コンピュータの構成と設計:

4. プロセッサ

4.1 はじめに

4.2 論理設計とクロック方式

4.3 データパスの構築

4.4 単純な実行方式

4.5 パイプライン処理の概要

4.6 データパスのパイプライン化と制御

4.7 データ・ハザード:フォワーディングとストール

4.8 制御ハザード

4.9 例外

4.7 データ・ハザード:フォワーディングとストール

p.295

ここまでで，パイプラインの効力とハードウェアの仕組みは分かった

▶ 問題点に目を向ける

用語の復習:

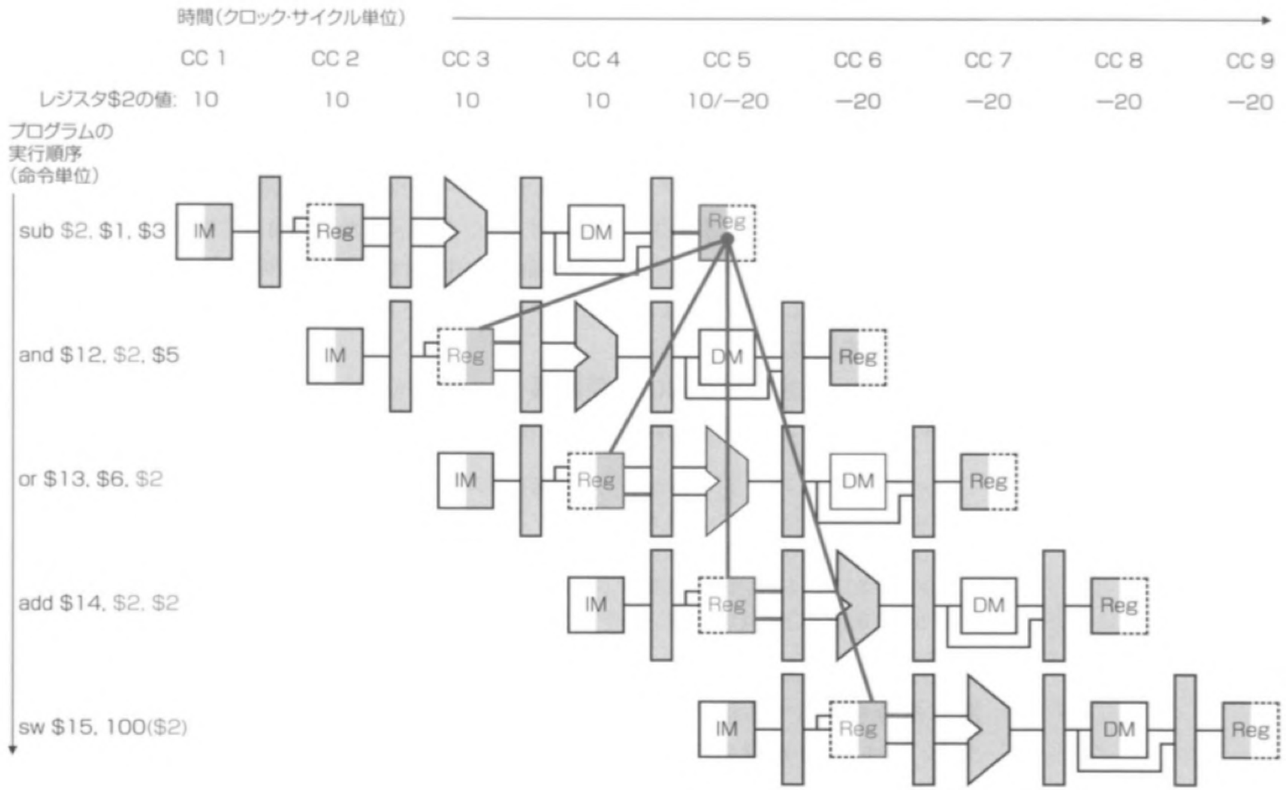
- **ハザード(hazard)**: 次のクロックサイクルで，次の命令を実行できないという事態
  - 構造ハザード: ハードウェア的な資源の競合に起因
  - データ・ハザード: 処理するデータの依存関係に起因
  - 制御ハザード(分岐ハザード): 制御の依存に起因
- **パイプライン・ストール(stall)/バブル(bubble)**: ハザードを解消するために命令の処理を一時停止すること
- **フォワーディング/バイパシング**: データがレジスタやメモリから読み出せるようになるのを待たず，内部バッファから直接引き出して使用すること

依存関係 (dependency)

後ろ4つの命令は全て，最初の命令の結果であるレジスタ**$2**と依存関係を持つ:

```
sub $2, $1, $3   # subの結果をレジスタ$2に収める
and $12, $2, $5  # 第1オペランド($2)がsubに依存
or $13, $6, $2   # 第2オペランド($2)がsubに依存
add $14, $2, $2  # 第1($2)と第2($2)オペランドがsubに依存
sw $15, 100($2) # インデックス($2)がsubに依存
```

