

第4章 プロセッサ・アーキテクチャ(1)

大阪大学 大学院 情報科学研究科
今井 正治

arch-2014@vlsilab.ics.es.osaka-u.ac.jp

講義内容(1)

☐ 実装方式の概要

- ☐ 論理設計とクロック方式
- ☐ データパスの構築
- ☐ 単純な実現方法
- ☐ パイプライン処理の概要

講義内容(2)

- ☐ データパスのパイプライン化と制御
- ☐ データ・ハザード: フォワーディングとストール
- ☐ 制御ハザード
- ☐ 例外
- ☐ 並列処理と高度な命令レベル並列性
- ☐ 誤信と落とし穴

使用する主要なMIPS命令セットの仮定

- ☐ メモリ参照命令
 - load word (lw)
 - store word (sw)
- ☐ 算術論理演算命令
 - add, sub, and, or, slt (set on less than)
- ☐ 分岐命令
 - branch equal (beq)
 - jump (j)

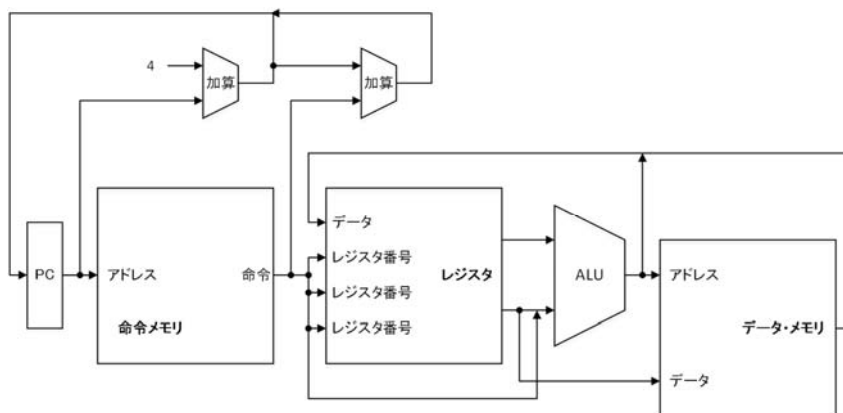
MIPSでの命令実行の5つのステップ

- IF: 命令フェッチ (Instruction Fetch)
- ID: 命令デコード (Instruction Decode)
- EX: 演算の実行 (Execution)
- MEM: メモリアクセス (Memory Access)
- WB: レジスタ書き込み (Write Back to Register)

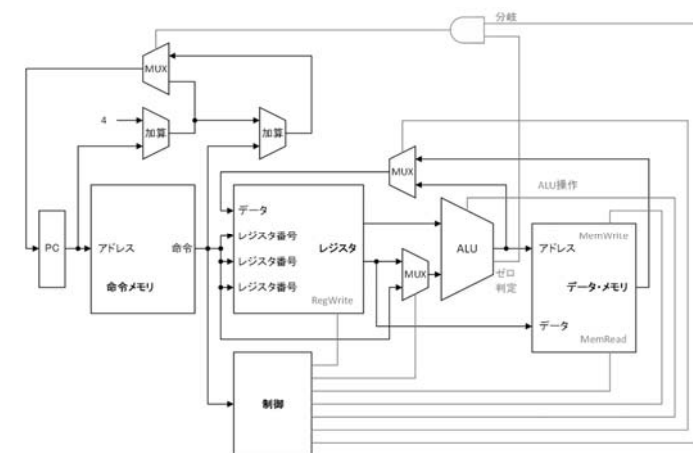
全ての命令に共通の最初の2つのステップ

- IF (命令フェッチ)
 - メモリから命令をフェッチするために、プログラム・カウンタ(PC)の値を命令(コード)が保持されているメモリに送る
 - PCの値をインクリメントする($PC = PC + 4$)
- DE (命令デコード)
 - 命令のレジスタ・フィールドに指定されている1つまたは2つのレジスタを読み出す
 - ロード命令(lw)の場合は、読みだす必要があるレジスタは1つだけ
 - その他の命令に関しては、すべて2つのレジスタ

MIPS命令の実現方式の概念図



マルチプレクサ, 制御線を追加したMIPS 命令のサブセットの基本的な実現方式



講義内容(1)

- 実装方式の概要
- 論理設計とクロック方式
- データパスの構築
- 単純な実現方法
- パイプライン処理の概要

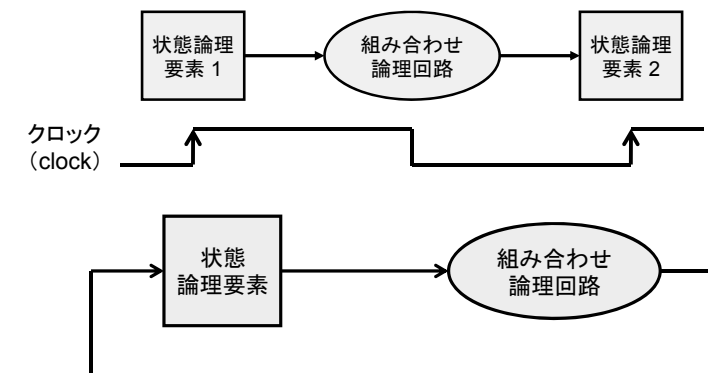
論理設計

- 組合せ論理要素 (combinational element)
 - 出力が現在の入力のみによって決まる論理要素
 - 論理ゲート, マルチプレクサ, ALUなどの処理要素
- 状態論理要素 (state element)
 - 状態 (state) を記憶する
 - 出力と次の時刻の内部状態は現在の内部状態と入力によって決まる
 - 命令メモリ, データメモリ, レジスタは状態論理要素
- アサート (assert) とネゲート (negate, deassert)
 - アサート: 信号を論理的に高く (high) 設定する
 - ネゲート: 信号を論理的に低く (low) 設定する

クロック方式

- エッジ・トリガ・クロック方式 (edge-triggered clocking methodology) を仮定
- 1クロック・サイクル内で以下の操作を行う
 - レジスタ内容を読み出す
 - 組合せ論理回路で演算
 - 結果をレジスタに書き込む
- 制御信号 (control signal)
 - クロック・エッジで行うべき動作を指定する信号例: データの読み込み, シフト, 演算機能

エッジ・トリガ・クロック方式を用いた状態論理要素の実現



講義内容(1)

- 実装方式の概要
- 論理設計とクロック方式
- データパスの構築
- 単純な実現方法
- パイプライン処理の概要

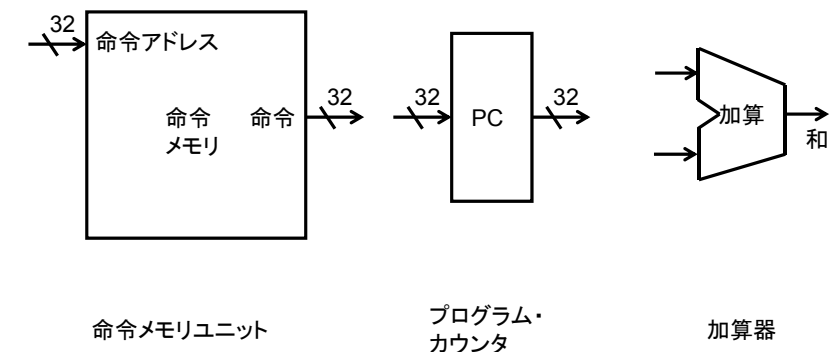
データパス要素(datapath element)

