情報工学実験 IV 第 6 回： データハザードとフォワーディング

柴田裕一郎・松尾堅太郎・元島晃伸

shibata@cis.nagasaki-u.ac.jp
情報工学コース

2019 年 1 月 22 日

3 つのパイプラインハザード

パイプライン ⇒ 理想的には 1 命令／1 クロックだが…

- 構造ハザード（structural hazard）
  異なるステージで同一の資源（たとえば演算器など）を使おうとして起こる．
- データハザード（data hazard）
  ある命令が計算する結果を後続の命令が使おうとしたときに，前の命令がまだパイプラインの中にあって，必要なデータが計算できていないために起こる．
- 制御ハザード（control hazard）
  分岐命令がまだパイプラインの中にあり PC（プログラムカウンタ）に飛び先番地を書き込んでいないのに，次々と後続の命令を読んでしまうために起こる．

データハザードを起こすプログラムの例

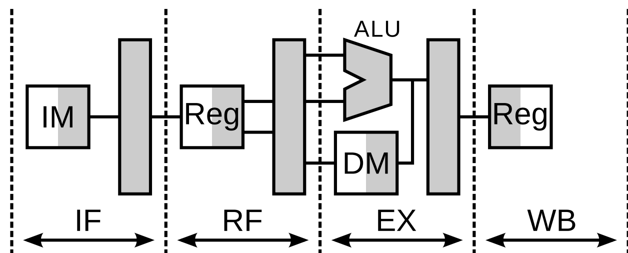
r1 に 5 を入れて，これを r2，r3，r4 にコピーするプログラム

```
LLI r1, #5         ... r1 に「5」を代入
MV  r2, r1         ... r1 を r2 にコピー
MV  r3, r1         ... r1 を r3 にコピー
MV  r4, r1         ... r1 を r4 にコピー
```

実行結果

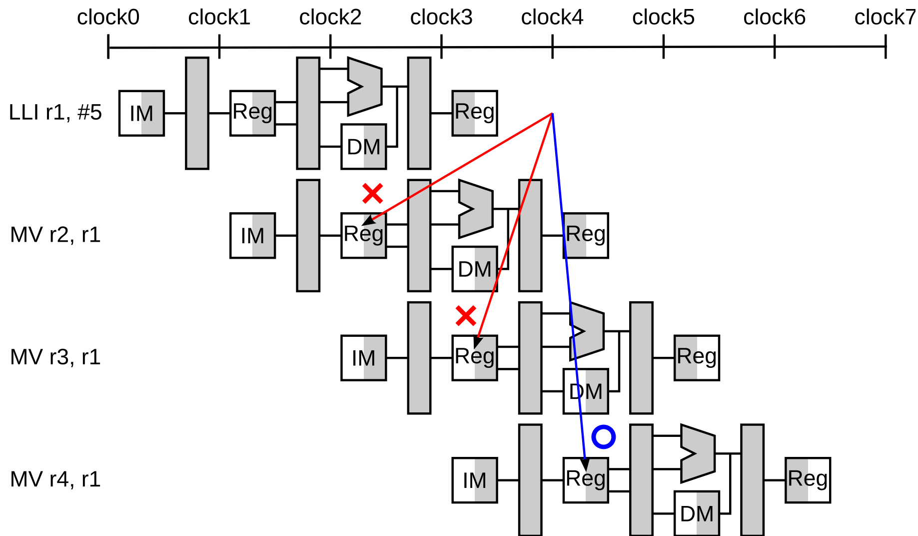
```
==== clock: 0 ====
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 1 ====
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 2 ====
 alu_ain:0000 alu_bin:0005 alu_op:0001 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 3 ====
 alu_ain:0000 alu_bin:0000 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 4 ====
 alu_ain:0000 alu_bin:0000 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0005 0000 0000 0000 0000 0000 0000      <--- r1 に「5」が入った

==== clock: 5 ====
 alu_ain:0000 alu_bin:0005 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0005 0000 0000 0000 0000 0000 0000      <--- r2 にはコピーされない

==== clock: 6 ====
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0005 0000 0000 0000 0000 0000 0000      <--- r3 も「0」のまま

==== clock: 7 ====
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0005 0000 0000 0005 0000 0000 0000      <--- r4 は「5」になった
```

パイプラインの簡略記法



IM: 命令メモリ
DM: データメモリ
Reg: レジスタファイル

データハザード



データハザードとは?