後ろ向きの線 = データ・ハザード

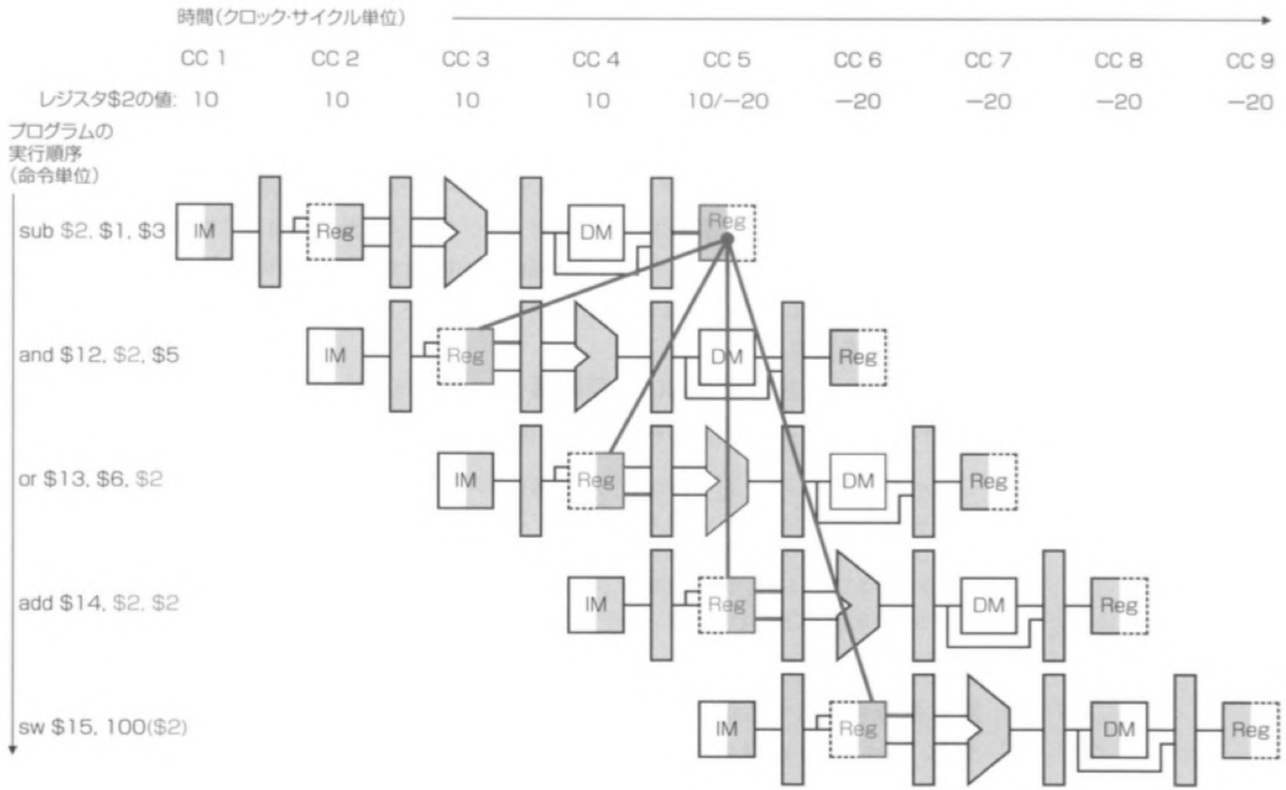


```
sub $2, $1, $3   # subの結果をレジスタ$2に収める
and $12, $2, $5  # 第1オペランド($2)がsubに依存
or $13, $6, $2   # 第2オペランド($2)がsubに依存
add $14, $2, $2  # 第1($2)と第2($2)オペランドがsubに依存
sw $15, 100($2) # インデックス($2)がsubに依存
```

しかし，sub の結果は**CC3**の終わりに利用可能になる.

そして，and, orで結果が必要になるのは，**CC4,5** である.

→ 必要とする任意のユニットにフォワーディングすることができれば，ストールしない!



依存関係を詳しく表す表記方法:

"IF/ID.RegisterRs"
- "IF/ID": パイプライン・レジスタの名前

- "RegisterRs": パイプライン・レジスタ内のフィールド名

▶ IF/IDパイプライン・レジスタの中の読み出しレジスタ1番を指す

データ・ハザードが発生する条件は以下のように表せる(不完全版):

```
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
```

例に戻り，subとandのハザードは，上の1aに相当する.

```
EX/MEM.RegisterRd = ID/EX.RegisterRs = $2
```

subがMEMステージ

andがEXステージ

問題点:

- 命令の中には書き込みを行わないものがある

→ 不必要なフォワーディングを行ってしまう

- $zeroオペランドが使用している場合[1]

→ ゼロでない値をフォワーディングしても意味がない

解決策:

