各ステージが自分に必要な資源だけを持てばよいのに対して並列処理は全プロセッサが全資源を持つ必要があります。すなわち、パイプライン処理の方がはるかにコストが安いのです。このため、例えば工場でも生産効率を上げようとすればまず流れ作業のベルトコンベアを構築し、生産が追いつかなくなるとラインを複数化します。これが並列処理に当たります。

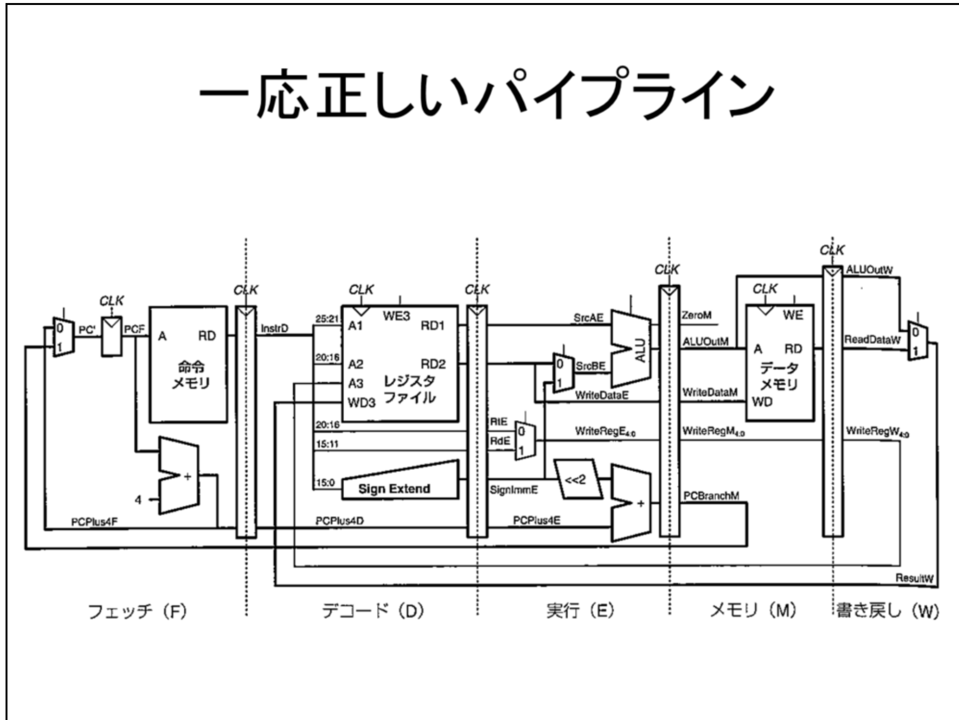
とりあえずパイプラインレジスタを入れてみる



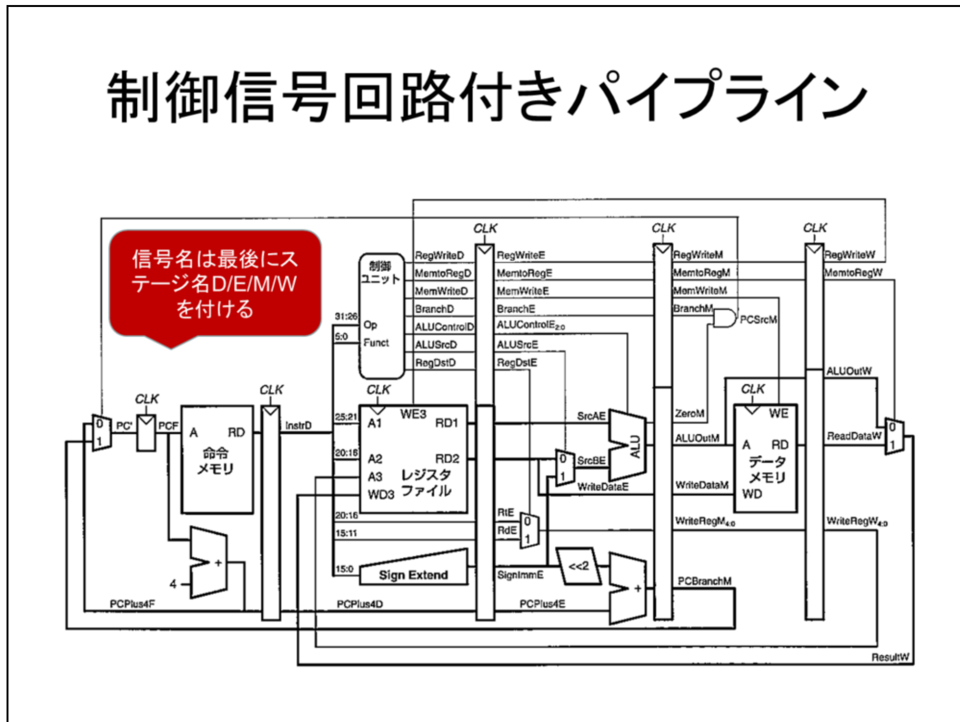


問題が
生じる！

ではMIPSを先の5ステージにパイプライン化してみましょう。ステージ間にパイプラインレジスタを入れ、ステージ間のデータの受け渡しを行います。しかし、これには問題があります。書き込むレジスタ番号は、書き込むデータと共に流れていく必要があり、同期させなければならないので、答えをレジスタファイルに書き込む部分に修正が必要です。



この部分を修正すれば、一応正しいパイプライン構成が実現可能です。書き込むレジスタ番号は最後のWステージまで進んでから書き戻すように修正してあります。



では具体的に制御信号を付けていきましょう。制御信号はパイプラインを流れていくので、名前が非常にわかりにくいです。ここでは、信号名の終わりにステージ名を付けてシステマティックに識別します。

例題

- test.asmをコンパイルして実行し、動作を確認する。パイプラインレジスタの内容確認はgtkwaveを使った方が便利

では例題をやってみましょう。

パイプラインのVerilog記述

```
`include "def.h"
module mipse(
input clk, rst_n,
input [`DATA_W-1:0] instr,
input [`DATA_W-1:0] readdata,
output reg [`DATA_W-1:0] pc,
output [`DATA_W-1:0] aluout,
output [`DATA_W-1:0] writedata,
output Mwrite);
```

入出力はシングルサイクル版と同じ

以下、パイプライン処理のVerilog記述を紹介します。まず入出力ではシングルサイクル版と同じ。

Dステージの記述　信号名は、図と同じになっている

```
wire [`REG_W-1:0] rsD, rdD, rtD, cadrW;    // レジスタ番号
wire [`OPCODE_W-1:0] opcodeD;  // opcode
wire [`SHAMT_W-1:0] shamtD;      // ここのフィールドは未使用
wire [`OPCODE_W-1:0] funcD;    //  functフィールド