- データパスを構成する論理要素
- データパス要素の例
 - メモリ・ユニット(memory unit)
 - 命令メモリ(instruction memory)
 - データ・メモリ(data memory)
 - プログラムカウンタ(program counter: PC)
 - ALU(arithmetic logic unit)
 - 加算器(adder)

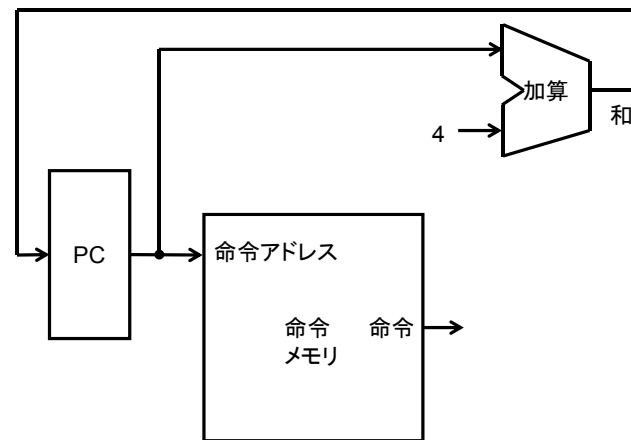
命令フェッチ(instruction fetch)

- 命令メモリ(instruction memory)から命令(instruction)を読み出す
 - 命令のアドレスは, プログラム・カウンタ(program counter: PC)で指定
- プログラム・カウンタを更新
 - 次の命令を読み出すための準備
 - プログラム・カウンタの値に4を加える
MIPSでは, 命令長はすべて1語(=4バイト)

命令フェッチに必要なデータパス要素



データパスのうち、命令をフェッチしてプログラム・カウンタを繰り上げる部分



算術論理演算命令の実装

□ R形式命令 (R-type instruction)

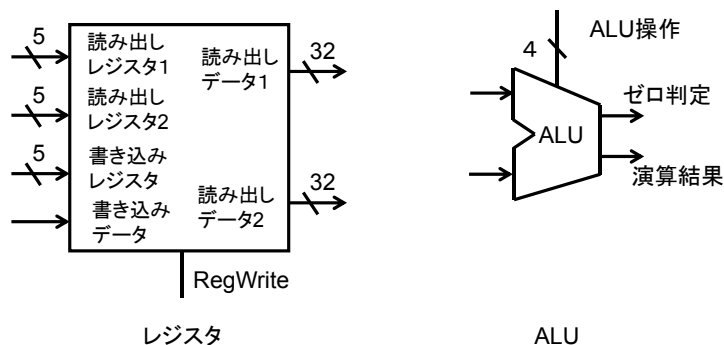
- 2つのレジスタの値を読み出し、結果をレジスタに書き込む

■ 例: `add $t1, $t2, $t3`

□ レジスタ・ファイル (register file)

- レジスタの集合
- レジスタ番号を指定することによって任意のレジスタの読み込み、書き込みが可能
- レジスタが32個の場合、レジスタ番号は5ビット

R形式のALU演算を実現するために必要なデータパス要素



ロード、ストア命令の実現

□ 命令の形式

- `lw $t1, offset($t2)`
- `sw $t1, offset($t2)`

□ アクセスするメモリ・アドレスの計算

- ベース・レジスタ(\$t2)が保持している値にオフセット (16bit の符号なし数)を加算する

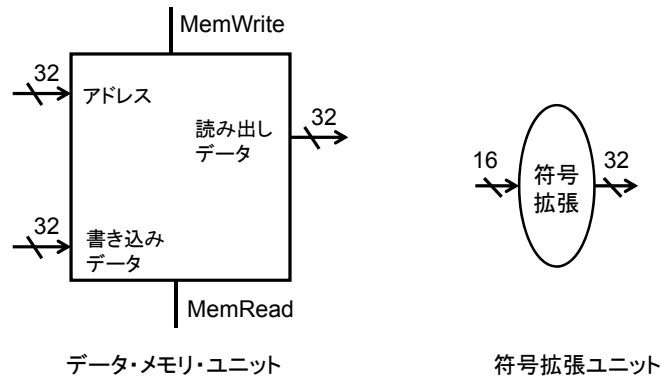
□ sw命令の場合

- メモリに書き込むデータは \$t1 の内容

□ 必要な演算ユニット

- 加算器 (adder), 符号拡張ユニット (sign extender)

ロード, ストア命令の実現



分岐命令 (beq) の実現 (1)

□ 命令の形式

- `beq $t1, $t2, offset`

□ 動作

- 2つのレジスタ `$t1` と `$t2` の内容が等しければ, `offset` フィールドの値を符号拡張して, PCに加算する

□ 分岐先のアドレスの計算方法

- ベースアドレス (base address) は, 命令フェッチユニットが保持している `PC+4` の値
- `PC+4` に `offset*4` の値を加えることにより分岐

分岐命令 (beq) の実現 (2)

- MIPSアーキテクチャでは, `offset` フィールドは語アドレスでのオフセット値として用いられる
 - バイトアドレスとしては, `offset` フィールドの値を2ビット左シフトし, 符号拡張すれば良い
 - これによって, オフセットフィールドの有効範囲を4倍に拡大できる
- 2つのオペランドの内容が等しいとき, 分岐先のアドレスは $PC+4+(\text{offset} \times 4)$
- 等しくないときは次の命令 (アドレスは `PC+4`) を実行
- 必要な演算ユニット
 - 比較器 (comparator) (ALUを使用)
 - 加算器 (adder), 符号拡張ユニット (sign extender), シフタ (shifter)

符号拡張ユニット

□ 入力

- 16ビットの符号付きまたは符号なし数

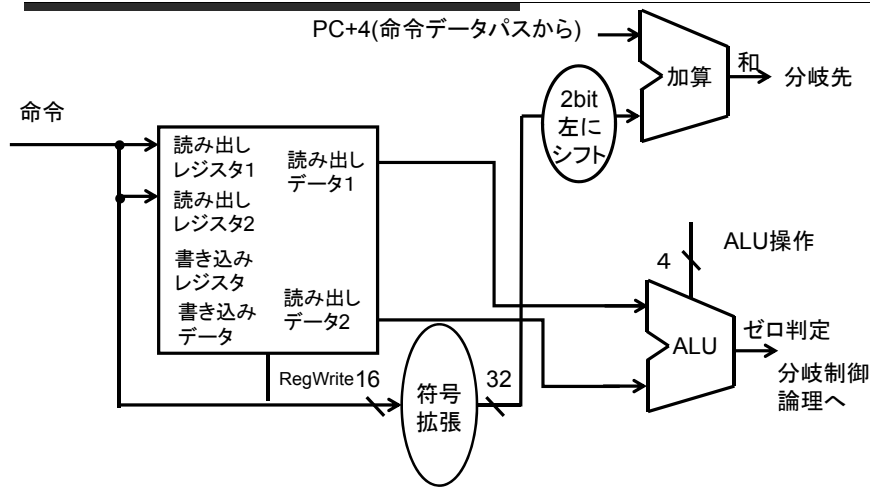
□ 出力

- 32ビットの符号なし数

□ 演算モード

- 符号なしモード (unsigned mode)
 - 上位16ビットに 0 を埋める
- 符号ありモード (signed mode)
 - 上位16ビットを入力の前16ビットで埋める

