FPGA＋SoC+Linux 実践勉強会

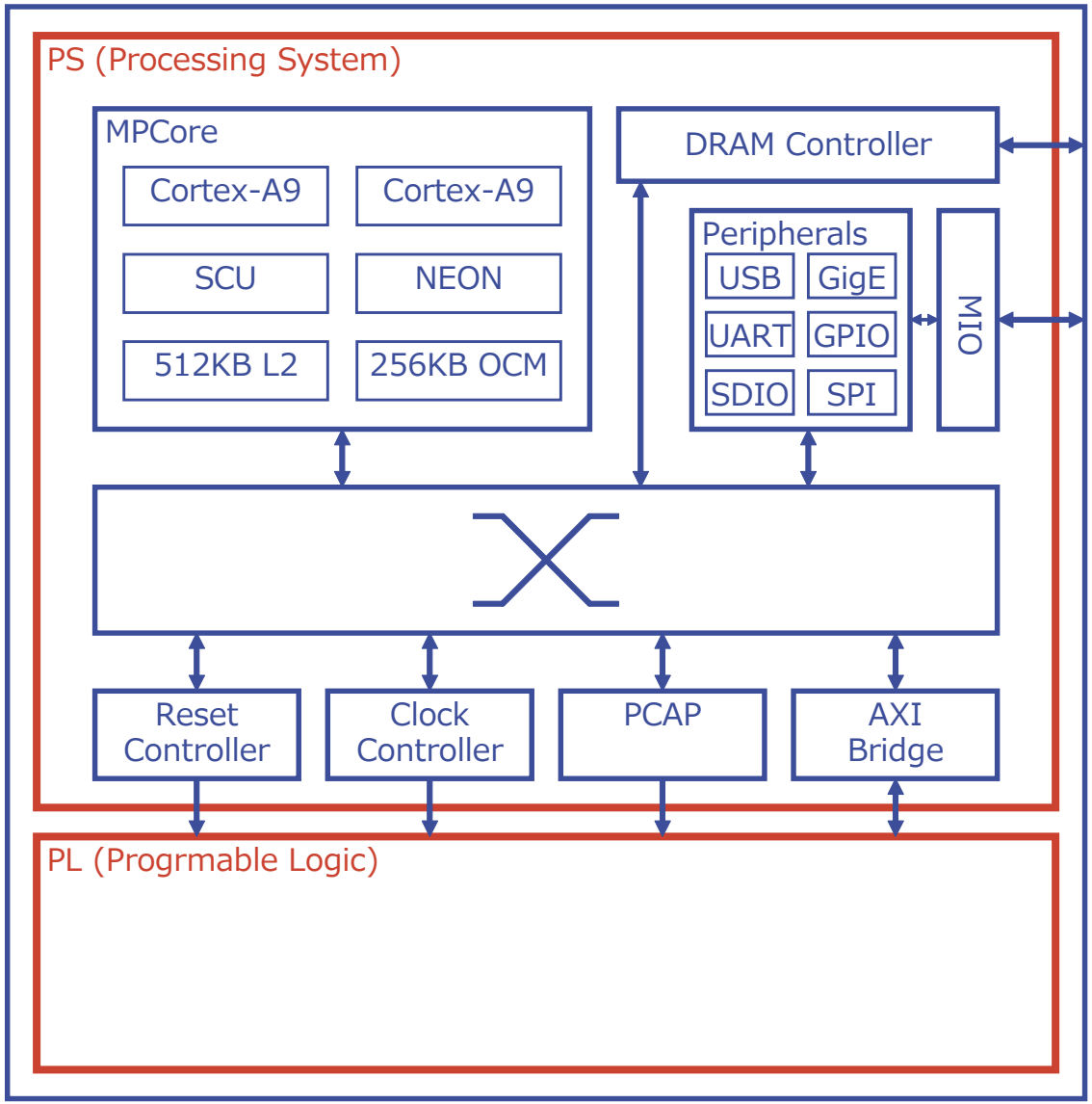
2017 年 12 月 2 日

@ikwzm

SoC ? FPGA ?

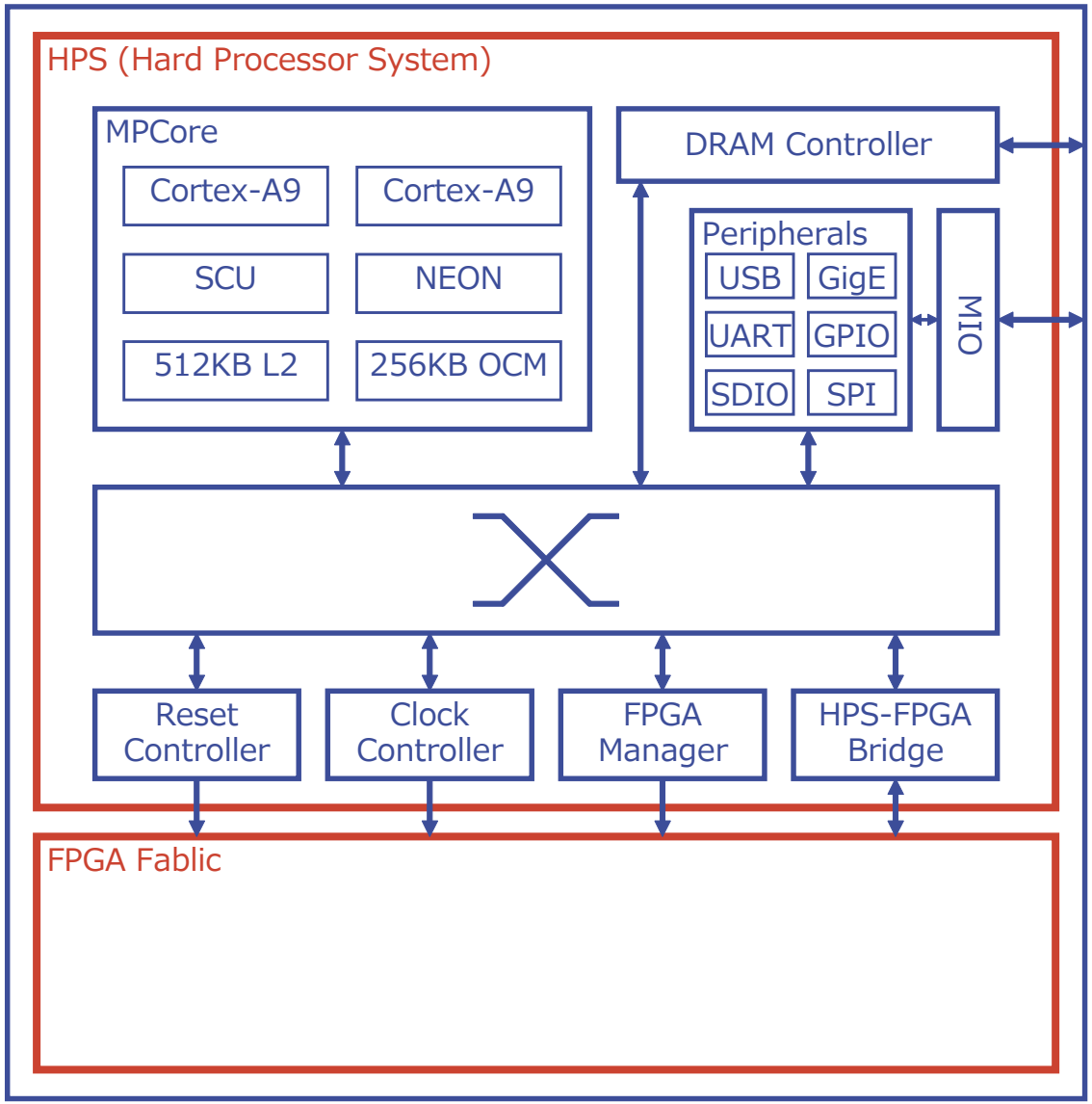
- SoC : System on Chip
  - CPU Core + Peripherals
  - Xilinx → PS (Processing System)
  - Altera → HPS (Hard Processor System)
- FPGA : Field Programmable Gate Array
  - Xilinx → PL (Programmable Logic)
  - Altera → FPGA Fablic

FPGA+SoC ? (Xilinx Zynq)



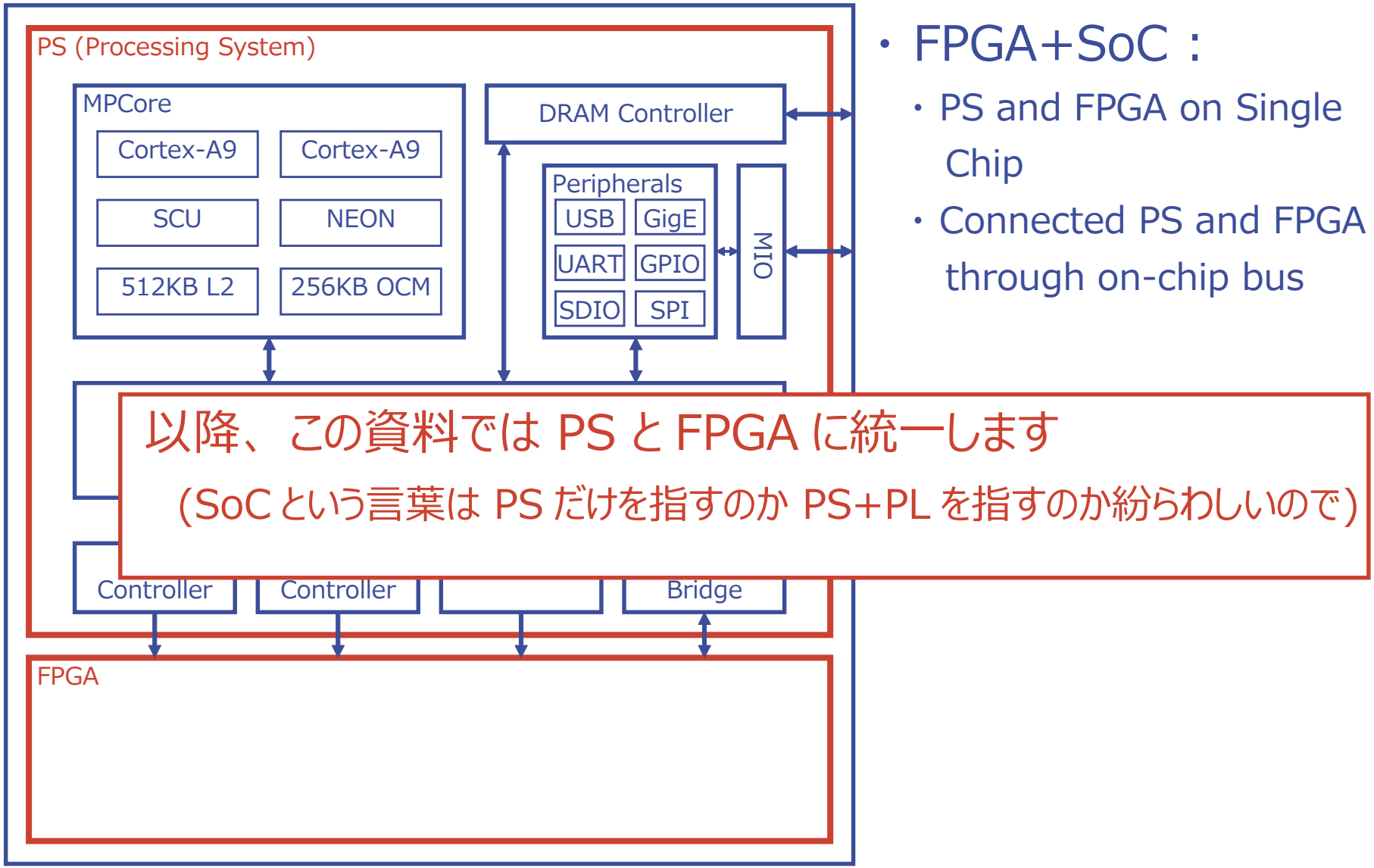
- FPGA+SoC :
  - PS and PL on Single Chip
  - Connected PS and PL through on-chip bus

FPGA+SoC ? (Altera CycloneV-SoC)



- FPGA+SoC :
  - HPS and FPGA Fablic on Single Chip
  - Connected HPS and FPGA through on-chip bus

FPGA+SoC ?



- FPGA+SoC：
  - PS and FPGA on Single Chip
  - Connected PS and FPGA through on-chip bus

以降、この資料では PS と FPGA に統一します

(SoC という言葉は PS だけを指すのか PS+PL を指すのか紛らわしいので)

Agenda

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

without device driver

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

∧
without
device driver

FPGA Configuration に必要な基本要素

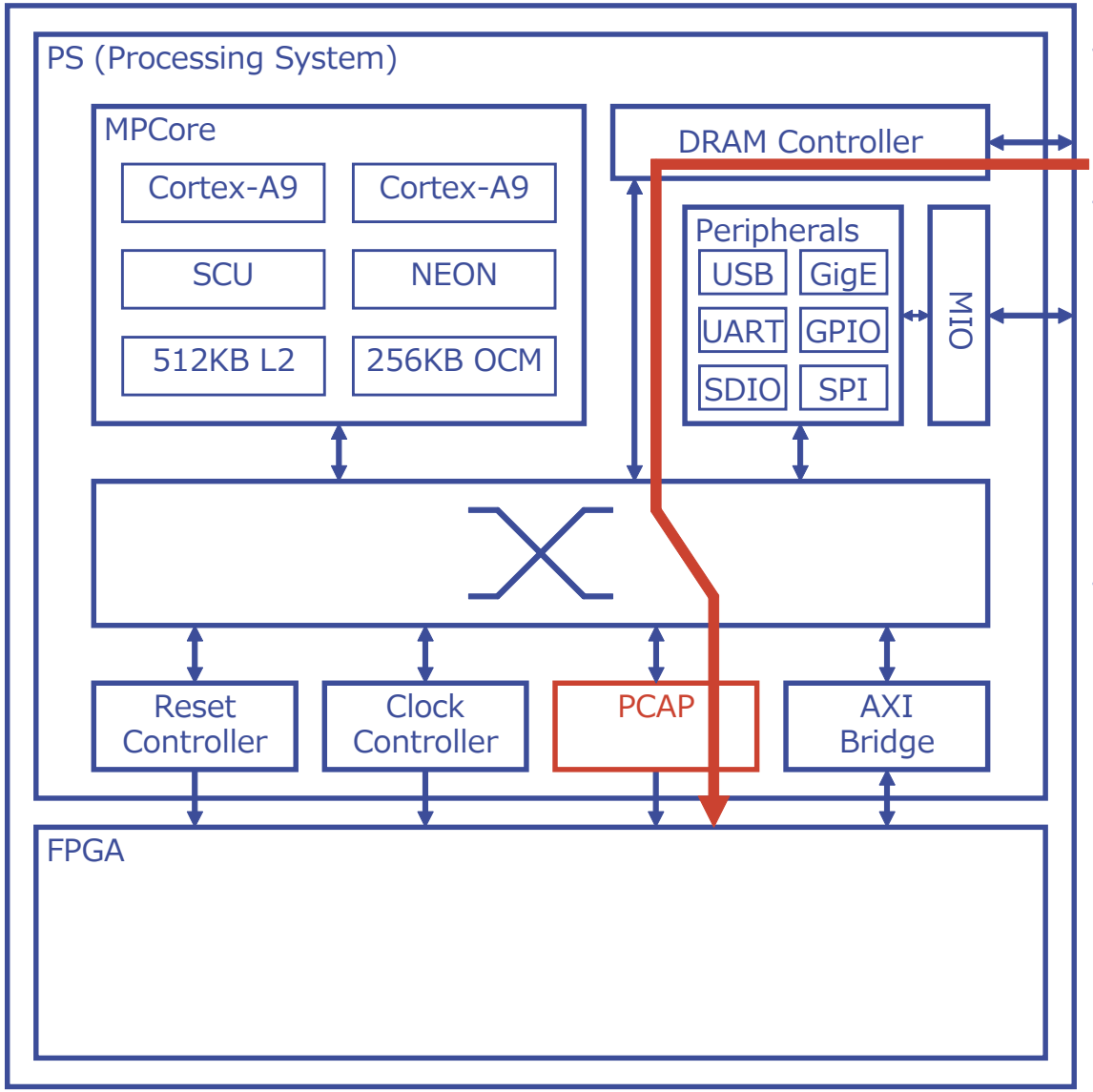
(1) Cofiguration Hardware

(2) Reset

(3) Bridge

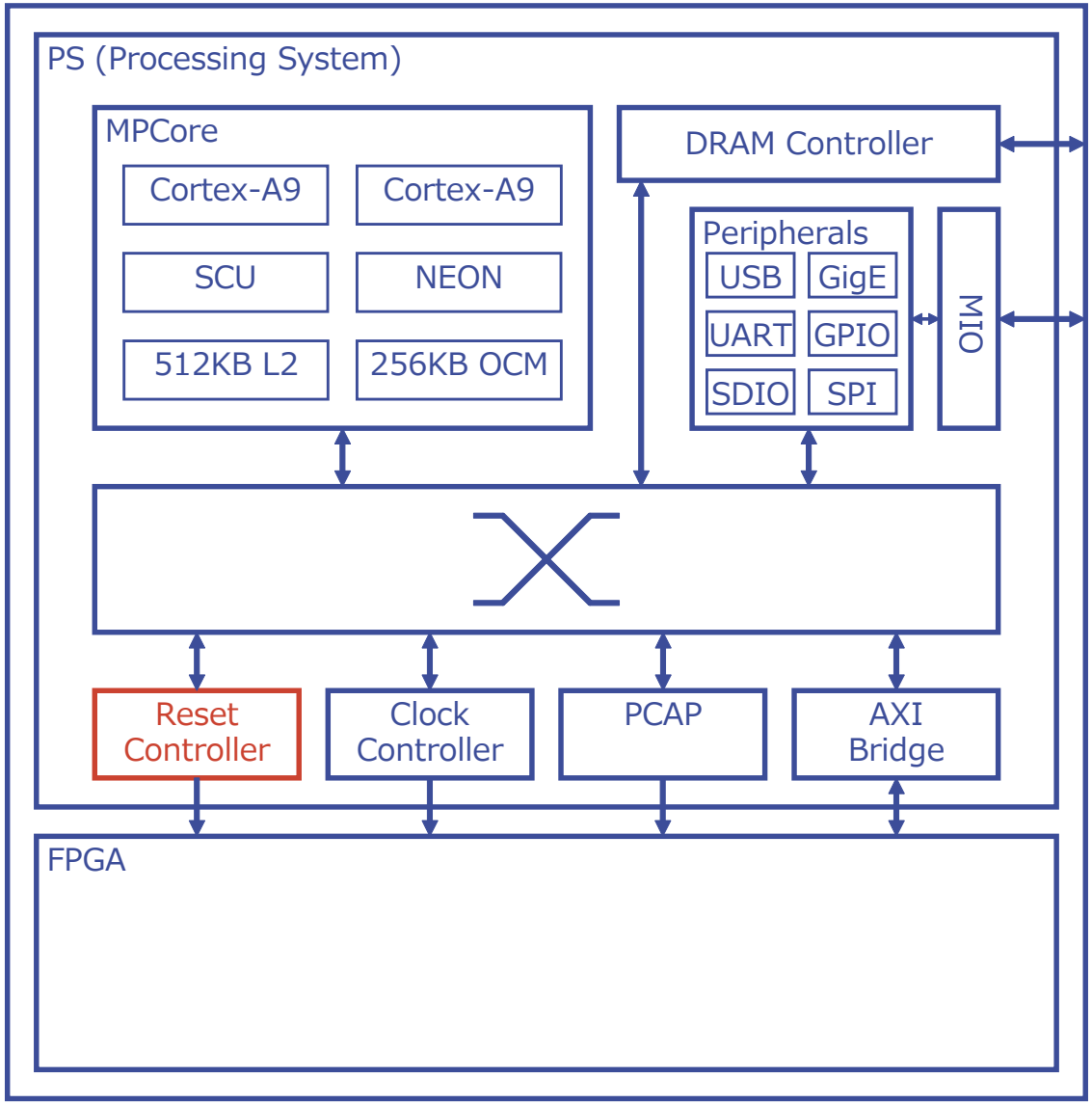
(4) Clock

FPGA Configuration Hardware (1) - Cofiguration H/W



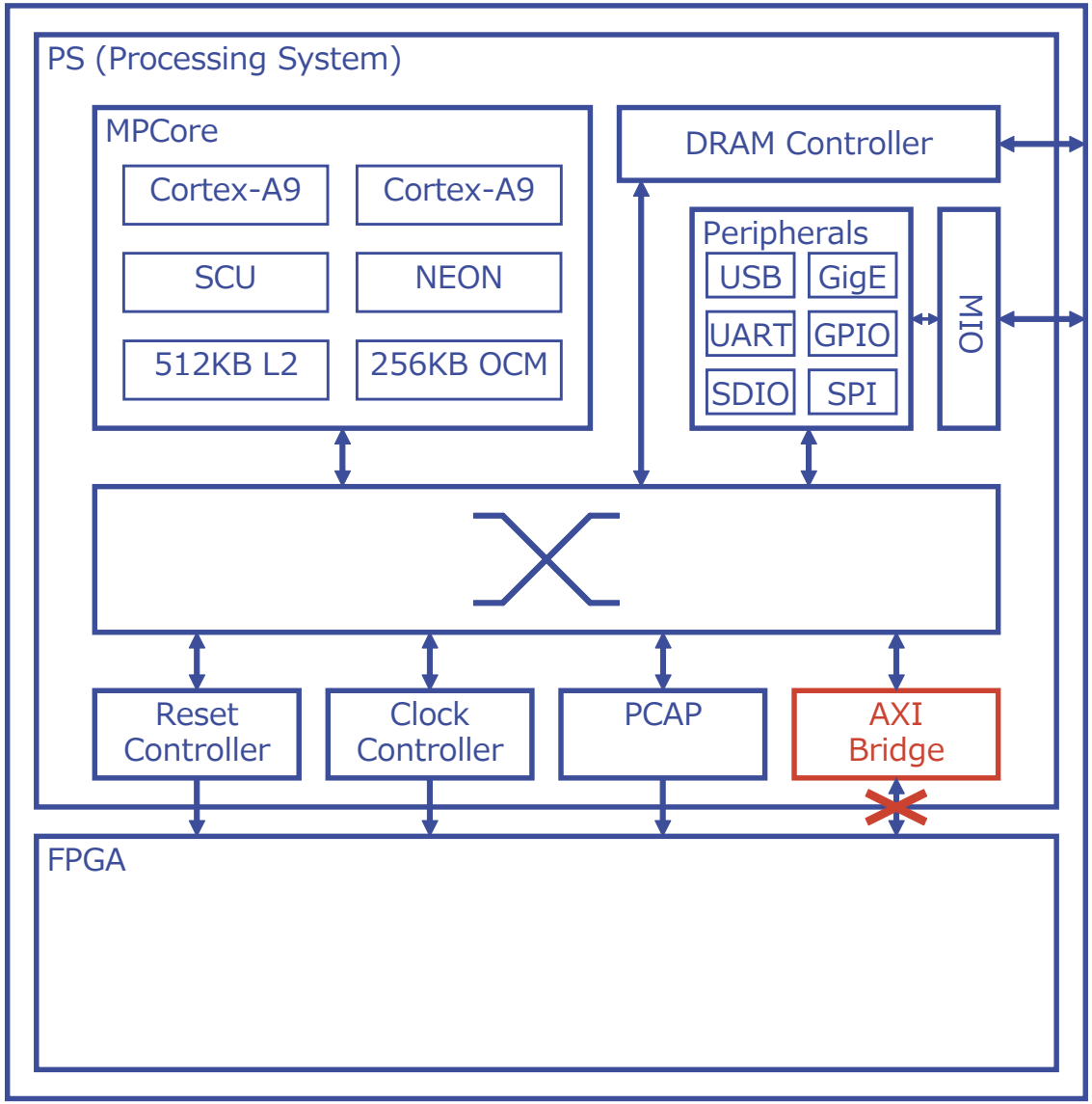
- BitStream をメモリから読んで FPGA に書き込む
- PCAP (Xilinx)
  - Processor Configuration Access Port
- FPGA Manager(Altera)

FPGA Configuration Hardware (2) - Reset Controller



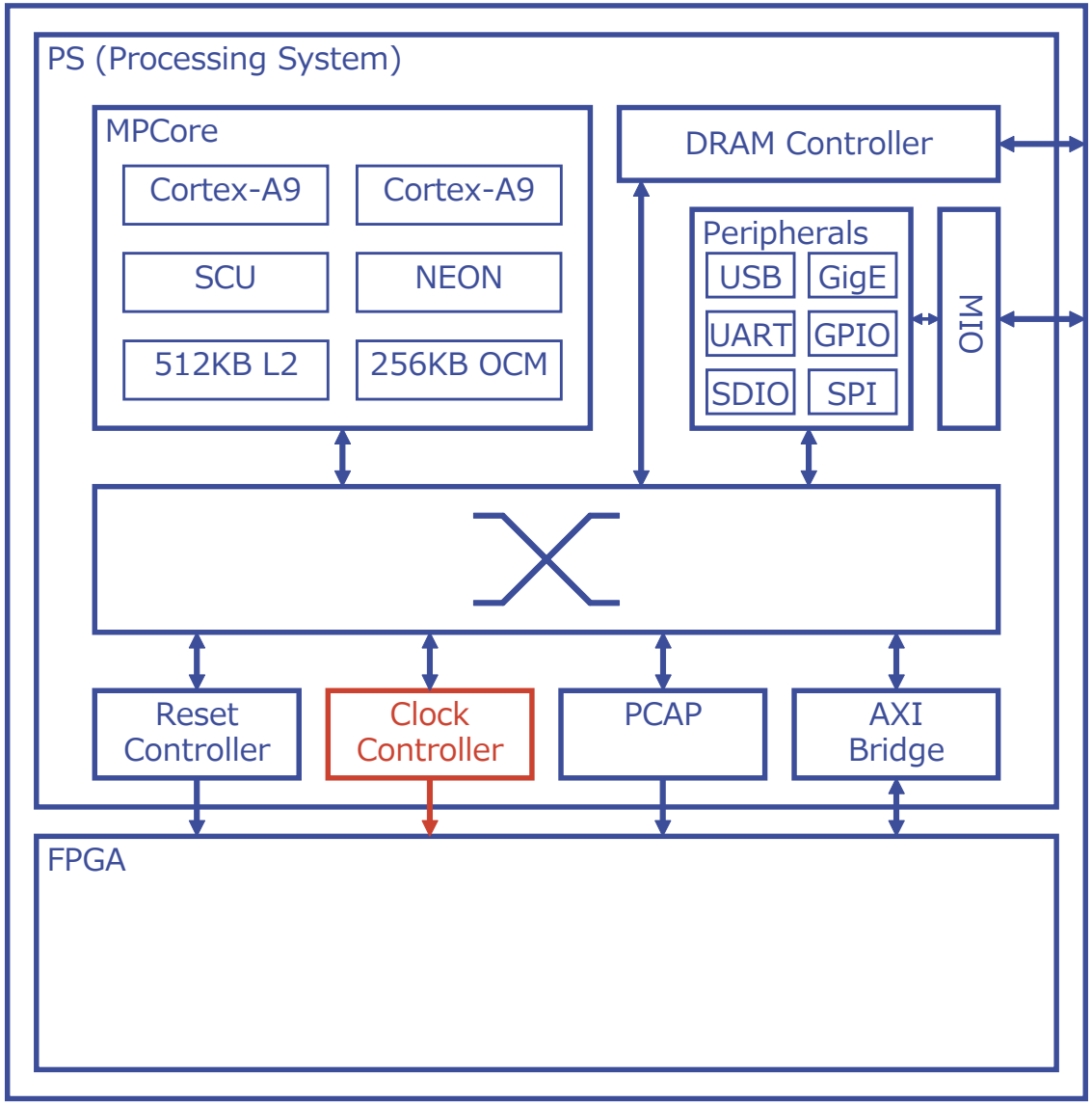
- コンフィギュレーション中に FPGA が誤動作しないように Register をリセット状態にしておく必要がある
- Full Configuration の時は PCAP がリセットをかけるので特に考慮する必要はない
- Partial Reconfiguration の時は考慮する必要があるので注意