RegWrite信号が設定されているかをチェックする:

```
EX/MEM.RegisterRd ≠ 0
MEM/WB.RegisterRd ≠ 0
```

この2つを追加すればよい

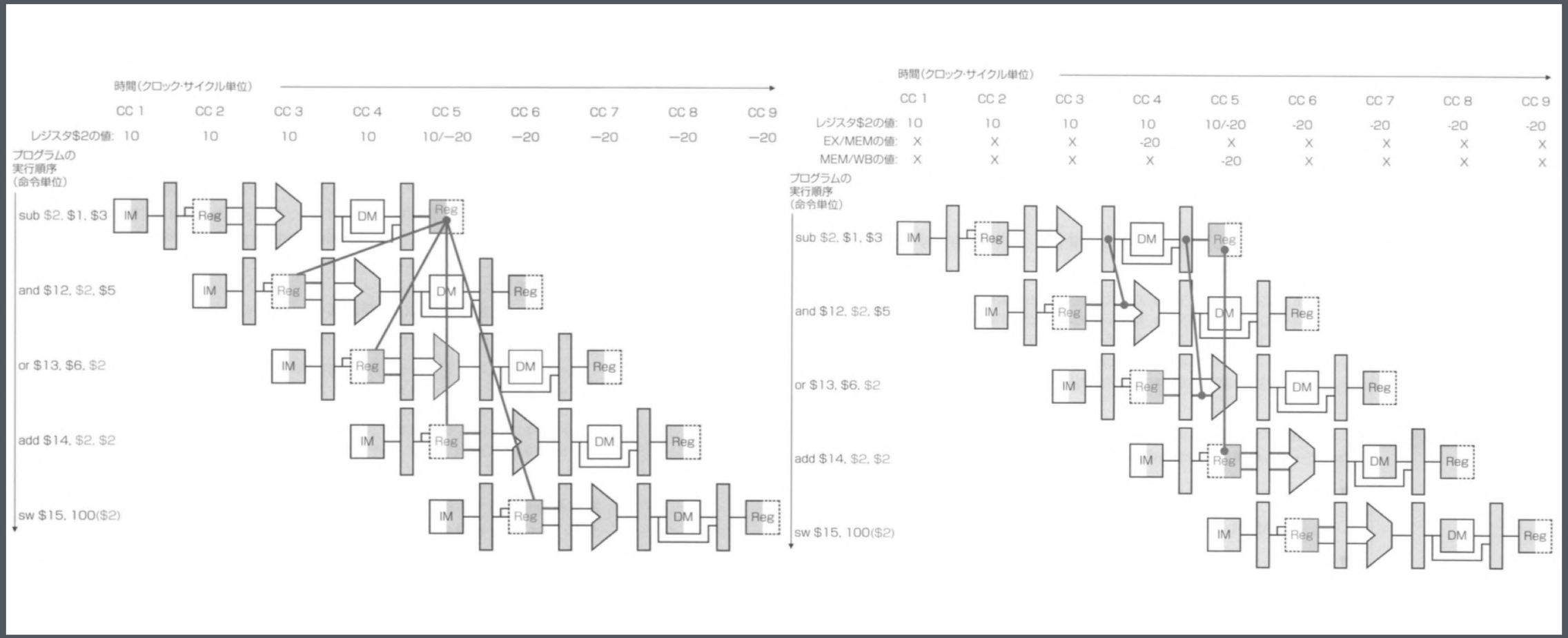
---

[1] 例: `sll $zero, $1, 2`

データ・ハザードを検出できるようになった

→ どうやって適切なデータをフォワーディングするのか?

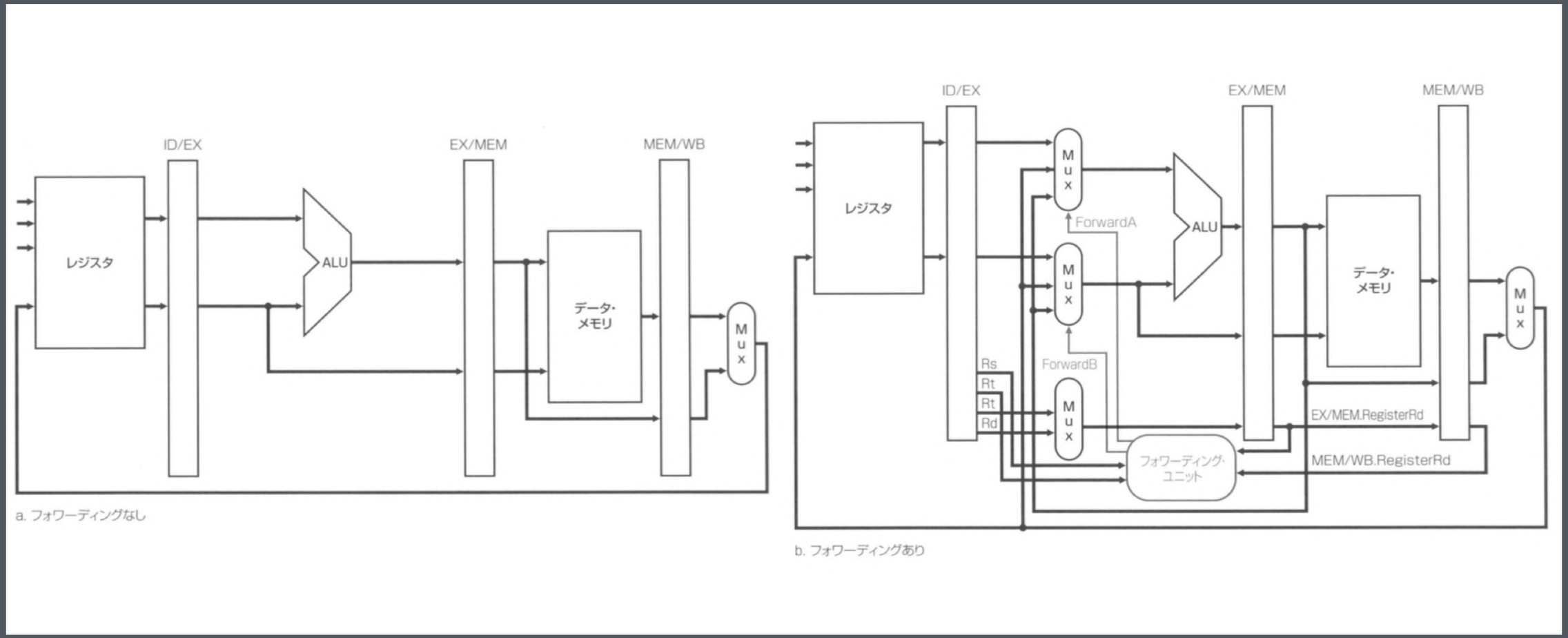
パイプラインレジスタとALUの依存関係(右)



- ALU入力を任意のパイプライン・レジスタから取り出せれば，適切なデータをフォワーディングできる
- ALU入力にマルチプレクサを追加し，ハザード検出ユニットと同様の制御を行えばデータ・ハザードがあっても全速力で稼働できる

ここでは，R形式の4命令: add, sub, and, or
だけがフォワーディングを行う必要があると仮定

フォワーディング機構


ID/EX
EX/MEM
MEM/WB
レジスタ
ALU
データ・
メモリ
M
u
x
a. フォワーディングなし
ID/EX
EX/MEM
MEM/WB
レジスタ
M
u
x
ForwardA
ALU
ForwardB
Rs
Rt
Rt
Rd
データ・
メモリ
EX/MEM.RegisterRd
MEM/WB.RegisterRd
フォワーディング・
ユニット
b. フォワーディングあり

フォワーディング・マルチプレクサ用の制御値

マルチプレクサ制御	ソース	摘要
ForwardA = 00	ID/EX	ALU の第 1 オペランドがレジスタ・ファイルから得られる
ForwardA = 10	EX/MEM	ALU の第 1 オペランドが 1 つ前の ALU の結果から先送りされて来る
ForwardA = 01	MEM/WB	ALU の第 1 オペランドがデータ・メモリまたは 2 つ前の ALU の結果から先送りされて来る
ForwardB = 00	ID/EX	ALU の第 2 オペランドがレジスタ・ファイルから得られる
ForwardB = 10	EX/MEM	ALU の第 2 オペランドが 1 つ前の ALU の結果から先送りされて来る
ForwardB = 01	MEM/WB	ALU の第 2 オペランドがデータ・メモリつまり 2 つ前の ALU の結果から先送りされて来る

- ALUにフォワーディングを行うマルチプレクサがEXステージに配置されているため，フォワーディングの制御回路はEXステージに置かれる.
- フォワーディングを行うかどうかを判断するには，IDステージのオペランド・レジスタ番号をID/EXパイプライン・レジスタ経由で渡す必要がある.

