高位合成友の会 2018 秋
Cache Coherency の話

2018 年 9 月 22 日初版

@ikwzm

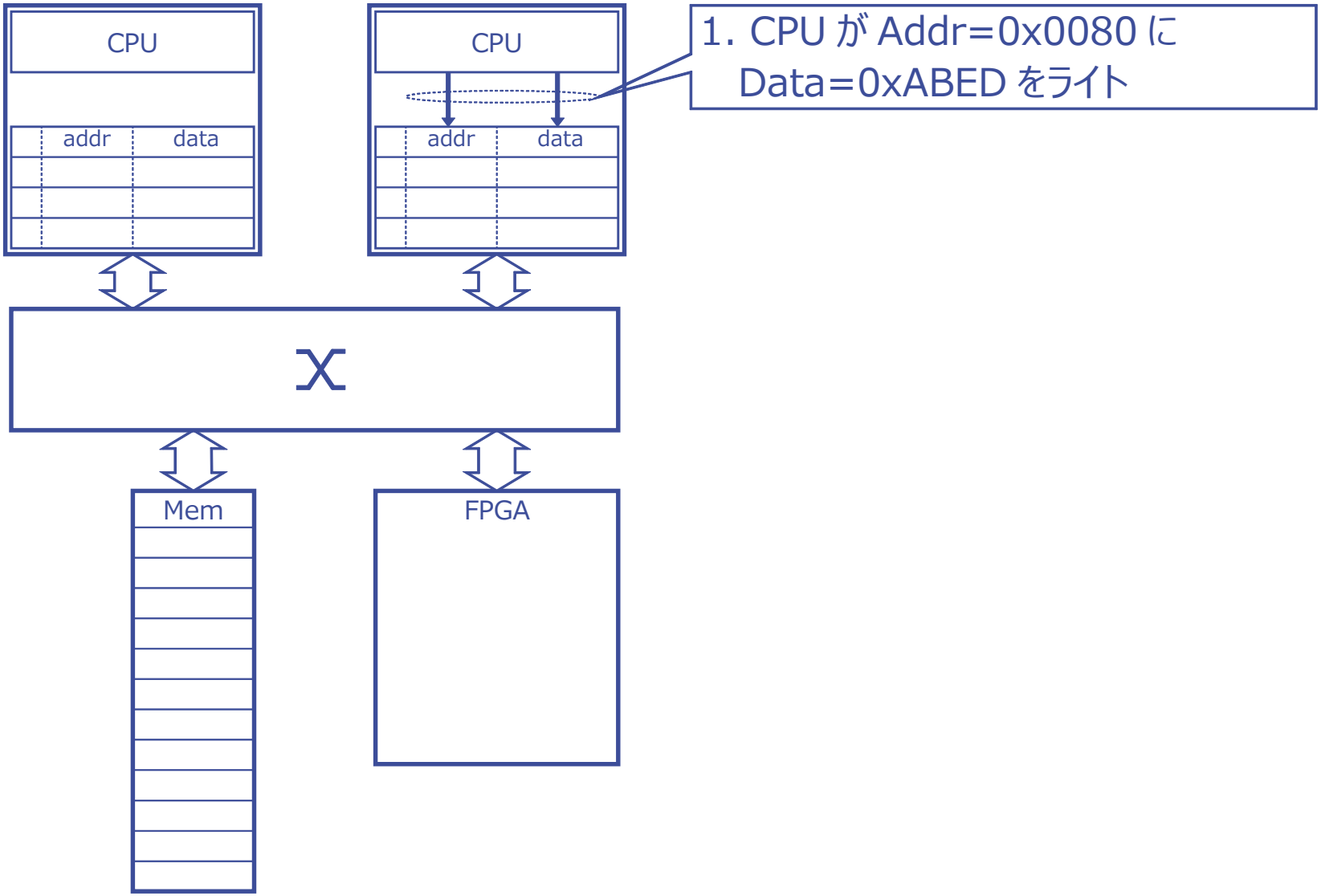
自己紹介みたいなもの

- ハンドルネーム ikwzm
- 現在隠居中
- もうすぐ 54 才(けっこう年)
- 主に論理回路設計 (回路図～VHDL)
- たまにプログラム(アセンブラ～ C /Ruby)
- 言語の設計は素人