FPGA Configuration Hardware (3) - Bridge



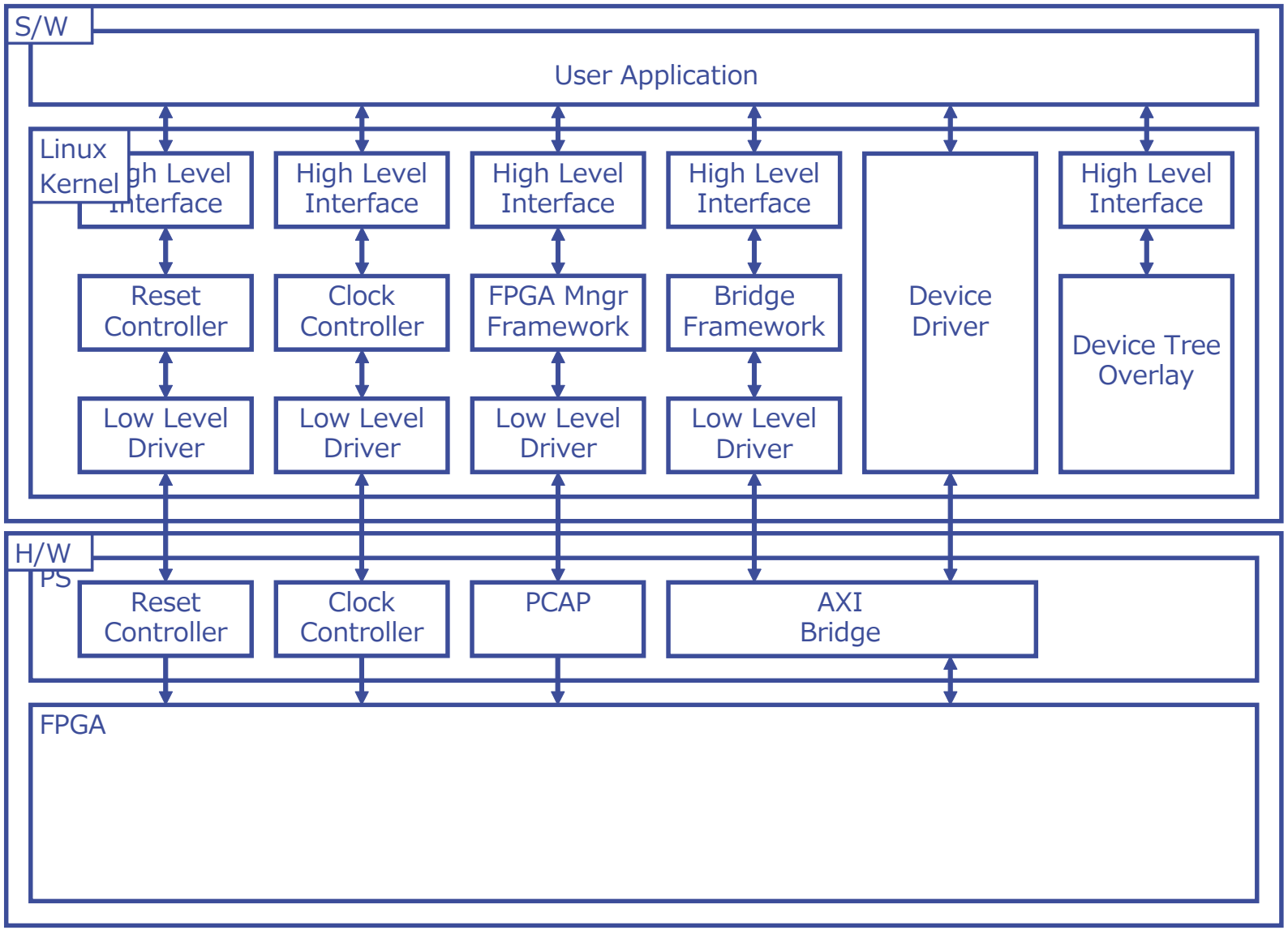
- コンフィギュレーション中に FPGA や PS が誤動作しないように Bridge をオフにしておく必要がある

FPGA Configuration Hardware (4) - Clock Controller

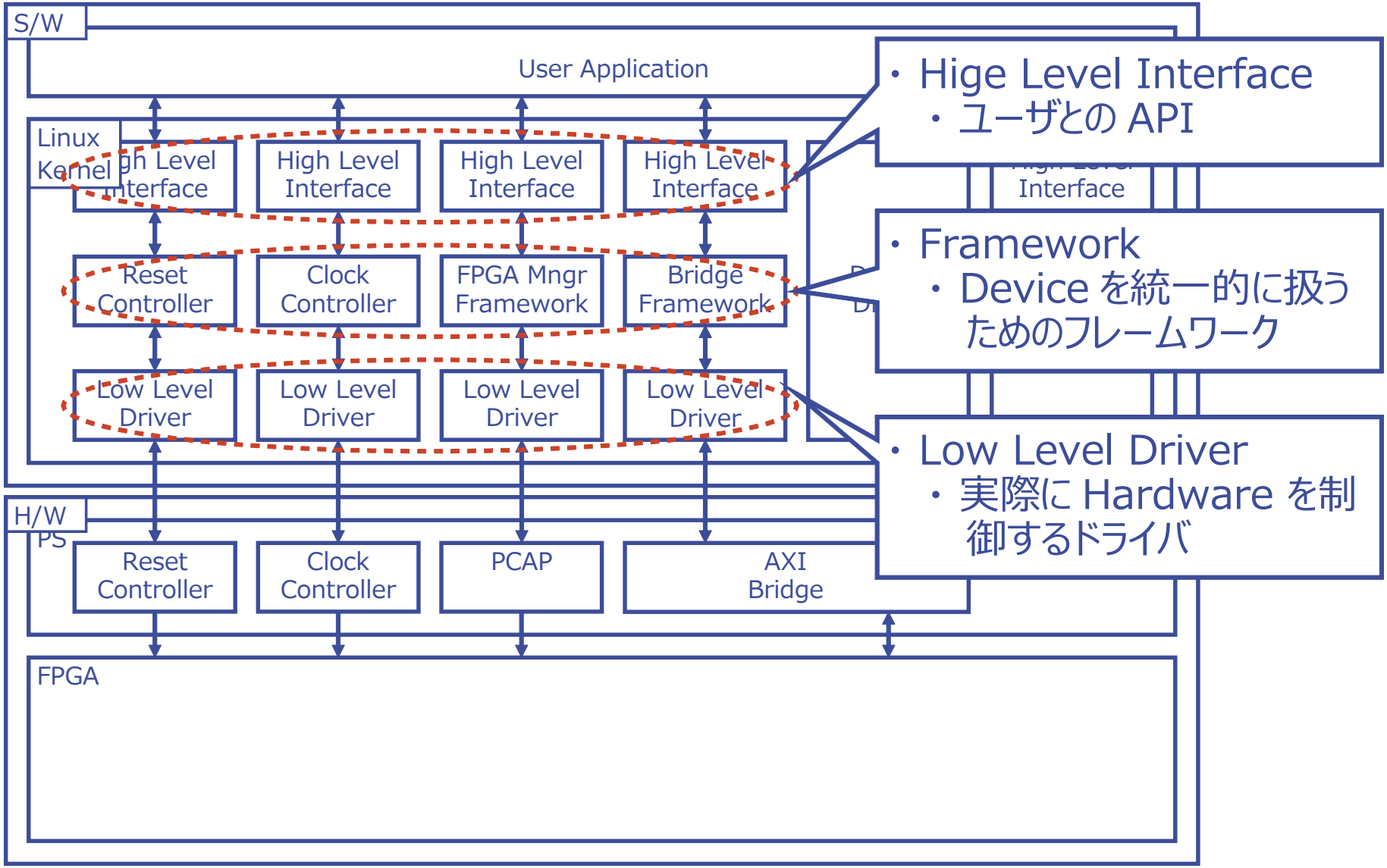


- デザイン毎に異なる周波数や使うクロックが異なることがあり、その場合はデザイン毎にクロックの設定を変更する必要がある
- FPGA を使わないときはクロックを止めておいたほうが省エネ上良いことがある

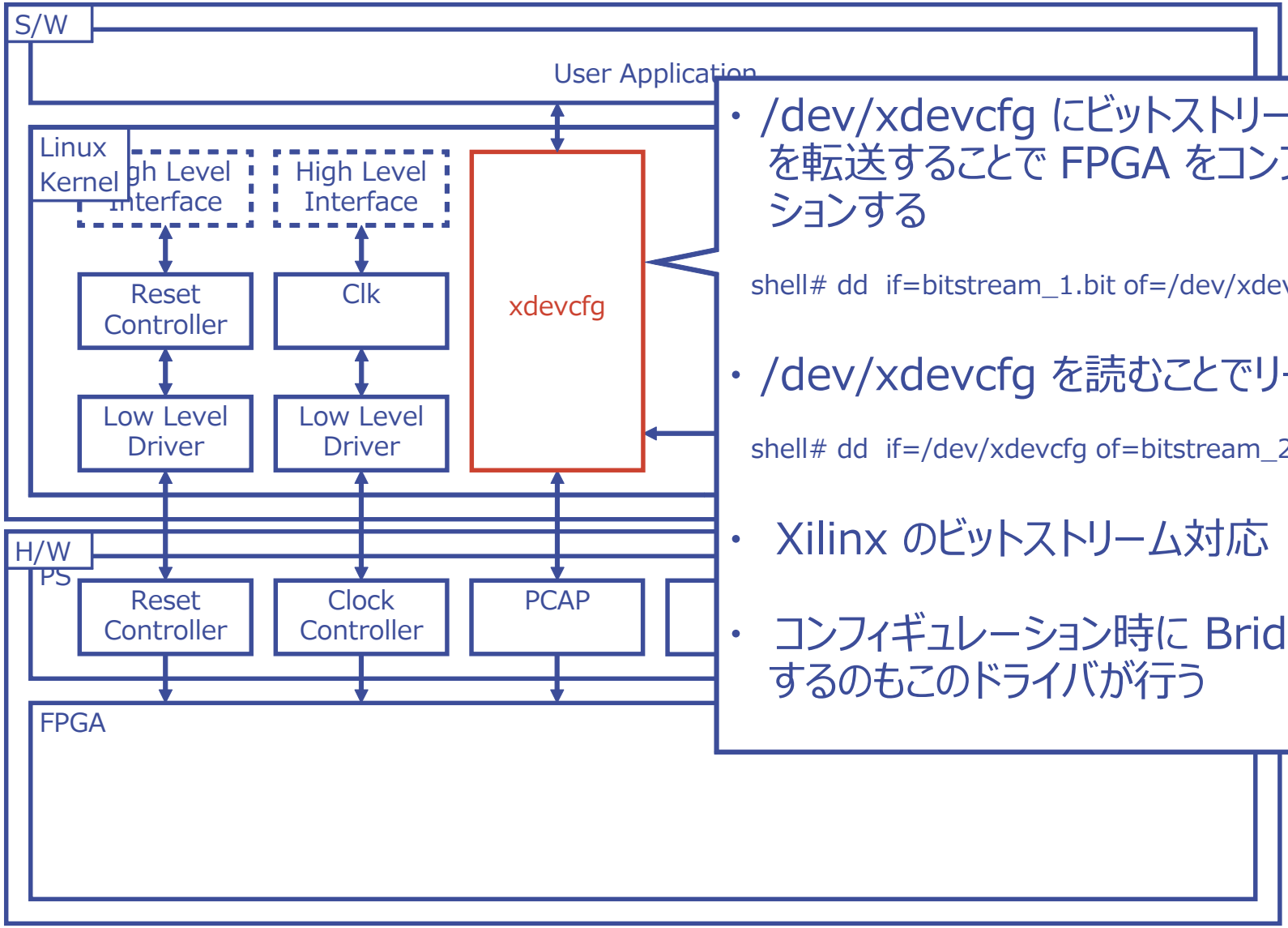
FPGA Configuration Software Stack



FPGA Configuration Software Stack



FPGA Configuration Software Stack - Vender Kernel Xilinx



- /dev/xdevcfg にビットストリームファイルを転送することで FPGA をコンフィギュレーションする

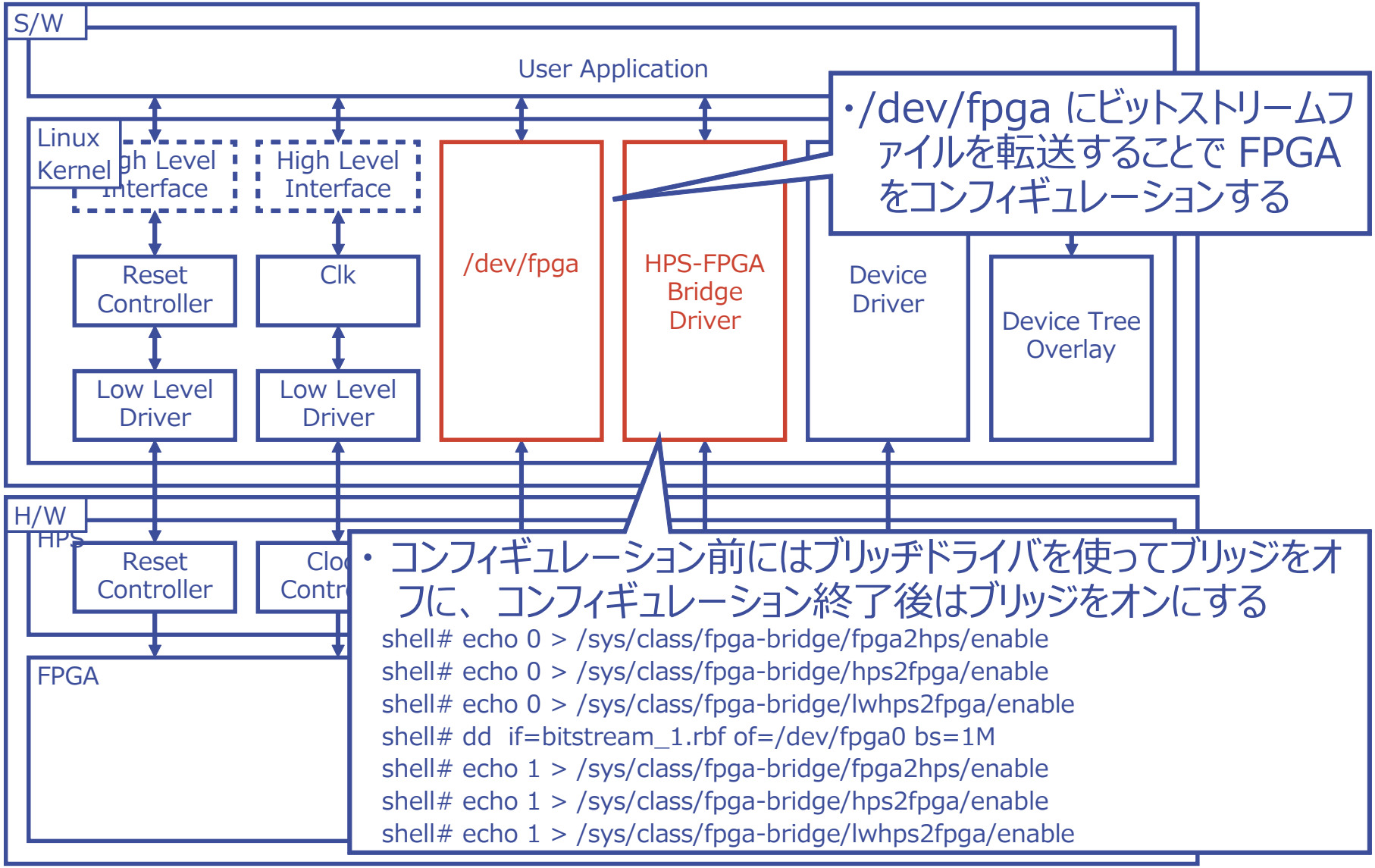
```
shell# dd  if=bitstream_1.bit of=/dev/xdevcfg bs=1M
```

- /dev/xdevcfg を読むことでリードバック

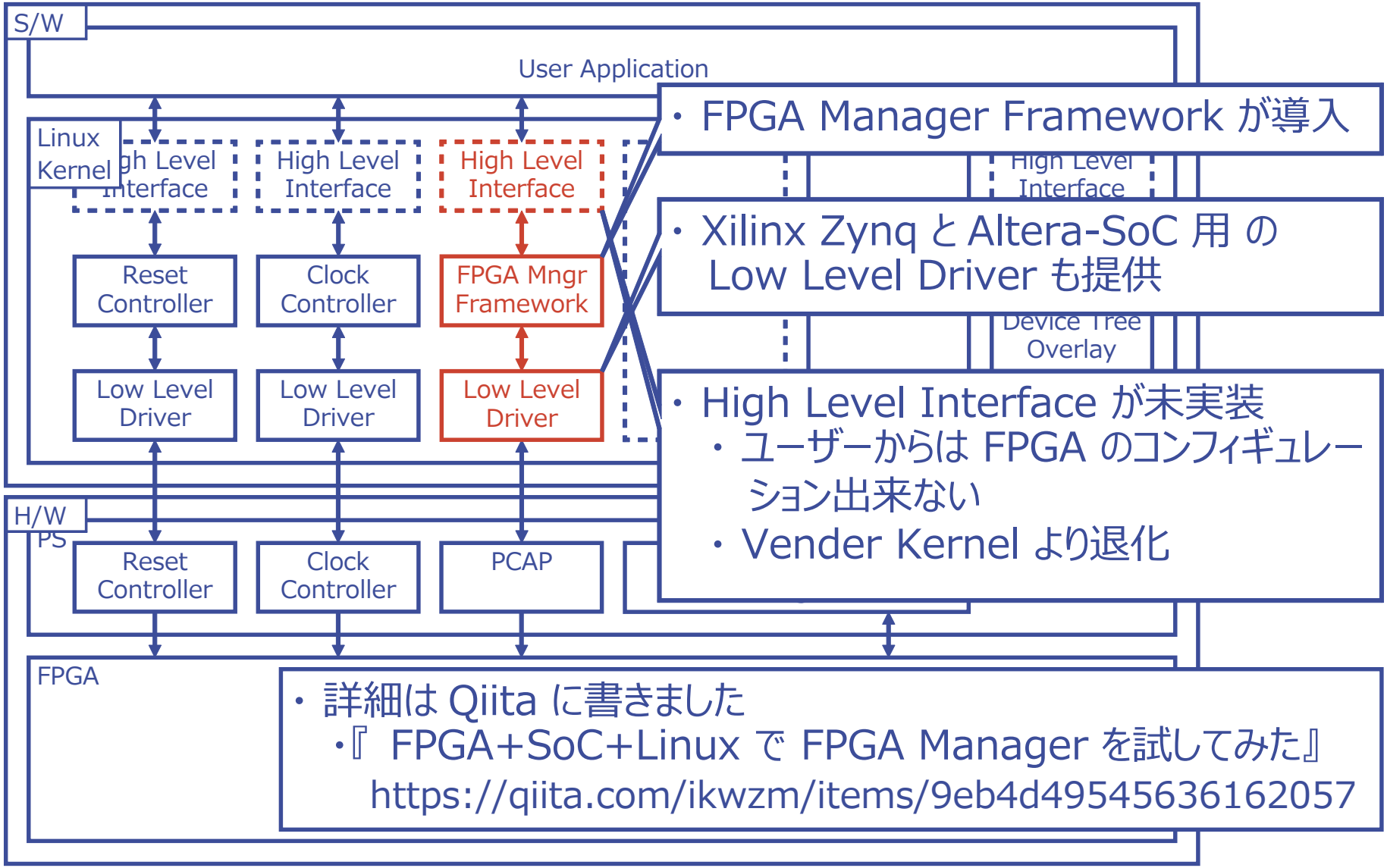
```
shell# dd  if=/dev/xdevcfg of=bitstream_2.bit  bs=1M
```

- Xilinx のビットストリーム対応
- コンフィギュレーション時に Bridge をオフするのもこのドライバが行う

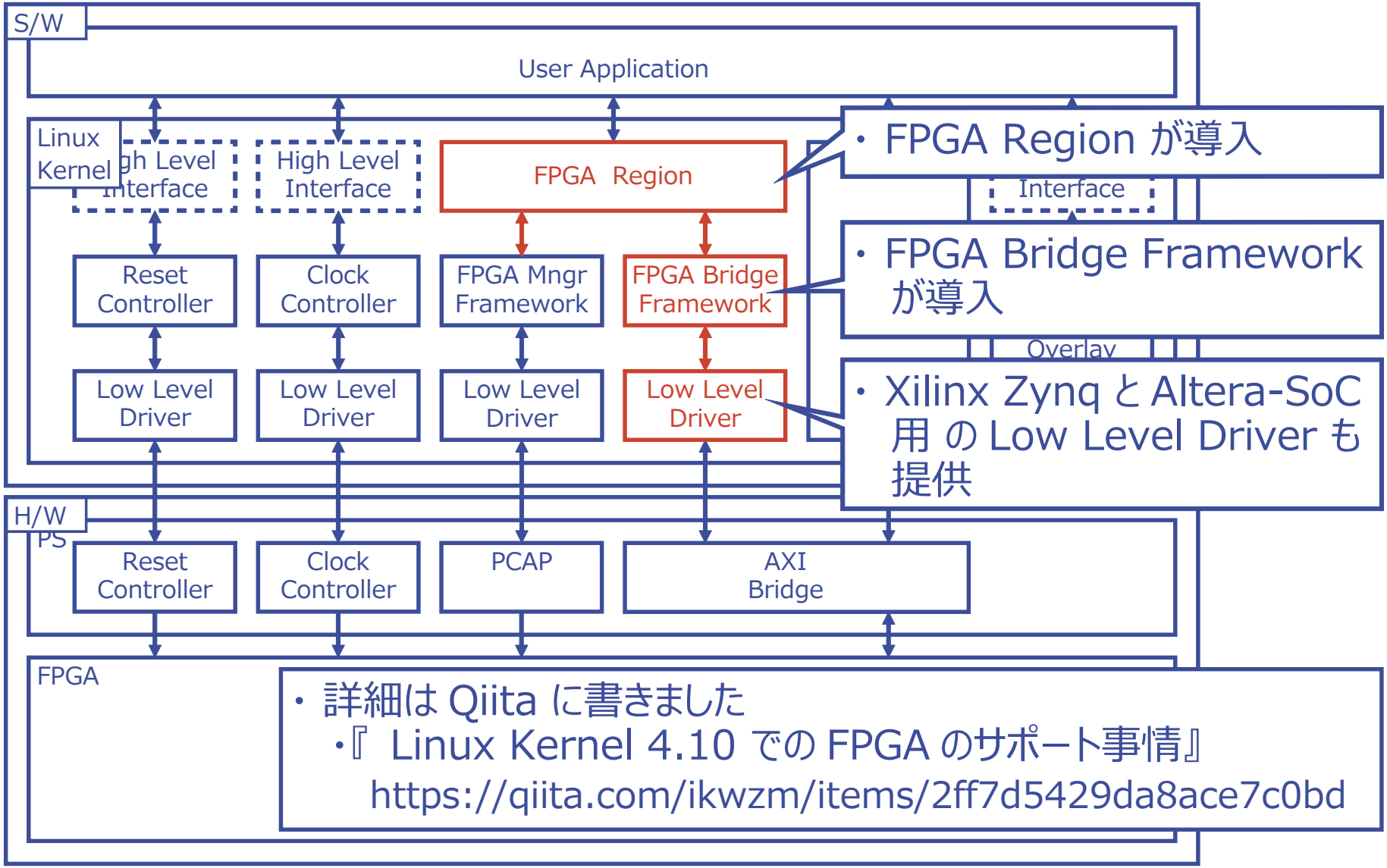
FPGA Configuration Software Stack - Vender Kernel Altera



