情253「ディジタルシステム設計」

組み込みシステム・プログラミング

Agenda

1. CPUとPeripheral HWとプログラム
2. 組み込みシステム・プログラミングの特徴
3. 開発環境
4. 簡単な応用例

Agenda

1. **CPUとPeripheral HWとプログラム**
2. 組み込みシステム・プログラミングの特徴
3. 開発環境
4. 簡単な応用例

CPUの種類について

- CISC = Complex Instruction Set Computer
  - １命令に複数の機能を持たせた高機能な命令セットを具備。
    - IntelやAMDのx86系が有名
- RISC = Reduced Instruction Set Computer
  - １命令１機能を基本とし、単純化した命令セットを具備。
  - 多数の単機能命令を組み合わせてプログラムを構成する。
  - ロード／ストア・アーキテクチャと組み合わせるのが一般的。
    - ARM、MIPS、PowerPC、SH、PIC、8051などが有名
- DSP = Digital Signal Processor
  - デジタル信号処理に特化した命令セットを具備。
    - TI、Analog Deviceなどが有名

✓ RISC型で、ロード／ストア・アーキテクチャのプロセッサを想定して話を進めます。

プログラムが実行される仕組み(1/3)

- プログラムとは？
  - CPUにやって欲しい仕事の処理手順書
  - 処理手順とデータの集まり
- どこに格納される？
  - メモリ・デバイス　→　ROM、RAM、Cache
- どういう形式で格納される？
  - 0/1のマシン語　→　CPU固有のバイナリ形式

プログラムが実行される仕組み(2/3)

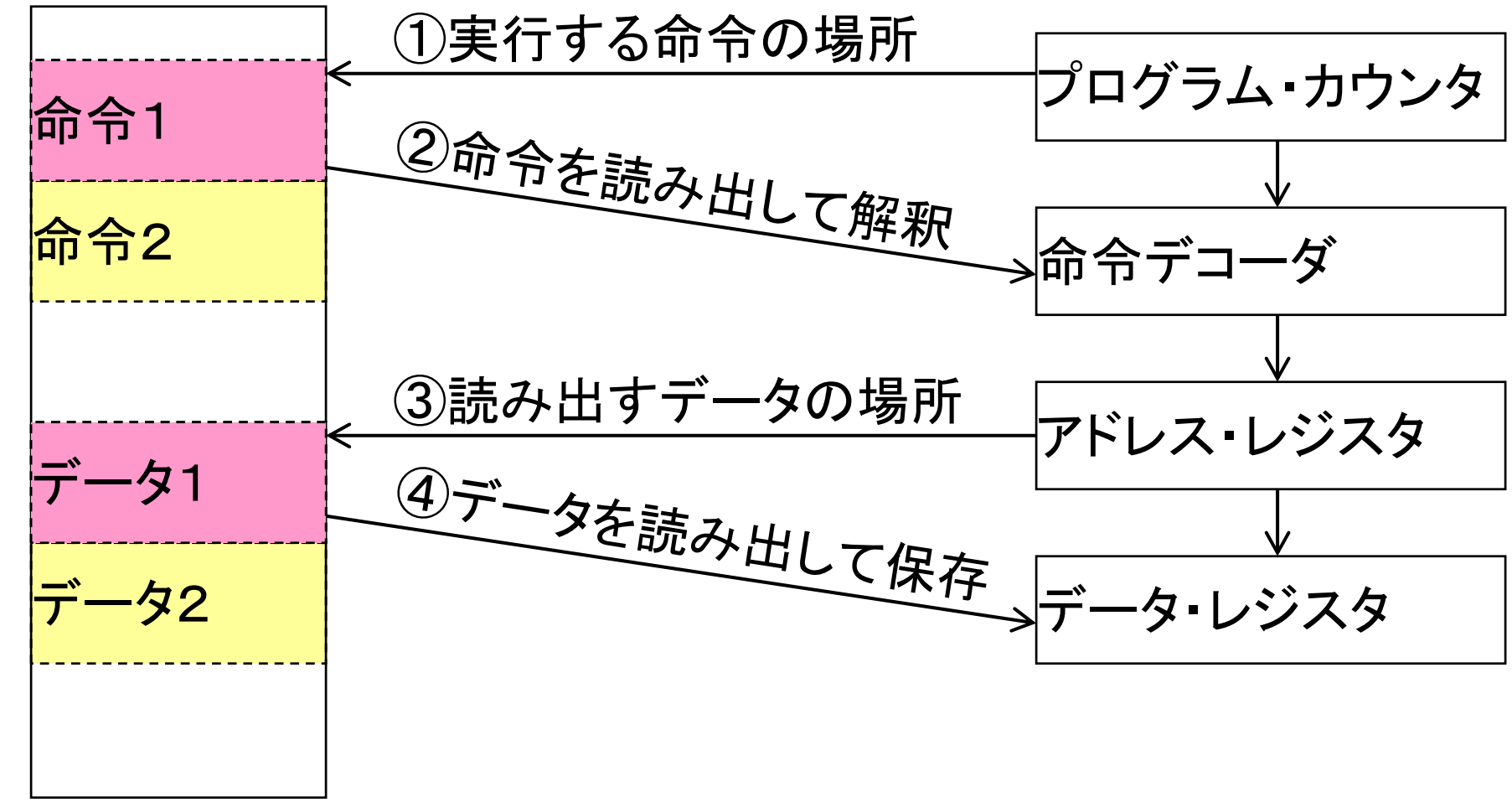
- 必要最低限の用語解説
  - プログラム・カウンタ
    - 実行する処理手順のメモリ上の場所を示す。
  - アドレス・レジスタ（汎用レジスタ）
    - 処理対象データのメモリ上の場所を示す。
    - C言語のポインタ変数のようなもの。

      *int i, a[10], ***ptr**=a;*

      *for (i=0; i<10; i++) *__ptr__++ = i;*
  - データ・レジスタ（汎用レジスタ）
    - CPUがこれから処理する、または処理を終えたデータを一時保管する場所。

プログラムが実行される仕組み(3/3)

「所定の場所からデータを読み出す」という処理手順が、「命令１」の部分に書かれているとします。

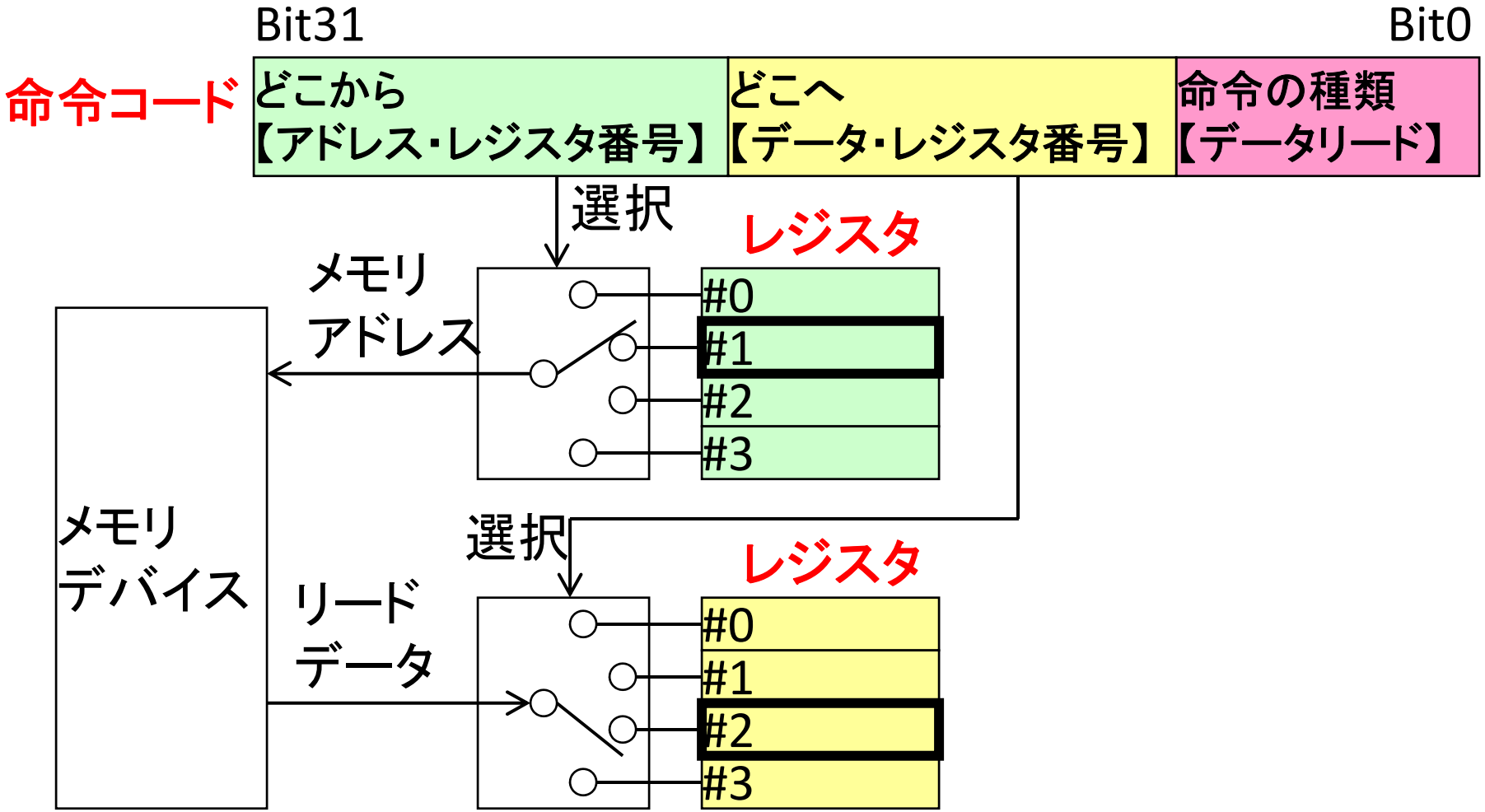


命令コードとレジスタ(1/3)

- 命令コード
  - CPUにさせる仕事の内容を0/1のビット列で表現したコード
    - ビット位置を効率的に区切り、命令の種類、レジスタの番号、イミディエート値などを定義している。
  - RISC CPUにおける主流は固定ビット長
    - 例えば、PowerPCは32bit長、ARMは32bitと16bit長
- レジスタ
  - CPU内部の高速アクセス可能な一時記憶場所
    - １サイクルでアクセス可能。
    - 複雑なプログラムに対応する為に、複数個のレジスタを搭載している。
    - CPUの演算処理はレジスタ間で行われる。