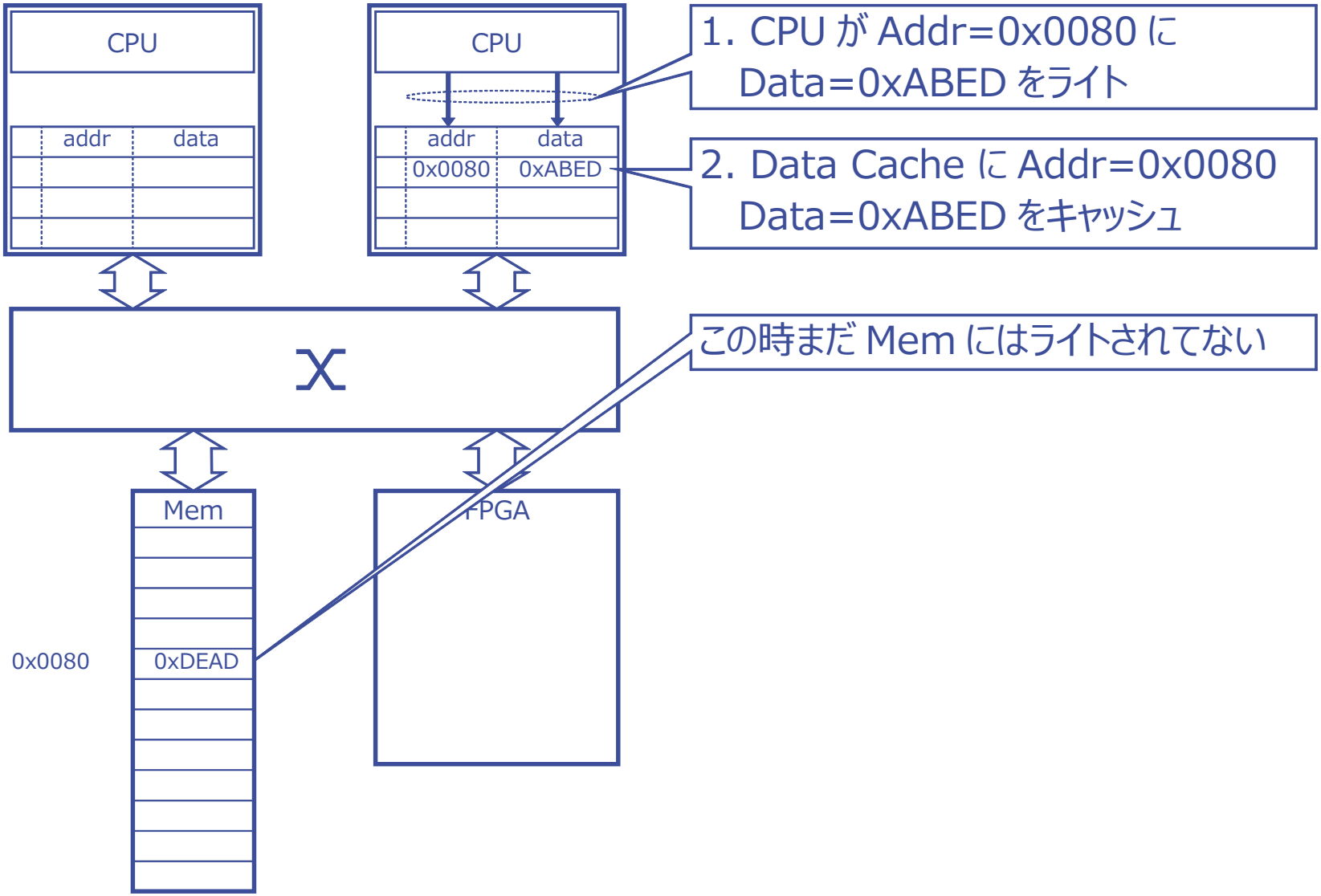
Cache Coherency トラブルの症状

- 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

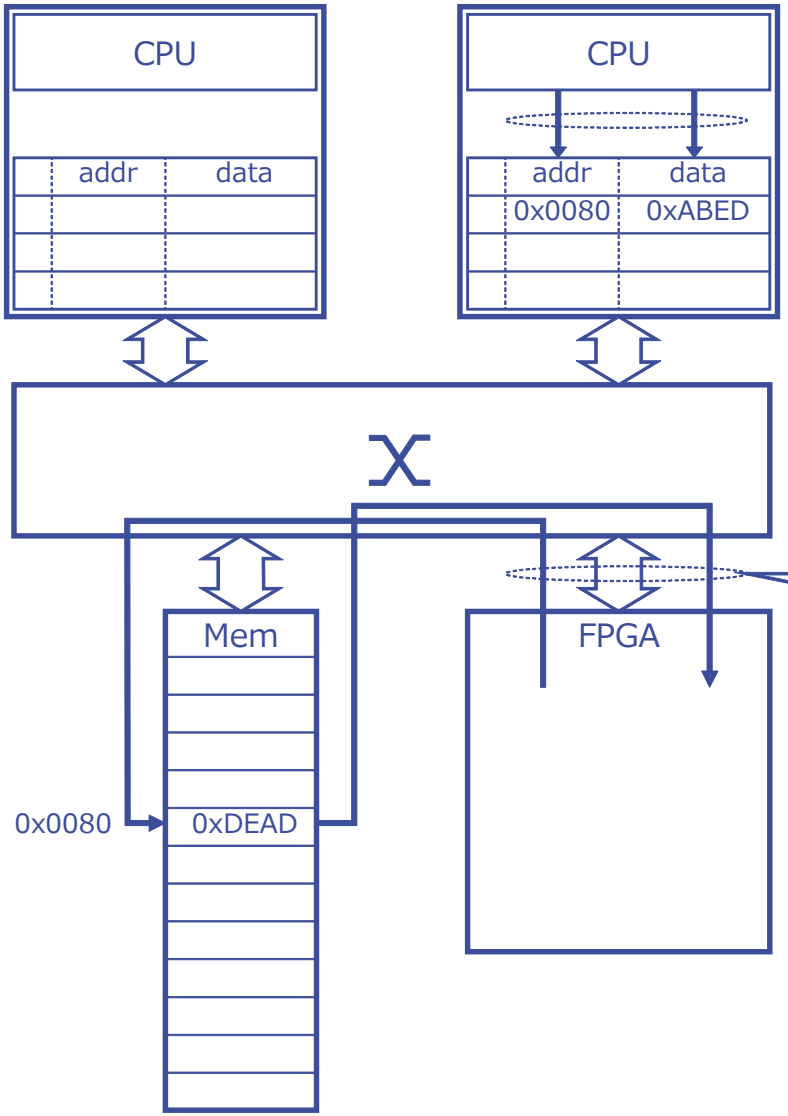
Cache Coherency でトラブルが発生するケース 1



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



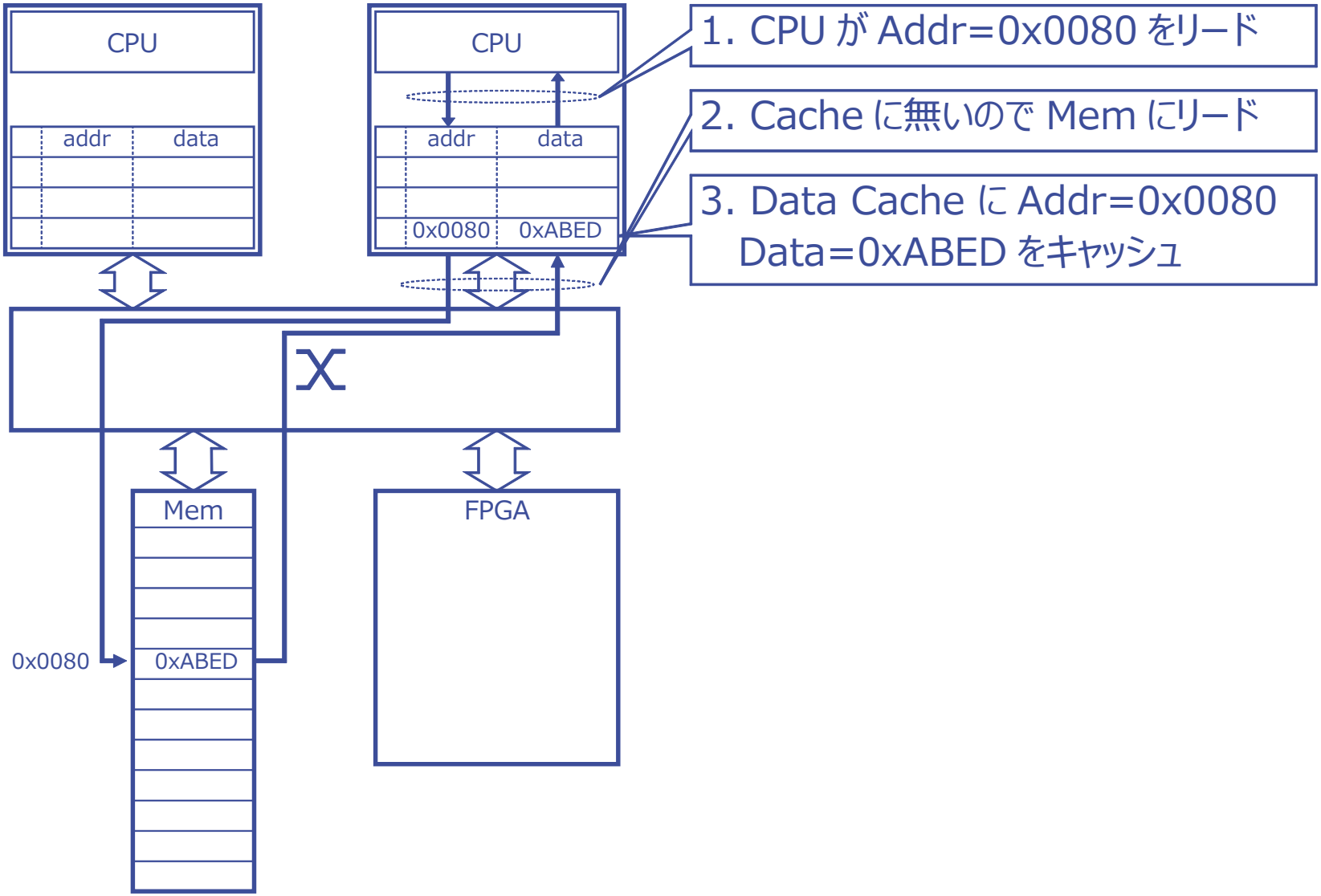
1. CPU が Addr=0x0080 に
   Data=0xABED をライト

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

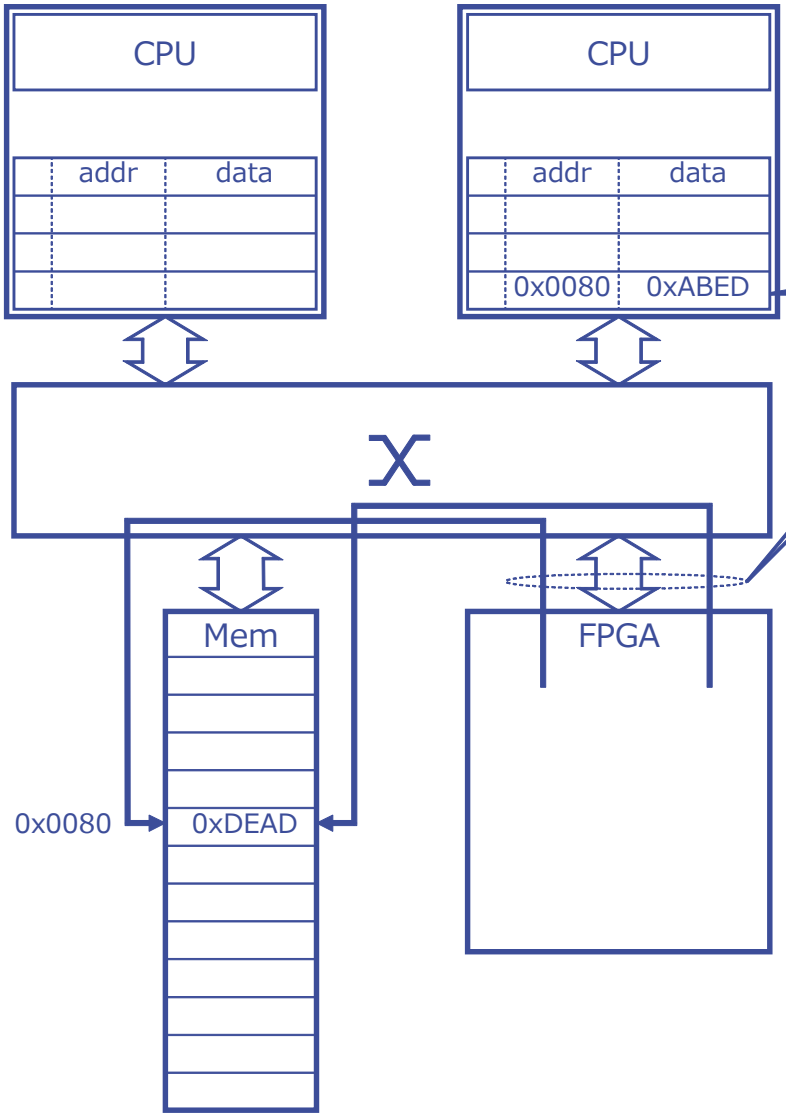
3. FPGA が Addr=0x0080 をリード

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

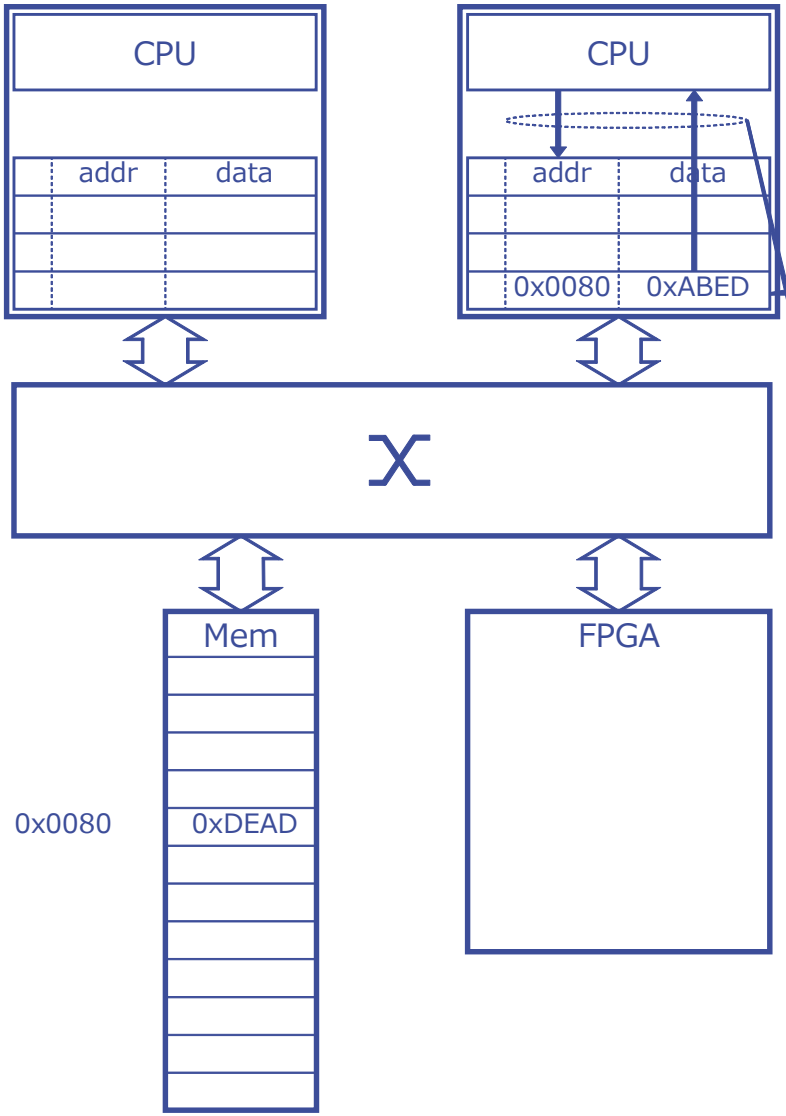


Cache Coherency でトラブルが発生するケース 2



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



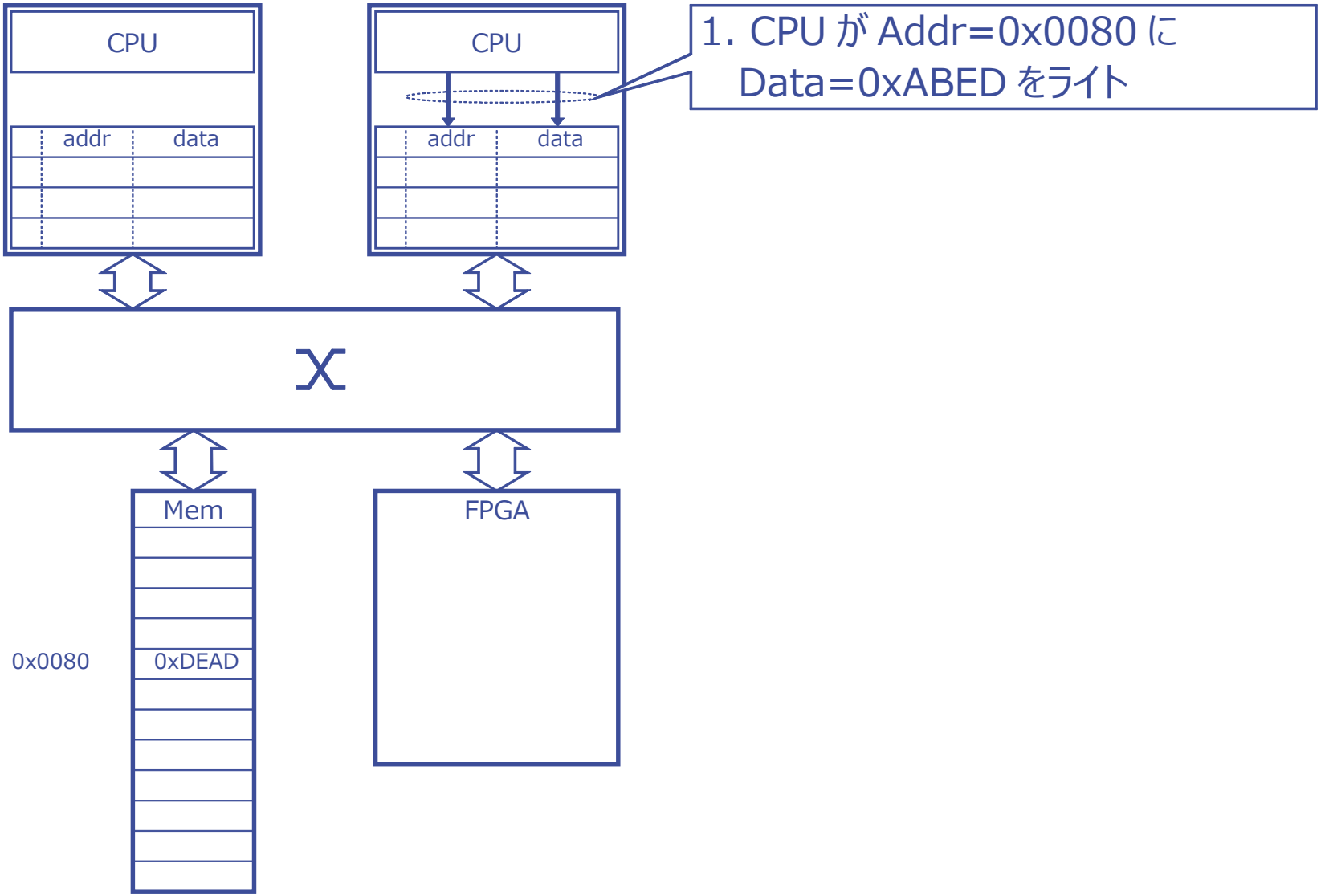
1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. CPU が Addr=0x0080 をリード

