マイクロプロセッサ特論　第11回
パイプライン化
テキスト第9章

慶應大学
天野英晴

いままで単一サイクル、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。
今回は現在のCPUのマイクロアーキテクチャの主流であるパイプラインを紹介します。

2サイクル版の合成

- 2サイクル版のPOCOをvivadoで論理合成して、動作周波数を評価し、1サイクル版と比較せよ。どちらがどちらよりもX倍速いという言い方で示せ。

xc7a15tcpg236-3を使おう。

周期は5.8nsecくらいまで行ける。（シングルサイクル版も、8くらいまで行ける。）

mvivado.tarを利用せよ。

2サイクル版は動作周波数は高いがCPIが2倍になる点を注意

- その性能を実現した際の利用率と電力も比較せよ。

前回できなかった2サイクル版の合成をやってみます。

1サイクル、2サイクル版POCOの性能面での問題

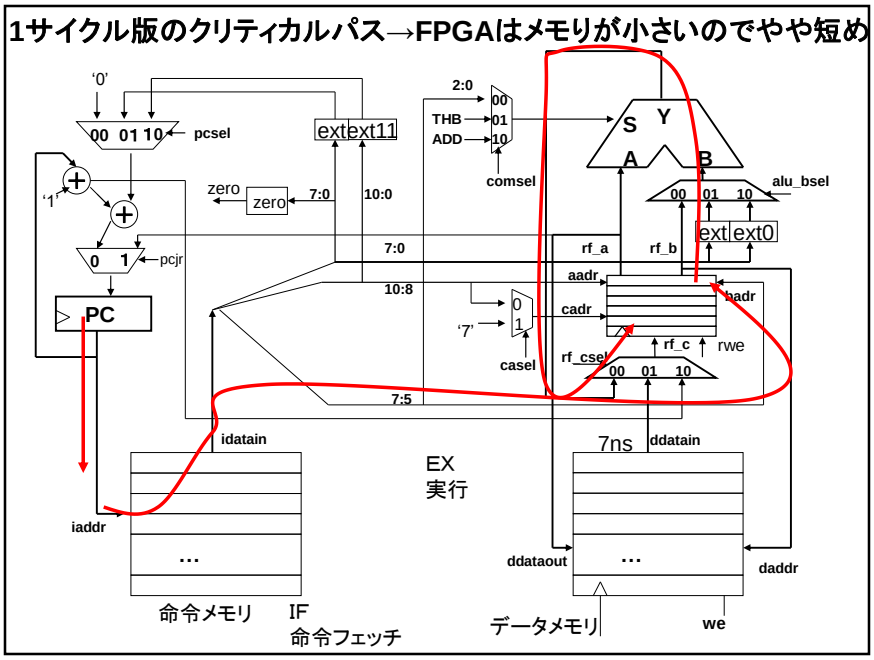
- 1サイクル版
  - 全ての命令を1サイクルで実行
  - 最長の命令遅延に全命令が合わせる必要がある
    - クリティカルパスが長い
    - 動作周波数を上げることが難しい
- 2サイクル版
  - CPIが2倍になる
  - 動作周波数は2倍にはならない

→ 結果として1サイクル版より遅い

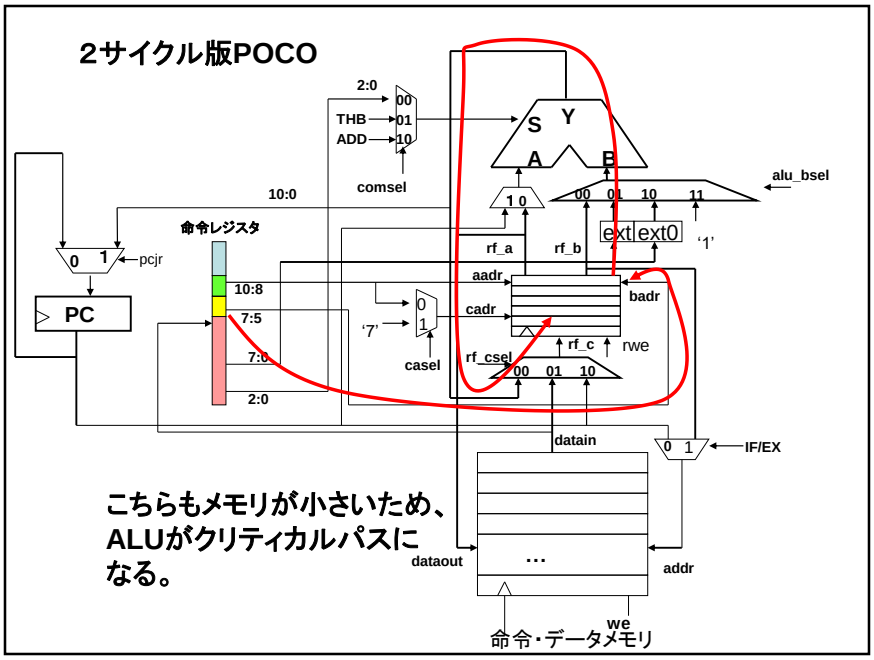
どうすれば性能が上がるか？

→パイプライン化

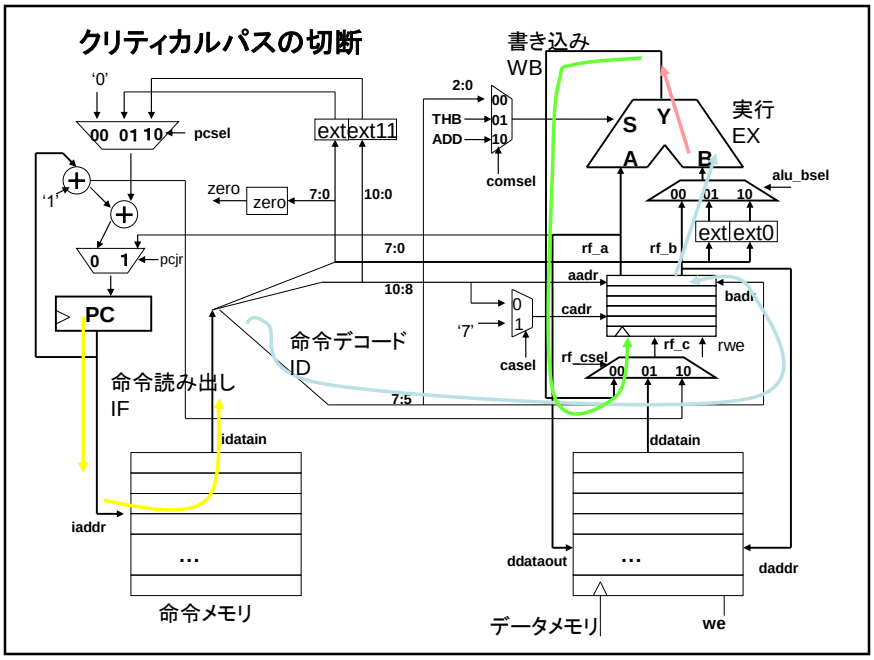
今まで紹介してきた単一サイクル版マイクロアーキテクチャは、全ての命令を1サイクルで実行するため、最もクリティカルパスを持つ命令に全ての命令が合わせる必要があります。このためクリティカルパスが長くなり、動作周波数を上げることが難しくなります。これをマルチサイクル版にすると、例えば2サイクル版は、CPIが2倍になるのに対して動作周波数を2倍にすることはできません。結果として1サイクル版より遅くなってしまいます。ではどうすれば性能が上がるのでしょうか？この手段がパイプライン化です。



