

# I470F 統合アーキテクチャ

組込みCPUの概要

(MIPS、ARMアーキテクチャ)

# 組込みシステムへの要求

- 低コスト(コストパフォーマンスも重要)
- ハードウェア制約
  - 低消費電力(バッテリ駆動・発熱の問題など)
    - 高性能CPUは10W～200W前後 (Core i9 は300W近い)
    - 組込みでは1W以下が望まれる
  - 低メモリ量
    - ~数メガバイト
- リアルタイム性
  - リアルタイムOS(~1MB)によるサポート

# 組込みCPUの特徴・傾向

汎用システムCPU	組込みCPU
32ビット、64ビット、CISC、(RISC)	8~32ビット RISC、(一部、CISC)
数GHz (Core i9のTurbo Boostでは最大5.5GHz)	数十MHz~数百MHz(電力削減が重要)
数千円~数十万円	数十円~数千円 (コアのライセンス販売形態もある)
10W~300W (Core i9のTurbo Boostでは最大241W)	数十mW~1W
ヒートシンク+ファン(一部、水冷)	自然空冷、(一部、ヒートシンク、ファン)
1次+2次キャッシュ+3次キャッシュ	1次(+2次)キャッシュ、あるいはメモリ空間はチップ内完結

# 組込みシステムでよく使用されるCPU

- x86/x64系、PowerPC系、SHは単独あるいは提携CPUメーカーのみが製造
- MIPS、ARM
  - ハードウェア・コア(ハードマクロ)のライセンス方式
    - ライセンスを受けることで、あらゆるメーカーがCPUチップや、周辺回路を付加したSoC(System on a chip)を製造可能
    - ハードマクロのみでなく、ソフトマクロ(RTL記述)やアーキテクチャ自体のライセンスもあり
      - ソフトマクロのカスタマイズ可否はライセンス条件による
      - IntelのStrongARM／Xscale
    - 組込みCPUに向けたコア面積、動作周波数、消費電力

# MIPSアーキテクチャ(1)

- MIPSは32ビットの命令長
- 命令フォーマット(3種類: R, I, J)

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op		26 bit address			

op (6bits): オペコード (basic operation)

rs (5bits): 第一オペランドのレジスタ番号

rt (5bits): 第二ペランドのレジスタ番号

rd (5bits): 格納先レジスタ番号

shamt (5bits): シフト命令用(シフトビット数)

funct (6bits): 機能コード(function code)

15/26-bit address: 即値(定数)

Rフォーマット: 算術・論理演算

Iフォーマット: ロード・ストア、条件分岐

Jフォーマット: 無条件ジャンプ命令

# MIPSアーキテクチャ(2)

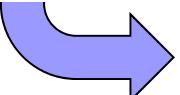
## ■ レジスタ間演算命令(算術・論理演算命令. Rフォーマット)

- 限られた数(32個)のレジスタ(registers)間で演算
  - MIPS(MIPS-32)では、32ビット×32個の汎用レジスタ
  - アセンブリレベルでは通し番号あるいは別名を使用
    - \$r0～\$r31: 通し番号指定
    - \$s0～\$s7: Cプログラムの変数に対応したレジスタ(16～23)
    - \$t0～\$t9: 一時値で使用(8～15, 24, 25)
    - その他のレジスタ

$f = (g+h) - (i+j)$  の計算

( $f \rightarrow \$r16$ ,  $g \rightarrow \$r17$ ,  $h \rightarrow \$r18$ ,  $i \rightarrow \$r19$ ,  $j \rightarrow \$r20$ に割り当て)

compile



```
add $r8, $r17, $r18      # $r8 = g + h  (add: 加算命令)
add $r9, $r19, $r20      # $r9 = i + j
sub $r16, $r8, $r9       # f = $r8 - $r9  (sub: 減算命令)
```

# MIPSアーキテクチャ(3)

## ■ メモリ参照命令(Iフォーマット)

- 一般的にコンピュータの処理では、メモリ内のデータを読み込み、演算を行い、結果をメモリに格納する
- 主な演算となる算術・論理演算はレジスタに対してのみ行われるため、メモリ上のデータをレジスタに移動、あるいはレジスタの値をメモリへ移動する必要あり
- ロード命令：メモリ→レジスタ “**lw**”: load word
- ストア命令：レジスタ→メモリ “**sw**”: store word

## $g = h + A[8]$ の計算(Aは4バイト整数の配列)

( $g \rightarrow \$r17$ ,  $h \rightarrow \$r18$ ,  $A$ のベースアドレス  $\rightarrow \$r19$ に割り当て)

オフセット → 32(\$r19)      ベースレジスタ  
 $8 * 4 = 32$       **lw**    \$r8, 32(\$r19)      # load from A[8]  
add    \$r17, \$r18, \$r8      #  $g = h + \$r8$

番地はバイト単位  
→ 1ワード(4バイト)データ  
は4番地ごとに配置される

# MIPSアーキテクチャ(4)

## ■ 条件分岐命令(Iフォーマット)

**beq** [レジスタ1], [レジスタ2], [飛び先ラベル] → branch if equal

**bne** [レジスタ1], [レジスタ2], [飛び先ラベル] → branch if not equal

## ■ 無条件分岐(ジャンプ)命令(Jフォーマット)

**j** [飛び先ラベル]

## ■ 例) if (i==j) f = g + h;

else f = g - h;

(f → \$r16, g → \$r17, h → \$r18, i → \$r19, j → \$r20)

bne \$r19, \$r20, L1 # i!=j ならば L1 へ

add \$r16, \$r17, \$r18 # f = g + h

j L2 # L2 へ

L1: sub \$r16, \$r17, \$r18 # f = g - h

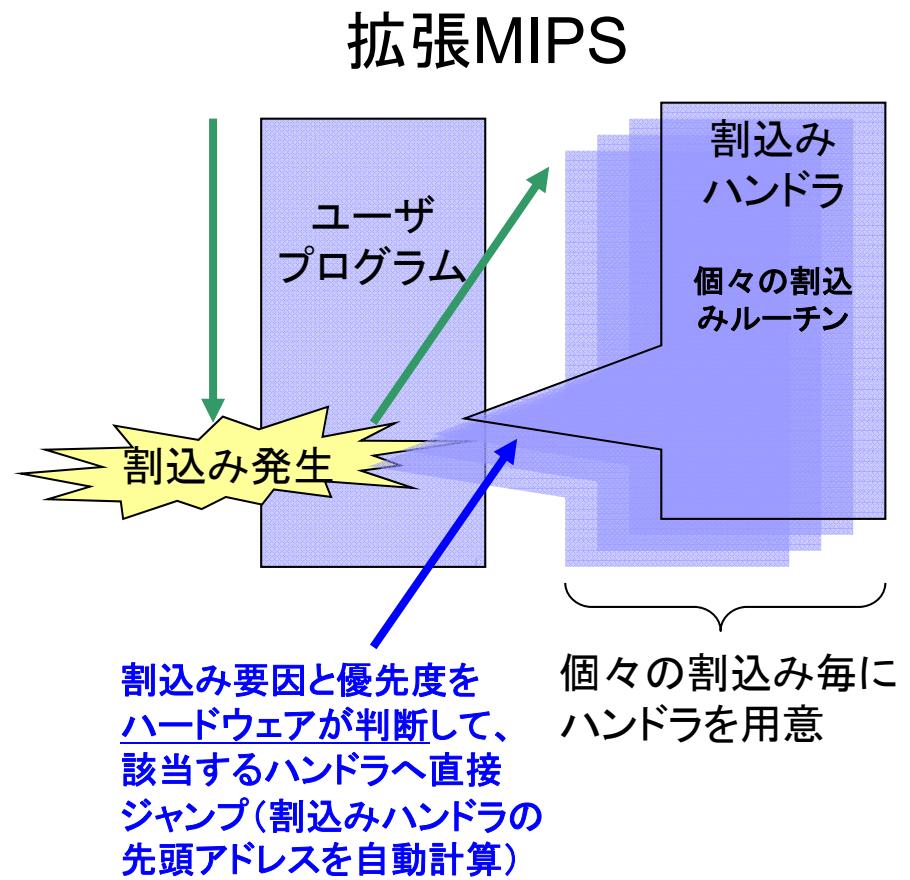
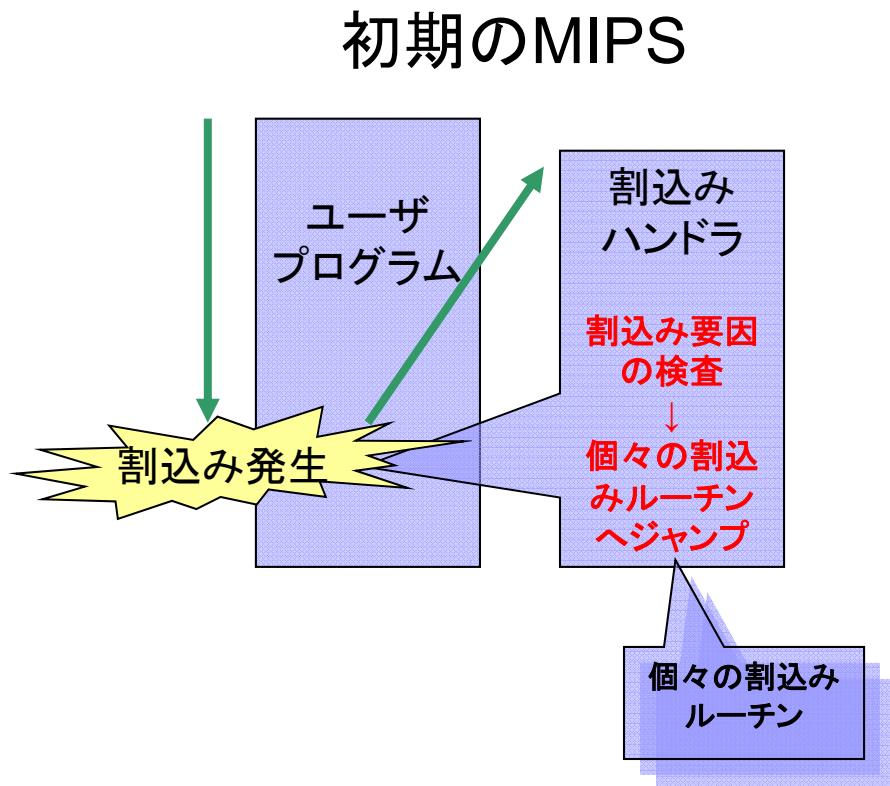
L2: ...

# MIPSアーキテクチャの拡張(1)

- MIPSプロセッサは元々、WSやサーバ機での使用が大半であったが、ライセンス戦略の流れで、現在は組込み用途での使用が主となっている
- 組込み／リアルタイムシステム用途を考慮して追加された仕組み
  - 優先ベクトル割込み方式／シャドウレジスタセットによる割込み応答性向上
    - ビット操作系命令の拡張
    - メモリ管理ユニットの拡張
    - コプロセッサ・サポートの拡張

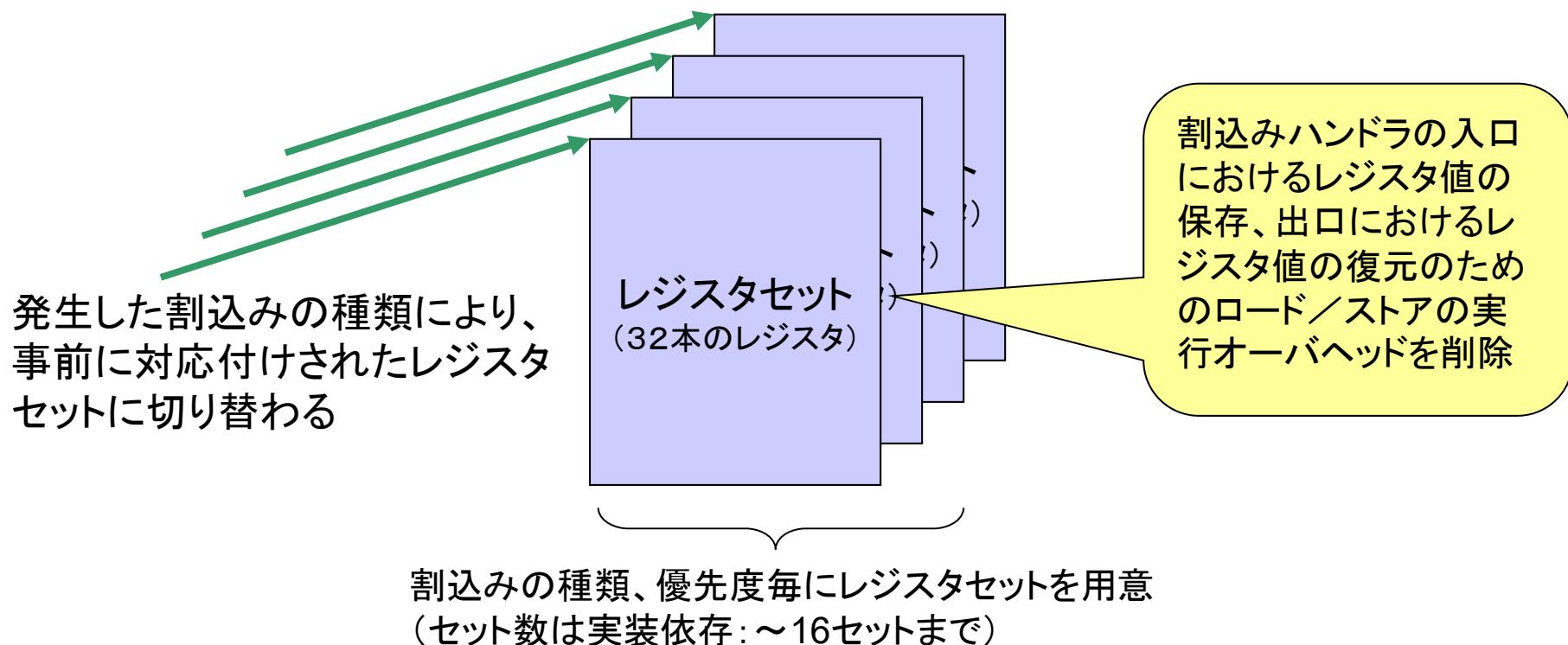
# MIPSアーキテクチャの拡張(2)

## ■ 優先ベクトル割込み



# MIPSアーキテクチャの拡張(3)

## ■ シャドウレジスタセット



skip

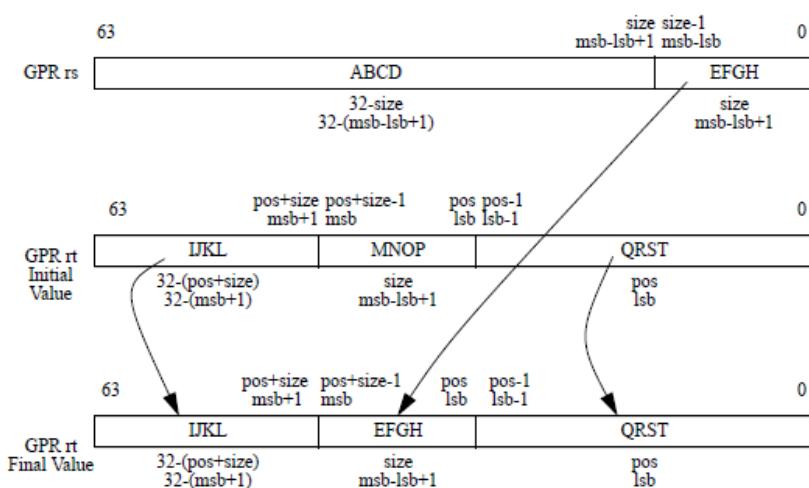
# MIPSアーキテクチャの拡張(4)

## ■ ビット操作系命令の拡張

### □ Insert bit field

アセンブリ記述

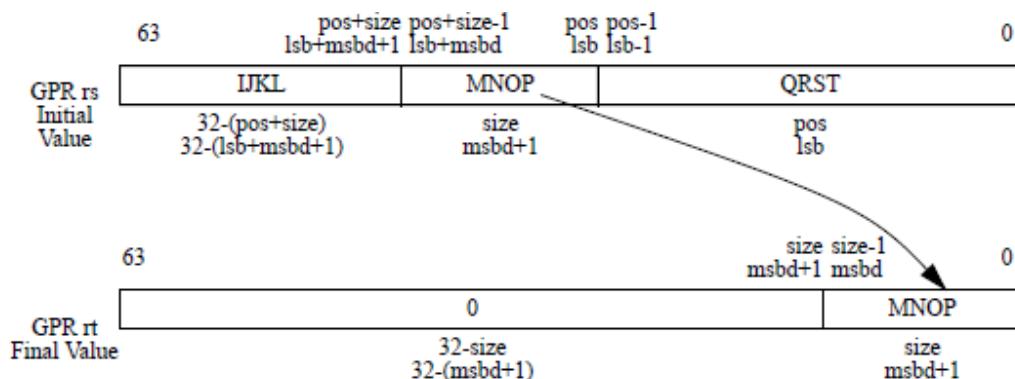
ins rt, rs, pos, size

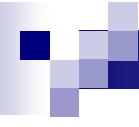


### □ Extract bit field

アセンブリ記述

ext rt, rs, pos, size



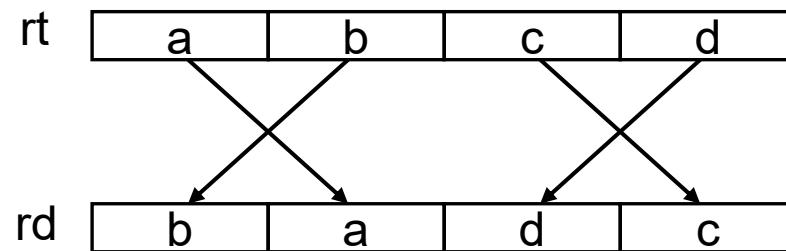


skip

# MIPSアーキテクチャの拡張(5)

## □ Swap bytes (halfwords)

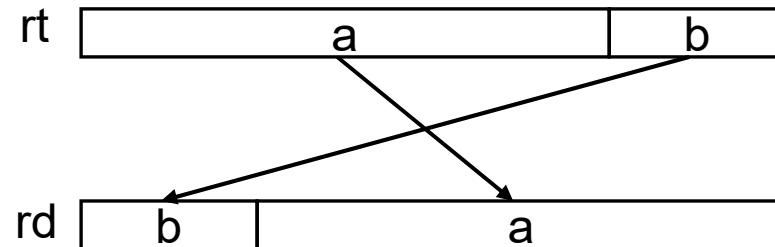
アセンブリ記述  
wsbh rd, rt

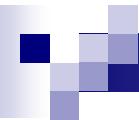


## □ Rotate right

アセンブリ記述  
rotr rd, rt, sa

エンディアン変換例  
lw \$r8, 0(\$r17)  
wsbh \$r8, \$r8  
rotr \$r8, \$r8, 16





skip

# MIPSアーキテクチャの拡張(6)

## ■ メモリ管理ユニットの拡張

- ページサイズの種類が増加
  - **1K、2K、4K～16M、64M、256M**バイト
    - メモリ使用量の少ないタスクの効率の良い保護
    - 大規模データの一括保護
- コンテキストレジスタに対するプログラマビリティの向上
  - CPU内のTLBフォーマットと、メモリ内のページテーブルフォーマットを分離し、柔軟なテーブル構築が可能

## ■ コプロセッサ・サポート拡張

- 32／64ビットCPUと、32／64ビットコプロセッサの自由な組合せが可能

# ARMの戦略(1)

- MIPS Technologies社と同様、ARM社に半導体工場は無く、プロセッサコアのデザインをライセンス販売
- 1000社以上のパートナー(ライセンシ)
- 組込みプロセッサの中で圧倒的なシェア
  - 主に携帯電話の普及による
  - スマートフォン分野では90%以上
- ARMファミリ
  - ARM7、ARM9、ARM11 → Cortex-A\*/R\*/M\*
  - Intel StrongARM、XScale

近年は64ビット化

iPhone/iPadなど

# ARMの戦略(2)

## ■ 段階的なコア群であらゆる用途に対応

### □ ARM7

- 3段パイプライン → 100MHz前後
- 命令・データ・ユニファイドキャッシュ、16ビット外部バス

### □ ARM9

- 5段パイプライン → 200MHz前後
- 命令／データ・セパレートキャッシュ、32ビット外部バス

### □ ARM11

- 8段パイプライン → 400MHz～
- ノンブロッキングキャッシュ(hit under miss)、64ビット外部バス
- コプロセッサ命令(浮動小数点演算)
- 分岐予測を採用

### □ Cortex-A\*

- 8～15段(浮動小数は24段)パイプライン → 600MHz～3.3GHz
- スーパースカラ(Out-of-Order構成もある)
- 2～8コアが主流
- NEON: SIMDエンジン搭載

最近のCortexのパイプ  
ライン段数は未発表

# ARMアーキテクチャ(1)

- RISCに分類されるが、通常のRISCよりは複雑な命令となっている
    - 全ての命令が条件コード付
    - 算術論理演算とシフト演算が統合可能
    - 複数レジスタ転送命令
  - 拡張機能
    - Thumb命令(16ビット長命令)
    - skip* □ DSP命令拡張
    - skip* □ Jazelle(Javaバイトコード自動変換モード)
      - big.LITTLE 構成マルチコア
    - skip* □ TrustZone(隔離実行モード)
- 
- コードサイズの削減が目的

# ARMアーキテクチャ(2)

- ARMの基本は32ビットの命令長
- 全命令(一部除く)が条件付実行
  - 命令の上位4ビットは条件を示すフィールド



- 状態レジスタ(CPSR:後述)の状態コード(N,Z,C,V)の値にしたがって、条件が成立すれば実行、不成立のときはスキップ
- 状態コードは、算術論理演算命令の演算結果にしたがってセットされる
  - ただし、各算術論理演算命令は、状態コードを更新するかどうかが命令内で指定されている → Sビット

# ARMアーキテクチャ(3)

最大公約数を求める例

```
int gcd(int i, int j)
{
    while (i != j) {
        if (i > j)
            i -= j;
        else
            j -= i;
    }
    return i;
}
```

これらのレジスタは疑似的な指定  
N==0, Z==0 ならば減算実行

b test /\* testへジャンプ \*/  
loop subgt Ri,Ri,Rj /\* > の場合 Ri = Ri - Rj \*/  
suble Rj,Rj,Ri /\* <= の場合 Rj = Rj - Ri \*/  
test cmp Ri,Rj /\* RiとRjを比較\*/  
bne loop /\* != の場合 loopへジャンプ \*/

状態コードをセットする命令

コメントは  
/\* コメント文 \*/  
# コメント文  
@コメント文 } のいずれか

# ARMアーキテクチャ(4)

- 算術論理演算とシフト演算の統合
  - 算術論理演算における3つのオペランドのうち、第2ソースオペランド(レジスタ)をシフト可能

$$R1 = R2 + (R3 \ll 10)$$

- シフトの種類: 論理左シフト(LSL)、論理右シフト(LSR)、算術右シフト(ASR)、右ローテイト(ROR)

ADD r1, r2, r3, LSL #10 @ r1 = r2 + (r3 << 10)

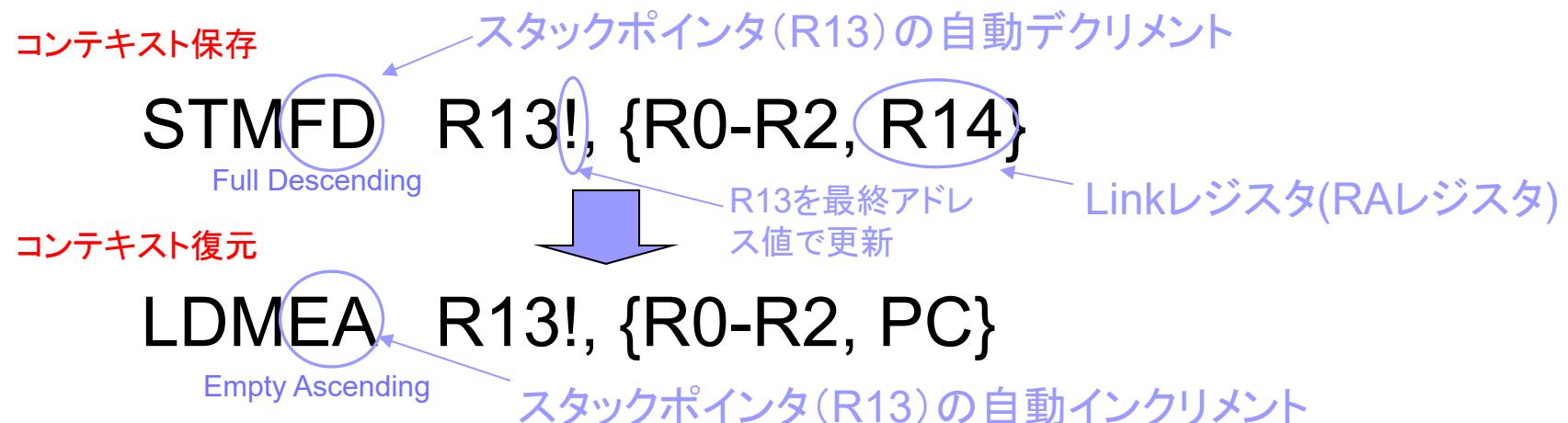
- 命令効率は向上するが、ALUの設計においてクリティカルパス遅延が増大する可能性
    - シフト演算→算術論理演算を直列に実行するため

# ARMアーキテクチャ(5)

## ■ 複数レジスタ転送命令

- 16個(後述)のレジスタ中、任意のサブセット(または全て)にメモリからロード、またはメモリへストア  
例) LDM R8, {R0, R2, R4-R7} # 6個のレジスタ

- OSによるユーザ処理状態の保存／復元に使用可能
  - コンテキスト切り替えルーチンの縮小

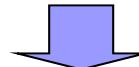


# ARMアーキテクチャ(6)

## ■ レジスタ構成

R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8	R8 FIQ					
R9	R9 FIQ					
R10	R10 FIQ					
R11	R11 FIQ					
R12	R12 FIQ					
R13(SP)	R13 FIQ	R13 SVC	R13 ABT	R13 IRQ	R13 UND	
R14(RA)	R14 FIQ	R14 SVC	R14 ABT	R14 IRQ	R14 UND	
R15(PC)						

実行モードの遷移により、レジスタセットの一部が自動的に切り替わる  
(切り替え前のPCが切り替え後のRAに保存される)



ハンドラは、切り替わった(専用の)レジスタは保存／復元無しで使用可能

# ARMアーキテクチャ(7)

## ■ 例外・割込みエントリ：ベクトル・アドレス方式

例外	モード	ベクトル・アドレス	アドレスに置かれる命令
リセット	SVC	0x00000000	ルーチンへのジャンプ命令
未定義命令割込み	UND	0x00000004	ルーチンへのジャンプ命令
SW割込み	SVC	0x00000008	ルーチンへのジャンプ命令
命令・メモリ・フォルト	ABT	0x0000000C	ルーチンへのジャンプ命令
データ・メモリ・フォルト	ABT	0x00000010	ルーチンへのジャンプ命令
標準割込み	IRQ	0x00000018	ルーチンへのジャンプ命令
高速割込み	FIQ	0x0000001C	ルーチンの先頭命令
			▪
			▪

# ARMアーキテクチャの拡張(1)

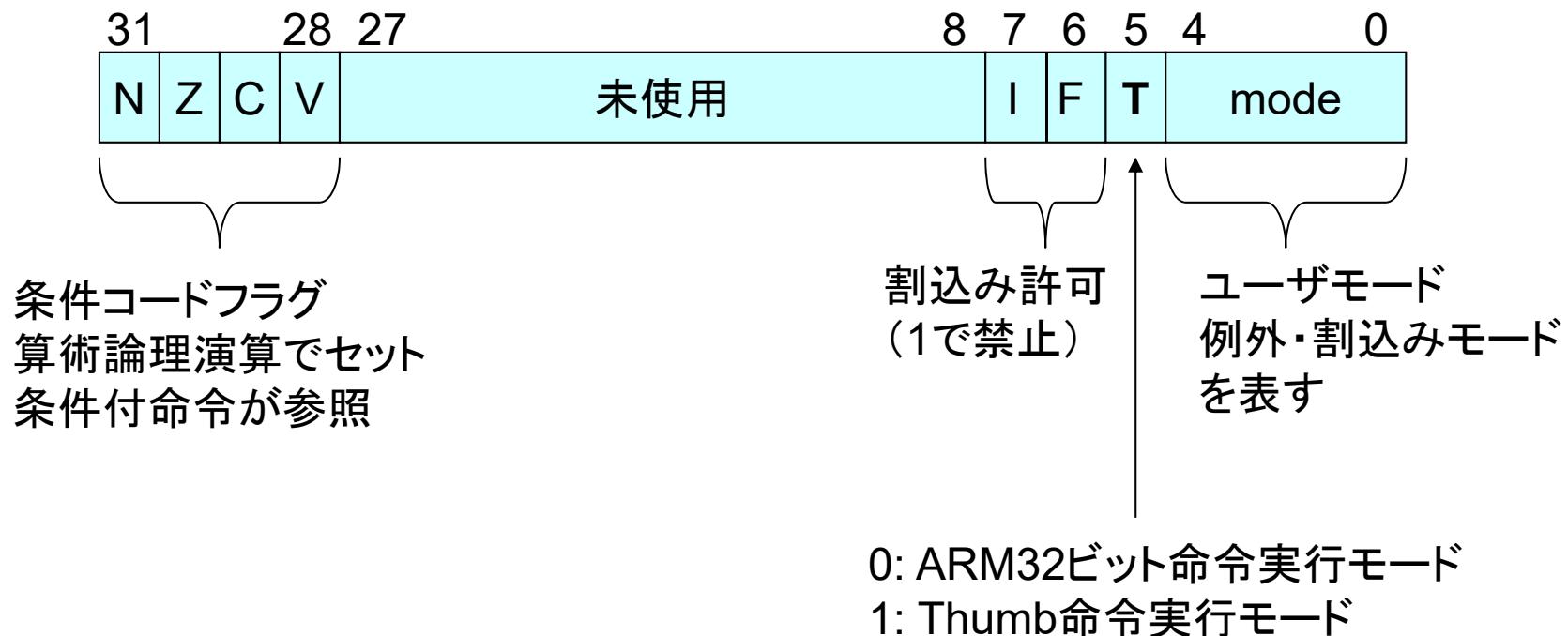
## ■ Thumb 命令セット

- 命令長を小さくし(16ビット)、コード密度の問題に対応
- アプリケーションバイナリ内で、標準の32ビットARM命令との混在が可能(ただし関数・手続き単位)
- 状態レジスタ(CPSR)内の“T”ビットにより、命令ストリームの解釈(ARM命令かThumb命令か)が決定
- Branch exchangeタイプの分岐命令("BX")の実行により、Tビットが反転。同時に、切り替え先命令ストリームへジャンプ
- Thumb命令ストリーム実行中に例外が発生した場合、自動的にARM命令実行モードへ移行

## ■ 現在では更なる拡張版のThumb-2／ThumbEEが使用されている

# ARMアーキテクチャの拡張(2)

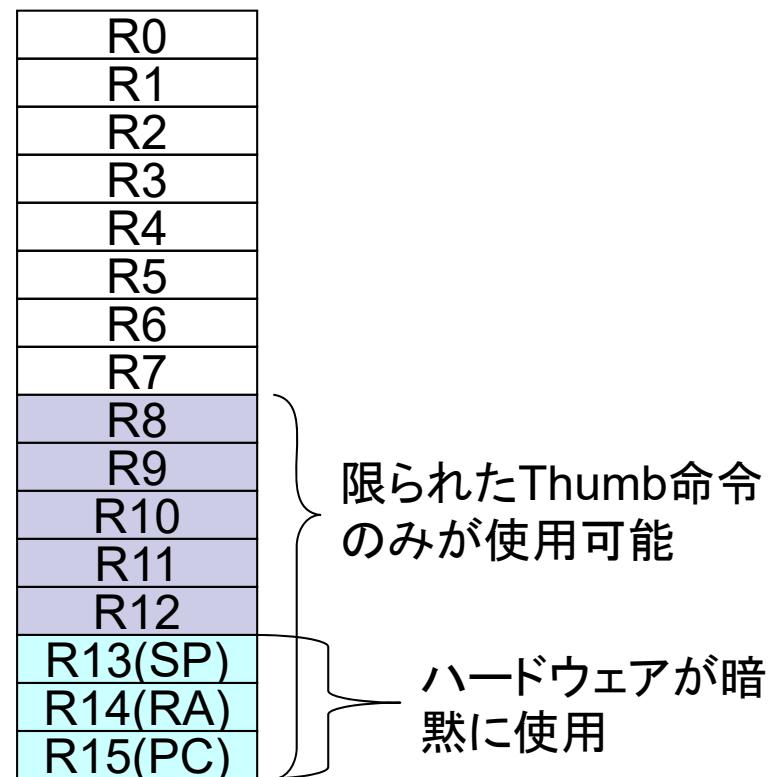
## ■ カレント・プログラム・ステータス・レジスタ (CPSR)



# ARMアーキテクチャの拡張(3)

## ■ ARM命令とThumb命令の相違点

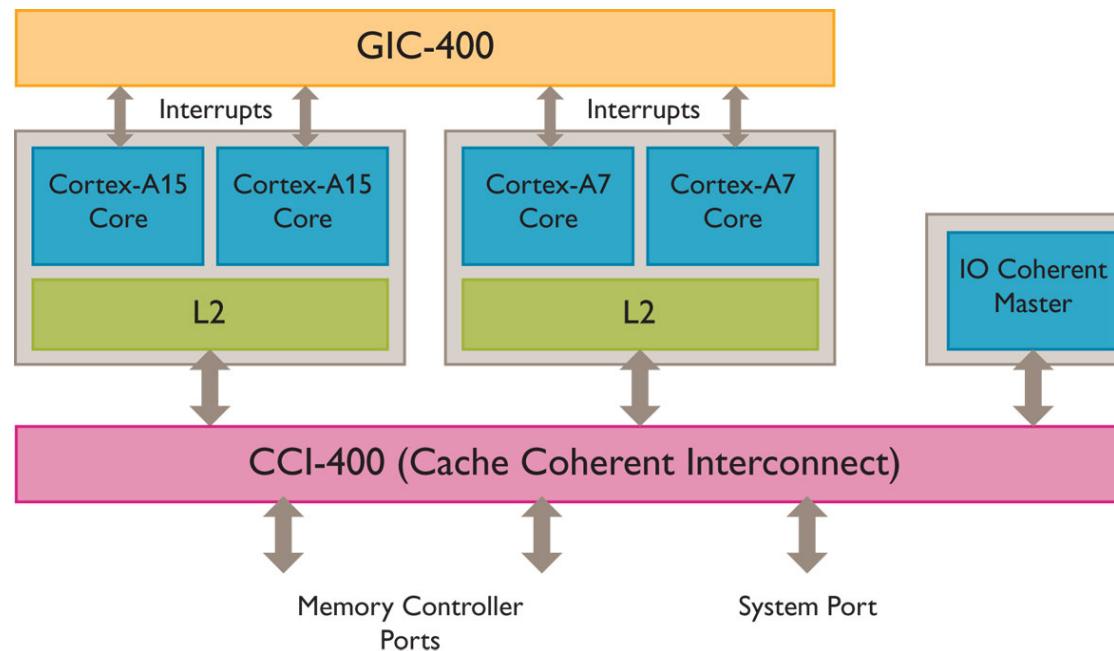
- 使用レジスタに制限
- ほとんどのThumb命令が無条件実行
- 多くのThumb命令が2オペランド指定  
(デスティネーションは一方のソースと同一)