ハザードを検出する条件:

1. **EXハザード**:

   条件

   EX/MEM.RegWrite
   and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs)
   が真の場合，FowardA = 10

   条件

   EX/MEM.RegWrite
   and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
   が真の場合，FowardB = 10

マルチプレクサ制御	ソース	摘要
ForwardA = 00	ID/EX	ALU の第 1 オペランドがレジスタ・ファイルから得られる
ForwardA = 10	EX/MEM	ALU の第 1 オペランドが 1 つ前の ALU の結果から先送りされて来る
ForwardA = 01	MEM/WB	ALU の第 1 オペランドがデータ・メモリまたは 2 つ前の ALU の結果から先送りされて来る
ForwardB = 00	ID/EX	ALU の第 2 オペランドがレジスタ・ファイルから得られる
ForwardB = 10	EX/MEM	ALU の第 2 オペランドが 1 つ前の ALU の結果から先送りされて来る
ForwardB = 01	MEM/WB	ALU の第 2 オペランドがデータ・メモリつまり 2 つ前の ALU の結果から先送りされて来る

1. **MEMハザード**:

条件

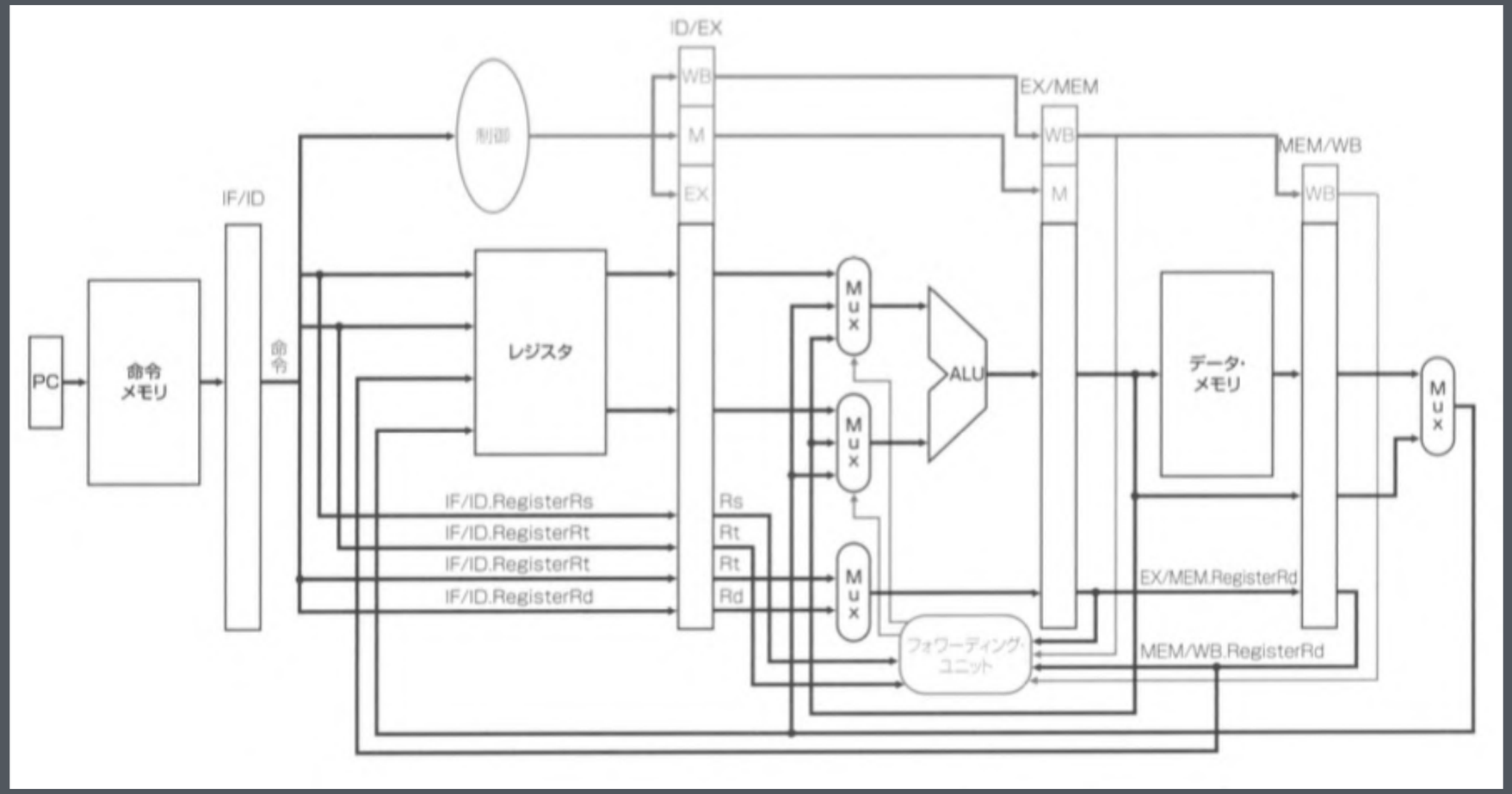
MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)
が真の場合，FowardA = 01

