情報工学実験 III
第 5 回： 16 ビット RISC マシンのパイプライン化

柴田裕一郎

shibata@cis.nagasaki-u.ac.jp
情報工学コース

2019 年 1 月 8 日

16 ビット RISC のパイプライン化

- パイプライン処理の基礎
- 構造ハザードとパイプラインストール
- RISC16p: パイプライン化した RISC16
  - パイプラインステージの設計
  - 構造ハザード除去に伴うハードウェアの変更

パイプライン処理

- 工場におけるベルトコンベア式の流れ作業に類似
- 1 つの命令の処理を複数のステップ（ステージ）に分割
- 各ステージの処理に専門の担当（ハードウェア）を配置
    - 直前のステージからデータを受けとり，処理を行なって，次のステージに渡す
    - たとえば「命令フェッチ」のステージでは毎クロックひたすら命令をフェッチして次のステージに渡す
    - 複数の命令がオーバーラップして処理される

パイプラインと性能

- 命令がフェッチされてから実行が完了するまでの時間（レイテンシ）はほとんど変わらない
  - むしろ悪化することもある
- 単位時間あたりの命令の実行数（スループット）は向上する
  - 1 クロック毎に新しい命令がフェッチされ実行される

$\Rightarrow$ 全体の性能はかなり向上する

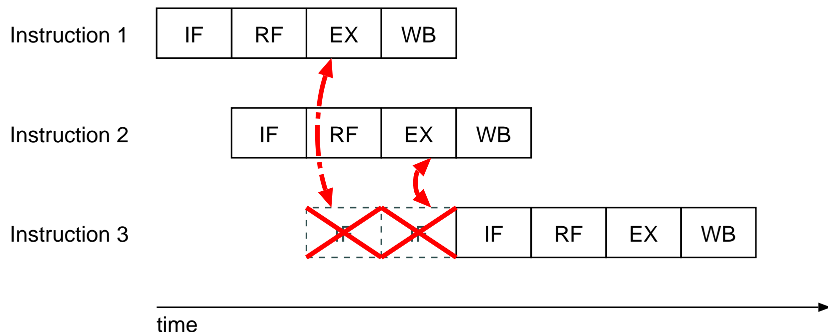
効率の良いパイプライン処理のために

- 各ステージの処理時間をなるべく等しくする
  - あるステージだけ早く終っても，その前のステージの処理が終らないと次の命令を処理できず待ち状態に···
- 複数のステージで同じハードウェア資源を使用しない
  - 同じハードウェアを使うステージは同時に実行できない！
  - 構造ハザード（structural hazard）が発生

RISC16p の設計

- 命令セットは RISC16 のまま
- 均等化のためすべての命令を 4 ステージで実行
  1. 命令フェッチ（IF: Instruction Fetch）
     メモリから命令を取ってくる．同時に PC（プログラムカウンタ）に 2 を加える．
  2. レジスタフェッチ（RF: Register Fetch）
     レジスタファイルから必要なデータを読み出す．
  3. 実行（EX: EXecution）
     ALU を使って計算処理を行なう．LD/ST 系の命令ではメモリへのアクセスを行なう．
  4. 書き戻し（WB: Write Back）
     計算結果を指定されたレジスタに書き込む．分岐成立時には PC を変更する．

構造ハザードによるパイプラインストール



「命令フェッチ（IF)」と「実行（EX)」の両方で ALU を使う場合のパイプライン

構造ハザードの除去

- ALU（IF および EX）
  - IF ステージに PC に 2 を足すための専用の加算器を付加
- メモリ（IF および EX）
  - 1 次キャッシュをデータ用と命令用に分離
  - ハーバードアーキテクチャ（Harvard architecture）
- レジスタファイル（RF および WB）
  - レジスタファイルを 1 入力 2 出力の 3 ポートメモリに変更