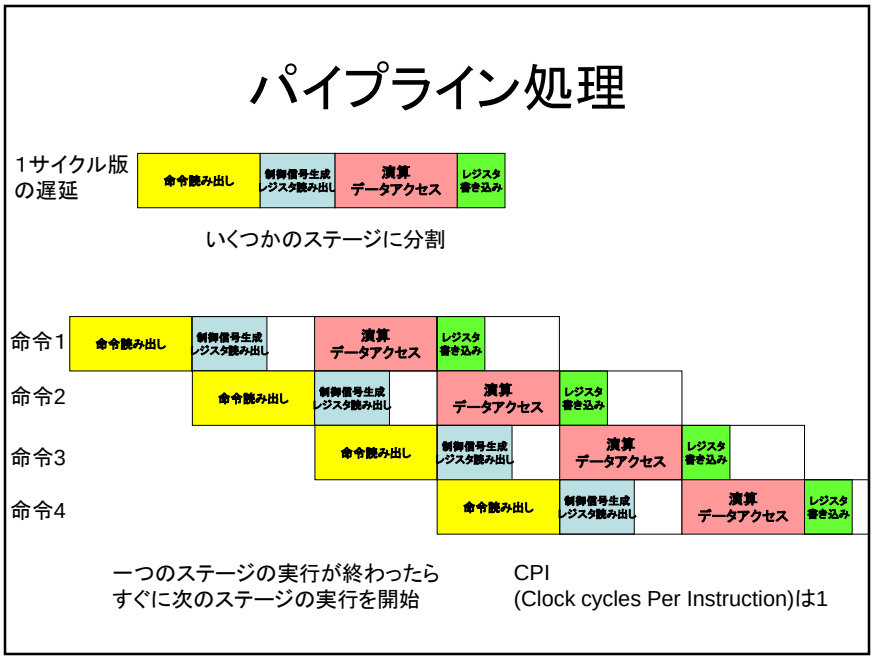
1サイクル版のクリティカルパスは、図に示すようにPCから命令メモリを通って、命令を読み出し、これによりレジスタファイルからレジスタを読み出してALUで演算を行い、結果をレジスタファイルに書き込みます。今回、命令メモリとデータメモリは比較的小さいサイズ（大きいと合成時間が大きくなるため）にしているので、やや短めです。



2サイクル版のPOCOもメモリのサイズが小さいですが、こちらはALUがクリティカルパスになるので、1サイクル版と違って顕在化せず、この点でやや不利な状況になっています。とはいえ、前回までの評価結果はまずまず両方の遅延の差を示しているといえます。
では次にパイプライン化を行います。

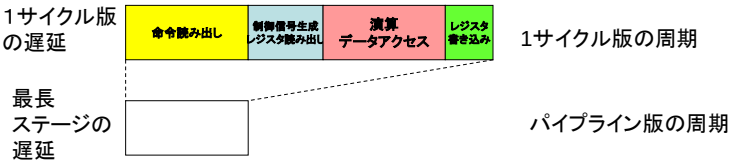


パイプライン化を行う場合、1サイクル版を基本に考えます。1サイクル版の長いクリティカルパスを、できるだけ同じ長さに切断します。命令の読み出しまでをIF（Instruction Fetch)、命令のデコードとレジスタファイルの読み出しをID（Instruction Decode)、ALUでの実行あるいはデータメモリの読み出しをEX（Execution)、最後に結果の書き込みをWB(Write Back)と呼びます。



それぞれの段階をステージと呼び、ある命令の一つのステージの実行が終わったら、すぐに次の命令に対して同じステージで実行を始めてしまいます。そうすると、図に示すように、命令１がWBの時に命令２がEX、命令３がID、命令４がIFを実行するようになります。すなわち、同時に4つの命令がずれながら実行されます。これをパイプライン処理と呼びます。パイプライン処理では、１クロックに1命令ずつ終了します。すなわちCPIは１です。一方、クリティカルパスは分割されているので、1サイクルアーキテクチャに比べずっと短くなります。この方式は、常に全てのステージが動いているので、各ステージは自分の資源を占有しなければなりません。このため、命令メモリとデータメモリを分離して同時に読み出せなければトラブル（構造ハザード）を生じますし、レジスタファイルも常に2つのデータを読み出して、１つのデータを書き込むことができなければなりません。

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

パイプラインのスループットは、もっとも長いステージの時間によって決まります。このため、なるべく同じ長さになるように分割するのが正しいです。理想的に均等に切れれば、ステージ数（深さともいいます）をdとすると性能はd倍になります。また、一番長いステージを少しでも短く出来るのならば、できるだけたくさんのステージに切った方がいいです。ちなみに律速になるステージでなければ切っても意味がなく、ステージ間のデータ受け渡しのための損失のみを受けてしまいます。