FPGA Configuration Software Stack - Mainline Kernel 4.4



FPGA Configuration Software Stack - Mainline Kernel 4.10



FPGA Configuration Software Stack - Mainline Kernel 4.10

詳細はこれから説明します

が

その前に

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

without device driver

Device Tree のど～でもいい話 (1) - 導入の経緯

- kernel 2.6.38 の時に、Linus 氏が肥大する ARM コードに激怒したのが発端らしい

- 引用: LKML: Linus Torvalds: Re: [GIT PULL] omap changes for v2.6.39 merge window

  https://lkml.org/lkml/2011/3/30/379

- 参考: [Linux][kernel] Device Tree についてのまとめ @Qiita

  https://qiita.com/koara-local/items/ed99a7b96a0ca252fc4e

Device Tree のど～でもいい話 (2) - 生まれ

- 元々は Open Firmware の規格の一つ
  - Open Firmware（または OpenBoot）はハードウェアに依存しないファームウェア（オペレーティングシステムをロードするソフトウェア）であり、サン・マイクロシステムズのミッチ・ブラッドリーによって開発され、IEEE により標準化され、サン・マイクロシステムズ、アップル、IBM などによって使われている。
    https://ja.wikipedia.org/wiki/Open_Firmware
    https://en.wikipedia.org/wiki/Device_tree

- Linux Kernel のソースコードに名残
  - Device Tree 関連のソースコードは drivers/of/ にある
  - Device Tree 関連の API や構造体は of_ で始まる

Device Tree の問題点と Overlay による解決

- Boot 時に構成が決まっていなければならない
  - 元々 Open Firmware という Boot Loader の規格
  - 動的に構成が変わるのは想定外(Hot plug 未対応)
  - FPGA を動的に変更する用途には向かない

- そこで Device Tree Overlay
  - Device Tree を後から自由に追加削除できる機能
  - Linux Kernel 3.19 から導入

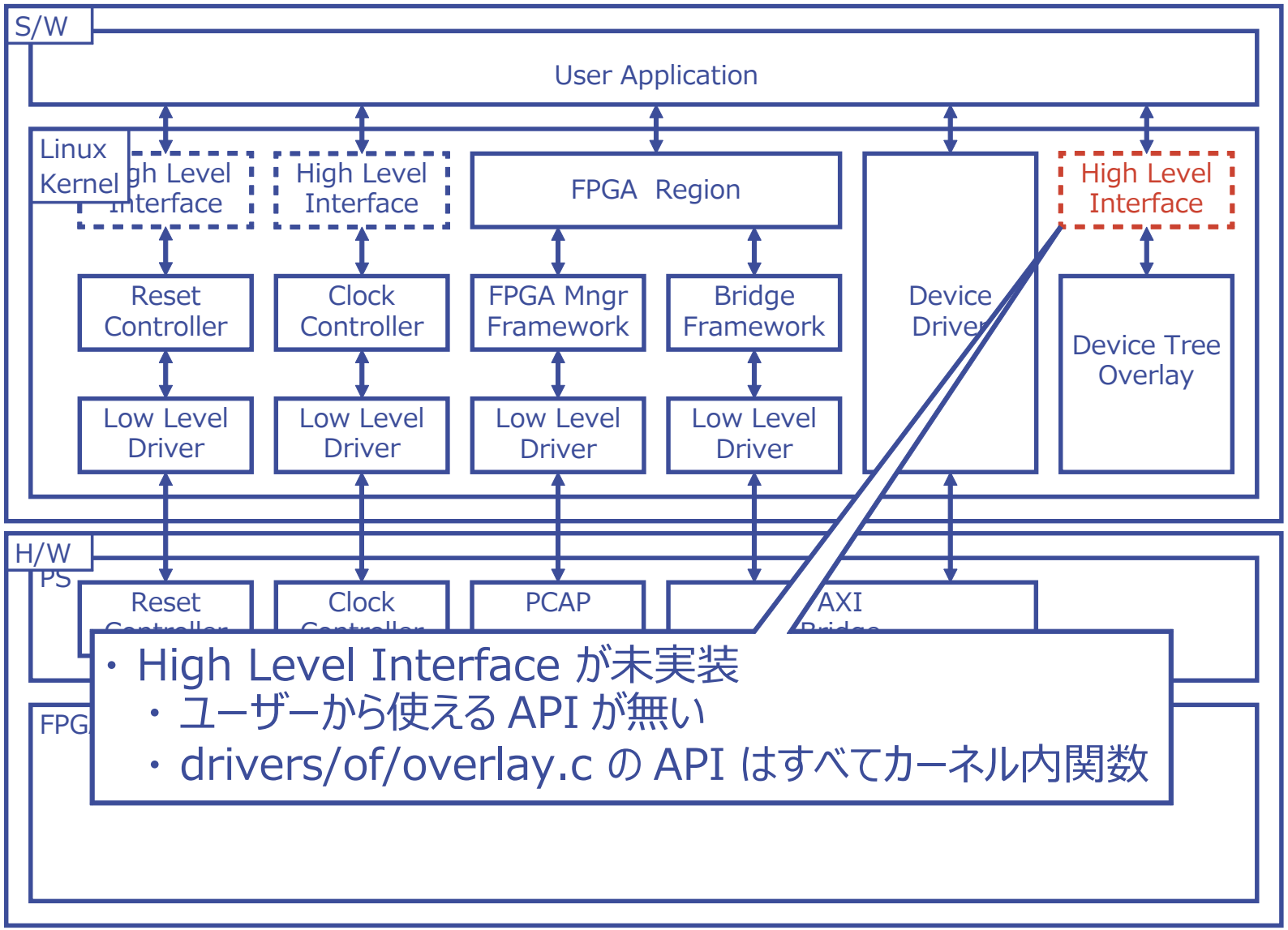
Overlay Device Tree Source Format

```
/dts-v1/;
/ {
        /* ignored properties by the overlay */

        fragment@0 {   /* first child node */
                target=<phandle>;       /* phandle target of the overlay */
        or
                target-path="/path";    /* target path of the overlay */

                __overlay__ {
                        property-a;     /* add property-a to the target */
                        node-a {        /* add to an existing, or create a node-a */
                                ...
                        };
                };
        }
        fragment@1 {   /* second child node */
                ...
        };
        /* more fragments follow */
} ;
```

Device Tree Overlay のサンプル

```
/dts-v1/;
/ {
        fragment@0 {
                target-path = "/amba";
                __overlay__ {
                        #address-cells = <0x1>;
                        #size-cells = <0x1>;
                        uio0@43c10000 {
                                compatible = "generic-uio";
                                reg = <0x43c10000 0x1000>;
                                interrupts = <0x0 0x1d 0x4>;
                        };
                };
        };
};
```

Device Tree Overlay の課題



dtbocfg

- dtbocfg を作りました
  - Device Tree Overlay Configuration File System
  - Device Tree Overlay をユーザーから使えるようにするデバイスドライバ
  - configfs を通して Device Tree を追加/削除する
  - Mainline Kernel で正式サポートされるまでの中継ぎ
  - https://github.com/ikwzm/dtbocfg

dtbocfg の使い方（1）

- ConfigFS にディレクトリを用意する
  - /config/device-tree/overlays の下に適当に名前を付けてディレクトリを作ります

    shell# mkdir /config/device-tree/overlays/uio0

  - /config/device-tree/overlays/uio0 の下に status と dtbo というファイルが自動的に出来ます。

    shell# ls -la /config/device-tree/overlays/uio0/
    合計 0
    drwxr-xr-x 2 root root   0 4 月 4 20:08 .
    drwxr-xr-x 3 root root   0 4 月 4 20:08 ..
    -rw-r--r-- 1 root root 4096 4 月 4 20:09 dtbo
    -rw-r--r-- 1 root root 4096 4 月 4 20:09 status

dtbocfg の使い方（2）

- Device Tree Blob を書き込む
  - /config/device-tree/overlays/uio0/dtbo に書き込む

```
shell# dtc -I dts -O dtb -o uio0.dtbo uio0.dts
shell# cp uio0.dtbo /config/device-tree/overlays/uio0/dtbo
```

- Device Tree に追加する
  - /config/device-tree/overlays/uio0/status に書き込む

```
shell# echo 1 > /config/device-tree/overlays/uio0/status
```

dtbocfg の使い方（3）

- Device Tree から削除（1）
  - /config/device-tree/overlays/uio0/status に書き込む

    shell# echo 0 > /config/device-tree/overlays/uio0/status

- Device Tree から削除（2）
  - /config/device-tree/overlays/uio0 を削除

    shell# rmdir /config/device-tree/overlays/uio0

dtbocfg.rb

・dtbocfg を少し使いやすくするための Ruby スクリプト

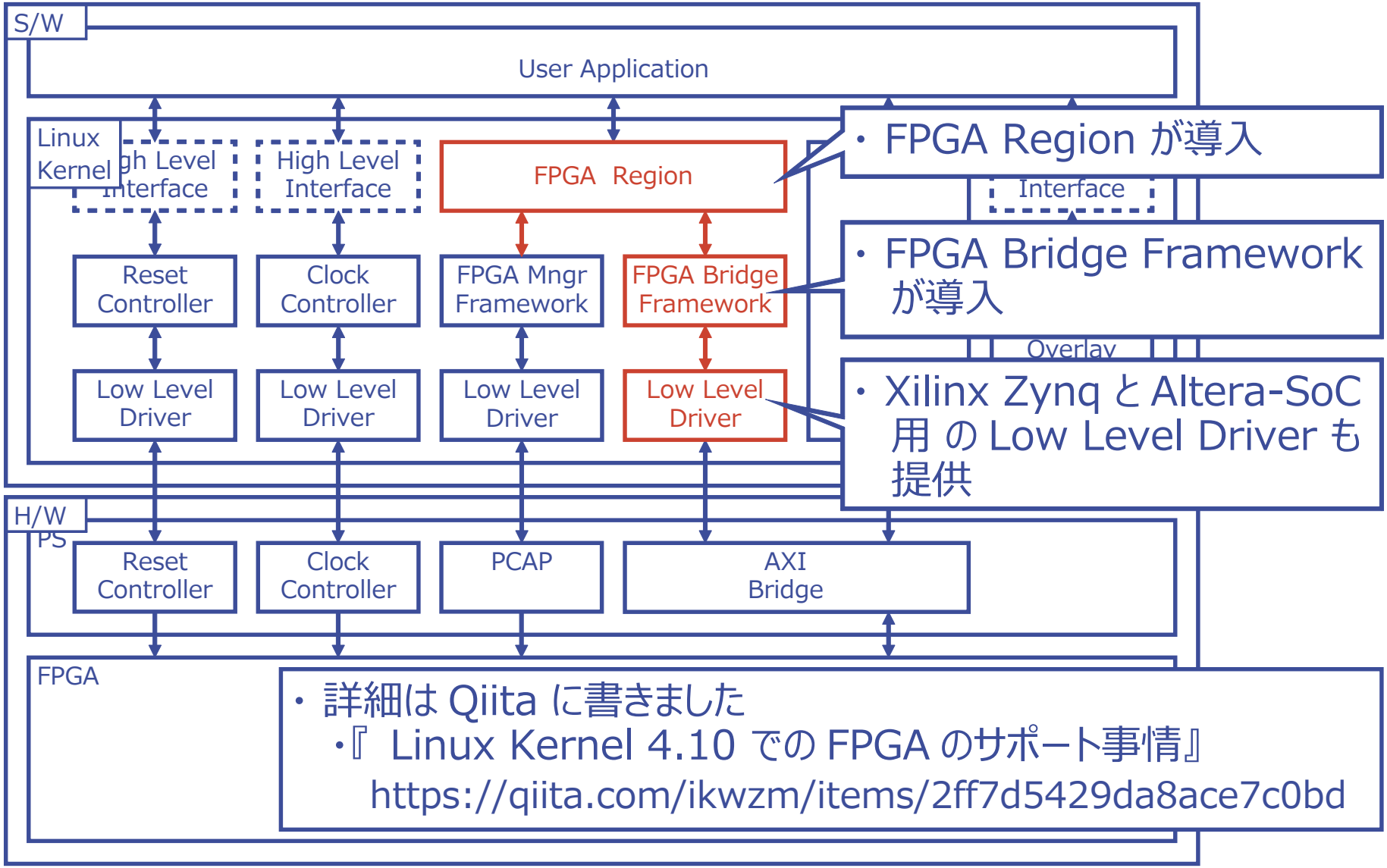
```
Usage: dtbocfg [command] [options] device_name
command
    -i, --install            Install (Create, Load, Start)
    -s, --start              Append Device Tree Blob on ConfigFS to   System
    -t, --stop               Remove Device Tree Blob on ConfigFS from System
    -l, --load               Load   Device Tree Overlay File to ConfigFS
    -c, --create             Create Device Tree Overlay Directory to ConfigFS
    -r, --remove             Remove Device Tree Overlay Directory to ConfigFS
options
        --dts FILE_NAME      Device Tree Source File
        --dtb FILE_NAME      Device Tree Blob File
    -v, --verbose
    -d, --debug
```

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region

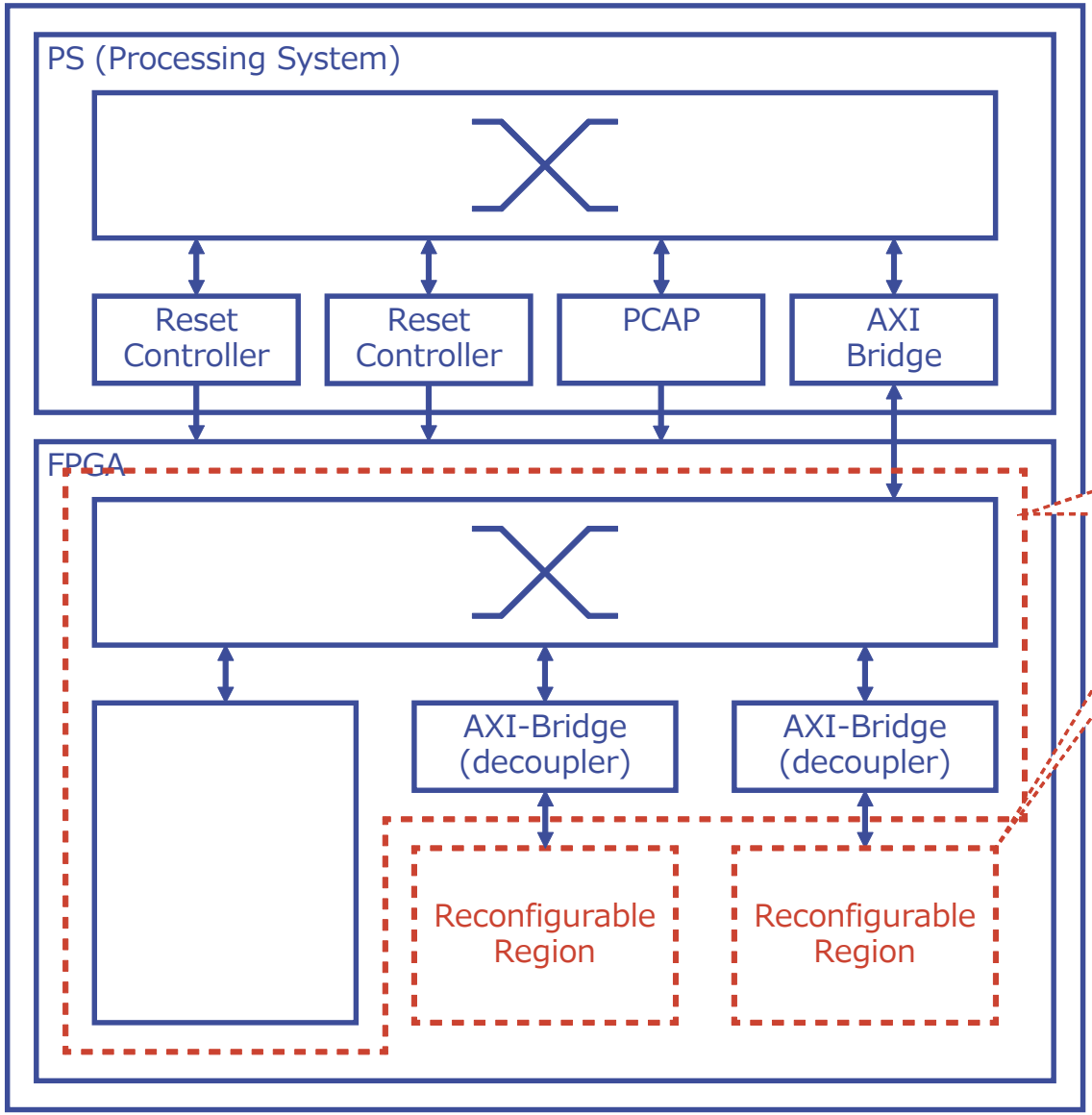
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

∧
without
device driver

FPGA Configuration Software Stack - Mainline Kernel 4.10



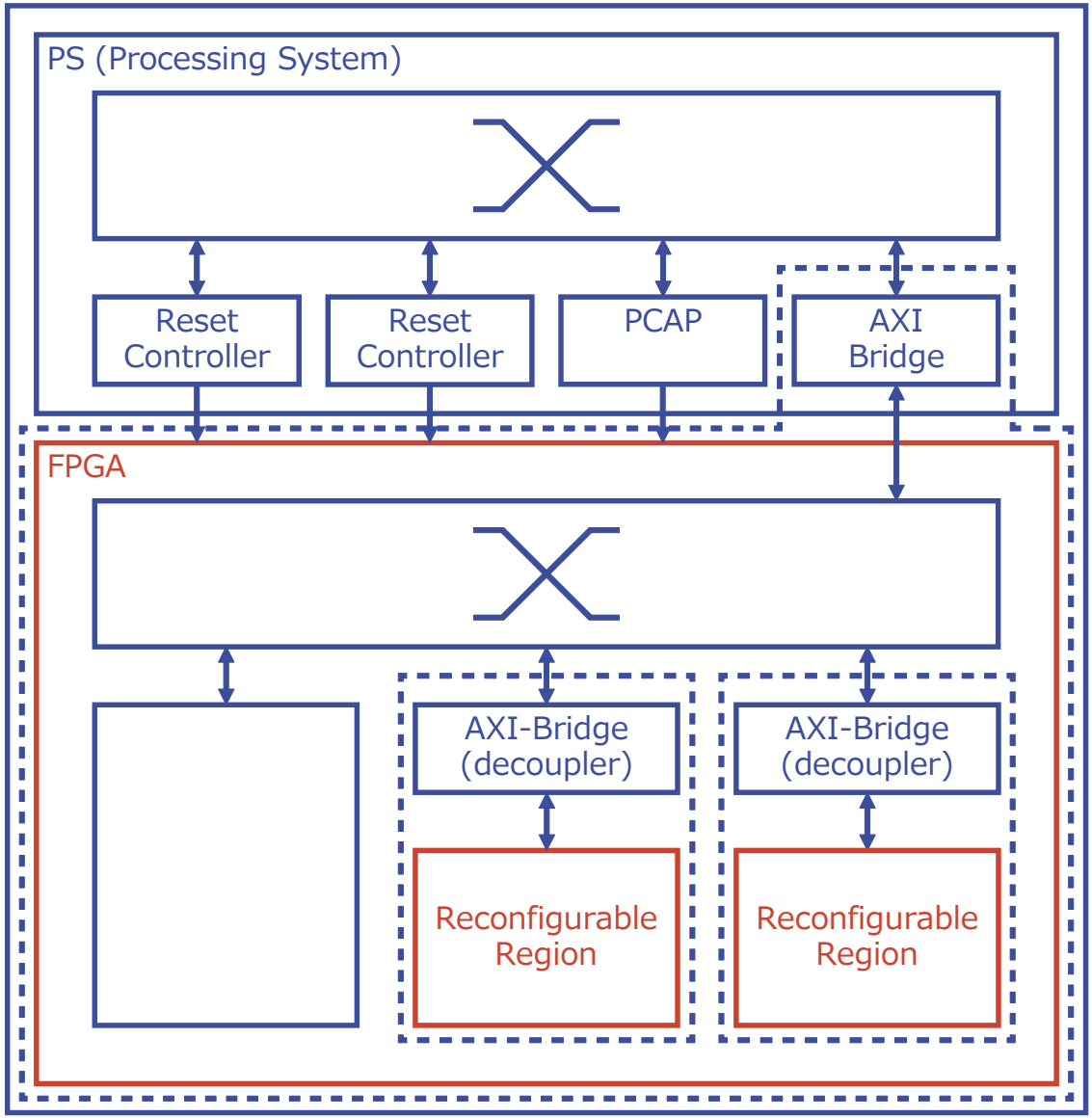
What Region ?



- Region - 領域
- Altera の Partial Reconfiguration 用語から (多分)
- FPGA-Region は Altera 製

- Static Region
- Rconfigurable Region

Why Region ?



- Region は Bridge を介して外部と接続
- FPGA 全部を一つの Region とみなすと、Region が入れ子になっていることに注目
- Full Configuration の場合は FPGA という Region にプログラムをロード
- Partial Reconfiguration の場合は、Reconfigurable Region にプログラムをロード
- プログラムをロードする対象を Region と見なしてデバイスドライバをモデル化

Device Tree による FPGA Region の定義

```
/dts-v1/;
/ {
        amba {
                devcfg: devcfg@f8007000 {/* Zynq 用 FPGA Mngr の Low Driver */
                        compatible = "xlnx,zynq-devcfg-1.0";
                        reg = <0xf8007000 0x100>;
                        interrupt-parent = <&intc>;
                        interrupts = <0 8 4>;
                        clocks = <&clkc 12>;
                        clock-names = "ref_clk";
                        syscon = <&slcr>;
                };
                fpga_region0: fpga-region0 {    /* FPGA 全体の Region の定義 */
                        compatible = "fpga-region";
                        fpga-mgr = <&devcfg>;/* FPGA Mngr の Low Driver を指定 */
                        #address-cells = <1>;
                        #size-cells = <1>;
                        ranges;
                };
        };
} ;
```

Device Tree Overlay for FPGA Full configuration (1)

・Device Tree のサンプルソース (sample.dts)

```
/dts-v1/; /plugin/;
/ {
        fragment@1 {
                target-path = "/amba/fpga-region0";     /* Region のパスを指定 */
                #address-cells = <1>;
                #size-cells = <1>;
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <1>;
                        firmware-name = "sample.bin"; /* ビットストリームファイル名 */
                        uio0@43c10000 {                 /* 追加するデバイスドライバ */
                                compatible = "generic-uio";
                                reg = <0x43c10000 0x1000>;
                                interrupts = <0x0 0x1d 0x4>;
                        };
                };
        };
};
```

Device Tree Overlay for FPGA Full configuration (2)

- ビットストリームファイルは /lib/firmware の下に置く
  - Xilinx の場合、何故か Vivado が生成したビットストリームファイルはそのままではロードできない。
  - Zynq 用の FPGA Manager の Low level Interface である devcfg が、Vivado のビットストリームファイルを扱えないのが原因
  - fpga-bit-to-bin.py 等でフォーマット変換する必要があります
  - 詳細は『 Linux の FPGA Manager で Xilinx のビットストリームファイルを扱う方法』@Qiita 参照

    https://qiita.com/ikwzm/items/1bb63be0b86a1e0e56fa

Device Tree Overlay for FPGA Full configuration (3)

- Device Tree Overlay でツリーに追加する

```
shell# dtbocfg.rb --install sample --dts sample.dts
```

  - ツリーに追加するだけで FPGA にビットストリームファイルがロード
  - コンフィギュレーション中は Bridge がオフになる
  - 使用するデバイスドライバも同時にツリーに追加できる
  - 使用するデバイスドライバは FPGA のコンフィギュレーション後に有効になる

Device Tree Overlay for FPGA Full configuration (4)

- Device Tree Overlay でツリーから削除する

```
shell# dtbocfg.rb --remove sample
```

  - 追加したデバイスドライバは無効になる
  - デバイスツリーから削除されても FPGA の回路は次にコンフィギュレーションするまで動き続ける
  - FPGA の回路を止めるには別の手段が必要

Mainline Kernel 4.10 の FPGA Configuration の懸念点 (1)

- Device Tree ありき
  - ARM アーキテクチャでは一般的だが x86 では？
- Device Tree Overlay のユーザー制御が未実装
  - FPGA の動的変更には不可欠なのになぜ？
- Clock の制御が未サポート
  - クロックの制御に関してノータッチ　どうすんの？
- そもそもカーネル内に実装する必要があるのか？

Mainline Kernel 4.10 の FPGA Configuration の懸念点 (2)

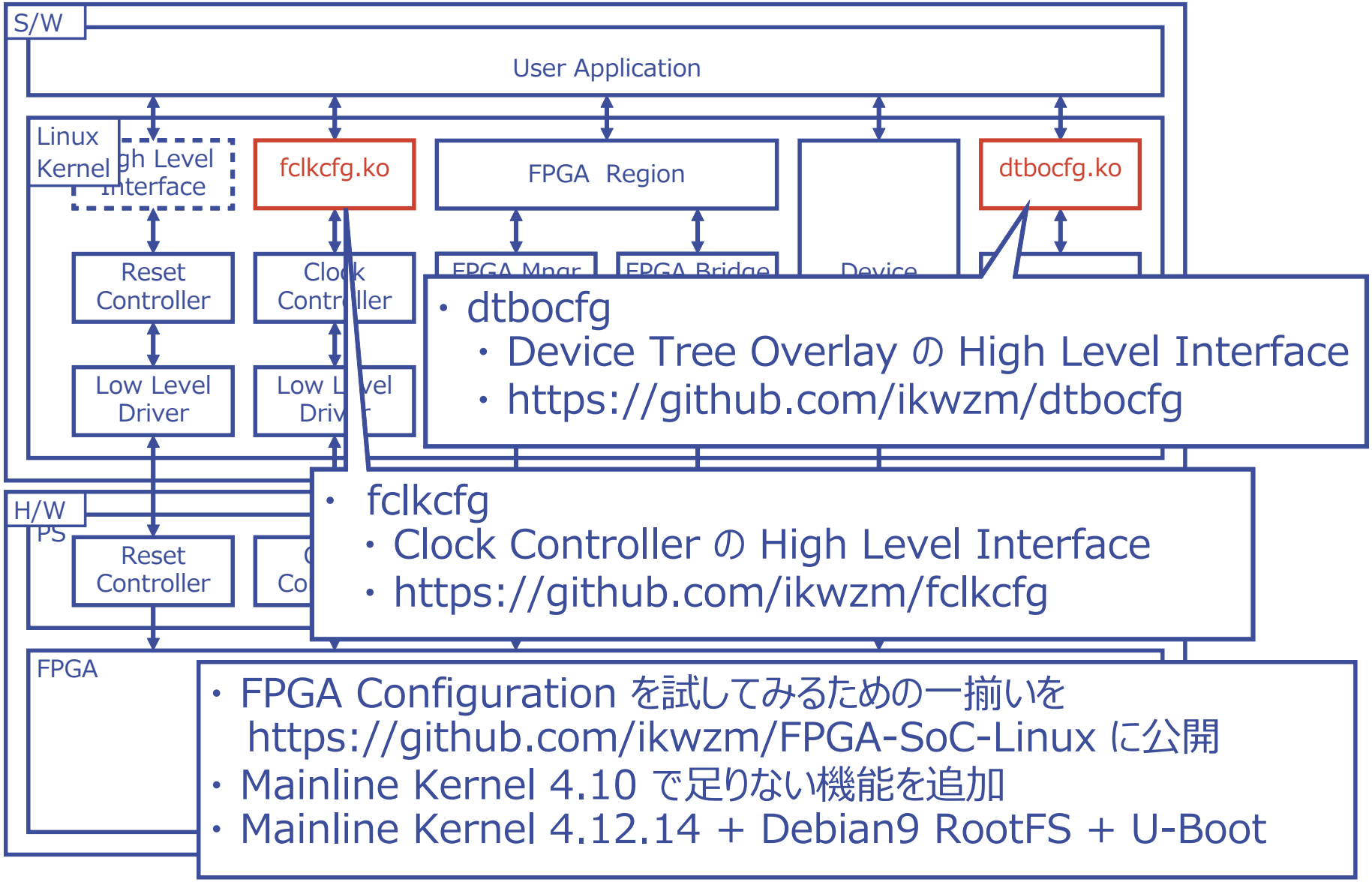
- Mainline Kernel 対 Vender Kernel
  - FPGA Region、FPGA Manager Framework、FPGA Bridge Framework は Altera 製
  - Xilinx は FPGA Framework 対応の Low Level Driver を Mainline に提供
  - Mainline Kernel に収束していくのか、Mainline Kernel と Vender Kernel の 2 本立てでいくのか？