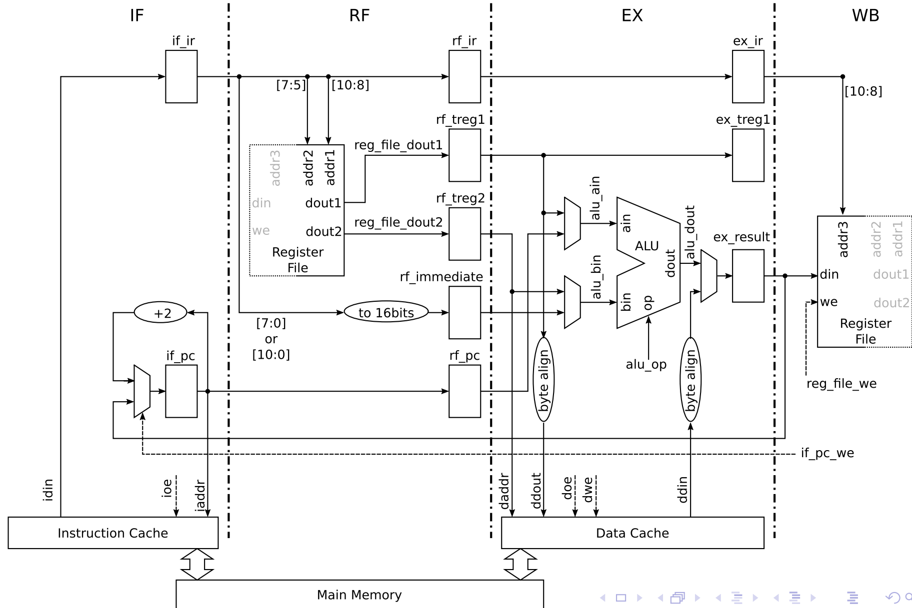
トレードオフ

- 構造ハザード ⇒ パイプラインストール（性能低下）

- 構造ハザードの除去 ⇒ ハードウェアコストの増大

  - 常に性能とハードウェアコストを秤にかけながらアーキテクチャを決める必要
  - 今回の設計では構造ハザードはすべて除去

RISC16p の構成



RISC16p の制御信号

信号名	タイプ	ステージ	役割
`ioe`	出力	IF	データメモリ読み出し
`doe`	出力	EX	データメモリ読み出し
`dwe`	出力	EX	データメモリ書き込み
`reg_file_we`	内部信号	WB	レジスタファイル書き込み
`if_pc_we`	内部信号	WB	PC に飛び先番地を書き込み

IF ステージの処理

- プログラムカウンタ（if_pc）を命令メモリ用アドレスバス（if_addr）に繋いで読み出し制御信号（ioe）を '1' にする
  - これで命令が命令レジスタ（if_ir）にフェッチされる
- 同時に専用の加算器を使って PC の値に 2 を加える
- 分岐成立時には WB ステージで if_pc_we 信号が '1' になる
  - この場合には，ex_result レジスタの値（計算された飛び先のアドレス）が if_pc にセットされる

RF ステージの処理

- IF ステージから現在のプログラムカウンタとフェッチされた命令（if_pc と if_ir）を受取る
- レジスタ番号のフィールド（if_ir[10:8] と if_ir[7:5]）を使ってレジスタファイルからオペランドを読み出す
  - rf_treg1 と rf_treg2 に書き込まれる
- 即値フィールド（if_ir[7:0]，JUMP では if_ir[10:0]）
  - 16 ビットに拡張して即値用レジスタ rf_immediate に格納
  - ADDI 命令，分岐命令，JUMP 命令 ⇒ 符号拡張
  - その他の命令 ⇒ ゼロ拡張
- if_pc と if_ir の内容はそのまま rf_pc と rf_ir に代入して EX ステージへ渡す

EX ステージの処理

- RF ステージから受け取ったデータを ALU で処理
  - 命令に応じて alu_ain，alu_bin，alu_op に適切な信号を接続
- LD/ST 系の命令ではデータメモリにアクセス
- 演算結果を ex_result に書き込む
- WB ステージには rf_ir と rf_treg1 の内容をそのまま ex_ir と ex_treg1 に代入して渡す