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）

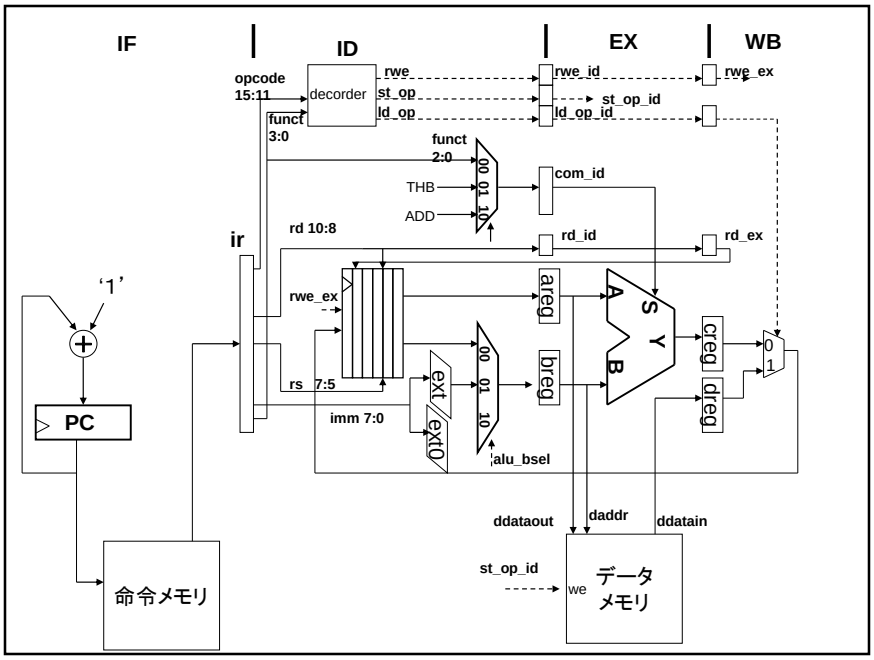
- 生産が追いつかなくなると、ラインを複数にする→並列化

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。

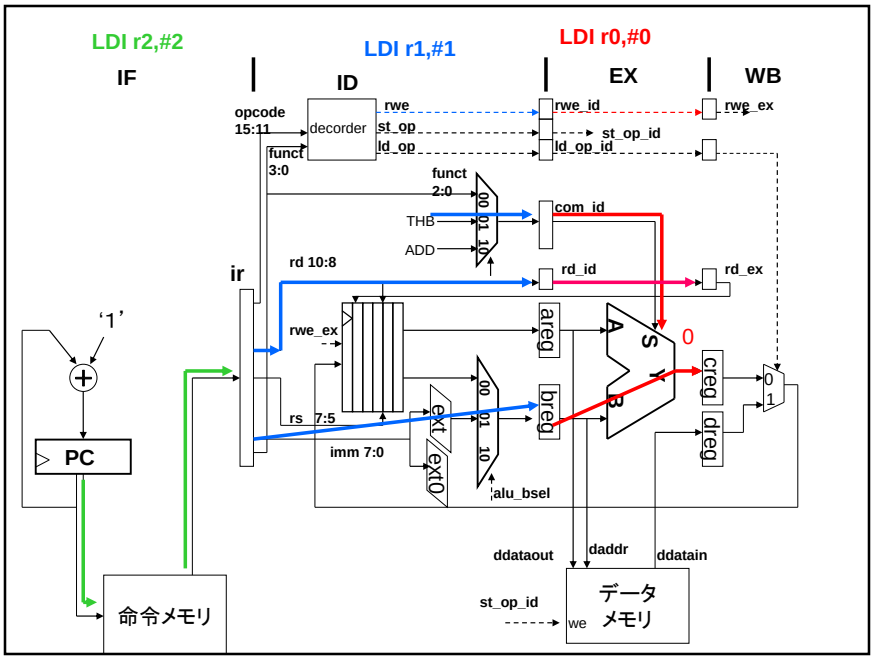
POCOの4段パイプライン

- IF:Instruciton Fetch 命令フェッチ
  - 命令メモリから命令を取ってきて ir(命令レジスタ)に入れる
- ID:Instruction Decode　命令デコード
  - ir中の命令に従って制御信号を作る
  - レジスタファイルからデータを読み出しrega、regbに入れる
- EX:Execution 命令実行
  - ALUで命令を実行する　or データメモリを読み書きする
- WB:Write Back　結果書き込み
  - ALUの計算結果 or データメモリから読み出した値をレジスタファイルに書き戻す
- それぞれのステージ間のパイプラインレジスタに注目！
  - 制御信号もレジスタで遅らせる必要がある

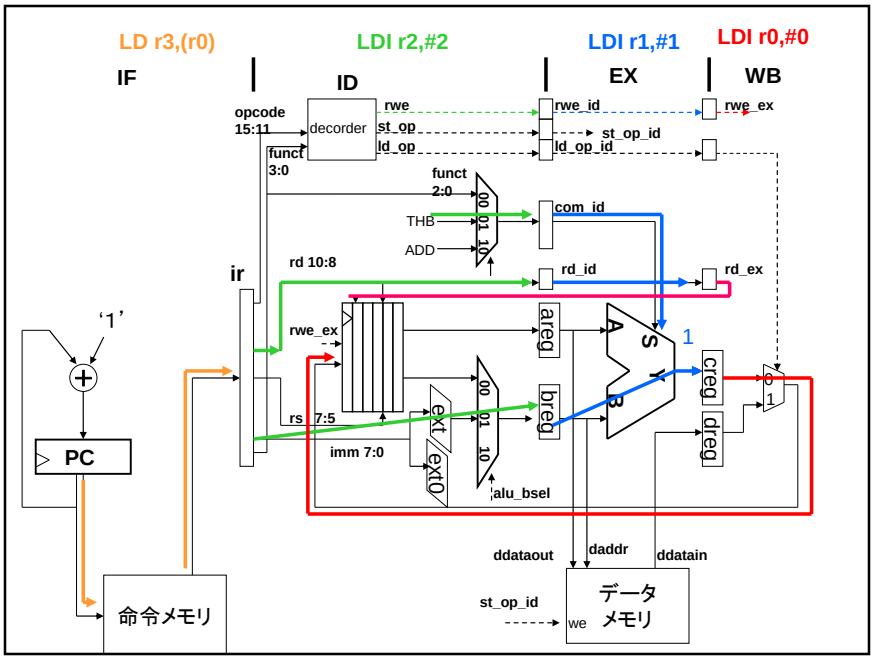
では先ほどのPOCOのパイプラインの役割を簡単に説明しましょう。IFは命令フェッチで、命令メモリから命令を取ってきて命令レジスタirに入れます。次にIDは命令デコードで、ir中の命令コードに従って制御信号を発生するとともに、レジスタファイルからデータを読み出して、これをrega, regbに入れます。EXは命令実行で、IDで生成された信号に従って、演算を行うか、メモリに対してデータを読み書きします。最後はWBでは、計算結果あるいはメモリから読んで来た出たデータをレジスタファイルに書き込みます。それぞれのステージ間にはパイプラインレジスタがあります。これは、データの流と制御信号の足並みをそろえる必要があるためです。