Mainline Kernel の FPGA Configuration で言いたいこと

- まだまだ発展途上
  - Device Tree ありき
  - Device Tree Overlay のユーザー制御が未実装
  - Clock の制御が未サポート
  - Vender Kernel との関係

- でも足りない部分を補えば便利
  - Linux 動作中に FPGA やドライバを変更できる
  - FPGA のデザインごとに Linux を再構築しなくて済む
  - Vivado SDK とか JTAG ケーブル不要 (私はもう使っていない)

https://github.com/ikwzm/FPGA-SoC-Linux の紹介



https://github.com/ikwzm/FPGA-SoC-Linux の紹介

- FPGA Configuration お試し用
- あくまでも Mainline が正式サポートするまでのつなぎ
- Mainline Kernel 4.10 で足りない機能を追加
    - Device Tree Overlay の High Level Interface
    - Clock Controller の High Level Interface
- Mainline Kernel 4.12.14 + Debian9 RootFS + U-Boot
- ZYBO/ZYBO-Z7/PYNQ-Z1/De0-Nano-SoC 対応
- 詳しくは『 FPGA+SoC+Linux+Device Tree Overlay+FPGA Region(ブートイメージの提供)』@Qiita 参照

  https://qiita.com/ikwzm/items/7e90f0ca2165dbb9a577

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

without device driver

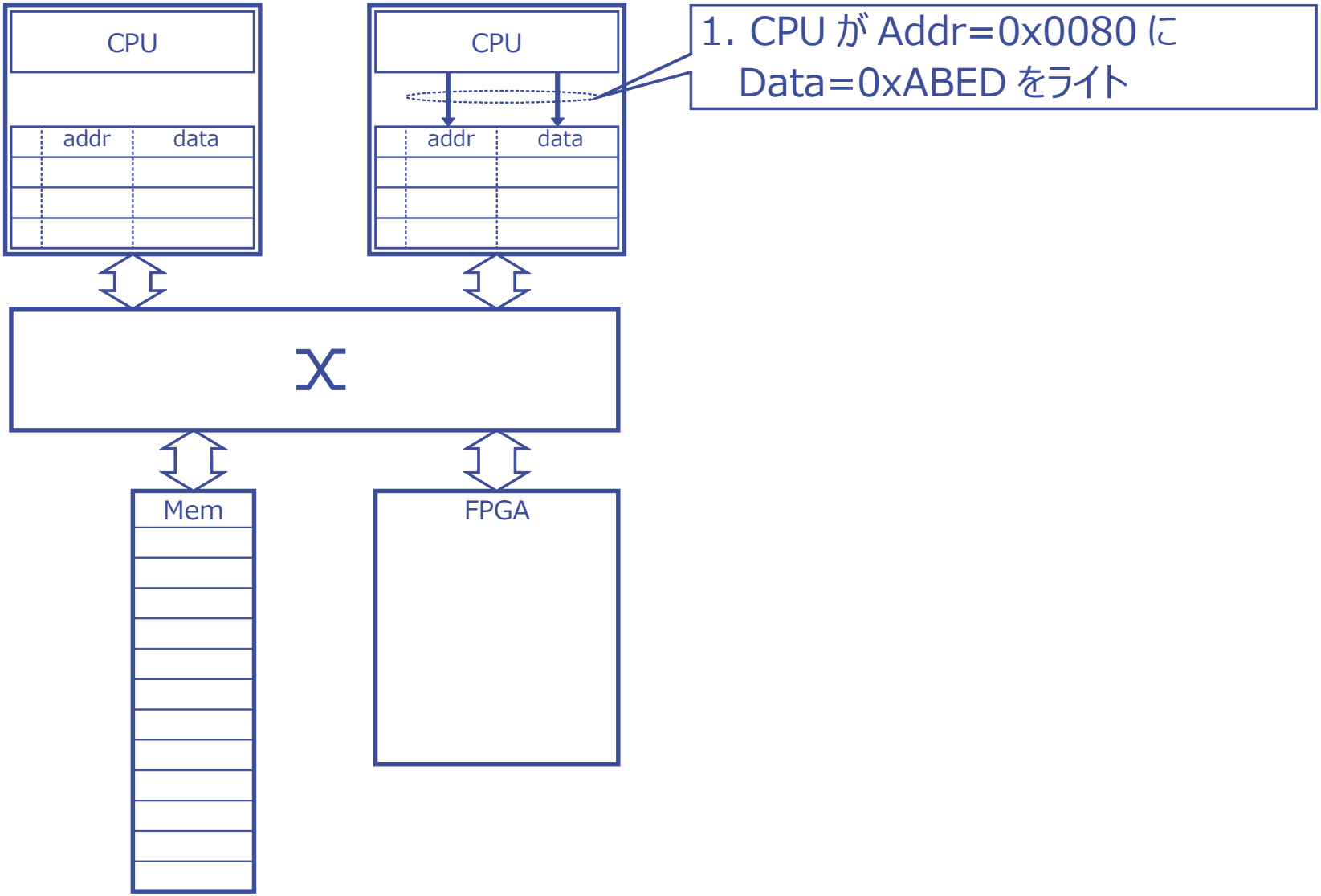
- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