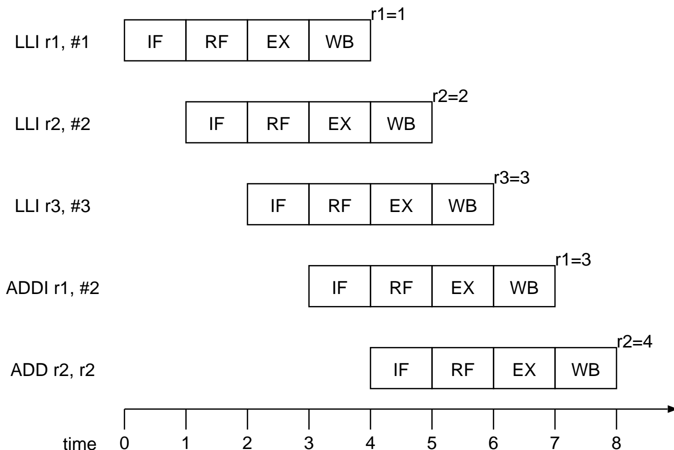
WB ステージの処理

- 命令（ex_ir）を見て，レジスタファイルに書き戻しを行う必要があるかを判断
  - 分岐，JUMP，ST，SB，NOP では必要なし
  - それ以外の命令では reg_file_we を '1' にする
  - ex_result の値が ex_ir[10:7] で示されたレジスタへ書き込まれる
- 分岐命令では ex_treg1 レジスタの値に基づき分岐が成立するかどうかを判断
  - 分岐が成立した場合には，if_pc_we を '1' にする
  - ex_result レジスタに入っている飛び先番地が if_pc にセットされる
  - JUMP 命令の場合には ex_treg1 の値に関わらず常に if_pc_we を '1' にする

プログラム例

```
@00 00001001 00000001 // LLI  r1,   #1
@02 00001010 00000010 // LLI  r2,   #2
@04 00001011 00000011 // LLI  r3,   #3
@06 00100001 00000010 // ADDI r1,   #2
@08 00000010 01000100 // ADD  r2,   r2
```

サンプルプログラムの実行の様子



実行例（1/4）

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

実行例（2/4）

```
==== clock: 3 ====
 if_pc:0006 if_ir:0000101100000011
 rf_pc:0004 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:0006 idin:2102 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: 4 ====
 if_pc:0008 if_ir:0010000100000010
 rf_pc:0006 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:0008 idin:0244 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: 5 ====
 if_pc:000a if_ir:0000001001000100
 rf_pc:0008 rf_ir:0010000100000010 rf_treg1:0001 rf_treg2:0000 rf_immediate:0002
 ex_ir:0000101100000011 ex_result:0003
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000a idin:0000 ioe:1
 alu_ain:0001 alu_bin:0002 alu_op:0100 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0002 0000 0000 0000 0000 0000
```

実行例（3/4）

```
==== clock: 6 ====
 if_pc:000c if_ir:0000000000000000
 rf_pc:000a rf_ir:0000001001000100 rf_treg1:0002 rf_treg2:0002 rf_immediate:0044
 ex_ir:0010000100000010 ex_result:0003
 daddr:0002 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000c idin:0000 ioe:1
 alu_ain:0002 alu_bin:0002 alu_op:0100 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0002 0003 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:0000001001000100 ex_result:0004
 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:1 if_pc_we:0 led:000000
 regs: 0000 0003 0002 0003 0000 0000 0000 0000

==== clock: 8 ====
 if_pc:0010 if_ir:0000000000000000
 rf_pc:000e 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:0010 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 0003 0004 0003 0000 0000 0000 0000
```

実行例（4/4）

```
==== clock: 9 ====
 if_pc:0012 if_ir:0000000000000000
 rf_pc:0010 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 dwe0:0 dwe1:0
 iaddr:0012 idin:xxxx ioe:1
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0003 0004 0003 0000 0000 0000 0000
```

- 初めの 3 クロックはレジスタの値の変化なし
- 4 クロック目で最初のデータがレジスタに格納
- それ以降は 1 クロックごとに命令の実行が完了している