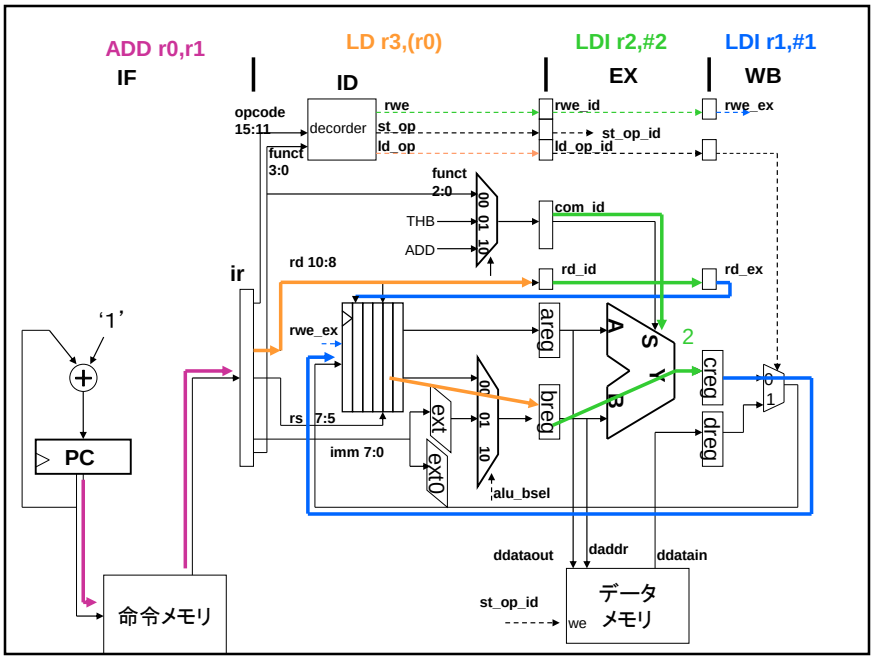
POCOのパイプラインの基本構成を図に示します。まだ分岐命令は実装されておらず、PCは毎回1増えます。演習用のコードの実行の様子をシミュレーションしながら信号線をたどってみましょう。pipeというディレクトリの中のプログラムをシミュレーションしてきます。パイプラインのステージ間で情報を一時記憶しておくレジスタをパイプラインレジスタと呼びます。パイプラインレジスタは複数のステージで同じ情報を記憶するため、名前の区別をする必要があります。ここでは、そのレジスタにデータを書き込むステージの名前を後ろに付けるルールにしています。例えばディスティネーションレジスタ番号を入れておくrdは、IDとEXの間のものをrd_id、EXとWBの間のものをrd_exと呼んでいます。



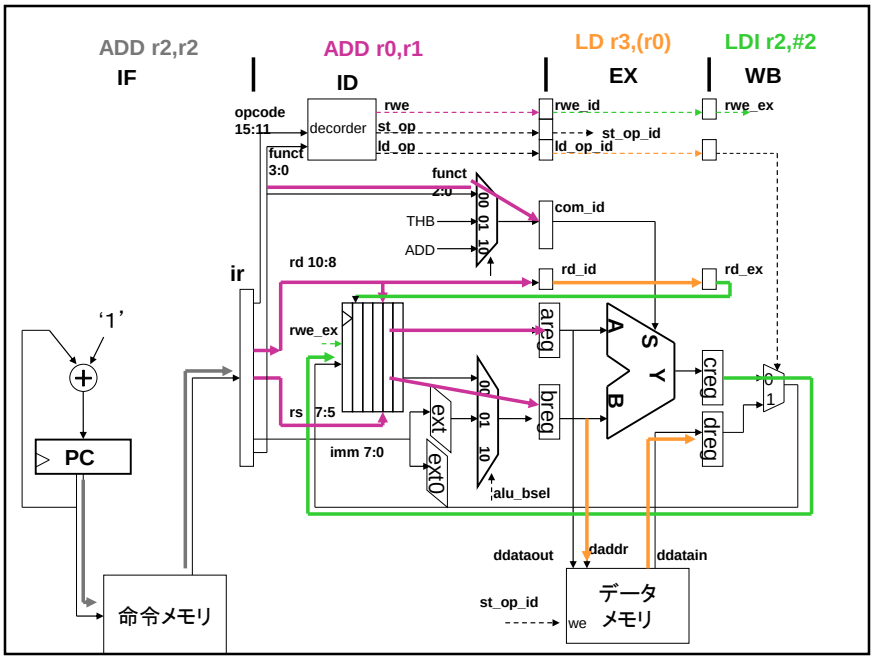
この図はp1kai/example.asmをシミュレーションした際の4クロック目（pc:0002)に対応します。最初に読み出された命令であるLDIが実行を行い、cregに0が入ります。次の命令であるLDI r1,#1はIDステージ中にあり、ir中の命令コードに従ってイミーディエイトの値（1）をbregに入れ、ディスティネーションレジスタの番号をrd_idに入れ、ALUの操作としてTHBをcom_idに入れます。IFでは次の命令のLDI r2,#2をフェッチしています。



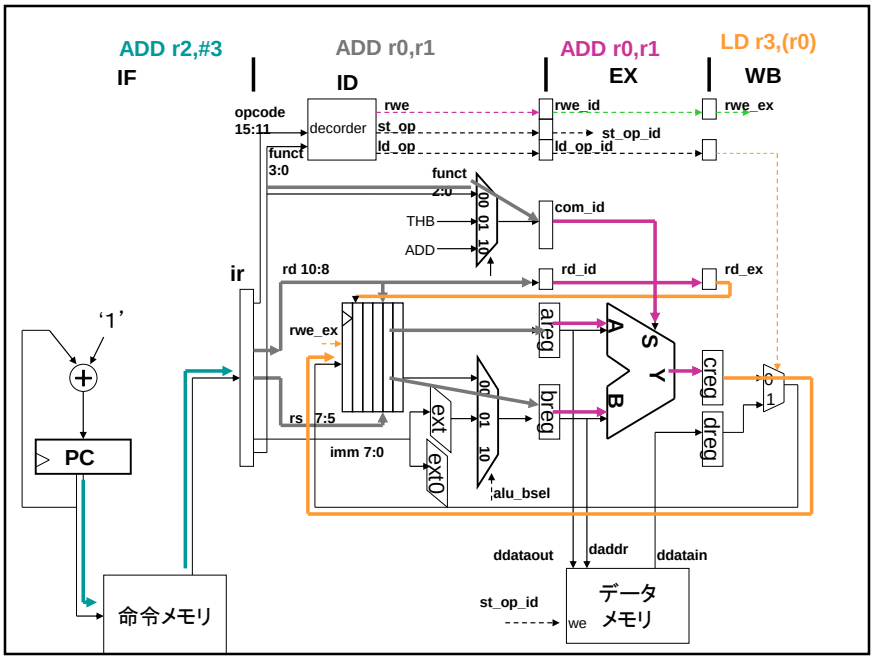
次のクロックの様子です。最初の命令はWBステージに居て、結果をレジスタファイルに書き込みます。書き込むべきレジスタの番号はrd_exに入っており、書き込むデータとあしなみをそろえてWBステージまで送られてきた様子がわかります。