without device driver

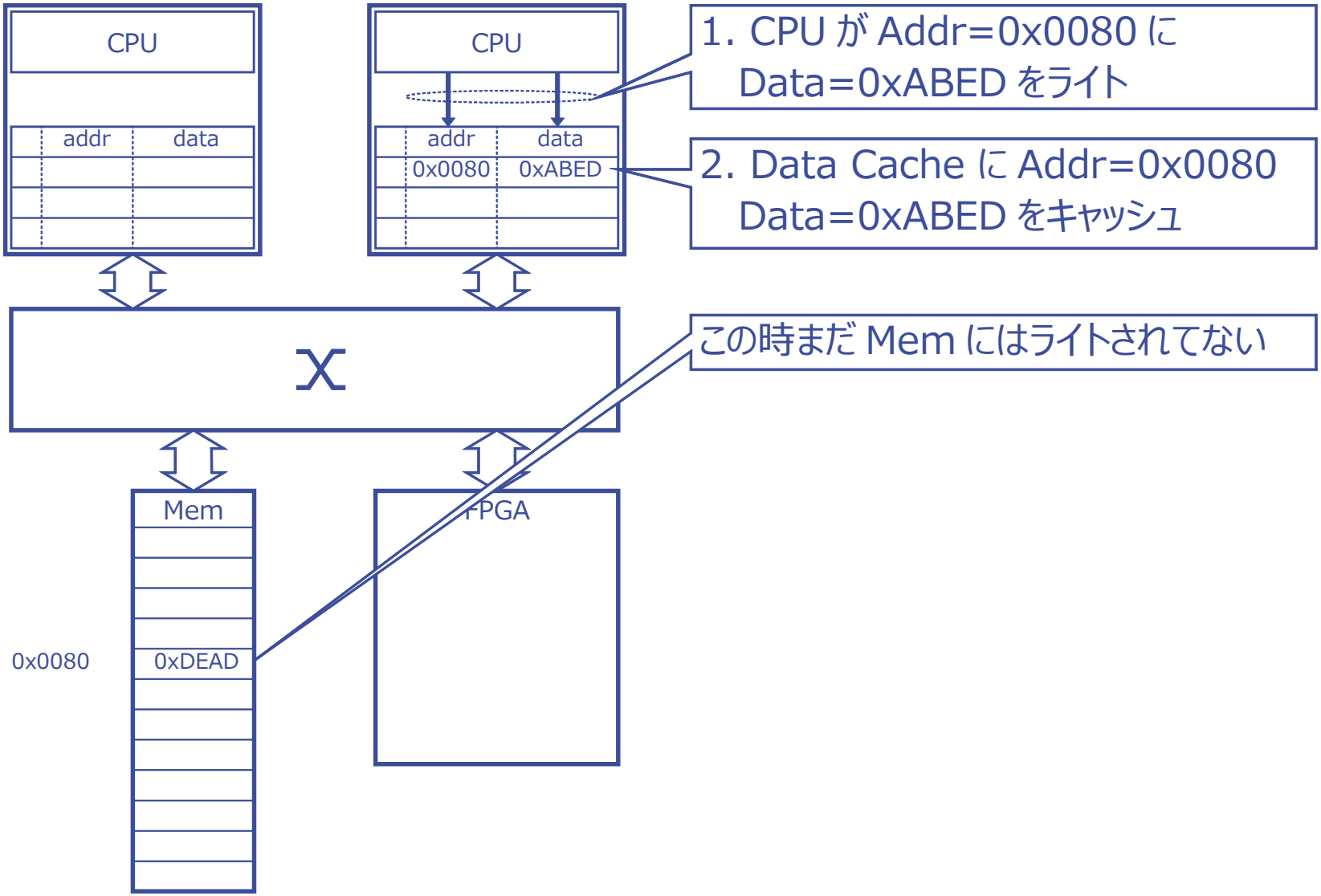
Cache Coherency トラブルの症状

- ケース 1　CPU が書いたデータが FPGA から正しく読めない
- ケース 2 FPGA が書いたデータが CPU から正しく読めない

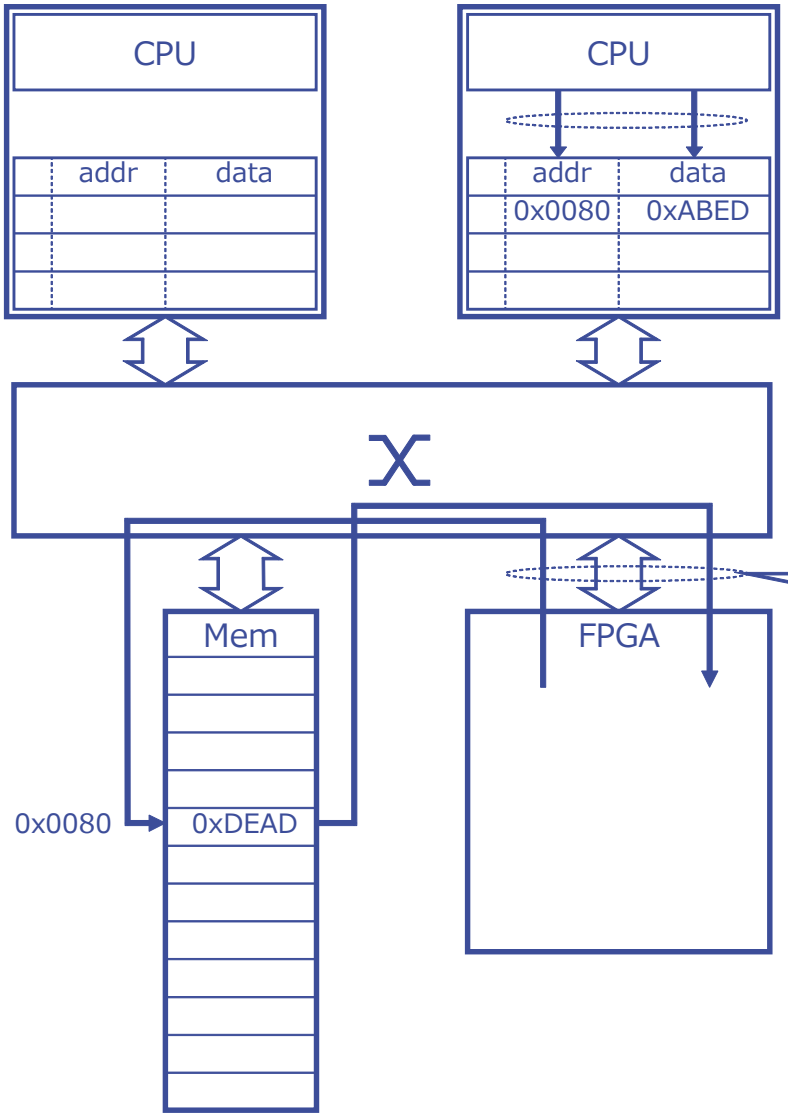
Cache Coherency でトラブルが発生するケース 1



Cache Coherency でトラブルが発生するケース 1



Cache Coherency でトラブルが発生するケース 1



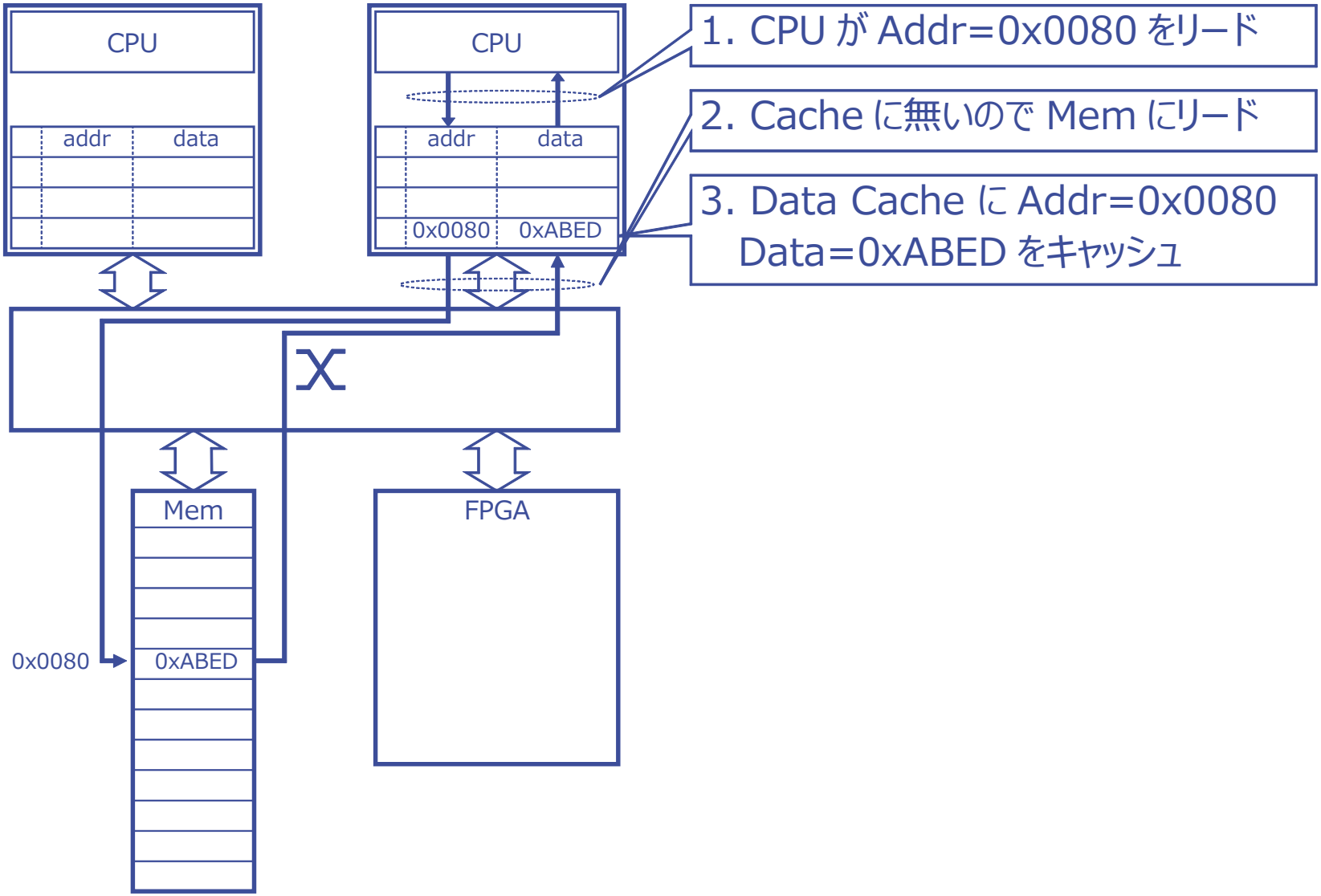
1. CPU が Addr=0x0080 に Data=0xABED をライト

2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

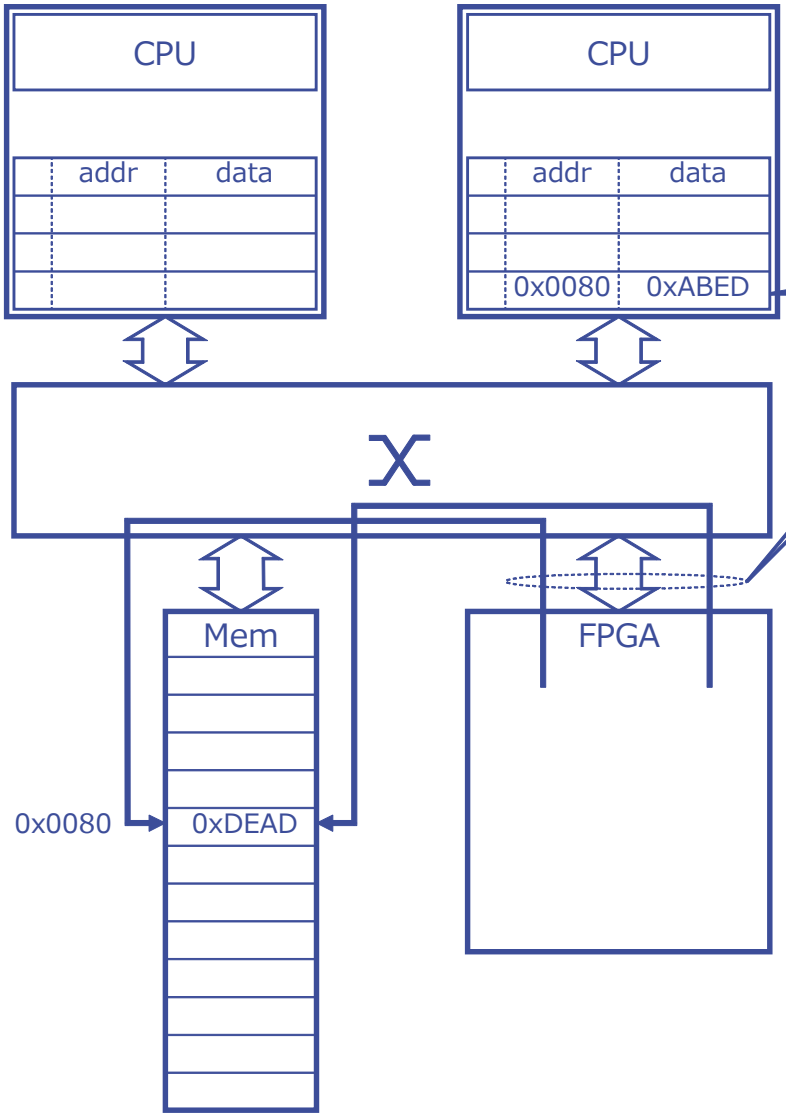
3. FPGA が Addr=0x0080 をリード

4. 未だ Mem の Addr=0x0080 には Data=0xABED は書き込まれていないので、FPGA は Data=0xDEAD を読んでしまう

Cache Coherency でトラブルが発生するケース 2

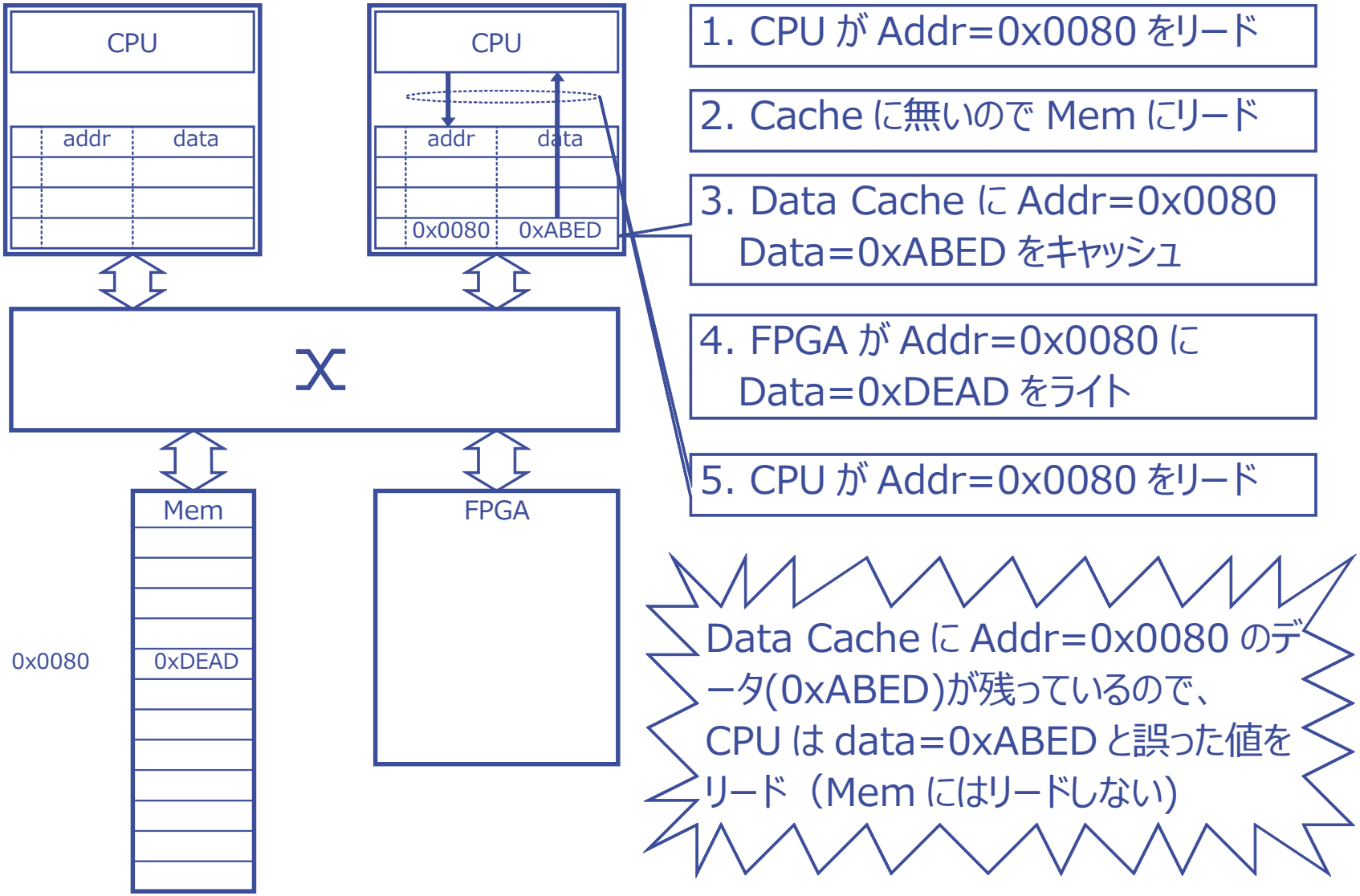


Cache Coherency でトラブルが発生するケース 2



1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト

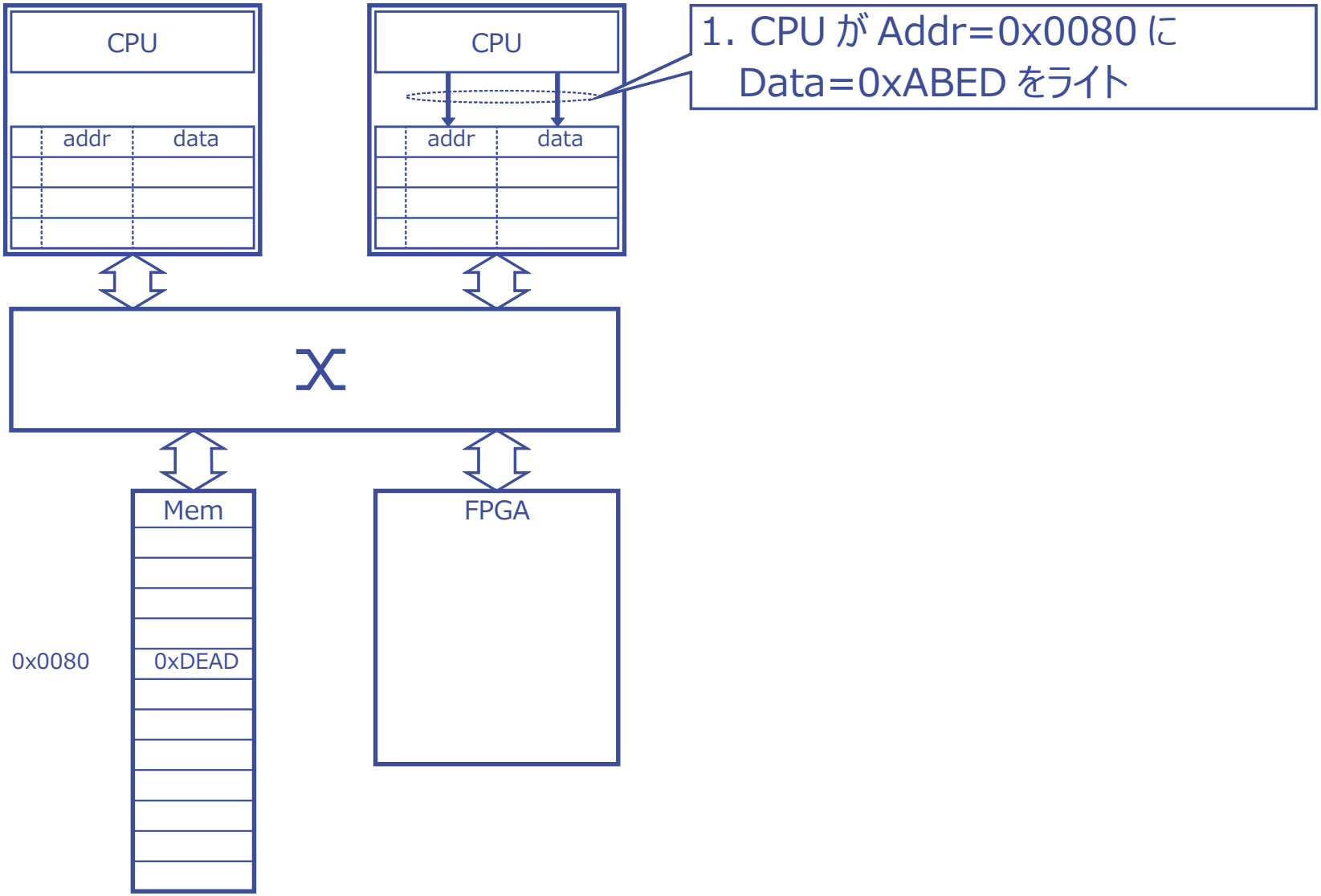
Cache Coherency でトラブルが発生するケース 2



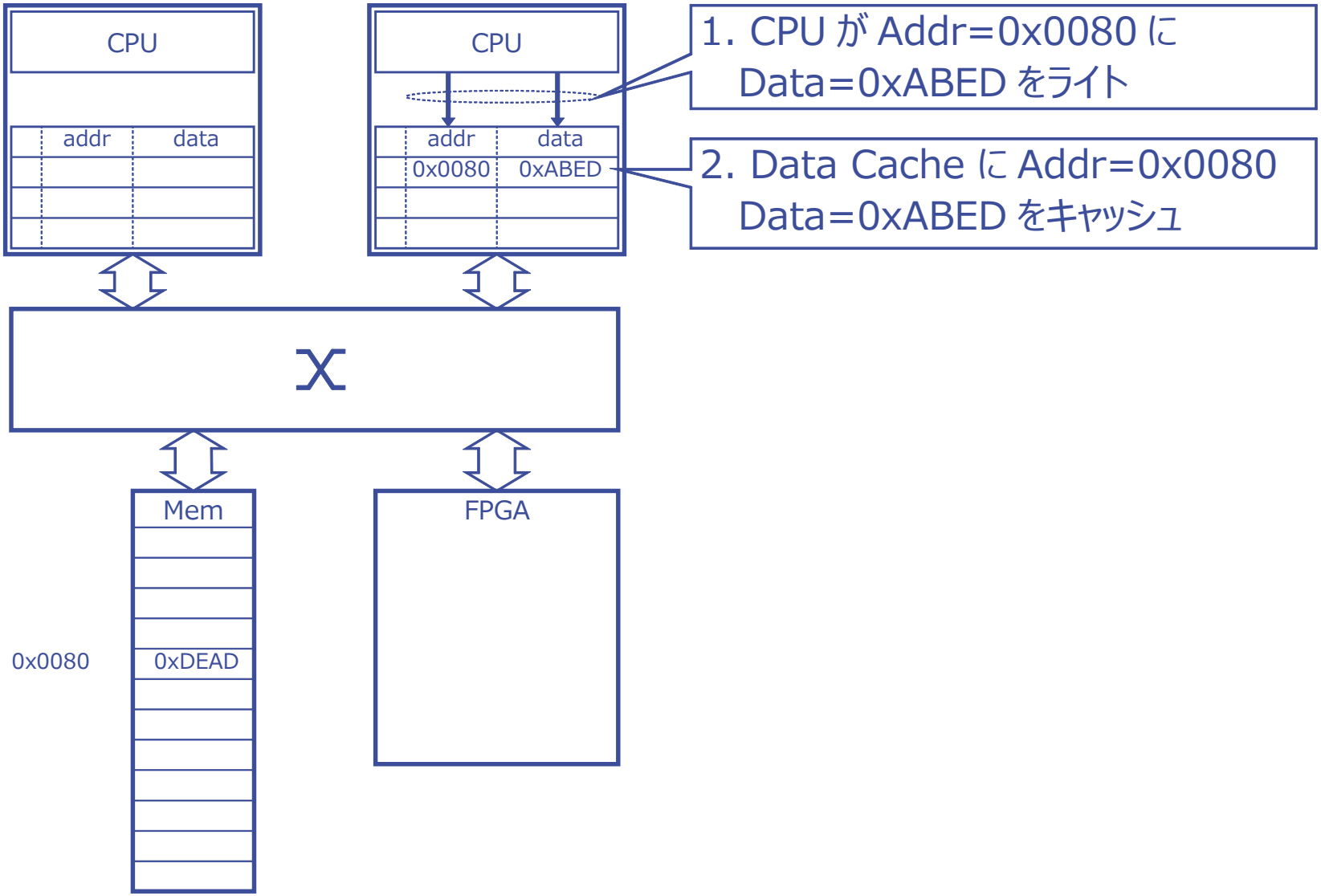
Cache Coherency トラブルの解決方法

- キャッシュを使わない
- ソフトウェアによる解決方法
  ソフトウェアで Cache Flush/Invalidiate する
- ハードウェアによる解決方法
  Cache Coherency に対応した Cache と Interconnect

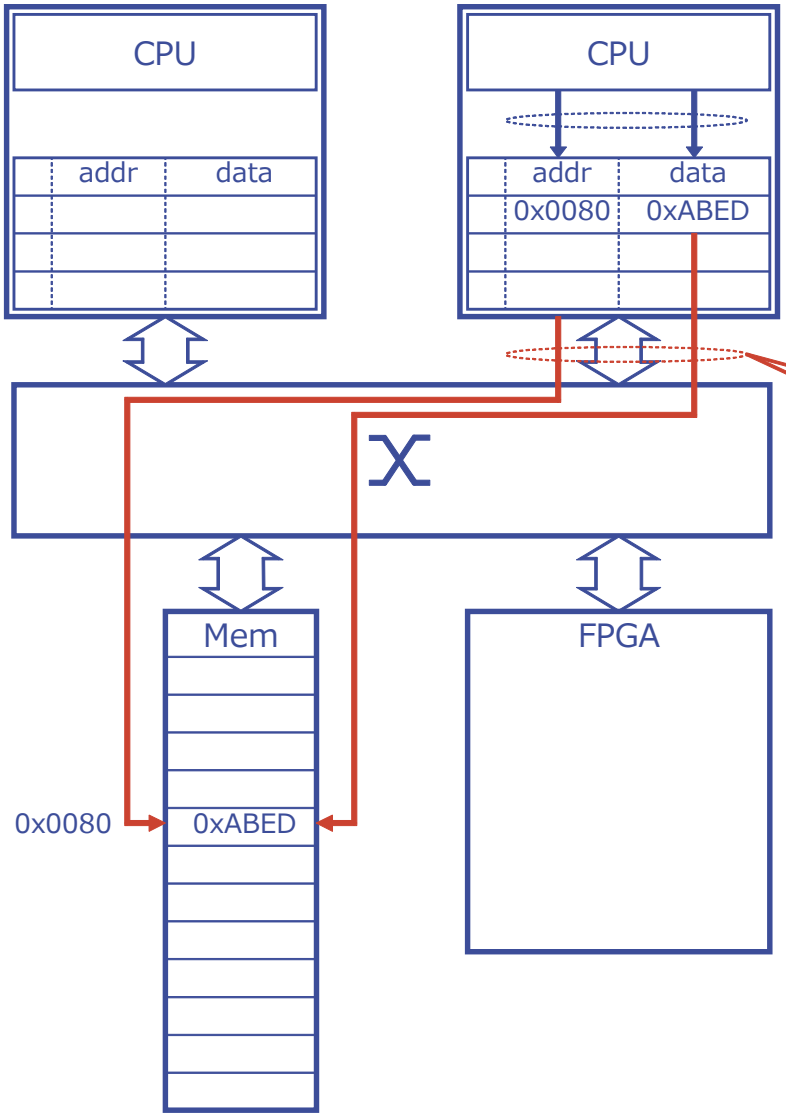
ソフトウェアで Cache を制御してトラブル回避　ケース１



ソフトウェアで Cache を制御してトラブル回避　ケース１



ソフトウェアで Cache を制御してトラブル回避　ケース１

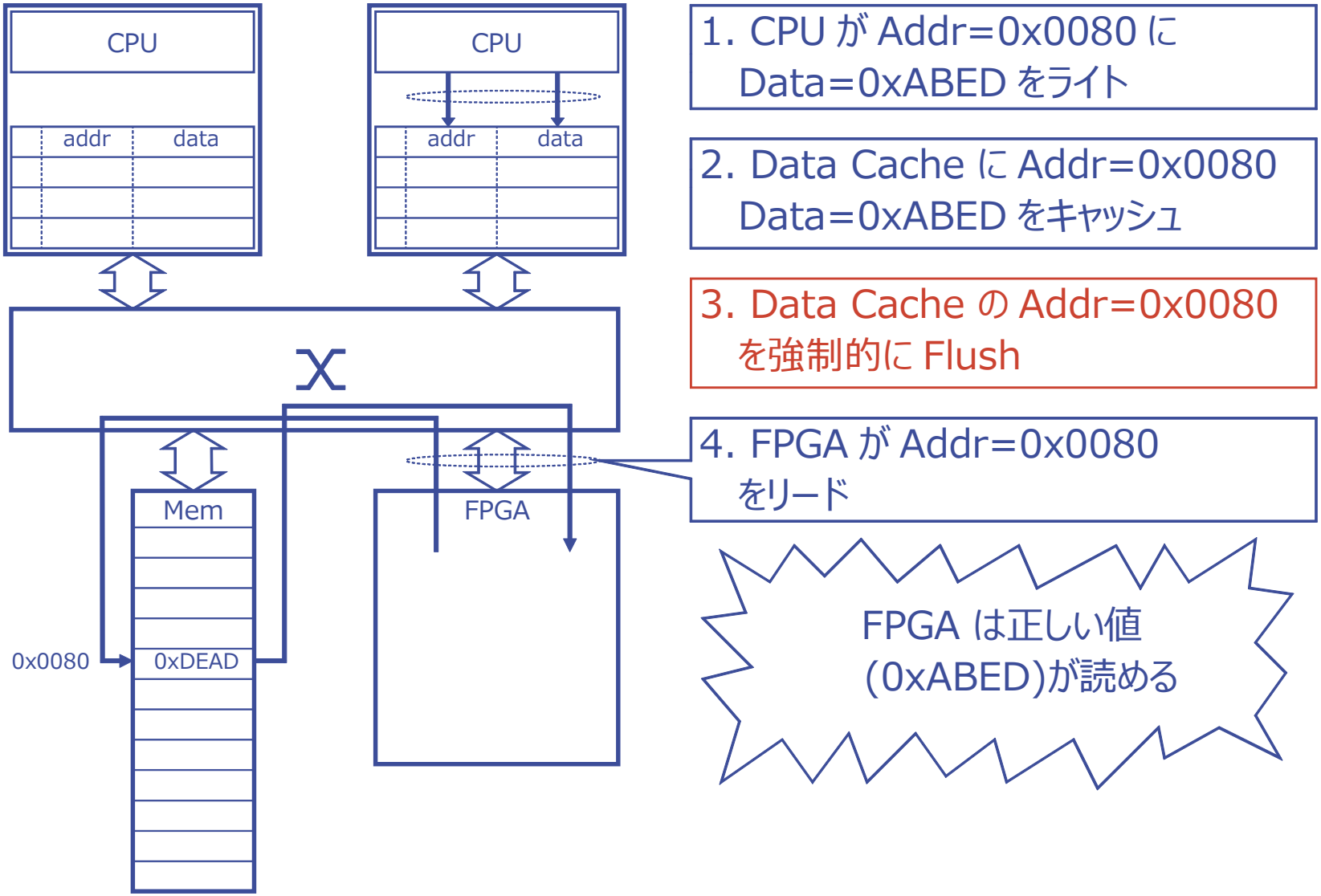


1. CPU が Addr=0x0080 に Data=0xABED をライト

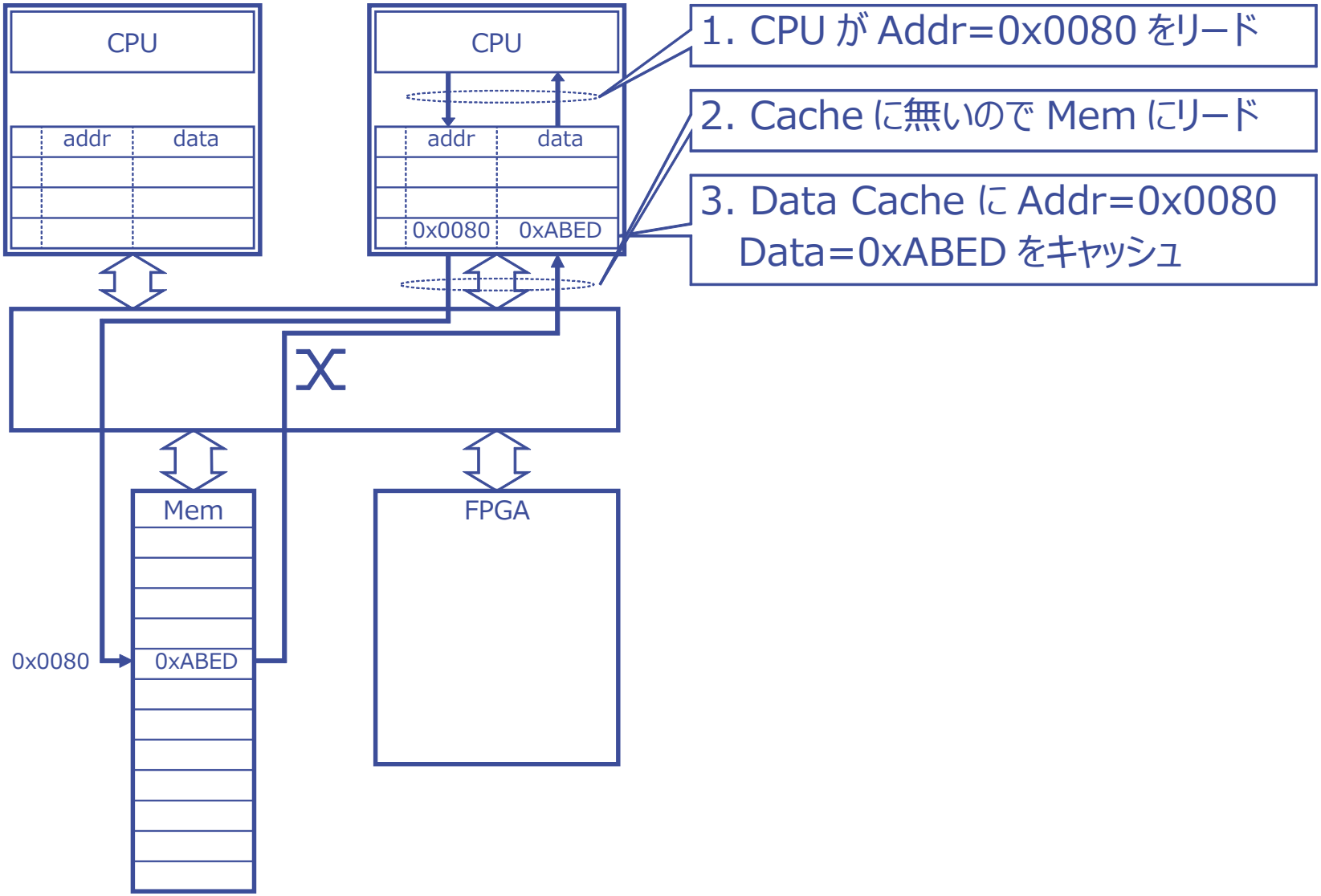
2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

3. Data Cache の Addr=0x0080 を強制的に Flush

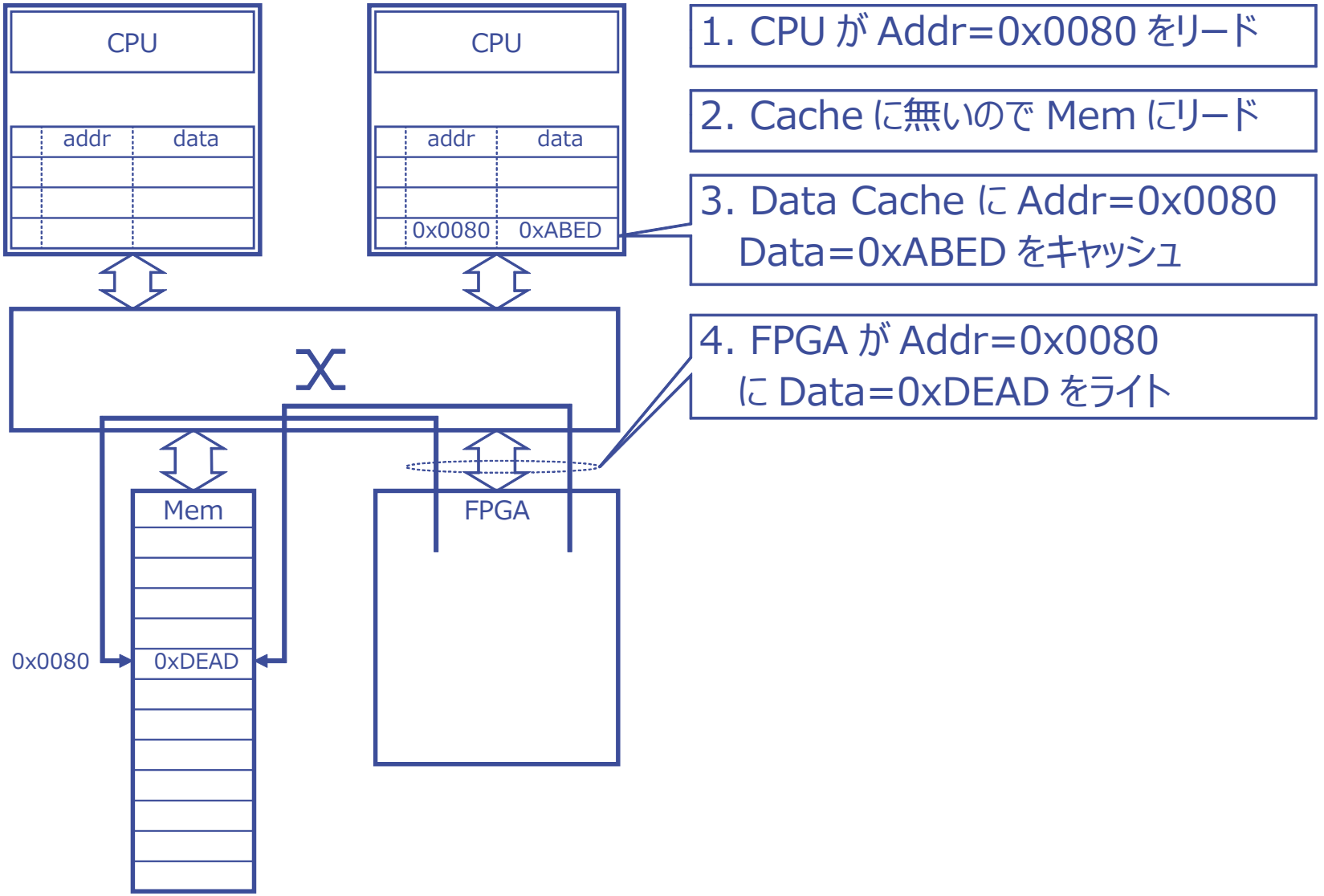
ソフトウェアで Cache を制御してトラブル回避　ケース１



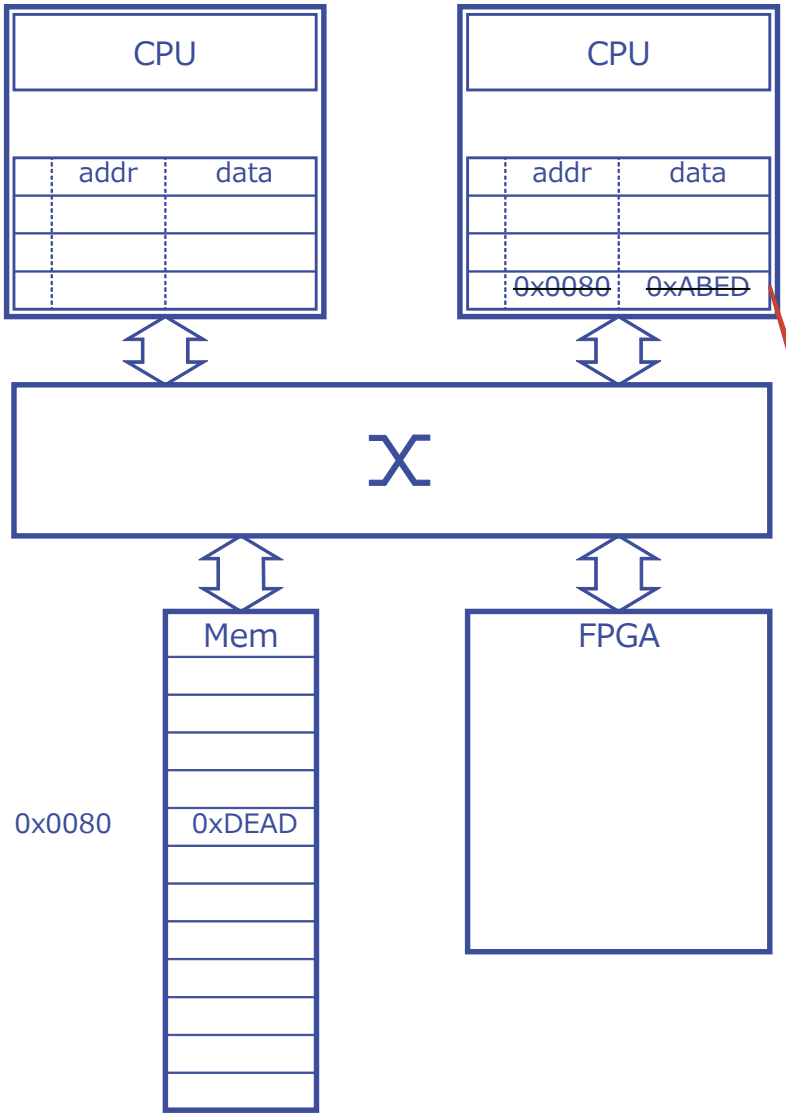
ソフトウェアで Cache を制御してトラブル回避　ケース２



ソフトウェアで Cache を制御してトラブル回避　ケース２

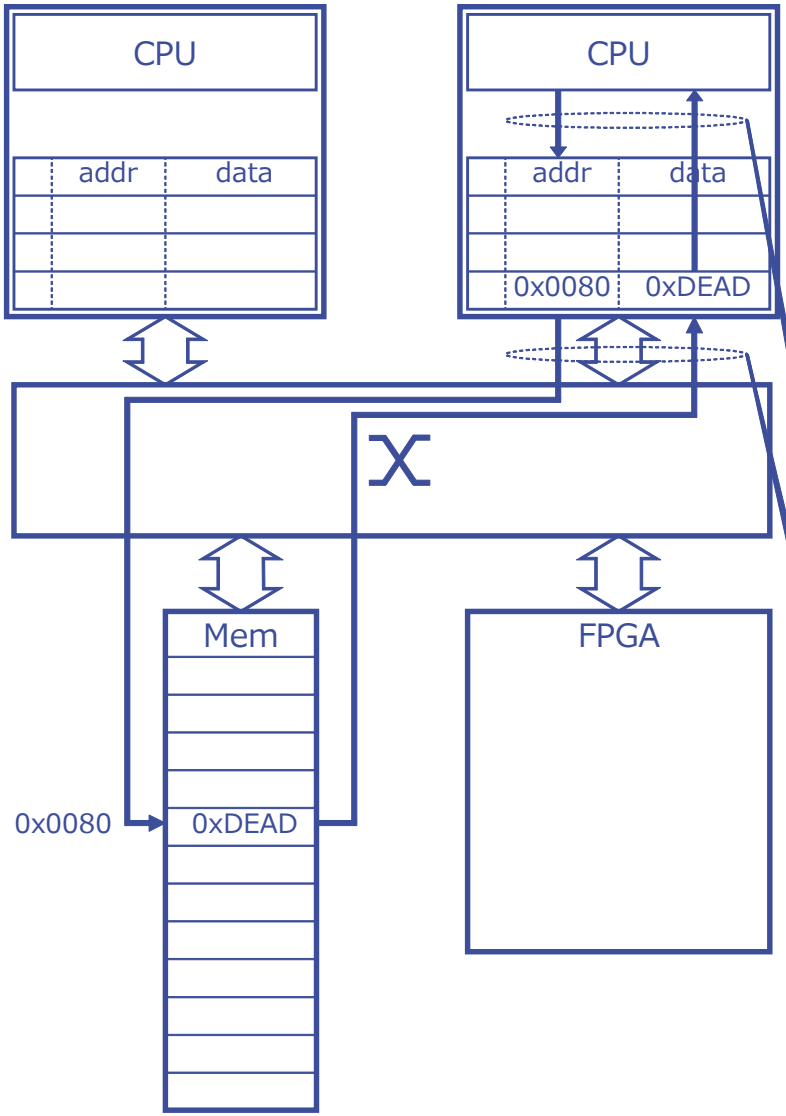


ソフトウェアで Cache を制御してトラブル回避　ケース２



1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. Data Cache の Addr=0x0080 を Invalidiate(無効化)