条件

MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)
が真の場合，FowardB = 01

マルチプレクサ制御	ソース	摘要
ForwardA = 00	ID/EX	ALU の第 1 オペランドがレジスタ・ファイルから得られる
ForwardA = 10	EX/MEM	ALU の第 1 オペランドが 1 つ前の ALU の結果から先送りされて来る
ForwardA = 01	MEM/WB	ALU の第 1 オペランドがデータ・メモリまたは 2 つ前の ALU の結果から先送りされて来る
ForwardB = 00	ID/EX	ALU の第 2 オペランドがレジスタ・ファイルから得られる
ForwardB = 10	EX/MEM	ALU の第 2 オペランドが 1 つ前の ALU の結果から先送りされて来る
ForwardB = 01	MEM/WB	ALU の第 2 オペランドがデータ・メモリつまり 2 つ前の ALU の結果から先送りされて来る

EXステージでの結果を利用する処理に関するフォワーディングの実装:

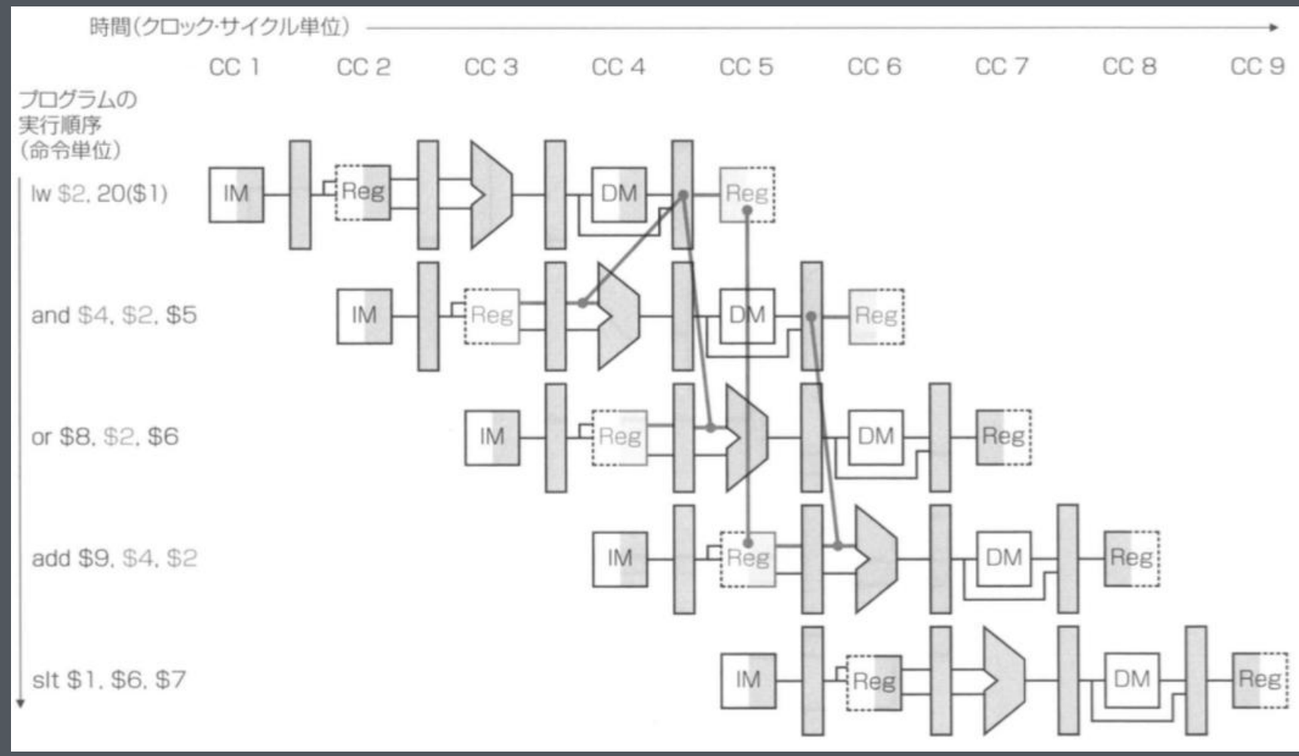


データ・ハザードとストール

フォワーディングで救済できない場合:
ロード命令が書き込むのと同じレジスタを，直後の命令が読み出そうとする

```
lw $2, 20($1)
and $4, $2, $5
...
```

→ ハザード検出ユニットが必要になる



ハザード検出ユニット

- IDステージで動作
- ロード命令とその結果を読み出す命令との間にストールを挿入する

ハザード検出ユニットの条件:

条件

```
ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt))
```

が真の場合，パイプラインをストールさせる

最初の行は命令がロードであるか否かをチェック

次の2行は，EXステージにあるロード命令のデスティネーション・レジスタ・フィールドがIDステージにある命令のどちらかのソース・レジスタと合致するかどうかをチェック

4.8 制御ハザード

p.308

分岐に起因するハザード:

制御ハザード(control hazard) または
分岐ハザード (branch hazard) という

- 比較的単純
- データ・ハザードより発生頻度が低い
- フォワーディングのような効果的な手段がない

2つの対策と，1つの最適化手法を検討する

対策1: 分岐が不成立と仮定する

分岐が完了するまでストールさせると，速度が遅くなりすぎる

→ 分岐が不成立と**予測**して，後の命令を実行する

分岐が成立した場合:

- フェッチ，デコードを進めた命令を破棄[2]
- 分岐先の命令から処理を実行

---

[2] フラッシュ(flush): パイプライン上のIF, ID, EXの各ステージ上の命令を一括消去できなければいけない

対策2: 分岐による遅延の削減

成立した分岐のコストを下げることで，性能を改善できる

ここまでの想定: 分岐用の次のPCは，MEMステージで取り上げる
→ 分岐をパイプラインの早いステージに移せば，フラッシュすべき命令が少なくて済む

観察の結果: 分岐の多くはALUがいらない単純な条件判定に基づいている
→ 複雑な条件判定が必要な際だけALUを用いて比較を行う命令を使用する.