分岐命令用のデータパス

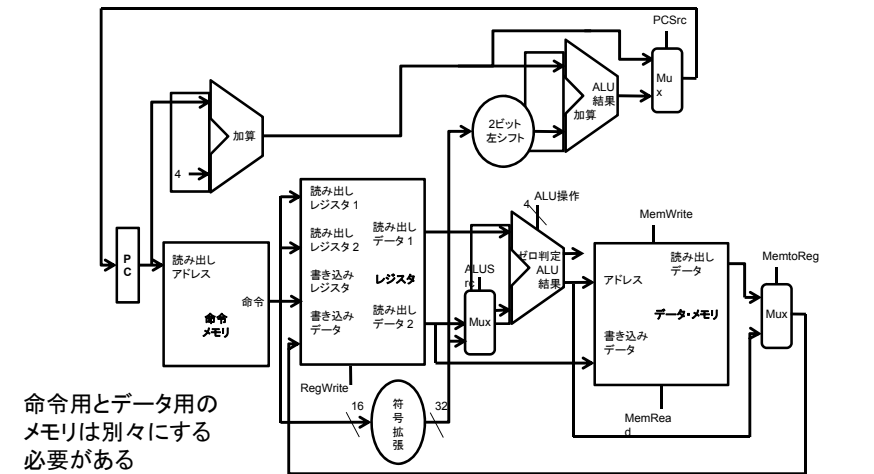


2014/12/02

©2014, Masaharu Imai

25

MIPSアーキテクチャの単純なデータパス



2014/12/02

©2014, Masaharu Imai

26

講義内容(1)

- ☐ 実装方式の概要
- ☐ 論理設計とクロック方式
- ☐ データパスの構築
- ☒ 単純な実現方法
- ☐ パイプライン処理の概要

2014/12/02

©2014, Masaharu Imai

27

MIPSアーキテクチャの実現

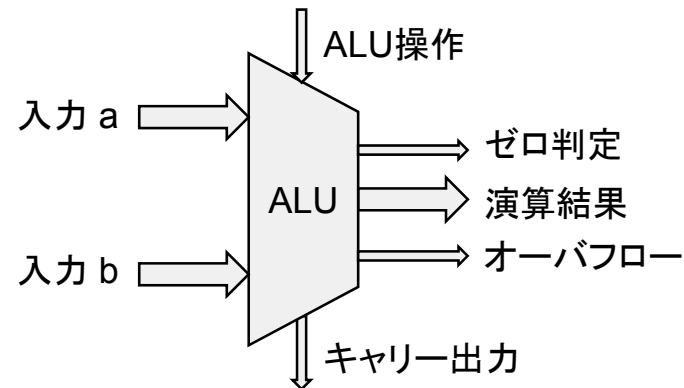
- ☐ 対象とする命令
 - lw (load word), sw (store word)
 - beq (branch equal), j (jump)
 - add (addition), sub (subtraction)
 - and (logical and), or (logical or)
 - slt (set on less than)
- ☐ 設計フロー
 - ALU制御ユニットの仕様
 - ALU制御ビットの構成
 - ALU制御ビットの決定

2014/12/02

©2014, Masaharu Imai

28

ALU



2014/12/02

©2014, Masaharu Imai

29

ALU制御ユニットの仕様

ALU制御入力	機能
0000	AND
0001	OR
0010	加算
0110	減算
0111	Set on less than
1100	NOR

2014/12/02

©2014, Masaharu Imai

30

ALU制御ビットの構成

命令操作コード	ALUOp(制御フィールド)	命令操作	機能コード(funcnt)	実行する演算	ALU制御コード
LW	00	Load word	XXXXXX	Add	0010
SW	00	Store word	XXXXXX	Add	0010
Branch equal	01	Branch equal	XXXXXX	Subtract	0110
R形式	10	Add	100000	Add	0010
R形式	10	Subtract	100010	Subtract	0110
R形式	10	AND	100100	And	0000
R形式	10	OR	100101	Or	0001
R形式	10	Set on less than	101010	Set on less than	0111

ALUOp: 命令中の制御フィールド(2ビット), 主制御ユニットで使用

2014/12/02

©2014, Masaharu Imai

31

ALUOpと機能コードの組み合わせによって決まるALU制御コードの真理値表

ALUOp		機能コード						ALU制御入力 (操作ビット)
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

X: ドントケア(don't care); ALUOp は, 11 という値を取らない

2014/12/02

©2014, Masaharu Imai

32

命令操作コード

フィールド	0	rs	rt	rd	shamt	funct
ビット位置	31-26	25-21	20-16	15-11	10-6	5-0

(a) R形式命令

フィールド	35 or 43	rs	rd	address
ビット位置	31-26	25-21	20-16	15-0

(b) ロード/ストア命令

フィールド	4	rs	rt	address
ビット位置	31-26	25-21	20-16	15-0

(c) 分岐命令

2014/12/02

©2014, Masaharu Imai

33

主制御ユニットの設計

□ 設計手順

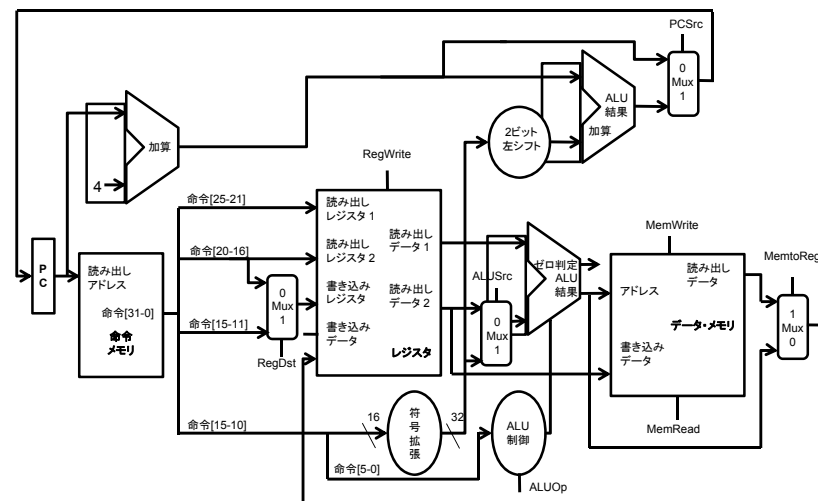
- 命令操作コード(opcode)の抽出(Op[5-0])
 - 命令中のビット位置: 31~26
- 入力レジスタ(rs, rt)の抽出(R形式命令, beq命令)
 - 命令中のビット位置: 25~21, 20~16
- ロード命令およびストア命令用ベースレジスタの抽出
 - 命令中のビット位置: 25~21
- beq, lw, sw命令でのオフセット値
 - 命令中のビット位置: 15~0
- デスティネーション・レジスタ(rd)の抽出
 - ロード命令の場合: ビット位置 20~16
 - R形式命令の場合: ビット位置 15~11