その次のクロックです。今までと違ってLD命令がIDステージに入っています。この命令はレジスタファイルからレジスタを読み出し、bregに入れます。



さらに次のクロックです。LD命令はEXステージで読み出したレジスタの内容を使ってデータメモリを読み出します。読んだデータはdregに入れます。ADD命令はIDステージで、演算に用いるレジスタを読み出し、areg, bregに入れています。行う演算の種類はfunctからcom_idに送られます。線香するLDI命令はWBステージで書き込みを行っています。



さらに次のクロックです。ADD命令はALUで実行されますが、前のクロックでIDステージで全てのお膳立ては済んでいるので、単に演算結果をcregに書き込むだけです。

パイプライン化POCOの記述

IF(命令フェッチ）

```
// Instruction Fetch //
reg [`DATA_W-1:0] pc;
reg [`DATA_W-1:0] ir;

assign iaddr = pc;
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else
    pc <= pc+1;
end

always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) ir <= 0;
  else ir <= idatain;
end
```

今は分岐命令がないので簡単

irに取ってきた命令を入れる

では、パイプライン化POCOのVerilog記述を解説します。基本的に信号名は図と同じにしてありますので、図を見ながら記述を理解してください。通常パイプライン構造を記述する場合、それぞれのステージをモジュール化しますが、これをやるとモジュール間の線を辿るのが大変なので、今回は全体を一つのモジュールとしました。ではまずIFステージ部分を説明しましょう。ここではPCを一つカウントアップし、irに取ってきた命令を入れます。いずれも1クロックに1回ずつ行われます。現時点ではIFステージからIDステージへのパイプラインレジスタはirのみです。

```
// Instruction Decode & Register Fetch //
```

ID(命令デコード1）

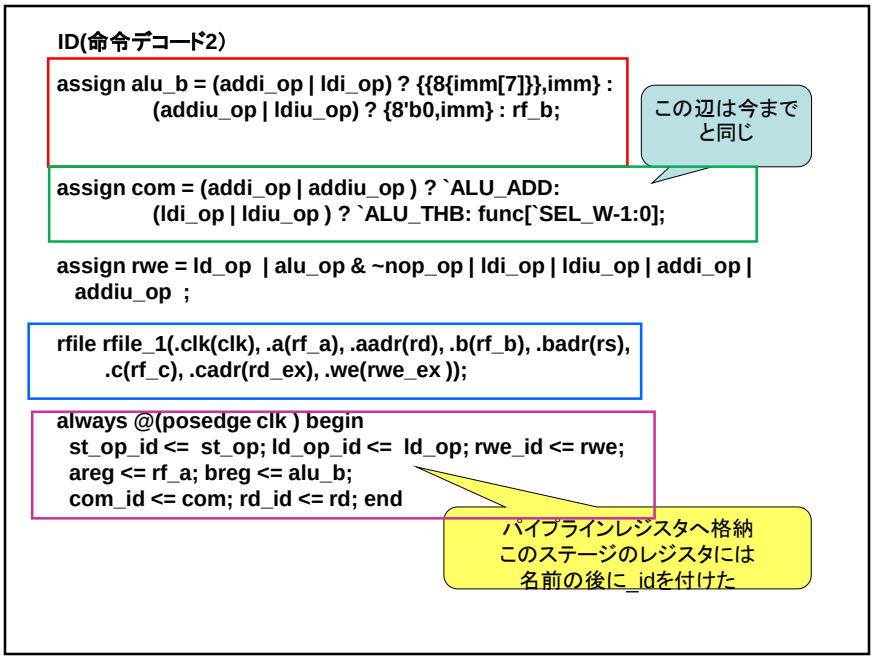
```
wire st_op, addi_op, ld_op, alu_op, nop_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op;
wire [`SEL_W-1:0] com;
wire [`OPCODE_W-1:0] opcode;
wire [`OPCODE_W-1:0] func;
wire [`REG_W-1:0] rs, rd;
wire [`IMM_W-1:0] imm;
wire [`DATA_W-1:0] rf_a, rf_b, rf_c, alu_b;
reg [`SEL_W-1:0] com_id;
reg st_op_id, ld_op_id;
wire rwe;
reg rwe_id;
reg [`REG_W-1:0] rd_id;
reg [`DATA_W-1:0] areg, breg;
reg [`REG_W-1:0] rd_ex;
reg rwe_ex;

assign {opcode, rd, rs, func} = ir;
assign imm = ir[`IMM_W-1:0];
assign st_op = (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = (opcode == `OP_REG) & (func == `F_LD);
assign alu_op = (opcode == `OP_REG) & (func[4:3] == 2'b00);
assign nop_op = (opcode == `OP_REG) & (func == `F_NOP);
assign ldi_op = (opcode == `OP_LDI);
assign ldiu_op = (opcode == `OP_LDIU);
assign addi_op = (opcode == `OP_ADDI);
assign addiu_op = (opcode == `OP_ADDIU);
assign ldhi_op = (opcode == `OP_LDHI);
```

