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

2019 年 1 月 22 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1 パイプラインハザード

パイプライン構造を導入することによって，理想的には 1 クロックで 1 命令の処理を行うことができるようになりました．しかし，この構造ですべてがうまく行くわけではありません．命令が完了する前に次々と後続の命令をフェッチしていくため，正しく動作しないこともあります．これをパイプラインハザードといいます．パイプラインハザードの原因には，前回説明した構造ハザードを含めて以下の 3 種類があります．

- 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）を使おうとして起るもの．
- データハザード（data hazard）
  ある命令がまだパイプラインの中にあって実行結果をまだ書き込んでいないのに，その結果を後続の命令が使おうとして起こるもの．
- 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って 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るもの．

構造ハザードについてはすでに対処しました．今回は 2 番目のデータハザードへの対処法について考えます．

2 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

前回作ったパイプライン型プロセッサ RISC16p で以下のプログラムを実行してみます．やっていることは，まず r1 に 5 を入れて，これを r2，r3，r4 にコピーしていくだけです．

```
LLI r1, #5      (00001001 00000101)      // r1 に「5」を代入
MV  r2, r1      (00000010 00100001)      // r1 を r2 にコピー
MV  r3, r1      (00000011 00100001)      // r1 を r3 にコピー
MV  r4, r1      (00000100 00100001)      // r1 を r4 にコピー
```

これを実行した結果を以下に示します．各クロックでの r0 から r7 までのレジスタの値に注目してください．

```
==== clock: 0 ====
 if_pc:0000 if_ir:0000000000000000
```

```
 rf_pc:0000 rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0000 idin:0905 ioe: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: 1 ====
 if_pc:0002 if_ir:0000100100000101
 rf_pc:0000 rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0002 idin:0221 ioe: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 ====
 if_pc:0004 if_ir:0000001000100001
 rf_pc:0002 rf_ir:0000100100000101 rf_treg1:0000 rf_treg2:0000 rf_immediate:0005
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0004 idin:0321 ioe:1
 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 ====
 if_pc:0006 if_ir:0000001100100001
 rf_pc:0004 rf_ir:0000001000100001 rf_treg1:0000 rf_treg2:0000 rf_immediate:0021
 ex_ir:0000100100000101 ex_result:0005
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0006 idin:0421 ioe:1
 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 ====
 if_pc:0008 if_ir:0000010000100001
 rf_pc:0006 rf_ir:0000001100100001 rf_treg1:0000 rf_treg2:0000 rf_immediate:0021
 ex_ir:0000001000100001 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0008 idin:0000 ioe:1
```

```
 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

==== clock: 5 ====
 if_pc:000a if_ir:0000000000000000
 rf_pc:0008 rf_ir:0000010000100001 rf_treg1:0000 rf_treg2:0005 rf_immediate:0021
 ex_ir:0000001100100001 ex_result:0000
 daddr:0005 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000a idin:0000 ioe:1
 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

==== clock: 6 ====
 if_pc:000c if_ir:0000000000000000
 rf_pc:000a rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000010000100001 ex_result:0005
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000c idin:0000 ioe:1
 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

==== clock: 7 ====
 if_pc:000e if_ir:0000000000000000
 rf_pc:000c rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000e idin:0000 ioe:1
 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
```

4 クロック目で最初の命令（LLI r1，#5）の WB ステージの動作が完了して r1 に 5 が入っているのが分かります．しかしその次のクロックでは，2 番目の命令（MV r2，r1）が完了して r2 にも 5 が入るべきですが，そうなっていません．その次のクロック（6 クロック目）でも r3 に 5 が入るべきですが，値は 0 のままです．しかしながら，その次の 7 クロック目では，最後の命令（MV r4，r1）の動作が正常に完了し，r4 に 5 が入っています．