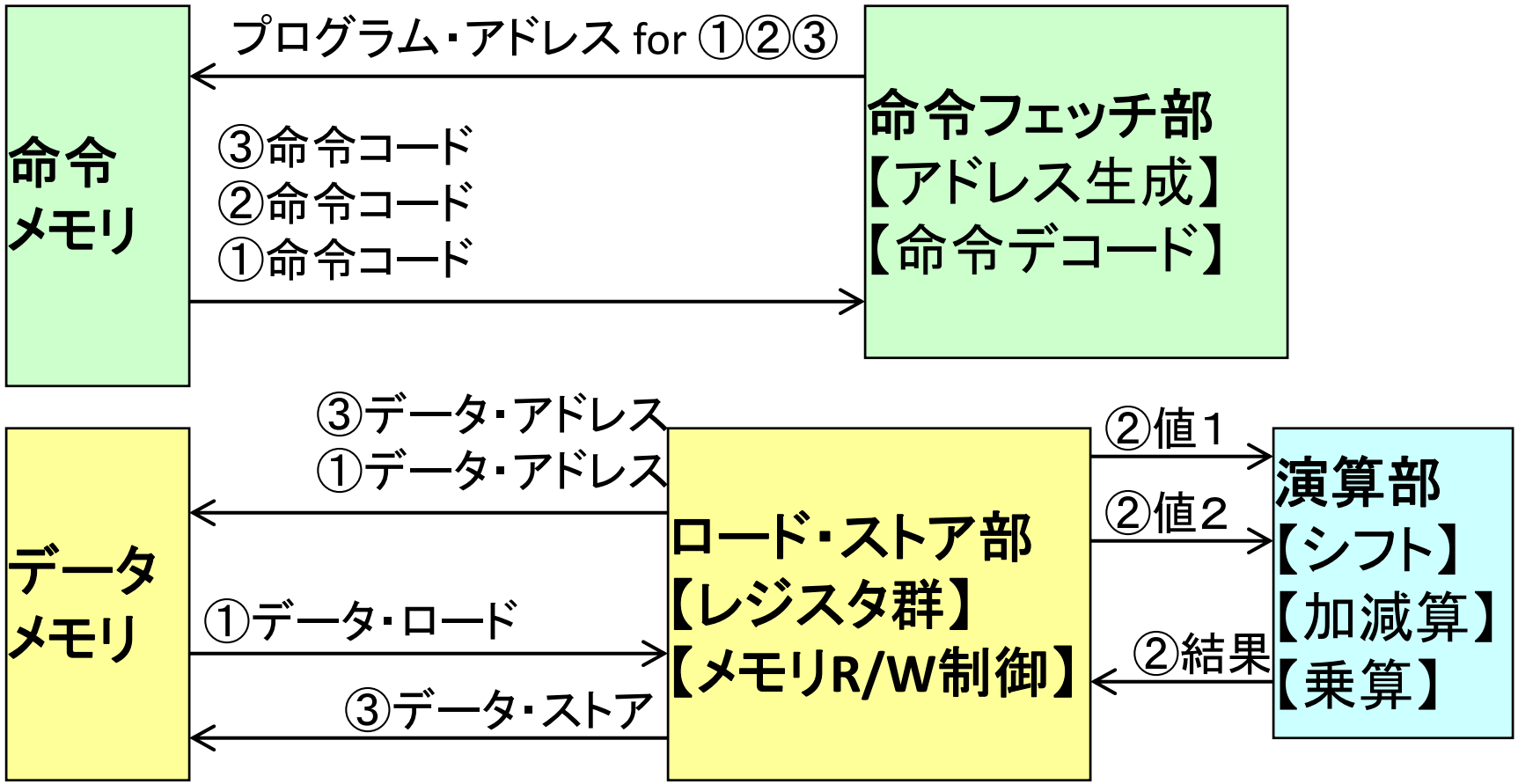
命令コードとレジスタ(2/3)

- メモリからデータを読み出す（ロード）命令の例



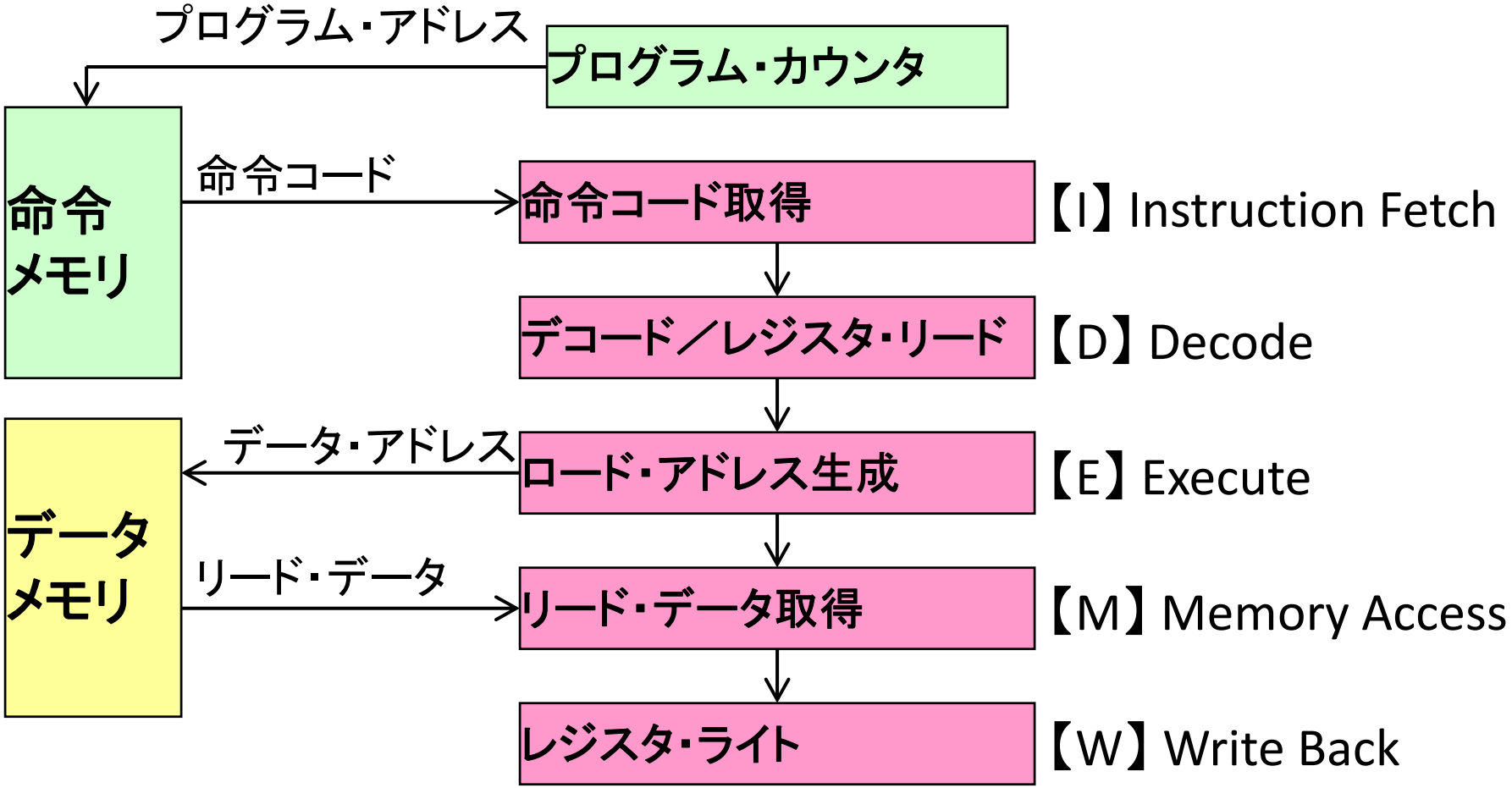
命令コードとレジスタ(3/3)

① 命令コード＝データ・ロード　：メモリからレジスタにロード
② 命令コード＝演算実行　：レジスタ間で演算
③ 命令コード＝データ・ストア　：レジスタからメモリへストア



パイプライン動作 (1/3)

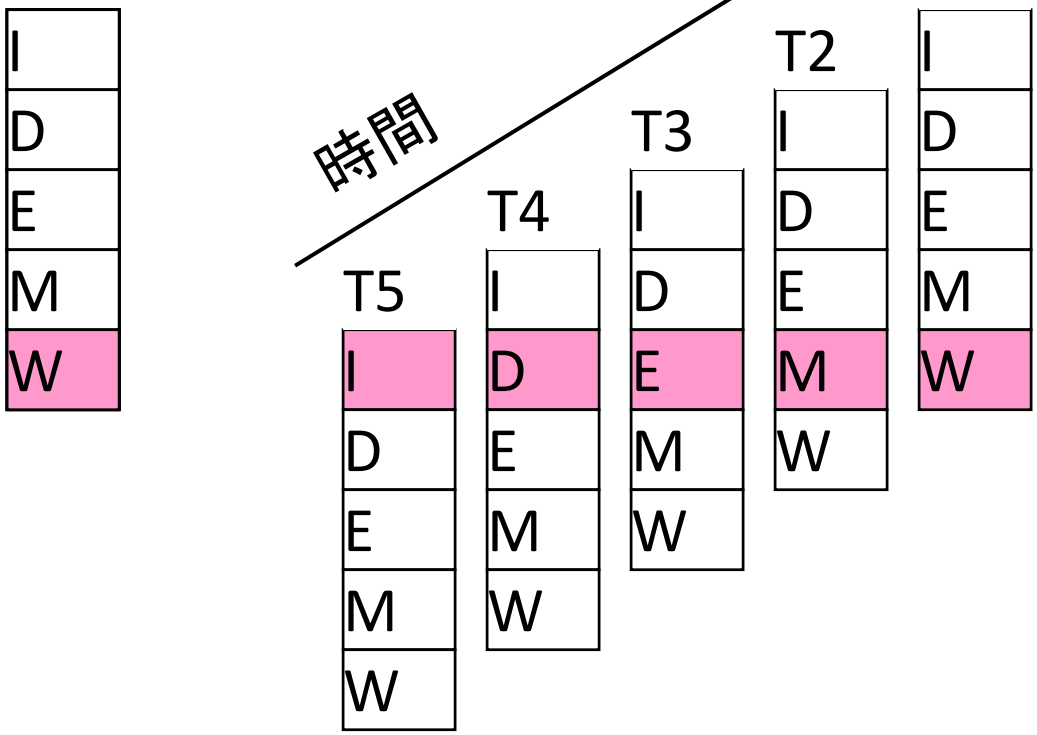
- ５段パイプラインにおける、ロード命令の例



パイプライン動作 (2/3)

