情報工学実験 IV

第 7 回： 制御ハザードと遅延分岐

2019 年 1 月 29 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1 制御ハザードのしくみ

3 つのパイプラインハザードのうち，構造ハザードとデータハザードへの対処はすみました．今回は，唯一残った制御ハザード（control hazard）に対する対策について考えます．

復習になりますが，制御ハザードとは分岐命令がパイプラインの中で PC（プログラムカウンタ）を変更する前に，次々と後続の命令を読んでしまうために起るものです．分岐ハザード（branch hazard）と呼ばれることもあります．それでは制御ハザードの具体的な例を見てみましょう．

```
 0 番地:  JMP  #-2     (11000111 11111110)
 2 番地:  LLI  r1, #1  (00001001 00000001)
 4 番地:  LLI  r2, #2  (00001010 00000010)
 6 番地:  LLI  r3, #3  (00001011 00000011)
 8 番地:  LLI  r4, #4  (00001100 00000100)
10 番地:  LLI  r5, #5  (00001101 00000101)
```

このプログラムは先頭の 0 番地に置かれた JMP 命令が自分自身に無限ループするつもりです．2 番地以降の LLI 命令は実行されないはずです．したがって，どれだけクロックを進めても r1〜r5 レジスタの値はいっさい変更されないはずです．しかし，実際には以下のように不思議な動作を生じます．

```
==== 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:c7fe 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:1100011111111110
 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:0901 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:0000100100000001
 rf_pc:0002 rf_ir:1100011111111110 rf_treg1:0000 rf_treg2:0000 rf_immediate:fffe
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0004 idin:0a02 ioe:1
 alu_ain:0002 alu_bin:fffe alu_op:0100 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:0000101000000010
 rf_pc:0004 rf_ir:0000100100000001 rf_treg1:0000 rf_treg2:0000 rf_immediate:0001
 ex_ir:1100011111111110 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0006 idin:0b03 ioe:1
 alu_ain:0000 alu_bin:0001 alu_op:0001 reg_file_we:0 if_pc_we:1 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 4 ====
 if_pc:0000 if_ir:0000101100000011
 rf_pc:0006 rf_ir:0000101000000010 rf_treg1:0000 rf_treg2:0000 rf_immediate:0002
 ex_ir:0000100100000001 ex_result:0001
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0000 idin:c7fe ioe:1
 alu_ain:0000 alu_bin:0002 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 5 ====
 if_pc:0002 if_ir:1100011111111110
 rf_pc:0000 rf_ir:0000101100000011 rf_treg1:0000 rf_treg2:0000 rf_immediate:0003
 ex_ir:0000101000000010 ex_result:0002
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0002 idin:0901 ioe:1
 alu_ain:0000 alu_bin:0003 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0000 0000 0000 0000 0000 0000

==== clock: 6 ====
 if_pc:0004 if_ir:0000100100000001
```

```
 rf_pc:0002 rf_ir:1100011111111110 rf_treg1:0000 rf_treg2:0000 rf_immediate:fffe
 ex_ir:0000101100000011 ex_result:0003
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0004 idin:0a02 ioe:1
 alu_ain:0002 alu_bin:fffe alu_op:0100 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0002 0000 0000 0000 0000 0000

==== clock: 7 ====
 if_pc:0006 if_ir:0000101000000010
 rf_pc:0004 rf_ir:0000100100000001 rf_treg1:0001 rf_treg2:0000 rf_immediate:0001
 ex_ir:1100011111111110 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0006 idin:0b03 ioe:1
 alu_ain:0001 alu_bin:0001 alu_op:0001 reg_file_we:0 if_pc_we:1 led:000000
 regs: 0000 0001 0002 0003 0000 0000 0000 0000
```

レジスタ r1，r2，r3 の値が変更されています．各クロックで IF ステージの命令レジスタ（if_ir）にフェッチされる命令を見ると，2 番地から 6 番地までの 3 つの LLI 命令がパイプラインに投入されてしまっていることが分かります．

この様子を前回と同じように図示してみると図 1 のようになります．RISC16f では分岐命令（無条件分岐命令も含む）は WB ステージで PC の値を変更するようになっています．したがって，最初の JMP 命令が PC の値を 0 番地に変更するのはクロック 4 の立ち上がりになります（図 1 の 1 段目）．これにより，クロック 4 でフェッチされるのは再び 0 番地の JMP 命令になりますが，この間，クロック 1 からクロック 3 では 2 番地から 6 番地までの 3 つの LLI 命令がフェッチされ実行されてしまっているのです．

制御ハザードの簡単な解決法は，分岐命令の直後の 3 命令は必ず NOP にするという方法です（図 2）．しかし，この方法では分岐命令が連続するような場合は，最悪 4 クロックに 1 命令しか処理できず，パイプライン化していないのと同じことになってしまいます．