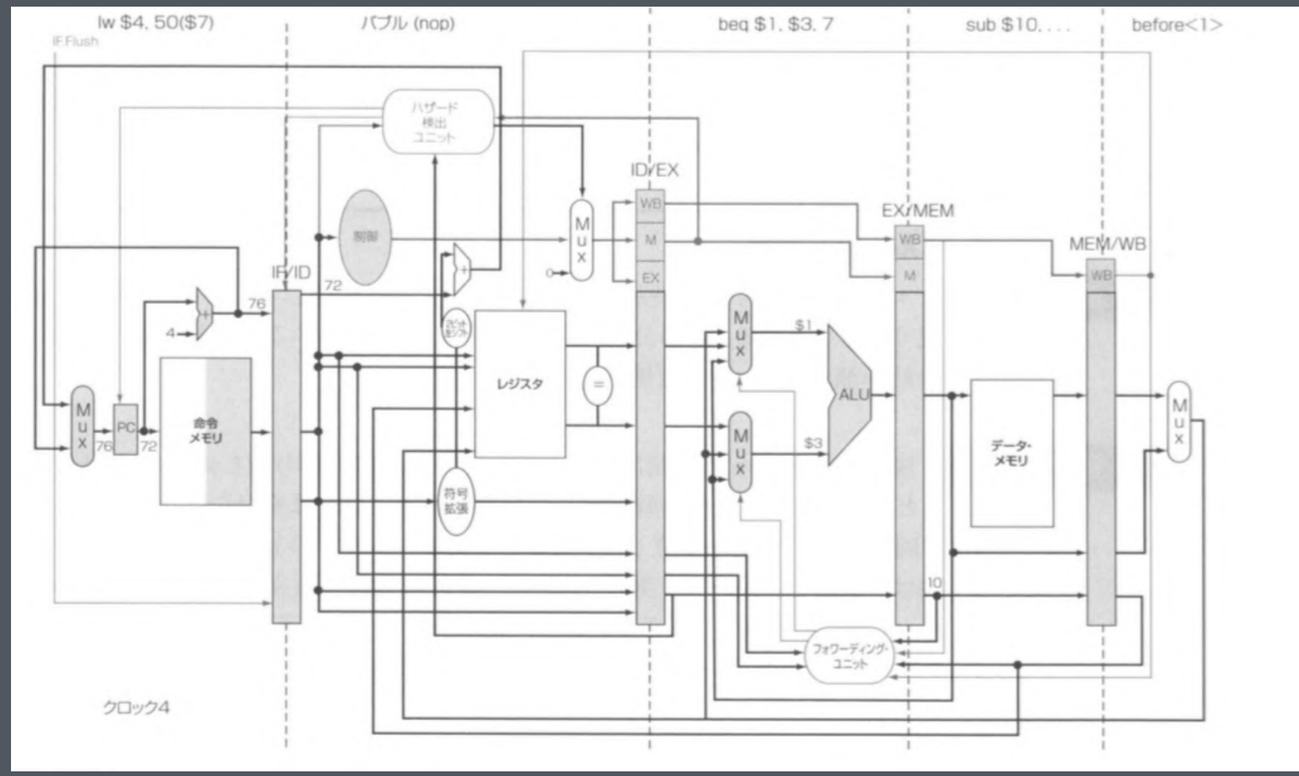
分岐を前倒しにするためには:

- 分岐先アドレスの計算 (簡単)
  - 分岐用の加算器をMEMステージからIDステージに移す
- 分岐判定の評価 (困難)
  - IDステージの間に読み出した2つのレジスタを比較して，両者が等しいかどうかチェックする
  - ← そのために，対応するビットのXORをとり，結果の全ビットのANDをとる
  - フォワーディングやハザード検出の回路を追加する

    ...

▶ それでも，分岐の実行をIDステージに移せば性能が改善される.

- IFステージ上の命令をフラッシュするために，IF.Flushという制御線を追加する.
- この制御線は，IF/IDパイプラン・レジスタの命令フィールドを0クリアする
- レジスタをクリアすると，フェッチした命令は何も行わない**nop**に変換される