2014/12/02

©2014, Masaharu Imai

34

必要なマルチプレクサと制御線を付け加えたデータパス



2014/12/02

©2014, Masaharu Imai

35

ネゲート: 信号が論理的に低い, または偽であること
アサート: 信号が論理的に高い, または真であること

7つの制御信号の機能

信号名	ネゲートされた時の働き	アサートされた時の働き
RegDst	書き込みレジスタのデスティネーション・レジスタ番号がrtフィールドから得られる	書き込みレジスタのデスティネーション・レジスタ番号がrdフィールドから得られる
RegWrite	なし	書き込みレジスタ入力に指定されているレジスタに書き込みデータ入力の値が書き込まれる
ALUSrc	ALUの第2オペランドがレジスタ・ファイルの第2出力から得られる	ALUの第2オペランドが命令の下位16ビットを符号拡張したものになる
PCSrc	PC+4を計算した加算器の出力によってPCが置き換えられる	分岐先を計算した加算器の出力によってPCが置き換えられる
MemRead	なし	読み出しアドレスによって指定されたデータ・メモリの内容が読み出しデータ出力上に流される
MemWrite	なし	書き込みアドレスによって指定されるアドレス上にあるデータ・メモリの内容が書き込みデータ入力の値によって書き換えられる
MemtoReg	レジスタの書き込みデータ入力へ渡される値がALUから得られる	レジスタの書き込みデータ入力へ渡される値がデータ・メモリから得られる

2014/12/02

©2014, Masaharu Imai

36

データパスの動作

1. 命令が命令メモリからフェッチされ、PCが繰り上げられる。
2. \$t2と\$t3の2つのレジスタの値がレジスタ・ファイルから読み出される。
このステップ間に、主制御ユニットは制御線の設定をどうするか算出する。
3. 機能コードから、ALUの演算機能を決定し、レジスタ・ファイルから読みだされたデータを操作する
4. ALUで処理した結果をレジスタ・ファイルに書き込む。
デスティネーション・レジスタ(\$t1)の選択のために、命令ビット15-11が使用される

add \$t1, \$t2, \$t3の動作

1. 命令が命令メモリからフェッチされ、PCが繰り上げられる
2. レジスタ・ファイルから\$t2, \$t3の2つのレジスタの値が読みだされる。このステップの間に、主制御ユニットは制御線の設定をどうするかを算出する
3. 機能コード(命令のビット5-0つまりfunctフィールド)からALU機能を決定し、レジスタ・ファイルから読みだされたデータを操作する
4. ALUで処理した結果をレジスタ・ファイルに書き込む。
デスティネーション・レジスタ(\$t1)の選択のために、命令のビット15-11が使用される

lw \$t1, offset(\$t2)の動作

1. 命令が命令メモリからフェッチされ、PCが繰り上げられる
2. レジスタ・ファイルからレジスタ(\$t2)の値が読みだされる
3. レジスタ・ファイルから読みだされた値と命令の下位16bitを符号拡張した値の和がALUによって計算される
4. その和がデータ・メモリ用のアドレスとして使用される
5. メモリ・ユニットからデータが読みだされてレジスタ・ファイルに書き込まれるレジスタのデスティネーションは命令ビット20-16(\$t1)によって指定される

beq \$t1, \$t2, offsetの動作

1. 命令が命令メモリからフェッチされ、PCが繰り上げられる
2. レジスタ・ファイルから\$t1, \$t2の2つのレジスタの値が読みだされる
3. レジスタ・ファイルから読みだされた値の差がALUによって計算される。PC+4の値が命令の下位16ビット(offset)を2ビット左にシフトして符号拡張した値に加えられ、その結果が分岐先アドレスとされる
4. ALUのゼロ判定出力に基づいて、どちらかの加算器の結果をPCに格納するかが決定される

制御信号の真理値表(1/2)

入力/出力	信号名	R形式	lw	sw	beq
入力	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0

制御信号の真理値表(2/2)

入力/出力	信号名	R形式	lw	sw	beq
出力	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Fig. 4.11 MIPSアーキテクチャの単純なデータパス

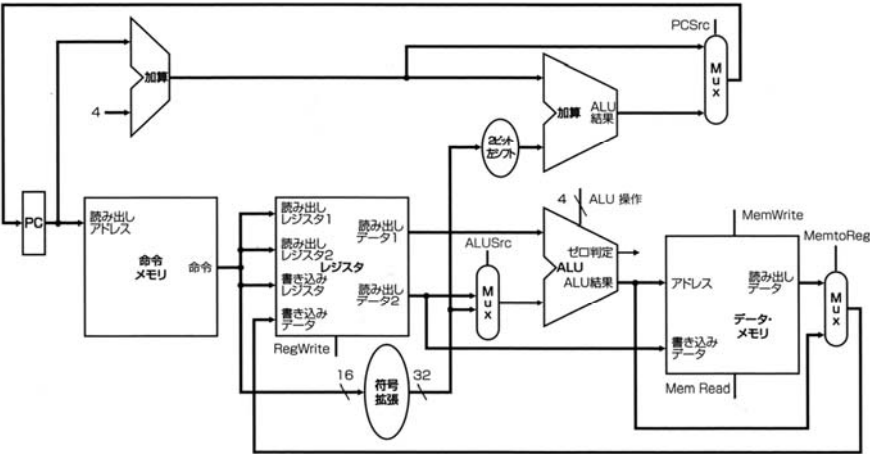


Fig. 4.15 必要なすべてのマルチプレクサと制御線を付け加えたデータパス

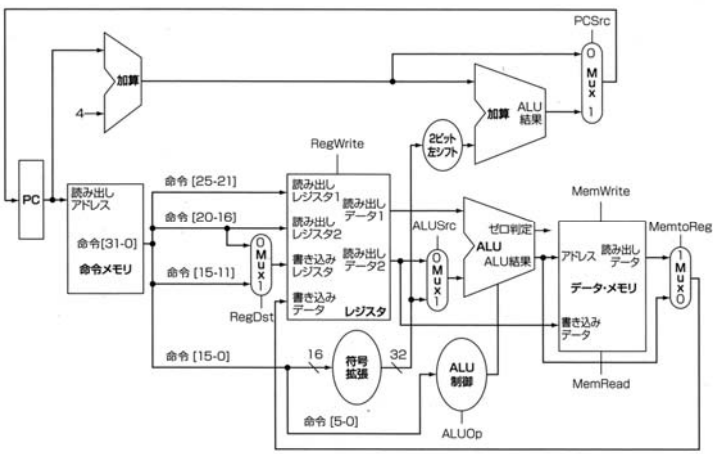
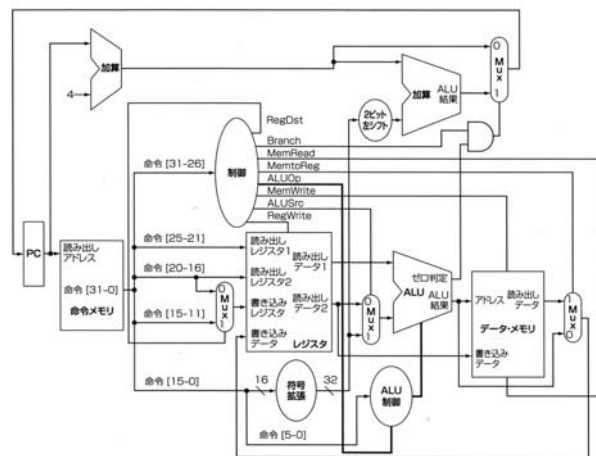