- 各フェーズをパイプライン実行することで、見かけ上命令は1サイクルで実行される。

【I】Instruction Fetch
【D】Decode
【E】Execute
【M】Memory Access
【W】Write Back

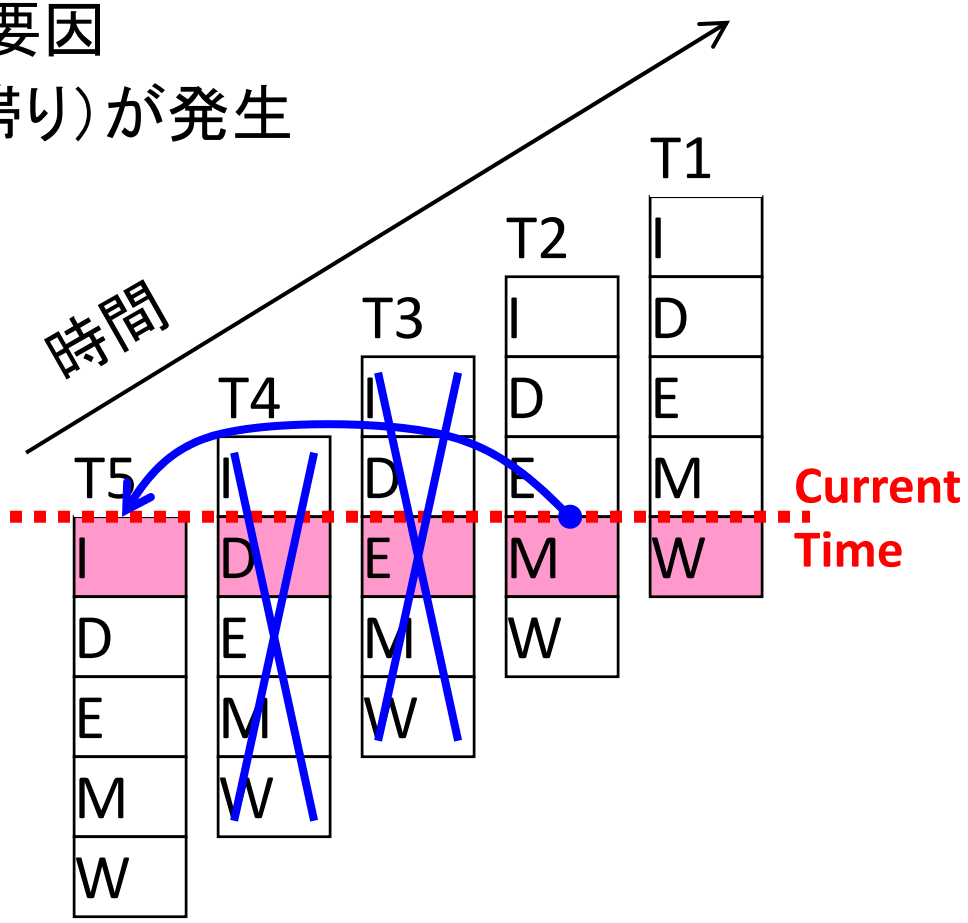


パイプライン動作 (3/3)

- パイプライン・ハザード
  - パイプライン動作を乱す要因
  - パイプライン・ストール（滞り）が発生

例）#2の実行結果（【E】完了）を受けて、#5の命令が実行（【I】開始）される場合。
→例えばIf文による分岐

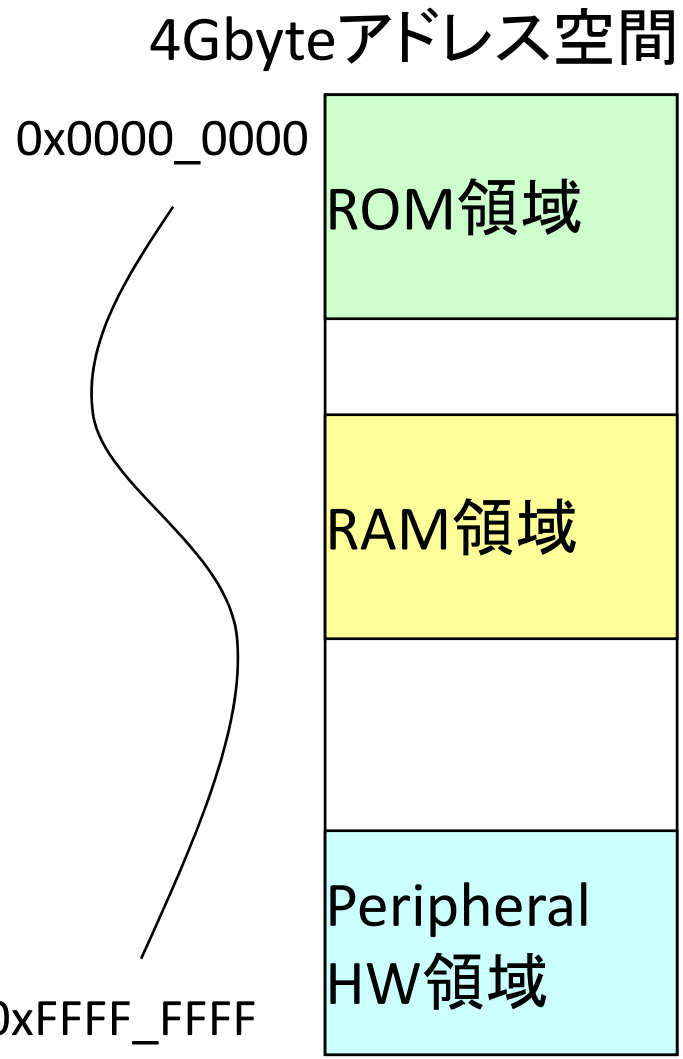
#3と#4の命令は無駄になる。
→命令実行効率が低下
→1命令／1cycleが崩れる



アドレッシング

- 32bit のアドレス幅を持つなら
  - 2^{32} = 4Gbyteのアドレス空間

- 主な割り当て対象は、
  - ROM領域 （不揮発）
    - ✓プログラムや定数データ
  - RAM領域 （揮発）
    - ✓実行時に書き換える変数
    - ✓(プログラムを置くこともある)
  - Peripheral HW領域
    - ✓CPUを補助するHWの領域

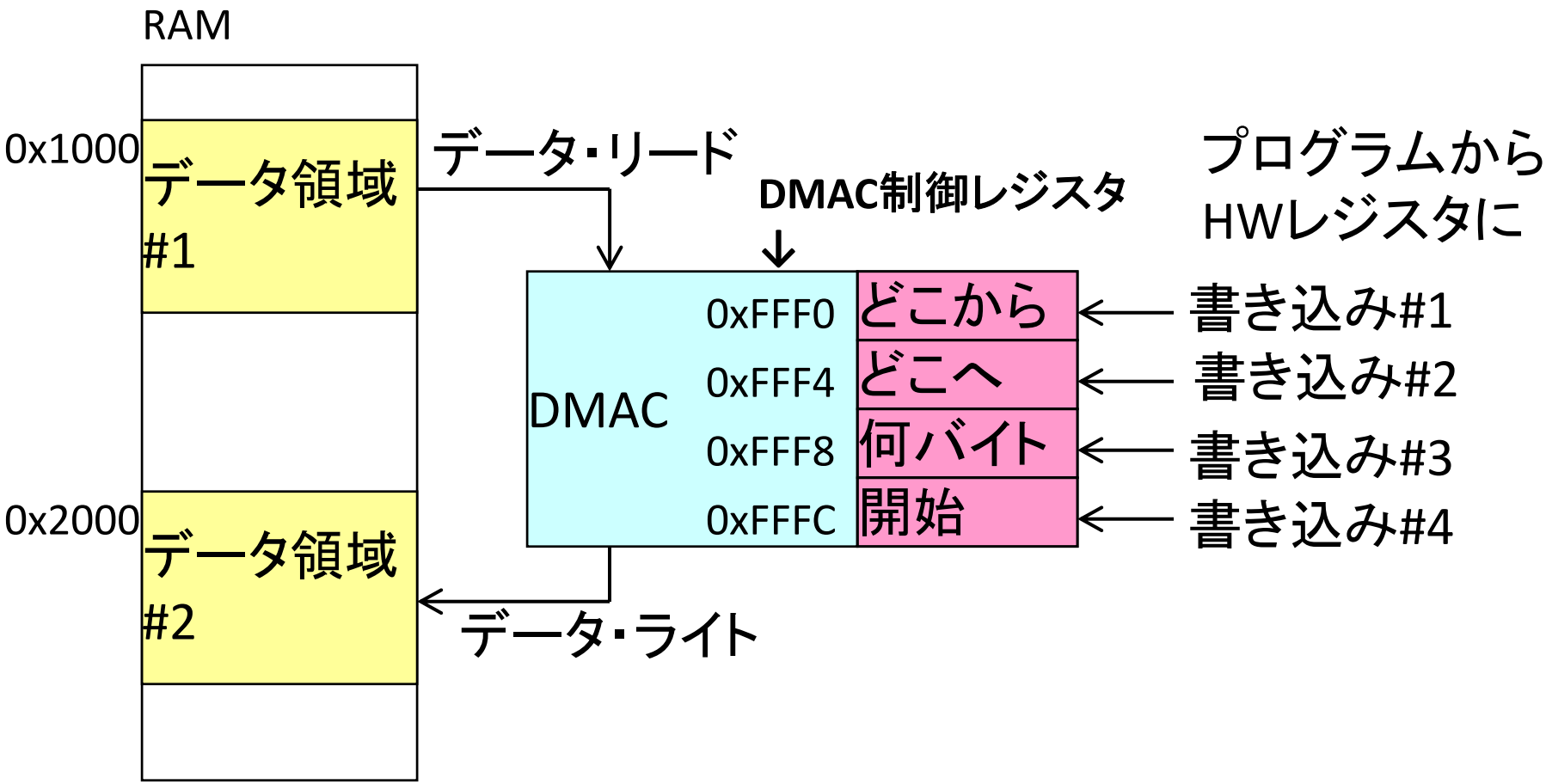


Peripheral Hardware (1/2)

- CPUが苦手な処理や、定型処理をHWが担当する。
  - プログラム（CPU）で実行するよりも高速。
  - 正確なリアルタイム（時間管理）処理が出来る。
  - HWが仕事をしている間、プログラムは別の仕事が出来る。
    - プログラムはHWが仕事を完了したことをどうやって知る？
- いろいろな仕様のHWが用意されていて、ワンチップ・マイコン選定の際の判断基準になる。
  - 実際は、マイコンが狙う市場によって仕様は似ている。
- 組み込みシステム設計では、Peripheral HWをいかに効率的に使うかがキーになる。
  - プログラミングの知識＋HWの知識が必要！

Peripheral Hardware (2/2)

- DMAC (Direct Memory Access Controller)の例



Agenda

1. CPUとPeripheral HWとプログラム
2. **組み込みシステム・プログラミングの特徴**
3. 開発環境
4. 簡単な応用例

ブート・アップ (1/3)

- システムの電源投入し、リセットを解除した後にCPUは何をするのか？
  - CPU内部もフリップ・フロップ、AND、ORといった論理ゲートで構成されている。
  - プログラム・カウンタ等のHWリソースは、ある特定の値に初期化される。
- キーワード
  - リセット・ベクタ
  - スタート・アップ・ルーチン

ブート・アップ (2/3)

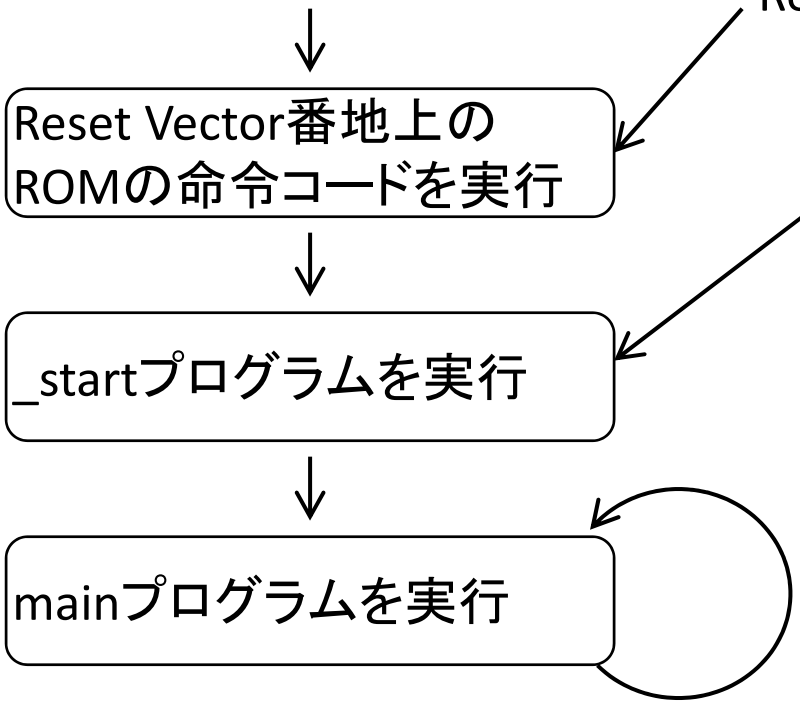
- リセット・ベクタ
  - CPUが一番最初に命令コードを取りに行くアドレスで、通常はROMの先頭番地をアサインする。
    - スタート・アップ・ルーチンへの無条件ジャンプ命令が格納されている。
- スタート・アップ・ルーチン
  - このルーチンで基本的な初期化を行う。
    - CPUやPeripheral HWを初期化。
    - C言語で記述したプログラムへジャンプする為のメモリの初期化。
  - main( ) 関数へジャンプする。