Data Cache に Addr=0x0080 のデータ(0xABED)が残っているので、CPU は data=0xABED と誤った値をリード（Mem にはリードしない)

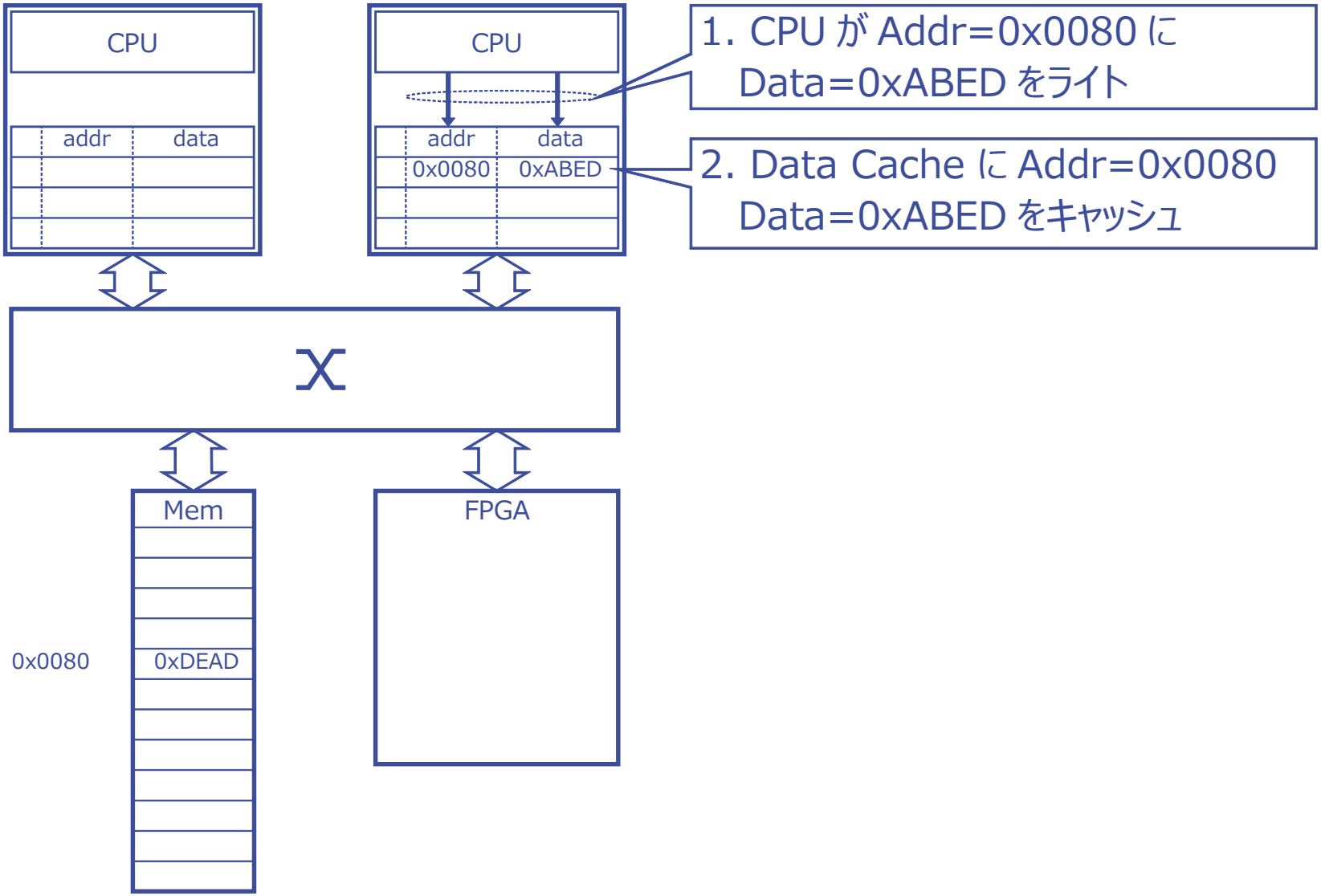
Cache Coherency トラブルの解決方法

- 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

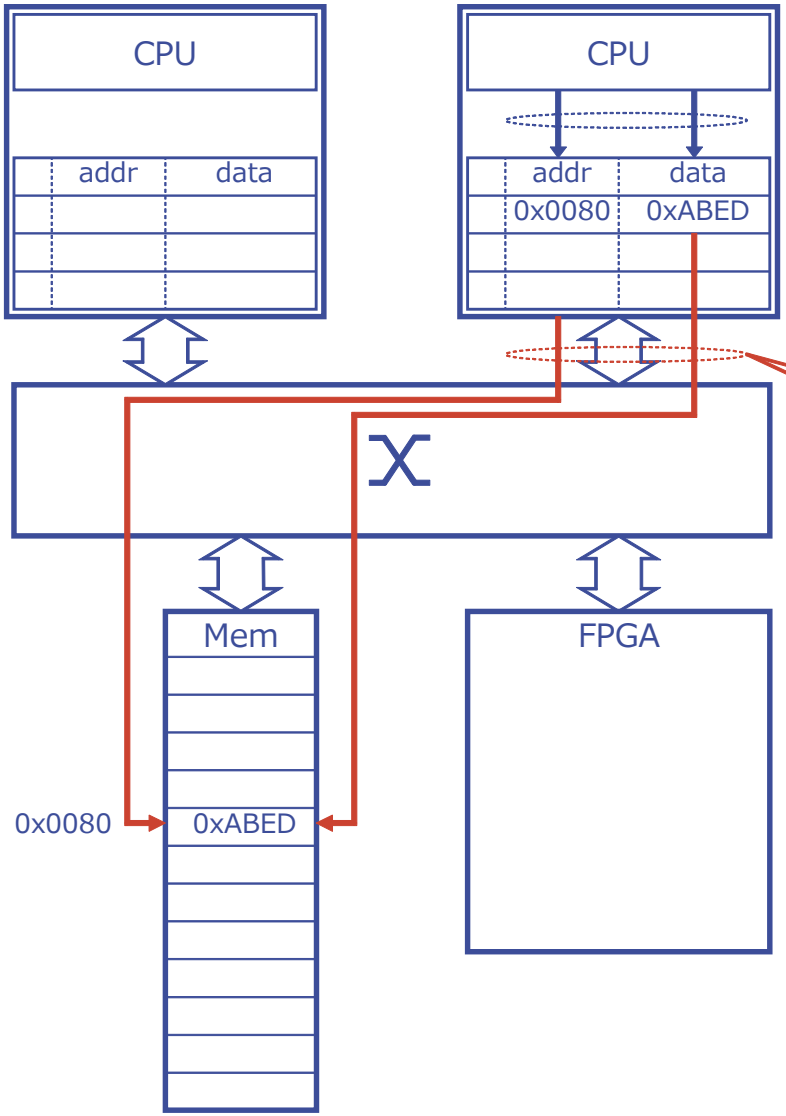
ソフトウェアで Cache を制御してトラブル回避　ケース 1



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

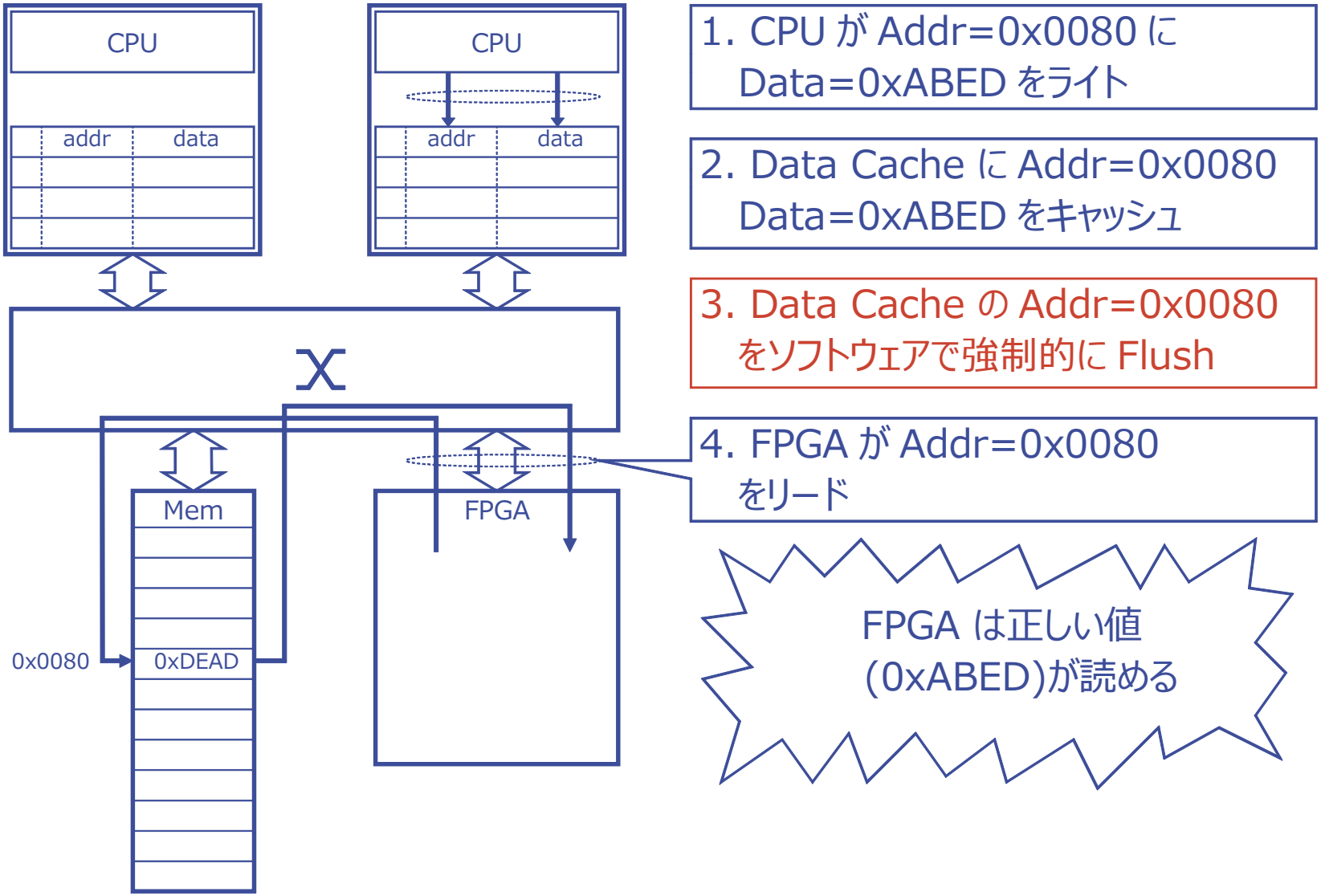


1. CPU が Addr=0x0080 に Data=0xABED をライト

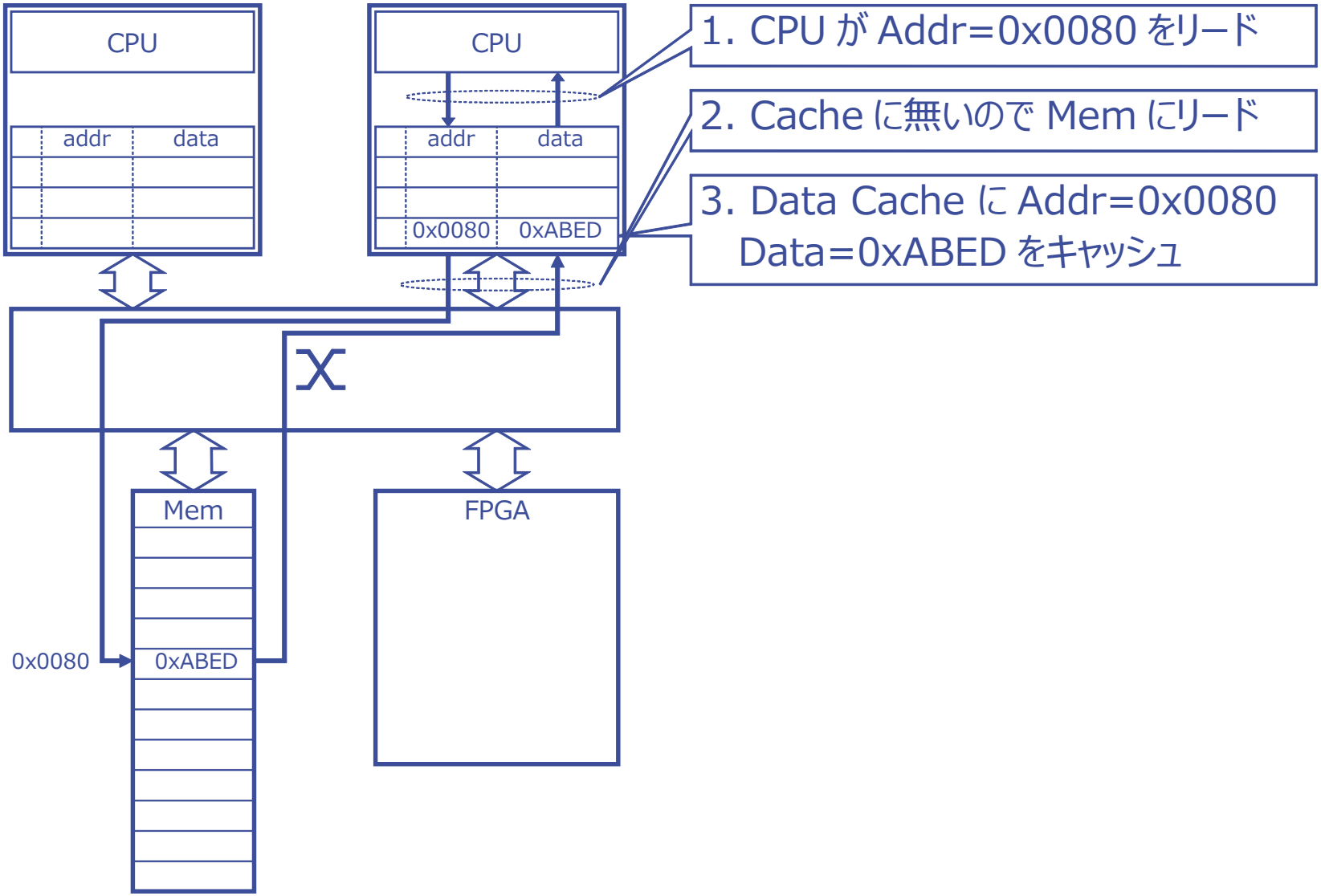
2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