# ARMアーキテクチャの拡張(4)

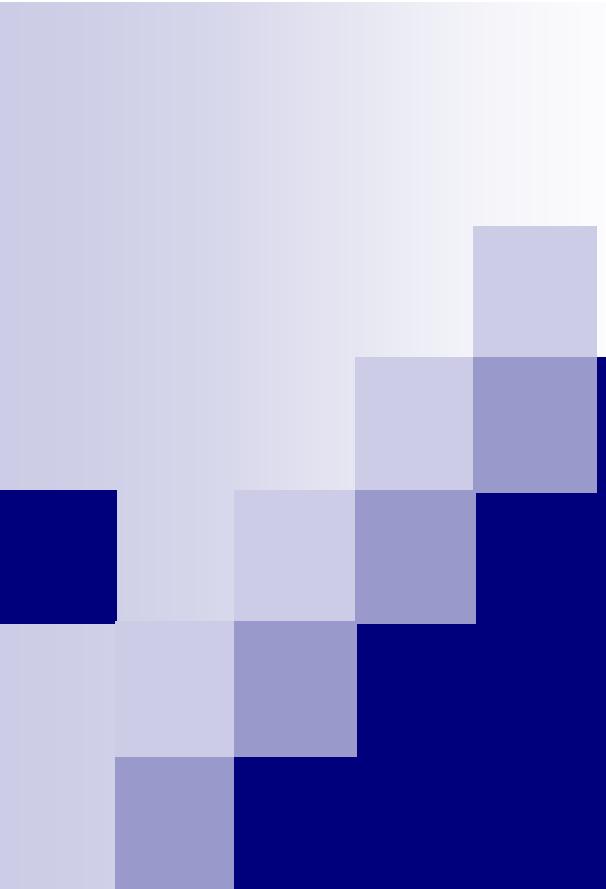
## ■ big.LITTLE構成マルチコア

- 高性能コア(例えばCortex-A15)と、低消費電力コア(例えばCortex-A7)が共存するマルチコア
- OSによるタスクのコアへの割当



# まとめ

- MIPS、ARMなどの組込み用CPUは、割込みに対する高速応答を可能とする仕組みを持つ
  - リアルタイム処理では、粒度の小さい割込みが頻発するため
- MIPSは、組込みアプリケーションで頻出するビット／バイト操作の命令を追加
  - 効率の良いプロトコル処理など
- ARMでは、コードサイズを小さくするために、条件付命令、シフト演算統合、複数レジスタ転送命令、Thumb命令を提供
  - コード用メモリ／ROMの縮小を強く意識



# I470F 統合アーキテクチャ

命令実行パイプライン

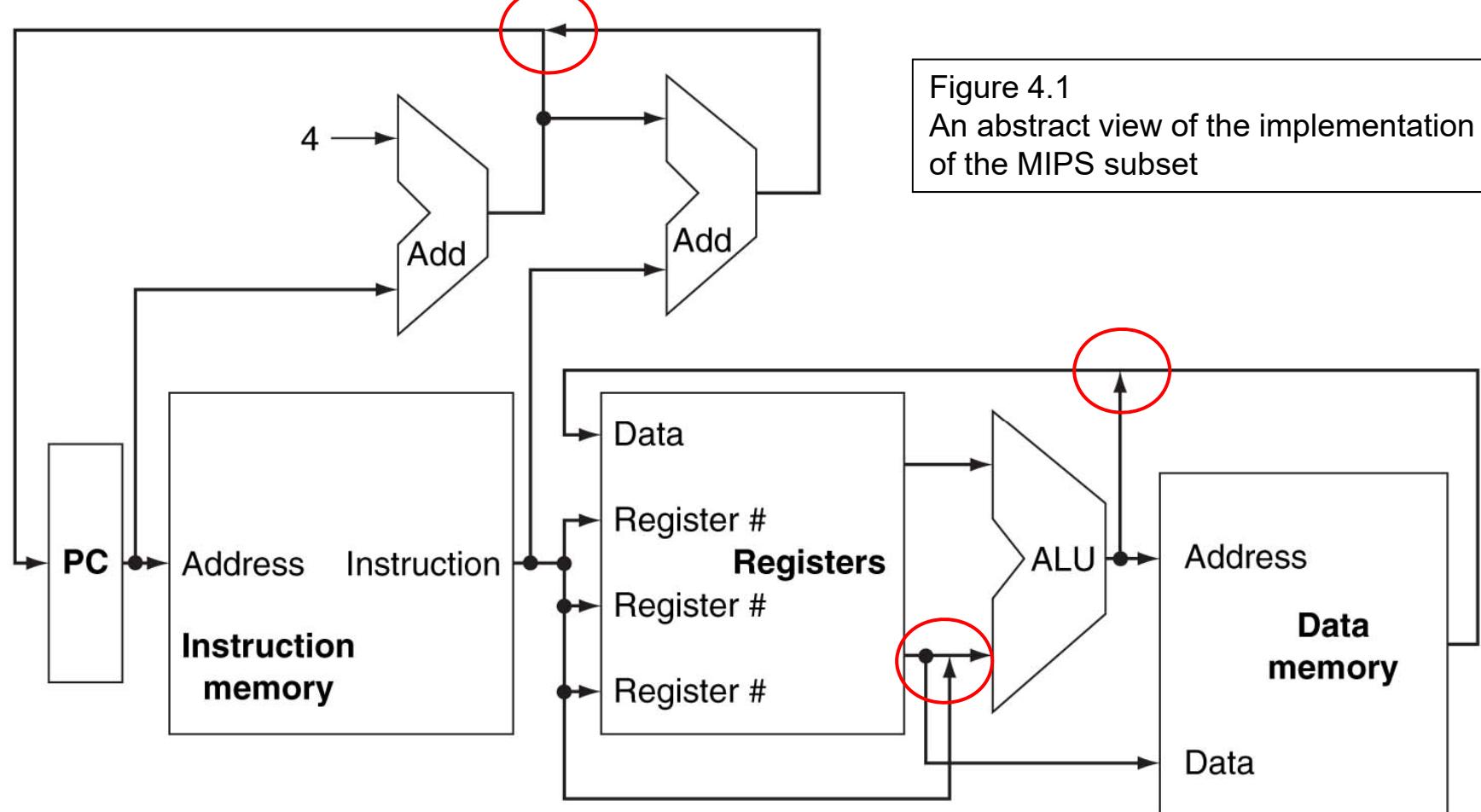
# Introduction of a processor (1)

- 扱う命令群 (MIPSサブセット)
  - メモリ参照命令: lw, sw
  - 算術論理演算命令: add, sub, and, or
  - 制御命令: beq (conditional), j (unconditional)
- ほとんどの命令実行で同一／同様の処理
  - メモリから命令が読み込まれる
    - プログラムカウンタ (PC)により命令アドレスが提供される
  - レジスタファイルからレジスタ値(0,1,または2個)が読み出される
    - lw: 1個のレジスタ値
    - add/sub/and/or/beq/sw: 2個のレジスタ値
    - j: 0個のレジスタ値
  - ALU(Arithmetic Logic Unit, 算術論理演算ユニット)が使用される
    - 算術論理演算命令 → 算術論理演算
    - メモリ参照命令(lw/sw) → アドレス計算
    - 制御(条件分岐)命令(beq) → レジスタ値の比較

# Introduction of a processor (2)

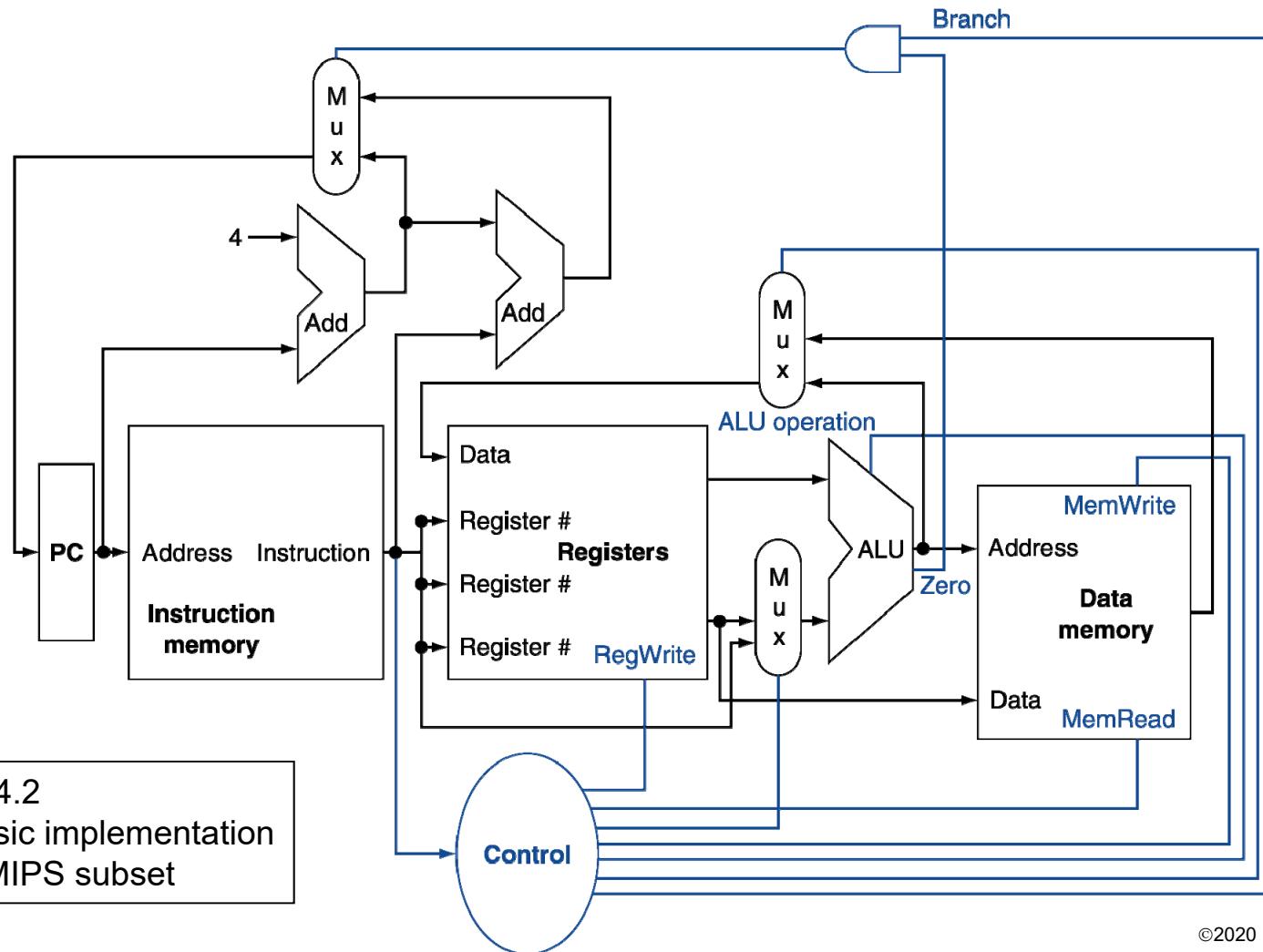
## ■ 基本的なプロセッサの抽象的構成

全ての命令が1クロックサイクルで完了するシンプルな設計の場合



# Introduction of a processor (3)

## ■ 少し具体的な構成(完全ではない)



# Introduction of a processor (4)

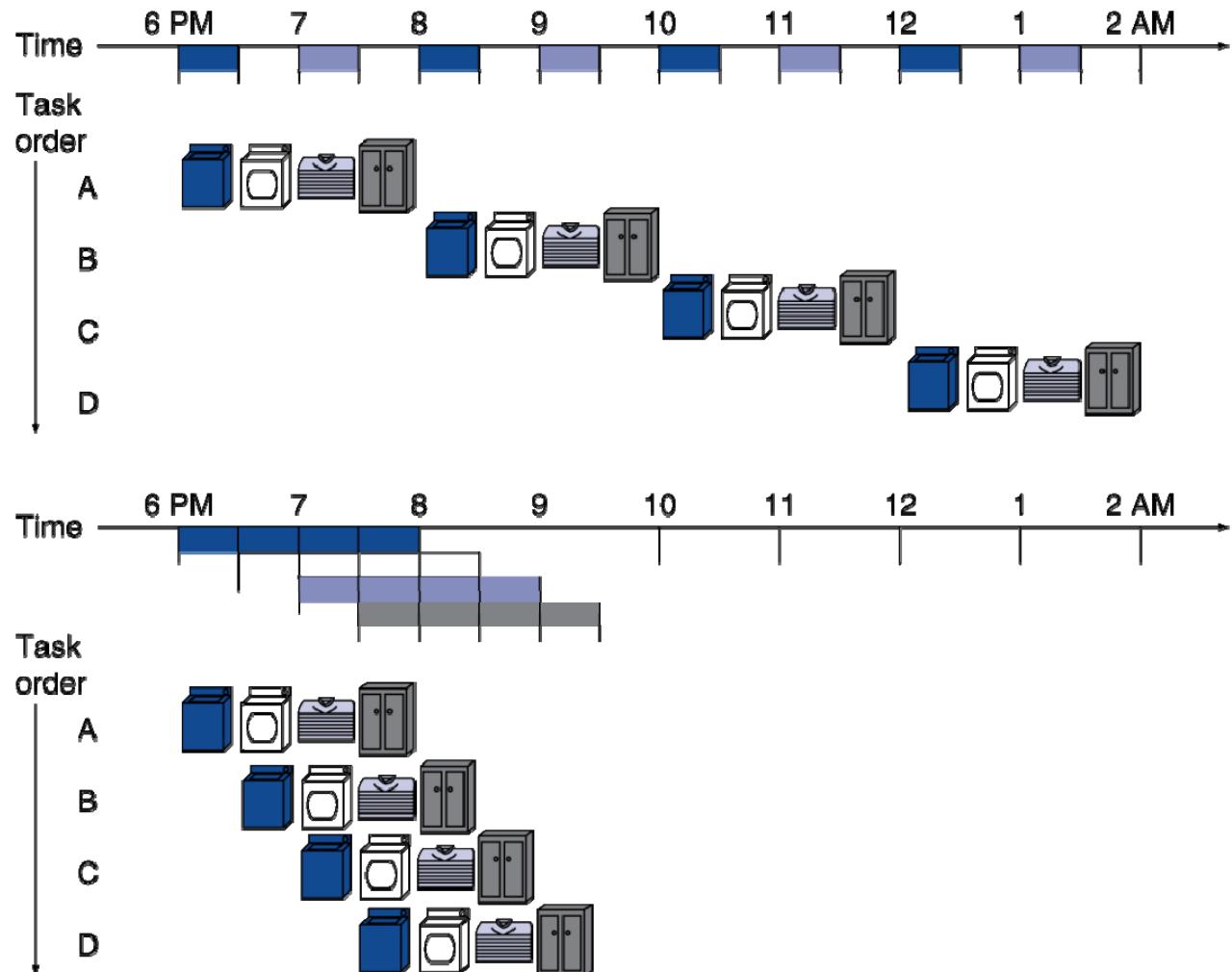
- 以上のシングルサイクル実行方式は正しく動作するが、非効率性のために実際に使われることはほとんどない
  - 全ての命令が1クロックサイクルで実行される
  - 最も時間を要する命令(=lw)にクロックサイクル時間を合わせる必要がある
    - lwは5ステップ(命令フェッチ, レジスタ読み出し, アドレス計算, データメモリ参照, レジスタ書き込み)を逐次的に処理する必要がある
  - 命令ごとに異なるサイクル数で実行する方法もあるが、これを拡張したほうが得策  
⇒パイプライン化

# An Overview of Pipelining (1)

## Pipelining

Figure 4.25  
The laundry analogy for pipelining

1. Washer
2. Dryer
3. Folding
4. Putting away



# An Overview of Pipelining (2)

- 命令実行のスループットを上げることによる性能向上

Figure 4.27

- 5-stage pipeline

- 1) 命令フェッチ

- 200ps

- 2) 命令デコード & レジスタリード

- 100ps

- 3) 実行 or アドレス計算

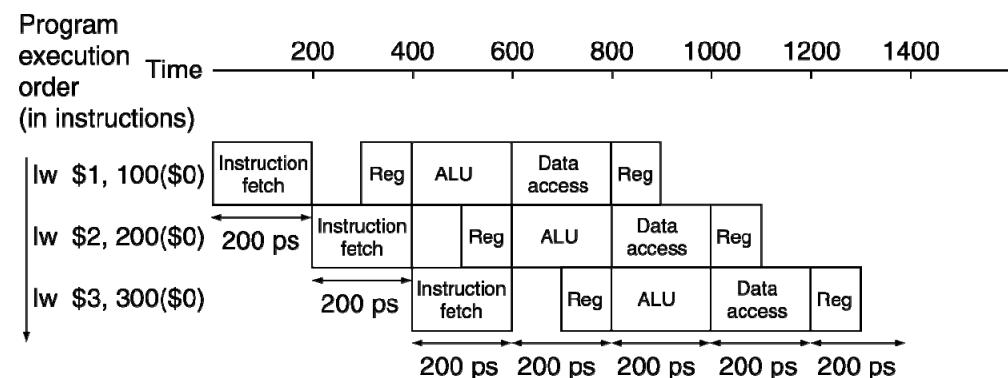
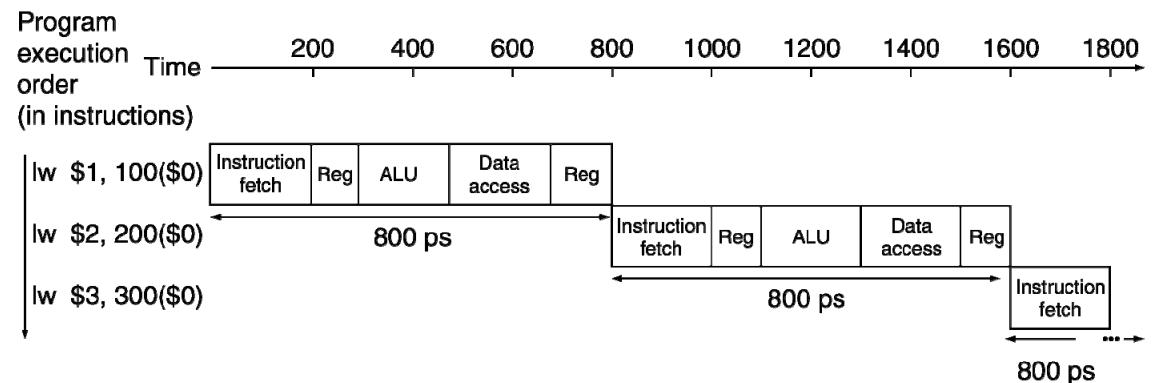
- 200ps

- 4) データメモリアクセス

- 200ps

- 5) レジスタ書き込み

- 100ps



注: レジスタ読み出しと書き込みはステージの前半と後半とでずらしている

# An Overview of Pipelining (3)

Formula:

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Ideal speedup is the number of stages in the pipeline.

この理想的なスピードアップが達成できるだろうか？

→ いくつかの障害(ハザード)がある

---

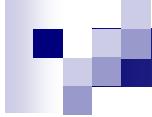
パイプライン処理の狙い



スループットの向上 *opposed to* 各命令の実行時間の短縮

# An Overview of Pipelining (4)

- MIPS(RISC)命令セットとパイプラインの相性
  - 固定命令長(32ビット)
    - 命令フェッチステージのサイクル時間を固定可能
  - 少数(3種類)の命令フォーマット
    - デコード、オペランド読み出しステージのサイクル時間を固定可能
  - メモリオペランドは load(lw)、store(sw) のみ
    - データメモリアクセスはメモリアクセスステージのみに限定
    - ステージ間でデータメモリアクセスの衝突が起こらない  
(CISC命令セットでは、メモリ内データを算術論理演算オペランドに直接指定可能であり、データメモリでの衝突が発生する可能性あり)



# An Overview of Pipelining (5)

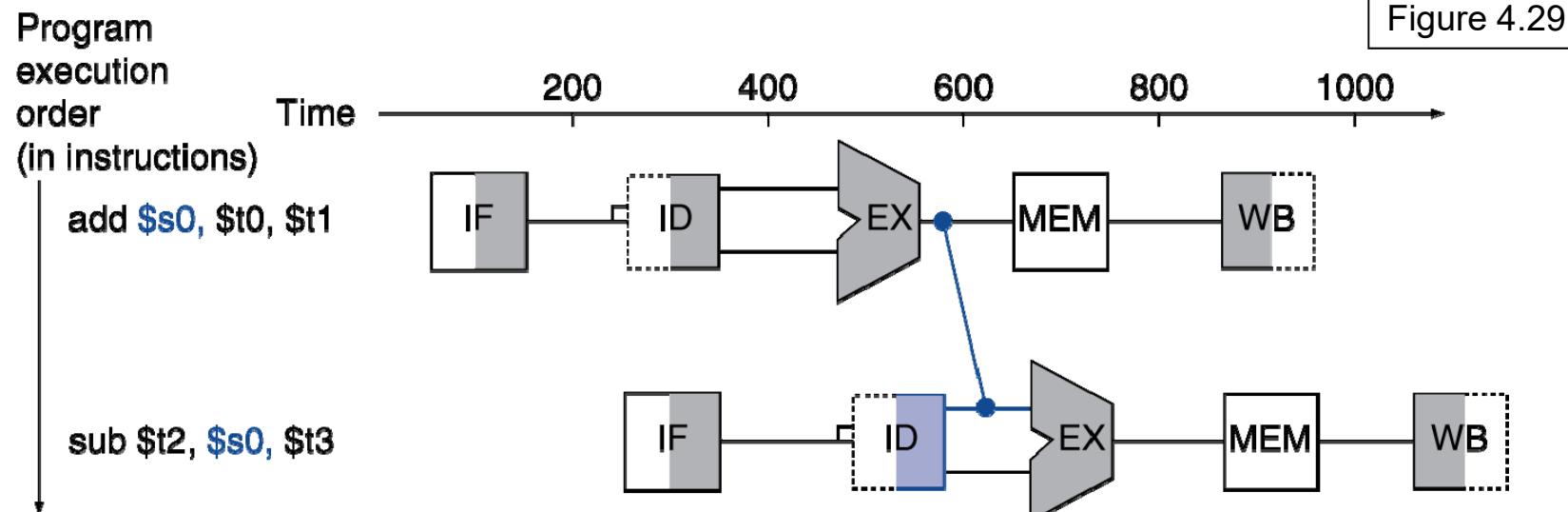
- 理想的パイプライン処理に対する障害(パイプラインハザード)
  - 構造ハザード(structural hazard)
  - データハザード(data hazard)
  - 制御ハザード(control hazard)

# An Overview of Pipelining (6)

## ■ データハザード(data hazard)

add \$s0, \$t0, \$t1  
sub \$t2, \$s0, \$t3

### □ 解決方法: フォワーディング (forwarding/bypassing)



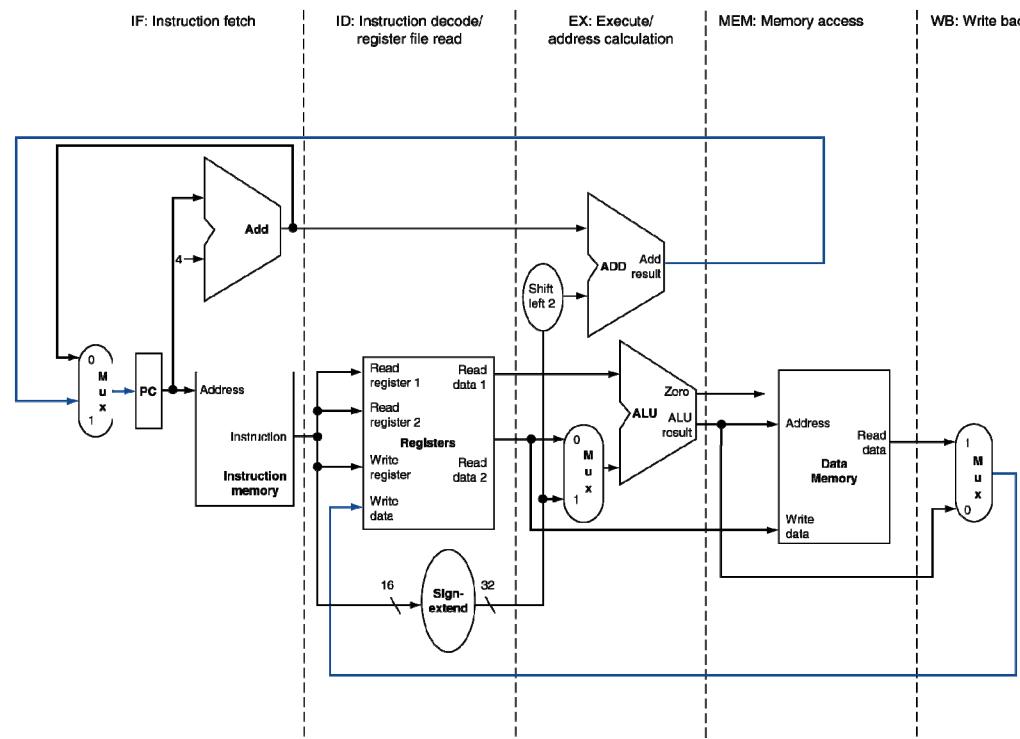
→ 詳しくは後ほど。

# A Pipelined Datapath (1)

- 一つの命令の実行を5つのステージに分割

- IF: 命令フェッチ
- ID: 命令デコード & レジスタリード
- EX: 実行 or アドレス計算
- MEM: メモリアクセス
- WB: ライトバック(レジスタ書き込み)

Figure 4.33

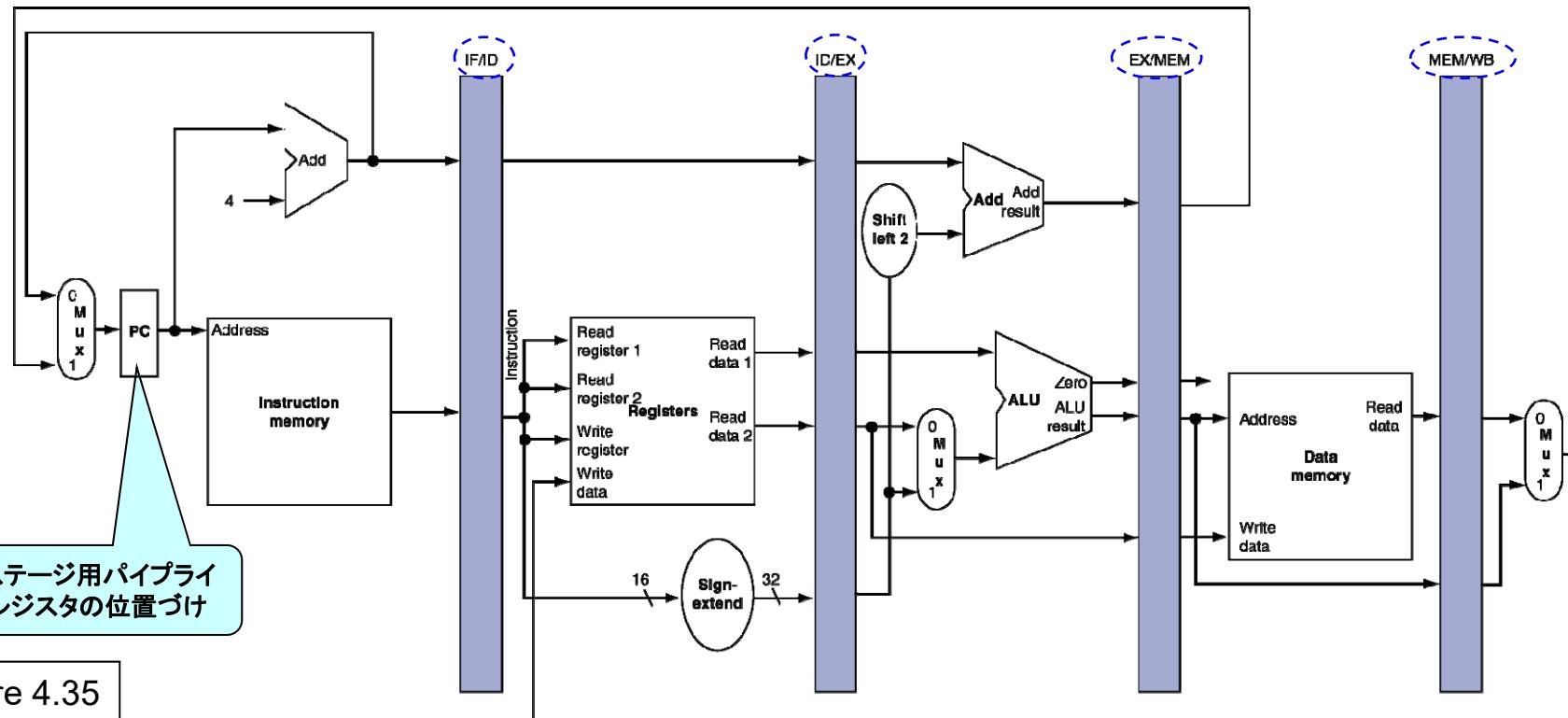


# A Pipelined Datapath (2)

## ■ パイプラインレジスタ

□ 命令は必要な値を保持しながら(右方向へ)ステージを移動していく

■ 移動前のステージは次の命令が使用するため



# A Pipelined Datapath (3)

## Iwの実行例

### ■ IFステージ(for Iw)

#### □ 命令フェッチ & 次PC値の生成

- 次PC値はIF/IDパイプラインレジスタにも書かれる

(EXステージにおける分岐先アドレス計算で使用される可能性)

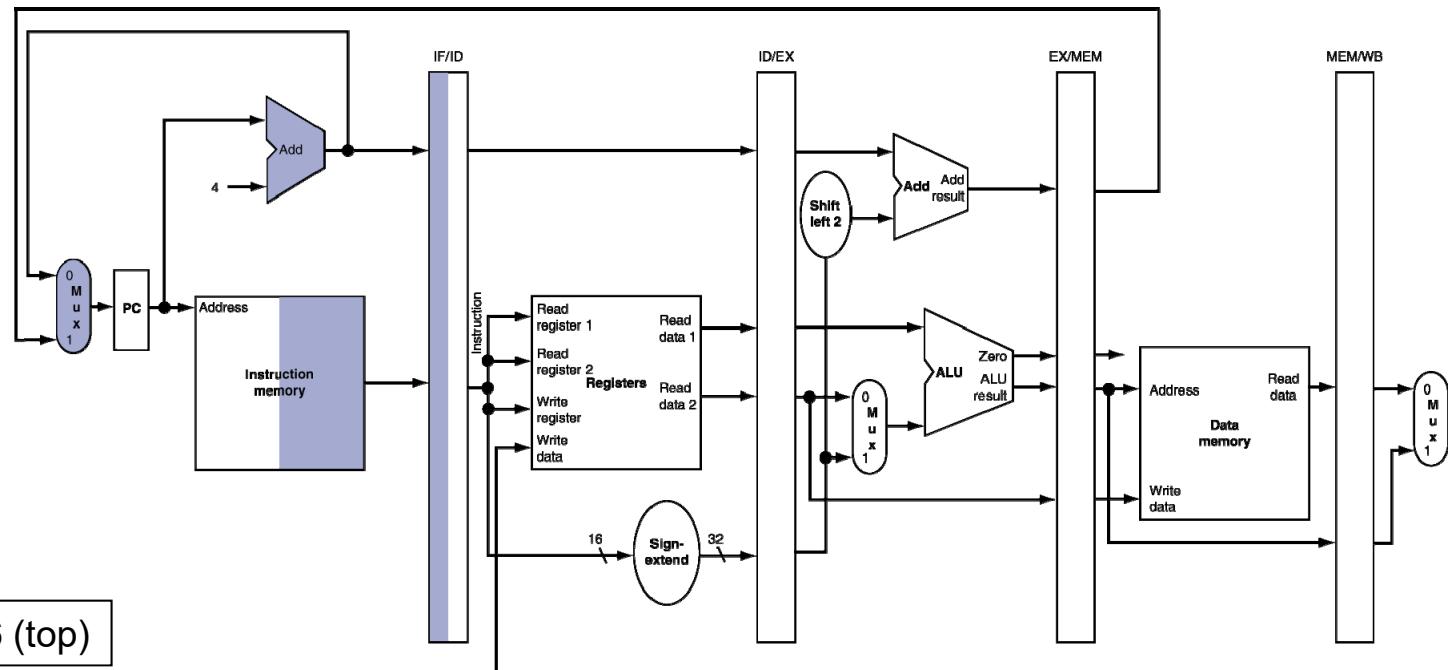
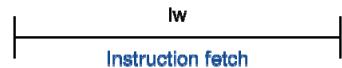


Figure 4.36 (top)

# A Pipelined Datapath (4)

## ■ IDステージ(for lw)

### □ 命令デコード & レジスタファイル読み出し

- 2つのレジスタ値と符号拡張即値がID/EXパイプラインレジスタに書かれる(後のステージで使用可能性のあるもの全て用意しておく)(例えば、rtレジスタ値はlwでは使用しない)