Fig. 4.17 制御ユニットを付加した簡単なデータパス

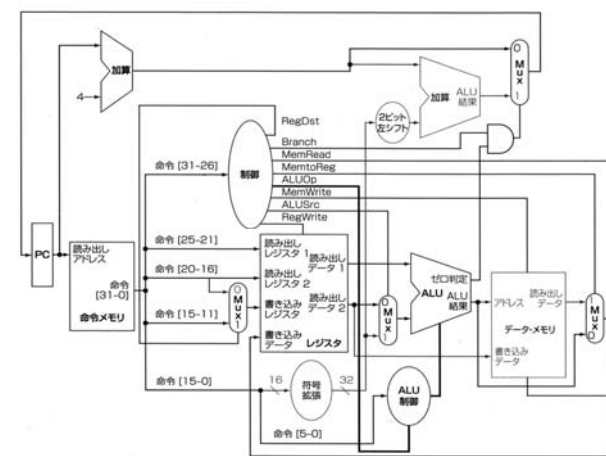


2014/12/02

©2014, Masaharu Imai

45

Fig. 4.19 add \$t1, \$t2, \$t3のようなR形式命令のデータパス

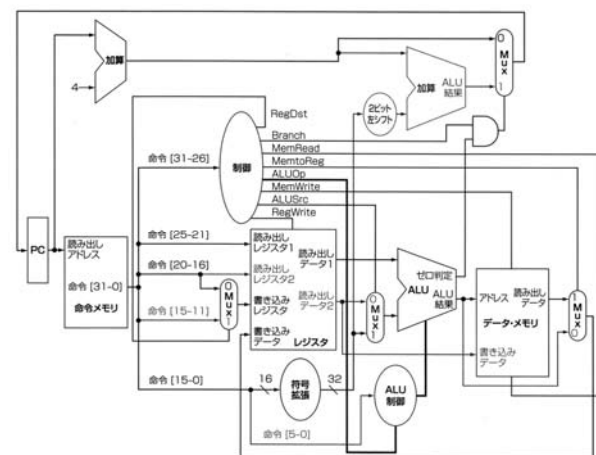


2014/12/02

©2014, Masaharu Imai

46

Fig. 4.20 ロード命令のデータパス制御

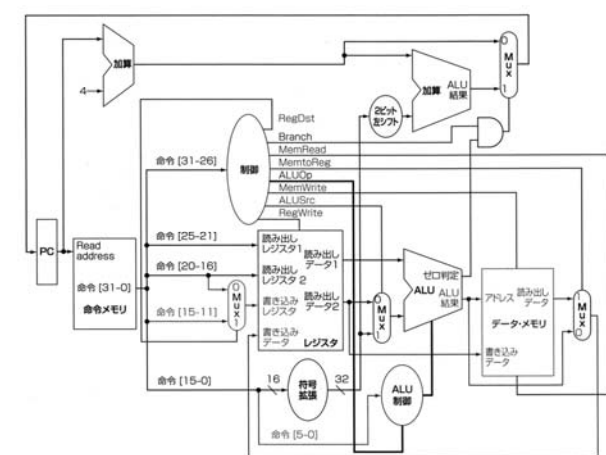


2014/12/02

©2014, Masaharu Imai

47

Fig. 4.21 branch-on-equal命令用のデータパス



2014/12/02

©2014, Masaharu Imai

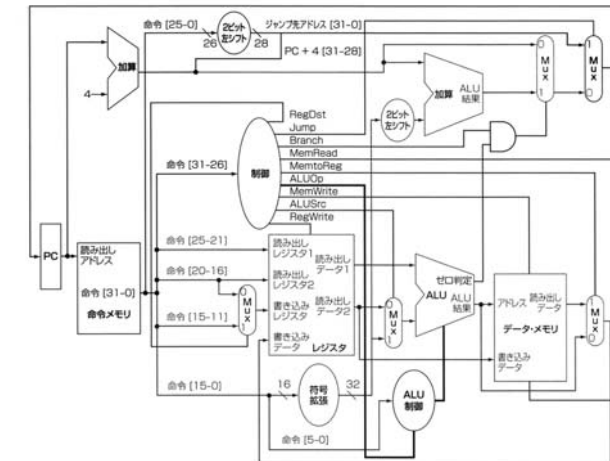
48

ジャンプ(j) 命令の実現

フィールド	000010	address
ビット位置	31-26	25-0

1. 現在のPC+4の上位4bit(現在の命令の後続命令のアドレスのbit 31-28)
 2. ジャンプ命令の即値フィールド(25-0)26bit
 3. 下位2bit00
- で構成される32bitアドレスにジャンプする

Fig. 4.24 ジャンプ命令を扱えるように拡張した単純な制御とデータパス



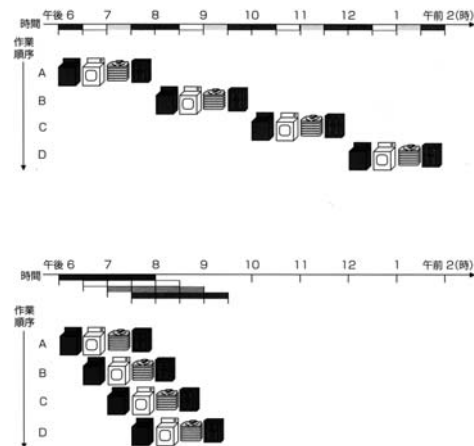
単一クロック・サイクルの制御方式

- データパスは、もちろん正しく動作する
- 効率が悪いので、実際には採用されない
 - 各命令のクロック・サイクルの長さは等しくなければならない
 - クロック・サイクルは最長のパスによって決まる
 - 5つの機能ユニットを順に使用するロード命令によって占められてしまう
 - クロック・サイクル数が長すぎるため、全体的な性能は低くなる

講義内容(1)

- 実装方式の概要
- 論理設計とクロック方式
- データパスの構築
- 単純な実現方法
- パイプライン処理の概要

Fig. 4.25 洗濯にたとえたパイプライン処理



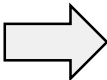
パイプライン処理の概要

パイプライン処理

- 複数の命令を少しずつずらして、同時並行的に実行する実現方式
- 今日のプロセッサは、ほぼパイプライン処理を採用

MIPSプロセッサの基本5ステップ

1. メモリから命令をフェッチする (IF)
2. 命令をデコードしながら、レジスタを読み出す (DE)
3. 命令操作の実行またはアドレスの生成を行う (EX)
4. データ・メモリ中のオペランドにアクセスする (MEM)
5. 結果をレジスタに書き込む (WB)