例えて言うと，チャーハンを作るとき，ごはんを炊きながら，野菜を洗ったり切ったり出来るが，ごはんが炊き上がるまでいためられないのと同じである．

太田 絢子 / 2003 年度レポート

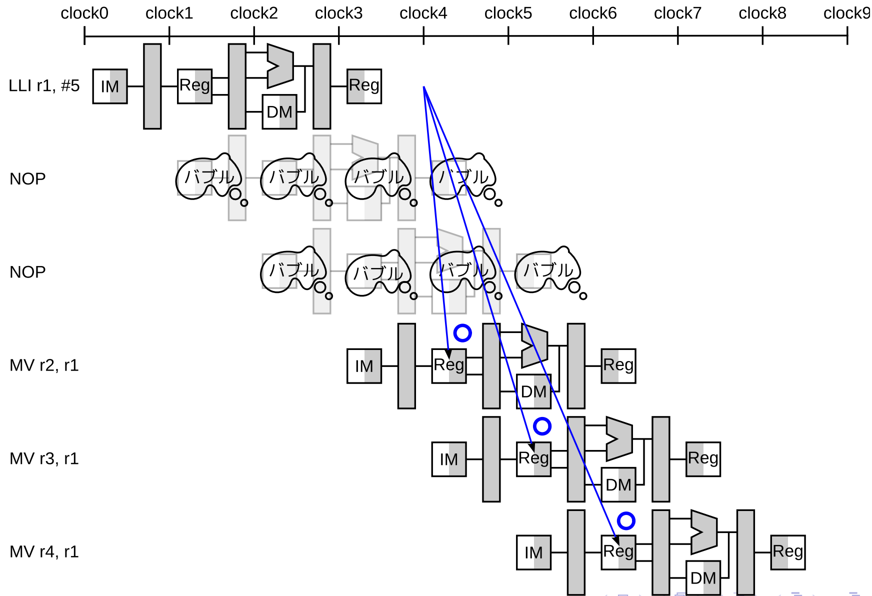
どうやって解決するか?

データハザード： ある命令がレジスタに保存した結果は，直後の 2 命令では使うことができない.