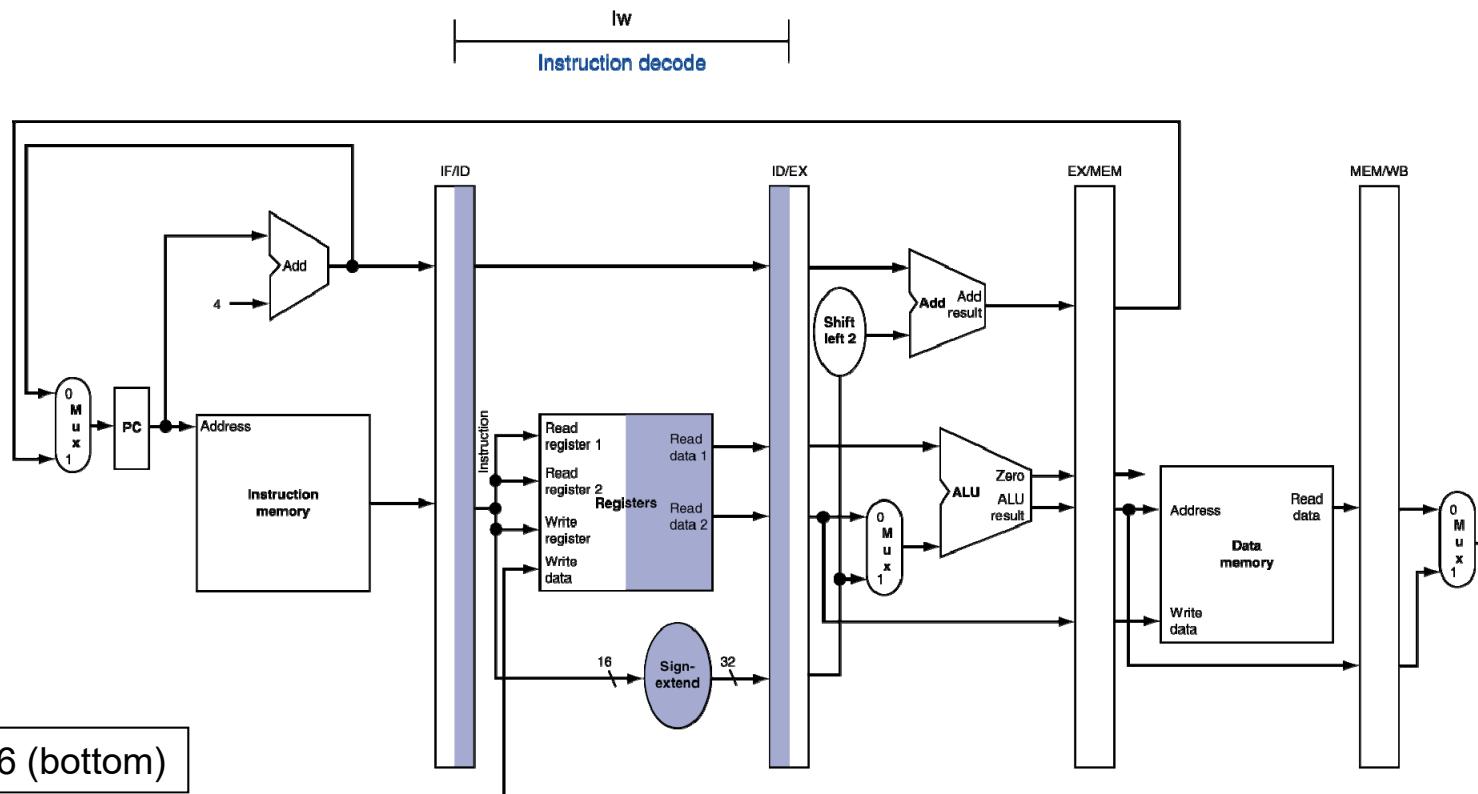


Figure 4.36 (bottom)

# A Pipelined Datapath (5)

## ■ EXステージ

### □ (実行、または) アドレス計算(for lw)

- ID/EXパイプラインレジスタ内の、第一レジスタ値(Read data 1)と符号拡張即値同士の加算を行うことでアドレス値を計算。これをEX/MEMパイプラインレジスタに書き込む

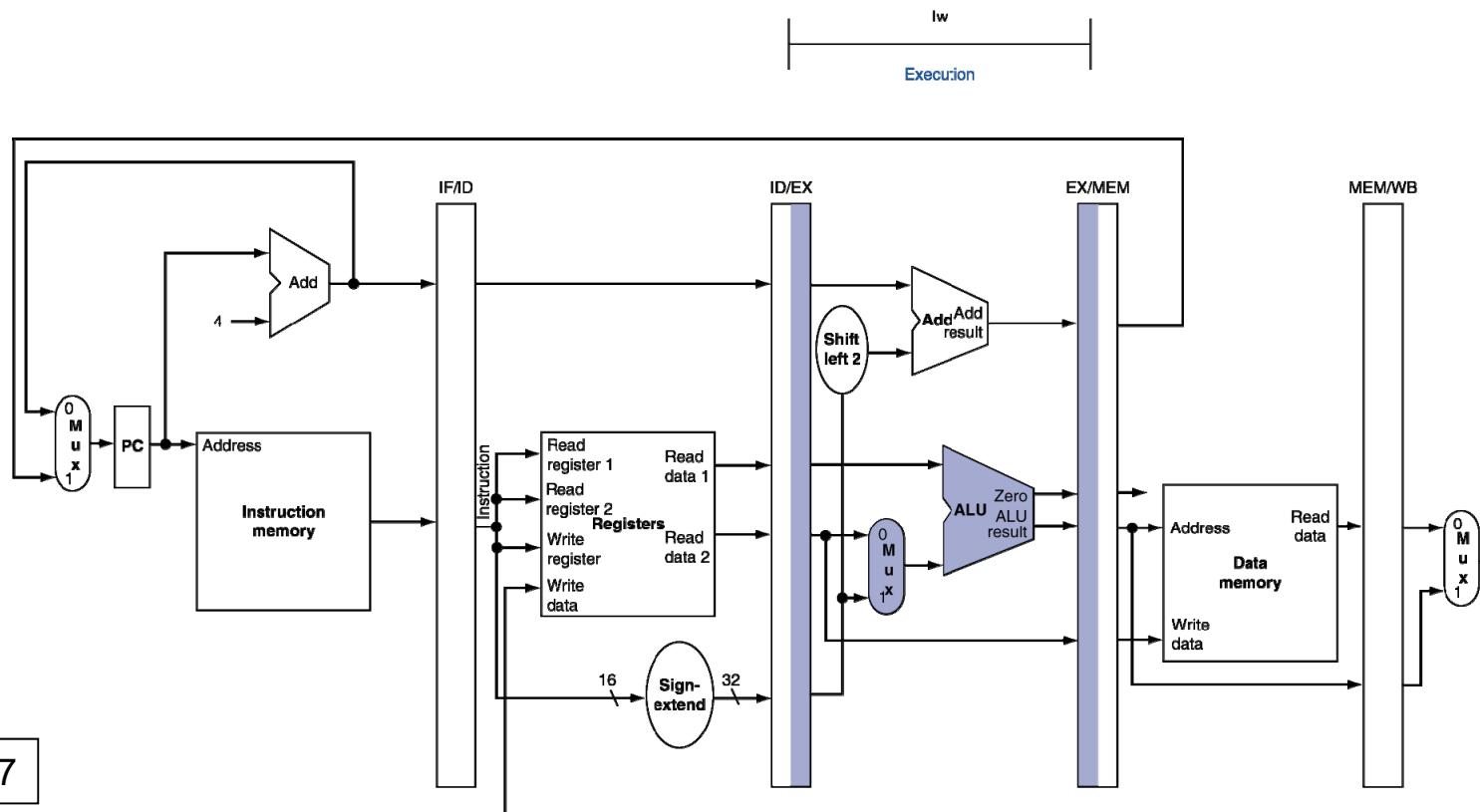


Figure 4.37

# A Pipelined Datapath (6)

## ■ MEMステージ(for lw)

### □ メモリアクセス

- EX/MEMパイプラインレジスタ内のアドレス値を使用してメモリ読み出しを行う。読み出した値をMEM/WBパイプラインレジスタに書き込む

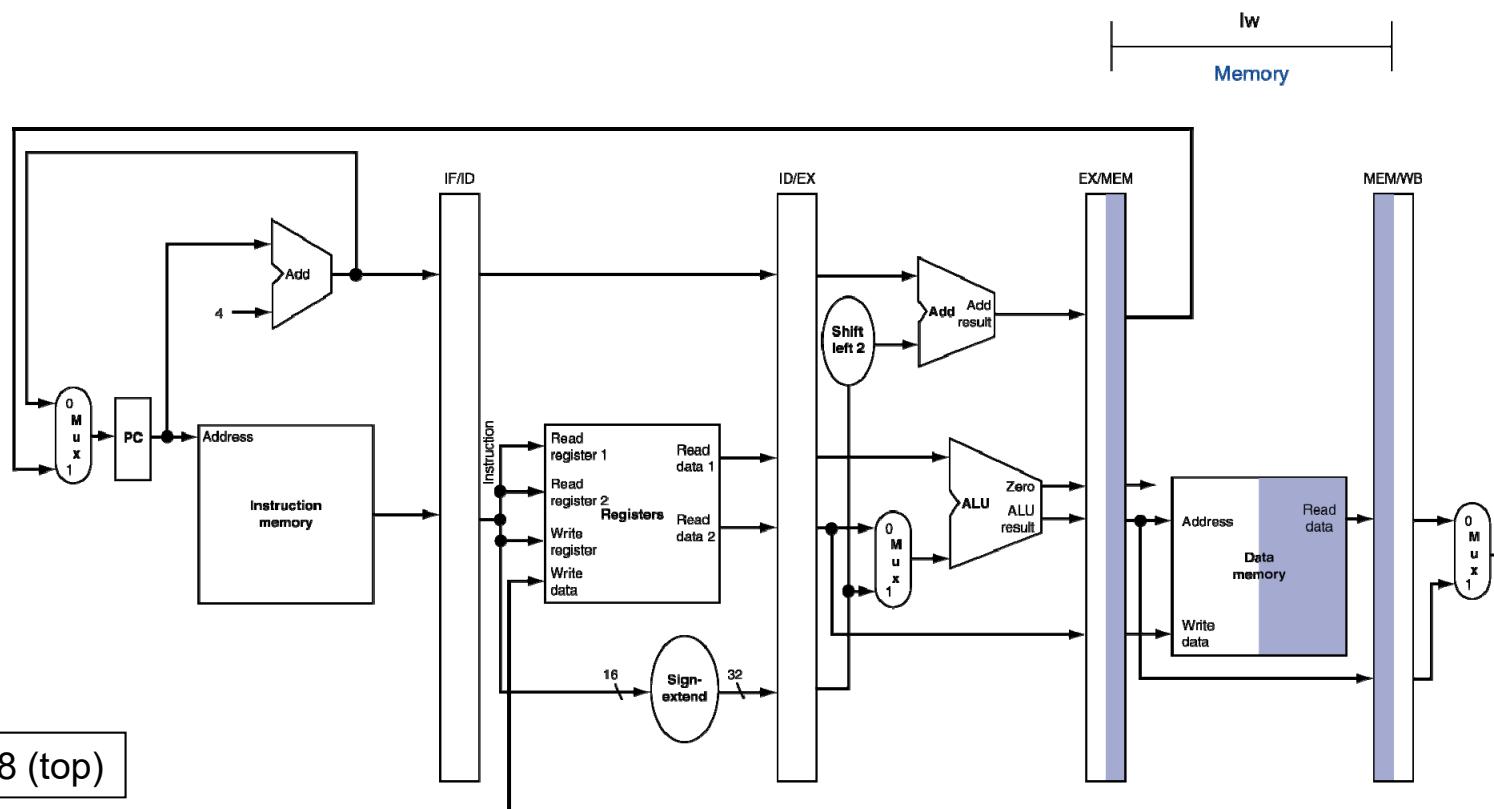


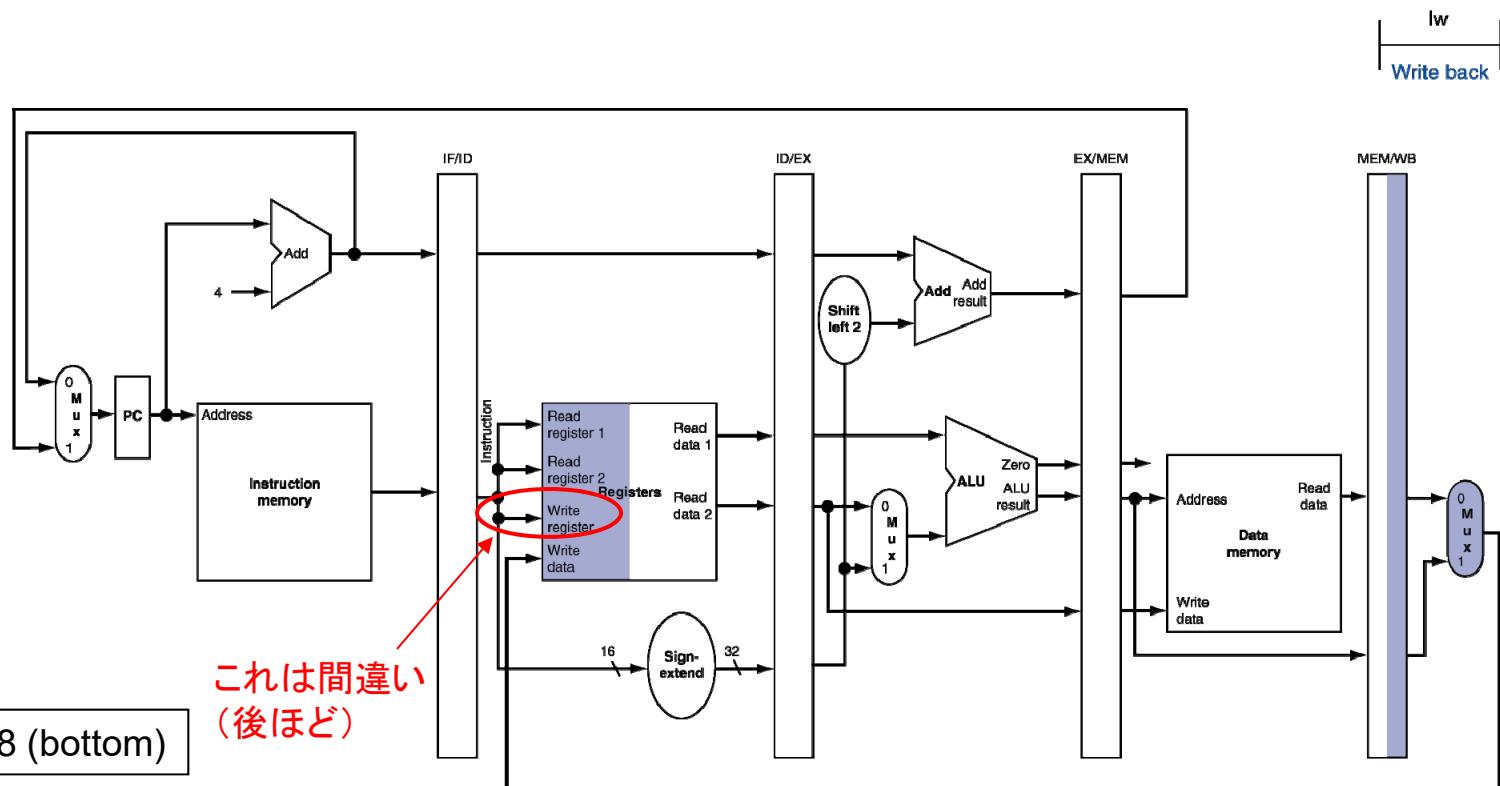
Figure 4.38 (top)

# A Pipelined Datapath (7)

## ■ WBステージ(for lw)

### □ ライトバック(レジスタ書き込み)

- MEM/WBパイプラインレジスタ内のメモリ読み出し値をレジスタファイル内のターゲットレジスタに書き込む



# A Pipelined Datapath (8)

swの実行例 (IF, IDステージは lw の場合と同じ)

- EXステージ(for sw) (このステージもlwと同じ)

- (実行、または) アドレス計算

- lwの場合と同様にアドレス計算を行い、EX/MEMパイプラインレジスタに書き込む。第二レジスタ値(Read data 2)もEX/MEMに書き込む

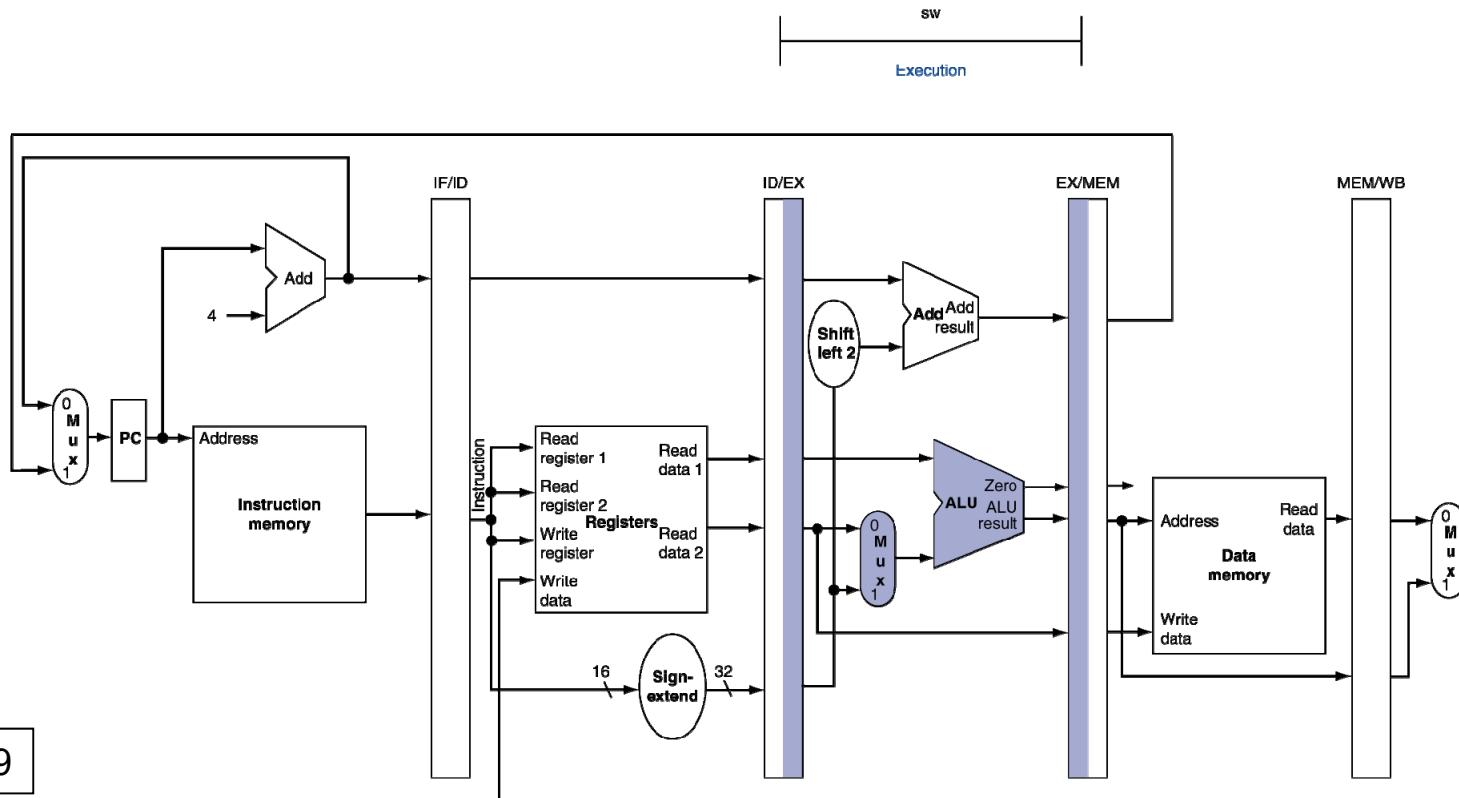


Figure 4.39

# A Pipelined Datapath (9)

## ■ MEMステージ(for sw)

### □ メモリアクセス

- EX/MEMパイプラインレジスタ内のアドレス値と第二レジスタ値(メモリ書き込み値)を使用してメモリ書き込みを行う。

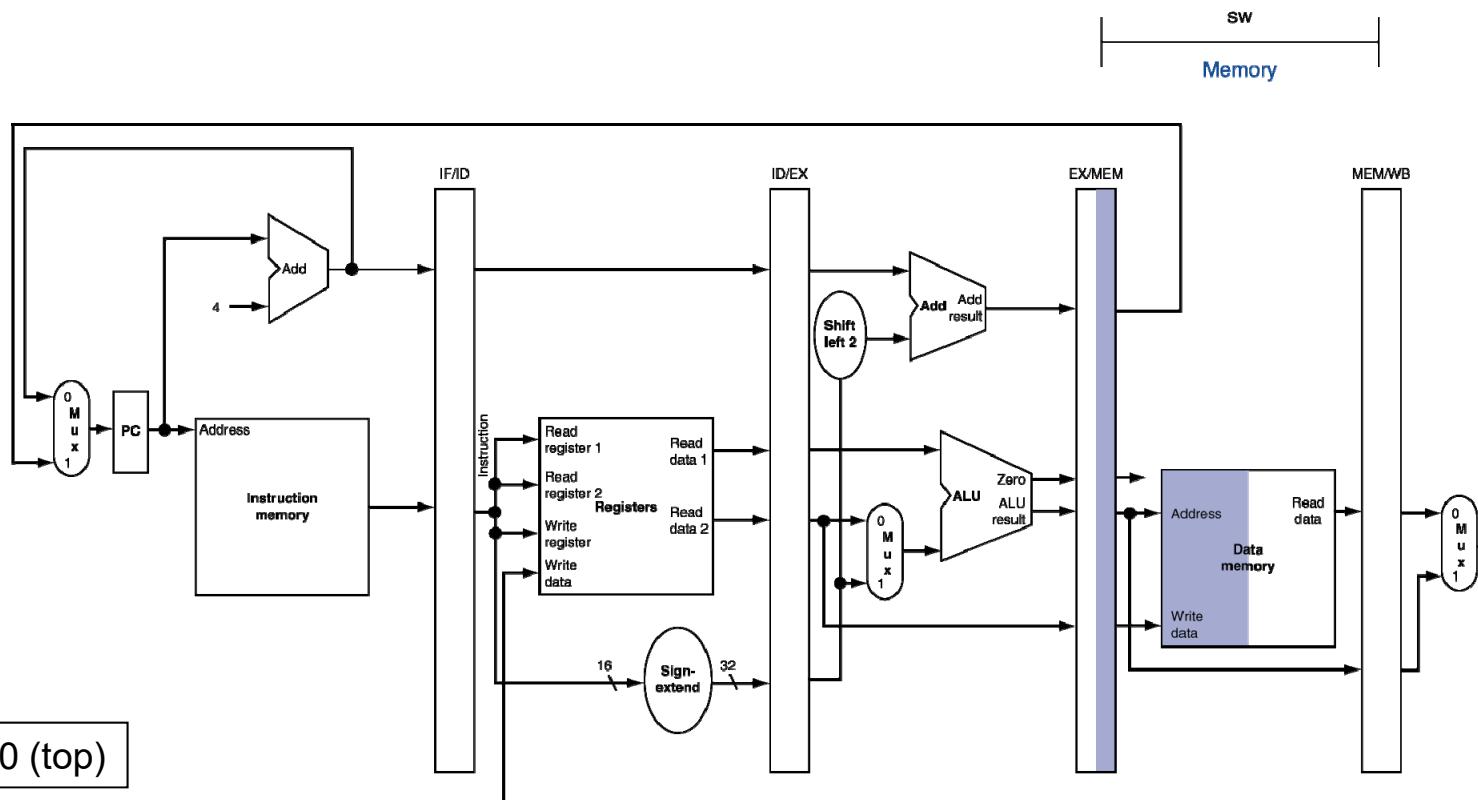


Figure 4.40 (top)

# A Pipelined Datapath (10)

## ■ WBステージ(for sw)

□ 何もしない

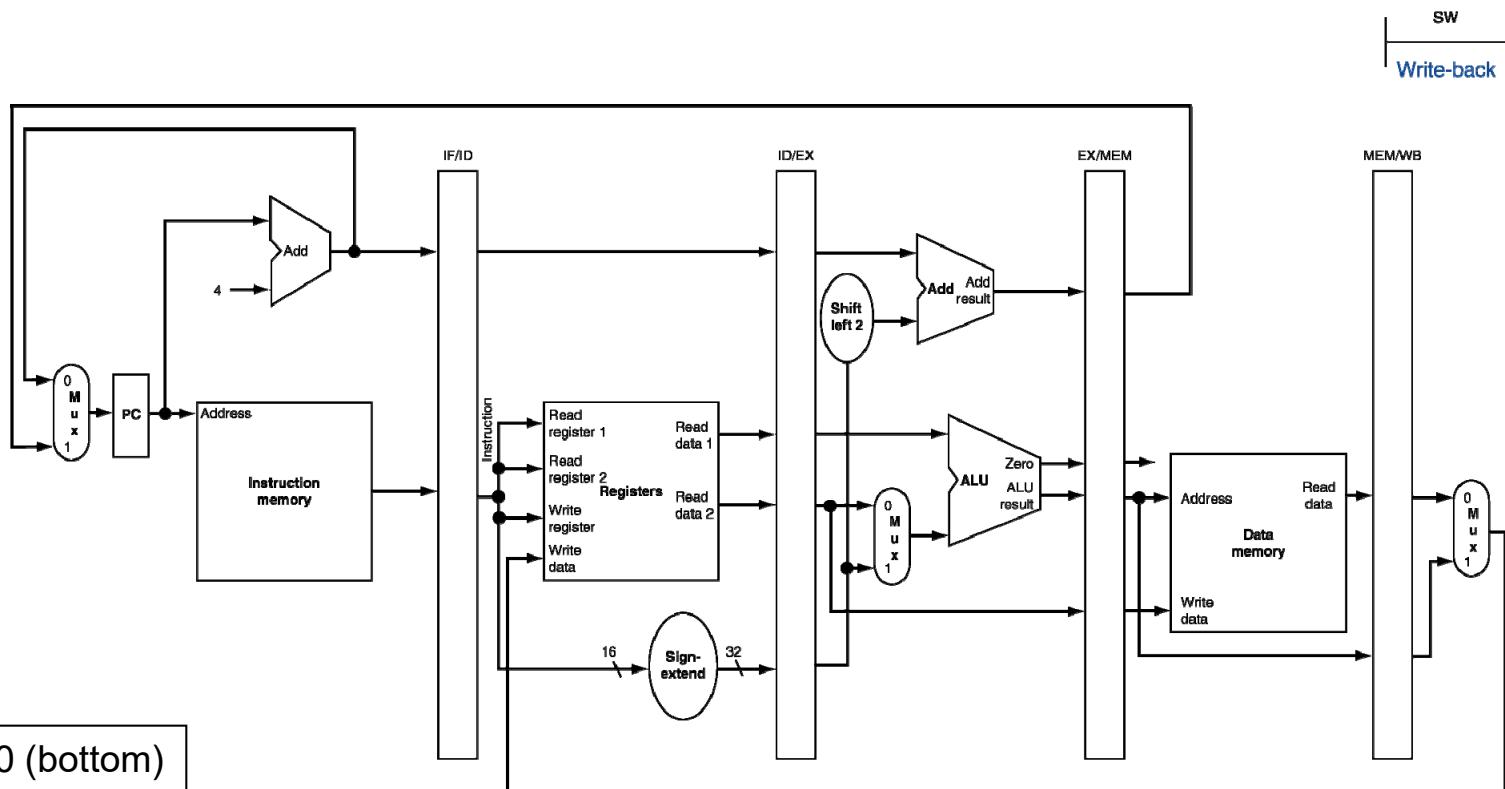


Figure 4.40 (bottom)

# A Pipelined Datapath (11)

- レジスタファイルを2つのステージで共用
  - コントロール／データを適切なステージから与える必要あり

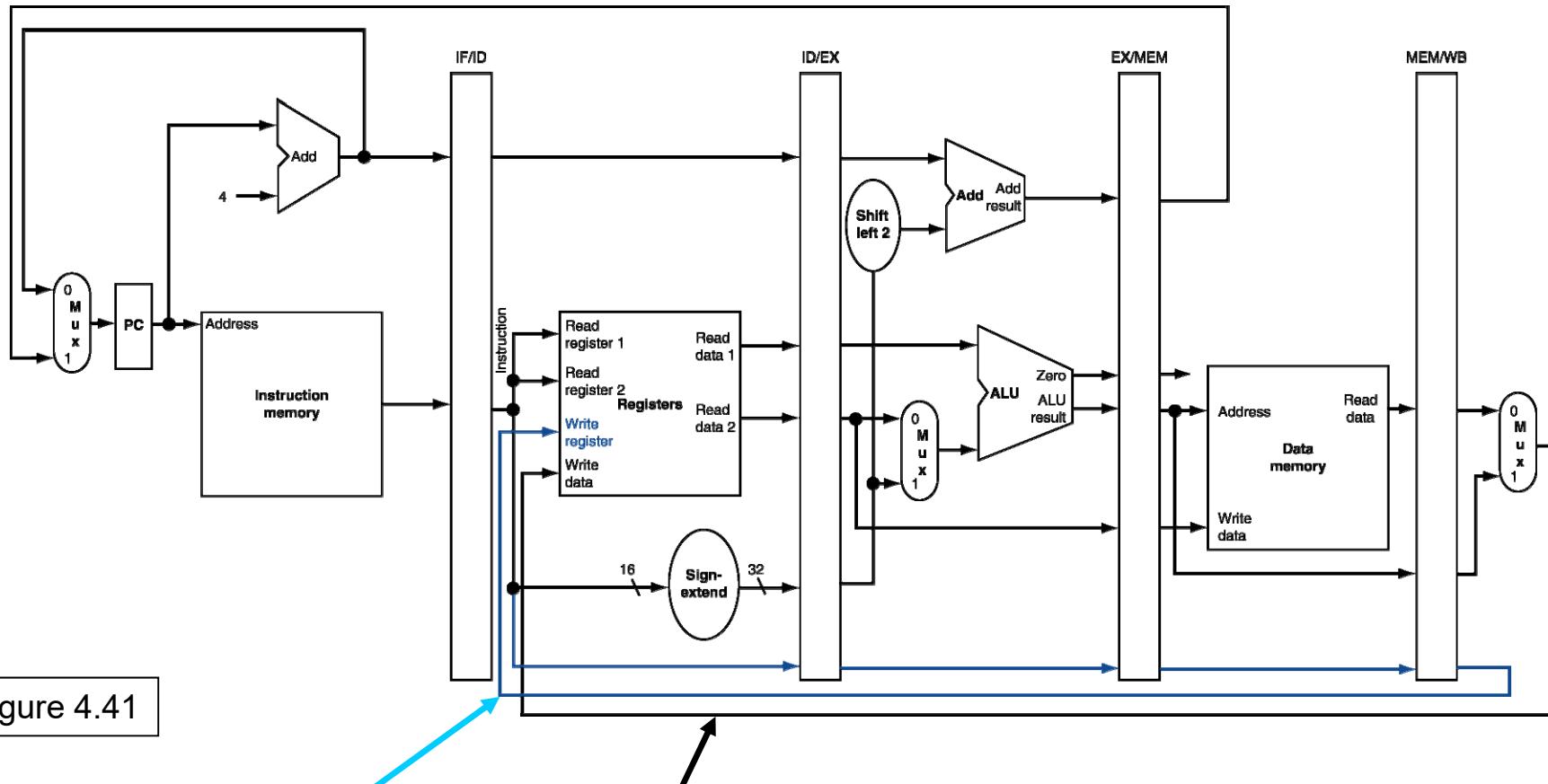


Figure 4.41

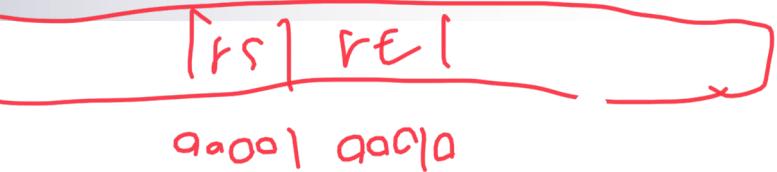
書き込みレジスタ番号、書き込み値は WBステージから与える必要がある  
(前ページまでの接続は間違い)

# Pipelined Control (1)

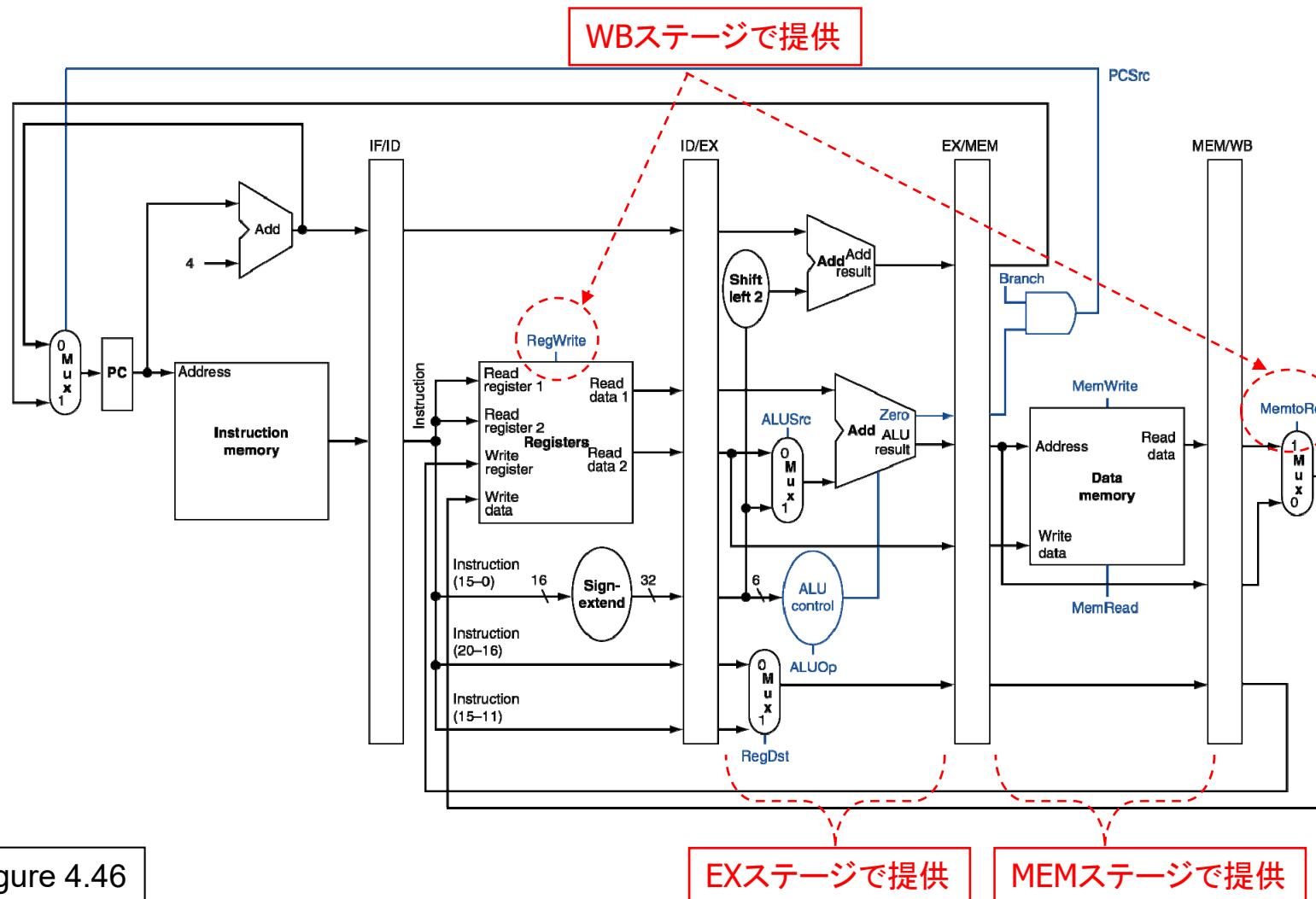
- パイプラインのコントロール(制御信号)は、各ステージから与える
  - 、 1) IFステージ(命令フェッチ)  
制御信号なし(全ての命令で同一動作)
  - 、 2) IDステージ(命令デコード & レジスタリード)  
制御信号なし(全ての命令で同一動作)
  - 、 3) EXステージ(実行／アドレス計算)  
RegDst, ALUOp, ALUSrc を与える
  - 4) MEMステージ(データメモリアクセス)  
MemRead, MemWrite, Branch (PCSrc) を与える
  - 5) WBステージ(レジスタ書き込み)  
MemtoReg, RegWrite を与える

add \$r3 \$r2, \$r1  
00011  
R [rs | rt | rd]

lw \$r2, 32(\$1)