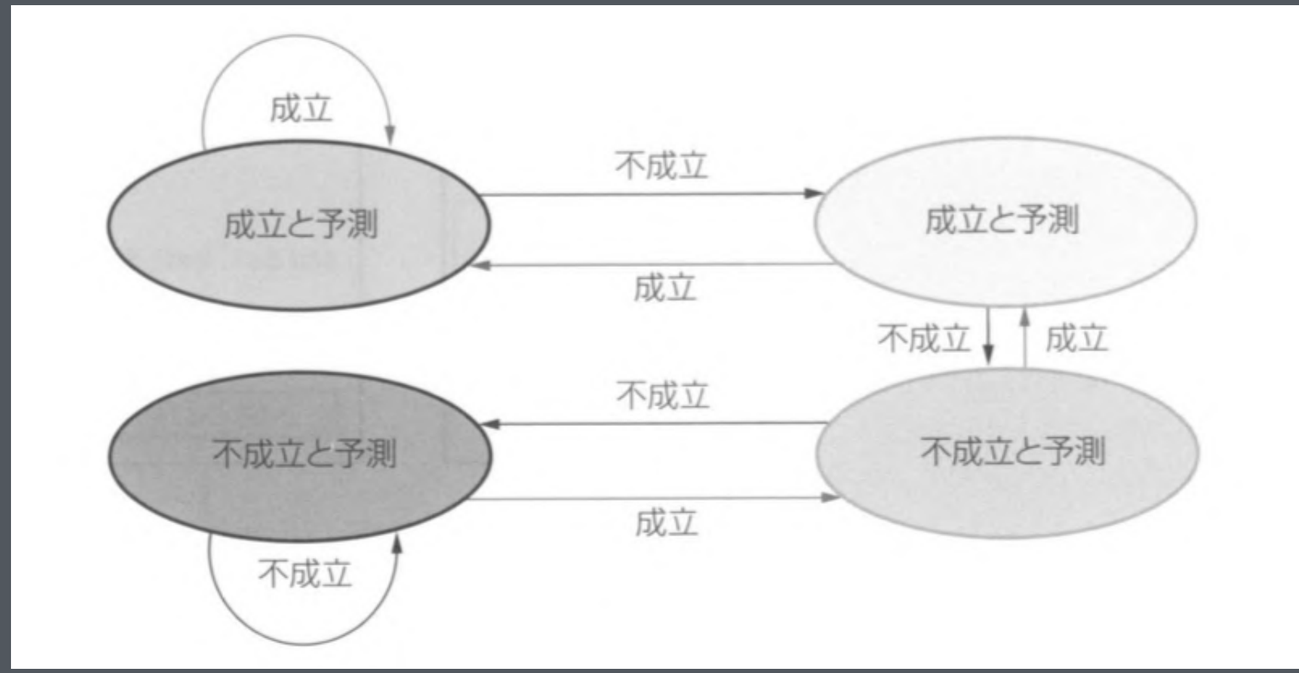
最適化手法: 動的分岐予測

- 分岐が成立しないと仮定する方法は，分岐予測 (branch prediction)の一形態.
- パイプの段階数が小さい間は適切だが，分岐の増えるとペナルティが大きくなる.
  単純な静的分岐予測では性能の無駄使いが大きい.
  → ハードウェアを付け加えることで，動的に分岐予測を行う (dynamic branch prediction)

実装形態

分岐予測バッファ(分岐履歴テーブル):
命令アドレスの階部分でインデックス付けされ，その分岐命令が最近成立したか否かを示す情報が何ビットか記録される.
→ 極めて単純なバッファで，予測の根拠が正しい保証はない.

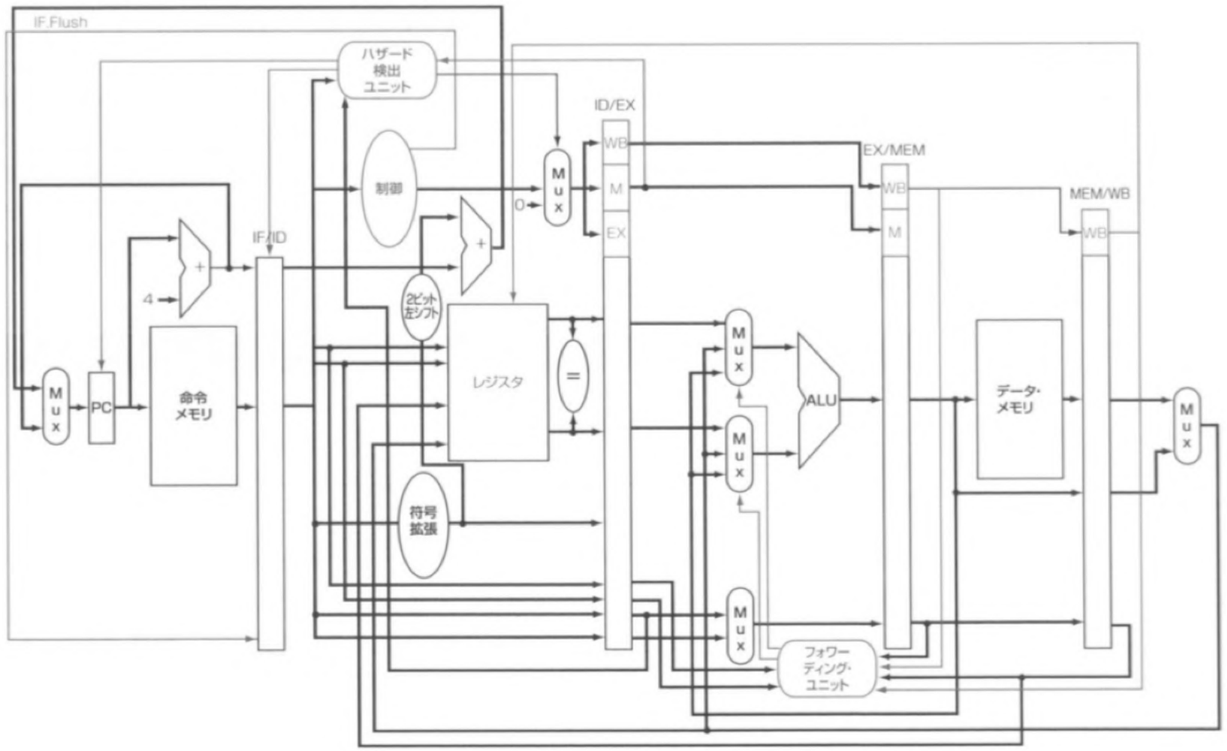
性能上の欠点:
1bitでは，ループの分岐で最初と最後に**2回予測を間違えてしまう.**
(最初に間違うのは前回の最後で反転させられているため)
→ 2bitにする方法がよく用いられる = 2回間違わないと反転されない.



パプラインのまとめ

- 単一サイクルのデータパス
- パイプライン・レジスタ
- フォワーディング・パス
- データ・ハザードの検出
- 分岐予測
- 命令のフラッシュ

完成したデータパス▶



4.9 例外

p.316

例外と割込み

プロセッサの設計において，**制御** はもっとも挑戦的な課題

その制御の中でももっとも困難な部分:

- 例外 (exception)
- 割込み (interrupt)

例外/割込みの例 ▶

事象のタイプ	発生源	MIPS の用語
入出力装置からのリクエスト	外部	割込み
ユーザー・プログラムからの OS 起動	内部	例外
算術オーバフロー	内部	例外
未定義命令の使用	内部	例外
ハードウエアの誤動作	内部または外部	例外または割込み

MIPSアーキテクチャにおける例外への対処法

2つのタイプの例外が発生する可能性がある:

1. 未定義命令の実行

2. 算術オーバーフロー

```
add $1, $2, $1
```

における算術オーバーフローを例としていく.

基本的な措置:

1. **例外プログラムカウンタ**(EPC)に問題を起こしている命令のアドレスを退避する.
2. OSの特定アドレスに制御を渡す

→ OSが適切な対処策を実施する．(プログラムを中止してエラーを中止するなど)

措置を行った後，OSは以下の処理をする:

- プログラムを強制終了する場合
- プログラムを実行を継続する場合
    - EPCを使用してプログラムをどこから再開するか決定する.