どうしてこのようになるのか，パイプラインの動作を考えてみましょう．4 段パイプラインの構造を簡単に表すため，図 1 に示した簡略図を用いることにします．この図は各ステージで使用するハードウェア資源と，次のステージに情報を渡すためのレジスタを示しています．IM は命令メモリ（命令キャッシュ），DM はデータメモリ（データキャッシュ），Reg はレジスタファイルを示しています．この記法を用いて前述のプログラムがパイプラインで実行される様子を図 2 に示します．図 2 の 1 段目を見ると分かるように，最初の LLI 命

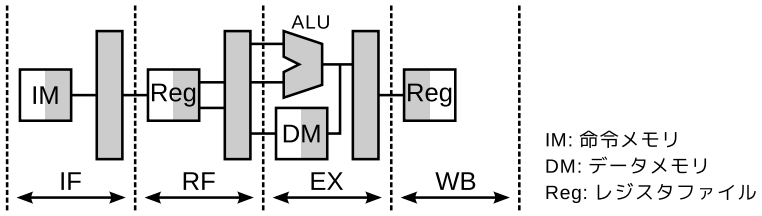

図 1　パイプラインの簡略図

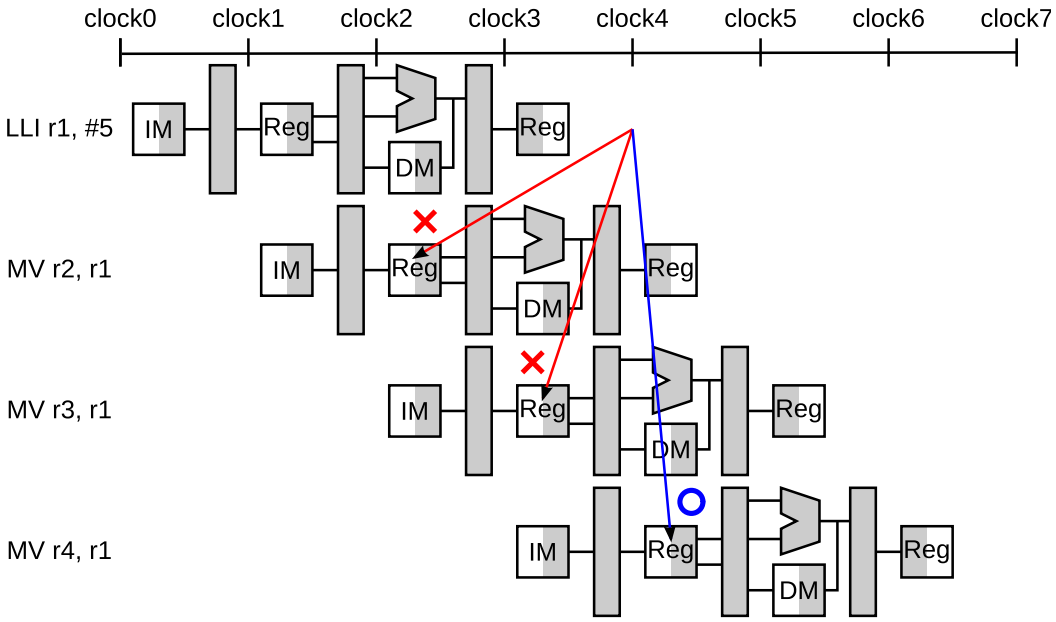

図 2　データハザードの例

令が完了して r1 に 5 が入るのは 4 クロック目の立ち上がりになります．

一方，2 段目を見ると分かるように，次の MV 命令（`MV r2, r1`）が r1 の値をレジスタファイルから値を読み出すのは RF ステージであり，2 クロック目です．これは，最初の命令が r1 に値を書き込むより前なので，正しい値を読むことができません．同様に，3 番目の命令（`MV r3, r1`）が r1 の値をレジスタファイルから読み出すのは 3 クロック目です．LLI 命令が r1 に 5 を書き込むのとほぼ同時なので，一見，正しい値が読めそうですが，レジスタファイルに FF を使っているため，実際に r1 に 5 が書かれるのは 4 クロック目の立ち上がりになります．したがって，ここでも正しい値を読むことはできません．r1 に書かれた 5 を正しく読めるのは 4 クロック目以降ということになり，4 番目の命令（`MV r4, r1`）で初めて正しいデータを読めるということになります．これがデータハザードです．