## Pipelined Control (2)

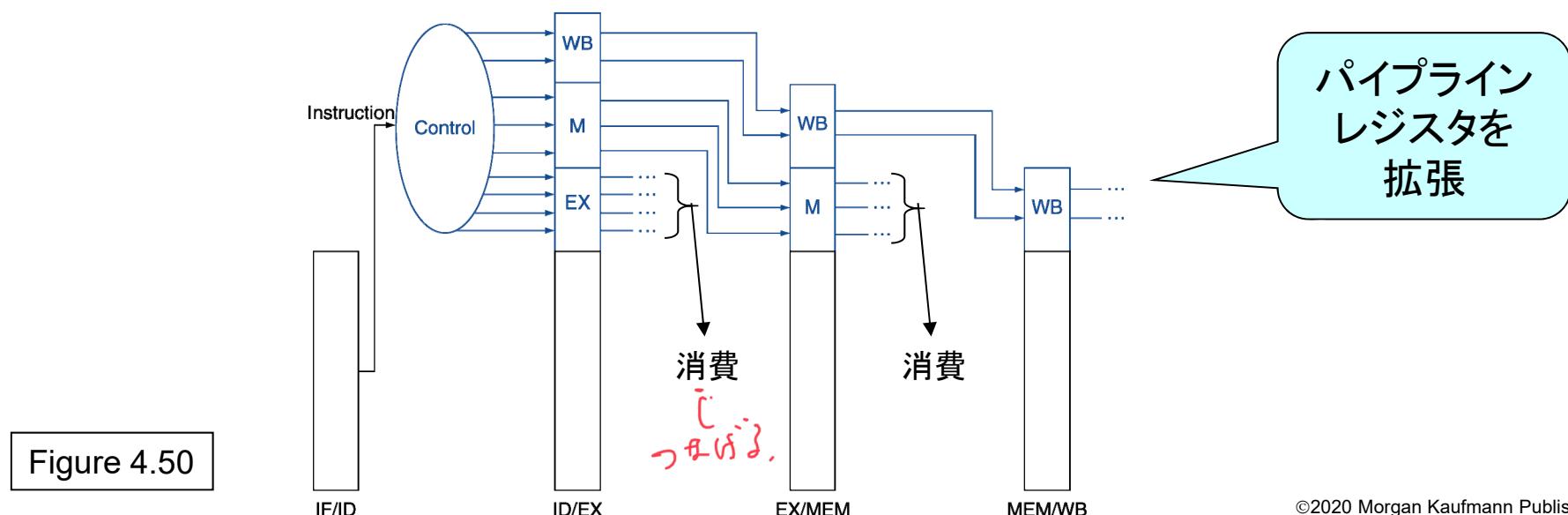


# Pipelined Control (3)

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0		0	1	0	1	1
sw	X	0	0		0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Skip

ここで戦略: 必要な制御信号を全てIDステージで生成しておく。  
あとは必要となるステージまで単純にコピーしていく。



# Pipelined Control (4)

はんて  
むじつけ

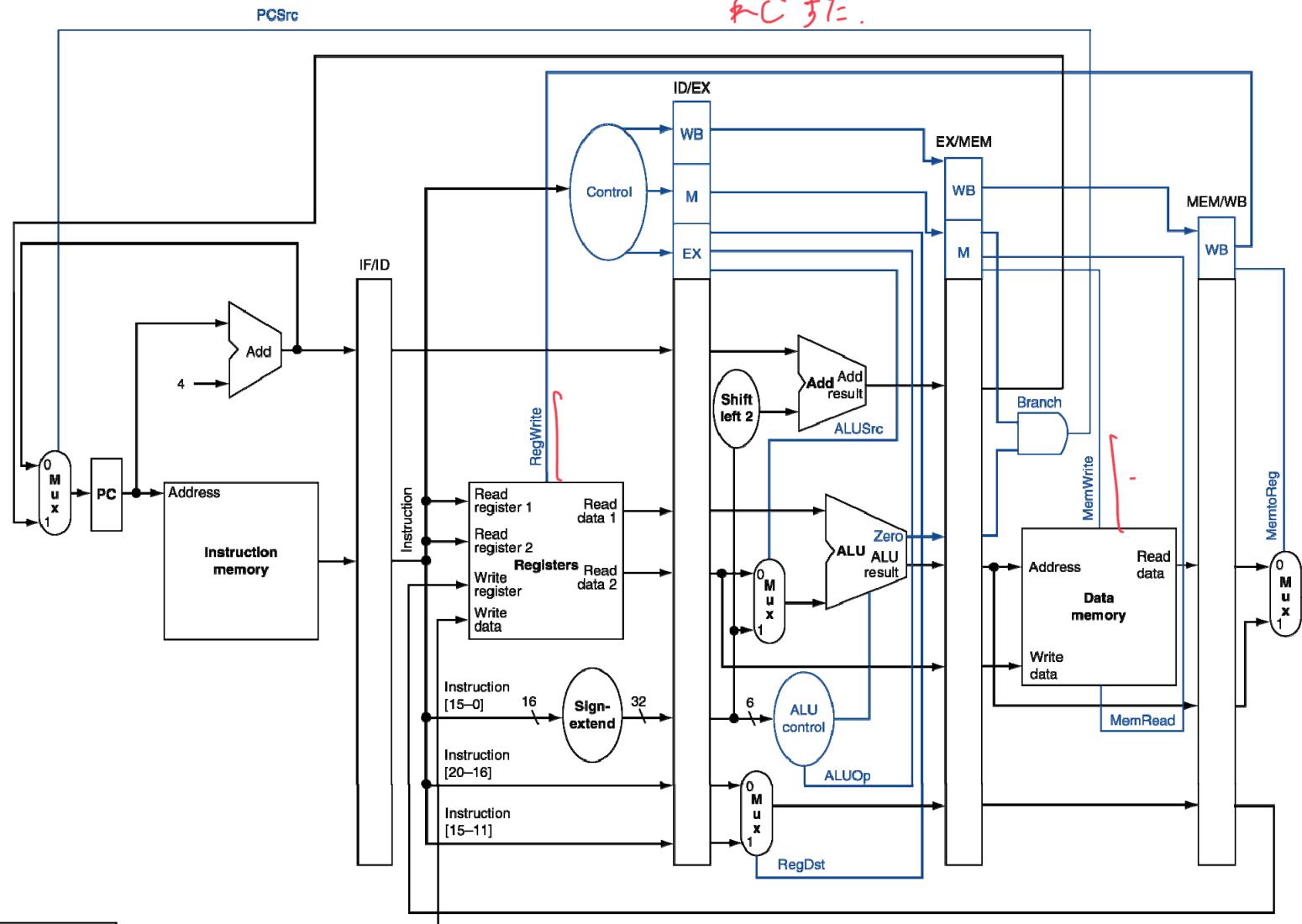


Figure 4.51

# Data Hazards and Forwarding (1)

sub \$2, \$1, -\$3  
 and \$12, \$2, \$5  
 or \$13, \$6, \$2  
 add \$14, \$2, \$2  
 SW \$15, 100(\$2)

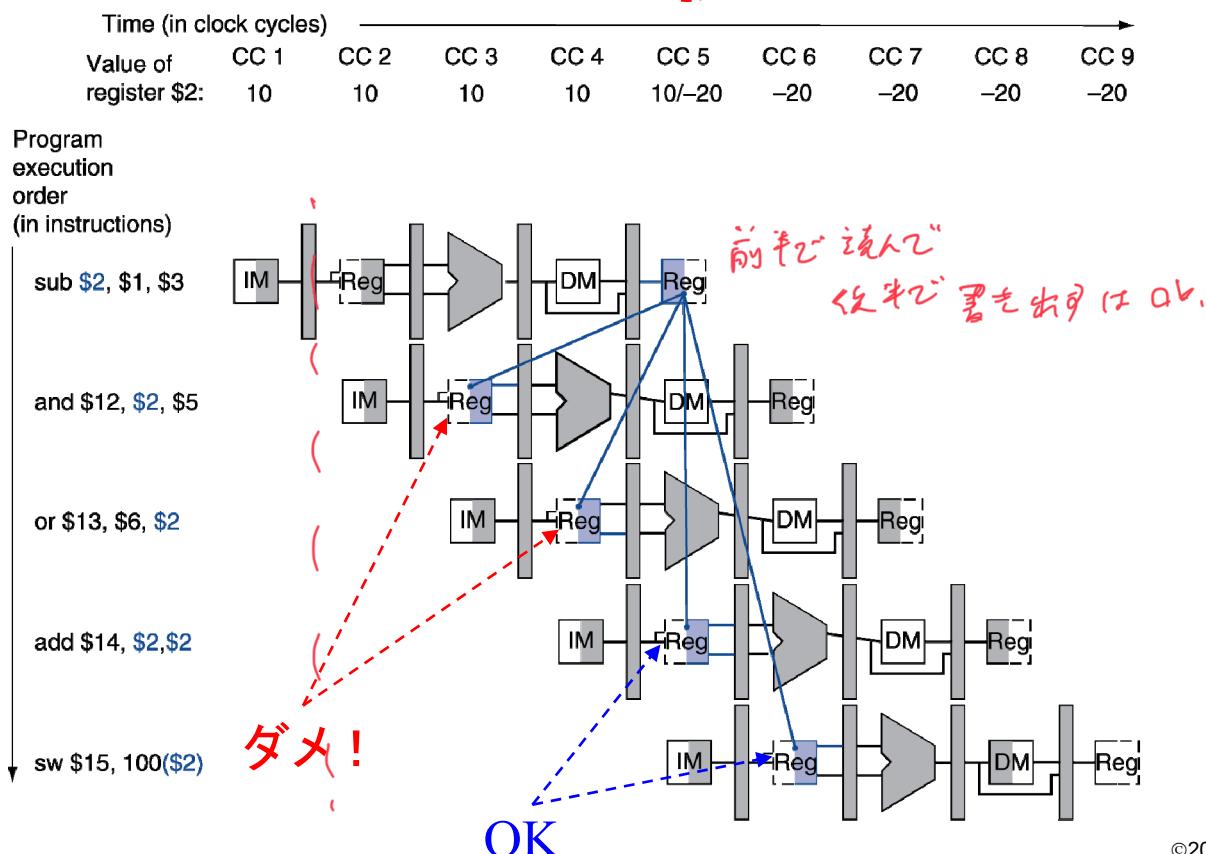
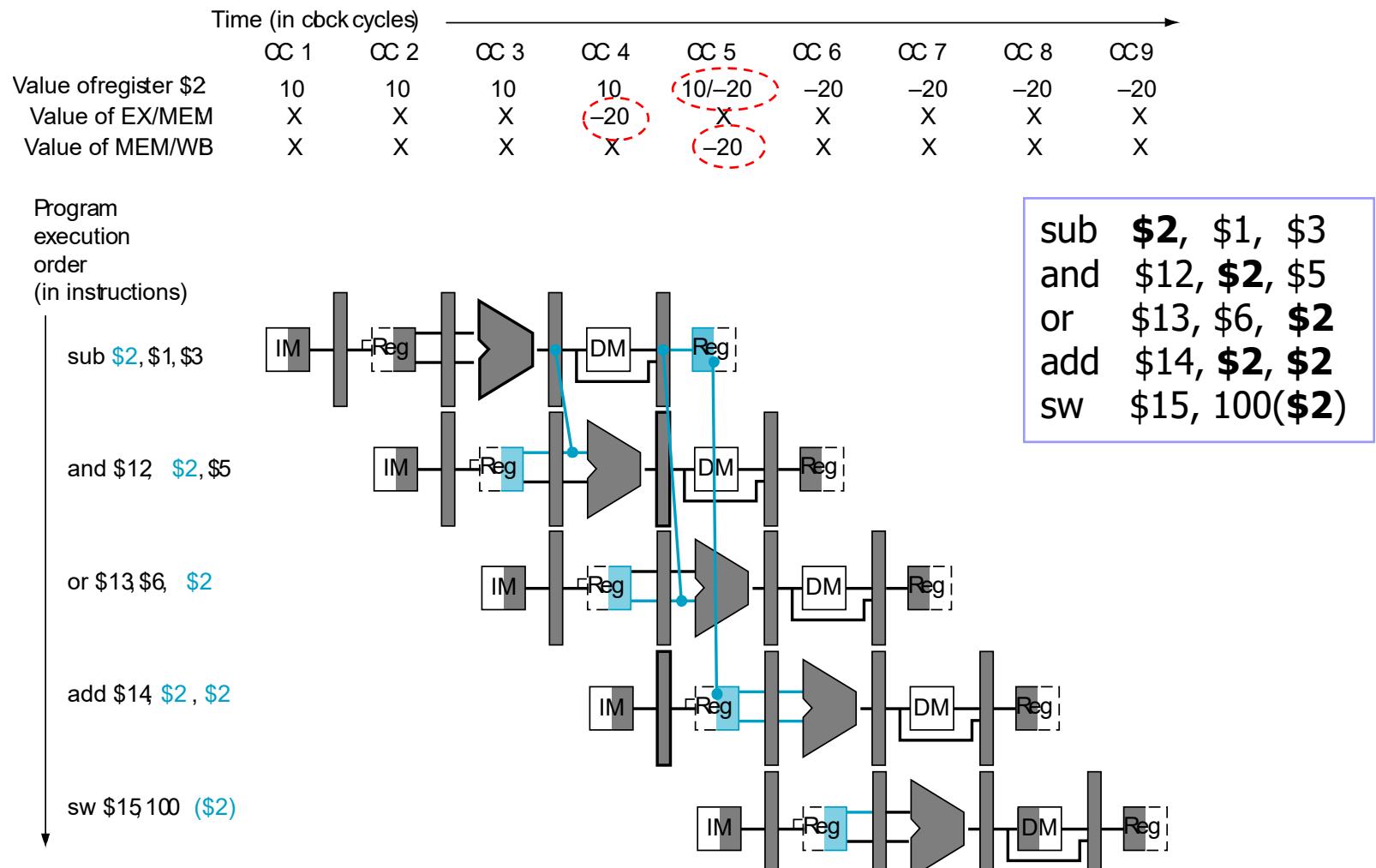


Figure 4.52

# Data Hazards and Forwarding (2)

## ■ パイプラインレジスタを利用してフォワーディング



# Data Hazards and Forwarding (3)

フォワーディングのために付加されるデータパス  
(即値オペランドはとりあえず省略)

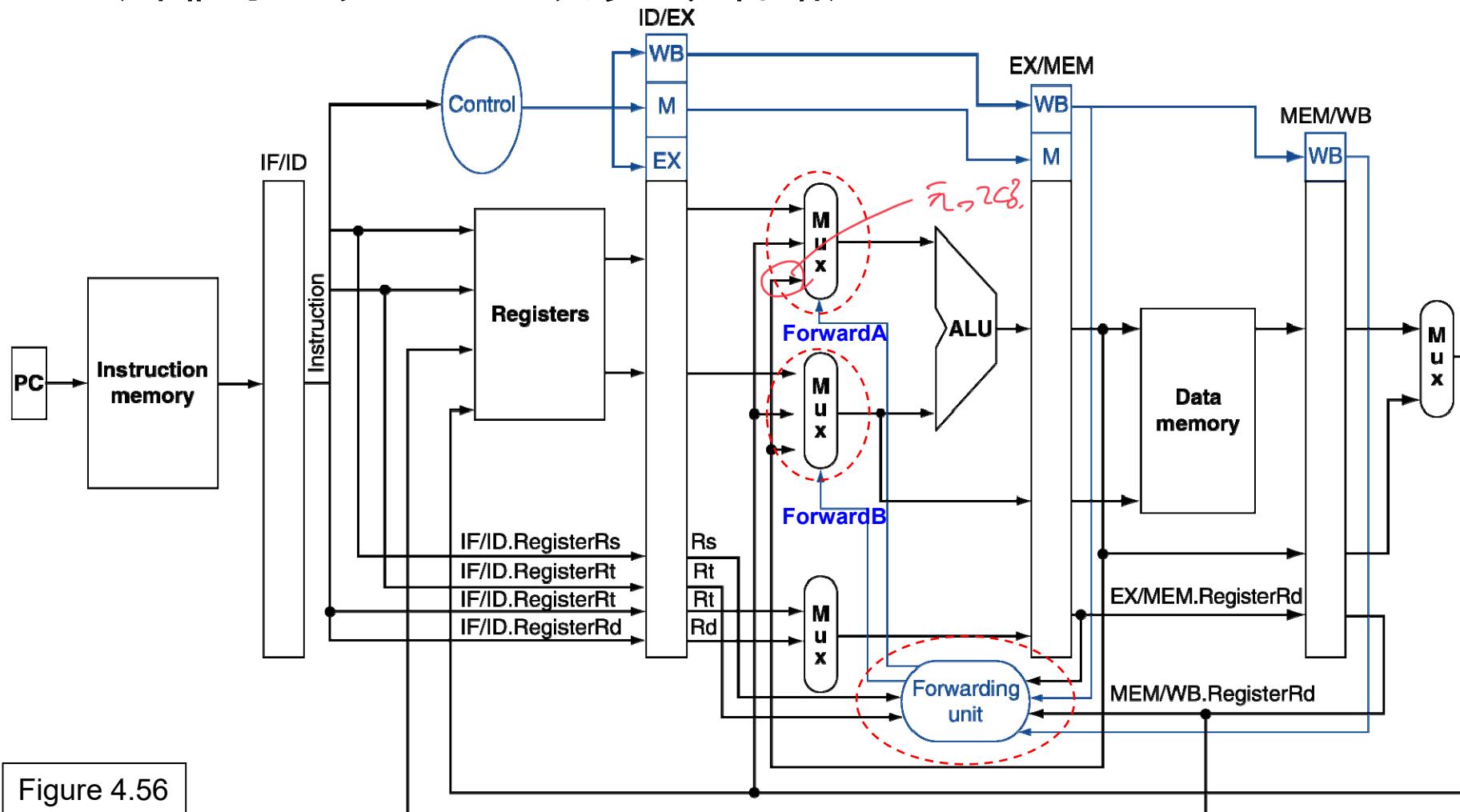


Figure 4.56

## ■ フォワーディングのためのマルチプレクサ制御値

Mux control	Source	Explanation
ForwardA=00	ID/EX	1 <sup>st</sup> ALU operand comes from register file
ForwardA=10	EX/MEM	1 <sup>st</sup> ALU operand is forwarded from the prior ALU result
ForwardA=01	MEM/WB	1 <sup>st</sup> ALU operand is forwarded from data memory or an earlier ALU result
ForwardB=00	ID/EX	2 <sup>nd</sup> ALU operand comes from register file
ForwardB=10	EX/MEM	2 <sup>nd</sup> ALU operand is forwarded from the prior ALU result
ForwardB=01	MEM/WB	2 <sup>nd</sup> ALU operand is forwarded from data memory or an earlier ALU result

Figure 4.55 The control values for the forwarding multiplexors.

# Data Hazards and Forwarding (5)

- ハザード検出とデータ供給の選択信号の生成
  - Rs側オペランドに関して(以下のForwardAによって選択)

```
/* EX hazard */  
if ( EX/MEM.RegWrite == '1' &&  
    EX/MEM.RegisterRd != 0 &&  
    EX/MEM.RegisterRd == ID/EX.RegisterRs )  
    ForwardA = "10"; /* from EX/MEM Reg. */
```

フォワーディング  
ユニット内の論理

```
/* MEM hazard */  
if ( MEM/WB.RegWrite == '1' &&  
    MEM/WB.RegisterRd != 0 &&  
    (EX/MEM.RegWrite = '0' || EX/MEM.RegisterRd != ID/EX.RegisterRs ) &&  
    MEM/WB.RegisterRd == ID/EX.RegisterRs )  
    ForwardA = "01"; /* from MEM/WB Reg. */
```

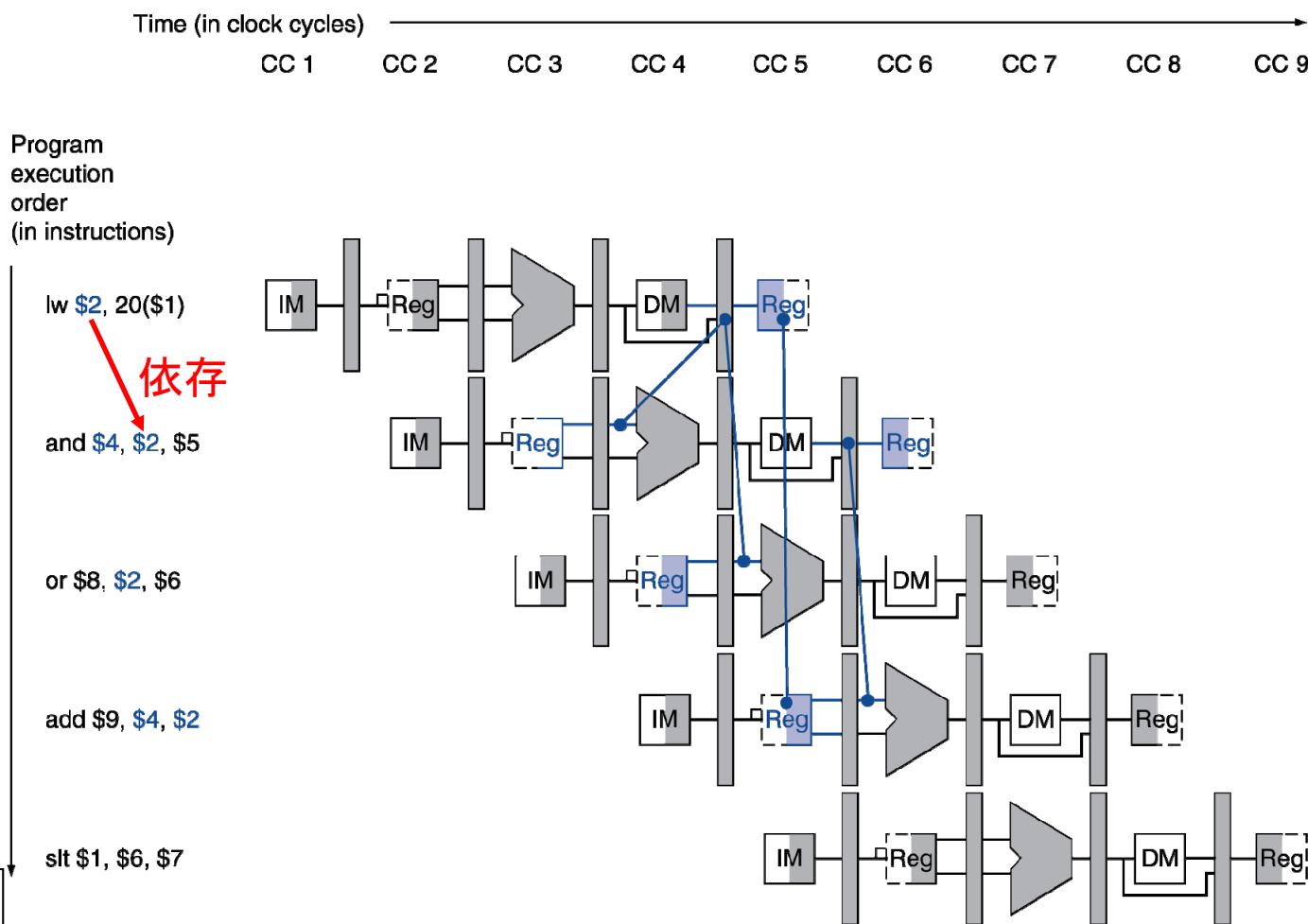
教科書と比較してみよう

multiplexorの選択信号としてForwardAが使用される

- Rt側オペランド(ForwardB)に関しても同様に検出

# Data Hazards and Stalls (1)

- 直前のロード命令(lw)にデータ依存している場合、フォワーディングによって解決不可能



# Data Hazards and Stalls (2)

- 対策: パイプラインをストールさせる(バブルの挿入)  
-回止まる。

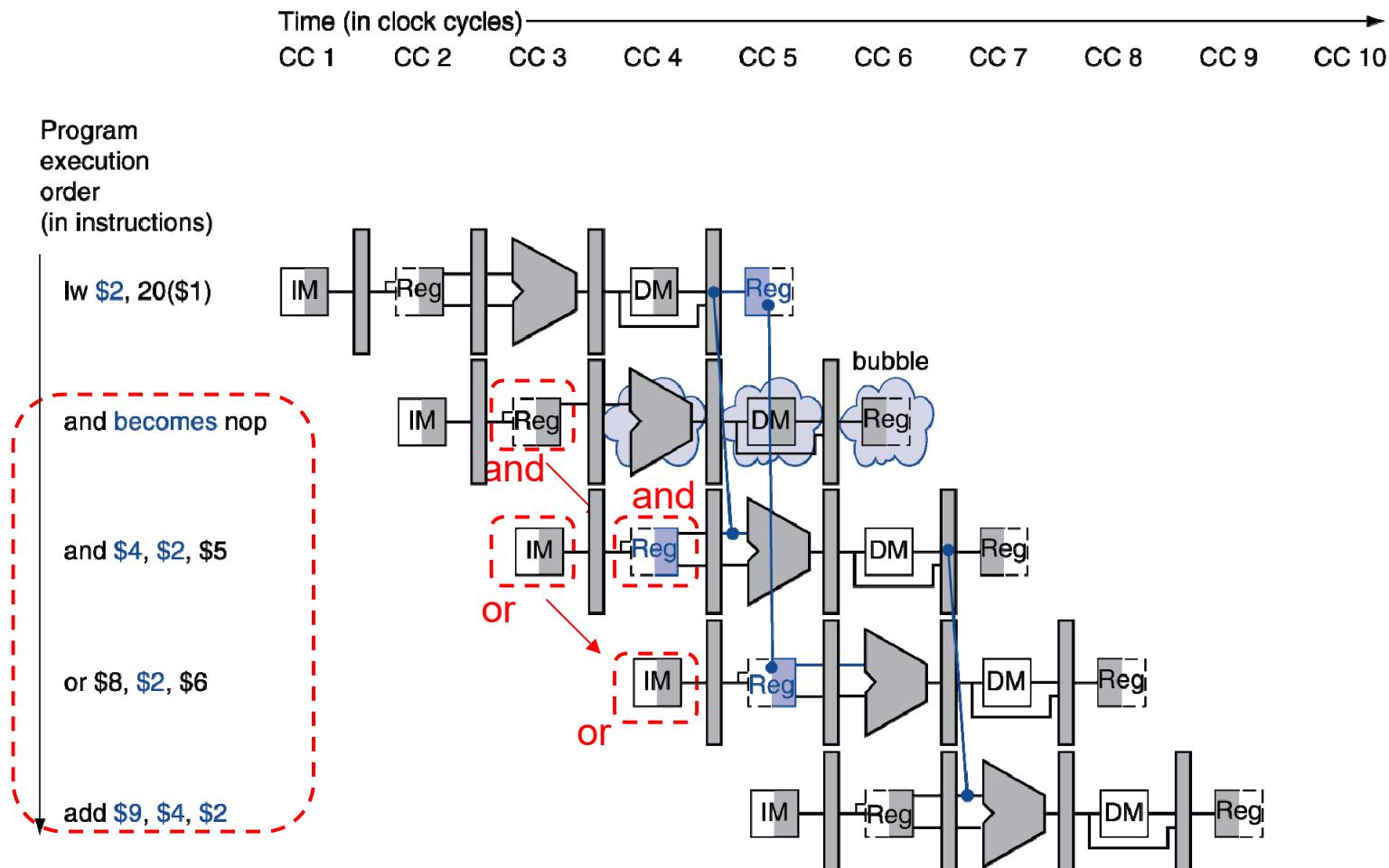


Figure 4.59

# Data Hazards and Stalls (3)

## ■ ハザード検出

- IDステージで検出

ハザード検出ユニット  
(Hazard detection unit) 内の論理

Skip

```
if ( ID/EX.MemRead == '1' &&
    ( ID/EX.RegisterRt == IF/ID.RegisterRs ||  

      ID/EX.RegisterRt == IF/ID.RegisterRt ) )
    stall the pipeline;
```

## ■ ストール方法(バブル(bubble)の挿入方法)

- PCとIF/IDパイプラインレジスタの書き込みを禁止
- ID/EXパイプラインレジスタのRegWrite、  
MemWriteフィールドを0にする(他はdon't care)  
⇒ nopの作用

# Data Hazards and Stalls (4)

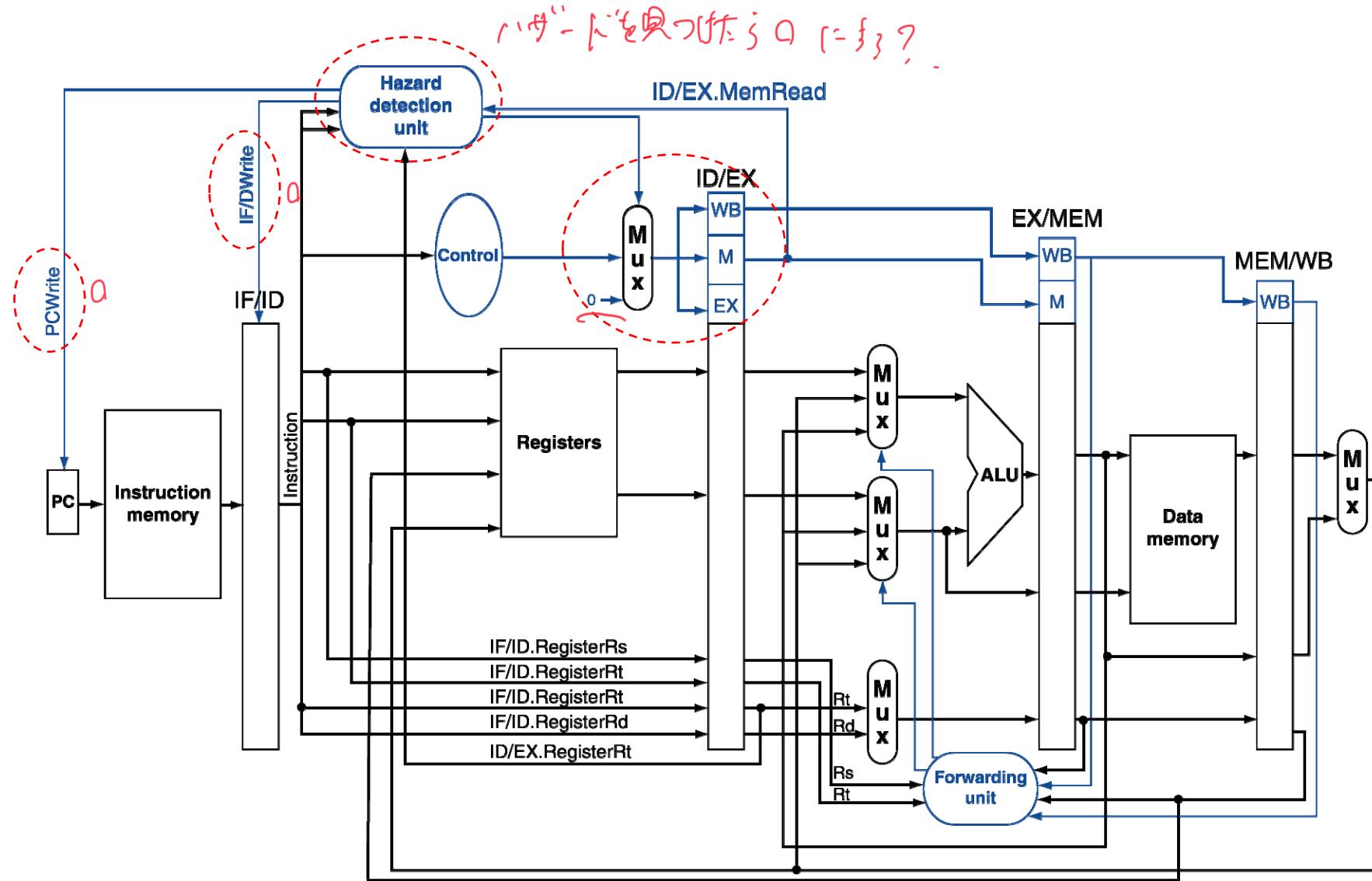
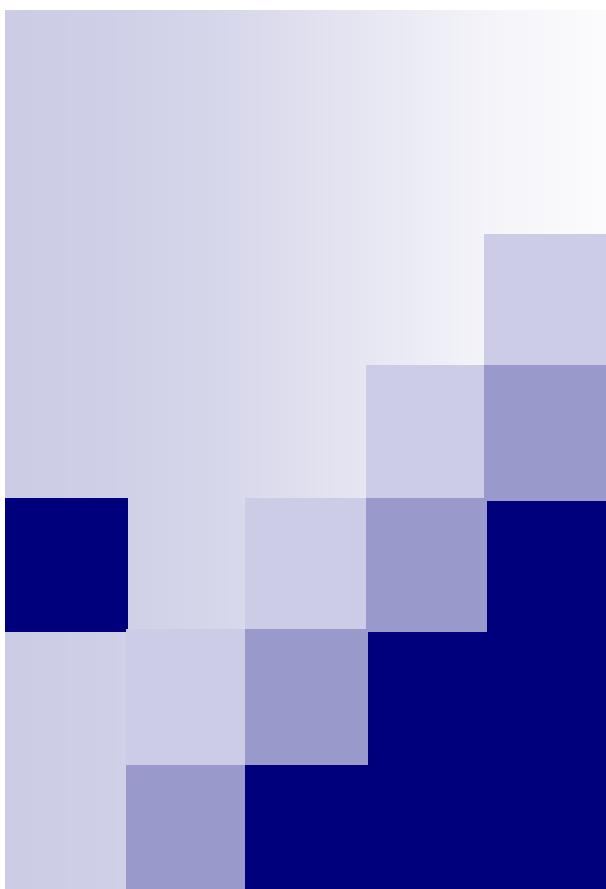


Figure 4.60



# I470F 統合アーキテクチャ

記憶階層(キャッシュ)