3. Data Cache の Addr=0x0080 をソフトウェアで強制的に Flush

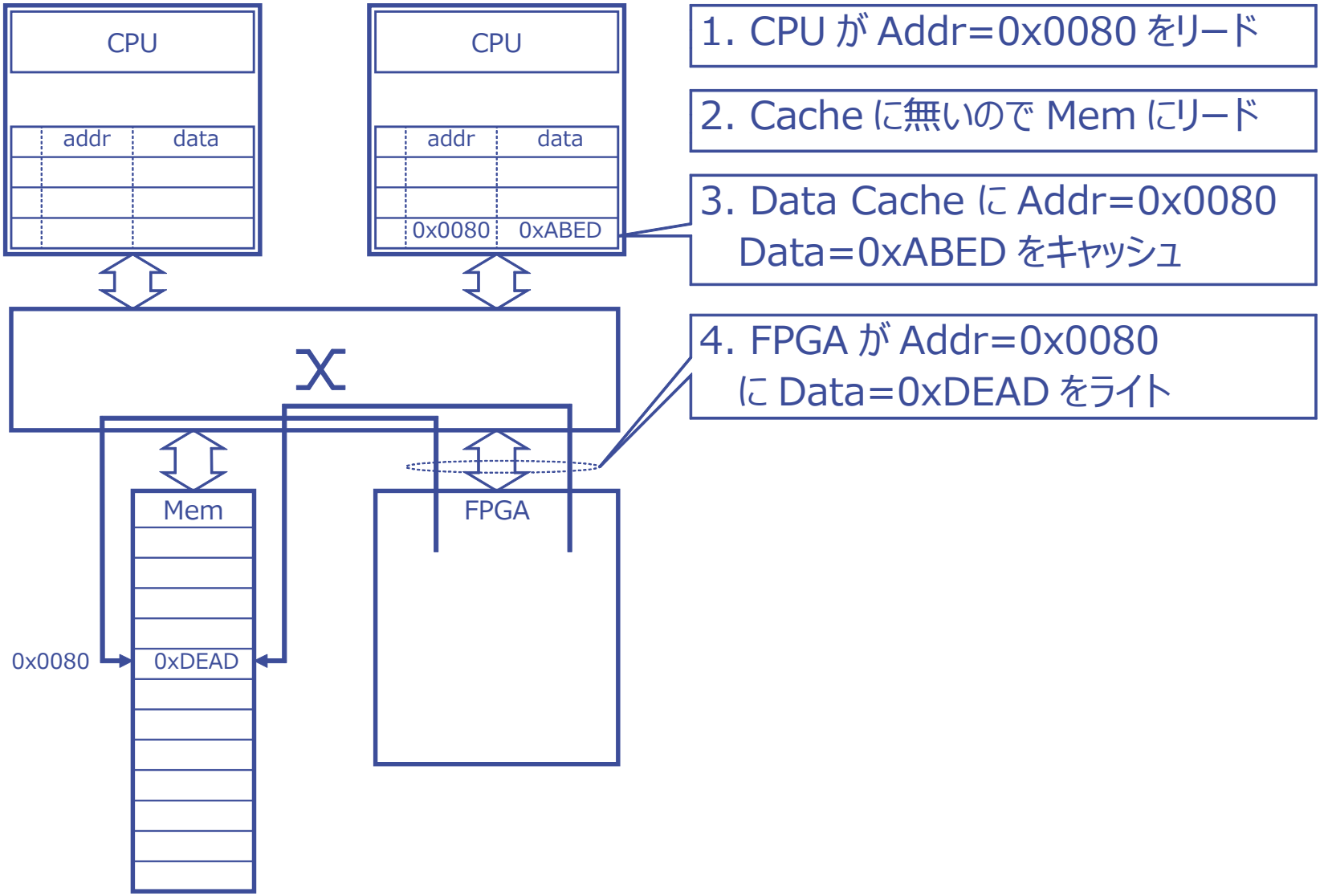
ソフトウェアで Cache を制御してトラブル回避　ケース 1



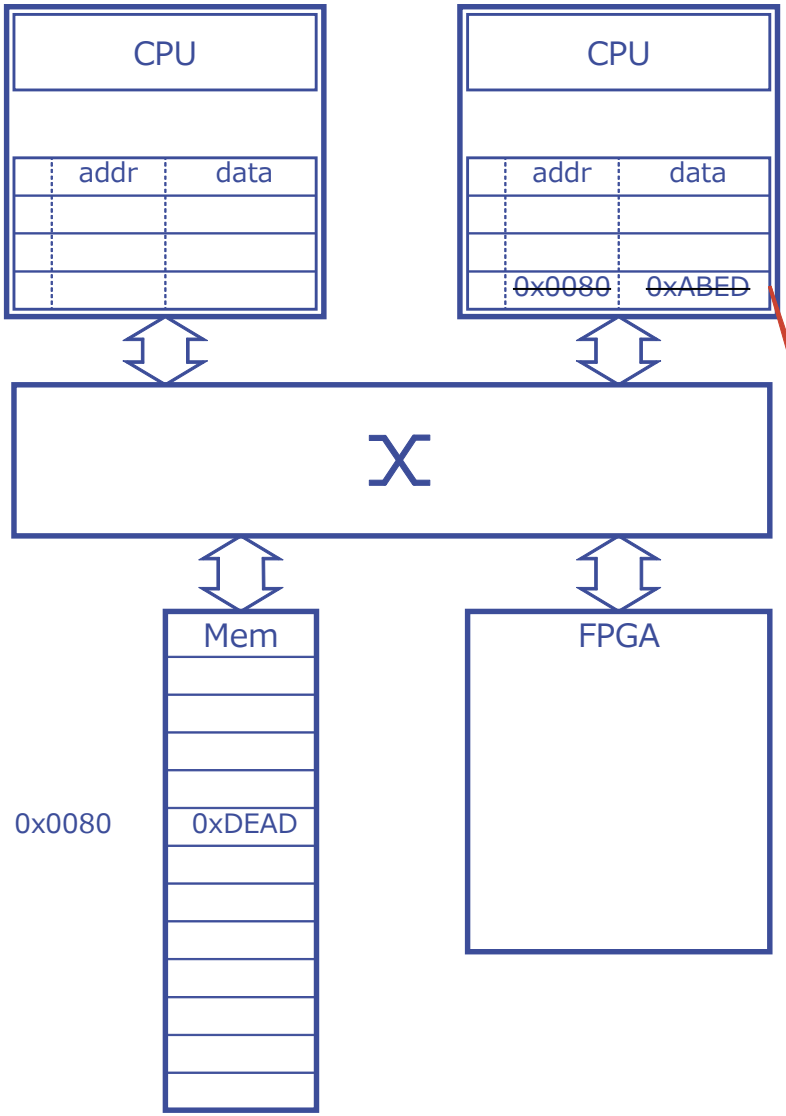
ソフトウェアで Cache を制御してトラブル回避　ケース２



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

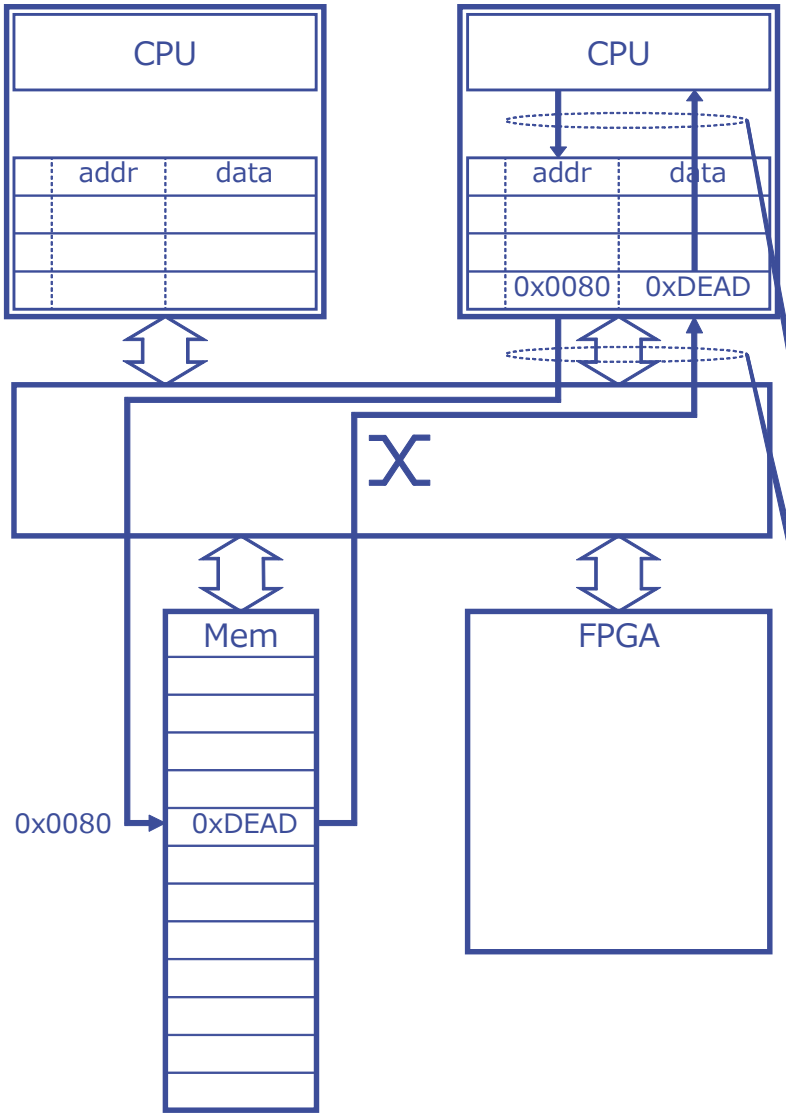
2. Cache に無いので Mem にリード

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 をソフトウェアで 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 をソフトウェアで 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