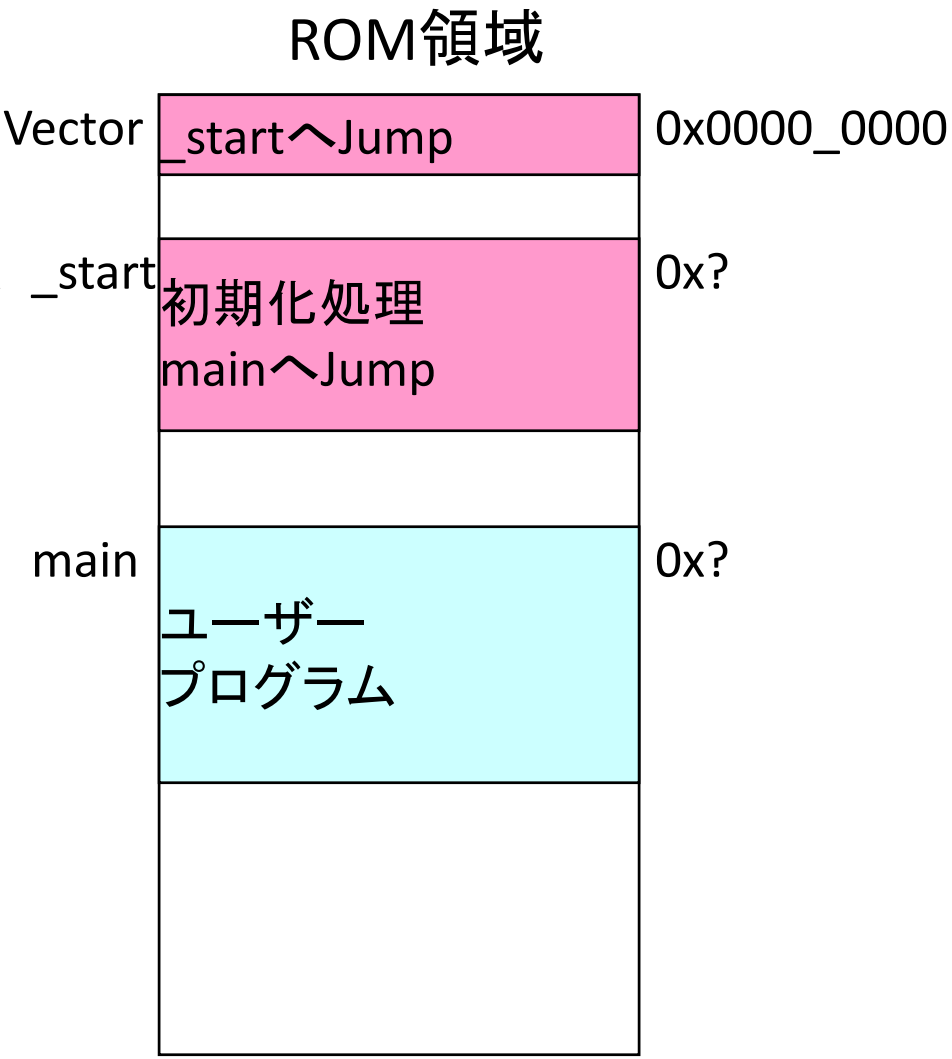
✓ crt0.S

- C Runtime Routine 0
- 一般的に、スタート・アップ・ルーチンを記述しているプログラム。
- ターゲットCPU専用のアセンブリ言語で記述する。

ブート・アップ (3/3)

電源オン
システム・リセット解除





main( )関数 (1/2)

- デスク･トップ･プログラムのmain( )関数
  - main( )関数実行時に引数を渡すことも可能。
  - 所望の処理が終わったらリターンする。
    - OSのプロンプトが表示される画面に戻る。
- 組み込みプログラムのmain( )関数
  - main( )関数実行時に引数を渡す？
    - 誰が何の為に？
  - 所望の処理が終わったらリターンする？
    - どこへリターンする？

main( )関数 (2/2)

```
int main(void){
    //普通に変数を定義
      unsigned int i,j,k;
    //いろいろな初期化処理
      i=j=k=0;
    //無限ループ
      do{
        //ファンクション1
        //ファンクション2
      }while(1)
    return(0);  //とりあえず書くけど実行されない
}
```

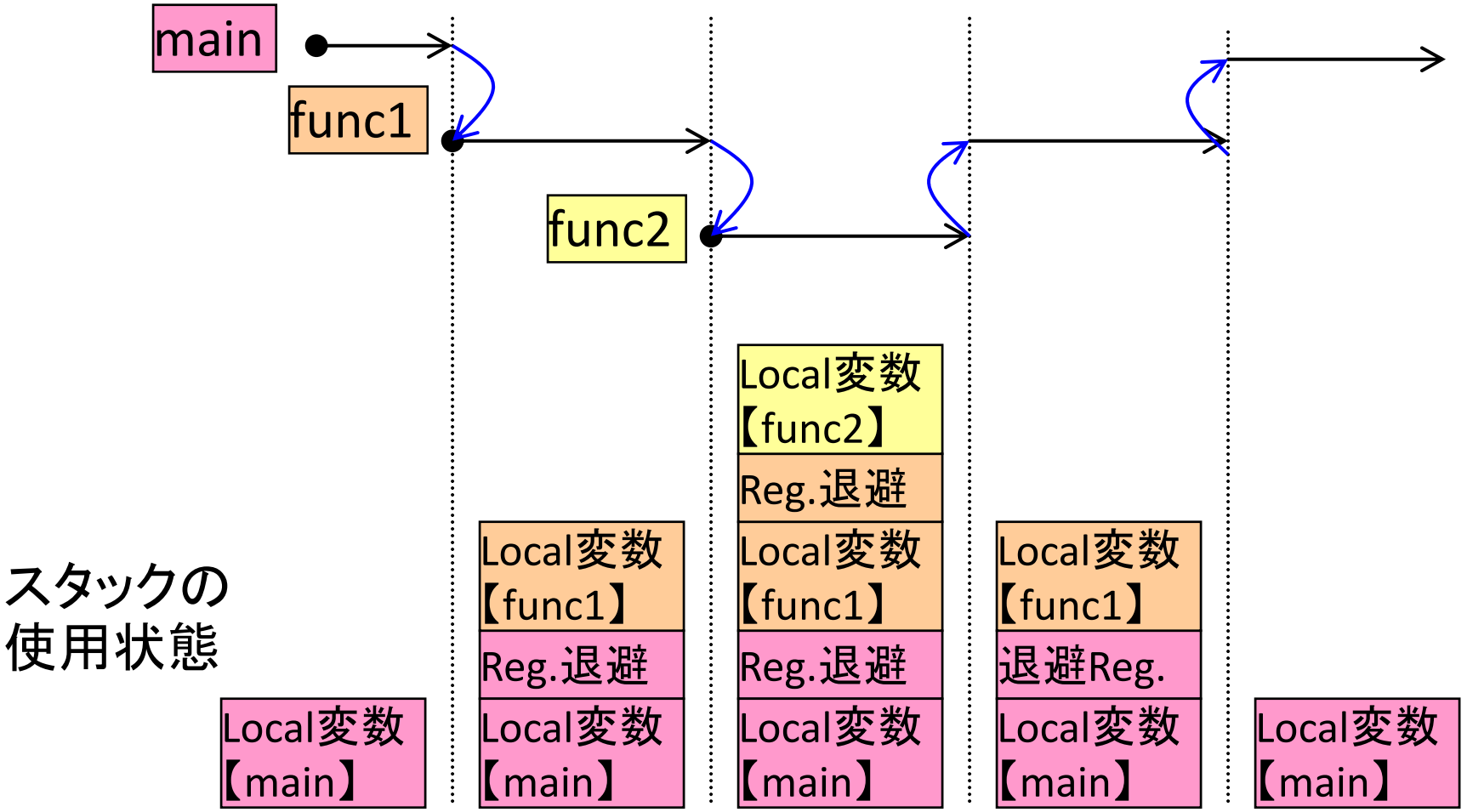
ファンクション・コールとスタック (1/3)

- ファンクションから別ファンクションをコールするのは日常茶飯事

```
int func1(<arg>, <arg>){
    //前処理
    val = func2(<arg>, <arg>); //ファンクション・コール
    //後処理
    return(val)
}
```

  - func1が「前処理」をした後、func1の実行を中断しfunc2が実行される。
  - func2の実行が終わると、func1の「後処理」を再開する。
- 中断したファンクションを、正しく再開するためには？
  - 途中経過を安全な場所に保存しておく場所が必要。
  - その保存場所が「スタック」と呼ばれるRAM上の領域。
- キーワード
  - CPUのレジスタ（プログラムカウンタ、CPUステータス、汎用レジスタ、etc）
  - 自動変数（ローカル変数）

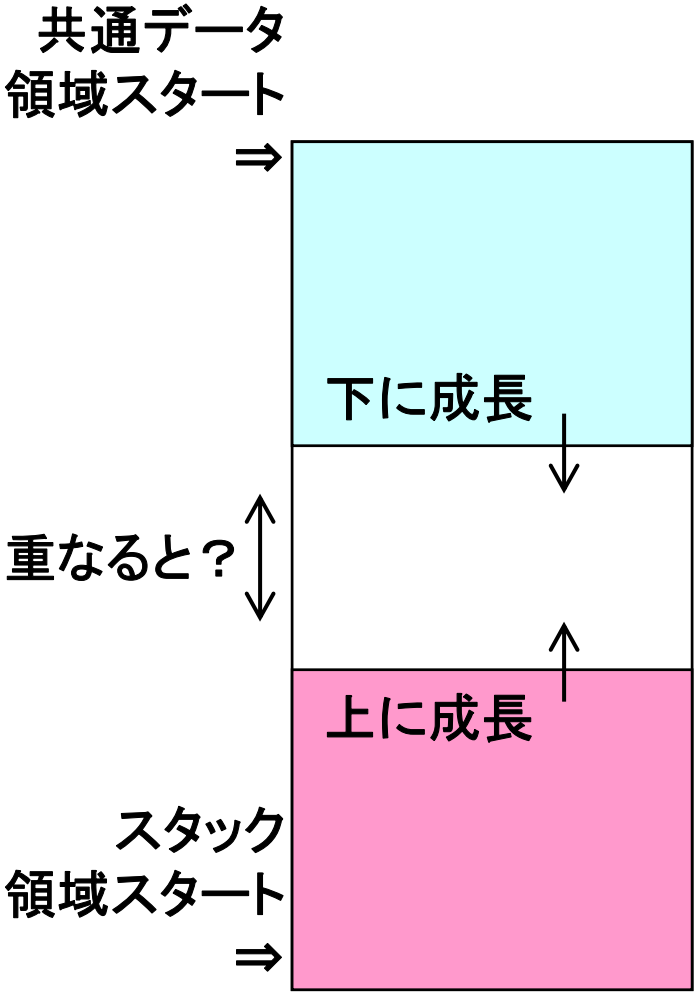
ファンクション・コールとスタック (2/3)