ソフトウェアで Cache を制御してトラブル回避　ケース２



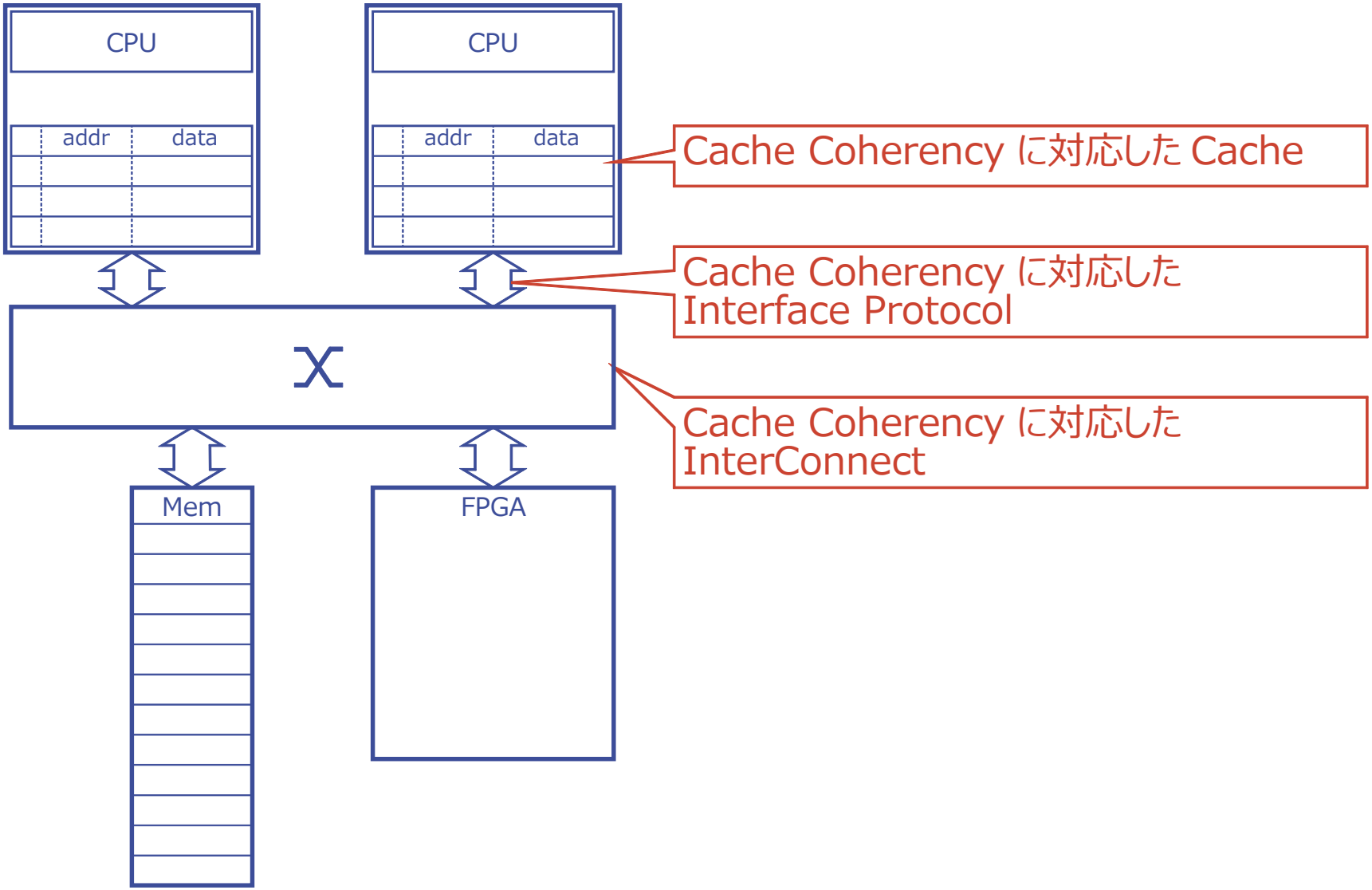
1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. Data Cache の Addr=0x0080 を Invalidiate(無効化)
6. CPU が Addr=0x0080 をリード
7. Cache に無いので Mem にリード

ソフトウェアで Cache を制御する方法の問題点

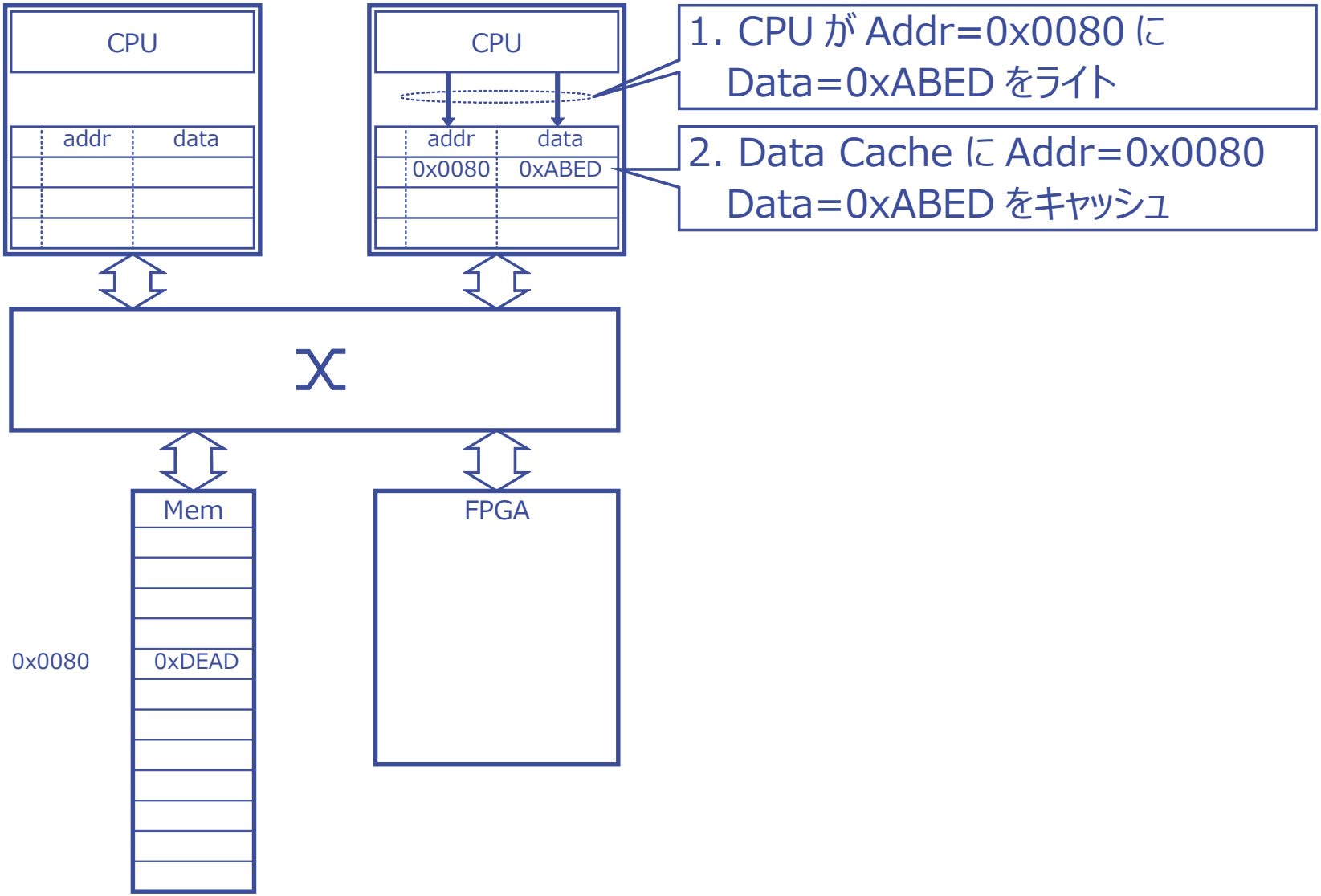
- 面倒くせ～よ
- 相手がリード/ライトするタイミングが判ってないと無理
  - CPU が DMA や FPGA を制御する場合は可能だけど、

    マルチプロセッサ間では難しい
- 時間がかかる
  - キャッシュの操作は意外と時間がかかる
  - しかもクリティカルセクション(他の処理ができない)

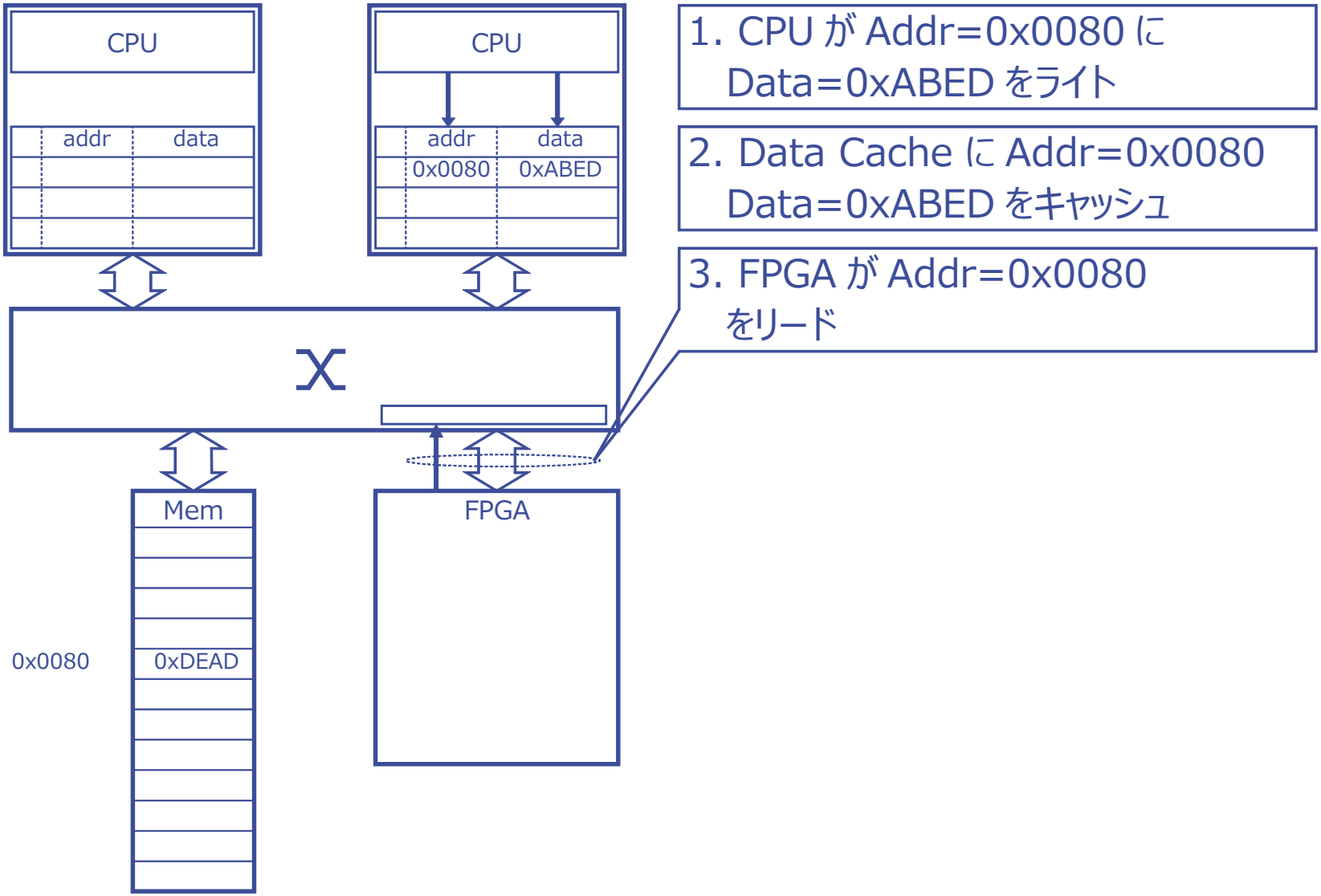
ハードウェアで Cache Coherency - 用意するもの



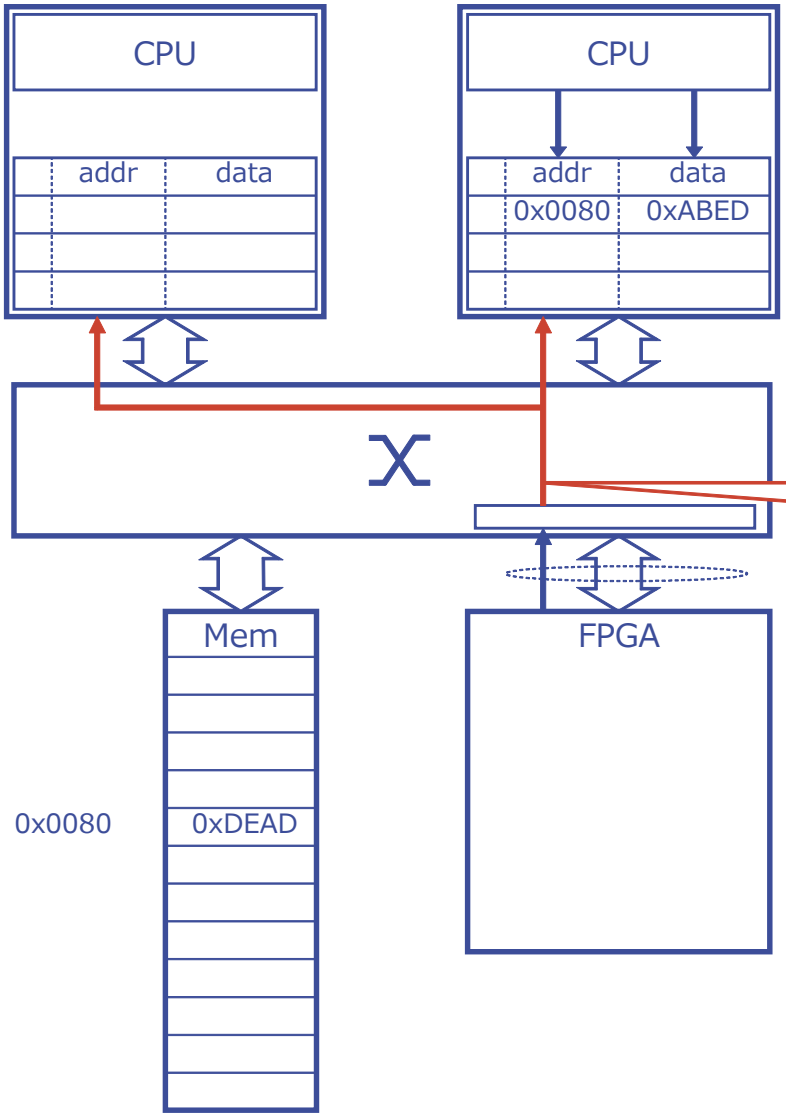
ハードウェアで Cache Coherency ケース 1



ハードウェアで Cache Coherency ケース 1



ハードウェアで Cache Coherency ケース 1



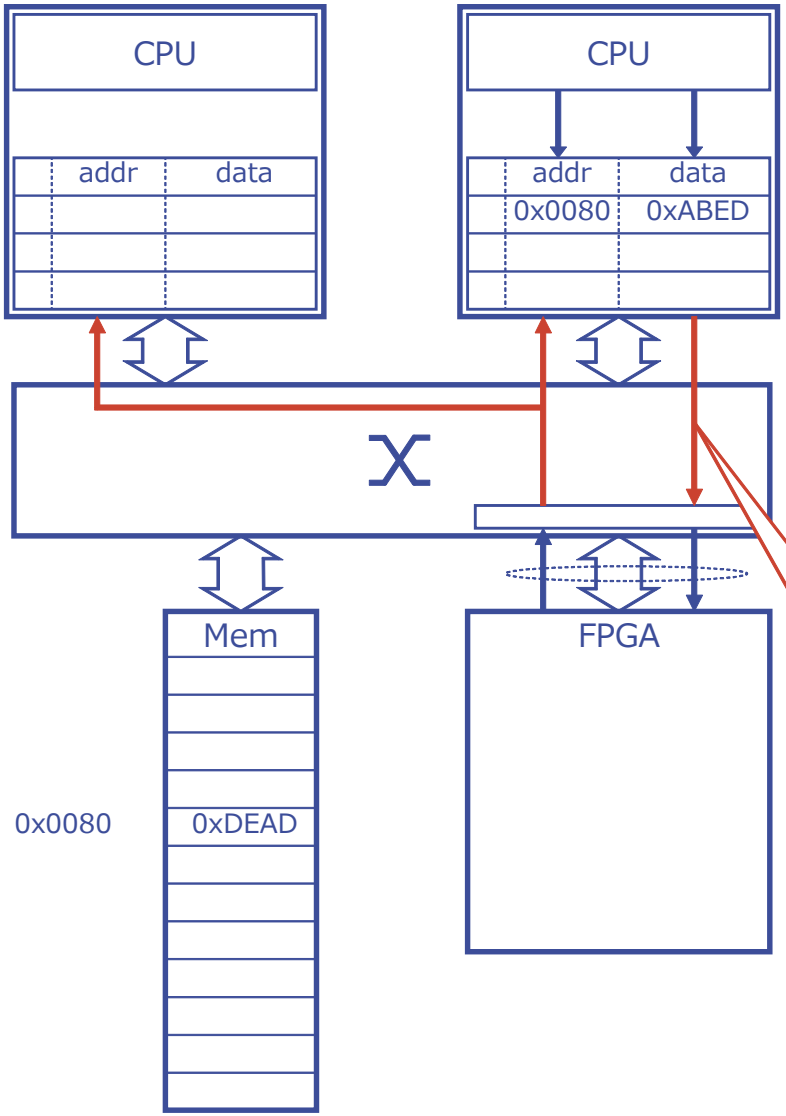
1. CPU が Addr=0x0080 に Data=0xABED をライト

2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

3. FPGA が Addr=0x0080 をリード

4.InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)

ハードウェアで Cache Coherency ケース 1



1. CPU が Addr=0x0080 に Data=0xABED をライト

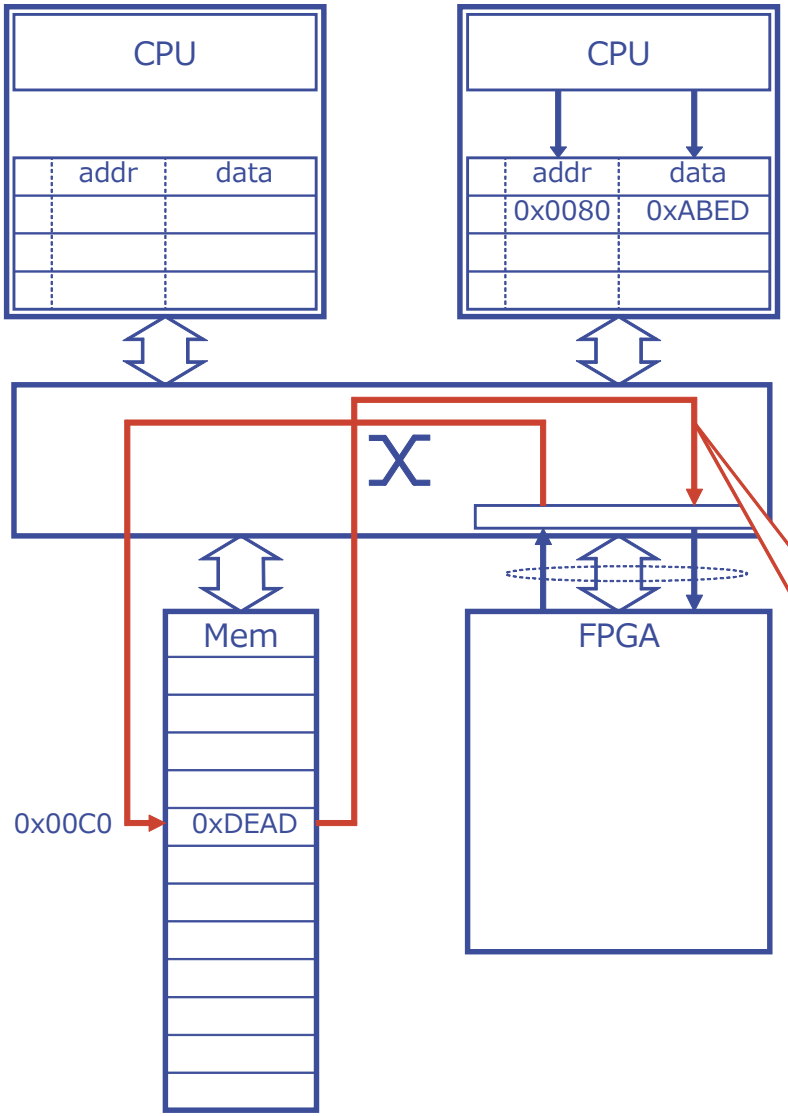
2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

3. FPGA が Addr=0x0080 をリード

4.InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)

5.該当する Addr(0x0080)のデータをキャッシュしているマスターがいる場合はマスターがデータを出力

ハードウェアで Cache Coherency ケース 1



1. CPU が Addr=0x0080 に Data=0xABED をライト

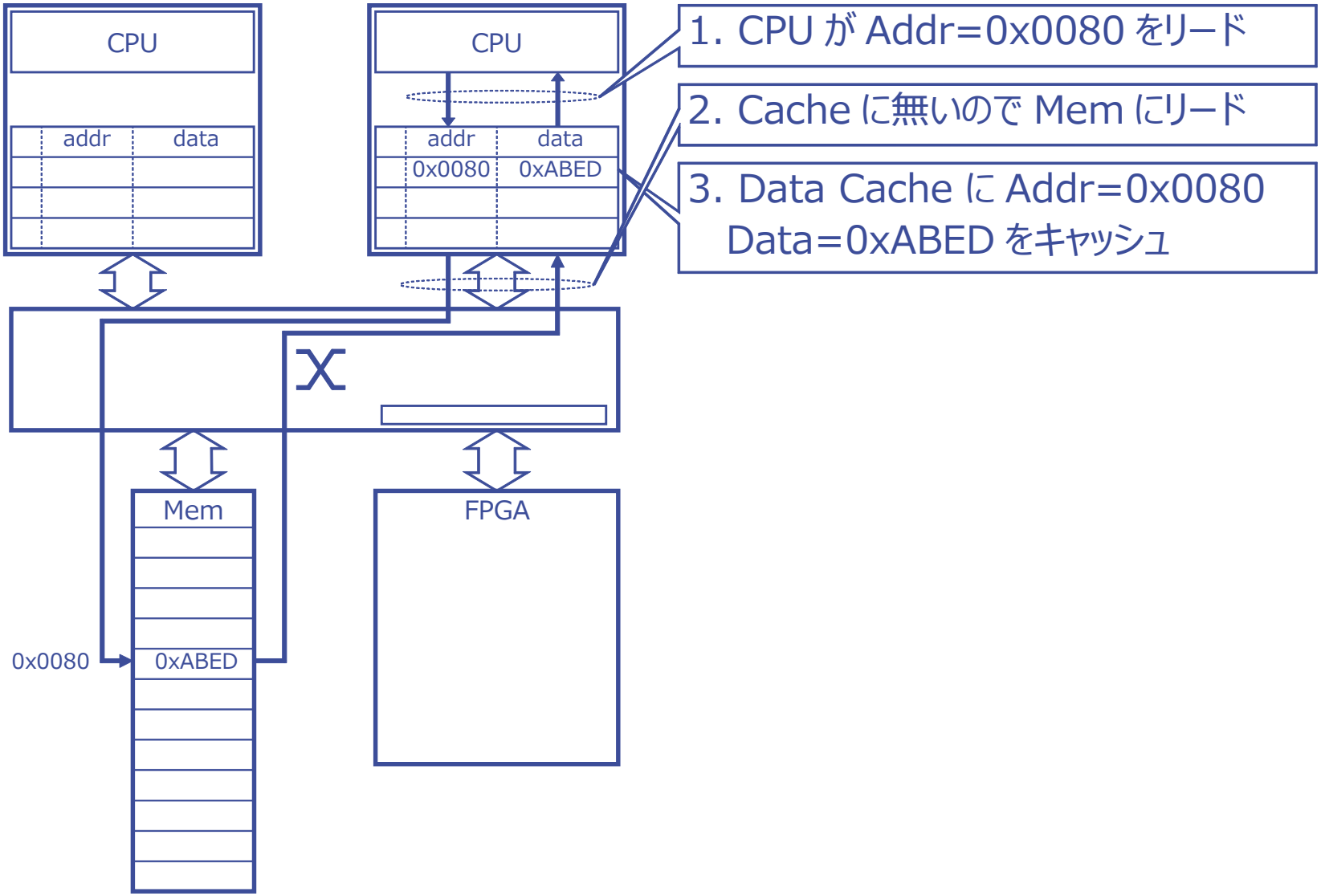
2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