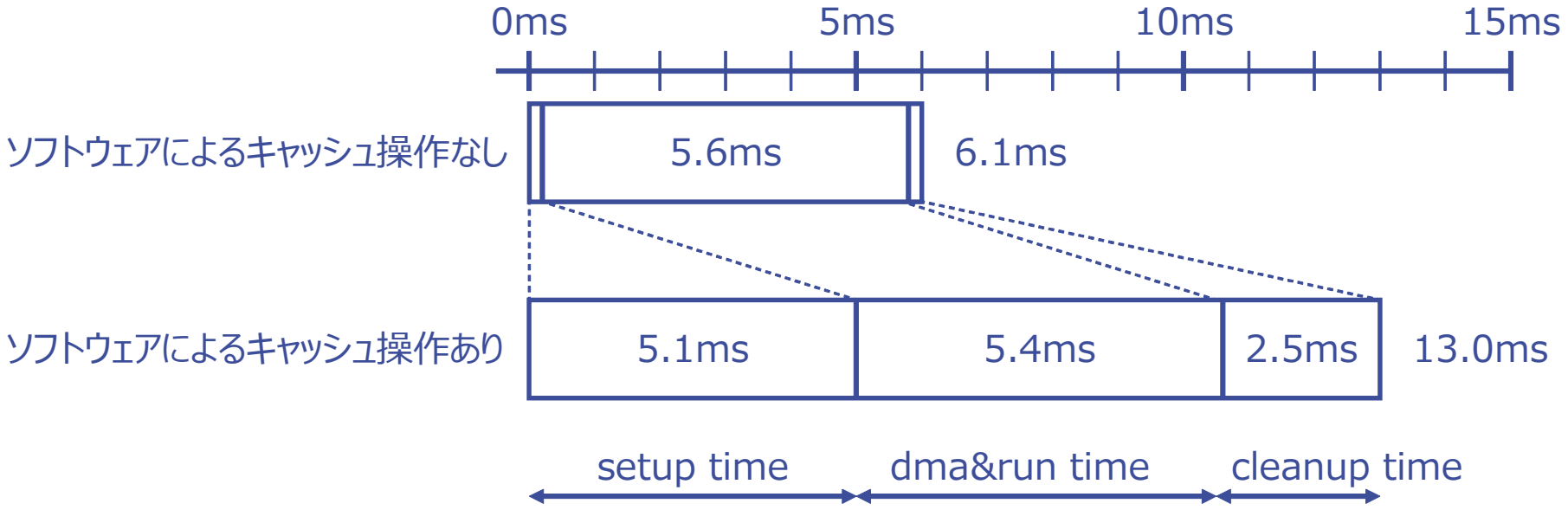
ソフトウェアで Cache を制御する方法の問題点

- 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)

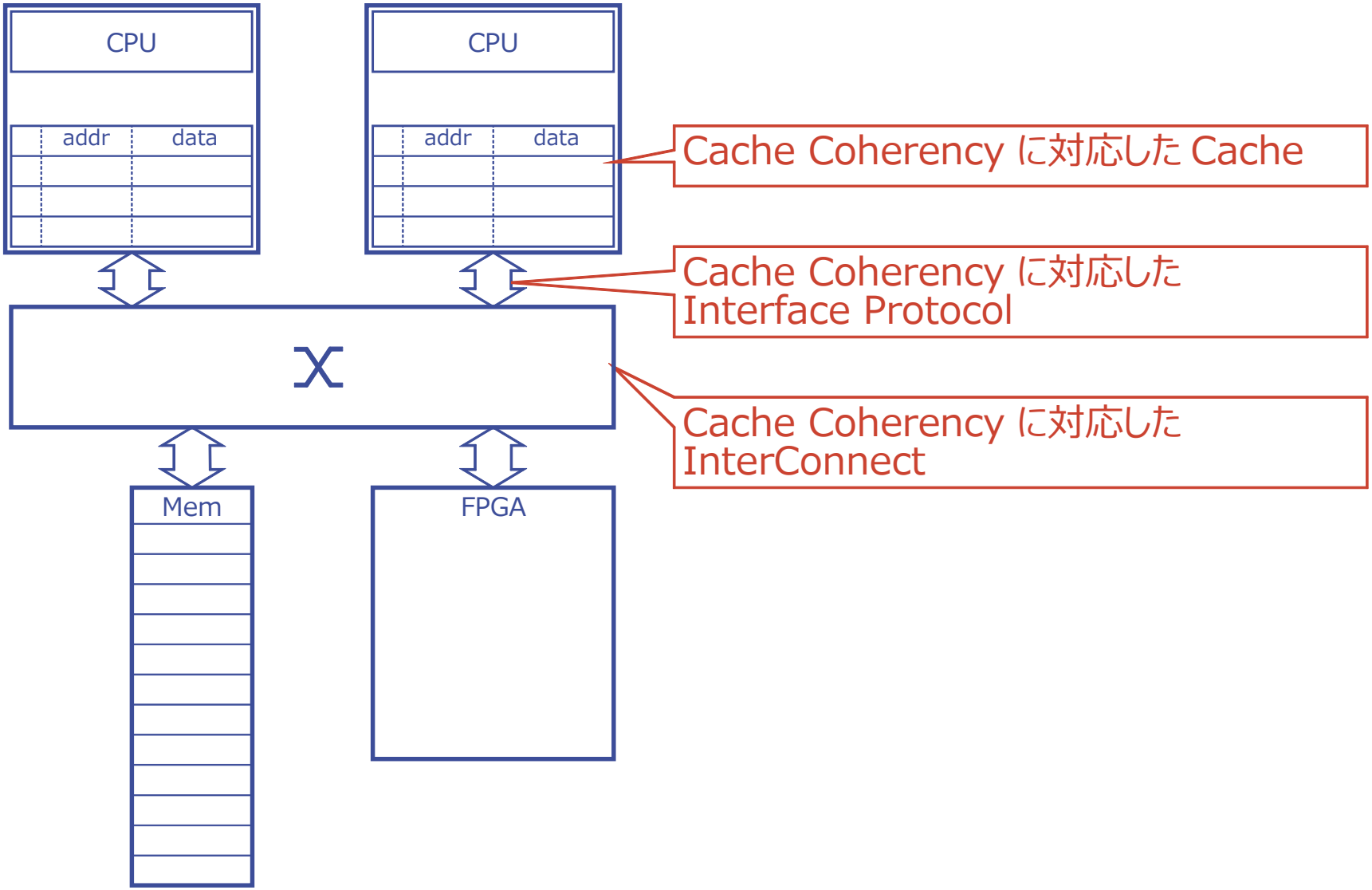
ソフトウェアによるキャッシュ操作時間の例 - Zynq の場合

- https://github.com/ikwzm/FPGA-SoC-Linux-Example-1-ZYBO-Z7
- 転送サイズ：1MiB(1048576Byte)
- DMA で Mem→FPGA と FPGA->Mem を同時に転送