# Introduction (1)

## ■ メモリアクセスの局所性

- 時間的局所性／Temporal locality (locality in time)

- アクセスした命令／データは近い将来再びアクセスする
    - ループ内の命令／データなど

- 空間的局所性／Spatial locality (locality in space)

- アクセスした命令／データの近くにある命令／データは近い将来アクセスされる
    - 命令アクセス(sequential)／配列データアクセス

## ■ メモリシステムの階層化 (memory hierarchy)

- 複数の階層からなるメモリで、それぞれの階層で速度とサイズが異なる

- 速度／サイズを考慮した階層化

- 高スピード＆小サイズ ⇒ processor に近い場所 (**Cache** by SRAM)
  - 中間的なスピード＆中間的なサイズ ⇒ 主記憶(**Main memory** by DRAM)
  - 低スピード＆大サイズ ⇒ processor から遠い場所 (磁気Disk or SSDによる  
二次記憶)

# Introduction (2)

## ■ メモリ要素 — アクセス時間とコスト —

Memory technology	Typical access time	\$ per GB in 2020
SRAM	0.5 – 2.5 ns	\$500 – \$1,000
DRAM	50 – 70 ns	\$3 – \$6
Flash(SSD)	5,000-50,000 ns	\$0.06 – \$0.12
Magnetic disk	5 – 20 million ns	\$0.01 – \$0.02

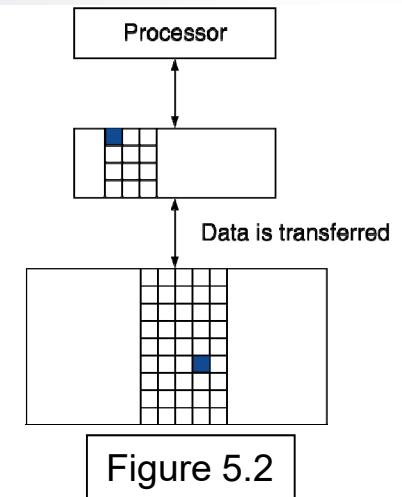
## ■ 目標

- 最も低価格なメモリ(磁気ディスク／SSD)が達成するサイズを提供, かつ同時に,
- 最も高速なメモリ(SRAM)が達成するアクセス時間を提供

# Introduction (3)

## ■ Cacheとは

- Processor(CPU) の近くに備えられるメモリ
  - 高速だが、サイズは大きくない
- より下の階層のメモリの内容の一部のコピーを持つ
- 2つのキャッシュ：命令キャッシュとデータキャッシュ
- **block** (16~128バイト) 単位で命令／データを格納する (**line** とも呼ばれる) ← 空間的局所性を考慮
- アクセスしたい命令／データを含む block がキャッシュ内に存在した場合は **hit**, 存在しなければ **miss (miss-hit)**
- 全てのメモリアクセス中、キャッシュに hit した割合を **hit rate (hit ratio)**, miss した割合を **miss rate** ( $= 1 - \text{hit rate}$ )

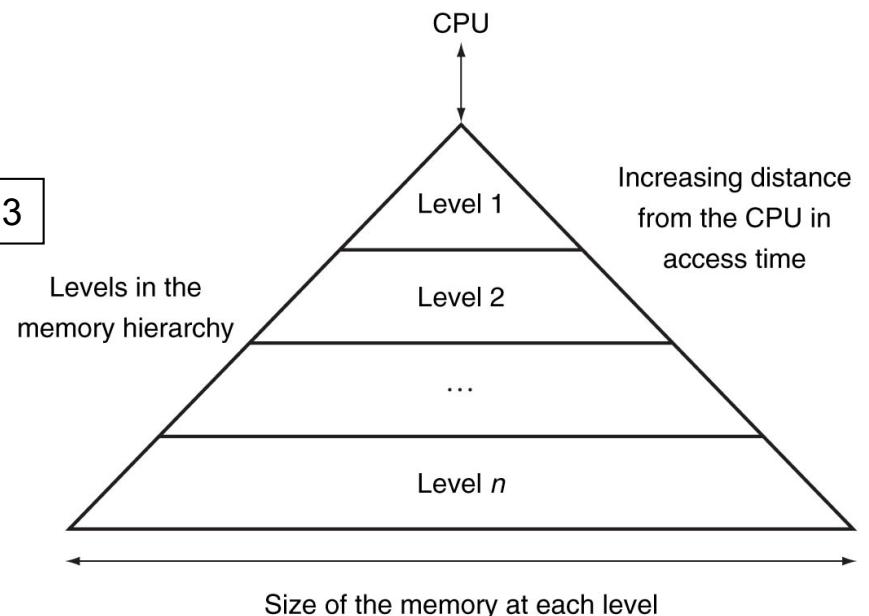


# Introduction (4)

## ■ Cacheとは(つづき)

- hit の場合のアクセス時間を **hit time**
- miss の場合の次階層のメモリアクセス時間 + ブロック移動時間 を **miss penalty** と呼ぶ
- **Inclusion property**
  - 階層  $i$  に含まれる block は、必ず階層  $i + 1$  に含まれる

Figure 5.3



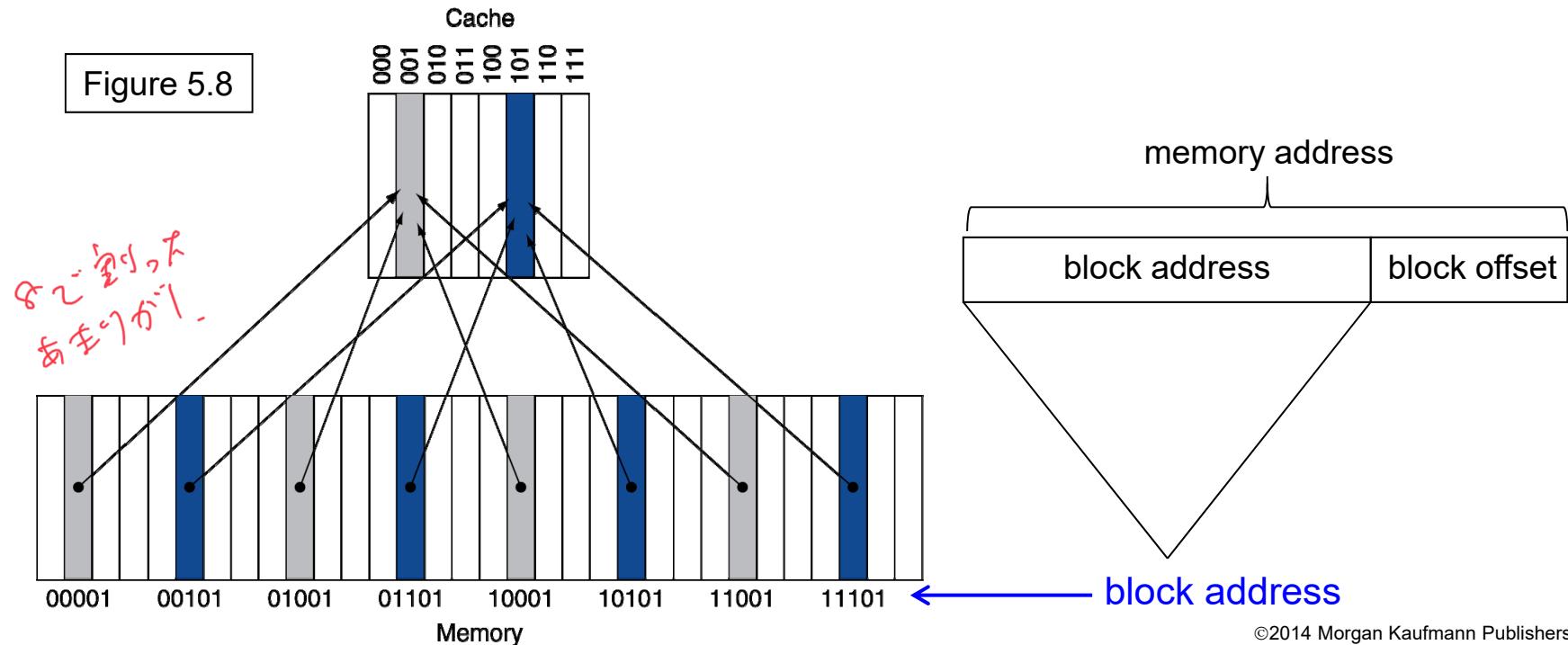
注) Exclusion property  
方式もある。この場合、  
階層間でblockは排他的

# The Basics of Caches (1)

- キャッシュの中に当該blockが存在するかどうかをどのようにして見つけるか？
- Direct-mapped cacheの場合

**(block address) % (the number of block entries in the cache)**

⇒ メモリアクセスのアドレスによって、cache内の場所が一意に決定



# The Basics of Caches (2)

- cache内の場所はわかったが、そこに存在するデータが当該メモリアドレスのものかどうかどうやって判断するか？
- アドレス・タグを block 毎に用意
  - (block address) / (the number of block entries in the cache)
    - ⇒ 当該メモリアドレスの上位アドレスとエントリのタグを比較し、一致すれば hit
      - ・ただし、タグの初期状態が偶然上位アドレスと一致する可能性あり
      - ・その block に有効なデータがない場合を考える必要 → 有効ビットを用意
  - 有効ビット (valid bit) を用意
    - 当該ブロックが有効なデータを持つかどうかを示す
    - タグが一致 & 有効ビットの値が1のときのみ hit とみなす
  - miss した場合
    - 下の階層のメモリからデータを読んできてエントリを fill する
    - 当該エントリに他のアドレスのデータが存在した場合はそれを追い出す(リプレース)(もちろん、タグも更新される)

# The Basics of Caches (3)

- 例: 初期状態から、以下の4つのメモリアクセスが生じた場合  
(ブロックサイズ = 16 bytes, キャッシュサイズ = 128 bytes と仮定)

	address	hit or miss	assigned entry
1.	010110 0000	miss	110
2.	011010 1100	miss	010
3.	010110 1000	hit	110
4.	001010 0100	miss	010 (replace!)

After 1.

Index	Valid	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
<b>110</b>	<b>1</b>	<b>010</b>	<b>block(010110)</b>
111	0		

After 2.

Index	Valid	Tag	Data
000	0		
001	0		
<b>010</b>	<b>1</b>	<b>011</b>	<b>block(011010)</b>
011	0		
100	0		
101	0		
<b>110</b>	<b>1</b>	<b>010</b>	<b>block(010110)</b>
111	0		

replace

After 4.

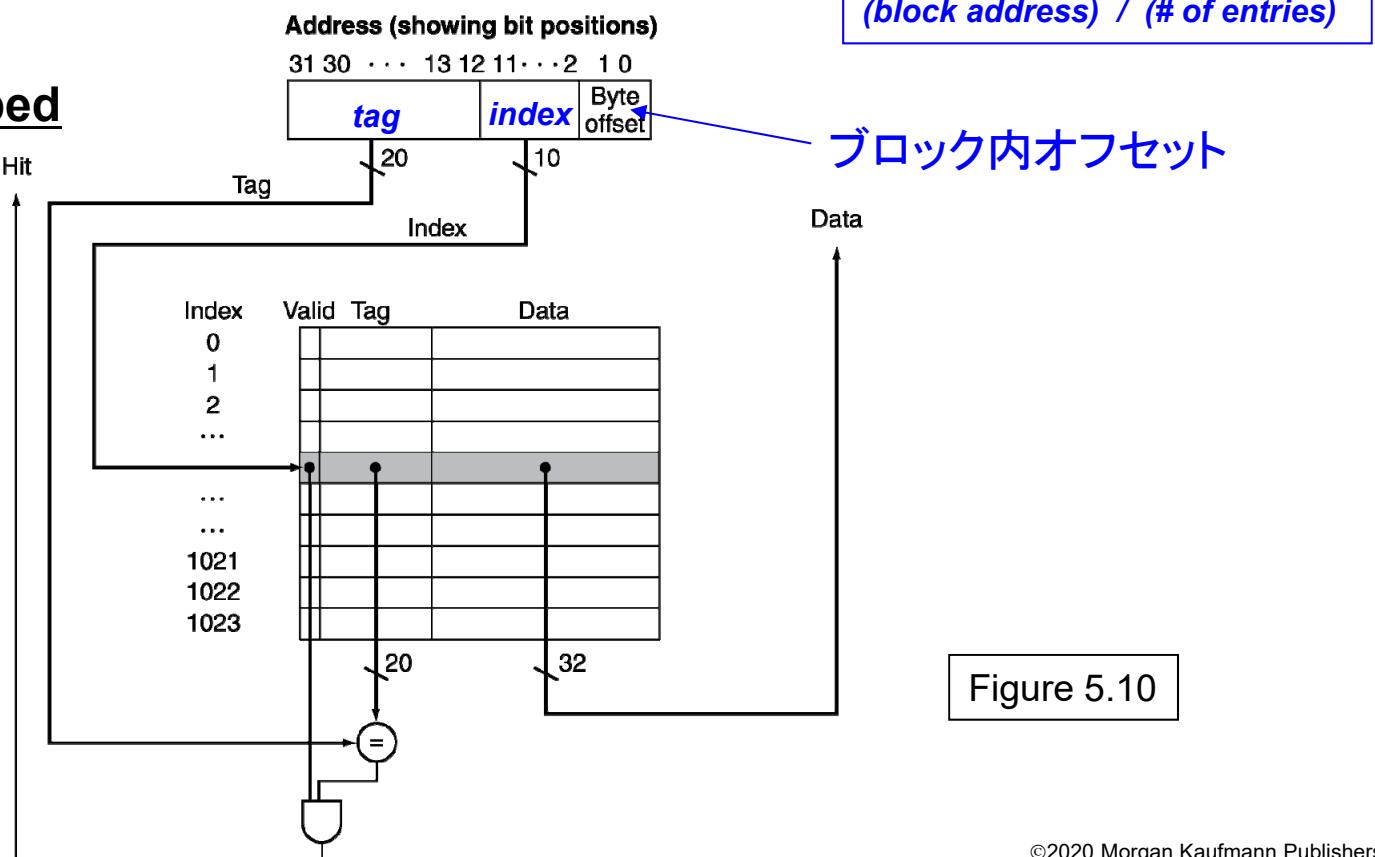
Index	Valid	Tag	Data
000	0		
001	0		
<b>010</b>	<b>1</b>	<b>001</b>	<b>block(001010)</b>
011	0		
100	0		
101	0		
<b>110</b>	<b>1</b>	<b>010</b>	<b>block(010110)</b>
111	0		

# The Basics of Caches (4)

- block size = 4 byte (1 word), エントリ数1024 の場合の hit 判定の方法

- メモリアドレスの中位10ビット(indexフィールド)で エントリの選択
- メモリアドレスの上位20ビット(tagフィールド)を当該エントリのタグと比較

Direct mapped



# The Basics of Caches (5)

- 空間的局所性を活用するためには
  - block の multiword化 (4~32 words = 16~128 bytes)
  - ただし, block size を大きくし過ぎると, 衝突で追い出される可能性大
  - miss penaltyとのトレードオフも重要

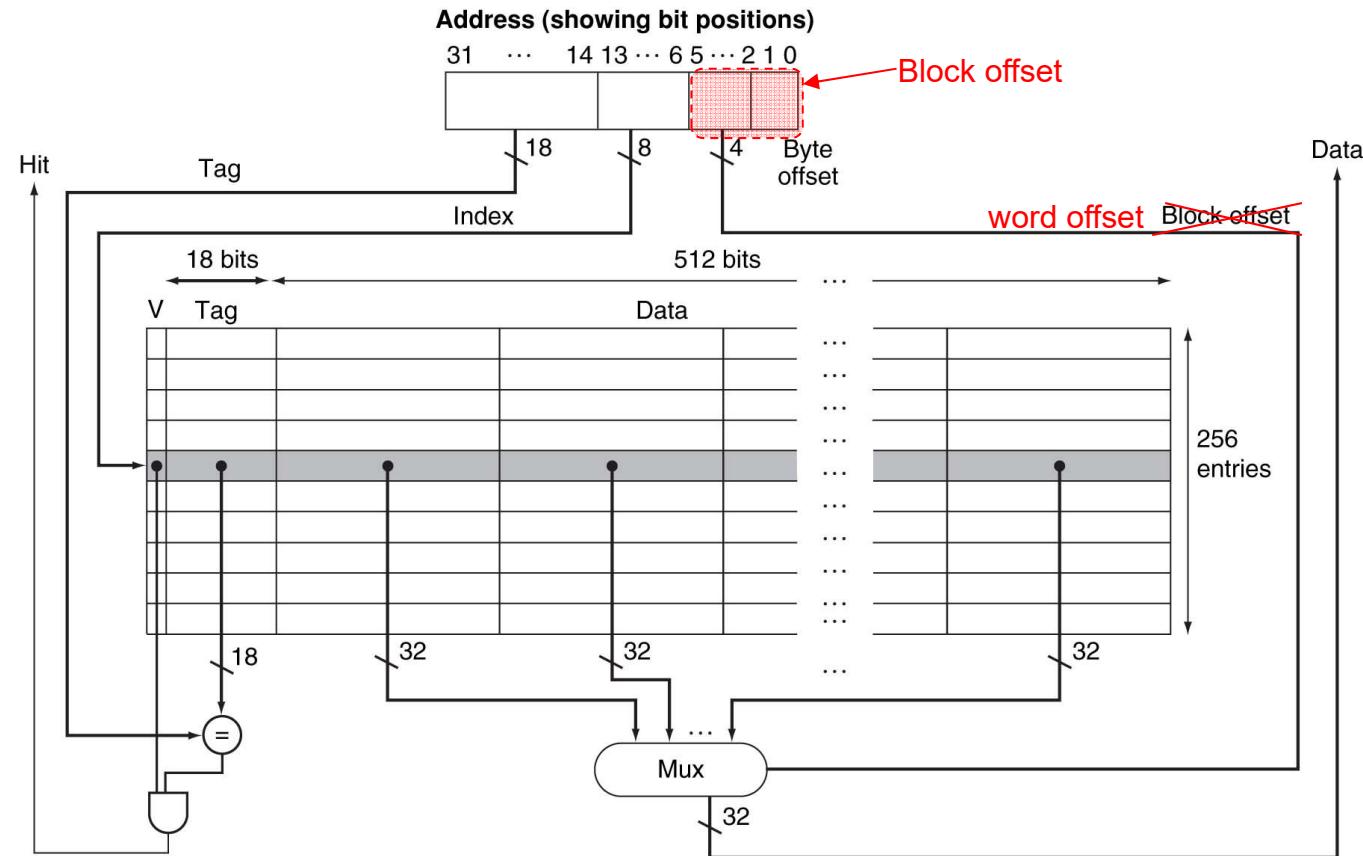


Figure 5.12

# Measuring and Improving Cache Performance

- Cacheのパフォーマンスに影響する2つの要因
  - miss ratio
  - miss penalty
- パフォーマンス向上のための2つの方法
  - **More flexible placement of blocks**
    - 競合を減らすことでミス率を削減
  - **Multilevel caches**
    - メモリ階層を増やすことにより、ミスした場合のペナルティをなるべく小さくする
      - 主記憶よりもアクセス時間が短い2次キャッシュを導入すると、1次キャッシュミス時のペナルティが短くなる

# More Flexible Placement of Blocks (1)

- Direct-mapped cache

*(block address) % (number of cache blocks in the cache)*

- Set-associative cache

*(block address) % (number of sets in the cache)*

→ セットを指定(セット内のいずれかのエントリを使用する)

- Fully-associative cache

anywhere in the cache

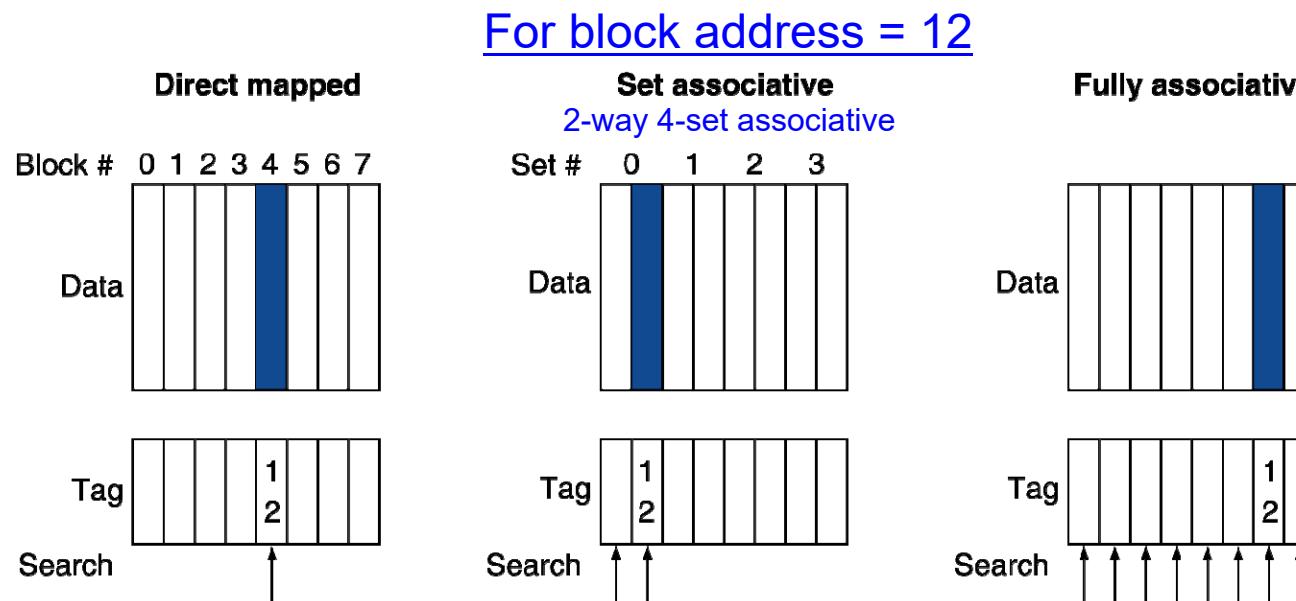


Figure 5.14

# More Flexible Placement of Blocks (2)

- block entry が 4つの cache において、初期状態から block address = 0, 8, 0, 6, 8の順でメモリアクセスが生じた場合

Direct-mapped

Block address	Hit/Miss	Contents of cache blocks			
		0	1	2	3
0000	Miss	Mem[0]			
1000	Miss	Mem[8]			
0000	Miss	Mem[0]			
0110	Miss	Mem[0]		Mem[6]	
1000	Miss	Mem[8]		Mem[6]	

2-way set-associative

Block address	Hit/Miss	Contents of cache blocks			
		Set 0		Set 1	
0000	Miss	Mem[0]			
1000	Miss	Mem[0]	Mem[8]		
0000	Hit	Mem[0]	Mem[8]		
0110	Miss	Mem[0]	Mem[6]		
1000	Miss	Mem[8]	Mem[6]		

Fully-associative

Block address	Hit/Miss	Contents of cache blocks			
		Block 0	Block 1	Block 2	Block 3
0000	Miss	Mem[0]			
1000	Miss	Mem[0]	Mem[8]		
0000	Hit	Mem[0]	Mem[8]		
0110	Miss	Mem[0]	Mem[8]	Mem[6]	
1000	Hit	Mem[0]	Mem[8]	Mem[6]	

LRU  
replacement  
(later)

# More Flexible Placement of Blocks (3)

- 連想度によってどのくらいミス率が小さくなるか？
  - 1ウェイから2ウェイにすると、大きな改善
  - 更に連想度を上げても改善率は小さい

SPEC2000 CPUの10個のプログラム実行におけるミス率

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

Steep !

Dull ...

# More Flexible Placement of Blocks (4)

## ■ アドレス内の3つのフィールド

- タグ・フィールド
- インデックス・フィールド
- ブロック内オフセット

同じキャッシュサイズにおいて、連想度が倍になると、インデックスフィールドは1ビット小さくなり、タグフィールドは1ビット大きくなる。

(フルアソシアティブの場合、インデックスフィールドはない。)

②

セット内のブロックの選択に使用。セット内の全タグと比較する。

①

キャッシュ内のセットの選択に使用

③

ブロック内のバイト、ワード等の選択に使用

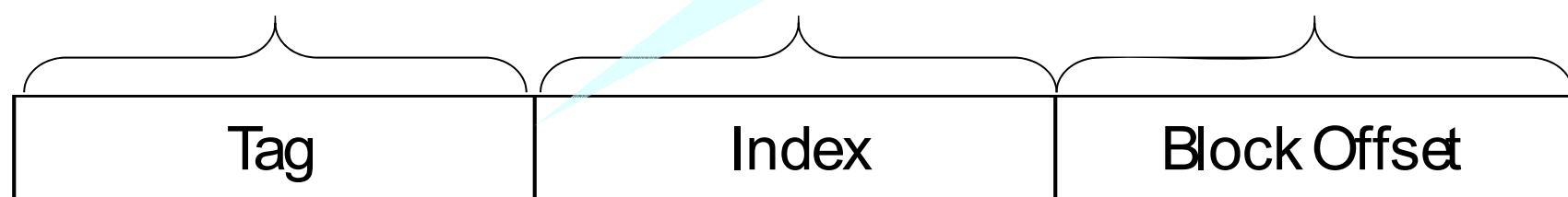
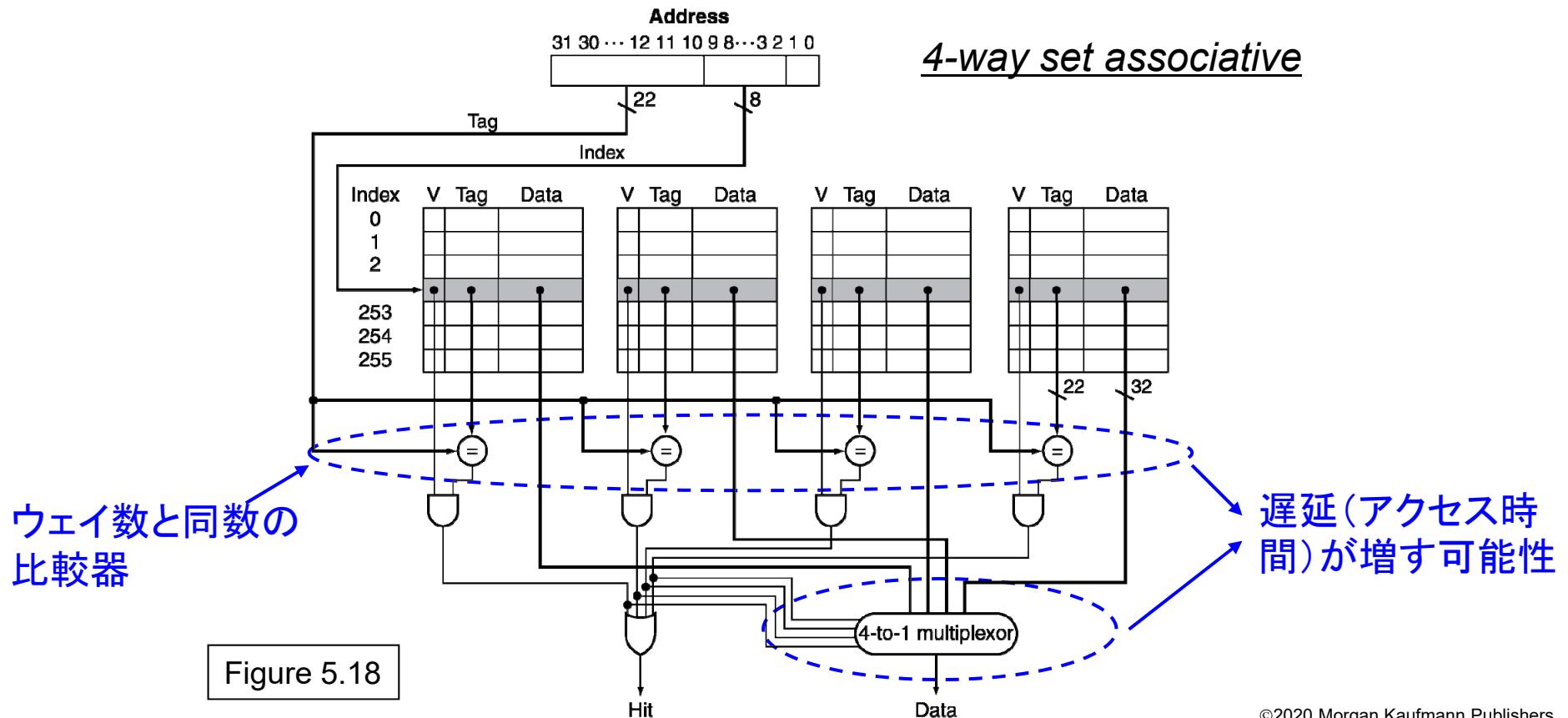


Figure 5.17

# More Flexible Placement of Blocks (5)

- ミスペナルティ、アクセス時間、ハードウェア量の兼ね合いを考慮して連想度を選択
- block size = 4 byte (1 word)の場合のヒット判定
  - 高速なヒット判定のために全wayを並列(同時)に比較



# More Flexible Placement of Blocks (6)

## ■ Replacement(置換)方式

- miss時に、set内の全てのblockがvalidの場合、どのblockを取り除く(追い出す)かを決定する必要
- $n$ -way set associativeにおいて、 $n$ 通りの候補

### Random

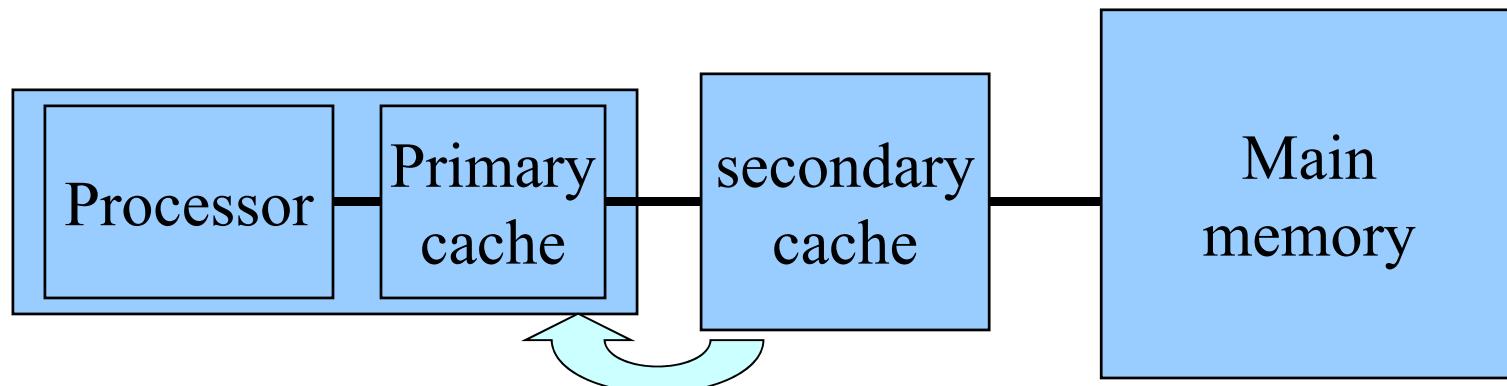
- ランダムに選択

### Least recently used (LRU)

- 最も長時間使用されなかったblockを選択
- 例) block entry が 4つ、4-way set associative(すなわちfully-associative) cacheにおいて、block address = 0, 8, 3, 2, 0, 4の順でメモリアクセスが生じた場合、最後のアクセスで block address=8 の blockが追い出される

# Multilevel Cache (1)

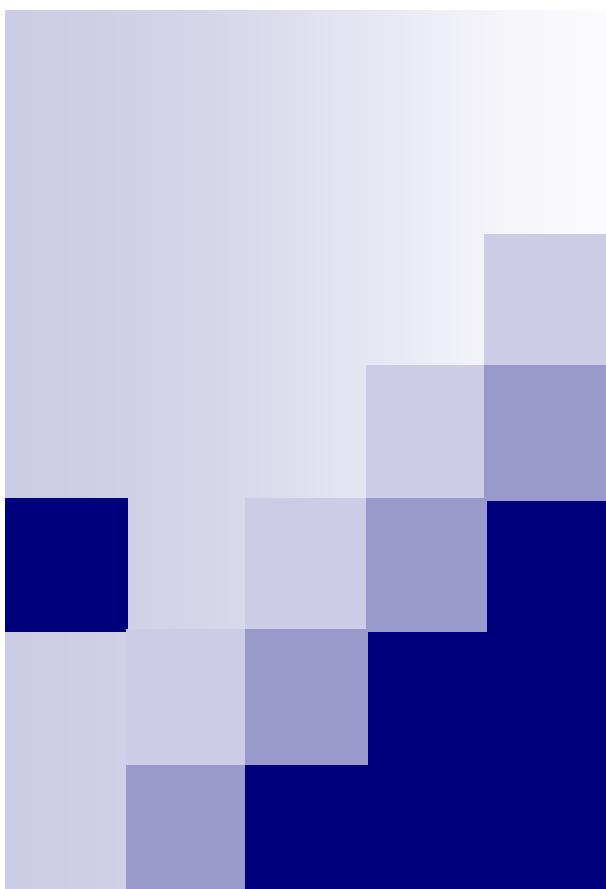
- Primary (First level/L1/1次) cache はプロセッサと同一チップ内に実装される → 高速アクセス
- Primary cacheとメインメモリはアクセス時間の差が大
- 中間的なアクセス時間およびサイズのメモリ(通常はSRAM)を primary cacheとメインメモリの間に挿入
  - Secondary (Second level/L2/2次) cache
  - primary cacheのmiss penaltyを緩和



近年は集積度の向上により同一チップ内に実装される。  
かつ、3次cacheも導入され、チップ内に搭載

# Multilevel Cache (2)

- Intel Core i9-11900K Processor (2021)
  - # of cores = 8
  - Base Clock Frequency = 3.5 GHz
  - Max Turbo Boost Frequency = 5.3 GHz
  - Caches
    - 1<sup>st</sup> level instruction cache (private)
      - **32KB** × 8 cores = total 256 KB, 8-way set associative
    - 1<sup>st</sup> level data cache (private)
      - **48KB** × 8 cores = total 384 KB, 12-way set associative, write-back
    - 2<sup>nd</sup> level unified cache (private)
      - **512KB** × 8 cores = total 4MB, 8-way set associative, write-back
    - 3<sup>rd</sup> level cache (shared, on-chip)
      - Total **16MB**, 16-way set associative



# I470F 統合アーキテクチャ

例外、割込み処理の実現

# Exception and Interrupt (1)

- 命令が逐次に移動しない要因
  - 条件分岐命令の実行
    - beq, bne
  - 無条件ジャンプの実行
    - j
  - 例外の発生
    - 命令実行による何らかの違反やページフォルトなど
  - 割込み発生
    - プロセッサ外部の要因
    - トランプ(システムコール／ソフトウェア割込み)命令の実行
      - teq, tne(意図的なOSへの移動)
- 例外／割込みが発生した場合、例外ハンドラ／割込み処理ルーチンへ移動し、処理完了後に元のプログラム実行に戻る

# Exception and Interrupt (2)

- MIPSの慣習では、例外(exception)と割込み(interrupt)の両方がしばしば例外(exception)と呼ばれる
- Intel IA-32では、両方を割込み(interrupt)と呼ぶ
- 明確にするために、本講義では内部(internal)要因を例外(exception)、外部要因を割込み(interrupt)と呼ぶことにする

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

# How Exceptions Are Handled (1)

- 例えば、ある実装では3種類の例外が起こりうるとする
  - 未定義命令の実行
  - 算術オーバーフロー
  - トランプ(システムコール)命令の実行
- 例外が発生したときの基本的な動作
  1. 例外を起こした命令のアドレス(後に復帰する場所)を exception program counter (EPC) に格納する
  2. OS内のある特定のアドレスへ実行を移す
    - 移動先はアーキテクチャ/OSによってあらかじめ定義される
- OSに移動後、OSは適切な動作を行う。例えば、
  - 未定義命令の場合、ユーザプログラムの実行を終了し、エラーを報告する
  - 算術オーバーフローの場合、あらかじめ決められた動作を行う
  - システムコールの場合、ユーザプログラムにあらかじめ決められたサービスを提供する

## How Exceptions Are Handled (2)

- 例外を処理するためには、OSは発生した例外の要因を知る必要がある。以下の2つの方法がある：

- 原因レジスタ(Cause register)

- 例外発生の理由を示すフィールドを保持。例外発生時にハードウェアによって自動的に値がセットされる

- Vectored exception/interrupt

- 例外発生時に制御が移る先のアドレスは、例外の要因ごとに定義される。例えば、

Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000
Arithmetic overflow	8000 0180
...	...