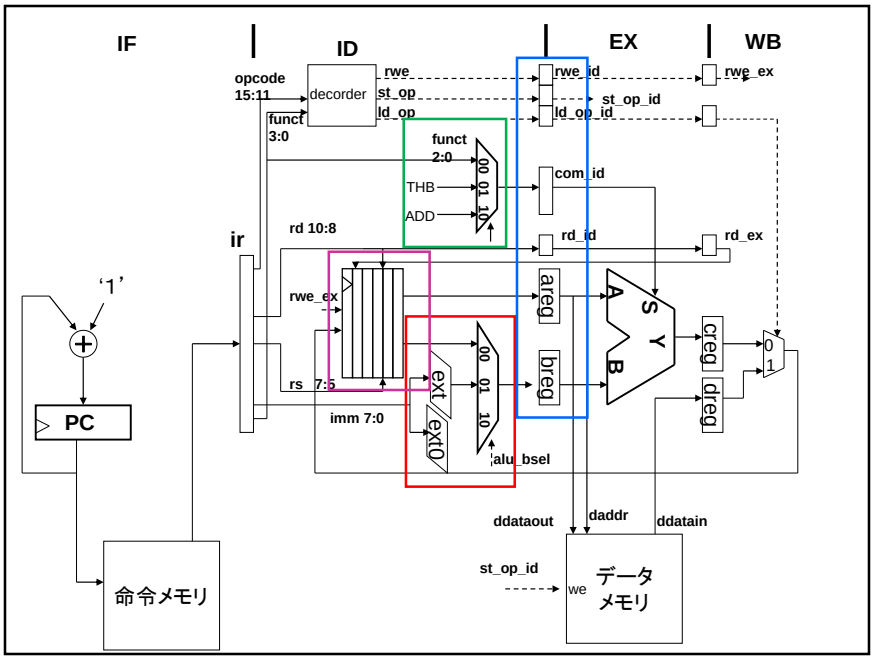
パイプラインレジスタ、途中の変数を定義

デコード

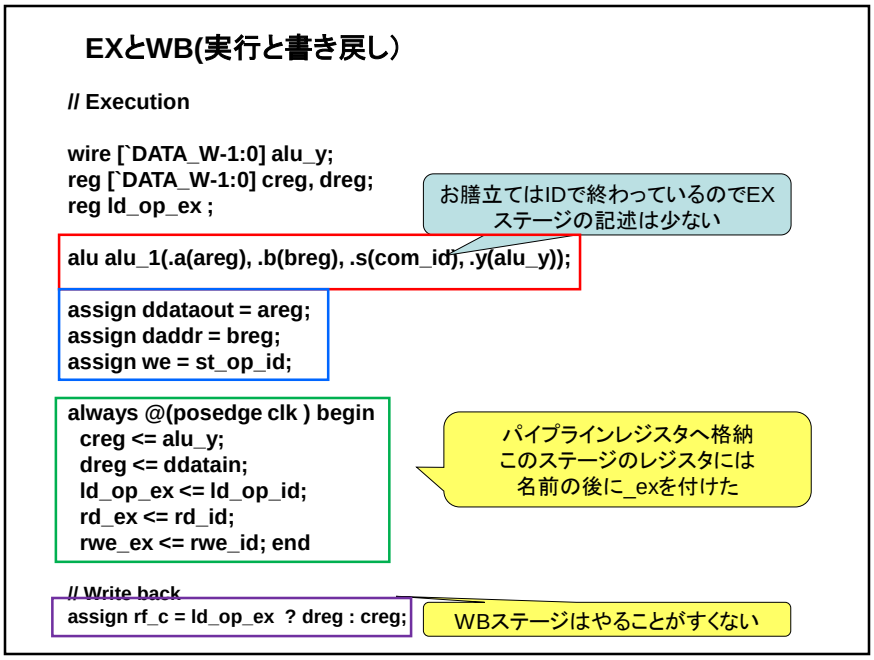
次はIDステージです。ここではir中の命令のデコードを行うと共に、命令コード中のレジスタ番号に従ってレジスタファイルからレジスタを読み出してareg, bregに入れます。デコードの様子はシングルサイクル版と同じです。このデコード信号の一部はIDで使われるだけなので次のステージには送られませんが、後のステージで使われるものはパイプラインレジスタで記憶されて、EXステージに送られます。



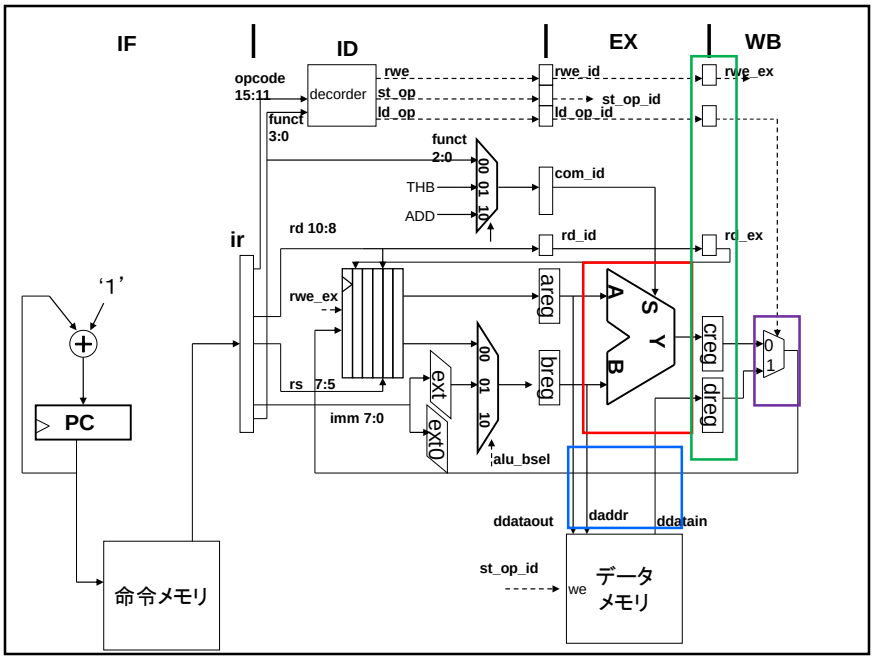
デコード信号、irの内容からALUのB入力の値、comの値を決めます。また、この命令がレジスタファイルに書き込む場合はrweを1にする必要があります。この信号もここで作ります。レジスタファイルからの読み出しは今までと同じです。書き込み用のrf_c, rd_ex, rwe_exは、WBステージから送られてきて、2つ先行した命令の結果を書き込みます。最後のalways文はパイプラインレジスタの記述で次のステージに情報を送っています。次のページの図にここの記述との対応を示します。