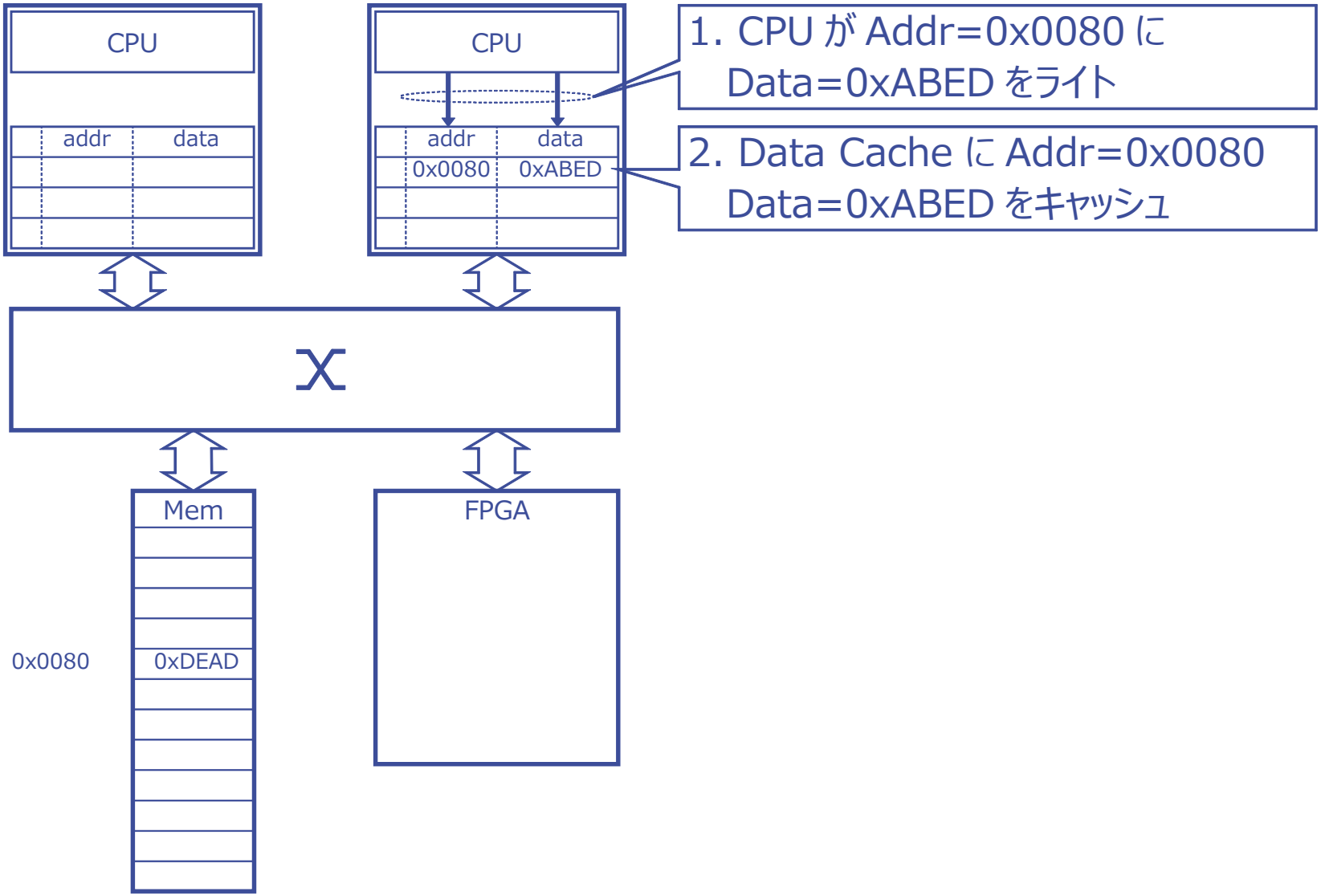


- 1MiB キャッシュフラッシュに 4.9msec
- 1MiB キャッシュ無効化に 2.3msec

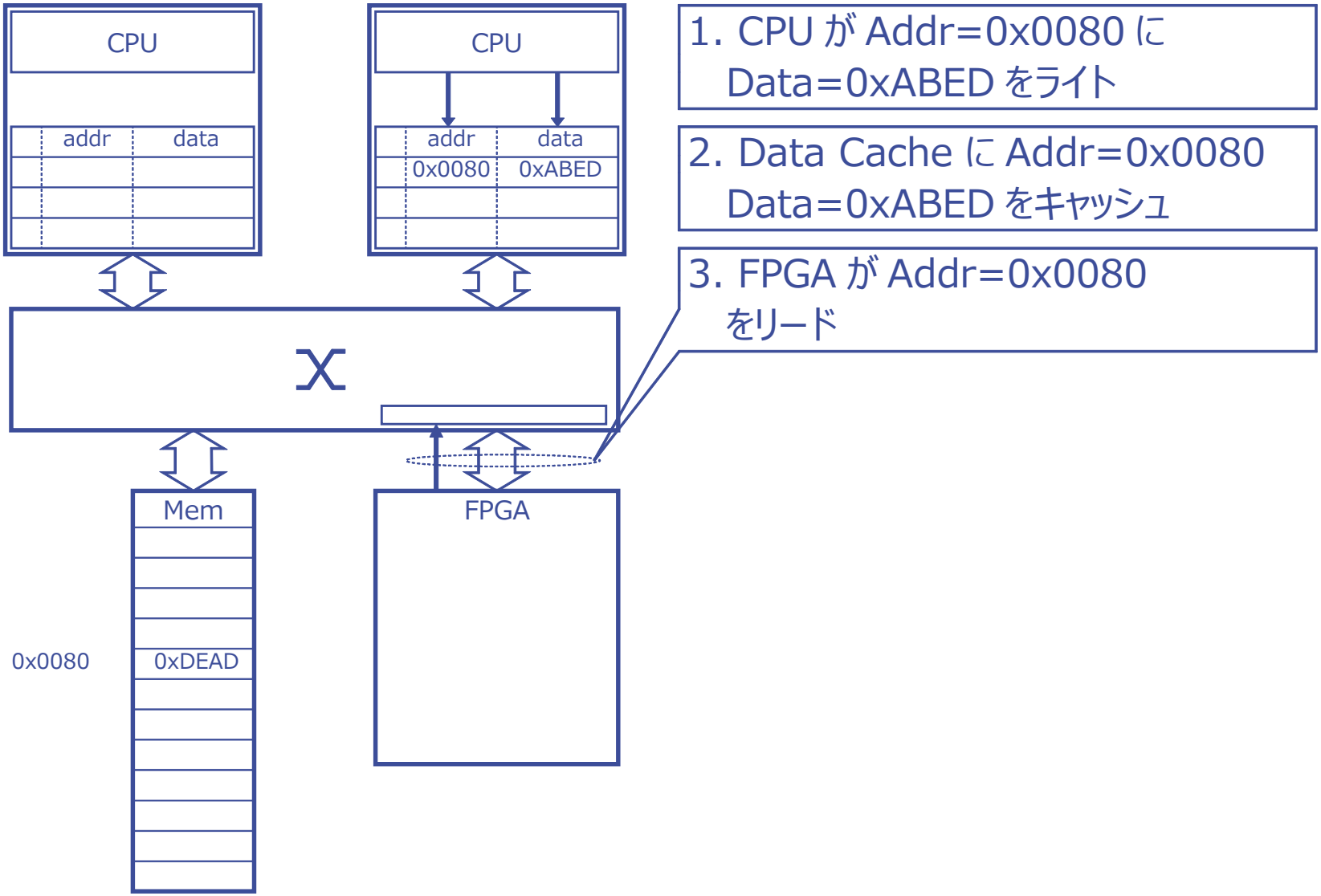
ハードウェアで Cache Coherency - 用意するもの



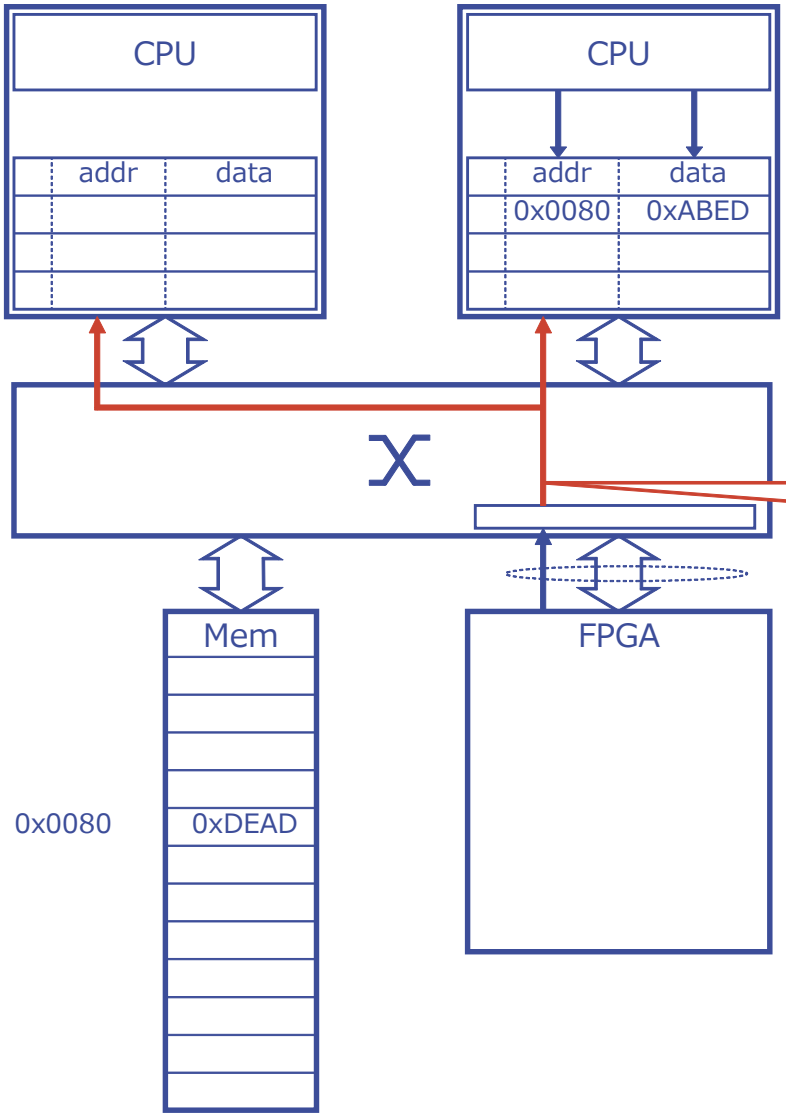
ハードウェアで Cache Coherency ケース 1



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



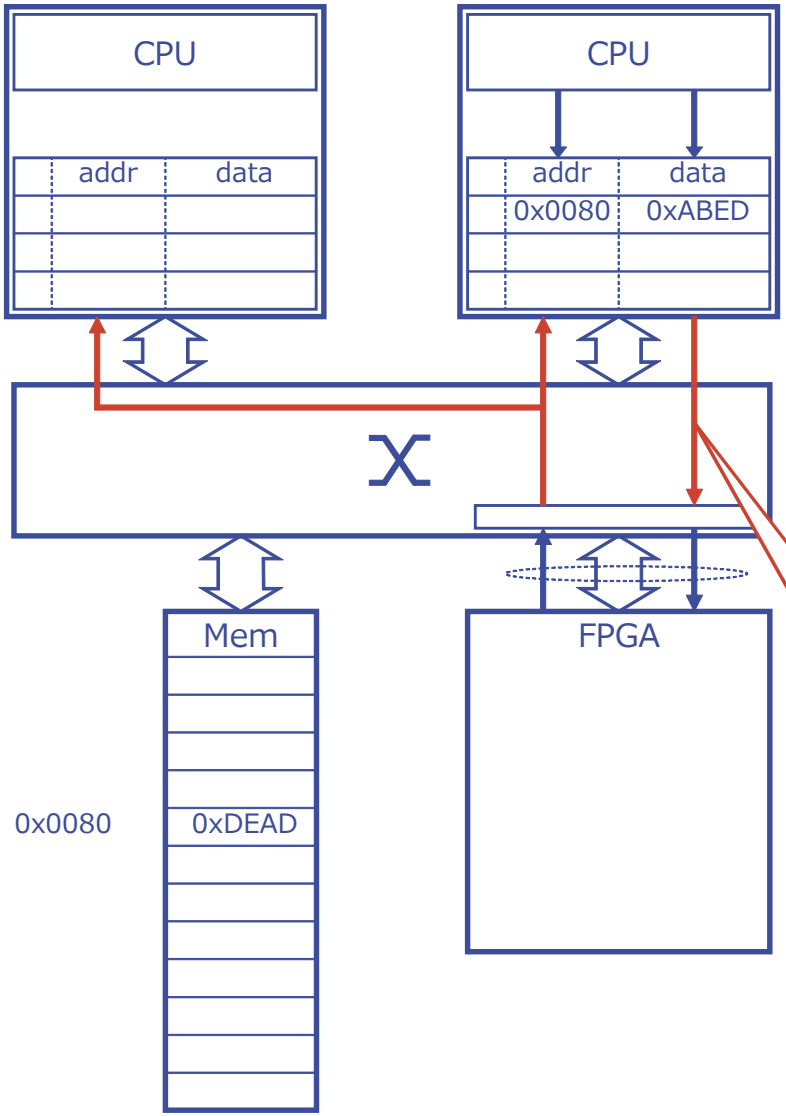
1. CPU が Addr=0x0080 に Data=0xABED をライト

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

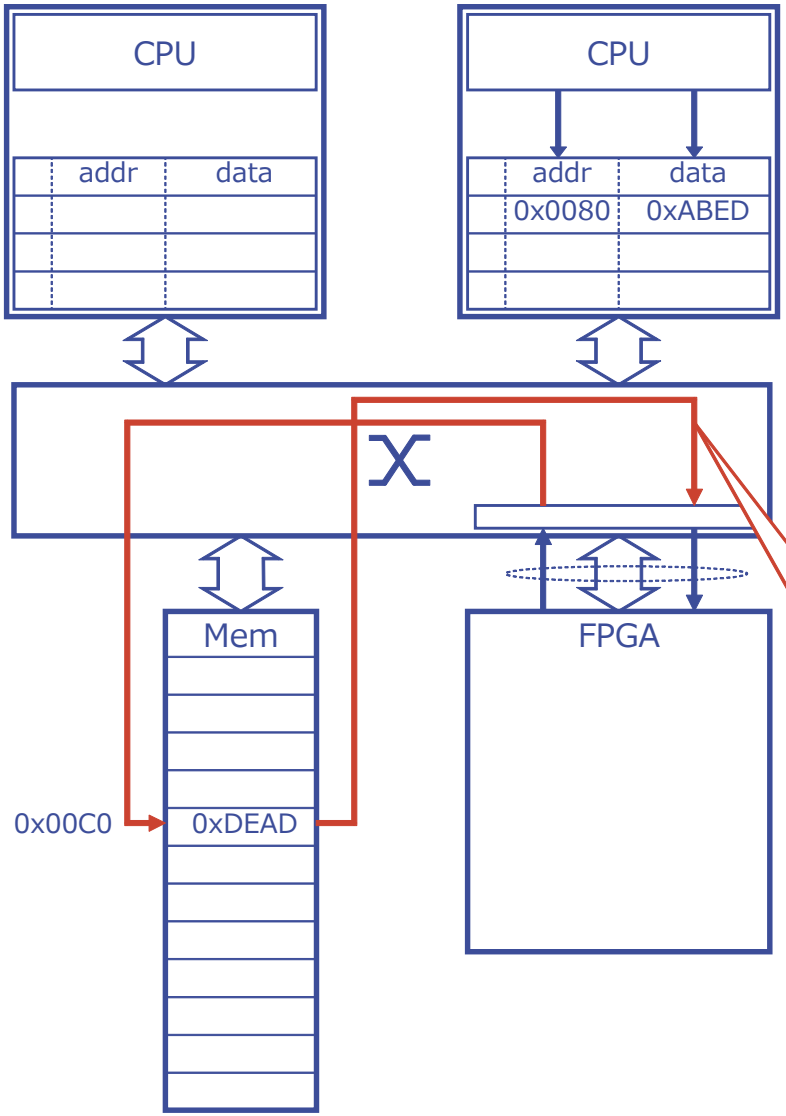
2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