3. FPGA が Addr=0x00C0 をリード

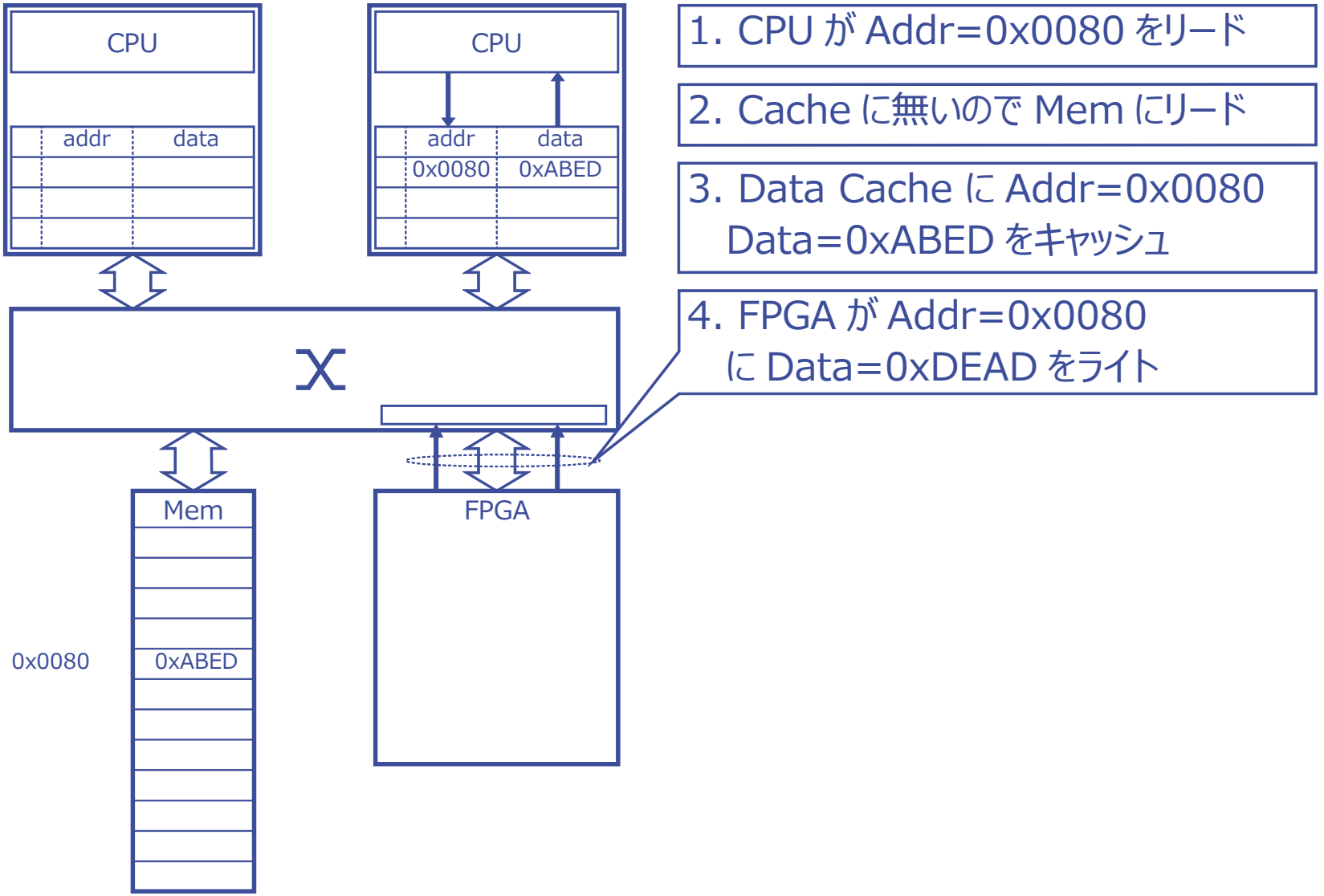
4.InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)

5.該当する Addr(0x00C0)のデータをキャッシュしているマスターがいない場合は Mem から読む

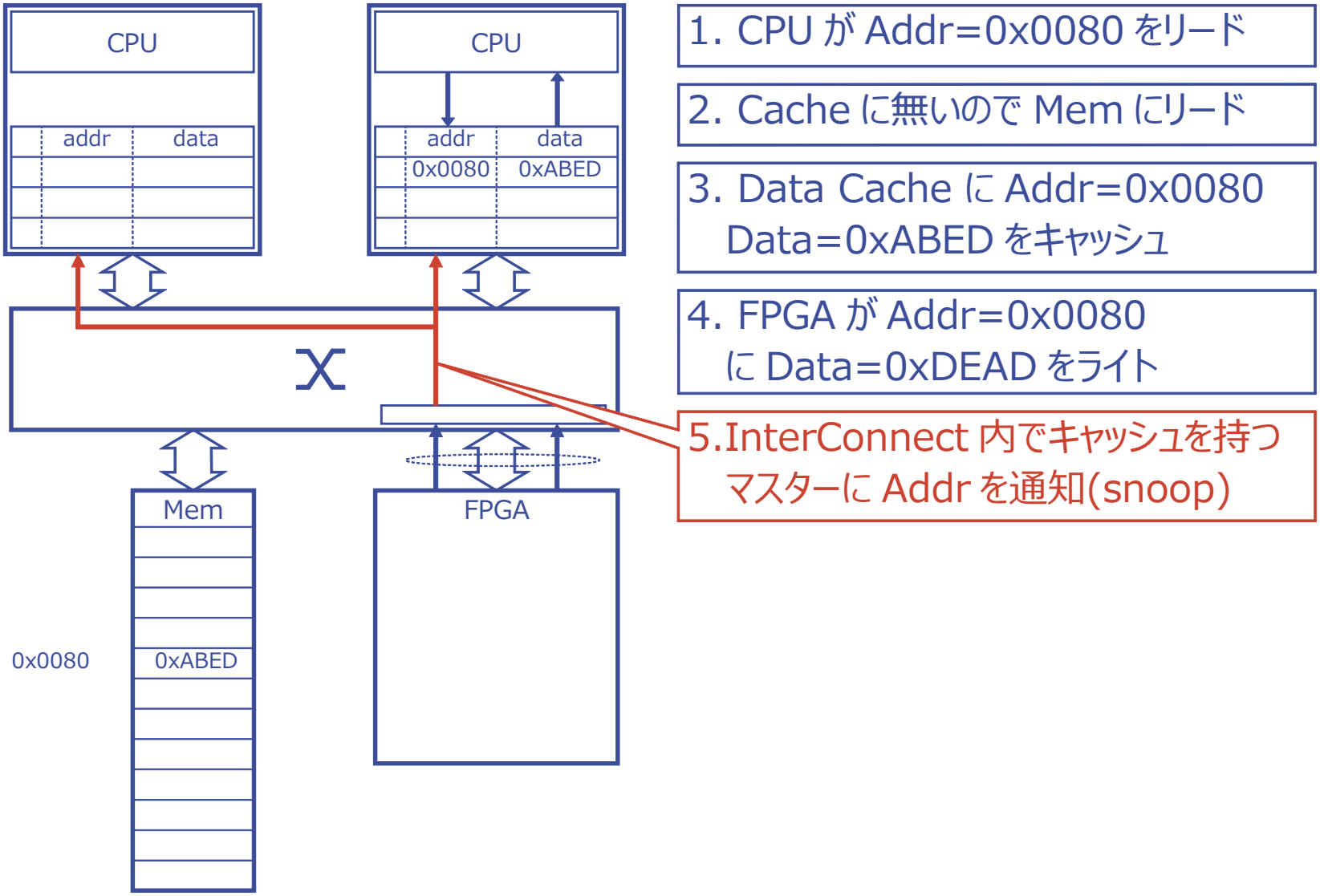
ハードウェアで Cache Coherency ケース 2



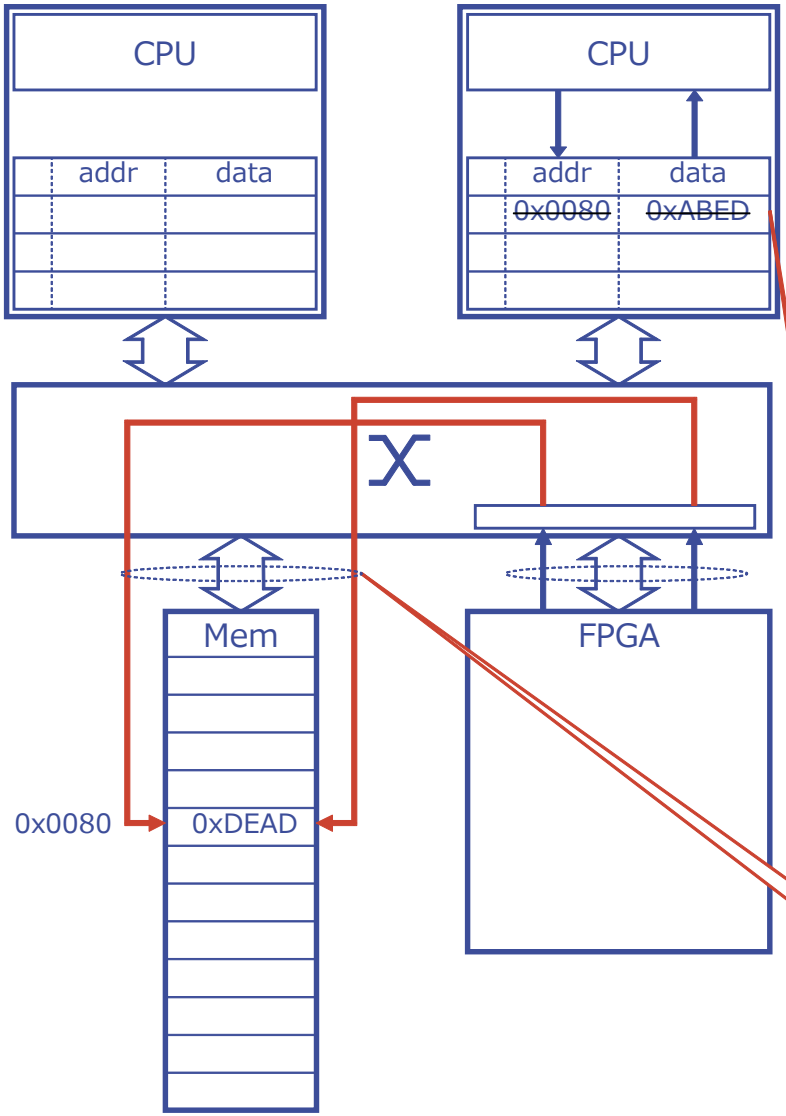
ハードウェアで Cache Coherency ケース 2



ハードウェアで Cache Coherency ケース 2

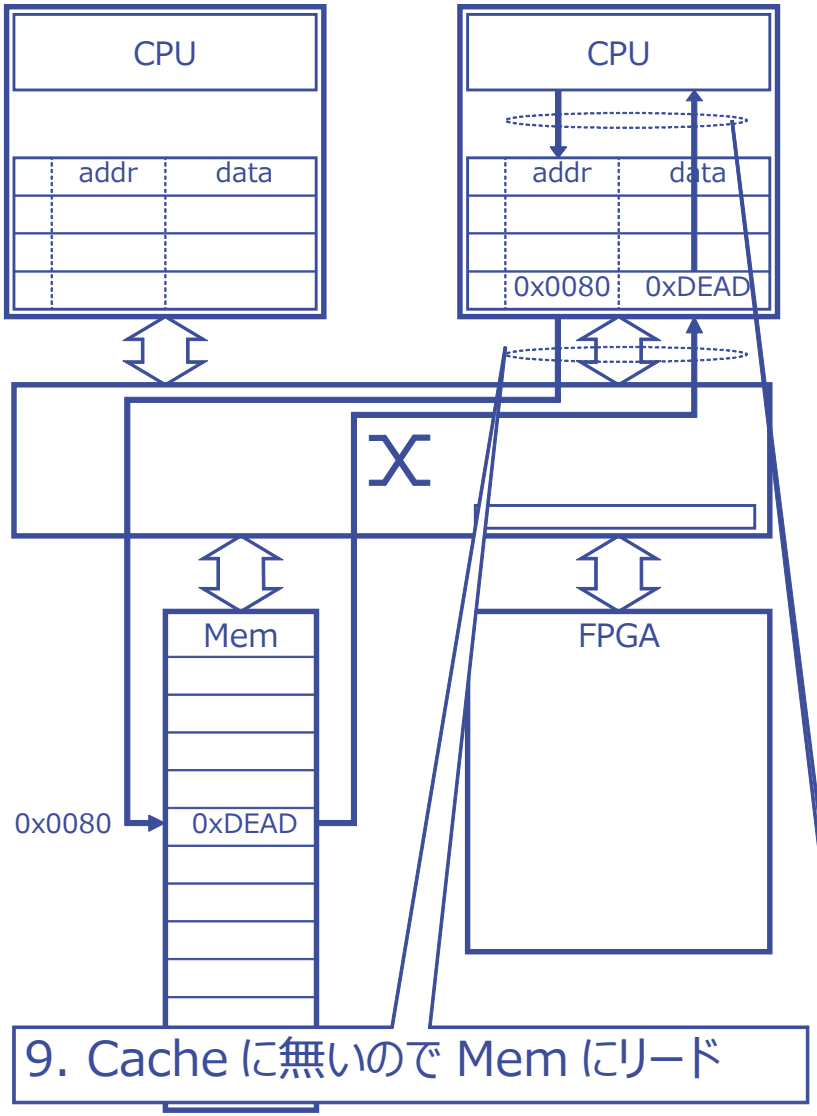


ハードウェアで Cache Coherency ケース 2 （キャッシュラインサイズの書き込みの場合）



1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)
6. 該当する Addr(0x0080)のデータをキャッシュしているマスターはキャッシュを無効化(Invalidiate)
7. Mem の Addr=0x0080 に Data=0xDEAD をライト

ハードウェアで Cache Coherency ケース 2 （キャッシュラインサイズの書き込みの場合）



1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)
6. 該当する Addr(0x0080)のデータをキャッシュしているマスターはキャッシュを無効化(Invalidiate)
7. Mem の Addr=0x0080 に Data=0xDEAD をライト
8. CPU が Addr=0x0080 をリード
9. Cache に無いので Mem にリード

Cache Coherency トラブル

- 症状
  - CPU が書いたデータが FPGA から正しく読めない
  - FPGA が書いたデータが CPU から正しく読めない
- 対策
  - キャッシュを使わない
  - ソフトウェアによる解決方法
    - ソフトウェアで Cache Flush/Invalidiate する
  - ハードウェアによる解決方法
    - Cache Coherency に対応した Cache と Interconnect

ハードウェアで Cache Coherency - Zynq の場合

- ACP(Accelarator Coherency Port)を使う
  - Zynq にはハードウェアで Cache Coherency を制御できる専用のアクセスポートがあります
  - HP(High Performance Port - Cache Coherency を制御しないポート) に比べて 30%~70%の帯域しかでません
    - 参照: FPGA の部屋 『 Zynq の AXI_ACP ポートと AXI_HP ポートの性能の違い 1 （AXI_ACP ポート)』 (http://marsee101.blog19.fc2.com/blog-entry-2773.html)
  - しかしソフトウェアで Cache を操作する苦労をしなくて済みます
  - とりあえず動くことを確認したい時には便利です

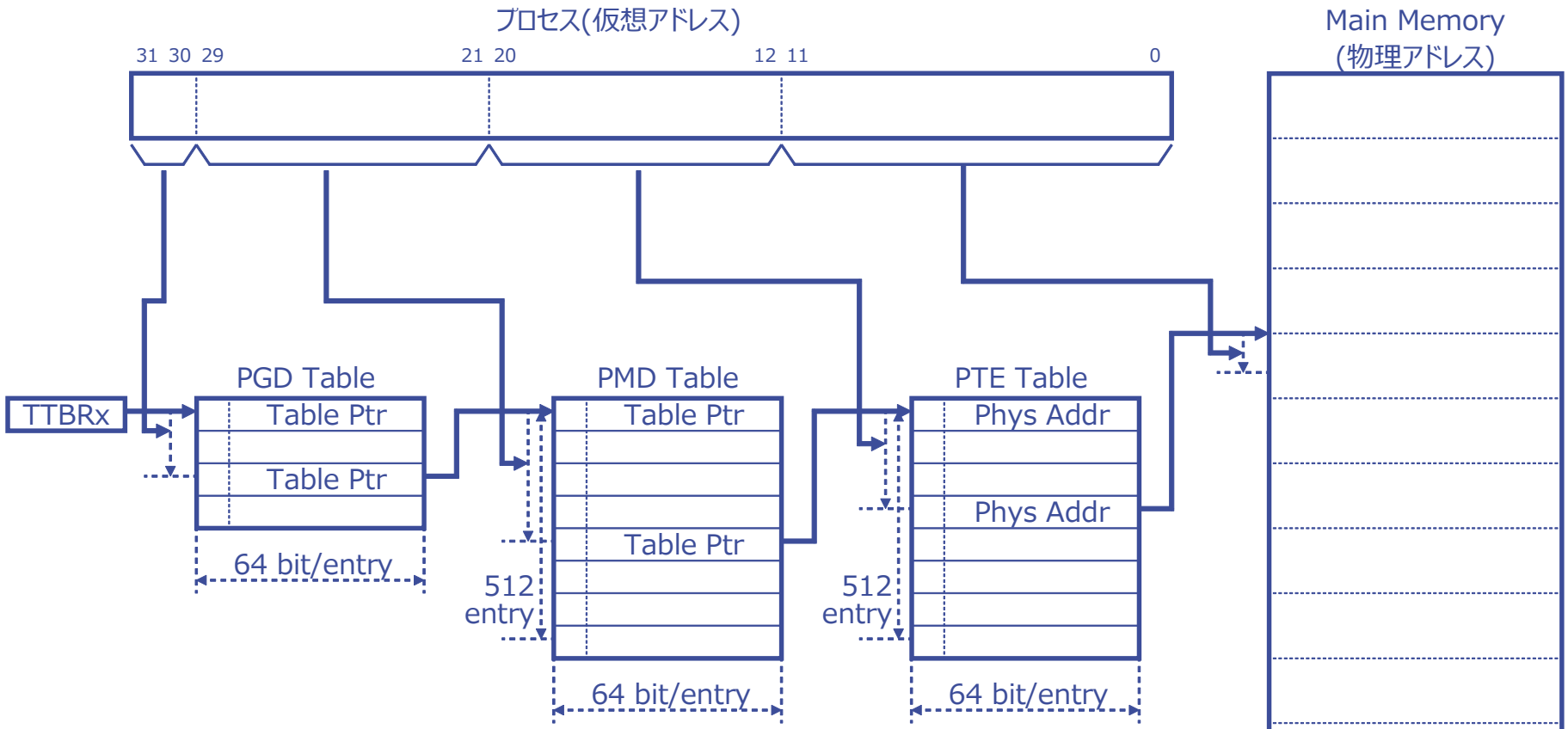
- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

∧
without
device driver

Memory Management Unit の働き

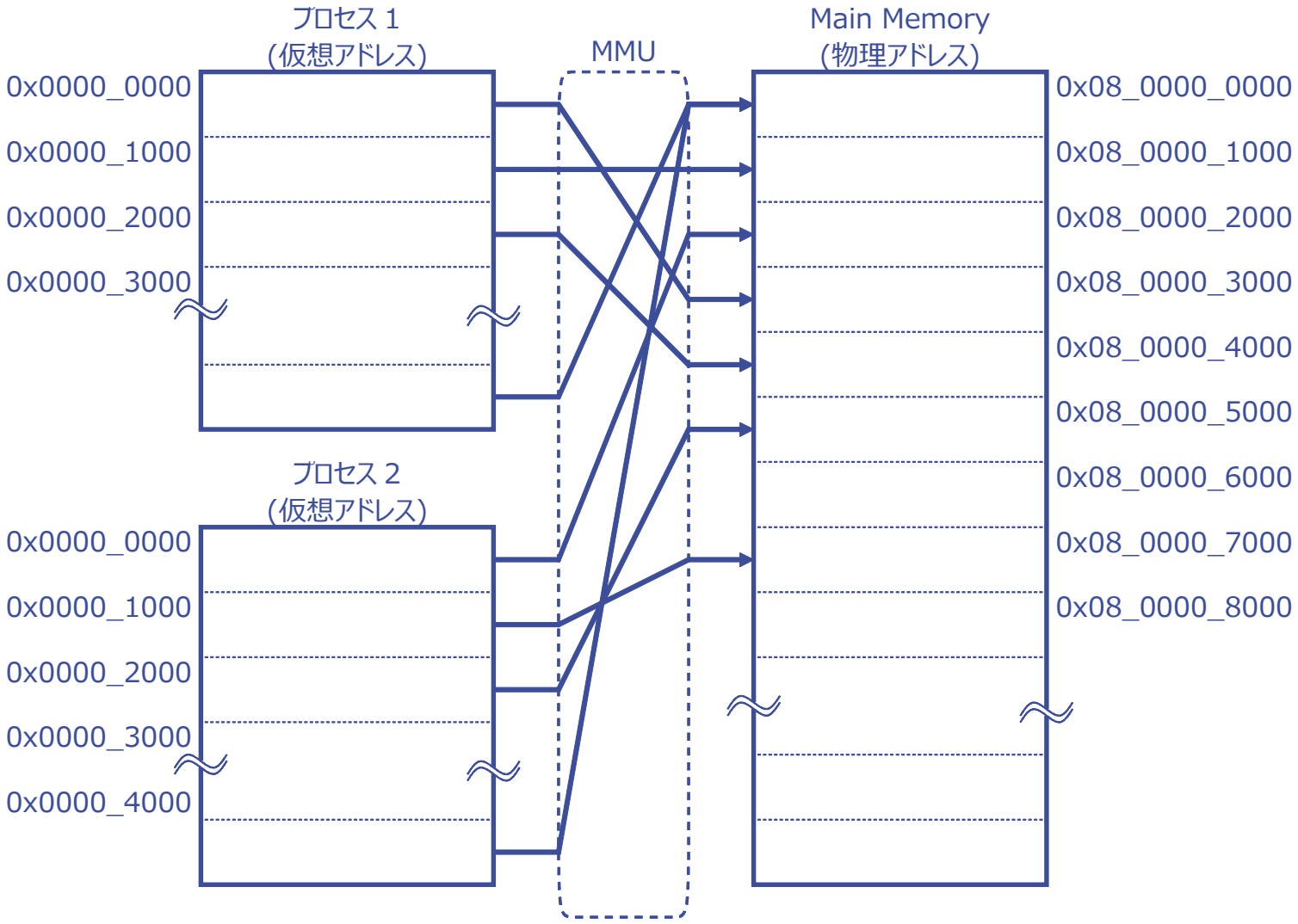
- 仮想アドレスから物理アドレスへの変換
  - 仮想記憶 - 個々のプロセスからは単一のメモリマップ
  - 物理メモリの有効利用
  - 物理アドレス空間と仮想アドレス空間の分離
- メモリ保護
- キャッシュ制御

仮想アドレスから物理アドレスへの変換(Aarch32-LPAE の例)

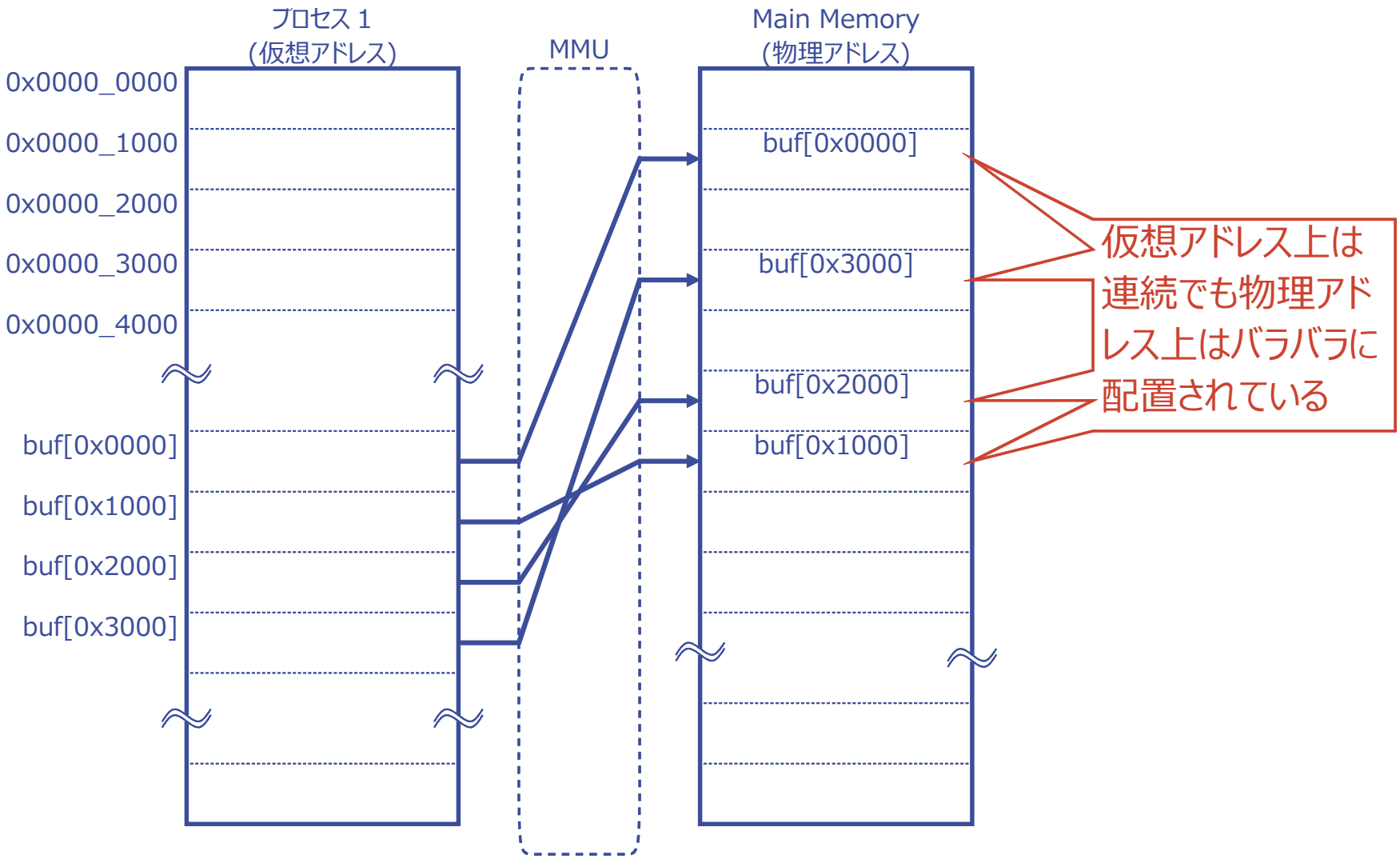


LPAE : Large Physical Address Extension
PTE : Page Table Entry
PMD : Page Middle Directory
PGD : Page Global Directory