# Exceptions/Interrupts In Pipelining (1)

## ■ パイプラインへの例外／割込み用データパス／制御の追加

Arithmetic overflowの場合

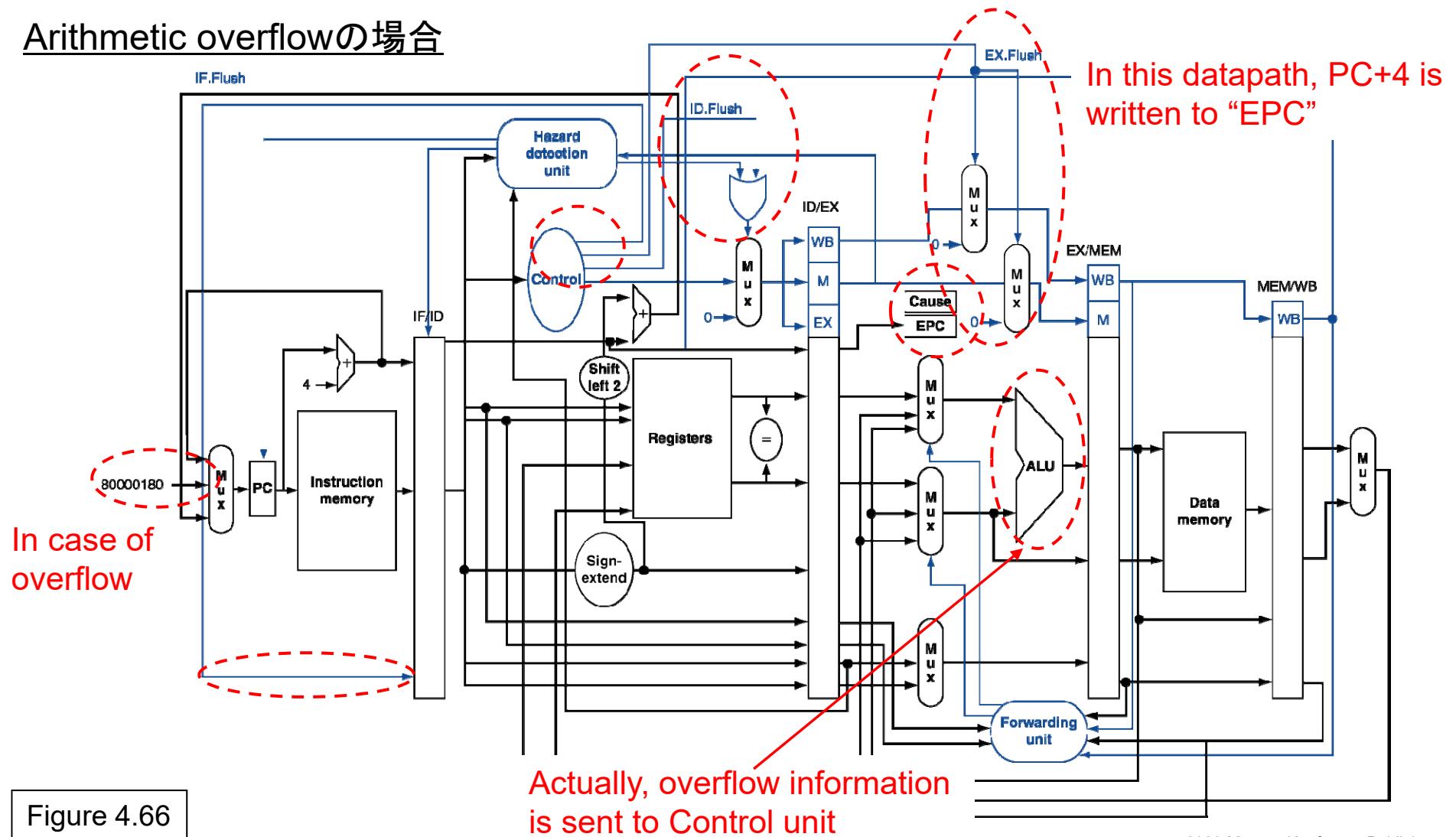


Figure 4.66

# Exceptions/Interrupts In Pipelining (2)

## ■ 以下の命令列を考える

40hex	sub	\$11, \$2, \$4
44hex	and	\$12, \$2, \$5
48hex	or	\$13, \$2, \$6
4Chex	add	\$1, \$2, \$1 ← オーバーフローが発生
50hex	slt	\$15, \$6, \$7
54hex	lw	\$16, 48(\$7)

## ■ 例外発生時に以下の命令列が起動されると仮定：

80000180hex	sw	\$25, 1000(\$0)
80000184hex	sw	\$26, 1004(\$0)

⋮  
⋮

## ■ add (4Chex)でオーバーフロー例外が発生すると、 パイプライン内部はどのような動作となるか？

 *Next page*

# Exceptions/Interrupts In Pipelining (3)

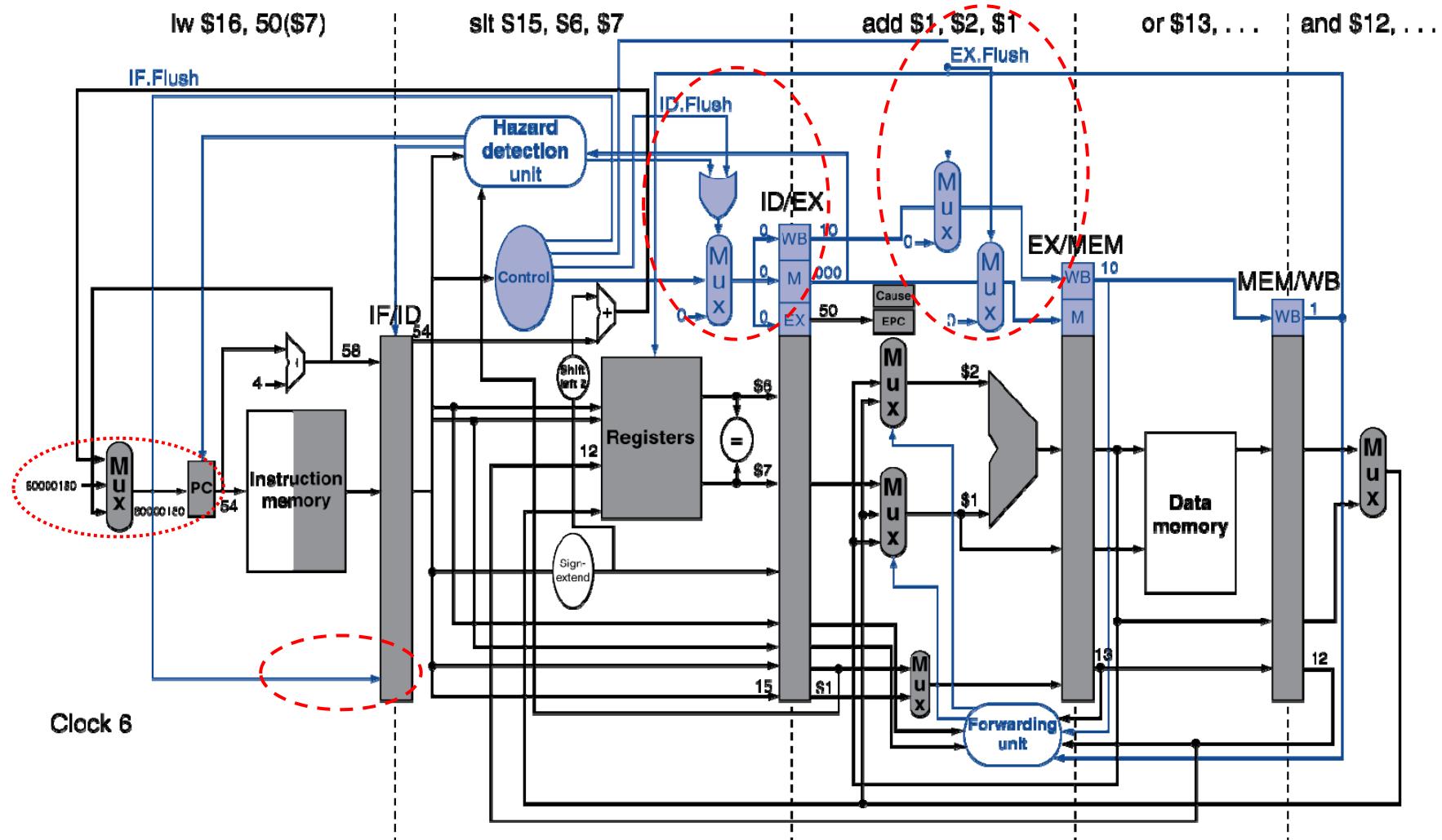


Figure 4.67 (top)

# Exceptions/Interrupts In Pipelining (4)

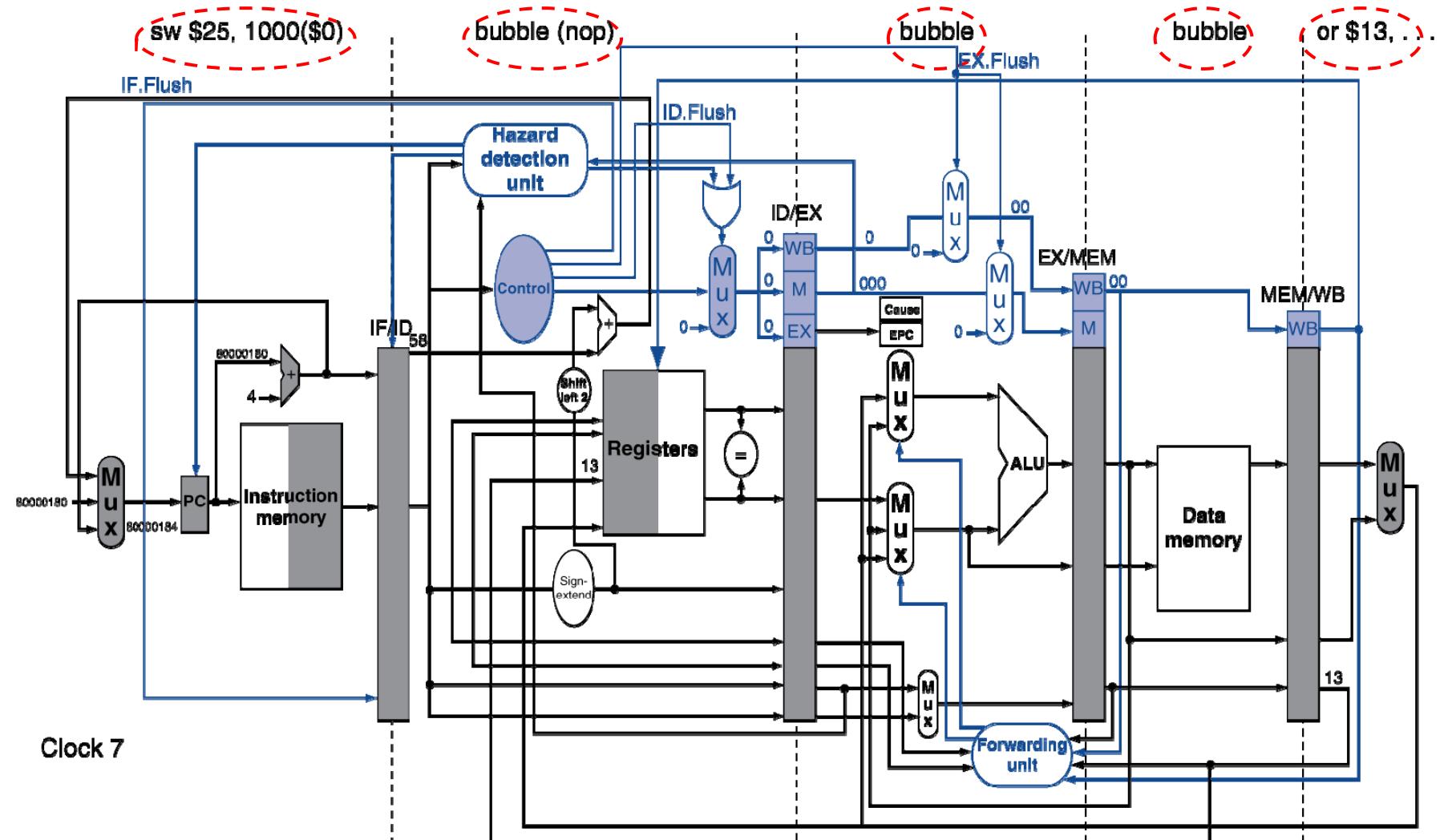
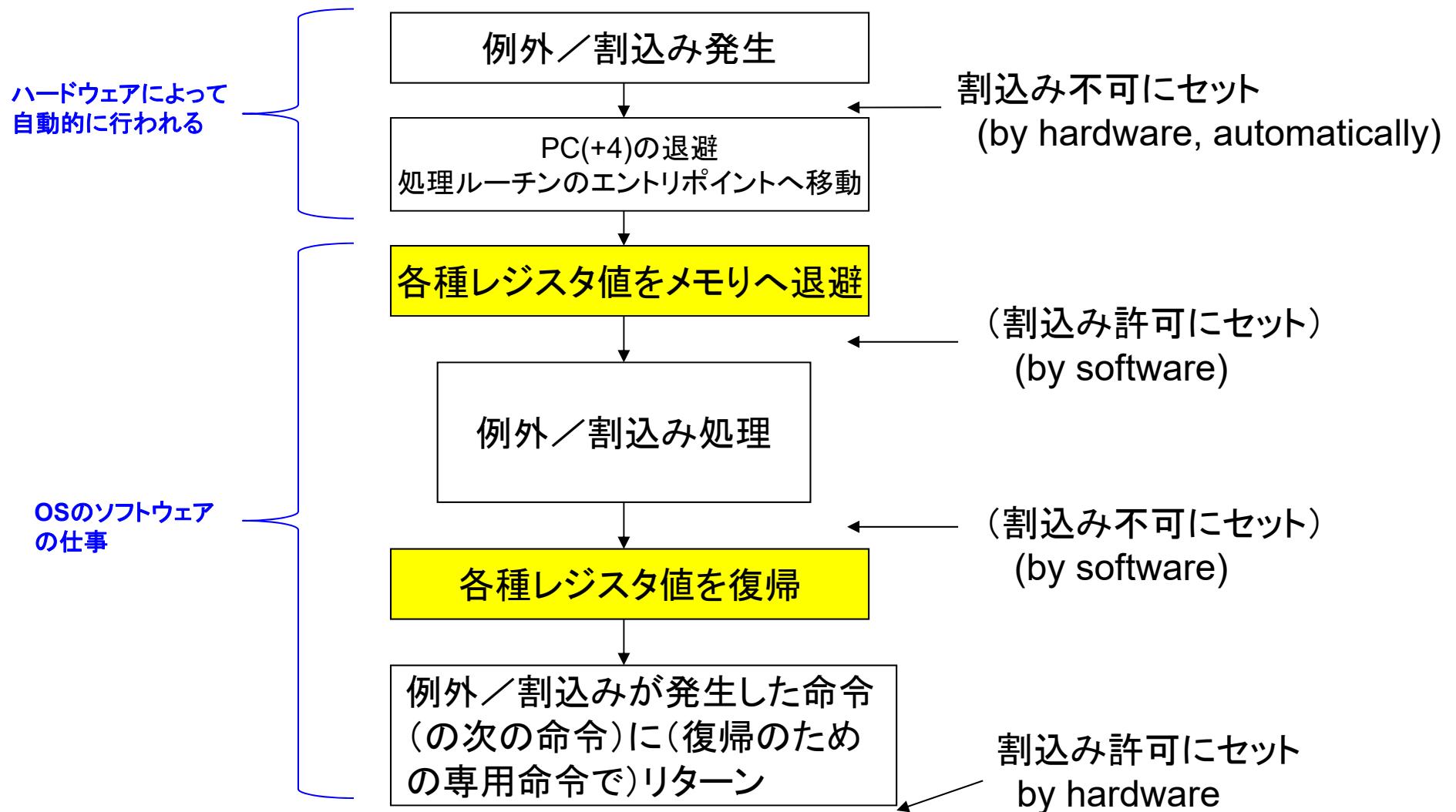


Figure 4.67 (bottom)

# Exception/Interrupt Handling

## ■ 例外／割込み発生後の流れ



# Exception/Interrupt Type and Priority

## ■ 例外／割込みの優先度(主なもの)

高

Reset

Address error : (Inst Fetch)

TLB refill : (Inst Fetch)

TLB invalid : (Inst Fetch)

Cache (parity/ECC) error : (Inst Fetch)

Bus (time-out/parity, etc.) error : (Inst Fetch/external)

Integer overflow, Trap Inst, Floating Point exception

Address error : (Data Access)

TLB refill : (Data Access)

TLB Invalid : (Data Access)

TLB modified : (Data Access)

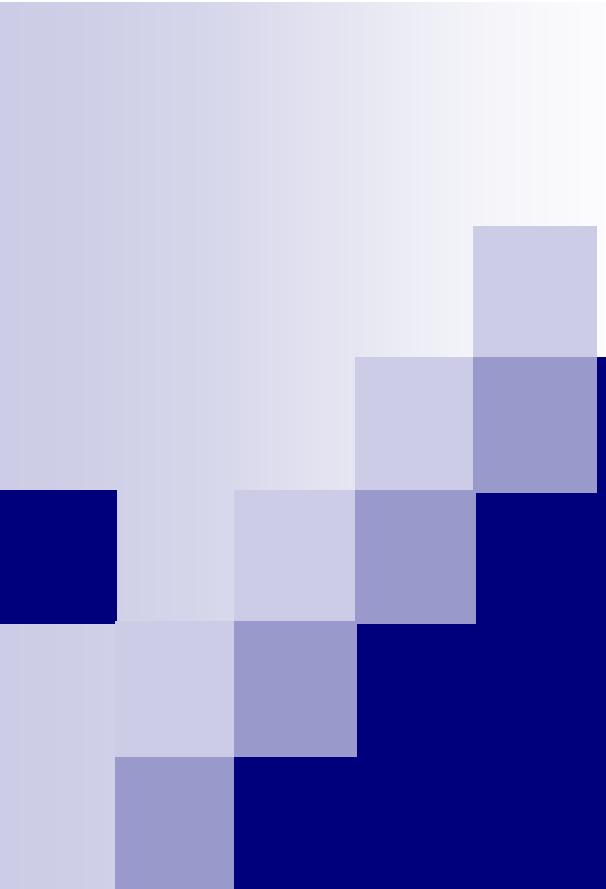
Cache (parity/ECC) error : (Data Access)

Bus (time-out/parity, etc.) : (Inst Fetch/external)

Interrupt (優先度は最も低い)

低

2つ以上の例外／割込みが同時に発生した場合は、先行する命令に関するもの、かつ最も高い優先度をもつものが最初に受け付けられる



# I470F 統合アーキテクチャ

リアルタイムタスクスケジューリング方式

# リアルタイム性(1)

## ■ プログラムの正確さ

### □ 論理的な正確さ

- 計算・処理結果が正しいこと

- $1 + 2 = 3$

- 通常のアプリケーションはこれだけで十分

### □ 時間的な正確さ

- 何時までに計算・処理を終わらせるべきか

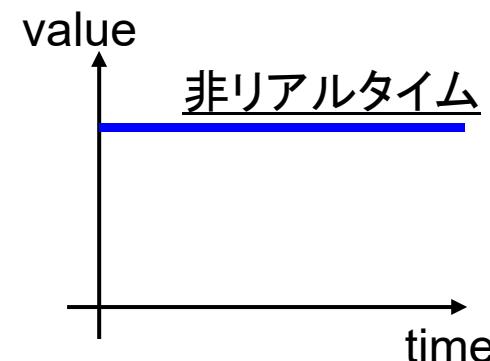
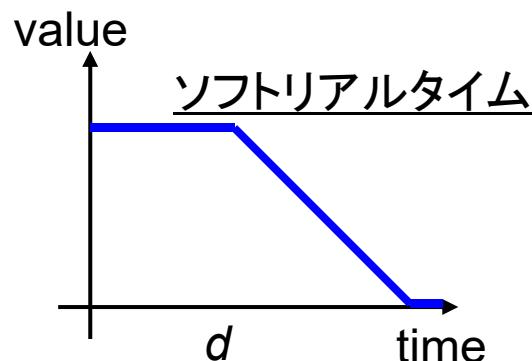
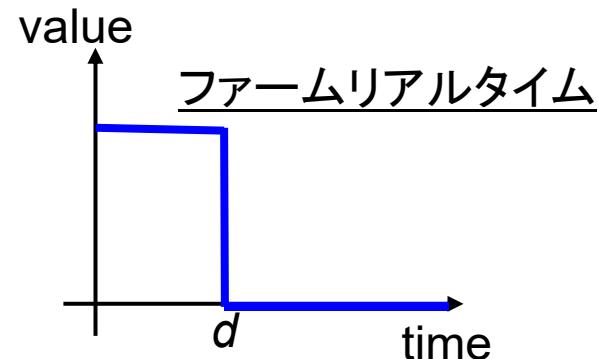
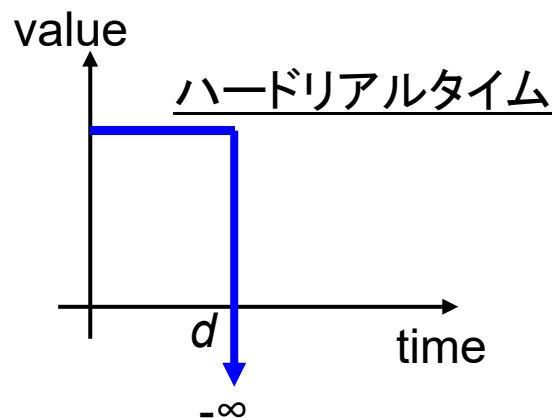
- 締切時刻(デッドライン)

### □ リアルタイムアプリケーションは両方の正確さが要求される

# リアルタイム性(2)

## ■ デッドラインとタスクの重要度の関係

$d$ : デッドライン時刻



# 最悪実行時間(1)

- リアルタイムアプリケーションの時間的な正しさを保障するためには、プログラムの実行時間を事前に知ることが必要
- 入力や実行時の状況で実行時間は変動するが、その中で最も長い実行時間のことを最悪実行時間(WCET: Worst Case Execution Time)という
- 正確なWCETの取得は極めて困難
  - プログラム中のループ回数が不明な場合
  - 動的なデータ構造を使用している場合  
⇒ 一般的にはプログラミングに制約を課すことで回避
  - それでもなお、割込み発生回数やキャッシュヒット／ミスは予測不可能

# 最悪実行時間(2)

## ■ WCETの見積り

### □ 実測による方法

- (リアルタイムOSが提供する) 時間計測のライブラリを使用して実測
  - VxWorksでは、時間計測やCPU使用率をレポートするライブラリがある
  - Worst Case(プログラム中の最悪パスの実行)である保障はない

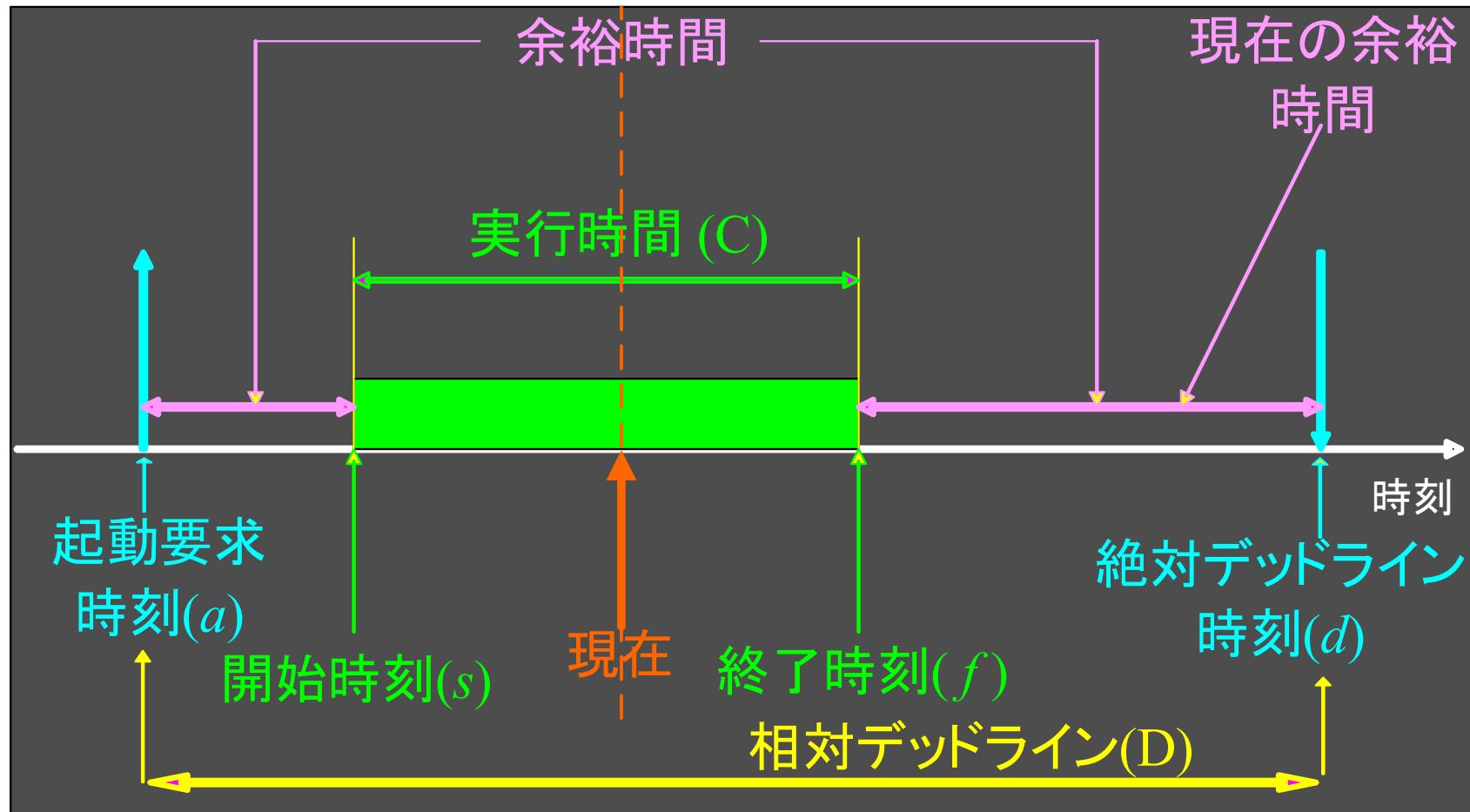
### □ プログラム解析による方法

- プログラムからControl flow graphを抽出し、実行パスを列挙し、基本ブロック毎の命令数や、メモリ参照のキャッシュヒット／ミスを解析
  - プログラミングに少なからず制約を設ける必要あり(固定ループ回数、再帰関数を使わない、等)
  - マルチタスク環境ではキャッシュヒット／ミスは変動する
    - このため、ハードリアルタイムではキャッシュを使用しないこともある
  - 割込みによる影響を予測できないため、本当のWCETになりえない
  - 他のタスクと同期・通信を行う場合は、待ち時間を予測できない

⇒ 結局、過大見積もりをせざるを得ない

# タスクの時間的属性

## ■ スケジューリングで使用される時間的属性



# リアルタイムスケジューリング

## ■ 1つのタスクのみのシステム

- 1つのプロセッサ(CPU)が1つのタスクを実行
- 十分な計算性能があればリアルタイム処理が可能

## ■ 複数のタスクからなるシステム

- タスク数と同数のプロセッサを使用するのはコスト大
- したがって、1つのプロセッサで複数のタスクを実行
- タスクの実行順序がリアルタイム性に影響

→ リアルタイムスケジューリング

# スケジューリング方式の分類(1)

## ■ オフラインスケジューリング

□ 静的にタスクの実行順序を決定しておく

☺ 実行時オーバヘッドが小さい

☹ 柔軟性に欠ける

→ システム稼働時に新たなタスクの発生は不可

→ 割込みによるタスクの起動要求を受け付けることが不可

## ■ オンラインスケジューリング

□ 動的に実行順序を決定

☺ 現実世界の事象に対応可能

→ 割込みによるタスクの起動要求

☹ 順序を決めるための計算量／オーバヘッドが問題

# スケジューリング方式の分類(2)

## ■ プリエンプティブ／ノンプリエンプティブ

- タスク実行の中斷あり／なし

## ■ 静的／動的優先度

- タスクの優先度が固定／変動

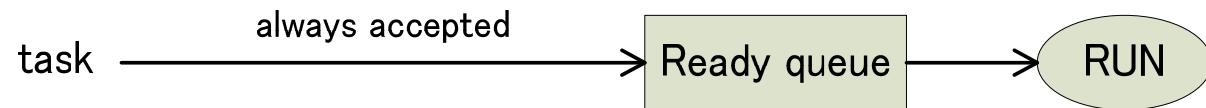
## ■ Best effort／Guarantee／Robust

- タスクの起動要求の際に、無条件に受け付ける／  
スケジュール可能性を確認してから受理または拒否／  
スケジュール不可能ならば拒否し、再トライさせる

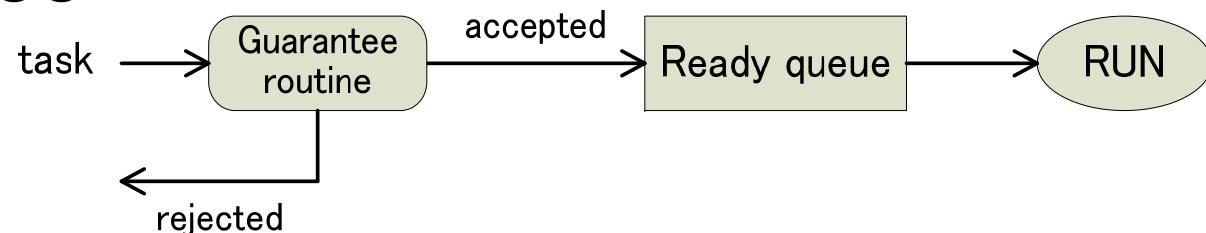
例)ITRONはオンライン＆プリエンプティブ＆(準)静的  
優先度＆Best effort型のスケジューリング

# スケジューリング方式の分類(3)

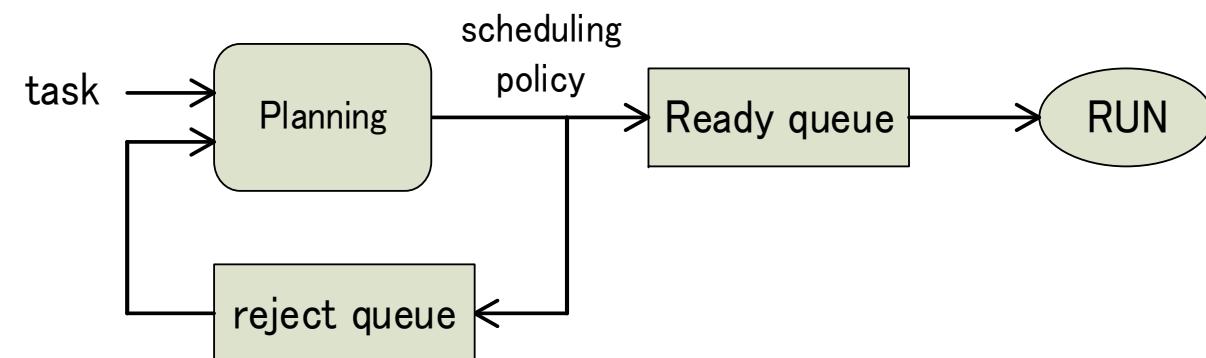
## ■ Best effort



## ■ Guarantee



## ■ Robust



# スケジュール評価関数

(1) 平均応答時間

$$\frac{1}{n} \sum_{i=1}^n (f_i - a_i)$$

(2) 全実行時間

$$\max(f_i) - \min(a_i)$$

(3) 重み付き終了時間

$$\sum_{i=1}^n w_i f_i$$

(4) デッドラインミスの最大オーバー時間

$$\max(f_i - d_i)$$

(5) デッドラインミスしたタスク数

$$\sum_{i=1}^n miss(f_i), \quad miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