ファンクション・コールとスタック (3/3)

- スタック破壊の可能性
  - これが起きると暴走
  - デバッグが難しい
  - 避ける手段は？

- PCやLinuxの場合だと
  - OSが領域をプロテクト
  - Core Dumpして終了



ポーリングと割り込み (1/4)

- ポーリング
  - ユーザー・プログラム中で、事象の発生を定期的にチェックする。
    - 例えばPeripheral HWの動作状態をチェック。
  - プログラムの構造は簡単。
  - リアルタイム応答性を保つのが難しい。
    - 他の処理の影響を受けやすく、負荷が高くなるとチェックの定時性が乱れてリアルタイム応答性が失われる。
- 割り込み
  - ユーザー・プログラムを強制中断し、緊急性の高い処理を優先実行する。
  - プログラムの構造は難解になりがち。（慣れが必要）
  - リアルタイム応答性を確保しやすい。
    - ある事象の発生を、割り込み信号などの方法でCPUに通知する。
    - CPUはその信号を受け、所定の割り込み処理ルーチンを実行する。
    - 割り込み輻輳時の応答性能について注意深く設計しないと破綻する。

ポーリングと割り込み (2/4)

```
//ポーリングの例
do{

 h_sts = read_hw_status( );
 if(h_sts==TRUE){  //①
   <Do time critical job>
   <Do non-critical job>
 }

 f_sts = func1( );
 if(f_sts==TRUE){  //②
   func2( );
   func3( );
 }

}while(1)
```

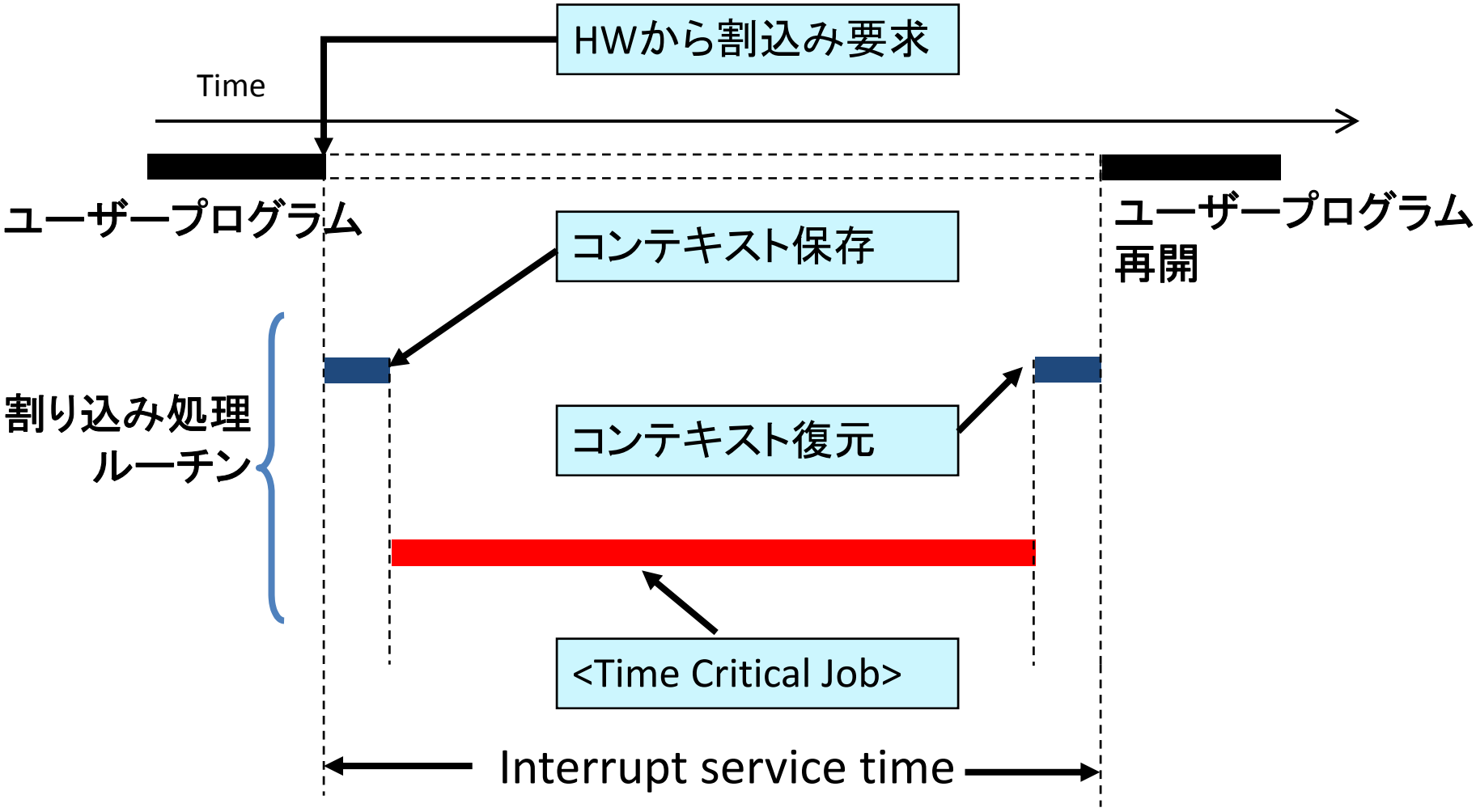
- ②のIF文が「FALSE」の間は、
  - ①の処理は定時性がある。
  - <Time critical job>は間に合う。
- ②のIF文が「TRUE」になると、
  - func2とfunc3が実行され、その処理が予想以上に重いと、
    - ①のリアルタイム性は崩れ、
    - <Time critical job>は破綻し、
    - システム・オーバーフローに至る可能性がある。

ポーリングと割り込み (3/4)

```
//割り込みを使用時の例
do{

 //h_sts = read_hw_status( );
 if(h_sts==TRUE){  //①
   //<Do time critical job>
   <Do non-critical job>
 }

 f_sts = func1( );
 if(f_sts==TRUE){  //②
   func2( );
   func3( );
 }

}while(1)
```

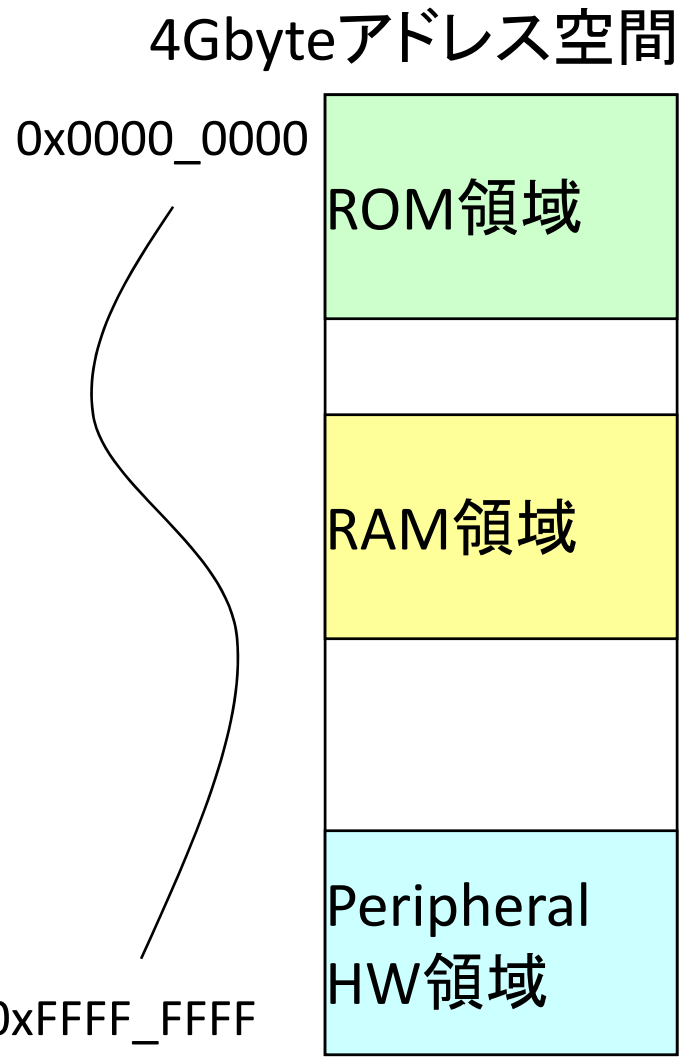
- 割り込み処理ルーチン
  - <Time critical job>を担当
  - 完了時にh_stsをTRUEに設定
- ①のIF文では、
  - <Non-critical job>を処理する。
- ②のIF文が「TRUE」でも、
  - 処理を中断し、割り込み処理ルーチンを優先的に実行することで、
  - <Time critical job>のリアルタイム性が保たれる。

ポーリングと割り込み (4/4)



メモリ・マップトI/O (1/2)

- CPUのアドレス空間が1枚（フラット）の時、
  - Peripheral HWは、その空間の一部に配置される。
  - メモリ・デバイス（ROM/RAM)も、その同じ空間の別アドレスに配置される。
- メモリと外部入出力（I/O）を担当するPeripheral HWが、アドレス空間的に区別が無い場合、
  - メモリ・マップトI/O方式と呼ぶ。



メモリ・マップトI/O (2/2)

- HWの制御レジスタにアクセスする場合、
  - 例えばDMACの制御レジスタが下記のようになっているのなら、
    - Source Address : 0xFFFFFFF0
    - Destination Address : 0xFFFFFFF4
    - Byte Length : 0xFFFFFFF8
    - Start : 0xFFFFFFFC
  - プログラムでも、そのアドレスをDefineし、

```
#define rDMA_SA          (*(volatile int *) 0xfffffff0)
#define rDMA_DA          (*(volatile int *) 0xfffffff4)
#define rDMA_BYTE        (*(volatile int *) 0xfffffff8)
#define rDMA_START       (*(volatile int *) 0xfffffffc)
```

  - それを使ってプログラムを書く。

```
rDMA_SA = &(rx_buf[0]);
rDMA_DA = &(tx_buf[0]);
rDMA_BYTE =  sizeof(rx_buf);
rDMA_START |= 1;
```

プログラムの移植性 (1/2)

- プログラムがシステム依存となる主な要因
  - ROM、RAM、Peripheral HWなどのアドレス配置
  - Peripheral HWの制御方法
  - CPUの割り込みアーキテクチャ
  - スタートアップ・ルーチン
  - 標準ライブラリ
  - ダミーの時間稼ぎループ
  - What else ?