前のページの記述と対応してください。



EXステージはALUで演算をし、メモリに読み書きするステージですが、お膳立てはIDステージで終わっているので、パイプラインレジスタareg, breg, com_idを使って演算すれば終わりです。メモリもbregをアドレスに繋いで、aregの中身をデータとして出力してやり、ST命令の場合（st_op_id)に書き込みを行います。計算結果はcreg,読んで来た値をdregに格納し、書き込みのために制御信号のうちLD命令かどうか(ld_op_id)、書き込むレジスタ番号（rd_id)、書き込むかどうか（rwe_id)をそれぞれパイプラインレジスタでWBステージに送ります。WBステージでやることは非常に簡単です。LD命令の場合はdregを、そうでない場合はcregをレジスタファイルの入力に戻してやります。rf_c, rd_ex, rwe_exはIDステージのレジスタファイルに送られていますので、これに従って結果が書き込まれます。

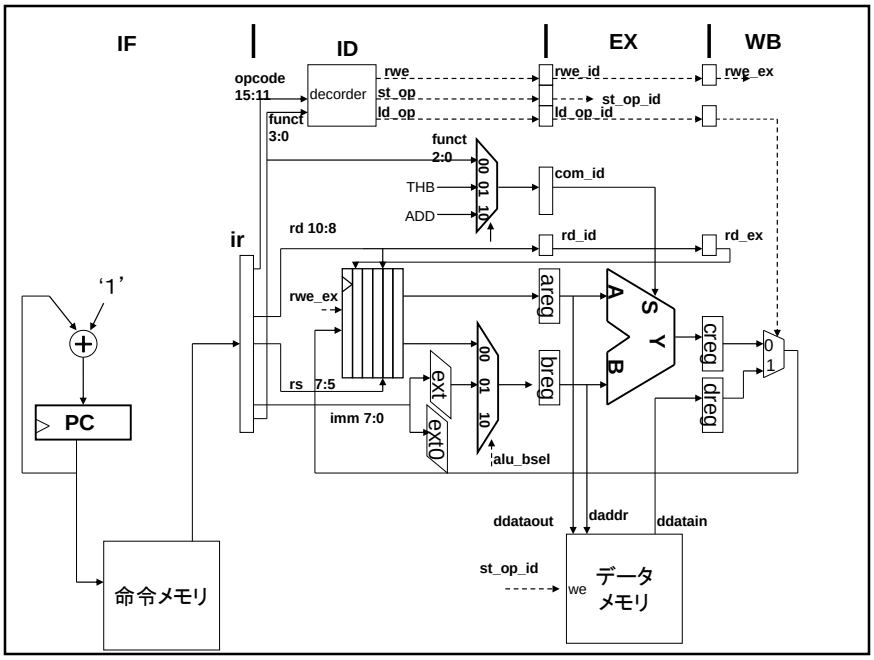


前のページの記述と対応してください。

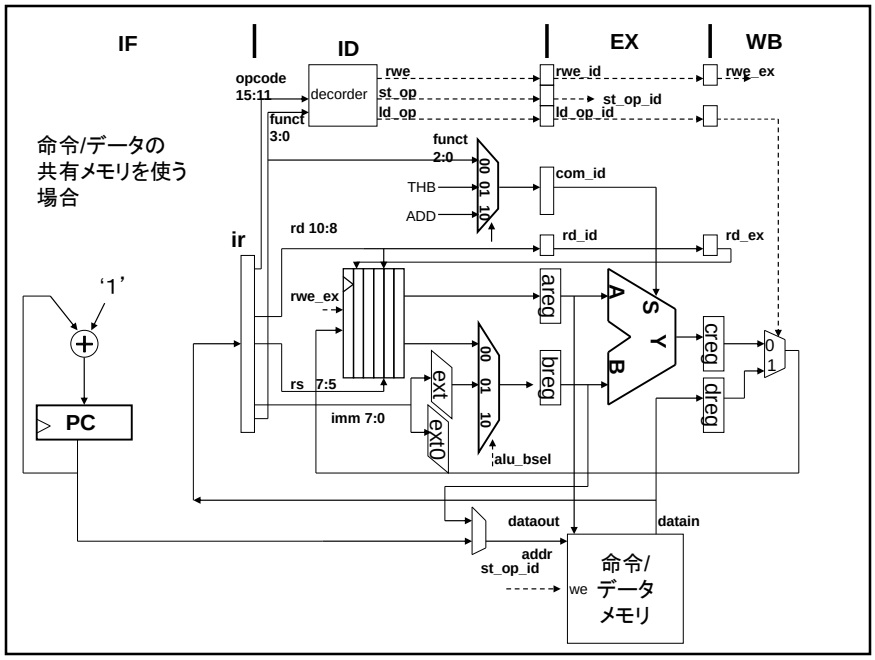
パイプラインハザードとは？

- 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。

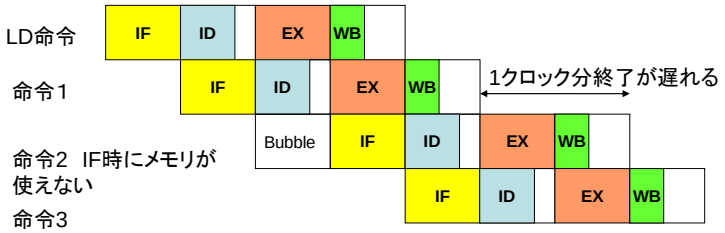


まず、最初に構造ハザードを紹介しましょう。構造ハザードは資源の共有によって生じます。今回のPOCOのパイプラインはそれぞれのステージがそれぞれの資源を占有しているために性能は低下しません。しかし、このため命令メモリとデータメモリは分離して持たなければなりません。