(6) ジッタ

$$\max_i(f_i - a_i) - \min_i(f_i - a_i)$$

# Rate Monotonic (RM) (1)

## ■ 前提条件

### 1. 周期タスク

- 各タスク  $\tau_i$  が周期  $T_i$  を持ち、 $T_i$  毎に一回実行される

### 2. 相対デッドライン ( $D_i$ ) は周期と同じ

### 3. プリエンプティブ（実行権の横取りあり）

### 4. タスク同士が独立

- タスク間で依存関係や同期通信が無い

## ■ スケジューリングルール

→ 起動周期の短いタスクほど高い優先度を割り当てる

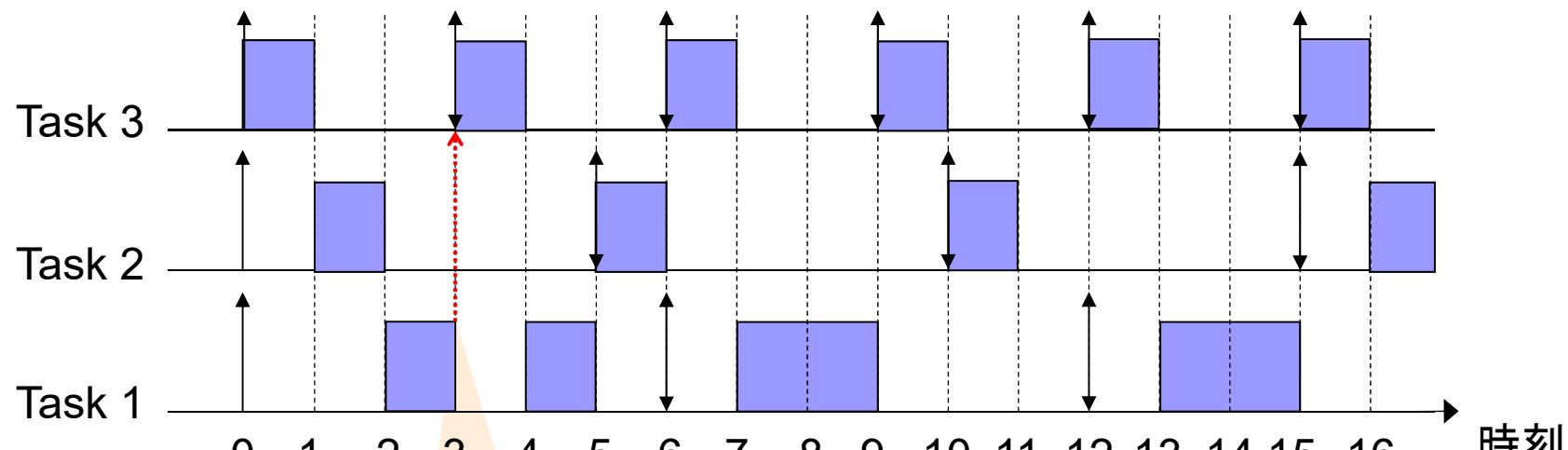
## ■ 最適性：(本前提条件での)最適な静的優先度割付け法

- 「RMによってスケジュール可能でないタスクセットは、他のいかなる静的優先度割付けスケジューリング法でもスケジュール可能ではない」ことが証明されている

# Rate Monotonic (RM) (2)

## ■ スケジューリング例

Task (i)	1	2	3
$C_i$	2	1	1
$T_i$	6	5	3



プリエンプション  
(実行権の横取り)

# Rate Monotonic (RM) (3)

- タスクセットがスケジュール可能となる十分条件

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \rightarrow \ln 2 \cong 0.69$$

n	上限
1	1.000
2	0.828
3	0.780
...	...
$\infty$	0.693

- タスク数が多い場合、プロセッサの能力は69%までしか使えない(= プロセッサ使用率は69%まで)
- ただし、上記は必要条件ではない
  - この値を超えていても、スケジュール可能なタスクセットは存在する

# Earliest Deadline First (EDF)(1)

- 前提条件
  - 1. 周期または非周期タスク
  - 2. 相対デッドラインは周期と同じ(周期タスクの場合)
  - 3. 一般的にはプリエンプティブが前提
  - 4. タスク同士が独立
- スケジューリングルール
  - 絶対デッドライン時刻が近い(早い)順に高い優先度を割り当てる
- 最適性:(本前提条件での)最適な動的優先度割付け法
  - 「EDFは、デッドラインミスの最大オーバー時間を最小化し、スケジュール可能なタスクセットならば、必ずスケジュール可能である」ことが証明されている

# Earliest Deadline First (EDF)(2)

- 周期タスクセットがスケジュール可能となる必要十分条件

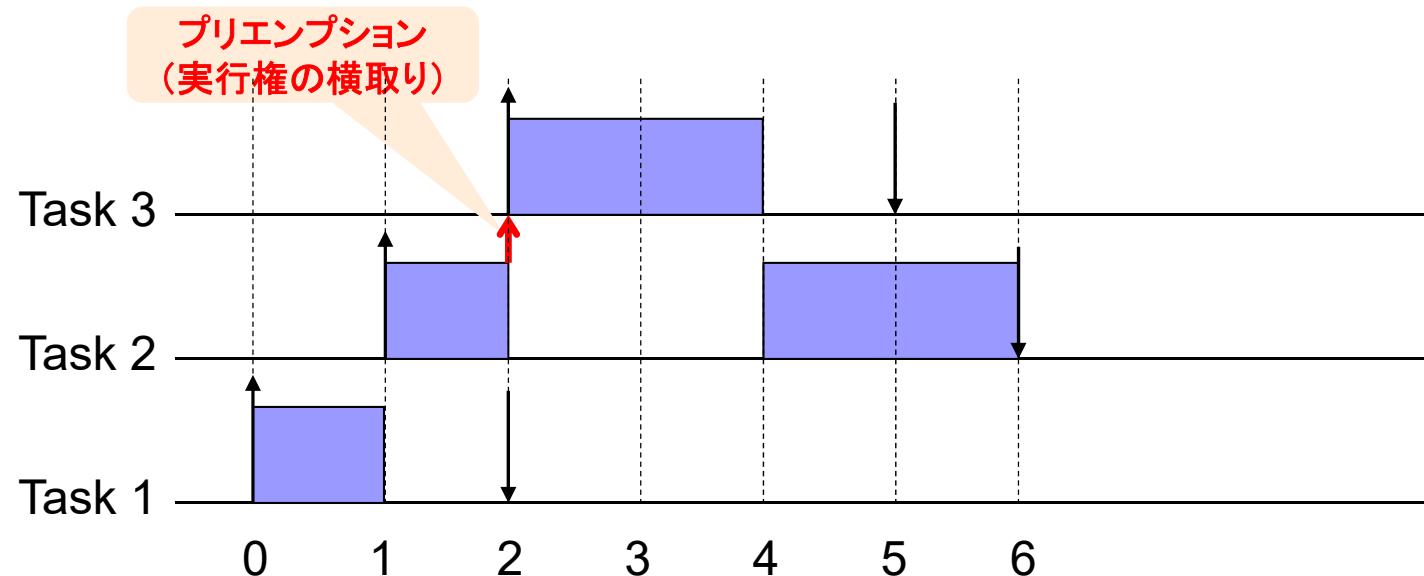
$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

→プロセッサの能力を100%使用可能  
= 100%のプロセッサ使用率を達成可能

# Earliest Deadline First (EDF) (3)

- スケジューリング例  
(プリエンプティブ)  
(非周期タスクの場合)

Task (i)	1	2	3
$a_i$	0	1	2
$C_i$	1	3	2
$D_i$	2	5	3



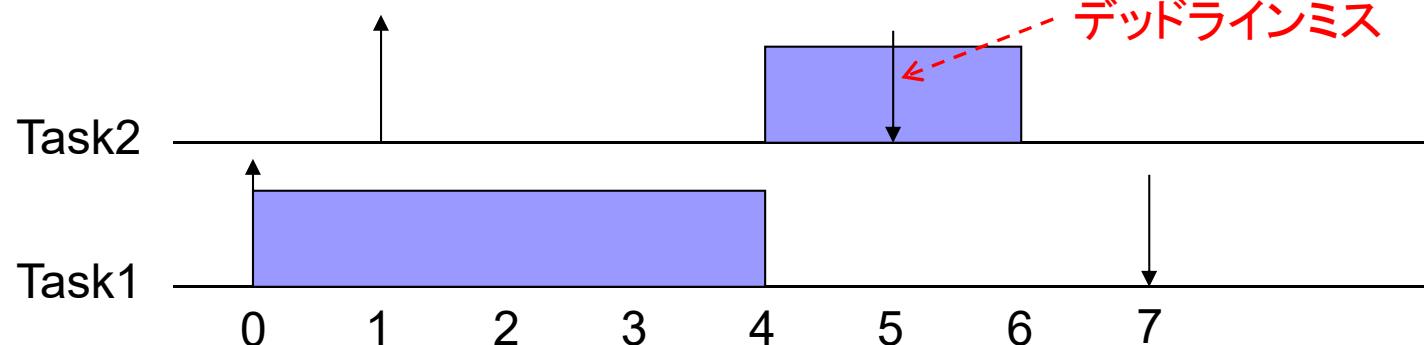
# Earliest Deadline First (EDF) (4)

ノンプリエンプティブにした場合

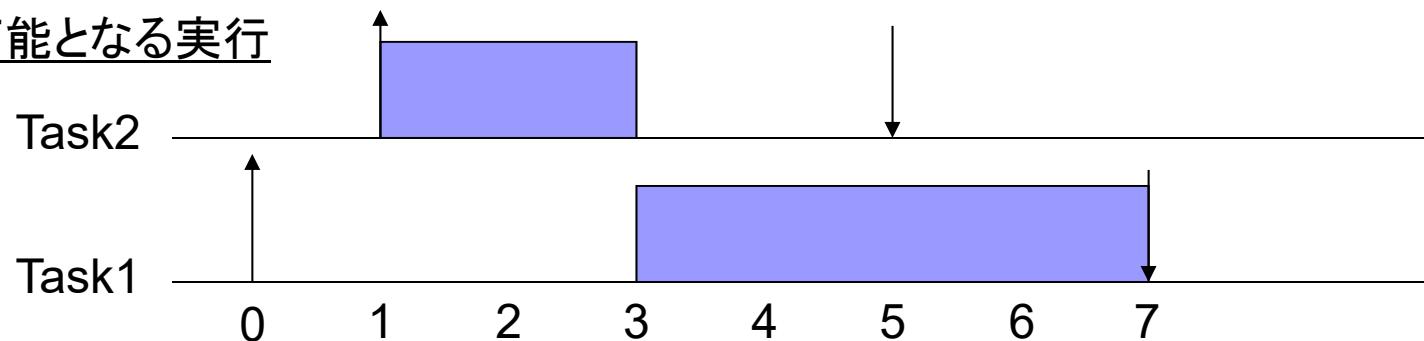
→ EDFの最適性は得られない

Task (i)	1	2
a <sub>i</sub>	0	1
C <sub>i</sub>	4	2
D <sub>i</sub>	7	4

EDFの場合



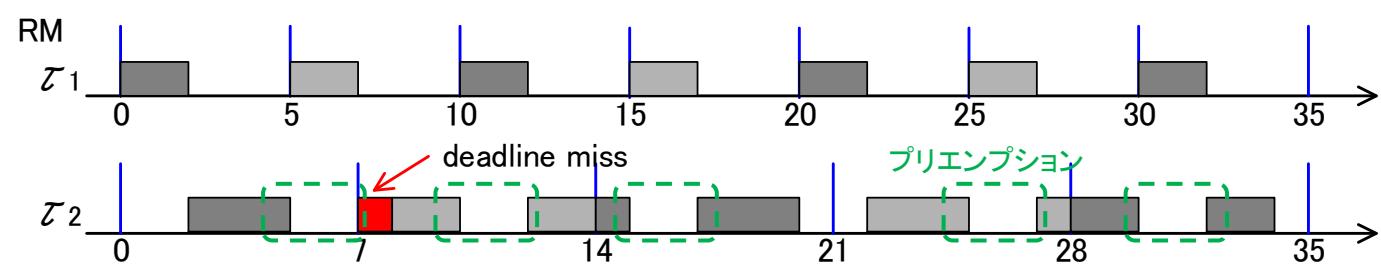
スケジュール可能となる実行



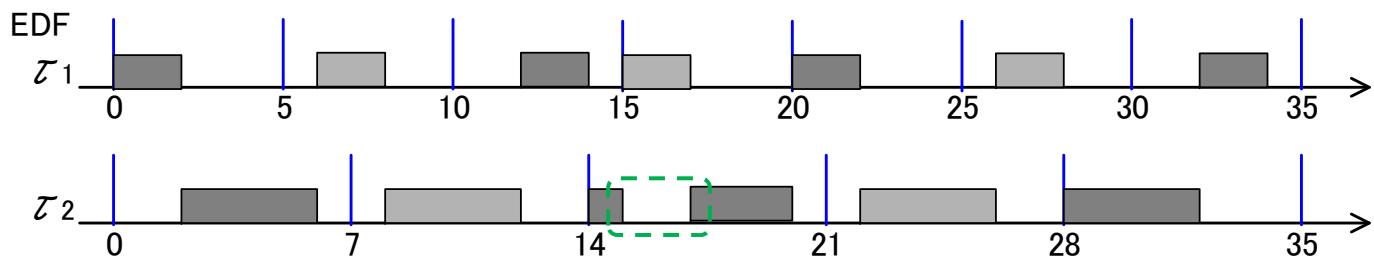
# RM vs. EDF (1)

- 周期タスクセットをターゲットとした場合、EDFのほうがより高いプロセッサ使用率のタスクセットに対してスケジュール可能となる
- EDFのほうがRMよりもプリエンプションが少ない傾向

	$T_i$	$C_i$
$\tau_1$	5	2
$\tau_2$	7	4

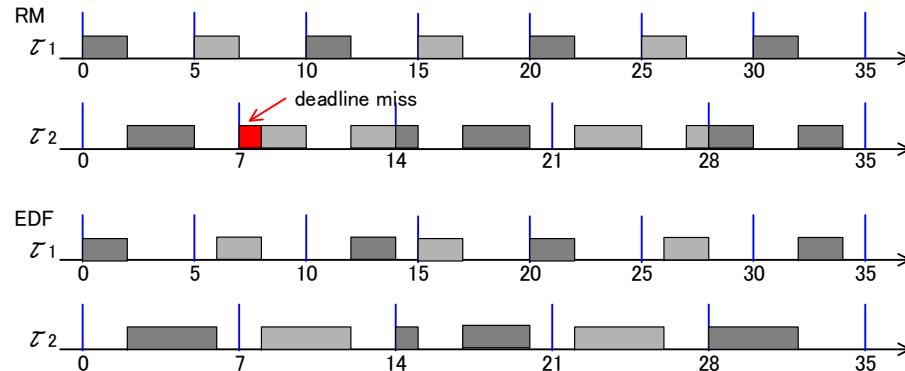


$$U = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} \cong 0.97$$



# RM vs. EDF (2)

- 「静的優先度=タスクの重要度」と考えると、高負荷時に、RMでは必ず重要度の低いものからデッドラインミスが起こる
- RMでは、重要度の高いタスクのjitter(同一タスクの応答時間の差)が小さい

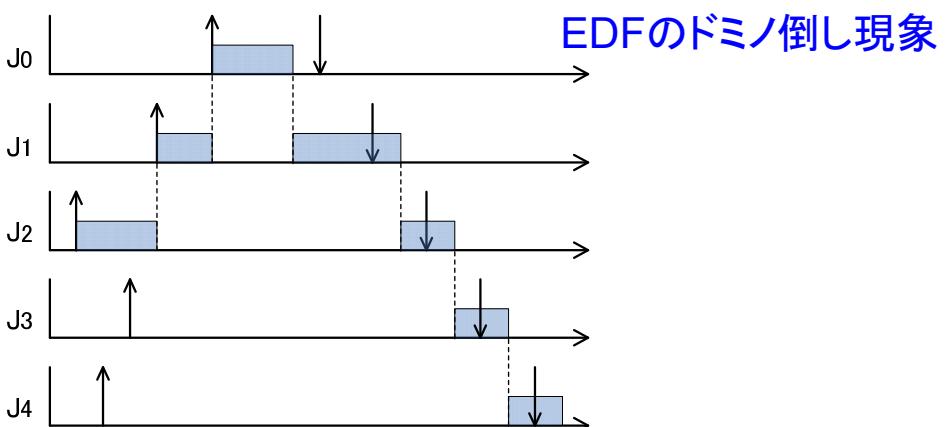
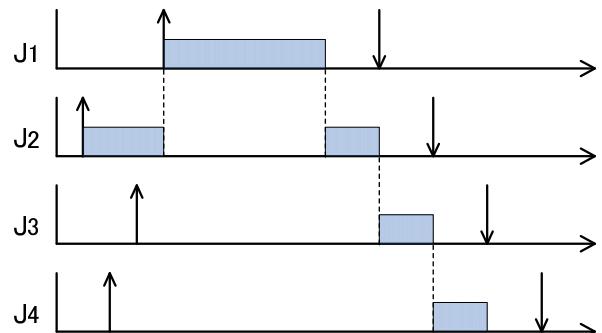


- RMのほうが、優先度を表現するためのビット数が少ない(RMでは優先度を表現するビット数, EDFでは時間を表現するビット数)  
→ 結局、どちらが優れているとは言えない。対象システムの要求に合わせて選択すべき

# その他のスケジューリング法(1)

## ■ Least Laxity First (LLF)

- 余裕時間の小さいタスクほど高い優先度を割り当てる
- EDFと同一の前提条件で、同じ意味で最適な動的優先度割付け法
- EDFのドミノ倒し現象を緩和できる可能性(best effortの場合)
- 余裕時間は時間とともに変化するため、余裕時間を再計算する必要がある。そのため計算オーバヘッドが大きい



# その他のスケジューリング法(2)

## ■ Deadline Monotonic (DM)

- RM法を“相対デッドライン  $< T$ ”の場合に拡張したもの
- 特定周期タスクの応答時間を短くしたい場合に使用される
- RMと同じ意味で最適, かつデッドラインミスの最大オーバー時間に対して最適

## ■ タスク設計者による固定(静的)優先度割り当て

- 各タスクの重要性を考慮して, あらかじめ優先度を割り当てる  
例) ITRON, OSEK, VxWorksなどにおけるタスクの優先度
- タスクの周期にしたがって優先度を割当ることにより RM を実現可能

## ■ First Come First Service (FCFS)

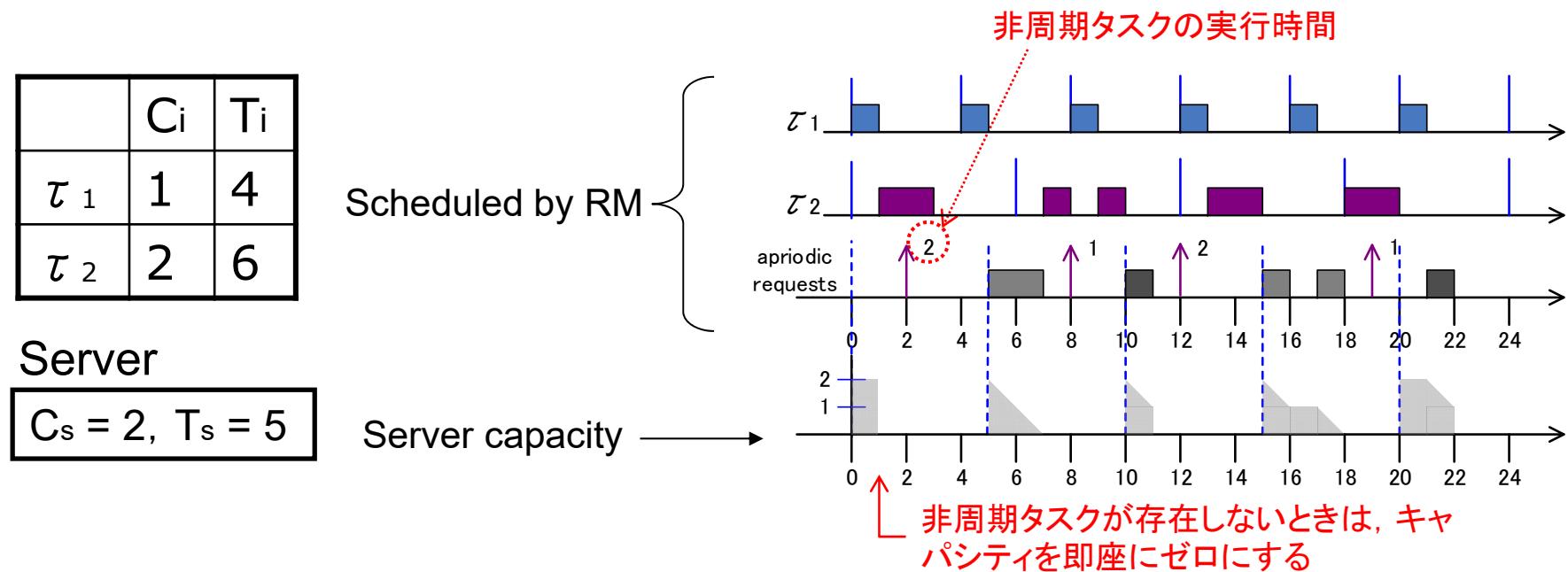
- 到着(起動要求)が早いタスクほど高い優先度を割り当てる
- 例) ITRONにおける, 同一静的優先度内の順序付け

# その他のスケジューリング法（3）

- 周期タスクと非周期タスク(ソフト or デッドラインを持たない)が混在するシステムのためのスケジューリング

## □ RMベースの例: Polling Server

- 周期と実行キャパシティを持つ仮想タスク(サーバ)を導入し, RM下での実行権を非周期タスク実行に充てる
  - サーバを含めてスケジュール可能性を保証する



# その他のスケジューリング法(4)

## □ EDFベースの例: Total Bandwidth Server

- プロセッサ使用率の余剰分を考慮し、以下の式で非周期タスクにデッドラインを与える

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s} \quad (d_0 = 0)$$

Release time  
 Deadline of the previous aperiodic task

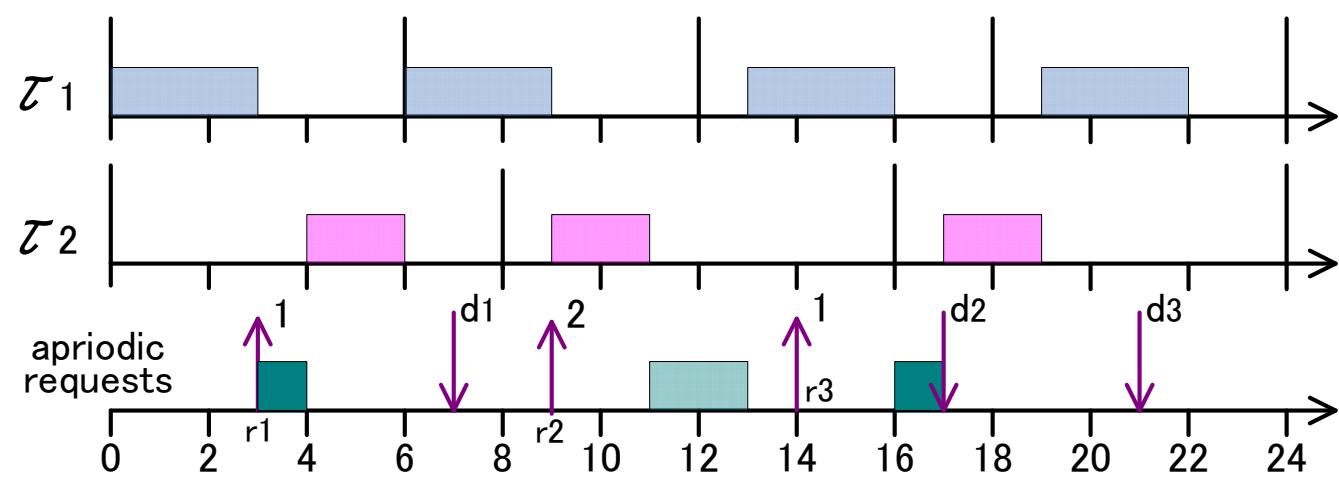
Execution time (= worst-case execution time)  
 Server utilization:  $U_s = 1 - U_p$

- 余剰使用率のみを使用することにより、周期タスクのスケジュール可能性は保たれる。（EDFは100%の使用率までスケジュール可能であるため）

	$C_i$	$T_i$
$T_1$	3	6
$T_2$	2	8

Server

$$U_s = 1 - U_p = 0.25$$

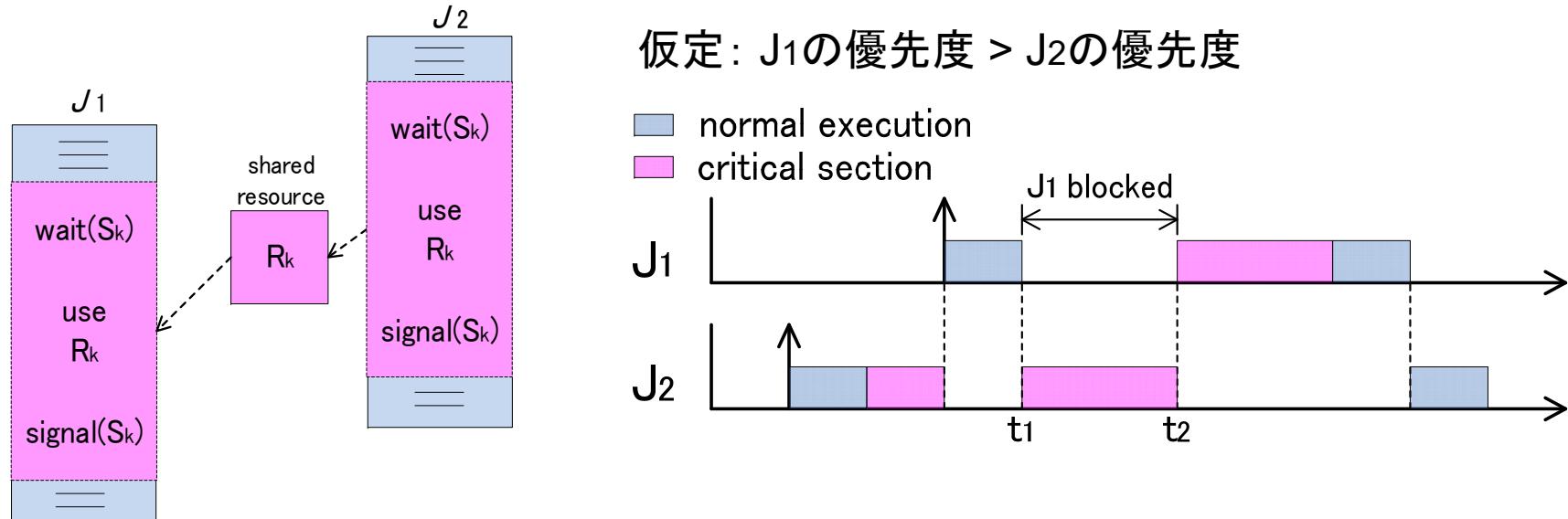


# 共有資源 (1)

- 共有資源(shared resource)とは、複数のタスクから参照されるもの
  - データ／変数、ファイル、周辺デバイス
- 共有資源の一貫性を維持するために、排他的(exclusively)に参照する必要がある
- 排他的参照を実現するために、タスクプログラムはクリティカルセクションを含む
  - クリティカルセクションはセマフォ等を利用して実現する

# 共有資源 (2)

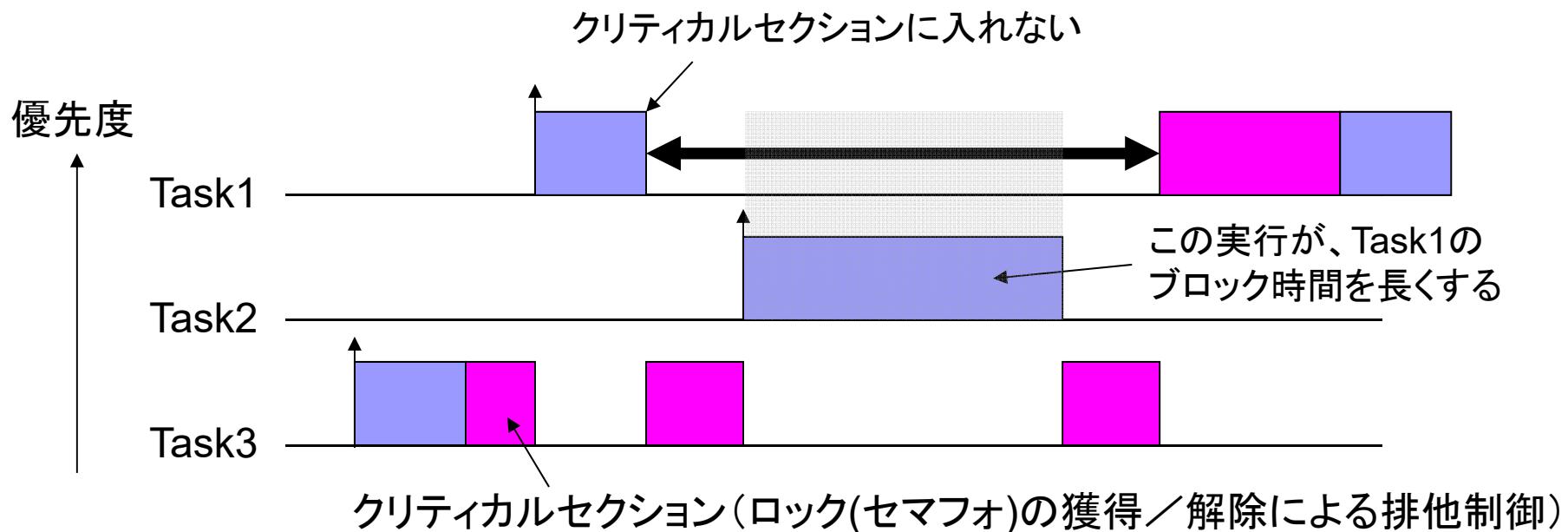
- クリティカルセクションの例
  - セマフォの手続き(wait, signal)を利用



- タスク間の優先度関係にかかわらず、クリティカルセクションの実行時にブロックが発生しうる

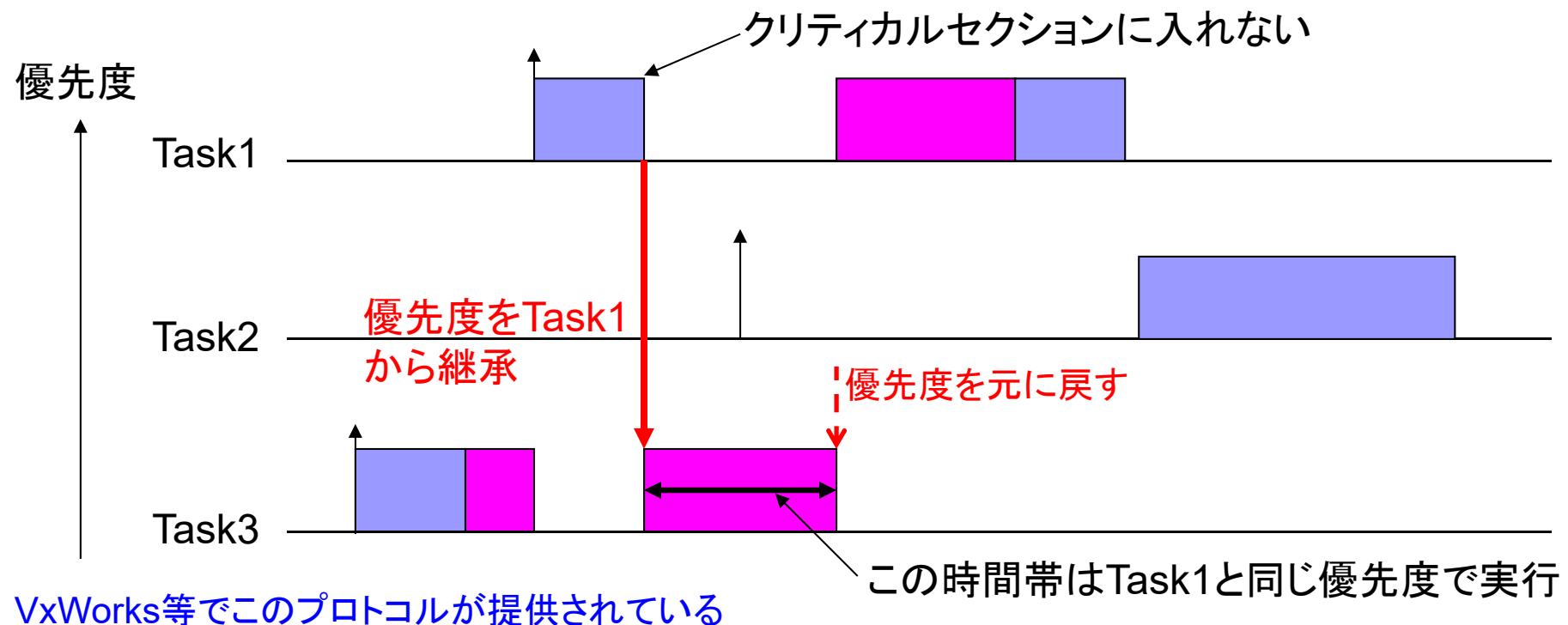
# 優先度逆転問題(Priority Inversion)

- タスク間で共有資源がある場合に、優先度の高いタスクが低いタスクに待たされることがある
- それにともない、共有関係の無い中間優先度のタスクが実行される場合の影響を問題視



# 優先度継承プロトコル

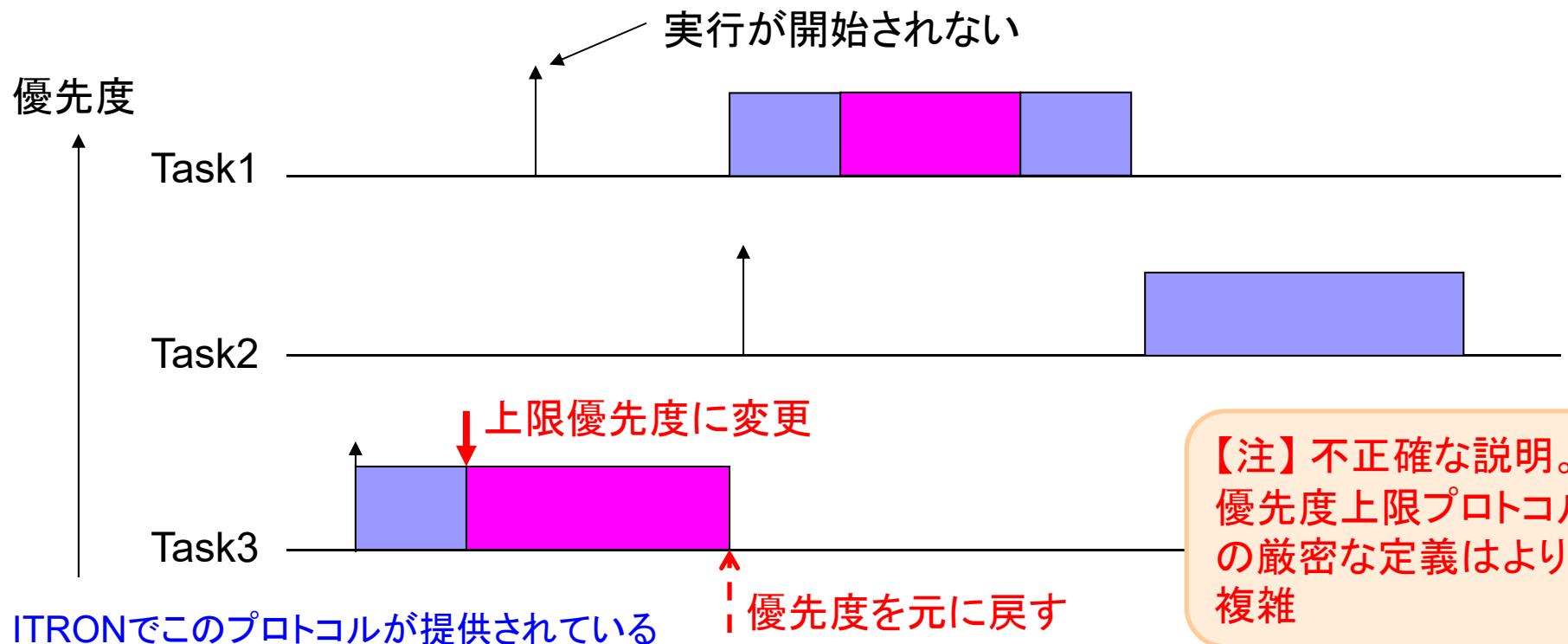
- 待ち要因を作っているタスクの優先度を、待たされるタスクの優先度まで一時的に上昇させる