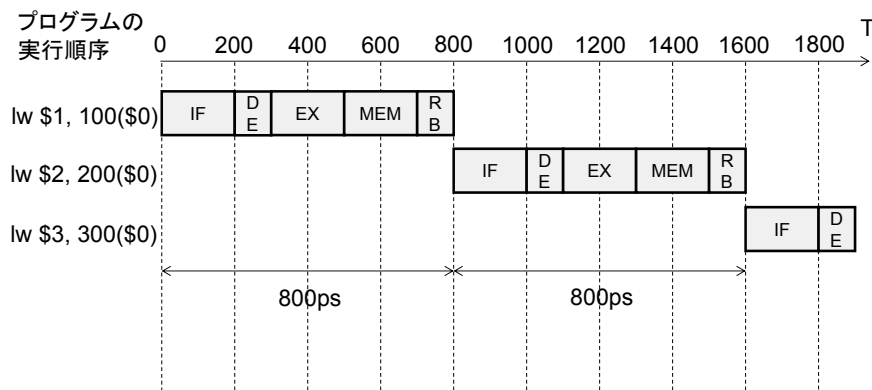


5ステップを5ステージでパイプライン実行

実行時間の仮定

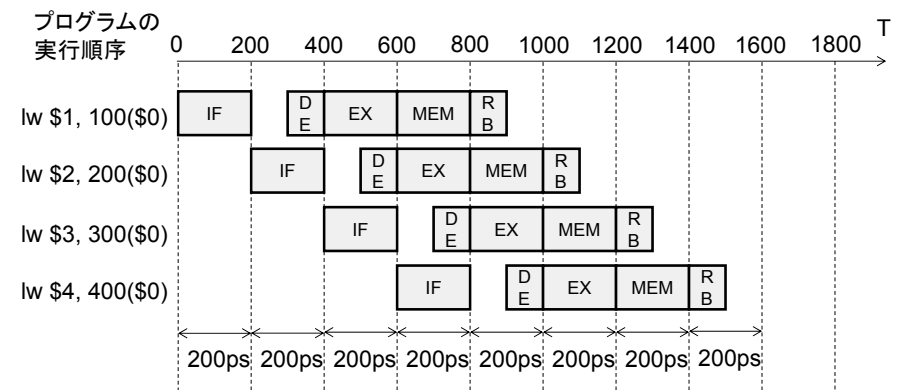
命令タイプ	命令フェッチ (IF)	レジスタの読み出し (DE)	ALU操作 (EX)	データ・アクセス (MEM)	レジスタの書き込み (RB)	合計時間
語のロード (lw)	200ps	100ps	200ps	200ps	100ps	800ps
語のストア (sw)	200ps	100ps	200ps	200ps		700ps
R形式 (add, sub, and, or, slt)	200ps	100ps	200ps		100ps	600ps
分岐 (beq)	200ps	100ps	200ps			500ps

単一クロック・サイクルでの実行



パイプラインでの実行

個々の命令の実行時間を
短縮するのではなく、命令
のスループットを増加させる



パイプライン処理による速度向上比

□ 速度向上比 (Speedup Rate)

$$= \frac{\text{非パイプライン処理での命令の実行間隔}}{\text{パイプライン処理での命令の実行間隔}}$$

□ 非パイプライン処理での命令の実行間隔

- 800 ps

□ パイプライン処理での命令の実行間隔 (命令数 n)

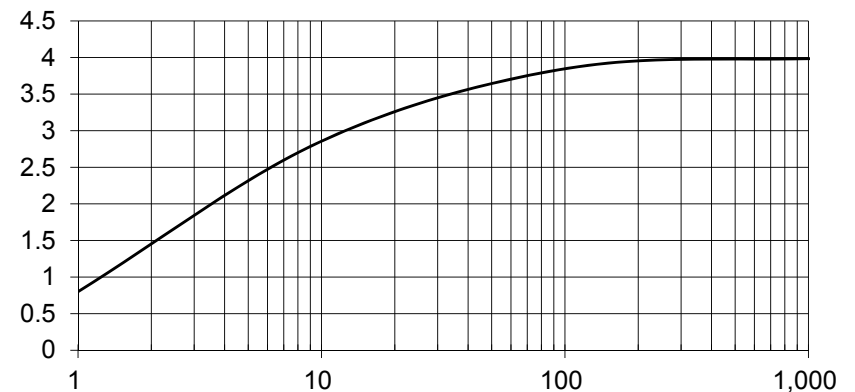
- $(200 \text{ ps} \times n + 800 \text{ ps}) / n > 200 \text{ ps}$

□ 速度向上比

- $(800 \times n) / (200 \times n + 800) < 4$

パイプライン処理による速度向上比

速度向上比



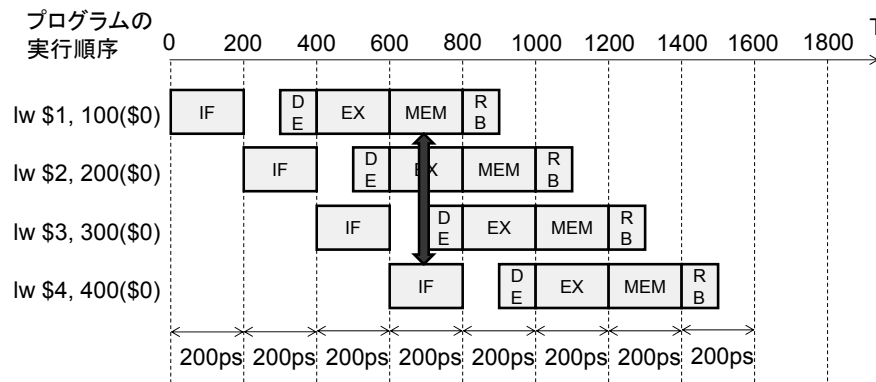
パイプライン処理の注意すべき点

- パイプライン・ハザード (pipeline hazard)
 - パイプラインの処理が, 何らかの理由で, 所定のクロック・サイクルで実行できない事態が起こること
- パイプラインハザードの分類
 - 構造ハザード (structure hazard)
 - データ・ハザード (data hazard)
 - 制御ハザード (control hazard)

構造ハザード

- 同時に実行される命令の組合せにハードウェアが対応できないために, 命令を所定のクロックサイクルで実行出来ない状況が生じること
- 例:
 - メモリが一つしかない場合 (命令とデータが同一のメモリ上に格納されている場合)
 - 4ステージ目のメモリアクセスと3命令先の命令フェッチが競合
 - 同じハードウェアモジュールを複数のステージで排他的に使用する場合
 - レジスタファイルに対するデータの書込みと読出しが同時に行えない場合にはハザードが生じる