2　分岐ペナルティの削減

そもそもパイプラインは，命令が順番に実行されることを前提として次々と命令をフェッチするわけですから，分岐命令に関する問題は本質的だといえます．効率を上げるためにはハードウェアに変更を加えて，

- 分岐命令が来たら，なるべく早く「飛ぶ」のか「飛ばない」のかを判断する[*1]
- 飛ぶ場合にはなるべく早く PC を変更する

しかありません．フェッチした命令が分岐命令かどうかは，RF ステージで調べることができます．そこで，

（1）フェッチした命令が分岐命令かどうか，分岐の条件を満たしているかどうかの判断
（2）飛び先のアドレスの計算

[*1] 分岐の条件が成立して PC を変更する（「飛ぶ」）ことを “taken”，「飛ばない」ことを “untaken”（または “not taken”）といいます．

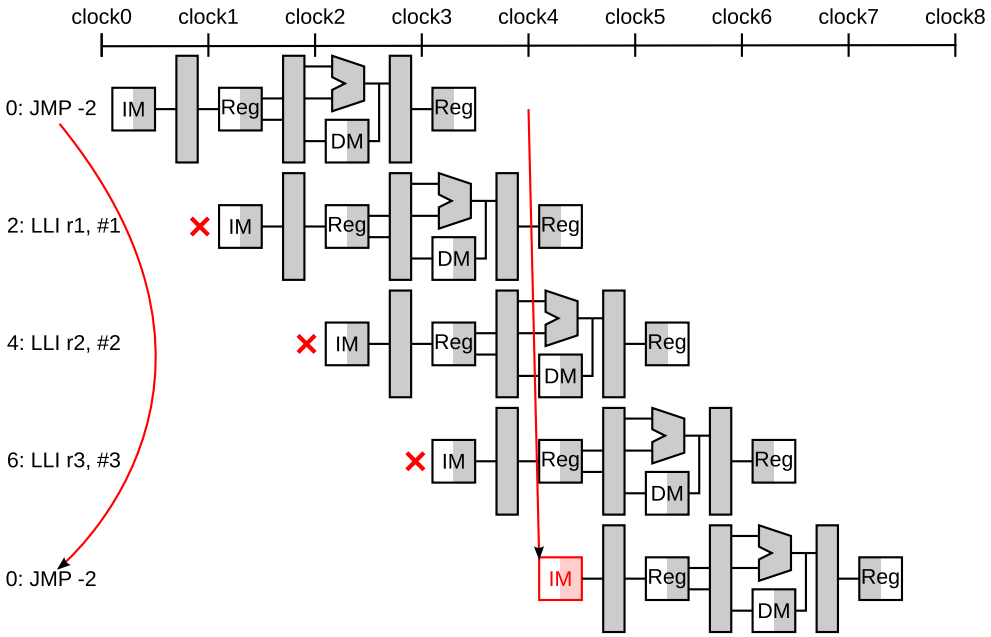

図 1　コントロールハザードの例

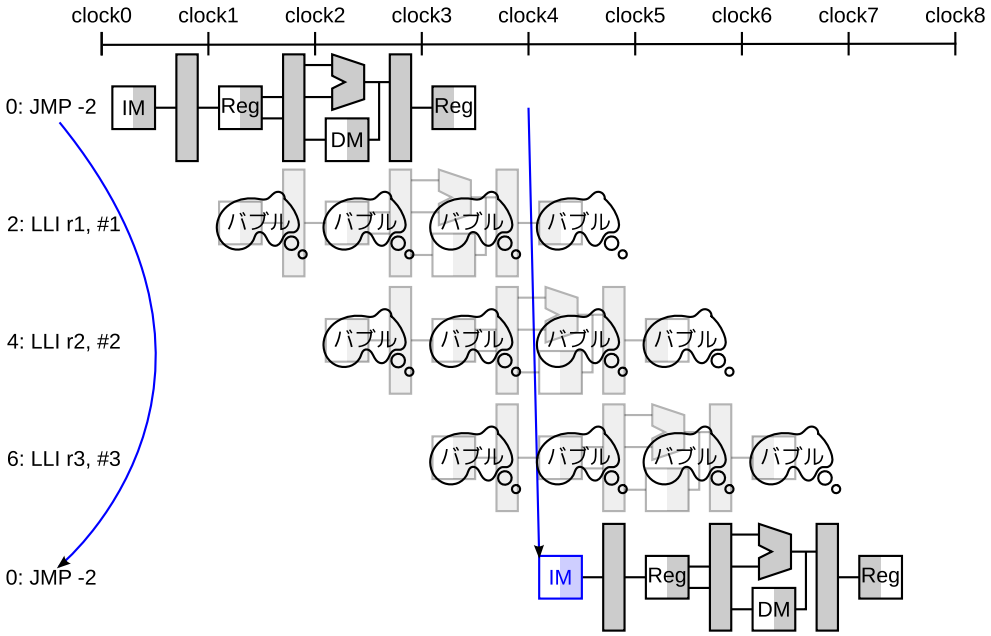

図 2　NOP 命令を挿入し制御ハザードを回避

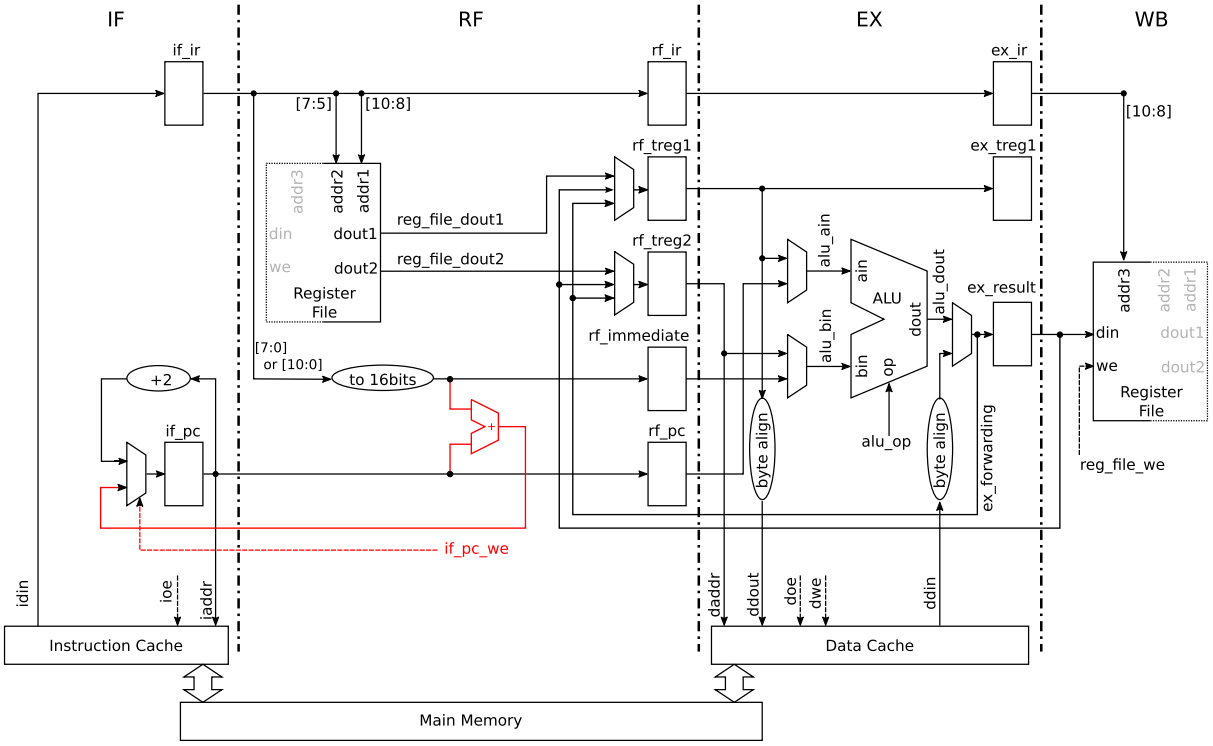

図 3　分岐ペナルティを削減したパイプラインプロセッサ RISC16b の構成

の両方を RF ステージで行うことにすれば最短 2 クロックで分岐を行うことができます．まず，(2) の飛び先のアドレスの計算のために，命令レジスタのオフセットフィールドとプログラムカウンタを加算するための専用の加算器を RF ステージに用意します（図 3)．

また，(1) については，命令レジスタを見て分岐命令ならすぐに判定用のレジスタの値を読み出し，それが分岐条件を満たしているか（たとえば BEQZ なら 0 かどうか）をチェックします．ここで注意が必要なのは，**判定用のレジスタの値は他のステージからフォワードされてくるかもしれない**という点です．したがって，分岐するかどうかは，RF ステージの rf_treg1 レジスタの直前にあるマルチプレクサの出口の値を使って判定する必要があります．また，分岐条件が満たされたときだけ PC を変更するので，上記の (1) と (2) の動作はそれぞれ独立に行うことができます．つまり，飛ぶか飛ばないかは分からないけど，とりあえず飛び先アドレスは計算しておこうというわけです．こうすれば，飛ぶことが確定したらすぐに PC を変更することができます．