解決法：

1. NOP 命令を 2 個挿入してストール
   何もしない命令（NOP）を 2 個はさんで時間を稼ぐ.
2. フォワーディング（バイパッシング）
   ハードウェアを変更して，ステージ間でデータを横流しするバイパス経路を設ける.

NOP 命令によるストール



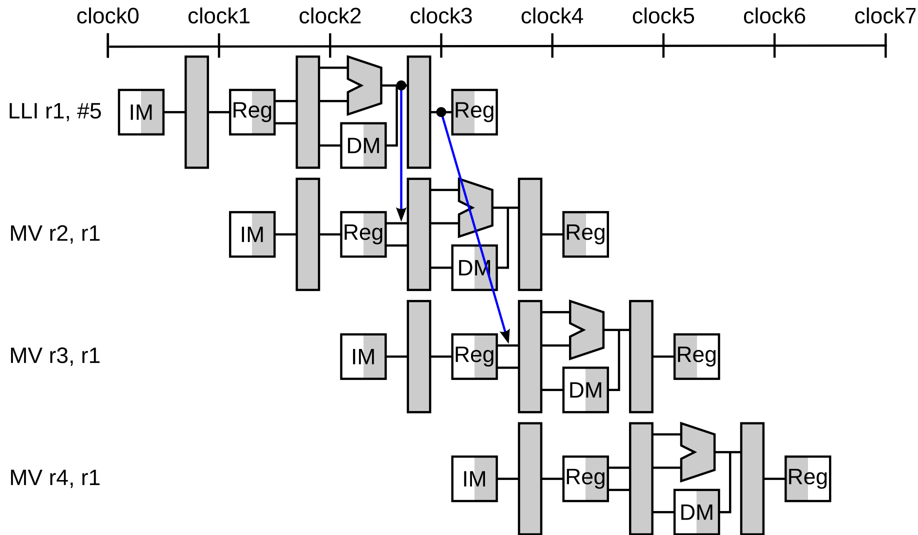
NOP 挿入法のデメリット

- 最悪の場合，3 クロックに 1 個の命令しか処理できない
- 直前の命令の計算した結果を利用したいことは頻繁にある

性能を低下させないでデータハザードを何とかしたい

フォワーディング

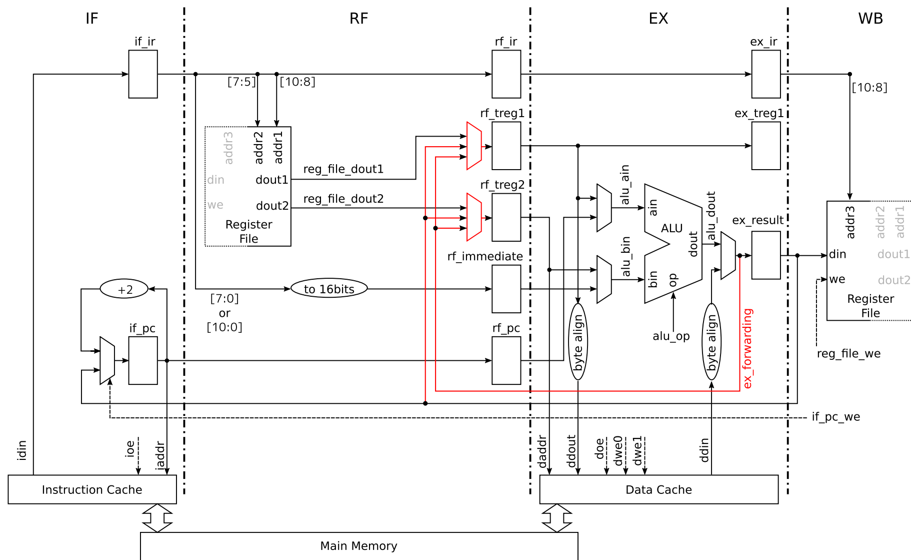
フォワーディングする場合の実行の様子



フォワーディングのための経路

1. EX ステージから RF ステージへのフォワーディング経路
   - 直後の命令にデータを受け渡すためのバイパス経路
   - WB ステージで書き込むデータは EX ステージですでに計算されている
   - これを ex_result レジスタに書き込む前に RF ステージへフォワーディング
2. WB ステージから RF ステージへのフォワーディング経路
   - 2 つ後の命令にデータを受け渡すためのバイパス経路
   - WB ステージでレジスタファイルに書き込むデータを RF ステージにフォワーディング

フォワーディングつき RISC16p の構成



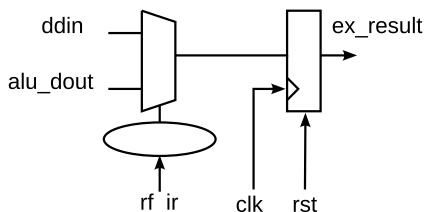
フォワーディングを行う条件

1. EX ステージからのフォワーディング条件
   - EX ステージで処理されている命令がレジスタに書き込む命令である
   - EX ステージで処理されている命令が書こうとしているレジスタの番号と RF ステージで読もうとしているレジスタの番号が等しい
2. WB ステージからのフォワーディング条件
   - WB ステージで処理されている命令がレジスタに書き込む命令である
   - WB ステージで処理されている命令が書こうとしているレジスタの番号と RF ステージで読もうとしているレジスタの番号が等しい

練習： 両方の条件が成り立った場合はどちらを選ぶべきか?

組み合わせ回路とレジスタ

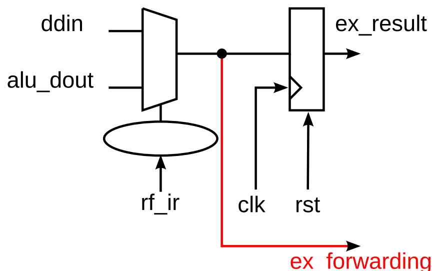
always_ff ブロックの中にも組み合わせ論理の記述が入っている.




```
reg [15:0] ex_result;

always_ff @(posedge clk) begin
  if (rst)
    ex_result <= 16'd0;
  else begin
    if (rf_ir[15:11] == 5'd0 && rf_ir[4] == 1'b1)
      ex_result <= ddin;
    else
      ex_result <= alu_dout;
  end
end
```

組み合わせ回路とレジスタ

レジスタの入力に接続されている配線を分岐させて使うには，組み合わせ回路とレジスタの記述を分離すればよい．




```
reg [15:0] ex_result;
logic [15:0] ex_forwarding;

always_comb begin
  if (rf_ir[15:11] == 5'd0 && rf_ir[4] == 1'b1)
      ex_forwarding <= ddin;
  else
      ex_forwarding <= alu_dout;
end

always_ff @(posedge clk) begin
  if (rst)
    ex_result <= 16'd0;
  else
    ex_result <= ex_forwarding;
end
```