wire sw_opD,  addi_opD, lw_opD, alu_opD, lui_opD,opr_opD; //デコード信号
wire regwriteD, MtoregD;    //メモリの書き込み、メモリとレジスタの選択
wire [`OPCODE_W-1:0] alucomD; // ALUのコマンド
wire [`IMM_W-1:0] immD;  // 直値
reg [`REG_W-1:0] writeregW;  //レジスタファイルへの書き込みデータ
wire [`DATA_W-1:0] rd1D, rd2D; //レジスタファイルからの読み出しデータ
wire MwriteD; // メモリへの書き込み信号
reg [`REG_W-1:0] rtE, rdE;  // 以下はEステージへのパイプラインレジスタ
reg [`DATA_W-1:0] signimmE, srcaE, writedataE;
wire [`DATA_W-1:0] resultW;
reg [`OPCODE_W-1:0] alucomE;
reg regwriteE, MtoregE, alusrcE, regdstE;
reg MwriteE;
```

Fステージ

```
/*  Instruction  Fetch Stage */
reg [`DATA_W-1:0] instrD ;
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) instrD <= 0;
  else instrD <= instr;
end
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else
    pc <= pc+4;
end
```

命令を命令レジスタにフェッチ

pcはとりあえず+4（分岐未実装）

今回の記述ではそれぞれのステージを一つのモジュールに入れています。Fステージは簡単です。読み出した命令はInstrDに入れます。

命令のデコード

```
assign {opcodeD, rsD, rtD, rdD, shamtD, funcD} = instrD;
assign immD = instrD[`IMM_W-1:0];
assign sw_opD = (opcodeD == `OP_SW);
assign lw_opD = (opcodeD == `OP_LW);
assign alu_opD = (opcodeD == `OP_REG) & (funcD[5:3] == 3'b100);
assign addi_opD = (opcodeD == `OP_ADDI);
assign ori_opD = (opcodeD == `OP_ORI);
assign MwriteD = sw_opD;
```