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

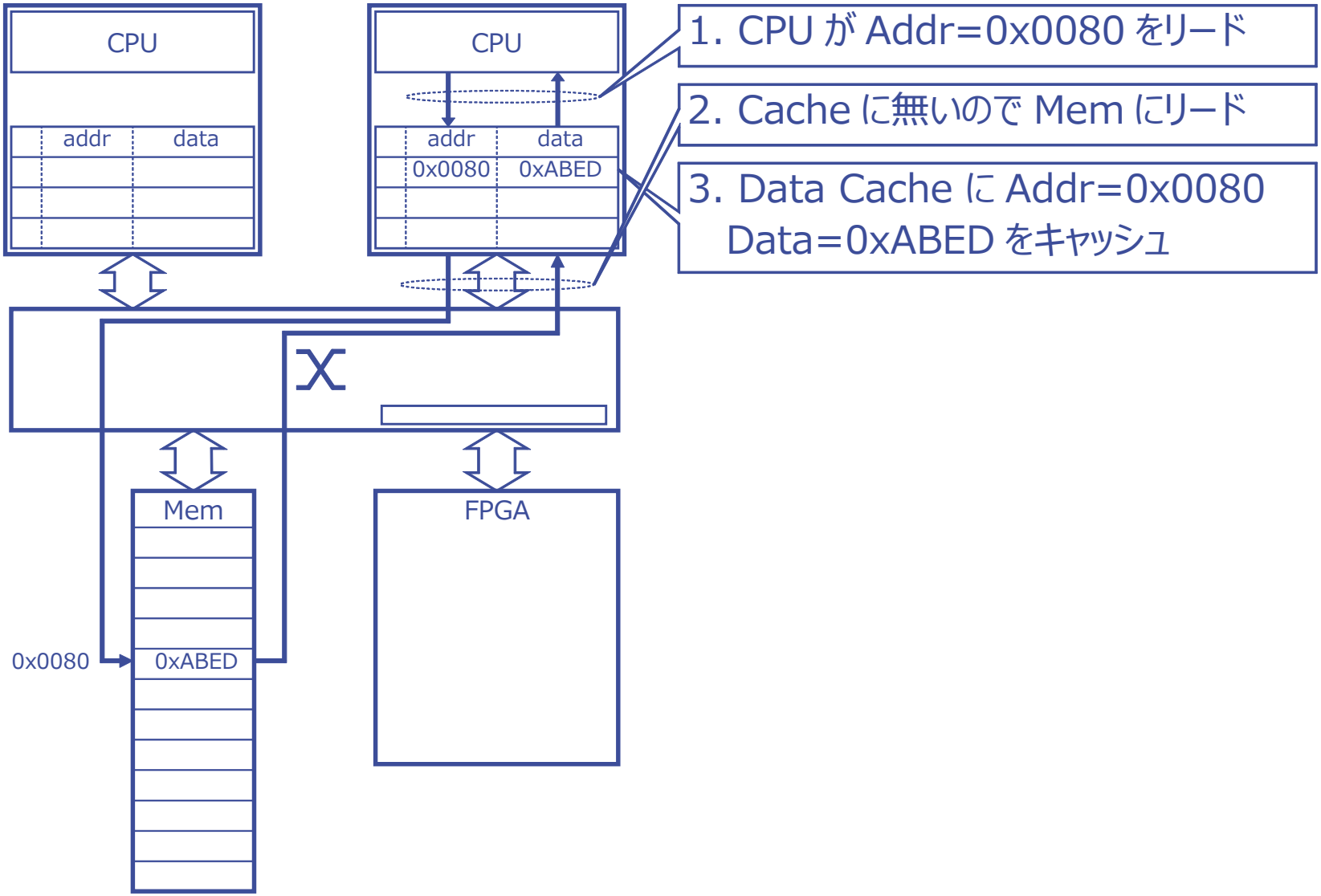
2. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ

3. FPGA が Addr=0x00C0 をリード

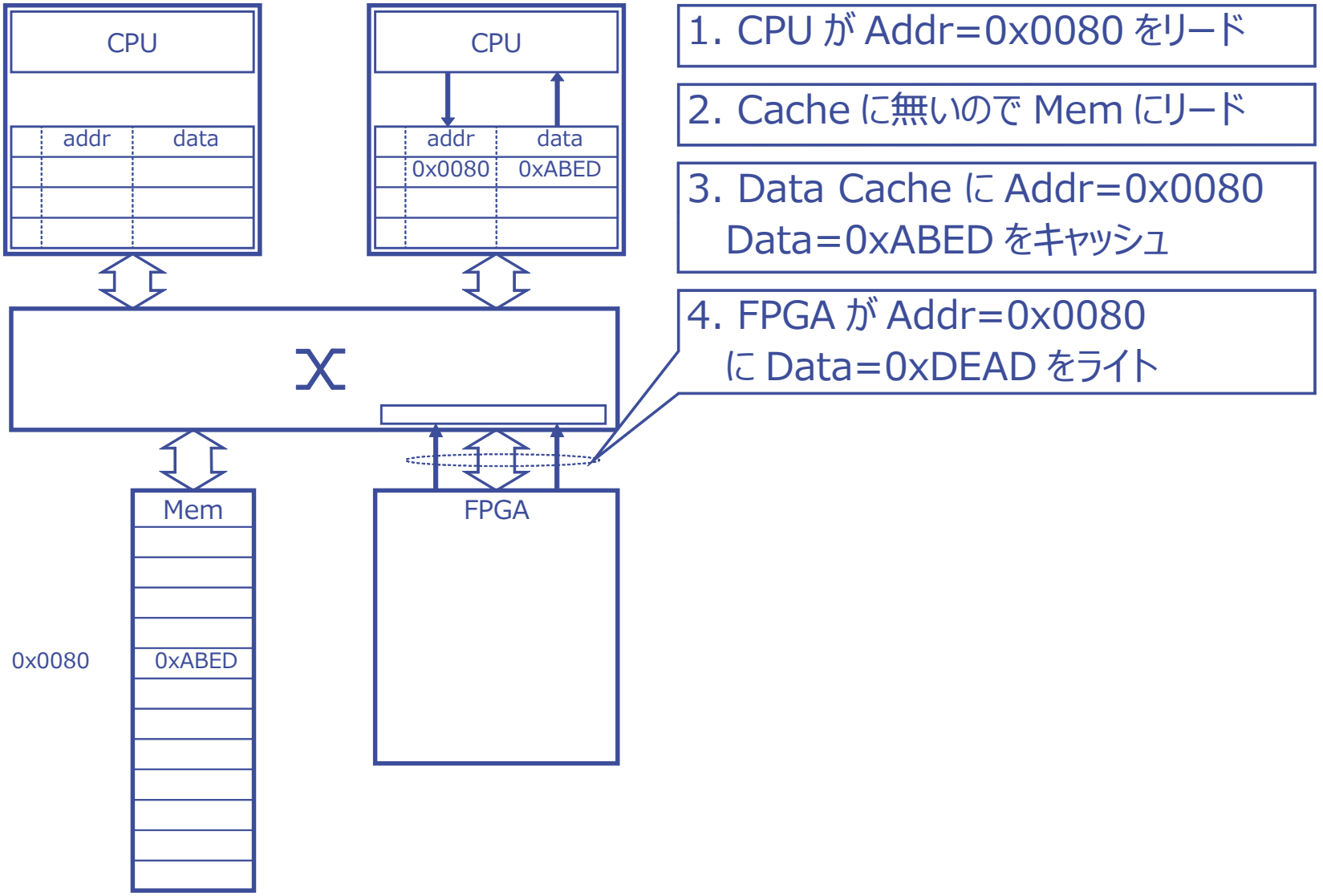
4.InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)