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。

メモリの共通化による構造ハザード



LD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
（LD/ST命令が合わせて25%とする）

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。

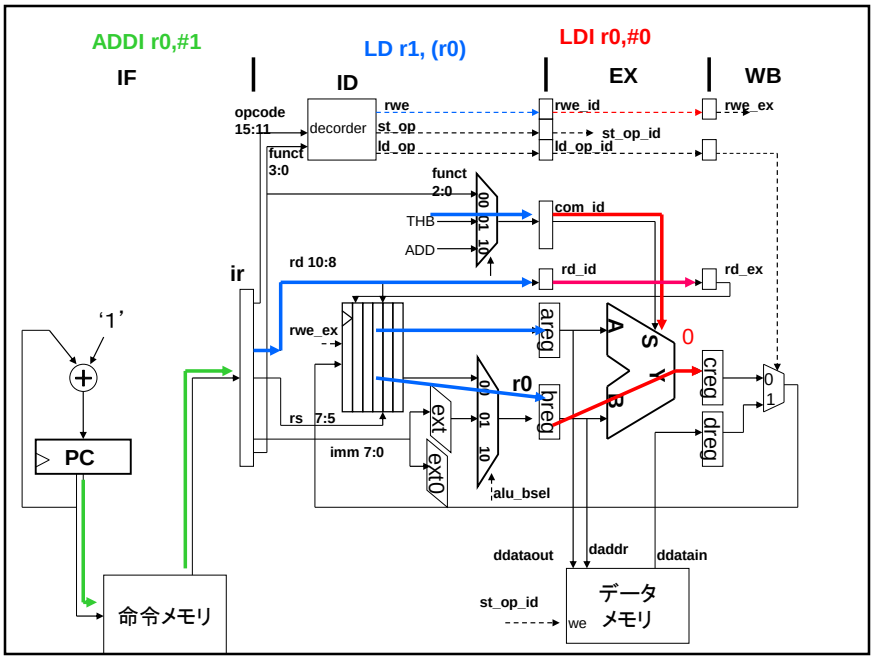
データハザード

- 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

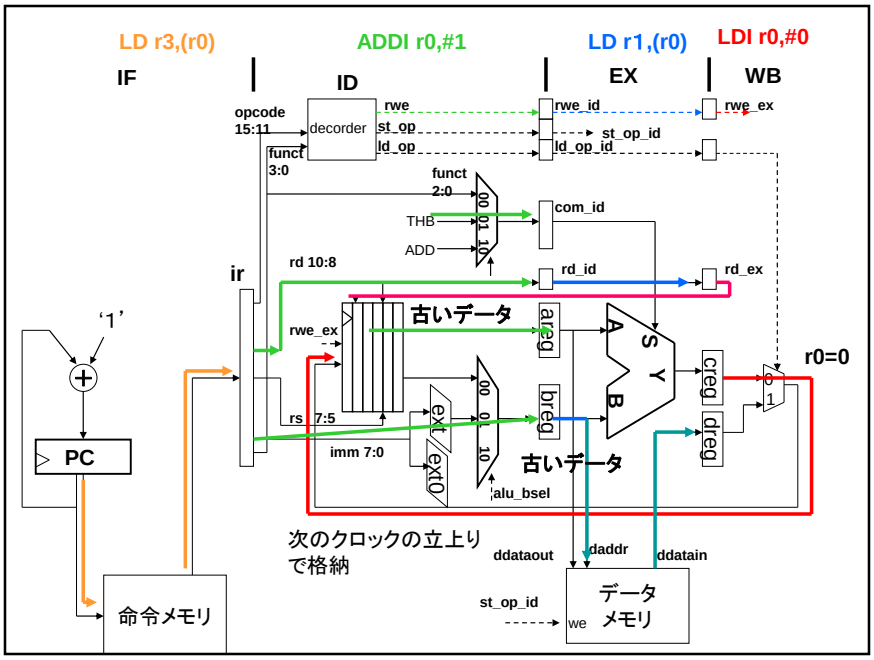
→ NOPを入れて命令の間隔を保持する

→ 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

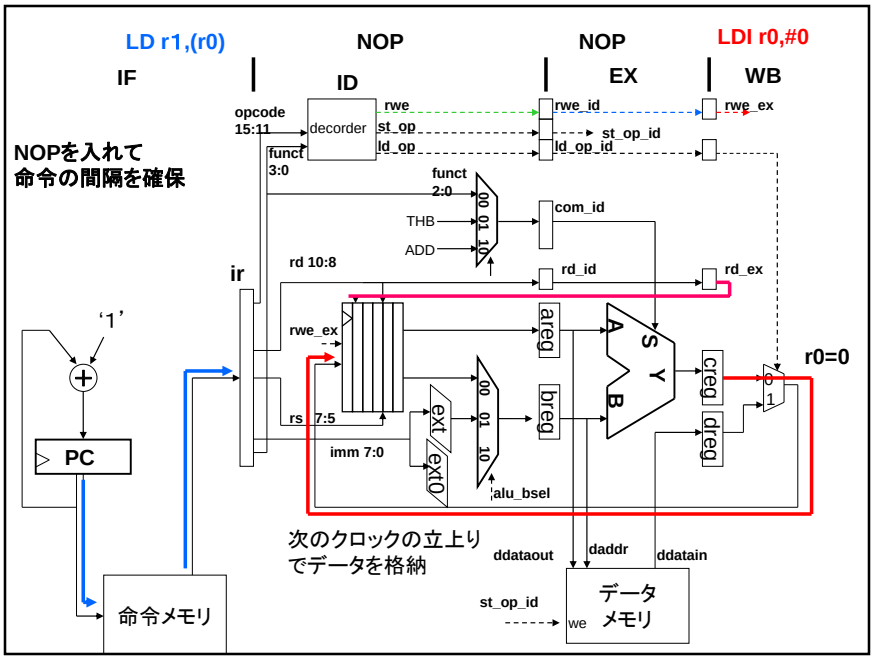
次にデータハザードを紹介します。データハザードは、直前の命令の結果がレジスタファイルに書き込まないうちに、後続の命令がこれを読みだすことにより起きます。相互に依存する命令を部分的に同時に実行しようとするために生じてしまい、パイプライン処理の本質的な問題点といえます。RAW、WAR、WAWの三つのハザードが考えられますが、POCOではRAWハザードだけが生じます。他のハザードについては後に説明します。



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

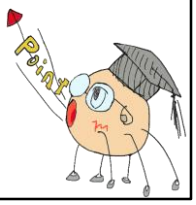
ハザード付きCPI＝
理想のCPI＋ストールの確率×ストールのダメージ

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。

本日のまとめ

- パイプライン化はCPUのクリティカルパスをステージに分割し、次々に命令を入れることで、時間的にずらして複数の命令を実行する方法
- 各ステージは自分で使う資源を占有する
- 一番長いステージで全体のクロックが制限される
  - 一番うまく切れればdステージで性能はd倍になる
- 1サイクル版を基本として、パイプラインレジスタでステージ間の情報を受け渡す
  - 命令と制御情報が足並みをそろえるようにする
- パイプラインハザード
  - 構造ハザード
    - 資源の競合により生じる
  - データハザード
    - データの依存性により生じる



インフォ丸が教えてくれる今日のまとめです。

演習

パイプライン化POCOをvivadoで合成し、動作周波数、Utilization、電力を求め、シングルサイクル版と比較せよ。