一部の命令しか実装してないので注意

Dステージでは取ってきた命令をでコードします。今回の実装は一部の命令なのでご注意ください。

レジスタファイルの読み出し、制御信号生成、

```
rfile rfile_1(.clk(clk), .rd1(rd1D), .a1(rsD), .rd2(rd2D), .a2(rtD),
        .wd3(resultW), .a3(writeregW), .we3(regwriteW));

assign alucomD = (addi_opD|lw_opD|sw_opD|sb_opD ) ?
                `ALU_ADD: ori_opD ? `ALU_OR:
                (lui_opD) ? `ALU_THB: funcD;

assign regwriteD = lw_opD | alu_opD | addi_opD | ori_opD ;
assign MtoregD = lw_opD ;
```

レジスタファイルは、読み出しはDステージの管轄ですが、書き込みはWステージからのフィードバック信号ですのでご注意ください。

```
パイプラインデータレジスタへの書き込み
always @(posedge clk) begin
        rtE <= rtD;
        rdE <= rdD;
        srcaE <= rd1D;
        writedataE <= rd2D;
        alucomE <= alucomD;
        MtoregE <= MtoregD;
        alusrcE <= ~alu_opD;
        regdstE <= alu_opD;
end
always @(posedge clk) begin
        if(ori_opD) signimmE <={16'b0,immD};
        else
                signimmE <= {{16{immD[15]}},immD} ;
end
```

直値フィールドの拡張

Dステージで読み出したレジスタ、レジスタ番号をEステージに送ります。Immediateの拡張もここでやってしまいます。

制御信号のパイプラインレジスタへの書き込み

```
always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
                regwriteE <= 0;
                MwriteE <= 0; end
        else begin
                regwriteE <= regwriteD;
                MwriteE <= MwriteD; end
end
```

リセット時に0にすること！

メモリやレジスタへの書き込み信号はきちんと初期化してやる必要があります。

Eステージの記述

```
wire [`DATA_W-1:0] srcbE, aluoutE;
wire [`REG_W-1:0] writeregE;
reg [`REG_W-1:0] writeregM;
reg [`DATA_W-1:0] writedataM, aluoutM;
reg regwriteM, MtoregM, MwriteM;
assign srcbE = alusrcE ? signimmE : writedataE;
assign writeregE = regdstE ? rdE: rtE;
alu alu_1(.a(srcaE), .b(srcbE), .s(alucomE), .y(aluoutE));
```

Dステージで生成した信号を使うだけなので簡単！

EステージではDステージで生成したデータ、信号を使って演算を行います。

```
パイプラインレジスタの設定
always @(posedge clk) begin
        aluoutM <= aluoutE;
        MtoregM <= MtoregE;
        writedataM <= writedataE;
        writeregM <= writeregE;
end
制御信号はやはりリセットを付ける
always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
                regwriteM <= 0;
                MwriteM <= 0; end
        else begin
                regwriteM <= regwriteE;
                MwriteM <= MwriteE; end
end
```

同様にパイプラインレジスタに入れて次のステージに送ります。

Mステージの記述

```
reg [`DATA_W-1:0] readdataW, aluoutW;
reg regwriteW, MtoregW;
assign aluout = aluoutM;
assign writedata = writedataM;
assign Mwrite = MwriteM;
always @(posedge clk) begin
        readdataW <= readdata;
        aluoutW <= aluoutM;
        MtoregW <= MtoregM;
        writeregW <= writeregM;
end
always @(posedge clk or negedge rst_n) begin
        if(!rst_n)   regwriteW <= 0;
        else   regwriteW <= regwriteM;
end
```


Mステージではメモリにアクセスします。メモリに対する配線は今まで同様、テストベンチで行います。

Wステージの記述

```
assign resultW = MtoregW ? readdataW: aluoutW;
```

結果の選択だけ
レジスタファイル周辺は
Dステージ内で記述したため、ここでは必要ない

パイプラインの記述は大分長くなっているが、増えたのは主としてパイプラインレジスタについてなのであまり恐れる必要はない。

Wステージは、結果を選ぶだけです。レジスタファイル周辺はDステージ内で記述済ですので、そちらをご覧ください。

パイプラインの性能

- 理想的に動けばCPI＝1
- 遅延は最も長いステージで決まる
  – これは論理合成の結果わかる→演習参照
- マルチサイクル、シングルサイクルに比べて圧勝だが実際はそうはいかない