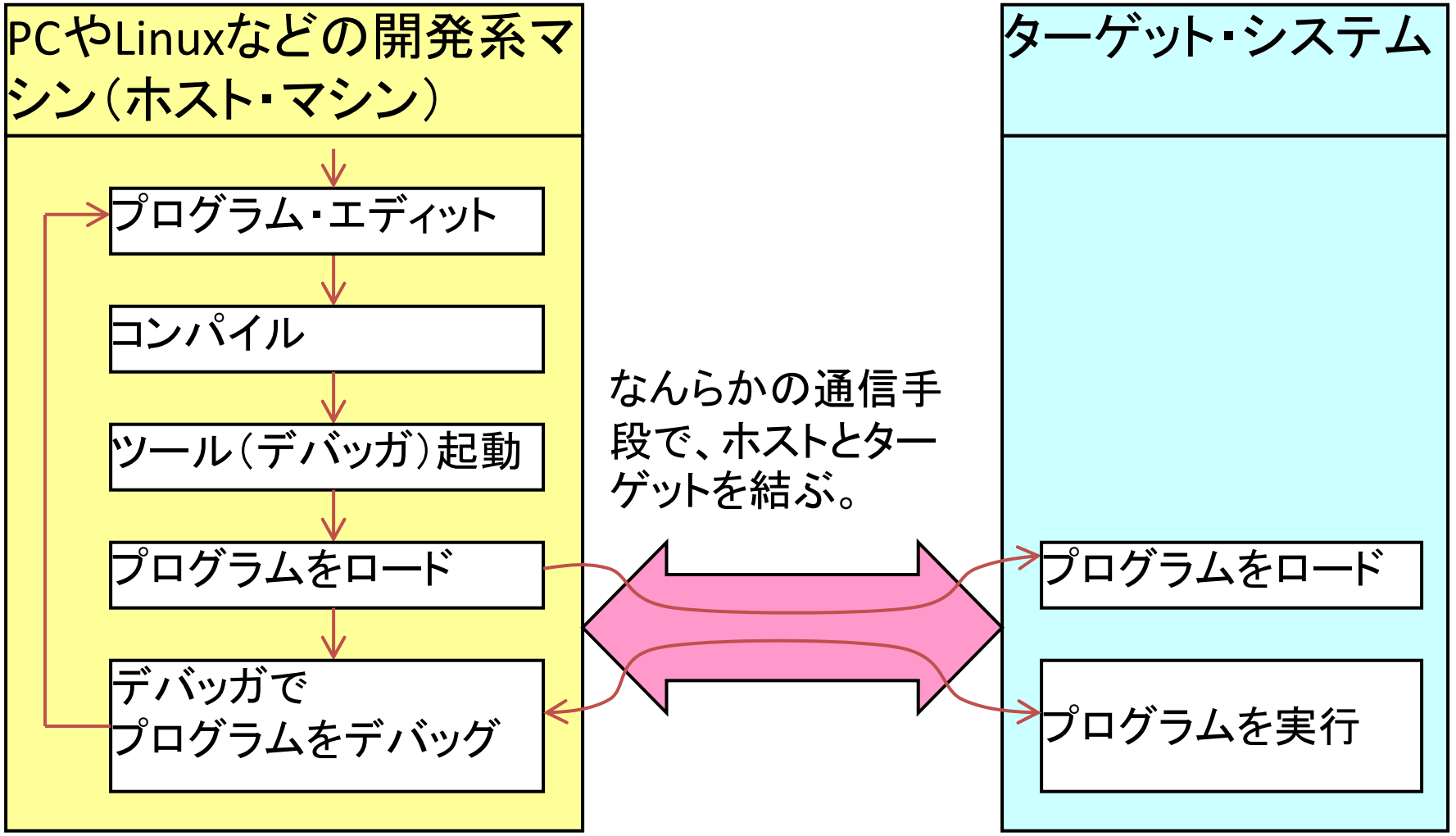
プログラムの移植性 (2/2)

- 機器依存となる要因を理解し局所化する。
  - 何が機器依存かを見極めるのは時に困難で、手間もかかる。
  - 将来機器間移植をするかどうかは不透明な場合が多い。
  - とは言っても、メンテナンスのことを考えるとやっていたほうがいい。
- 機器に組み込む前に、PCやLinuxマシン上でシミュレーションする時もある。（ある意味で移植を考えたプログラミング）
  - それを実現するコードを、スタブ・コードと呼ぶ。
  - スタブ・コード上に、システム依存部を押し込む。
  - 関数単体テストや、一部機能を切り出したテストなどでは使える。（全体となると難しい）

Agenda

1. CPUとPeripheral HWとプログラム
2. 組み込みシステム・プログラミングの特徴
3. **開発環境**
4. 簡単な応用例

クロス開発環境

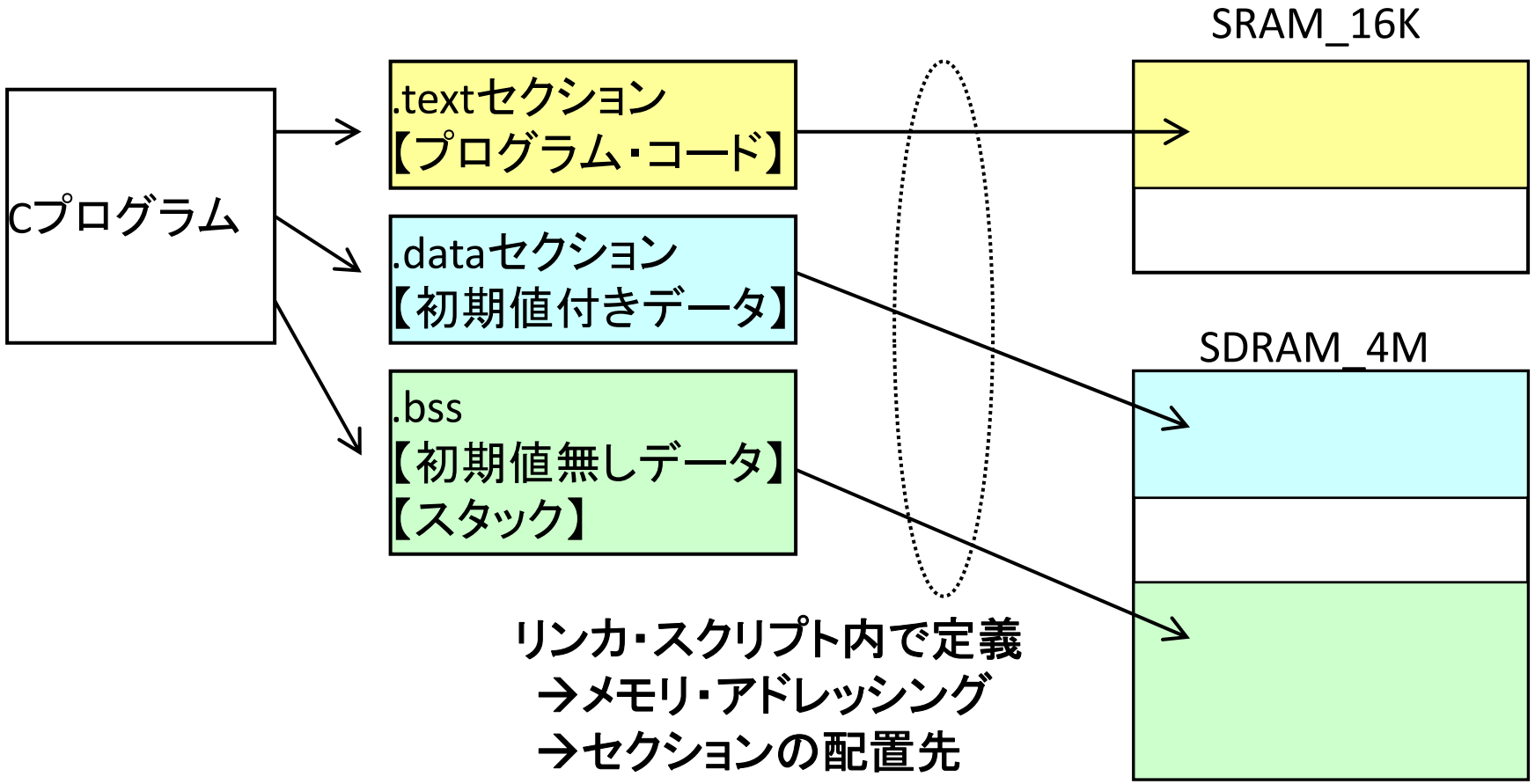

PCやLinuxなどの開発系マシン（ホスト・マシン）
プログラム・エディット
コンパイル
ツール（デバッガ）起動
プログラムをロード
デバッガで
プログラムをデバッグ
なんらかの通信手段で、ホストとターゲットを結ぶ。
ターゲット・システム
プログラムをロード
プログラムを実行

Cコンパイラ (1/3)

- ASCIIテキストで記述されたCプログラムを、CPU専用のマシン後に変換するツール。
- Cコンパイラと1言で呼ばれるが4つの段階がある。
  - **プリプロセス**
    - コンパイルの前処理 （#include、#define、#ifdef等を解決）
  - **コンパイル**
    - Cコード（Cプログラム）をアセンブラ・コードに変換
  - **アセンブル**
    - アセンブラ・コードをオブジェクト・コードに変換
    - オブジェクト・コードはマシン語に近いが、まだCPUが解釈できない中間的なコード。リンクに必要な情報が含まれている。
  - **リンク**
    - 全てのコードを結合
    - ターゲット・システムのメモリ・マッピングに合わせてコードを配置
    - マシン語の実行ファイルを生成

Cコンパイラ (2/3)

- リンクについてもう少し
  - ツール名としては、「リンカ」と呼ばれる。



Cコンパイラ (3/3)