OSへ例外が発生した理由を伝える方法:

1. 状態レジスタ(Causeレジスタ): MIPSで採用している.

2. **ベクタ割込み(vectored interrupt)**:

   - 例外の原因に基づいて，制御を移す先のアドレスを指定している.

   - OSは制御アドレスがどのアドレスに移されたかによって例外の原因を知ることができる.

例外のタイプ	例外時の飛び先アドレス（16 進）
未定義命令	$8000\ 0000_{16}$
算術オーバフロー	$8000\ 0180_{16}$

- MIPSでの例外対応を実装するには，以下2つのデータパスを追加する:
  - EPC: 問題を起こした命令のアドレスを記録しておくために使用する32ビットレジスタ[3]
  - Cause: 例外の原因を記録するために使用するレジスタ．MIPSアーキテクチャでは32ビット．

[3] 例外の対処法がベクタ方式の場合でも必要

パイプライン方式における例外

- 例外を制御ハザードのバリエーションとして扱う
- 算術オーバーフローが発生したときは，分岐のときと同じようにaddに続く命令をフラッシュして，新しいアドレスから命令をフェッチする.
  - ID.Flushという新しい制御信号とハザード検出ユニットからのストール信号との論理和を取ればIDステージでのフラッシュができる.
  - EX.Flushという新しい信号を使用し，制御線を0にする.
- MIPSの例外ルーチンが置かれているアドレス[4]からの命令フェッチを開始するには，PCにアドレスを送る第3の入力をPCマルチプレクサに追加するだけ.

---

[4] $8000\,0180_{16}$

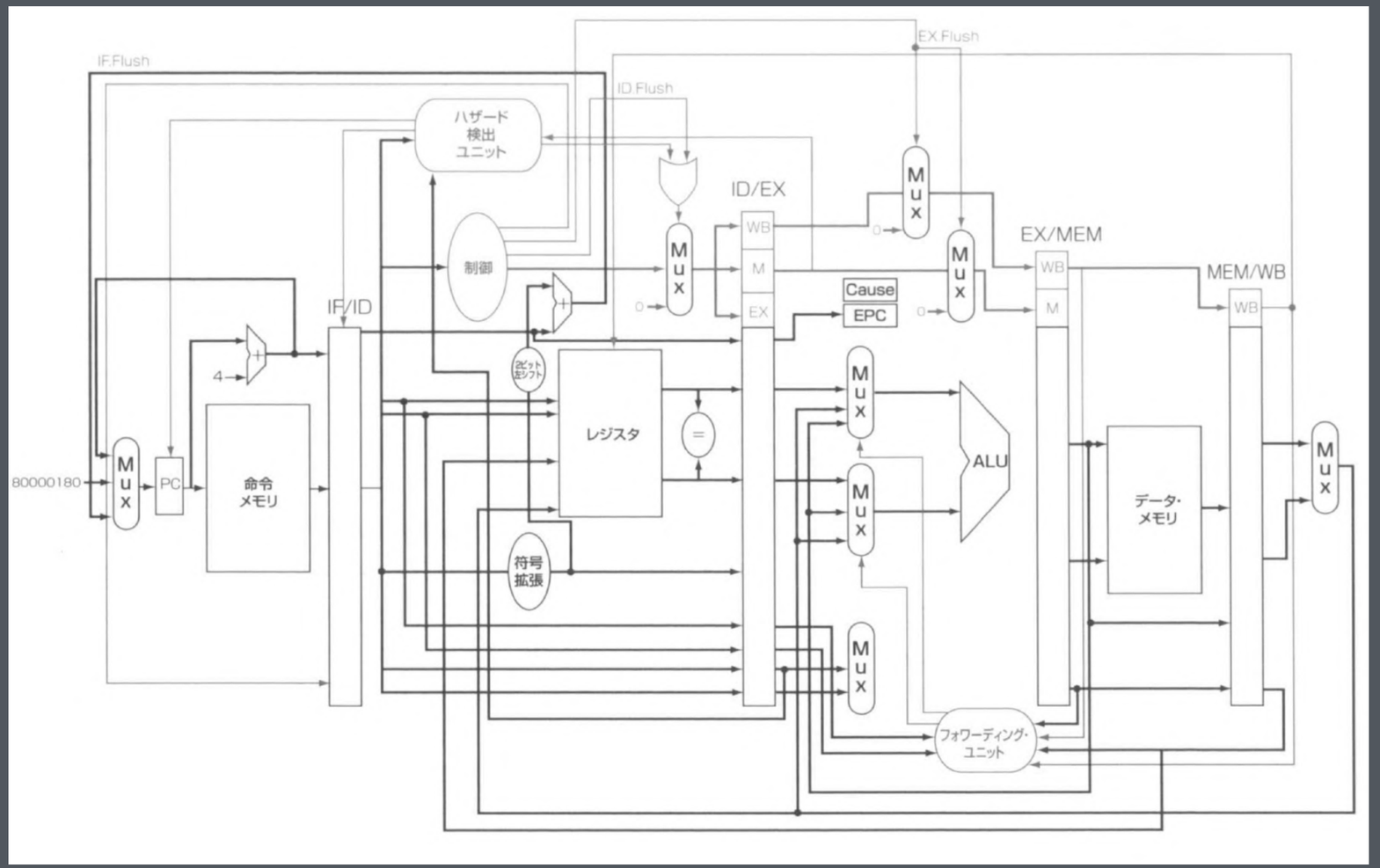
最後のステップ:

問題を起こした，命令アドレスをEPCに退避する.

現実には，アドレスに4を加えた値が保存される.

→ 例外処理ルーチンでは最初に保存されている値から4を引かなければいけない.

ここまで例外処理を制御するデータパス



- 各クロックで5つの命令が進行中なので，例外がどの命令で発生したのかを把握することは難しい
- MIPSでは例外をソートして先に実行された命令順に割り込みをかけるようにしている
- 割り込みを受けた命令のアドレスがEPCに記録され，
- あるクロックサイクルで発生しうるすべての例外がCauseレジスタには記録される

end