改良されたパイプライン上での分岐命令の実行の様子を図 4 に示します．今回は PC がクロック 2 の立ち上がりで分岐先のアドレスに変更されるため，分岐命令の後には NOP 命令を 1 個挿入するだけで済みます．今回の改造では，新たに分岐先アドレスの計算のために専用の加算器が必要になったり，フォワードされたレジスタの値を判定して PC に飛び先アドレスを設定するまでを 1 クロックで行うため，回路の最大遅延時間は延びてしまう（動作周波数が低下してしまう）ことが予想されます．しかし，図 2 と比較すると，分岐命令を実行するときの効率は格段に向上していることが分かります．

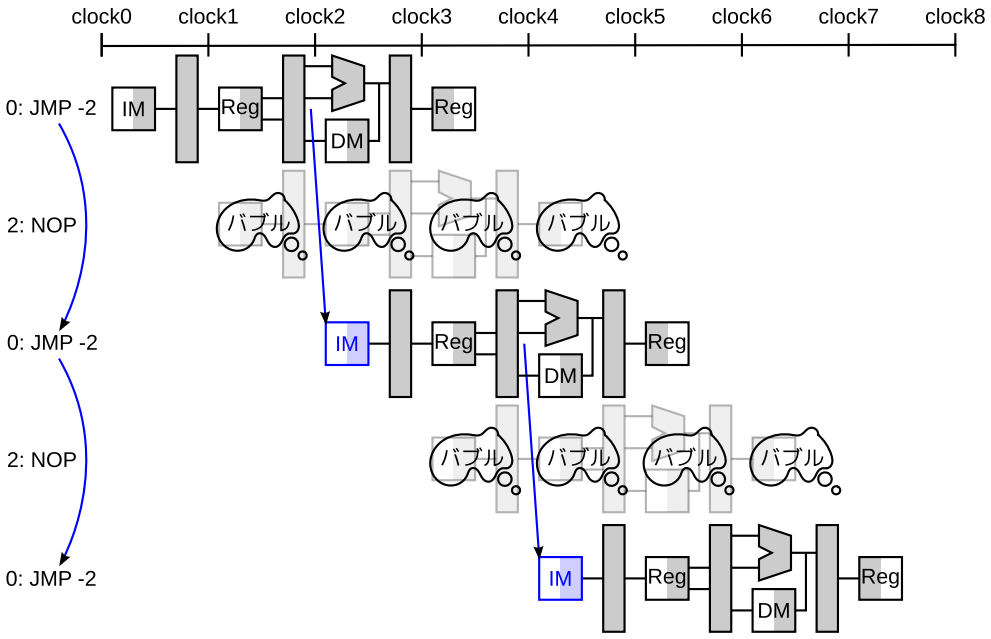

図 4　分岐ペナルティを削減したパイプライン上での実行の様子

3　遅延分岐と遅延スロット

さて，図 4 からも明らかなように，以上に説明したハードウェアの変更を行っても，残念ながら分岐命令の次には NOP を 1 個挿入する必要があります．しかし，これ以上分岐の決断を早く行うのは困難です．分岐命令の直後は NOP を入れて無駄にするしかないのでしょうか?

この問題を解決するために考え出されたのが「**遅延分岐（delayed branch）**」です．これはハードウェアに変更を加えるのではなく，発想の転換をして，最初からプロセッサの分岐というものは 1 命令遅れて起こるのだと考えるのです．

分岐命令より後にあるのに分岐の成立・不成立に関係なく実行されてしまう命令を遅延スロットと呼びます．今回のように分岐のときに後ろの命令 1 個がかならず実行される場合，遅延スロット数が 1 であるといいます．分岐の仕組みを改良する前の構成では，遅延スロット数が 3 だったわけです．

命令の順番をうまく工夫して並べ替えると，ほとんどの場合遅延スロットに NOP ではなく意味のある命令を割り当てることができます．この作業はコードスケジューリングとかパイプラインスケジューリングなどと呼ばれますが，通常はコンパイラによって自動的に行われています．

現在使われているほとんどの高性能プロセッサは，遅延スロットを持っています．この遅延スロットを有効活用できるかどうかは実行させるアプリケーションにも依存しますが，現在のコンパイラ技術はスロット数が 1 の場合は 80% から 90% のスロットを有効な命令で埋めることができます．最近のプロセッサでは，分岐命令のペナルティをさらに低減するために，taken なのか untaken なのかをハードウェアで予測する方法（分岐予測）が用いられています．さらに，分岐を予測するだけでなく，予測に基づいて分岐した後の命令群も実行しておき，予測が外れたらキャンセルしてやり直すという一種のギャンブルを行う手法（投機的実行）も導入されています．これらについては 4 年生の選択科目「コンピュータアーキテクチャ II」で詳しく解説します．