- リンカ･スクリプトの非常に簡単な例

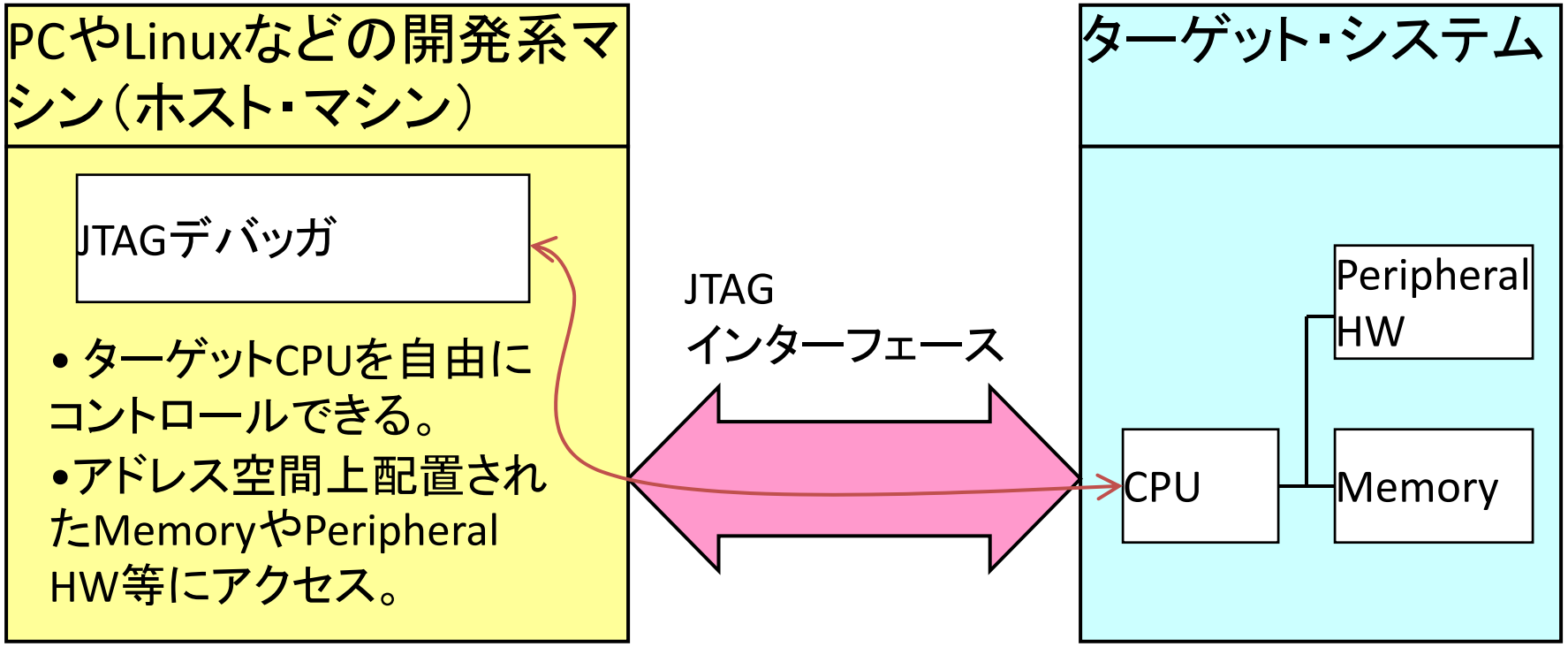
```
MEMORY
{
        SRAM_16K : ORIGIN = 0x20000000, LENGTH = 0x00001FFF
        SDRAM_4M : ORIGIN = 0x40000000, LENGTH = 0x003FFFFF
}
SECTIONS
{
        .text : { *(.text) } > SRAM_16K
        .data : { *(.data) } > SDRAM_4M
        .bss  : { *(.bss) *(COMMON) } > SDRAM_4M
}
```

デバッガ (1/3)

- プログラムをデバッグする為のツール。
- 基本的な機能
  - ブレイク・ポイント
    - プログラムを所望の場所で止める。
  - ステップ実行
    - プログラムを1行づつ実行する。
  - アドレス空間参照
    - データ、CPUのレジスタ、Peripheral HWのレジスタ等のメモリマッピングされた場所の値を参照できる。
  - 他にも非常に多くの機能がある。
- プログラムを一旦止めてデバッグするのが基本。
  - CPUとは独立して動作している部分がある場合には要注意。
    - 例えば、プログラムは止まってもPeripheral HWは止まらない。

デバッガ (2/3)

- 組み込みシステム向けはJTAGデバッガが主流。
  - JTAGインターフェースで、ホストとターゲットを接続する。
  - よく、JTAG-ICEと呼ばれる。（ICE = In-Circuit Emulator）



デバッガ (3/3)

- ブレイク・ポイントについてもう少し
  - インストラクション・ブレイク
    - 通常、ブレイク・ポイント設定といったらこれ。
    - 本来の命令コードを「ブレイク命令」に書き換える。
    - デバッガが勝手にやるのでユーザーからは見えない。
    - ROMの場合どうする？
  - データ・ブレイク
    - 命令コードでは無くデータなのでどうやって実現する？
    - あるアドレス空間へのデータ・リード／ライトを、どうやって検出する？
- ハードウェア・ブレイク・ポイント
  - CPUが専用のレジスタを具備している場合がある。
    - この特殊用途レジスタは、CPUのHWの一部と考えてこのように呼ぶ。
  - ブレイク・アドレスをそのレジスタに指定する方式。
    - 命令コード書き換えでは無いので、ROM上のプログラムにも適用可能。
    - データ・リード／ライトにも適用可能。
  - レジスタ数には限りがあるので、設定する数が制限され自由度は劣る。

その他のデバッグ補助ツール

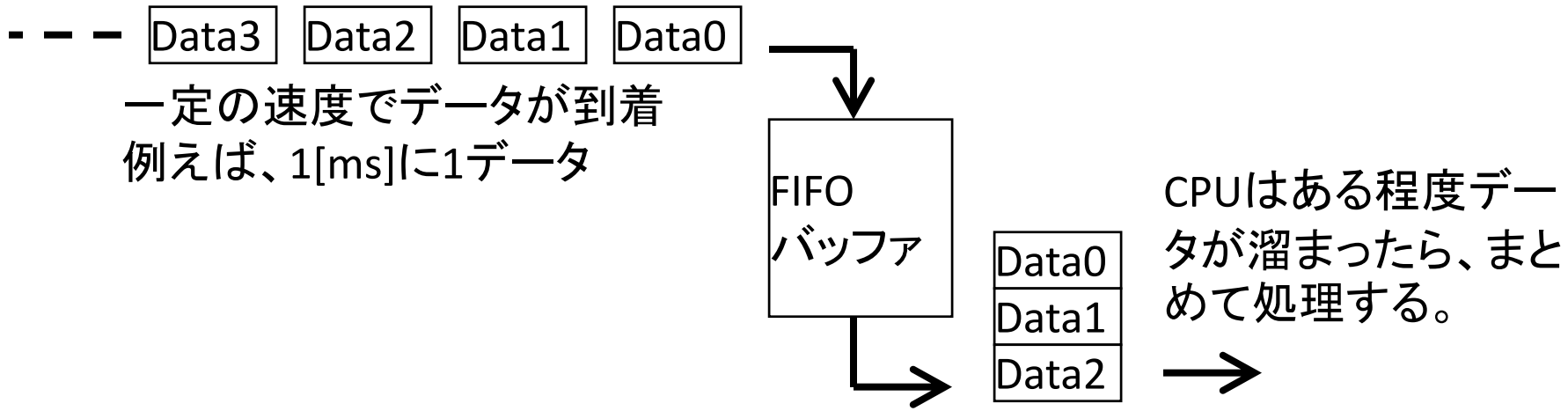
- print関数
  - 古典的だがやっぱり有効な手段。
  - プログラム（CPU）を止めなくてもいい。
  - 意外とコードサイズが大きいので注意。
  - 組み込みシステムの場合の出力先は？
- モニタ・プログラム
  - 本来のアプリケーションと一緒に実装してしまうデバッグ用プログラム。
  - HyperTerminal等を利用しRS232C等でホストと通信して、ターゲット・システム内部の状態を参照／設定する。
  - プログラム（CPU）を止めなくてもいいが、多少CPUの負荷が増える。
- ロジック・アナライザ
  - 信号線の1/0の状態を観測する測定器。
  - プログラムの状態をチップの外部に出力できれば使える。
  - 外部ポートへの出力なのでCPUの負担も小さい。

Agenda

1. CPUとPeripheral HWとプログラム
2. 組み込みシステム・プログラミングの特徴
3. 開発環境
4. **簡単な応用例**

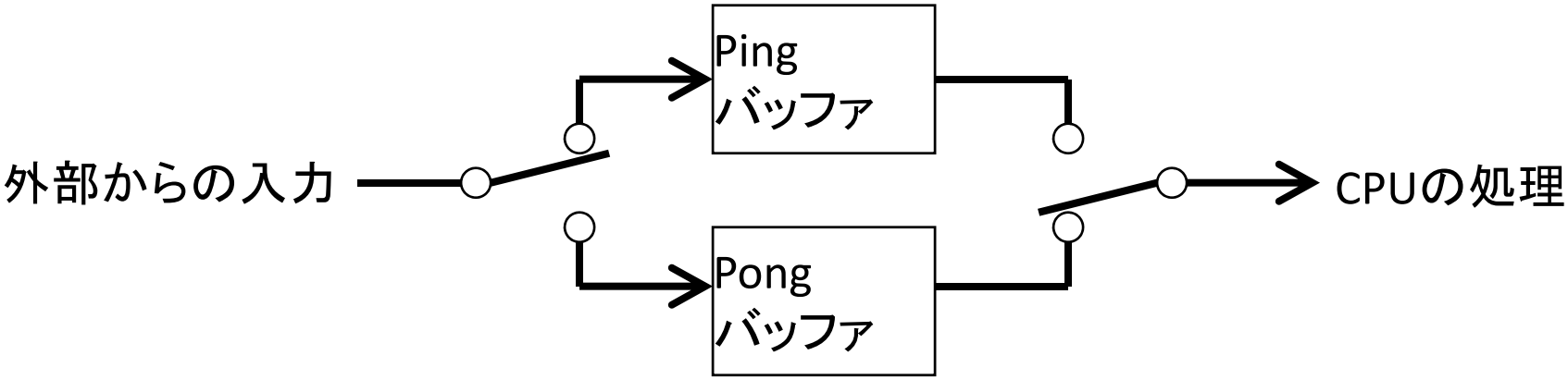
基本的なバッファ管理 (1/3)

- FIFOバッファ
  - FIFO = First In First Out
  - マイコンの外部からのデータ入力のスピードと、CPUの処理スピードのミスマッチを解消する時に有効。
    - 入力レートは一定だが、CPUの処理速度はバラつきがある場合。
  - 他にも使い道はいろいろあるが、各種のミスマッチ解消用のバッファ。