仮想記憶 - 個々のプロセスからは単一のメモリマップ



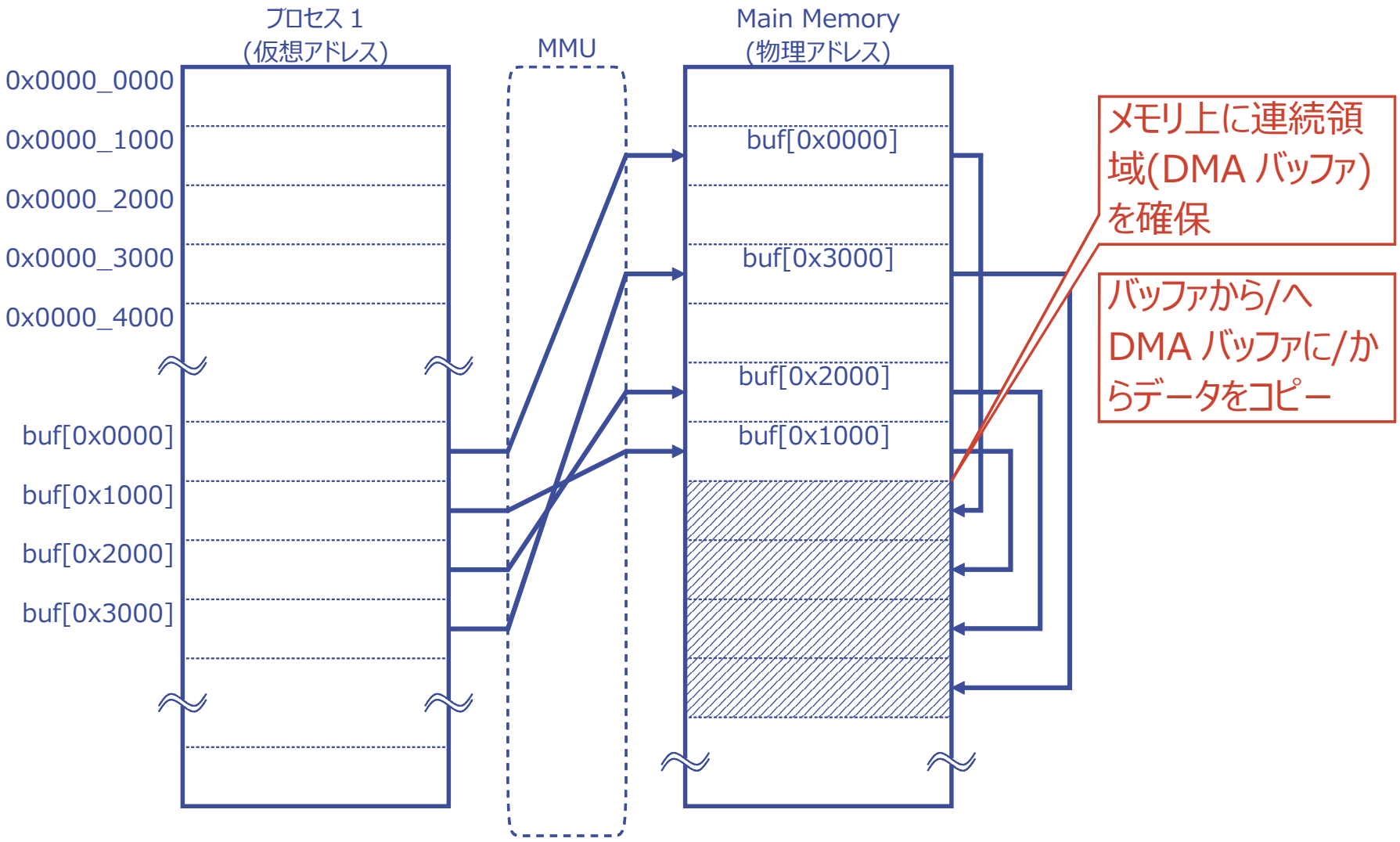
FPGA から見た場合の問題



FPGA からバラバラに配置されたバッファへのアクセス方法

- ハードウェアのアシスト有り (今回は説明対象外)
    - Scatter-Gather DMA
    - IOMMU
- ハードウェアのアシスト無し
    - 物理メモリ上に連続領域(DMA バッファ)を確保
        - バッファから/へ DMA バッファへ/からデータをコピー
        - DMA バッファをユーザープロセスからアクセス

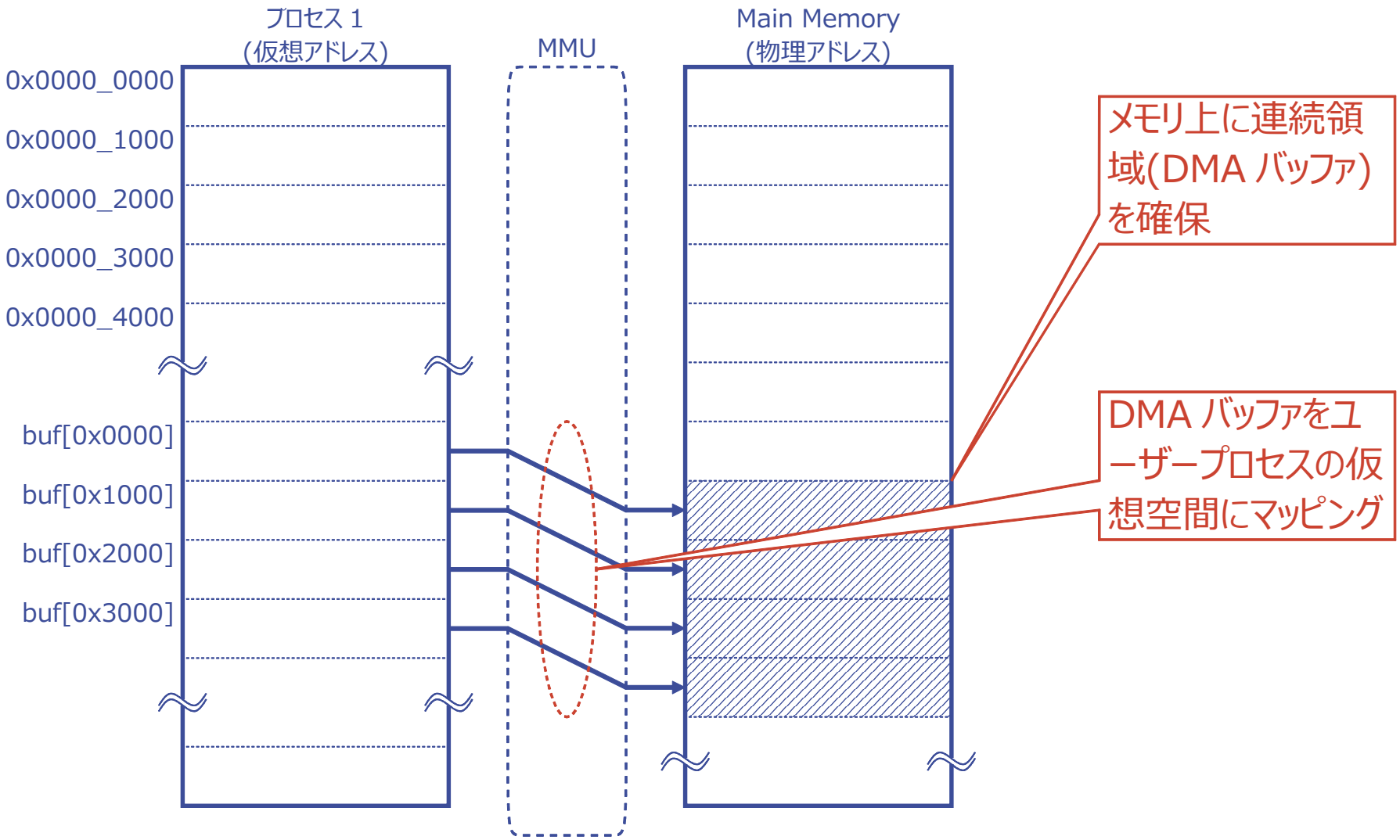
バッファから/へ DMA バッファへ/からデータをコピー



バッファから/へ DMA バッファへ/からデータをコピー

- 良い点
  - 特殊なハードウェアを必要としない
- 悪い点
  - データのコピーによる性能劣化
- 課題
  - 連続領域(DMA バッファ)の確保方法(後述)
- 例
  - レガシーなデバイス(一昔前はわりとメジャーな方法)
  - 今でも低速なデバイス(UART など)ではよく使われる

DMA バッファをユーザープロセスからアクセス



DMA バッファをユーザープロセスからアクセス

- 良い点
  - 特殊なハードウェアを必要としない
- 悪い点
  - ユーザープログラム側に(mmap()を使う等)対処が必要
- 課題
  - 連続領域(DMA バッファ)の確保方法(後述)
- 例
  - UIO(User space I/O)
  - /dev/mem
  - udmabuf

連続領域(DMA バッファ)の確保方法 - Linux の場合

- Linux の管理外領域に連続領域を確保
  - メモリ管理をユーザープログラムが行う必要がある
  - 領域は Linux 起動時に確保
  - /dev/mem や uio の mmap を使う場合はキャッシュ対象外
- CMA(Contiguous Memory Allocator)を使う
  - メモリ管理は Linux のカーネルが行う
  - CMA 領域の最大値は Linux 起動時に指定
  - "ある程度"動的にバッファを確保できる(フラグメンテーション問題)
  - キャッシュの対象

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

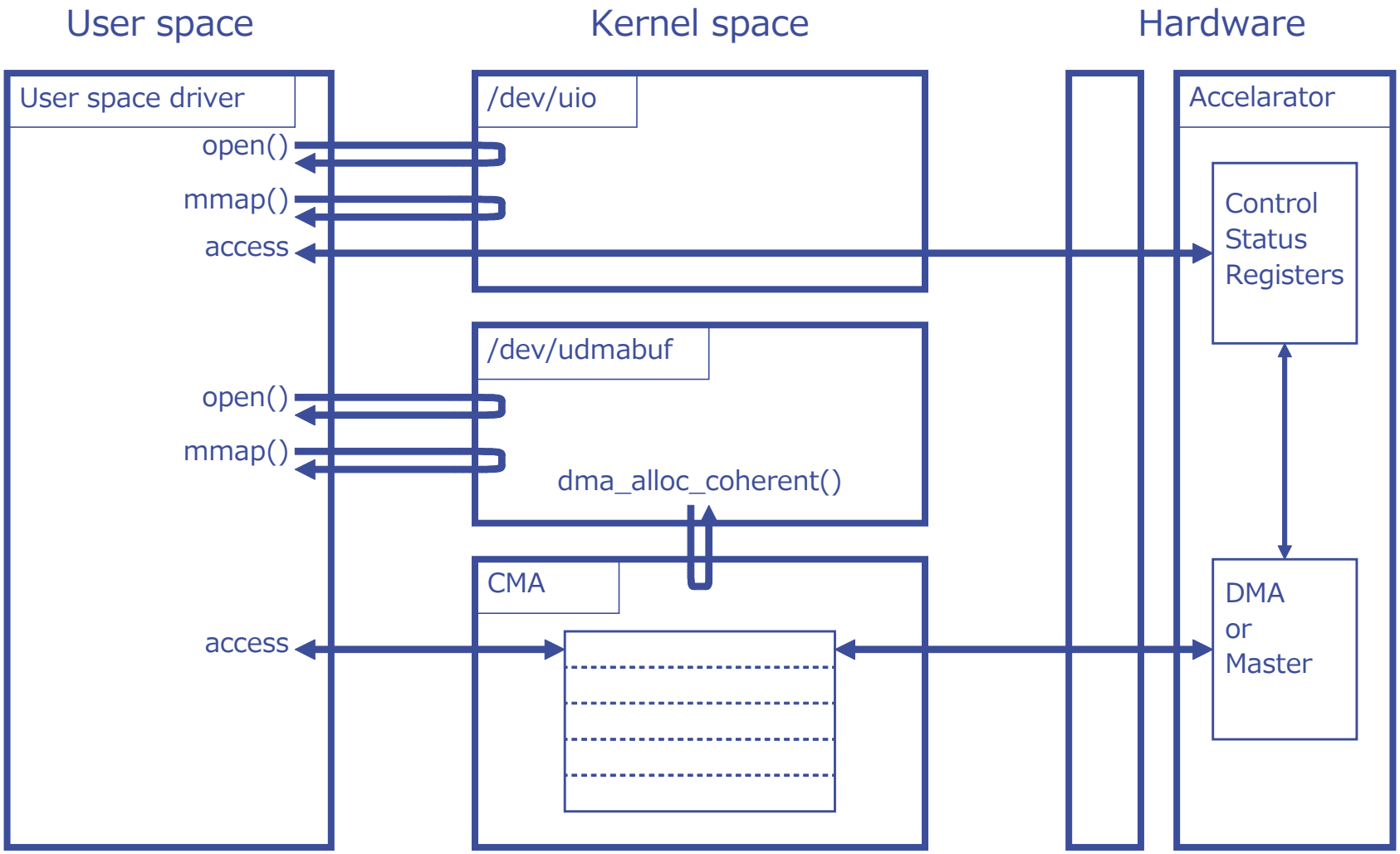
∧
without
device driver

udmabuf の紹介

- CMA の問題
  - ユーザー空間からは使えない(カーネル専用)
  - CMA を使うにはカーネルドライバを作る必要がある

- udmabuf を作りました
  - User mappable space DMA Buffer
  - CMA 領域に DMA バッファとして連続領域を確保
  - 確保した DMA バッファをユーザー空間にマッピングするための
    デバイスドライバ
  - https://github.com/ikwzm/udmabuf

udmabuf のアーキテクチャ



udmabuf デバイスドライバのロード/アンロード (FPGA-SoC-Linux)

- systemd による udmabuf.ko の制御
  - FPGA-SoC-Linux には udmabuf のデバイスドライバがインストールされています
  - systemctl コマンドでロード/アンロードできます

```
shell$ sudo systemctl status udmabuf.service
● udmabuf.service - User space mappable DMA Buffer Service.
   Loaded: loaded (/etc/systemd/system/udmabuf.service; enabled; vendor preset:
   Active: active (exited) since Fri 2017-11-03 17:38:47 JST; 8min ago
  Process: 2514 ExecStart=/sbin/modprobe udmabuf (code=exited, status=0/SUCCESS)
 Main PID: 2514 (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/udmabuf.service

Nov 03 17:38:47 debian-fpga systemd[1]: Starting User space mappable DMA Buffer Service....
Nov 03 17:38:47 debian-fpga systemd[1]: Started User space mappable DMA Buffer Service....
```

デバイスの追加 - DMA バッファの確保 (1)

- Device Tree Overlay のサンプルソース

```
/dts-v1/; /plugin/;
/ {
        fragment@1 {
                target-path = "/amba";
                #address-cells = <1>;
                #size-cells = <1>;
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <1>;
                        udmabuf0@0x00 {       /* 追加するデバイスツリーノード */
                                compatible = "ikwzm,udmabuf-0.10.a";
                                device-name = "udmabuf0";     /* デバイス名 */
                                size = <0x00100000>;          /* バッファサイズ*/
                        };
                };
        };
};
```

デバイスの追加 - DMA バッファの確保 (2)

- Device Tree Overlay でツリーに追加する

```
shell# dtbocfg.rb --install udmabuf0 --dts udmabuf0.dts
udmabuf udmabuf0: driver installed
udmabuf udmabuf0: major number       = 248
udmabuf udmabuf0: minor number       = 0
udmabuf udmabuf0: phys address       = 0x1e900000
udmabuf udmabuf0: buffer size        = 1048576
shell# ls -la /dev/udmabuf0
crw------- 1 root root 248, 0 Dec  1 09:34 /dev/udmabuf0
```

- Device Tree に追加することで初めてバッファが確保される
- Device Tree に追加することで初めてデバイスドライバ(この例では/dev/udmabuf0)が作成される

DMA バッファをユーザー空間へマッピング (1) C 言語

- udmabuf デバイスを mmap() でマッピング

```
if ((fd  = open("/dev/udmabuf0", O_RDWR)) != -1) {
        buf = mmap(NULL, buf_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        /* Do some read/write access to buf */
        close(fd);
}
```

  - open() 時のオプションに O_SYNC を付けるとキャッシュオフ
  - open() 時のオプションに O_SYNC を付けないとキャッシュオン

DMA バッファをユーザー空間へマッピング (2) Python+NumPy

・udmabuf デバイスを np.memmap() でマッピング

```
import numpy as np

class Udmabuf:
    """A simple udmabuf class"""

    def __init__(self, name):
        self.name        = name
        self.device_name = '/dev/%s' % self.name
        self.class_path  = '/sys/class/udmabuf/%s' % self.name
        for line in open(self.class_path + '/size'):
            self.buf_size = int(line)
            break
        for line in open(self.class_path + '/phys_addr'):
            self.phys_addr = int(line, 16)
            break

    def memmap(self, dtype, shape):
        self.item_size = np.dtype(dtype).itemsize
        self.array = np.memmap(self.device_name, dtype=dtype, mode='r+', shape=shape)
        return self.array
```

DMA バッファをユーザー空間へマッピング (3) Python+NumPy

- 確保した DMA バッファを np.array として使う

```
if __name__ == '__main__':
    udmabuf     = Udmabuf('udmabuf0')
    test_dtype  = np.uint8
    test_size   = int(udmabuf.buf_size/(np.dtype(test_dtype).itemsize))

    comparison = np.random.randint(0,255,(test_size))

    ubuf_array  = udmabuf.memmap(dtype=test_dtype, shape=(test_size))

    ubuf_array[:] = comparison

    if np.array_equal(ubuf_array, comparison):
        print ("ubuf_array == comparison : OK")
    else:
        print ("ubuf_array != comparison : NG")
```

DMA バッファのサイズを得る (1) C 言語

- /sys/class/udmabuf/<*device-name*>/size を読む

```
unsigned char   attr[1024];
unsigned long   buf_size;
if ((fd  = open("/sys/class/udmabuf/udmabuf0/size", O_RDONLY)) != -1) {
        read(fd, attr, 1024);
        sscanf(attr, "%d", &buf_size);
        close(fd);
}
```

DMA バッファのサイズを得る (2) Python

- /sys/class/udmabuf/<*device-name*>/size を読む

```
import numpy as np

class Udmabuf:
    """A simple udmabuf class"""

    def __init__(self, name):
        self.name        = name
        self.device_name = '/dev/%s' % self.name
        self.class_path  = '/sys/class/udmabuf/%s' % self.name
        for line in open(self.class_path + '/size'):
            self.buf_size = int(line)
            break
        for line in open(self.class_path + '/phys_addr'):
            self.phys_addr = int(line, 16)
            break

    def memmap(self, dtype, shape):
        self.item_size = np.dtype(dtype).itemsize
        self.array = np.memmap(self.device_name, dtype=dtype, mode='r+', shape=shape)
        return self.array
```

DMA バッファの物理アドレスを得る (1) C 言語

- /sys/class/udmabuf/<*device-name*>/phys_addr を読む

```
unsigned char   attr[1024];
unsigned long   phys_addr;

 if ((fd  = open("/sys/class/udmabuf/udmabuf0/phys_addr", O_RDONLY)) != -1) {
          read(fd, attr, 1024);
          sscanf(attr, "%x", &phys_addr);
          close(fd);
}
```

DMA バッファの物理アドレスを得る (2) Python

- /sys/class/udmabuf/<*device-name*>/phys_addr を読む

```
import numpy as np

class Udmabuf:
    """A simple udmabuf class"""

    def __init__(self, name):
        self.name           = name
        self.device_name    = '/dev/%s'  % self.name
        self.class_path     = '/sys/class/udmabuf/%s'  % self.name
        for line in open(self.class_path + '/size'):
            self.buf_size = int(line)
            break
        for line in open(self.class_path + '/phys_addr'):
            self.phys_addr = int(line, 16)
            break

    def memmap(self, dtype, shape):
        self.item_size = np.dtype(dtype).itemsize
        self.array = np.memmap(self.device_name, dtype=dtype, mode='r+', shape=shape)
        return self.array
```

udmabuf の記事 @Qiita

- 『 Linux でユーザー空間で動作するプログラムとハードウェアがメモリを共有するためのデバイスドライバ』

  https://qiita.com/ikwzm/items/cc1bb33ff43a491440ea
- 『 Linux でユーザー空間で動作するプログラムとハードウェアがメモリを共有するためのデバイスドライバ(Device Tree Overlay 対応)』

  https://qiita.com/ikwzm/items/db904df23748e4b957b5
- 『 Linux でユーザー空間で動作するプログラムとハードウェアがメモリを共有するためのデバイスドライバ（NumPy 対応）』

  https://qiita.com/ikwzm/items/e615ce90ca0c403b3794

- How to Configuration FPGA from PS with Linux
  - FPGA Configuration Overview
  - Device Tree Overlay
  - FPGA Region
- How to Control FPGA from PS with Linux
  - Cache Coherency
  - Memory Management Unit
  - UDMABUF
  - UIO and Interrupt

∧
without
device driver

Device Tree による UIO デバイスの追加 (1)

- Device Tree Overlay のサンプルソース

```
/dts-v1/; /plugin/;
/ {
        fragment@1 {
                target-path = "/amba";
                #address-cells = <1>;
                #size-cells = <1>;
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <1>;
                        uio0@43c10000 {
                                compatible = "generic-uio";
                                reg = <0x43c10000 0x1000>;
                                interrupts = <0x0 0x1d 0x4>;
                        };
                };
        };
};
```

Device Tree による UIO デバイスの追加 (2)

- compatible プロパティで uio のデバイスドライバ名を指定
- reg プロパティで レジスタのアドレスと範囲を指定
  - 1 番目のパラメータが CPU レジスタからみたアドレス
  - 2 番目のパラメータでレジスタ領域の大きさ
- interrupt プロパティで割り込み番号を指定
  - 1 番目のパラメータは GIC 内部の割り込みの種類
    0=共有ペリフェラル割り込み(SPI)
  - 2 番目のパラメータは割り込み番号
    IRQ_F2P[0]は SPI の 61 番に接続されている
    Device Tree では 61 から 32 を引いた値を指定する
  - 3 番目のパラメータはトリガータイプ
    1=エッジトリガー / 4=レベルトリガー

Device Tree による UIO デバイスの追加 (3)

- Device Tree Overlay でツリーに追加する

```
shell# dtbocfg.rb --install uio0 --dts uio0.dts
shell# ls -la /dev/uio0
crw------- 1 root root 248, 0 Dec  1 09:34 /dev/uio0
```

- Device Tree に追加することで初めてデバイスドライバ(この例では/dev/uio0)が作成される

レジスタ空間をユーザー空間へマッピング (1) C 言語

- uio デバイスを mmap() でマッピング

```
int       uio_fd;
void*     regs;

if ((uio_fd = open("/dev/uio0", O_RDWR | O_SYNC)) == -1) {
          printf("Can not open /dev/uio0¥n");
          exit(1);
}

regs = mmap(NULL, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED, uio_fd, 0);
if (regs == NULL) {
          printf("Can not mmap /dev/uio0¥n");
          exit(1);
}
```

レジスタ空間をユーザー空間へマッピング (2) Python+NumPy

・uio デバイスを mmap.mmap() でマッピング

```
import numpy as np
import mmap

class Uio:
    """A simple uio class"""

    def __init__(self, name, length=0x1000):
        self.name = name
        self.device_name = '/dev/%s' % self.name
        self.length =  length
        self.device_file = os.open(self.device_name, os.O_RDWR | os.O_SYNC)
        self.mem = mmap.mmap(self.device_file,  self.length,
                             mmap.MAP_SHARED,
                             mmap.PROT_READ |  mmap.PROT_WRITE, offset=0)
        self.word_array = np.frombuffer(self.mem, np.uint32, self.length>>2, 0)

    def read_word(self, offset):
        return int(self.word_array[offset>>2])

    def write_word(self, offset, data):
        self.word_array[offset>>2] = np.uint32(data)
```

割り込みの許可 or 禁止の設定 (1) C 言語

- uio デバイスに write() で 1 or 0 を書く
  - uio では割り込みの許可 or 禁止の設定は デバイスファイル (/dev/uio0)に write することで設定
  - 書き込むデータは 32bit の integer
  - 1 を write することで割り込みを許可
  - 0 を write することで割り込みを禁止

```
int uio_irq_on(int uio_fd)
{
        unsigned int  irq_on = 1;
        write(uio_fd, &irq_on, sizeof(irq_on));
}
int uio_irq_off(int uio_fd)
{
        unsigned int  irq_on = 0;
        write(uio_fd, &irq_on, sizeof(irq_on));
}
```

割り込みの許可 or 禁止の設定 (2) Python

・uio デバイスに os.write() で 1 or 0 を書く

```
import numpy as np
import mmap

class Uio:
    """A simple uio class"""

    def __init__(self, name, length=0x1000):
        self.name = name
        self.device_name = '/dev/%s' % self.name
        self.length =  length
        self.device_file = os.open(self.device_name, os.O_RDWR | os.O_SYNC)
        self.mem = mmap.mmap(self.device_file,  self.length,
                             mmap.MAP_SHARED,
                             mmap.PROT_READ |  mmap.PROT_WRITE, offset=0)
        self.word_array = np.frombuffer(self.mem, np.uint32, self.length>>2, 0)

    def irq_on(self):
        os.write(self.device_file, b'¥x01¥x00¥x00¥x00')

    def irq_off(sef):
        os.write(self.device_file, b'¥x00¥x00¥x00¥x00')
```

割り込み待ち (1) C 言語

- uio デバイスを read() で読む
  - uio では read() を使って、割り込みが発生するまで読み出しプロセスをブロック
  - 読み出すデータの型は 32bit の integer

```
int uio_wait_irq(int uio_fd)
{
        unsigned int  count = 0;
        return read(uio_fd, &count,  sizeof(count));
}
```

割り込み待ち (2) Python

・uio デバイスを os.read() で読む

```
import numpy as np
import mmap

class Uio:
    """A simple uio class"""

    def __init__(self, name, length=0x1000):
        self.name = name
        self.device_name = '/dev/%s' % self.name
        self.length =  length
        self.device_file = os.open(self.device_name, os.O_RDWR | os.O_SYNC)
        self.mem = mmap.mmap(self.device_file,  self.length,
                             mmap.MAP_SHARED,
                             mmap.PROT_READ |  mmap.PROT_WRITE, offset=0)
        self.word_array = np.frombuffer(self.mem, np.uint32, self.length>>2, 0)

    def wait_irq(self):
        os.read(self.device_file, 4)
```

uio の記事

- https://github.com/ikwzm/ZYBO_UIO_IRQ_SAMPLE

- 『 UIO(User space IO)の割り込みの使い方の例』 @Qiita

  https://qiita.com/ikwzm/items/b22592c31cdbb9ab6cf7

- 『 Python と Numpy で UIO を制御』 @Qiita

  https://qiita.com/ikwzm/items/d7559858d099c86c350c