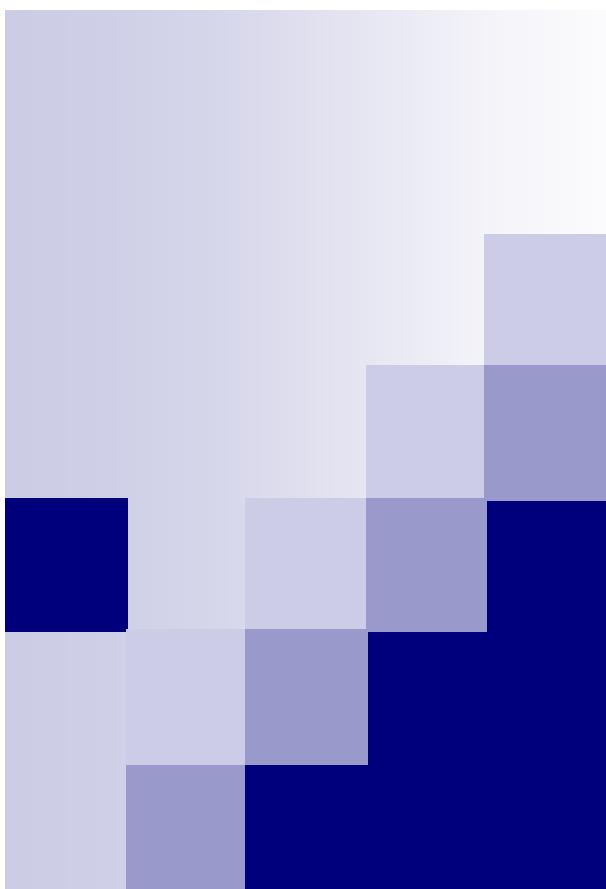


# 優先度上限プロトコル

- ロックを獲得したタスクの優先度を、待たされるタスクの有無によらずあらかじめ決められた優先度に一時的に上昇させる

⇒ 長所：プリエンプション回数を削減





# I470F 統合アーキテクチャ

リアルタイムOSの概要と機能  
および実装

# リアルタイムカーネルの構造(1)

## ■ カーネルは以下の基本機能を提供

### □ プロセス(タスク)管理

- OSが提供すべき主要サービス

- プロセス生成, 終了, スケジューリング, ディスパッチ(コンテキストスイッチ), など

### □ 割込み制御

- 周辺デバイスからの割込み要求への応答

- 割込み処理の実行時間を含めてスケジュール可能性を保証することが重要

### □ プロセス(タスク)間同期・通信

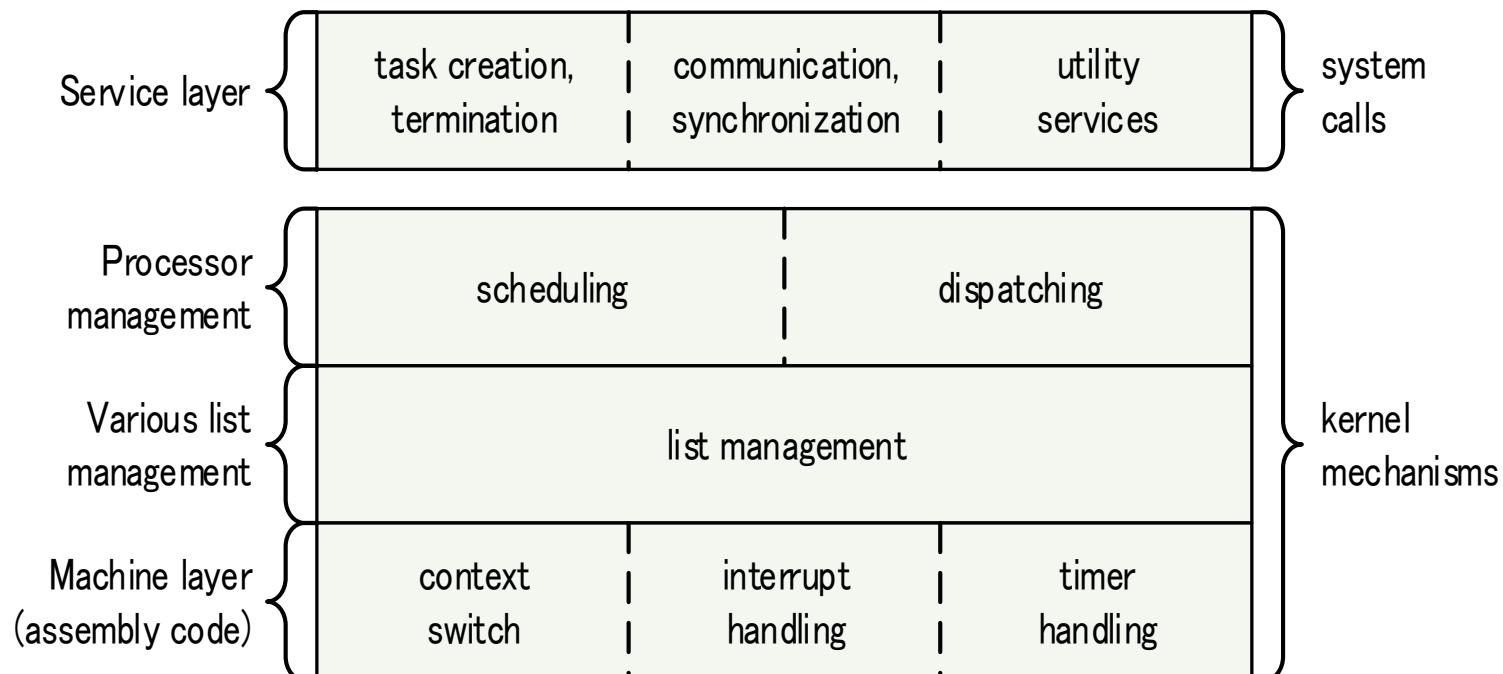
- セマフォ, その他の同期・通信機構

- 優先度逆転を回避するための特別な種類のセマフォ

# リアルタイムカーネルの構造(2)

## ■ DICK (Didactic C Kernel)

□ A small real-time kernel



Hierarchical structure of DICK.

# プロセス状態(1)

## ■ プロセス(タスク)の状態

### □ **RUN(実行状態)**

- プロセッサ上で実行されている状態

### □ **READY(実行可能状態)**

- 実行準備ができているが、他の（優先順位のより高い）実行状態タスクが存在するために実行されていないタスクの状態。このようなタスクは ready queue に入れられる

### □ **WAIT(待ち状態)**

- (セマフォ等の)同期オブジェクトへのプリミティブ手続きを実行し、待つ必要が生じたときにこの状態になる。このようなタスクは同期オブジェクトに関連付けられた wait queue に入れられる。待ち要因が解決されたとき、wait queueから削除し、ready queueに入れられる

# プロセス状態(2)

## □ IDLE(アイドル)

- 周期タスクがその周期の実行を終了したときに入る状態
- 次の周期の開始時に再度READY状態に移行

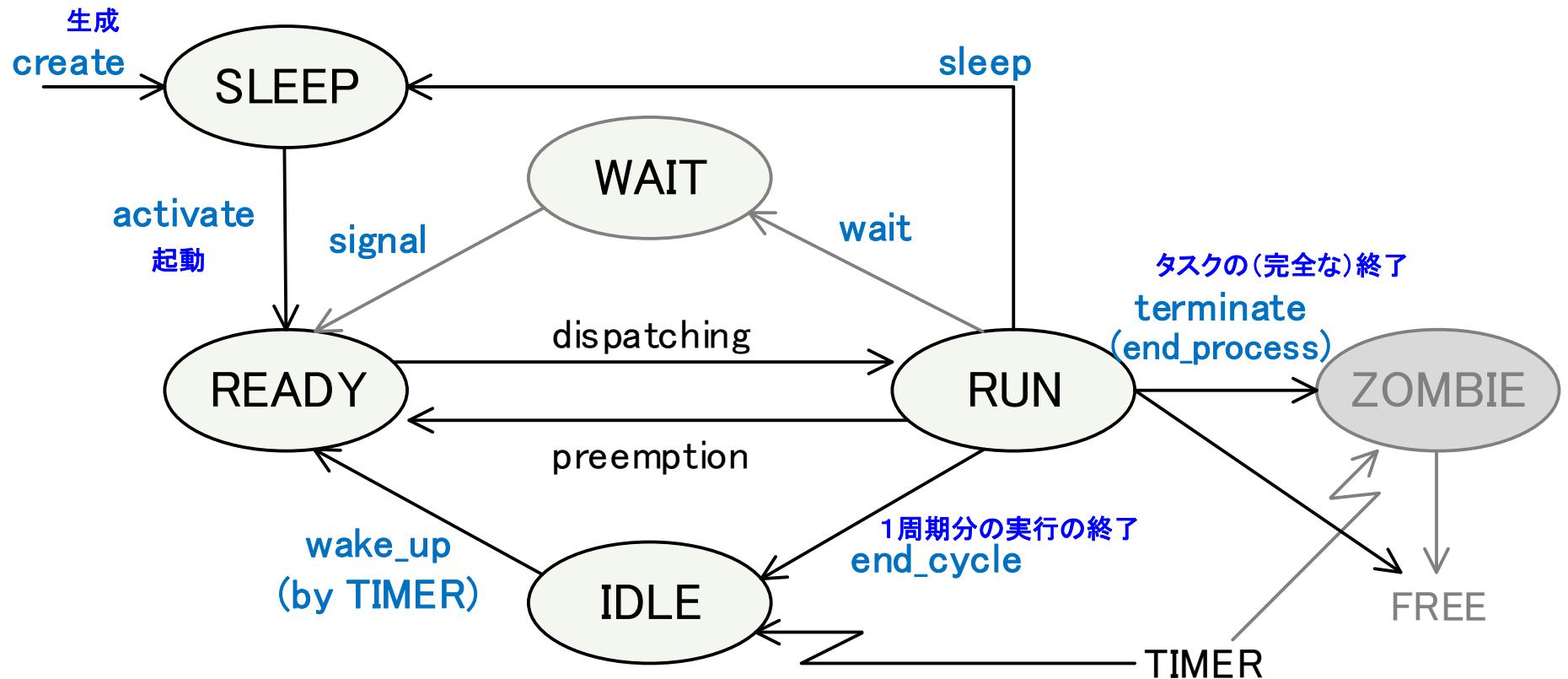
## □ SLEEP(スリープ)

- タスクが生成されたとき, あるいは“sleep”プリミティブを実行したときにこの状態になる. スリープ状態のタスクが他のタスクによって起動されたとき, ready queueに入る

## □ ZOMBIE(ゾンビ)

- 周期タスクが終了(terminate)したときにこの状態になる. 最終周期の最後に, タスクが開放される
- スケジューラビリティの保証のためにこの状態がある

# プロセス状態(3)



# データ構造(1)

## ■ タスク制御ブロック(**Task Control Block, TCB**)

- タスクについての情報のためのデータ構造
  - プログラマによって[生成時用のパラメータ](#)が指定される
  - その他, [タスクを管理するための一時的な情報を管理](#)
- **タスク指示子** (char name[MAXLEN+1];)
  - ユーザにメッセージを送るとき等に使用される文字列
- **タスクアドレス** (proc (\*addr)();)
  - タスクコードの先頭のメモリ番地
- **タスクタイプ** (int type;)
  - ハードリアルタイム(周期タスク) or 非リアルタイム(非周期タスク)
- **状態** (int state;)
  - 現在の状態 (RUN, READY, IDLE, WAIT, SLEEP, ZOMBIE).

# データ構造(2)

## □ 実行時間 (int wcet;)

- 最悪実行時間(worst-case execution time)

## □ 周期 (int period;)

- ハードリアルタイムタスクの場合は周期 (=相対デッドライン) , 非リアルタイムタスクの場合は優先度を格納 (後ほど)

## □ 絶対デッドライン (long dline;)

- 起動時にカーネルによって計算される絶対デッドライン”時刻”

## □ 利用率 (float util;) (wcetとperiodによって計算され, 不変)

- タスクによるプロセッサ利用率(周期タスク用)

## □ コンテキストポインタ (int \*context;) (セットされた後は不变)

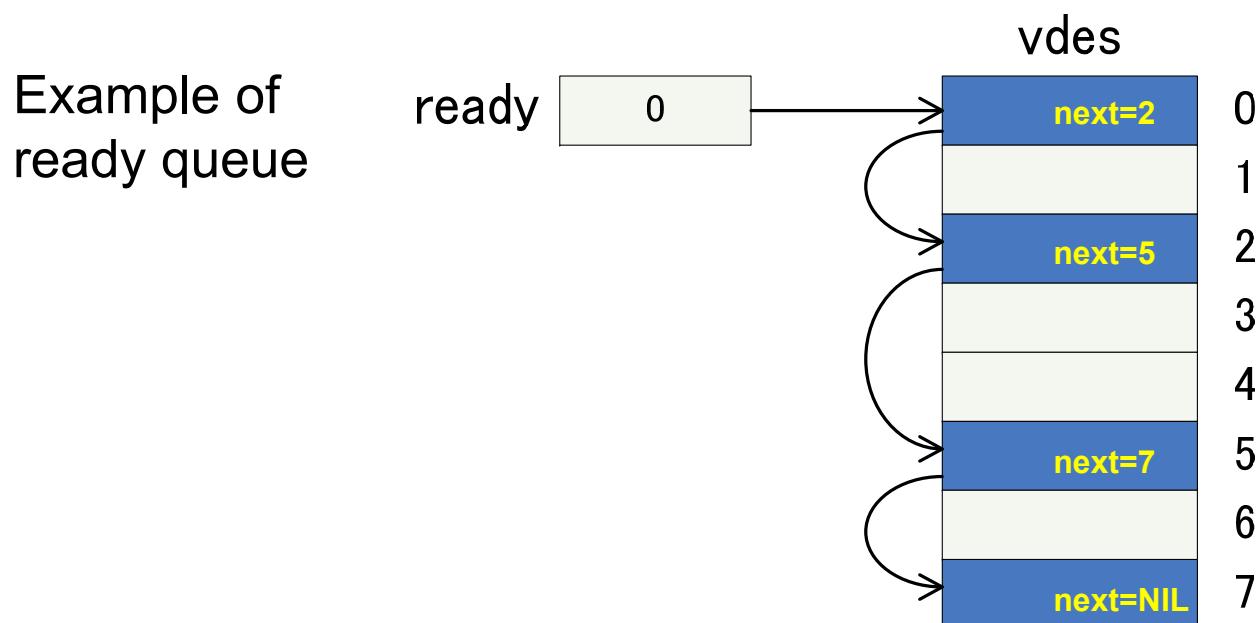
- タスクのコンテキスト(PC, レジスタ値など)が保存されるスタックへのポインタ

## □ 他のTCBへのポインタ (proc next; proc prev;)

- タスクを各種キューに挿入するときに使用されるポインタ

## データ構造(3)

- DICKにおいて、TCBは配列 vdes[MAXPROC] の要素
    - タスクのキュー(ready queueなど)は先頭からTCBのインデクスを表す整数値を使用して参照可能



# データ構造(4)

## ■ セマフォ制御ブロック(Semaphore Control Block、SCB)

- セマフォについての情報のためのデータ構造
- カウンタ (int count;)
  - セマフォ値を表すカウンタ
- セマフォ・キュー (queue qsem;)
  - 当該セマフォでブロックしたタスクの wait queue
- 他のSCBへのポインタ (sem next;)
  - フリーのセマフォのリストにおける次のSCBへのポインタ
- DICKでは、SCBは配列 vsem[MAXSEM]の要素

# データ構造(5)

## Type definitions

```
typedef int queue;      /* head index */  
typedef int proc;       /* process index */  
typedef int sem;        /* semaphore index */
```

## Structure of TCB

```
struct tcb {  
    char   name[MAXLEN+1];           /* task identifier/name */  
    proc   (*addr)();               /* task address */  
    int    type;                   /* task type (hard or non-real-time) */  
    int    state;                  /* task state */  
    long   dline;                  /* absolute deadline */  
    int    period;                 /* period(HARD) or priority(NRT) */  
    int    wcet;                   /* worst-case execution time */  
    float  util;                  /* task utilization factor */  
    int    *context;                /* pointer to the context */  
    proc   next;                  /* pointer to the next tcb */  
    proc   prev;                  /* pointer to previous tcb */  
};
```

# データ構造(6)

## Structure of SCB

```
struct scb {  
    int    count;           /* semaphore counter */  
    queue  qsem;          /* semaphore wait queue */  
    sem    next;           /* pointer to the next */  
};
```

## Kernel data

```
struct tcb    vdes[MAXPROC]; /* tcb array */  
struct scb   vsem[MAXSEM];  /* scb array */  
proc        pexe;          /* task in execution */  
queue      ready;          /* ready queue */  
queue      idle;           /* idle queue */  
queue      zombie;         /* zombie queue */  
queue      freetcb;        /* queue of free tcbs */  
queue      freesem;        /* queue of free semaphores */  
float      util_fact;      /* utilization factor */
```

# 時間管理

- 定期的にプロセッサに割込みを発生させるようにタイマ回路を設定し、タイマ割込みの度に内部のシステム時刻を更新する

```
unsigned long sys_clock;      /* system time (# of ticks after reset) */  
float          time_unit;    /* unit of time (ms) */
```

- システム初期化時からの実際の経過時間

$$t = \text{sys\_clock} \times \text{time\_unit};$$

- システムの存続期間は変数のサイズと時間単位(`time_unit`の長さ=ティック)に依存

32ビット変数の場合

time_unit	lifetime
1 ms	50 days
5 ms	8 months
10 ms	16 months
50 ms	7 years

# タスクのタイプとスケジューリングアルゴリズム

- DICKでは、タスクには2つのタイプがある
  - ハードタスク(周期タスク): クリティカルなデッドラインを持つ
    - 周期的に起動される
    - 各周期の実行はプリミティブ “*end\_cycle*” によって終了し, idle状態になり、次の周期の始めにタイマによって自動的に起動される
    - ハードタスクはEDFによってスケジュールされる
  - 非リアルタイムタスク(非周期タスク): 固定優先度を持つ
    - 非周期に起動される(周期はない)
    - 優先度にしたがい、ハードタスクのバックグラウンドで実行される
    - 実際には、優先度は常にハードタスクのデッドラインよりも大きいデッドラインに変換される → 全体をEDFでスケジュールできる

$$d_i^{NRT} = MAXDLINE - PRT\_LEV + P_i$$

( $MAXDLINE = 2^{31}-1$ ,  $PRT\_LEV = \# \text{ of priority levels}$ ,  $P_i$ : the smaller, the higher.)

# 記号定数(1)

## ■ 定数(Global constants)

```
#define MAXLEN 12      /* max string length */  
#define MAXPROC 32      /* max number of tasks */  
#define MAXSEM 32       /* max number of semaphores */  
#define MAXDLINE 0xFFFFFFF /* max deadline */  
#define PRT_LEV 255      /* priority levels */  
#define NIL -1          /* null pointer */  
#define TRUE 1  
#define FALSE 0  
#define LIFETIME MAXDLINE - PRT_LEV
```

# 記号定数(2)

## ■ タスクのタイプ, 状態, エラーメッセージ

#define HARD	1	/* critical task	*/
#define NRT	2	/* non real-time task	*/
#define FREE	0	/* TCB not allocated	*/
#define READY	1	/* ready state	*/
#define RUN	2	/* running state	*/
#define SLEEP	3	/* sleep state	*/
#define IDLE	4	/* idle state	*/
#define WAIT	5	/* wait state	*/
#define ZOMBIE	6	/* zombie state	*/
#define OK	0	/* no error	*/
#define TIME_OVERFLOW	1	/* missed deadline	*/
#define TIME_EXPIRED	2	/* lifetime reached	*/
#define NO_GUARANTEE	3	/* task not schedulable	*/
#define NO_TCB	4	/* too many tasks	*/
#define NO_SEM	5	/* too many semaphores	*/

# 初期化(1)

## ■ 初期化

- プリミティブ “*ini\_system()*” の実行によりシステムがスタート
  - タイマ周期をシステムのティック(*time\_unit*)に設定
  - 割込みベクトルをセット(CPUアーキテクチャ依存)
  - カーネル内の全てのキュー、その他を初期化
  - メイン(カーネル)プロセスのためのTCBを用意してセット
- この関数が実行された後、この main process は非リアルタイムタスクとなり、新たに他タスクを生成する

# 初期化(2)

```
void ini_system ( float tick )
{
    proc i;
    time_unit = tick;
    < enable the timer to interrupt every time_unit >
    < initialize the interrupt vector table : アーキテクチャ依存 >
    /* initialize the list of free TCBs and semaphores */
    for ( i=0; i<MAXPROC-1; i++ ) vdes[i].next = i + 1;
    vdes[MAXPROC-1].next = NIL;
    for ( i=0; i<MAXSEM-1; i++ ) vsem[i].next = i + 1;
    vsem[MAXSEM-1].next = NIL;
    freetcb = freesem = 0; /* initialize freetcb/freesem */
    ready = idle = zombie = NIL; /* initialize queues */
    util_fact = 0; /* initialize utilization factor */
    < prepare and set TCB of the main process >
    pexe = <main index>; /* maybe "0" as (kernel) main process */
}
```

# カーネルプリミティブ(1)

- 低レベルプリミティブ(コンテキストのセーブ／ロード)
  - 以下は疑似コード. 実際はアセンブリ言語で記述

```
void save_context ( void )
{
    int *pc;                      /* pointer to context of pexe */

    < disable interrupts >        /* これ以降、割り込み禁止 */
    pc = vdes[pexe].context;
    pc[0] = < register_0 >;       /* save register 0           */
    pc[1] = < register_1 >;       /* save register 1           */
    ...
    pc[n] = < register_n >;       /* save register n           */
}
```

# カーネルプリミティブ(2)

□ 以下は疑似コード. 実際はアセンブリ言語で記述

```
void load_context ( void )
{
    int *pc;                      /* pointer to context of pexe */

    pc = vdes[pexe].context;
    < register_0 > = pc[0];      /* load register 0 */
    < register_1 > = pc[1];      /* load register 1 */
    ...
    < register_n > = pc[n];      /* load register n */

    < enable interrupts &          /* 割り込み許可に戻す */
      jump to the new task, by e.g., "return from interrupt" >
}
```

# カーネルプリミティブ(3)

## ■ リスト管理(挿入)(based on EDF)

```
void insert ( proc i, queue *que ) /* i番タスクを que に挿入 */
{
    int p;                      /* pointer to the previous TCB */ 
    int q;                      /* pointer to the next TCB */ 

    p = NIL; q = *que;
    /* find the element before the insertion point */
    while ( ( q != NIL ) && ( vdes[i].dline >= vdes[q].dline ) ) {
        p = q; q = vdes[q].next;
    }
    if ( p != NIL ) vdes[p].next = i; else *que = i;
    if ( q != NIL ) vdes[q].prev = i;
    vdes[i].next = q; vdes[i].prev = p;
}
```

# カーネルプリミティブ(4)

## ■ リスト管理(抽出)

```
proc extract ( proc i, queue *que ) /* i番タスクを que から取り除く */
{
    int p, q; /* auxiliary pointers */

    p = vdes[i].prev;
    q = vdes[i].next;
    if ( p == NIL ) *que = q; /* vdes[i] is the first element */
    else vdes[p].next = vdes[i].next;
    if ( q != NIL ) vdes[q].prev = vdes[i].prev;
    return i;
}
```

# カーネルプリミティブ(5)

```
proc getfirst ( queue *que ) /* 先頭タスクをqueから取り除き、返す */
{
    int q;                  /* pointer to the first element */

    q = *que;
    if ( q == NIL ) return NIL;
    *que = vdes[q].next;
    vdes[*que].prev = NIL;
    return q;
}
```

```
long firstdline ( queue que )
{ /* 先頭タスクのデッドラインを返す */
    return vdes[que].dline;
}
```

```
int empty ( queue que )
{ /* queが空かどうか調べる */
    if ( que == NIL )
        return TRUE;
    else
        return FALSE;
}
```

# カーネルプリミティブ(6)

## ■ スケジューリング機構

```
void schedule ( void )
{
    /* Based on EDF */
    if ( firstdline ( ready ) < vdes[pexe].dline ) {
        vdes[pexe].state = READY;
        insert ( pexe, &ready );
        dispatch ();
    }
}
```

レディキュー(ready)に変更が加えられたときに呼び出される

```
void dispatch ( void )
{
    pexe = getfirst ( &ready );
    vdes[pexe].state = RUN;
}
```

RUNタスクが終了／中断したときには直接呼び出される

# カーネルプリミティブ(7)

```
void wake_up ( void ) /* timer interrupt handling routine */
{
    proc p;
    int count = 0;

    save_context (); /* 実行中タスクのコンテキストをセーブ */
    sys_clock++;      /* システム時刻を更新 */

    if ( sys_clock >= LIFETIME ) abort( TIME_EXPIRED ); // 時刻が存続期間を超えるとエラーを生成
    if ( vdes[pexe].type == HARD )
        if ( sys_clock > vdes[pexe].dline ) // 時刻がハードデッドラインを超えて
            abort( TIME_OVERFLOW ); // いたらオーバーフローを生成

    while ( !empty( zombie ) && ( firstdline( zombie ) <= sys_clock ) ) {
        p = getfirst( &zombie );
        util_fact = util_fact - vdes[p].util;
        vdes[p].state = FREE;
        insert ( p, &freetcb );
    }
/* continued on the following page */
```

# カーネルプリミティブ(8)

```
/* 次の周期が到達したタスクを起動 */
while ( !empty( idle ) && ( firstdline( idle ) <= sys_clock ) ) {
    p = getfirst ( &idle );
    vdes[p].dline += (long)vdes[p].period; // 次に開始すべきアイドルタス
    vdes[p].state = READY; // クがあれば、それを起こし、
                           // スケジューラを呼び出す
    insert ( p, &ready );
    count++;
}
if ( count > 0 ) // レディキュー(ready)に変更が加えられたときに
    schedule (); // schedule()が呼び出される

load_context (); // タスクのコンテキストをロード
}
```

# システムコール(1)

## ■ システムコール

- OSが提供する機能／サービス
- タスクプログラムから利用するためのインターフェース

## ■ タスク管理のためのシステムコール

- `create()`, `activate()`, `sleep()`, `end_cycle()`,  
`end_process()`, `kill()`

## ■ タスク間同期・通信のためのシステムコール

- `newsem()`, `delsem()`, `wait()`, `signal()`

## ■ その他

- `get_tim()`, `get_state()`, `get_dline()`, `get_period()`

# システムコール(2)

## ■ タスク管理のシステムコール

- *create (with guarantee), activate, sleep, end\_cycle, end\_process, kill*

create():タスクの生成

```
proc create ( char   name[MAXLEN+1],      /* task name          */
              proc    (*addr) (),           /* task address        */
              int     type,                /* type (HARD, NRT)   */
              float   period,              /* period or priority */
              float   wcet )               /* execution time     */
{
    proc p;
    < disable cpu interrupts >
    p = getfirst ( &freetcb );
    if ( p == NIL )
        abort( NO_TCB );
    /* continued on the following page */
```

# システムコール(3)

```
vdes[p].name      = name; /* Actually, characters should be copied */
vdes[p].addr      = addr;
vdes[p].type      = type;
vdes[p].state     = SLEEP;
vdes[p].period    = (int)( period / time_unit ); /* for HARD */
vdes[p].wcet      = (int)( wcet / time_unit );
vdes[p].util      = wcet / period;
vdes[p].dline     = MAX_DLINe + (long)( period - PRT_LEV );

if ( vdes[p].type == HARD )
    if ( guarantee( p ) == FALSE ) return NO_GUARANTEE;

< initialize process stack, including "context" >
< enable cpu interrupts >

return p;
}
```

とりあえず、NRTタスクとしてデッドラインを  
セット。HARDの場合は起動時に再度セット

# システムコール(4)

```
int guarantee ( proc p )
{
    util_fact = util_fact + vdes[p].util;

    if ( util_fact > 1.0 ) {
        util_fact = util_fact - vdes[p].util;
        return FALSE;
    }

    return TRUE;
}
```

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

# システムコール(5)

```
void activate ( proc p )
{
    save_context ();

    if ( vdes[p].type == HARD )
        vdes[p].dline = sys_clock + (long)vdes[p].period;

    vdes[p].state = READY;
    insert ( p, &ready ); //レディキュー変更
    schedule ();
    load_context ();
}
```

activate():タスクの起動

create()実行時にNRTタスクとしてデッドラインがセットされていた。  
HARDの場合はここで再度セット

```
void sleep ( void )
```

```
{
```

```
    save_context ();

    vdes[pexe].state = SLEEP;
```

sleep():タスクのスリープ

```
    dispatch ();
```

```
    load_context ();
```

レディキューの先頭タスクに  
切り替わる場合は  
dispatch()を呼び出す

レディキューに新たなタスク  
が加わったときに  
schedule()を実行

# システムコール(6)

```
void end_cycle ( void )
{
    long dl;
    save_context ();
    dl = vdes[pexe].dline;
    if ( sys_clock < dl ) { /* 周期内で終了した場合 */
        vdes[pexe].state = IDLE;
        insert ( pexe, &idle );
    }
    else { /* 周期の終わりと同時に終了した場合 */
        dl = dl + (long)vdes[pexe].period;
        vdes[pexe].dline = dl;
        vdes[pexe].state = READY;
        insert ( pexe, &ready );
    }
    dispatch ();
    load_context ();
}
```

end\_cycle():HARDタスク  
の1周期分の実行の終了

周期を過ぎていた場合は、タイ  
マ割込みハンドラによって既に  
TIME\_OVERFLOWを検出済

# システムコール(7)

end\_process():タスクの終了(terminate)

```
void end_process ( void )
{
    < disable cpu interrupts >
    if ( vdes[pexe].type == HARD )
        insert ( pexe, &zombie );
    else {
        vdes[pexe].state = FREE;
        insert ( pexe, &freetcb );
    }
    dispatch ();
    load_context ();
}
```

# システムコール(8)

```
void kill ( proc p )
{
    < disable cpu interrupts >
    if ( pexe == p ) {
        end_process ();
        return;
    }
    if ( vdes[p].state == READY ) extract ( p, &ready );
    if ( vdes[p].state == IDLE ) extract ( p, &idle );
    if ( vdes[p].type == HARD )
        insert ( p, &zombie );
    else {
        vdes[p].state = FREE;
        insert ( p, &freetcb );
    }
    < enable cpu interrupts >
}
```

kill():タスクの強制終了

このコードはWAIT状態のタスクのkillを考慮していない

# システムコール(9)

## ■ セマフォのシステムコール(優先度逆転のための機構無し)

□ *newsem, delsem, wait, signal*

```
sem newsem ( int n )
{
    sem s;
    < disable cpu interrupts >
    s = freesem;                                /* first free semaphore index */
    if ( s == NIL )
        abort( NO_SEM );
    freesem = vsem[s].next;                      /* update the freesem list */
    vsem[s].count = n;                            /* initialize counter */
    vsem[s].qsem = NIL;                           /* initialize sem.queue */
    < enable cpu interrupts >
    return s;
}
```

newsem():セマフォの生成

# システムコール(10)

```
void delsem ( sem s )
{
    < disable cpu interrupts >

    vsem[s].next = freesem;          /* inserts s at the head      */
    freesem = s;                   /* of the freesem list       */

    < enable cpu interrupts >
    return;
}
```

delsem():セマフォの削除

# システムコール(11)

```
void wait ( sem s )
{
    < disable cpu interrupts >

    if ( vsem[s].count > 0 )
        vsem[s].count--;
    else {
        save_context ();
        vdes[pexe].state = WAIT;
        insert ( pexe, &vsem[s].qsem );
        dispatch ();
        load_context ();
    }

    < enable cpu interrupts >
}
```

wait():セマフォの獲得

# システムコール(12)

```
void signal ( sem s )
{
    proc p;

    < disable cpu interrupts >

    if ( !empty( vsem[s].qsem ) ) {
        p = getfirst ( &vsem[s].qsem );
        vdes[p].state = READY;
        insert ( p, &ready ); // レディキュー変更
        save_context ();
        schedule ();
        load_context ();
    }
    else
        vsem[s].count++;

    < enable cpu interrupts >
}
```

wait():セマフォの解放

# システムコール(13)

## ■ 状態参照のシステムコール

- *get\_time, get\_state, get\_dline, get\_period*

```
float  get_time ( void )
{
    return time_unit * sys_clock;
```

```
int   get_state ( proc  p )
{
    return vdes[p].state;
```

```
long  get_dline ( proc  p )
{
    return vdes[p].dline;
```

```
float  get_period ( proc  p )
{
    return vdes[p].period;
```

# システムコールの使用例

```
/* 定義 */
proc tsk1, tsk2, tsk3;
sem sem1;
int x, y;

/* カーネルメインタスク */
int main ( void )
{
    ini_system ( 1.0 );
    vdes[0].dline = 0; /* メインタスクを最高優先度とする */
    tsk1 = create ( "task_1", task1, NRT, 10.0, 6.0 );
    tsk2 = create ( "task_2", task2, HARD, 5.0, 1.0 );
    tsk3 = create ( "task_3", task3, NRT, 5.0, 3.0 );
    sem1 = newsem ( 1 );
    activate ( tsk1 );
    activate ( tsk2 );
    activate ( tsk3 );
    vdes[0].dline = MAX_DLNE; /* 最低優先度にする */
    schedule ();
    for ( ; ); /* ここからはアイドル(何もしない)タスク */
    return 0;
}
```

```
/* タスク1 */
void task1 ( void )
{
    wait ( sem1 );
    y = x + ...;
    .
    .
    signal ( sem1 );

    /* 何らかの処理 */
    .
    .
    sleep ();
}
```

```
/* タスク2 */
void task2 ( void )
{
    /* 周期タスク */
    .
    .
    end_cycle ();
}
```

```
/* タスク3 */
void task3 ( void )
{
    /* 何らかの処理 */
    .
    .
    wait ( sem1 );
    .
    .
    x = ...;
    signal ( sem1 );
    sleep ();
}
```