3　解決法その 1： NOP 命令挿入によるストール

データハザードの簡単な解決法は，NOP 命令（何もしない命令）を間に挟んで時間をかせぐことです．図 2 から分かるように，ある命令が計算した結果を他の命令が利用するためには，少なくとも 2 命令分実行を遅らせる必要があります．そこで，間に 2 つの NOP 命令を挿入することによって，データの受け渡しができるよ

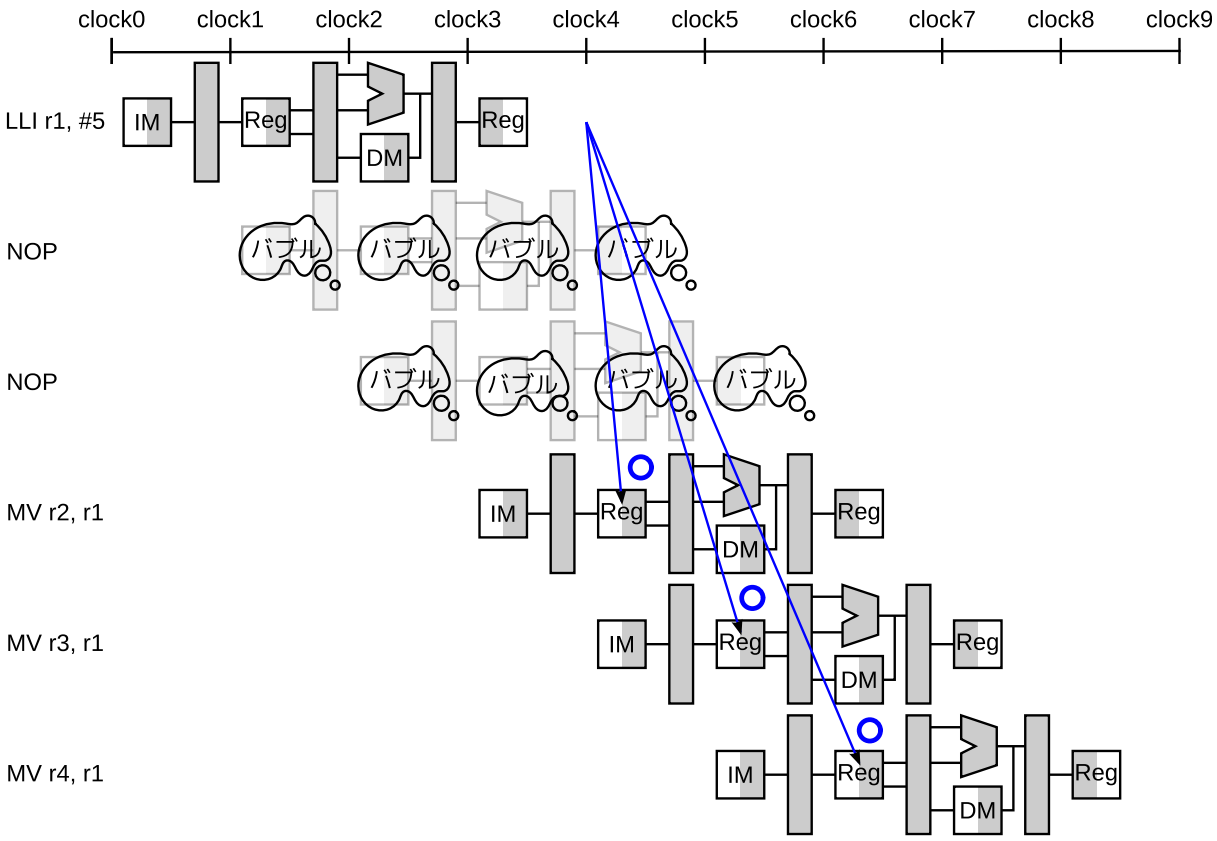

図 3　NOP 命令を挿入しパイプラインをストールさせてデータハザードを回避

うにしてやります．この様子を図 3 に示します．

NOP 命令は何もしない命令なので，これはパイプラインの動作が 2 クロック止まっていることと同じです．このようにパプラインの動作が止めることを，パイプラインをストール（stall）させるといいます．また，NOP 命令を受け取って何もしないステージは，水の流れの中の泡に似ていることからバブル（bubble）とも呼ばれます．NOP 命令を挿入してパイプラインをストールさせる方法は，ハードウェアの変更が一切必要なく簡単ですが，最悪の場合は 3 クロックに 1 個の命令しか処理できなくなってしまい，性能の面で問題があります．

4　解決法その 2： フォワーディング