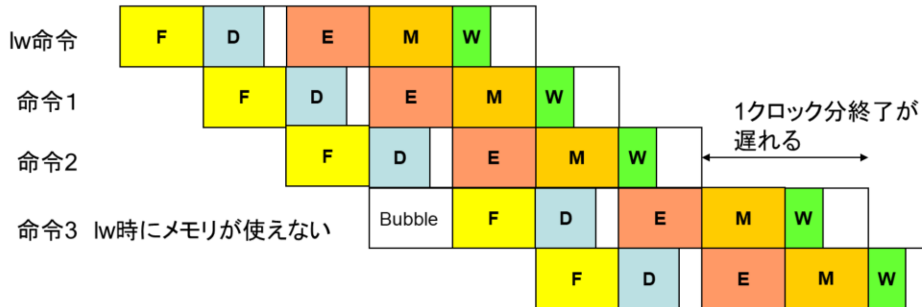
では簡単にパイプラインの性能を見積もりましょう。合成結果を後ほど検討しますが、遅延は圧倒的に減っており、CPIは1なので、性能的には圧勝できます。しかし実際にはそうはいきません。

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

パイプラインは様々な要因で流れがスムーズに行かなくなります。パイプラインがうまく流れなくなる危険、障害をパイプラインハザードと呼びます。これには構造ハザード、データハザード、コントロールハザードの3つがあります。ハザードによりパイプラインの性能が低下することをパイプラインストールと呼びます。ストールによりCPIは1よりも大きくなります。

メモリの共通化による構造ハザード



lw命令の次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
（lw/sw命令が合わせて25%とする）

パイプラインの各ステージは利用する資源を占有する必要があります。片方のステージで資源を使っている間、他のステージで使えない場合、その時間帯パイプラインを待たせてやる必要があります。この例では命令メモリとデータメモリを分けることができず、統合メモリを使った場合を考えます。この場合、先行命令がMステージでメモリをアクセスしている間は命令のフェッチができないので、一クロック遅らせてやります。この分パイプラインは1クロック遅れます。この遅れは、パイプを流れる液体中の泡にたとえられ、バブルと呼ばれます。このため、1クロック分命令の終了が遅れます。この損失は、CPIの増大として考えます。理想のCPIを1とすると、これにストールする確率×ストール時のダメージを加えます。例えばlw/swが合わせて25％の生起確率があるとすると、ストール付きのCPIは1.25となります。

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の競合で起きるので、資源を複製してやれば解決できます。しかし、これにはコストが掛かるので、結局のところコストと性能のトレードオフになります。この場合、1.25のCPIの増大を我慢してメモリを共有化することでコスト減を取るという選択もあり得ます。メモリを分離するのは確かにコストが大きくて現実的ではないのですが、キャッシュの分離は可能なので、実際のプロセッサでは、命令キャッシュとデータキャッシュを独立して設けて、メモリに関する構造ハザードの問題を解決しています。他にも貴重な資源を共有する場合構造ハザードは生じる可能性があるのですが、最近は半導体の面積に余裕があるので、構造ハザードはさほど問題にならなくなっています。

本日のまとめ

- パイプライン処理は命令実行のステップをいくつかのステージに切り、これを流れ作業的に実行する。
- 理想的には1クロックに1命令終わる
- パイプラインの設計指針
  - それぞれのステージで占有して使える資源を用意する
  - なるべく同じ長さに切る
  - ステージ間にはパイプラインレジスタを設けて、制御信号と、データの足並みを揃える
- パイプラインの動きを妨げるものをハザードと呼ぶ
  - 構造ハザードは資源の競合で生じる
  - データハザードは命令間の依存性により生じる
- データハザードはフォワーディングで回避
- しかしLoad命令ではインターロックが必要

インフォ丸が教えてくれる今日のまとめです。

演習5

- pipe2.tarのパイプラインを論理合成し、動作周波数、面積、消費電力（Total Dynamic Power)を求めよ。