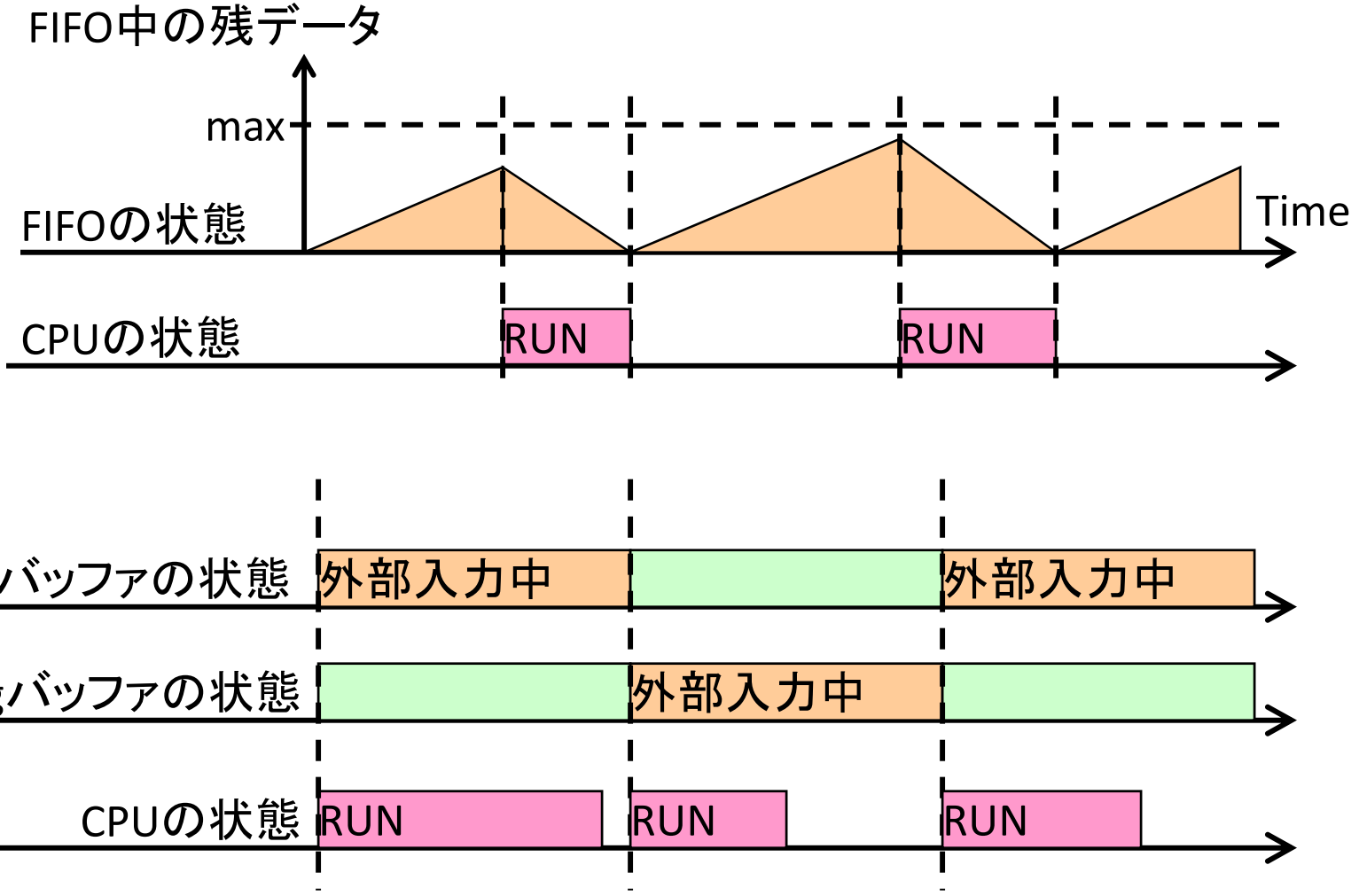


基本的なバッファ管理 (2/3)

- Ping-Pongバッファ
  - 2つのバッファを交互に使う。
  - データ入力と、データ処理をオーバーラップさせるときに有効。
  - FIFOと似たような使い方。
    - 入力と処理のミスマッチ解消。
  - メモリを2倍持つことになる。



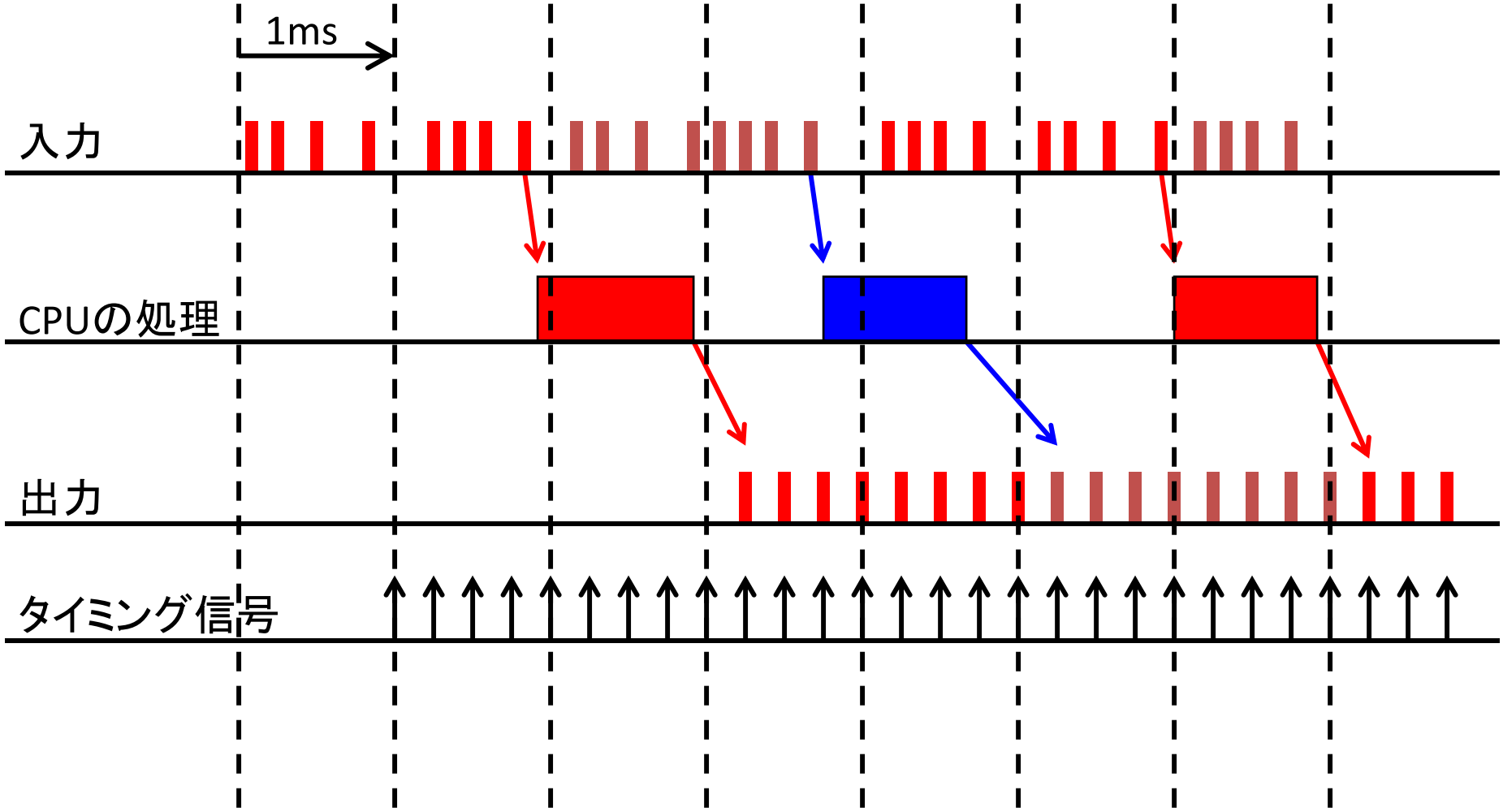
基本的なバッファ管理 (3/3)



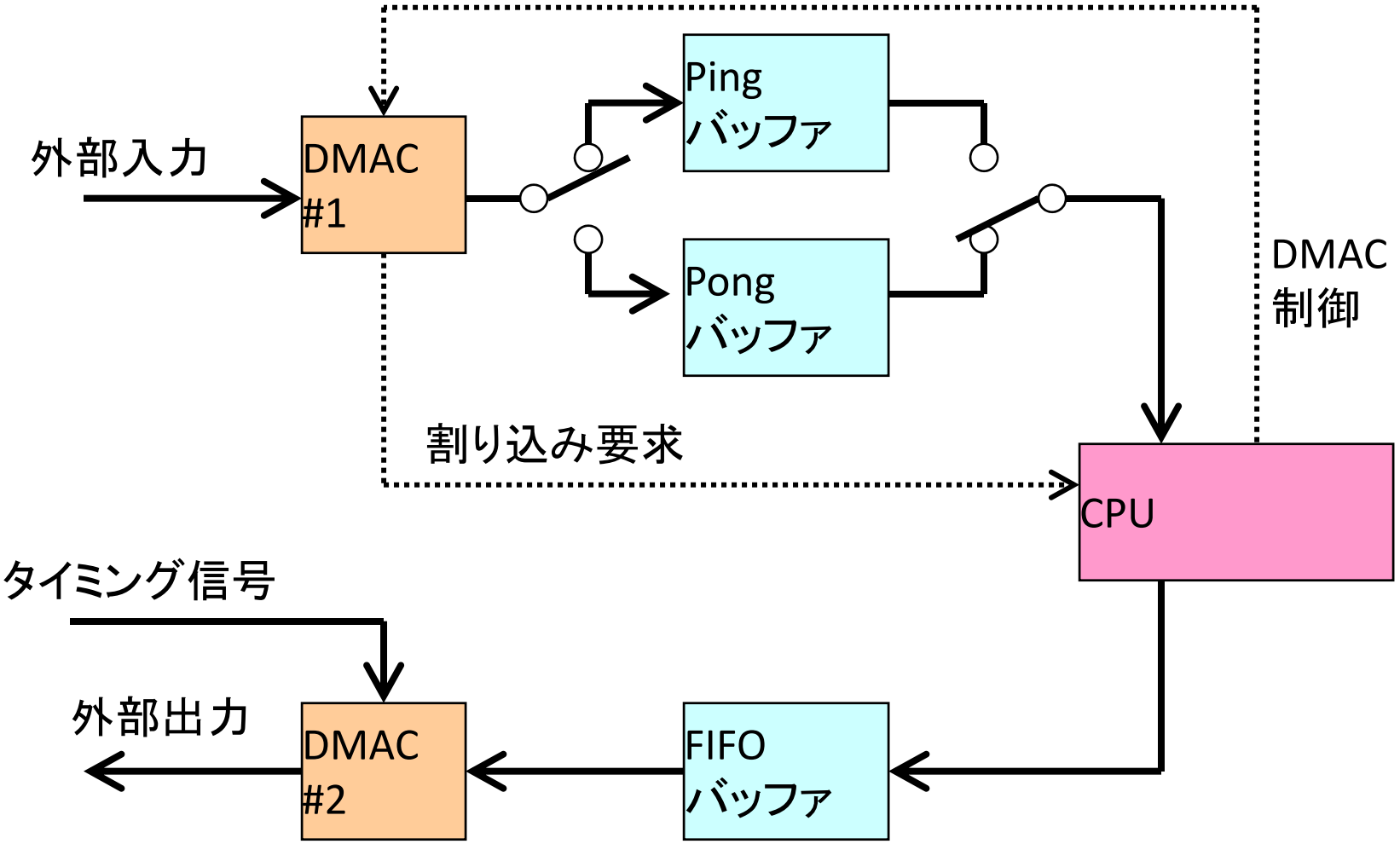
応用問題1

- 入力データ
  - 1[ms]の間に1byte単位で4byteのデータが到着する。
  - 1byte毎の間隔は一定ではない。
    - 4byte/1[ms]は確定、1byte/0.25[ms]は不確定。
- CPUの処理
  - １回の処理には8byteのデータが必要。
  - 8byteのデータをランダムにアクセスして計算する。
  - 出力は8byte。
- 出力データ
  - タイミング信号に同期して一定の間隔で出力する。
  - 出力単位は1byte。

応用問題1



応用問題1


外部入力
DMAC
#1
Ping
バッファ
Pong
バッファ
DMAC
制御
割り込み要求
CPU
タイミング信号
外部出力
DMAC
#2
FIFO
バッファ

HW1

- 以下の課題に対して回答レポートを作成せよ
- 採点基準（各問に対してA4 1ページ程度のレポートを作成せよ）
  1. 入力整数Nに対して、階乗N！を計算するプログラムをC言語にて作成せよ。そして、N＝３の時のSTACKの使用され方を説明せよ。
  2. CPU周辺のある処理ハードウエア(ディジタル回路)がそのデータをCPUに対して転送するときに、割り込みを用いて行う方法を適当なコンピュータを例に取り動作を説明せよ。
  3. ハードウエアインターラプトとソフトウエア・インターラプトの違いを説明せよ。
  4. 適当なCPUを調査し、そのコンピュータのDMACの使い方、機能を説明せよ。