性能を低下させないでデータハザードを回避するためには，ハードウェアの構成に変更を加えてステージ間にバイパス経路をつくってデータを横流ししてやる必要があります．このような方法をフォワーディング（forwarding）あるいはバイパッシング（bypassing）と呼びます．

図 2 にからも分かるように，RISC16p のデータハザードには，直後の命令にデータを受け渡すときに生じるケースと，2 つ後の命令にデータを受け渡すときに生じるケースの 2 種類があります．このため，データを横流しするための経路もそれぞれのケースに対応した以下の 2 種類が必要になります（図 4）．

（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です．WB ステージで書き込まれるデータは EX ス

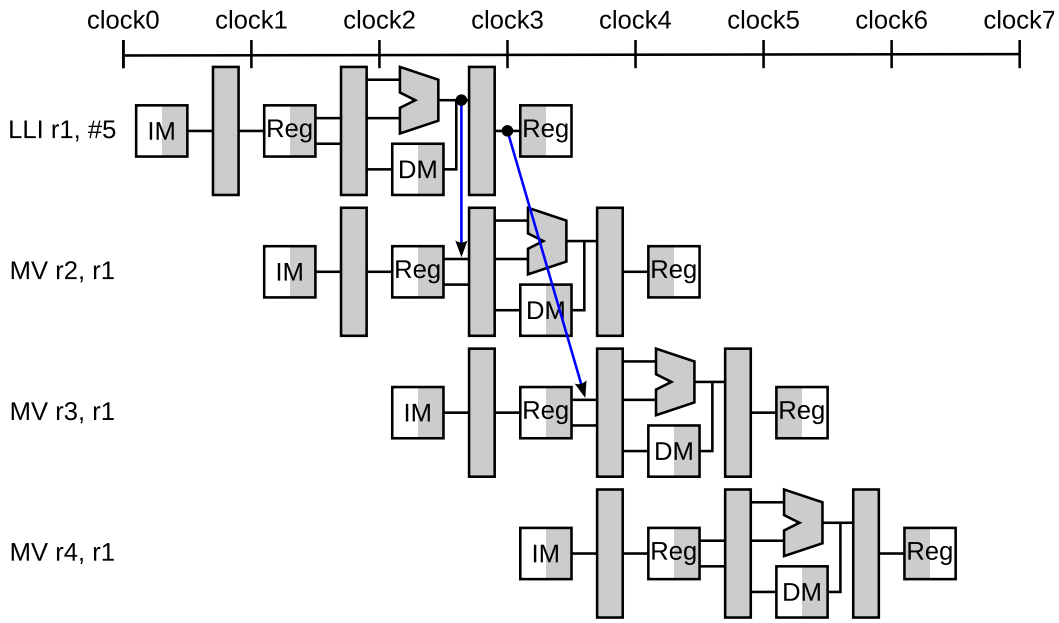

図 4　フォワーディングによってデータハザードを回避

テージですでに計算されるので，これを WB ステージに渡すためのレジスタ（ex_result）に書き込む前に RF ステージへフォワーディングしてやります．

（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ものです．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します．