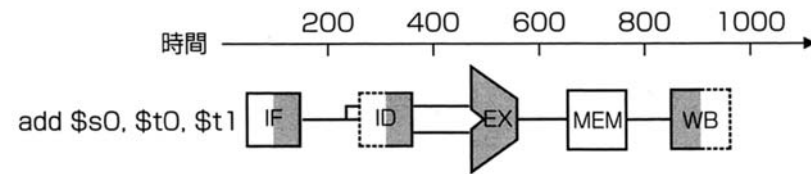
構造ハザードの例



データハザード

- 他のステップが完了するのを別のステップが待つ必要があるために, パイプラインをストールさせなければならない事態が起こること
 - ある命令がパイプライン中にある先行命令に依存する場合
add \$s0, \$t0, \$t1
sub \$t2, \$s0, \$t3
- 解決方法
 - フォワーディング (forwarding) / バイパスング (bypassing)
 - 命令コードの並べ替え

Fig. 4.28 命令パイプラインの模式的な表現



データハザードの例(R形式命令)

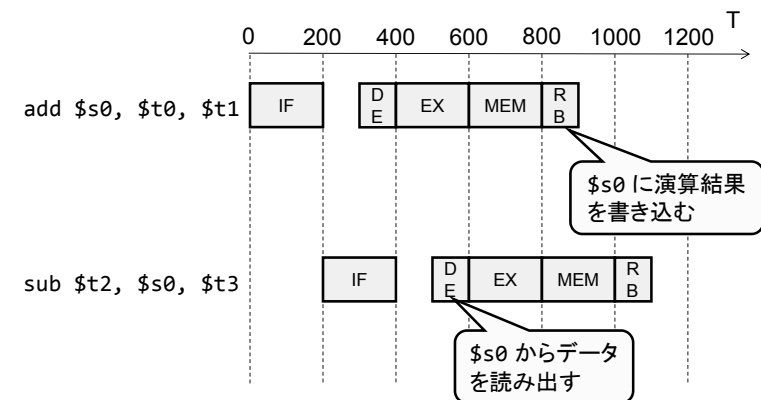
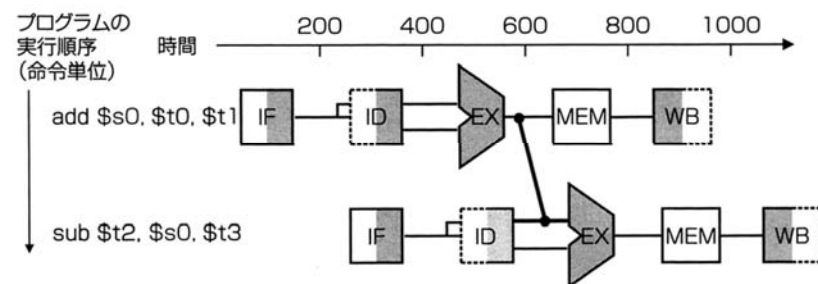
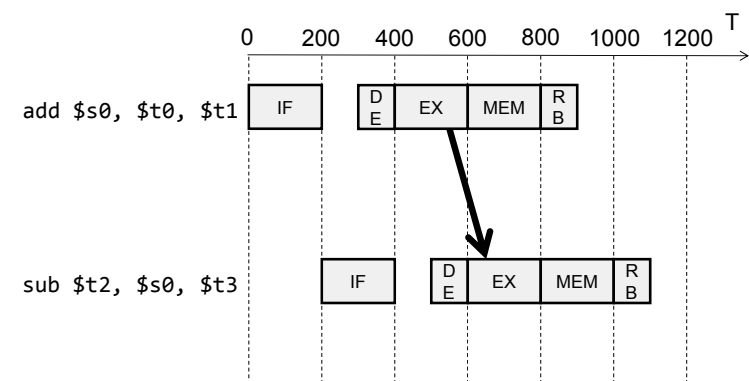


Fig. 4.29 フォワーディングの模式的表現



データハザードの回避(R形式命令)



データハザードの例(ロード命令)

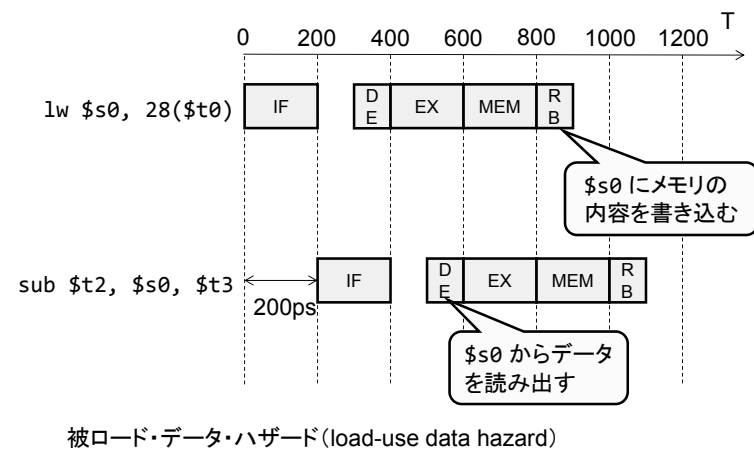
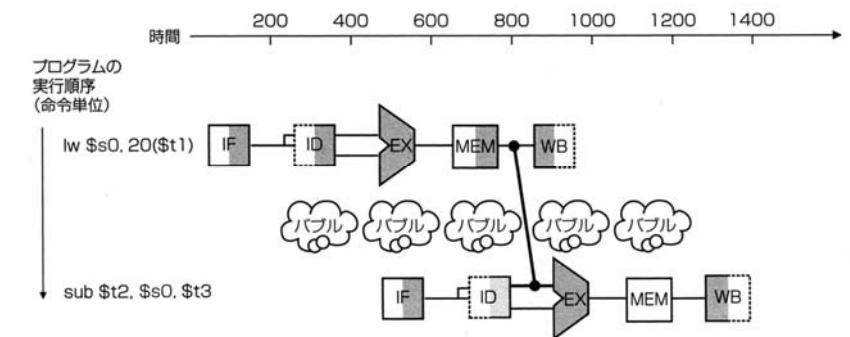
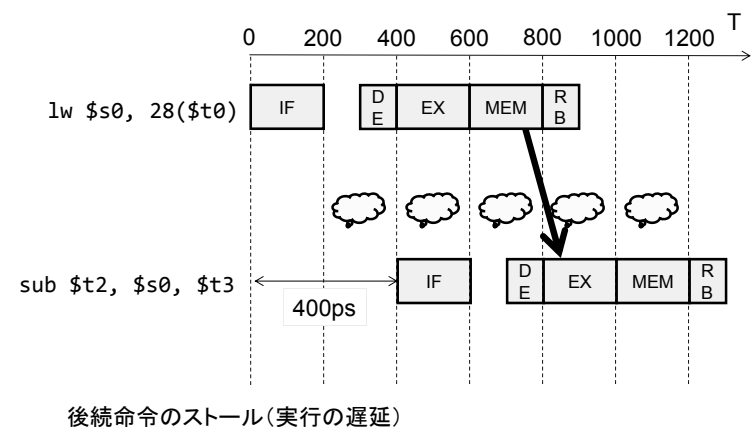


Fig. 4.30 ロード命令に後続するR形式命令がロードされたデータを使用する場合



データハザードの回避(ロード命令)



パイプラインストールを回避するコード並び替え

- ソフトウェアでコードの順序を入れ替えて、パイプラインストールを回避する
- 例(C言語):


```
a = b + e;
c = b + f;
```

例:

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2  - - ハザード発生
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4  - - ハザード発生
sw  $t5, 16($t0)
```

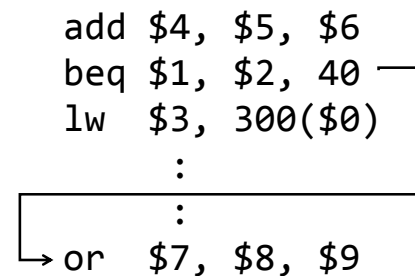
改良されたコード:

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2  - - ハザード回避できた
sw  $t3, 12($t0)
add $t5, $t1, $t4  - - ハザード回避できた
sw  $t5, 16($t0)
```

制御ハザードの例

□ プログラム

```
add $4, $5, $6
beq $1, $2, 40
lw  $3, 300($0)
    :
    :
    :
or  $7, $8, $9
```



制御ハザードの解決方法(1)

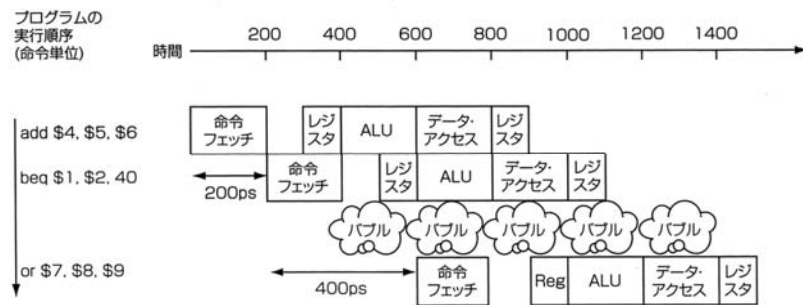
□ 制御ハザード回避の必要性

- 長いパイプラインでは、第2ステージで分岐先を決定出来ない場合があるので、分岐命令をストールすると速度がさらに低下する

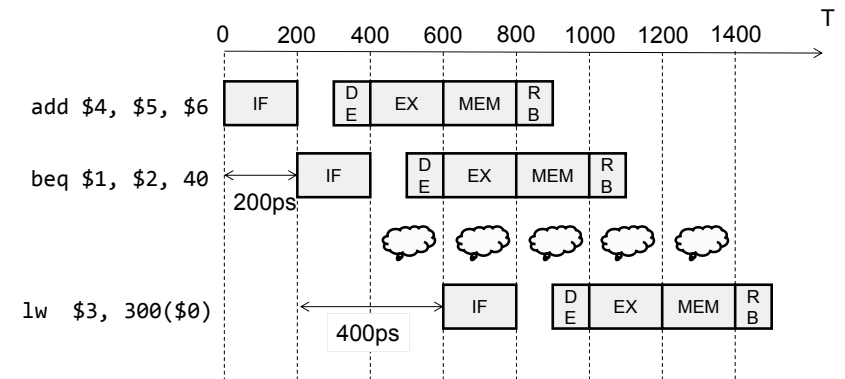
□ 分岐命令をフェッチしたら、直ちに後続命令をストールし、分岐の判定結果が得られてから次の命令のアドレスが決定されるまで待つ。

- 問題点: 常に後続命令がストールされる

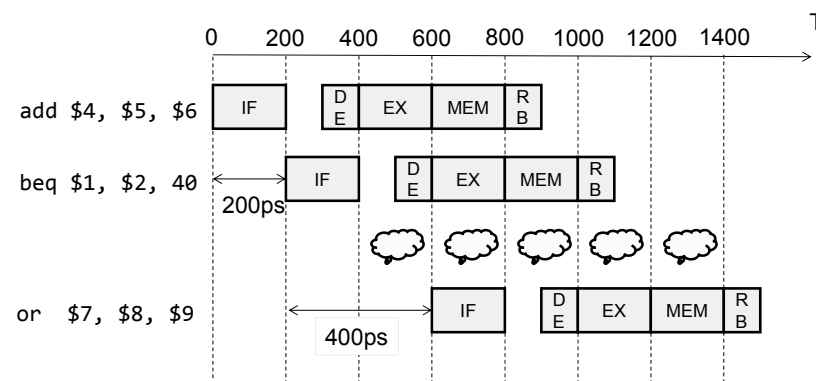
Fig. 4.31 制御ハザードの解決策として、あらゆる条件分岐をストールさせるパイプライン



解決策1でのパイプラインの動作 (分岐不成立の場合)



解決策1でのパイプラインの動作 (分岐成立の場合)

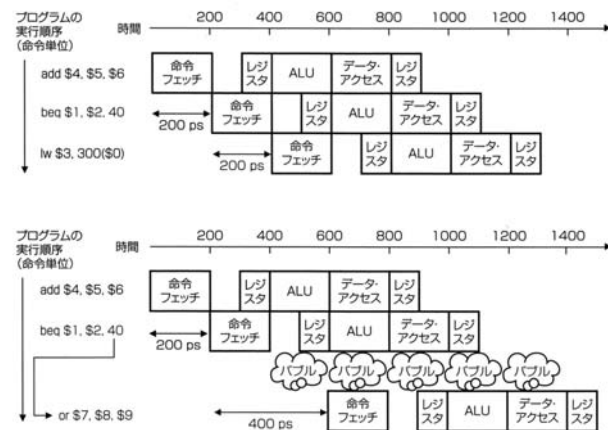


制御ハザードの解決方法(2)

□ 単純な分岐予測を行う

- 分岐は常に不成立と予測する
- 問題点: 分岐が成立すると、後続命令は常にストールされる

Fig. 4.32 制御ハザードの解決策として分岐は成立しないと予測する方法



2014/12/02

©2014, Masaharu Imai

81

制御ハザードの解決方法(3)

□ 分岐命令の一部を分岐が成立するもの、それ以外を分岐が成立しないものと予測する。

■ 静的予測の例

- 低位のアドレスに戻る分岐は成立すると予測
- ループの末尾にはループの先頭に戻る分岐命令がある
- この分岐は成立する確率が高い

■ 動的予測

- 最後の予測の成功率を考慮して次の予測を調整
- 例：各分岐が成立したかどうかの履歴を取っておき、近い過去に基づいて未来を予測する

2014/12/02

©2014, Masaharu Imai

82

制御ハザードの解決方法(4)

■ 予測が正しくなかった場合の処置

- 予測が外れた分岐以降の命令列を無効にして、適切な分岐アドレスから処理を再開する

2014/12/02

©2014, Masaharu Imai

83

パイプラインについての補足

- 命令パイプラインは、同時に実行される命令数を増やして、速度を向上させる技術
- スループット (throughput)
 - 一定時間内に実行された命令の総数
- レイテンシ (latency)
 - 個々の命令の実行の開始から完了までに必要な時間
- パイプライン化によってスループットは改善されるが、レイテンシは改善されない

2014/12/02

©2014, Masaharu Imai

84