以上のフォワーディングを備えた RISC16f の構成を図 5 に示します．なお，ここではデータの流れだけを示しており，制御用の信号線は省略しています．RF ステージには，レジスタファイルからの読み出しと 2 つのバイパス経路のからデータを選ぶためのマルチプレクサが追加されます．このマルチプレクサでは，以下の条件でフォワーディング経路からのデータを選択します．

（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

RISC16p では命令中のレジスタフィールドの位置が固定されており，パイプライン段数も 4 段と単純であるため，比較的簡単にフォワーディングを実現することができます．しかし，一般に用いられている段数の多いパイプラインでは，フォワーディング経路とその制御機構はかなり複雑になり，データハザードを完全には除去できない場合もあります．

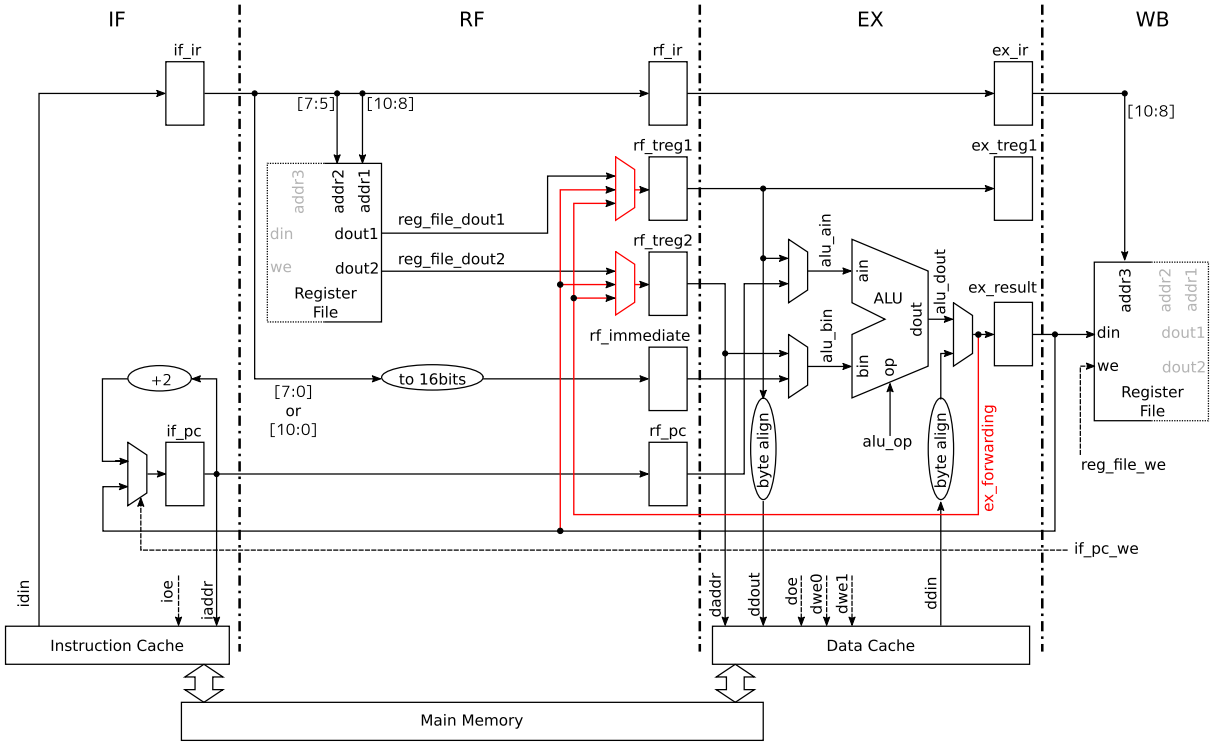

図 5　フォワーディングつきパイプラインプロセッサ RISC16f の構成

練習： EX ステージからのフォワーディング条件と WB ステージからのフォワーディング条件の両方が成り立った場合，RF ステージではどちらのステージからフォワードされるデータを選ぶべきか?