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

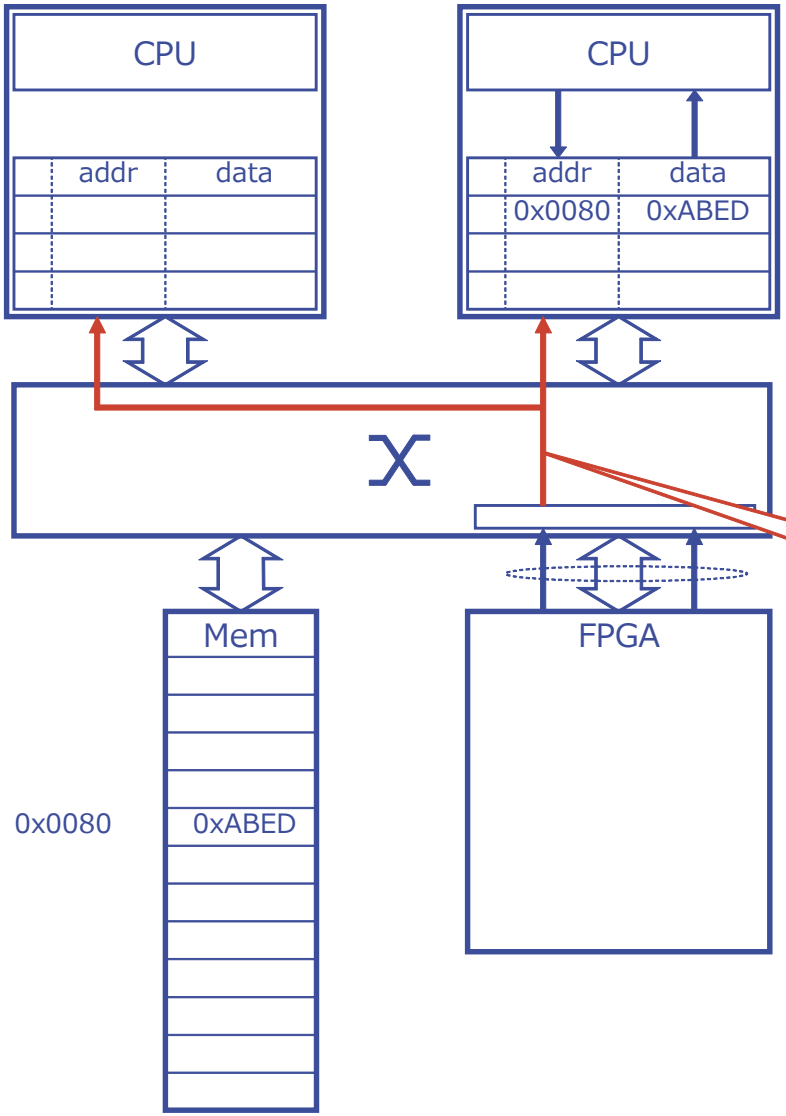
ハードウェアで Cache Coherency ケース 2



ハードウェアで Cache Coherency ケース 2

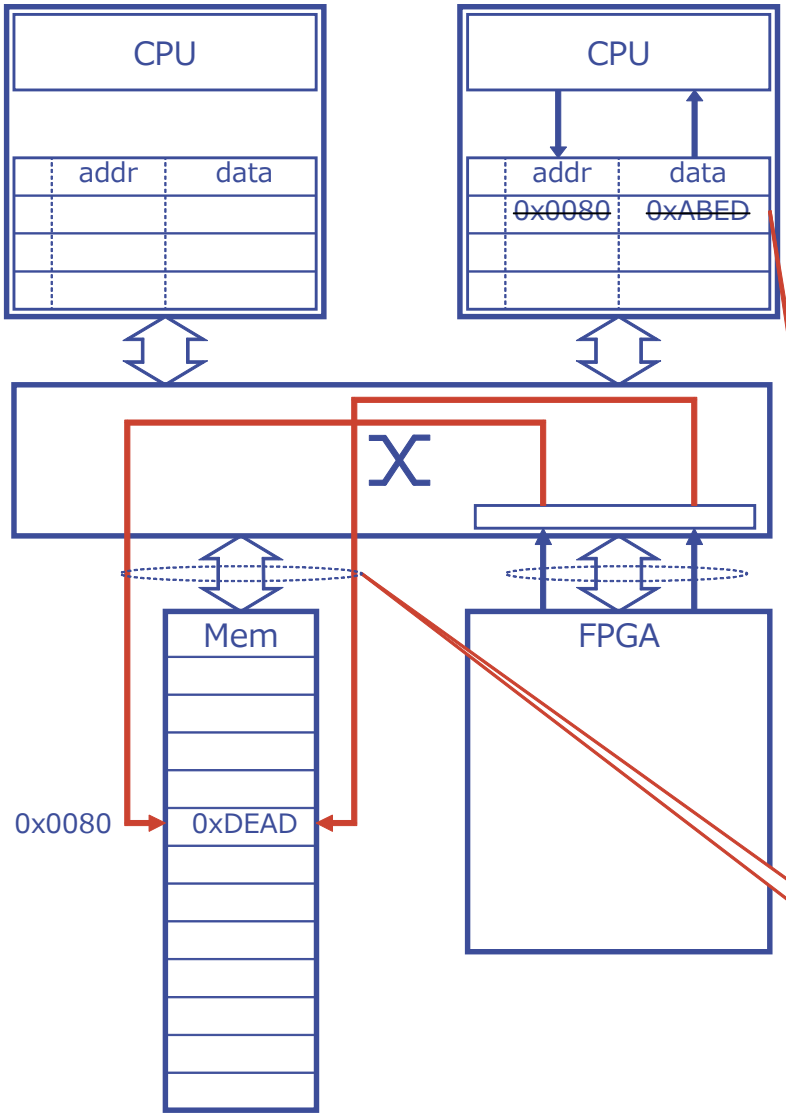


ハードウェアで Cache Coherency ケース 2



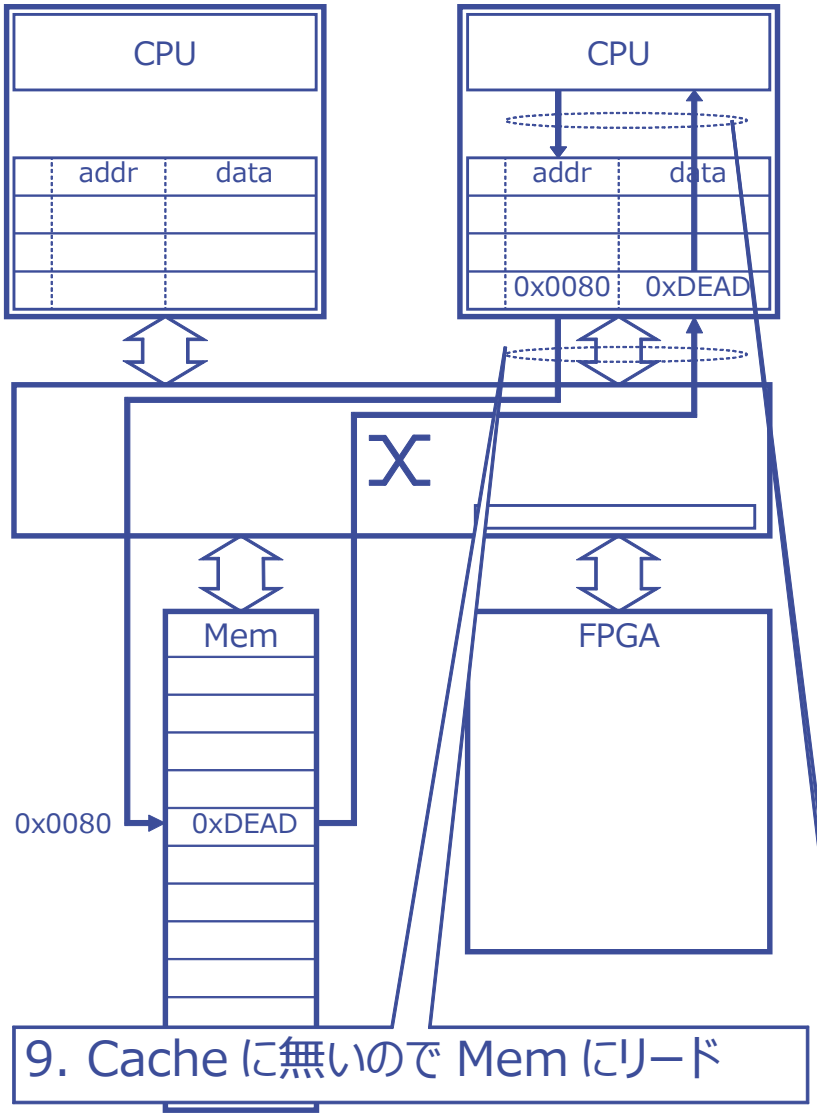
1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)

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）



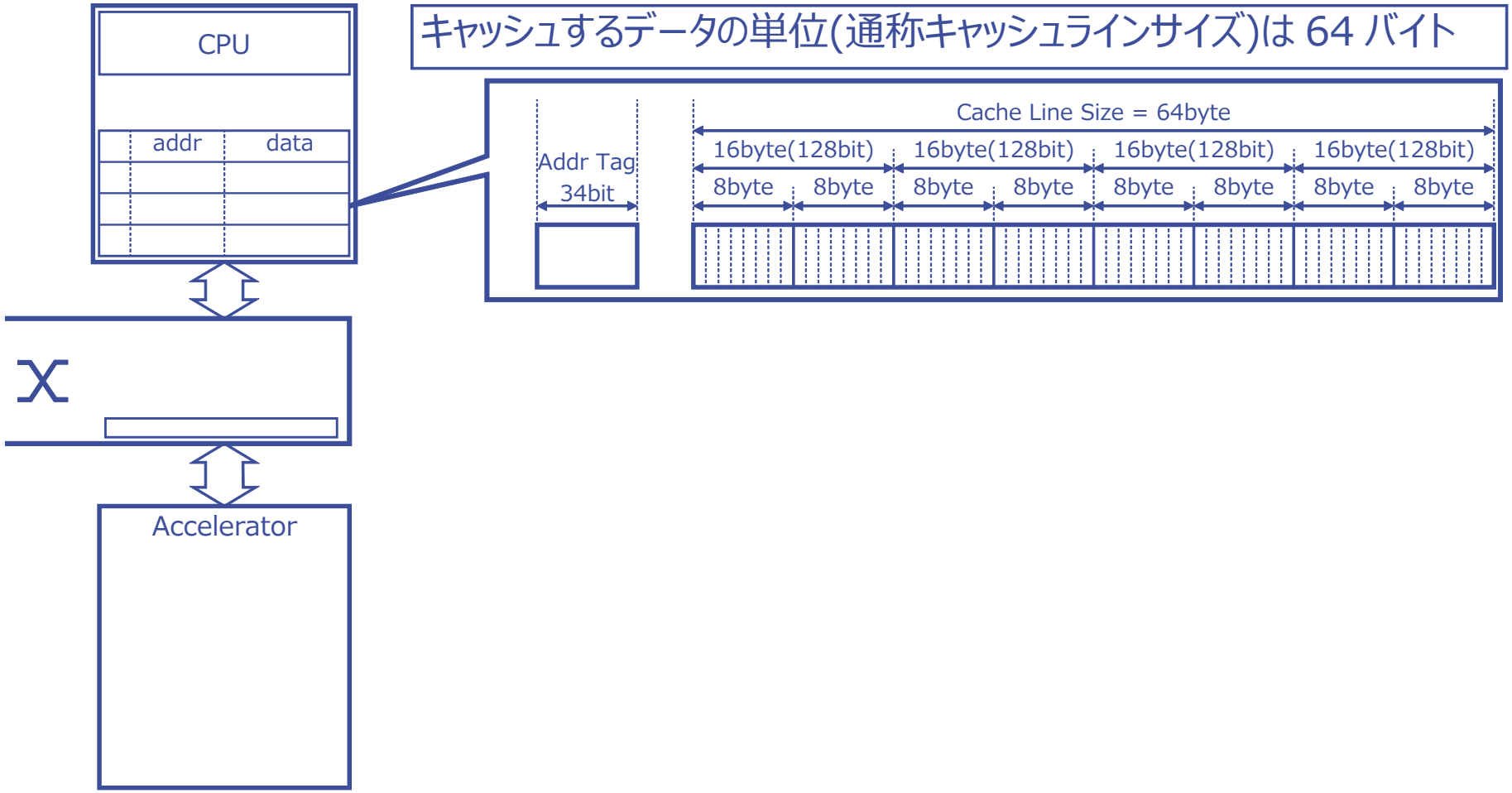
1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)
6. 該当する Addr(0x0080)のデータをキャッシュしているマスターはキャッシュを無効化(Invalidiate)
7. Mem の Addr=0x0080 に Data=0xDEAD をライト

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）

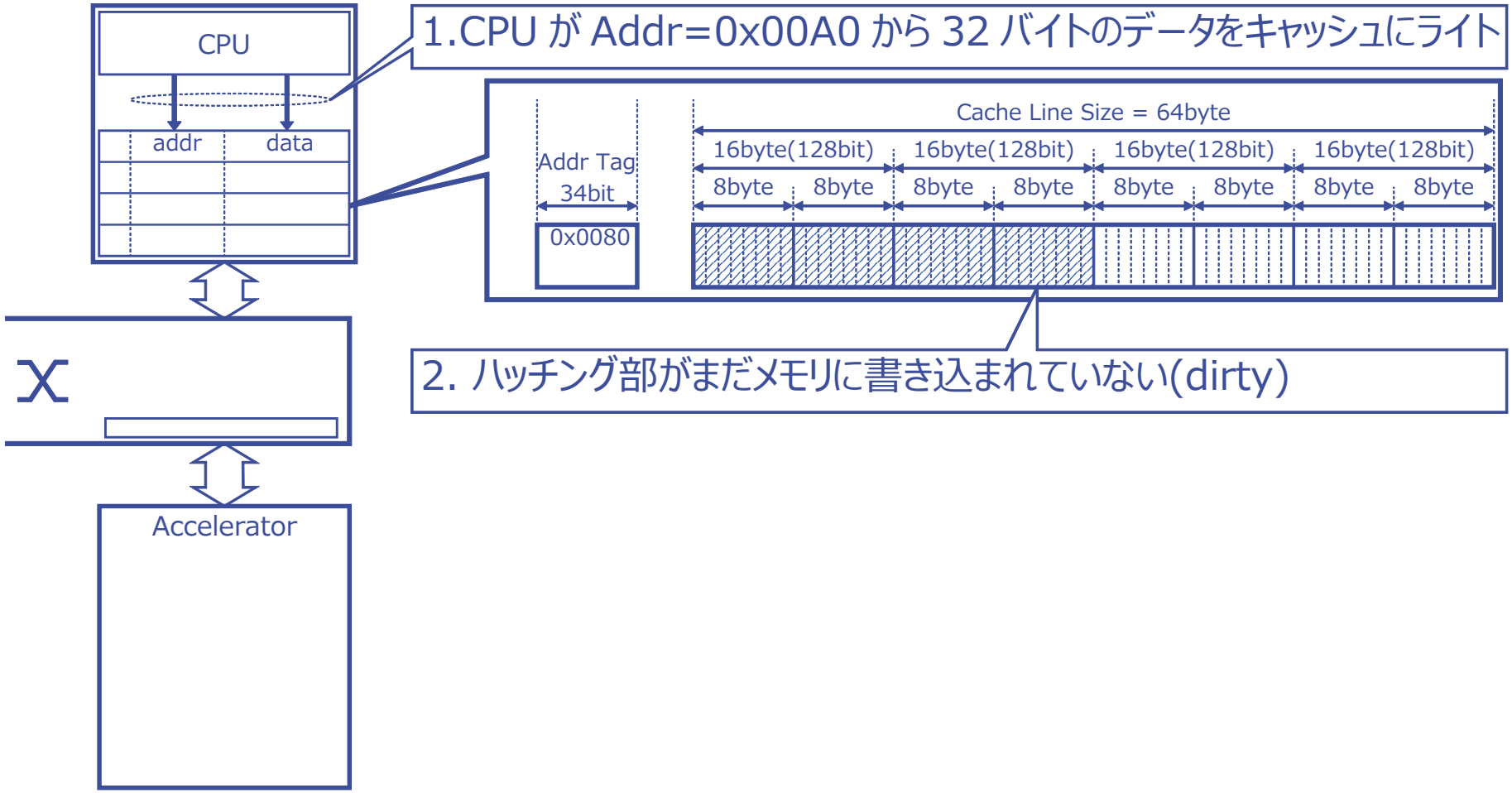


1. CPU が Addr=0x0080 をリード
2. Cache に無いので Mem にリード
3. Data Cache に Addr=0x0080 Data=0xABED をキャッシュ
4. FPGA が Addr=0x0080 に Data=0xDEAD をライト
5. InterConnect 内でキャッシュを持つマスターに Addr を通知(snoop)
6. 該当する Addr(0x0080)のデータをキャッシュしているマスターはキャッシュを無効化(Invalidiate)
7. Mem の Addr=0x0080 に Data=0xDEAD をライト
8. CPU が Addr=0x0080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

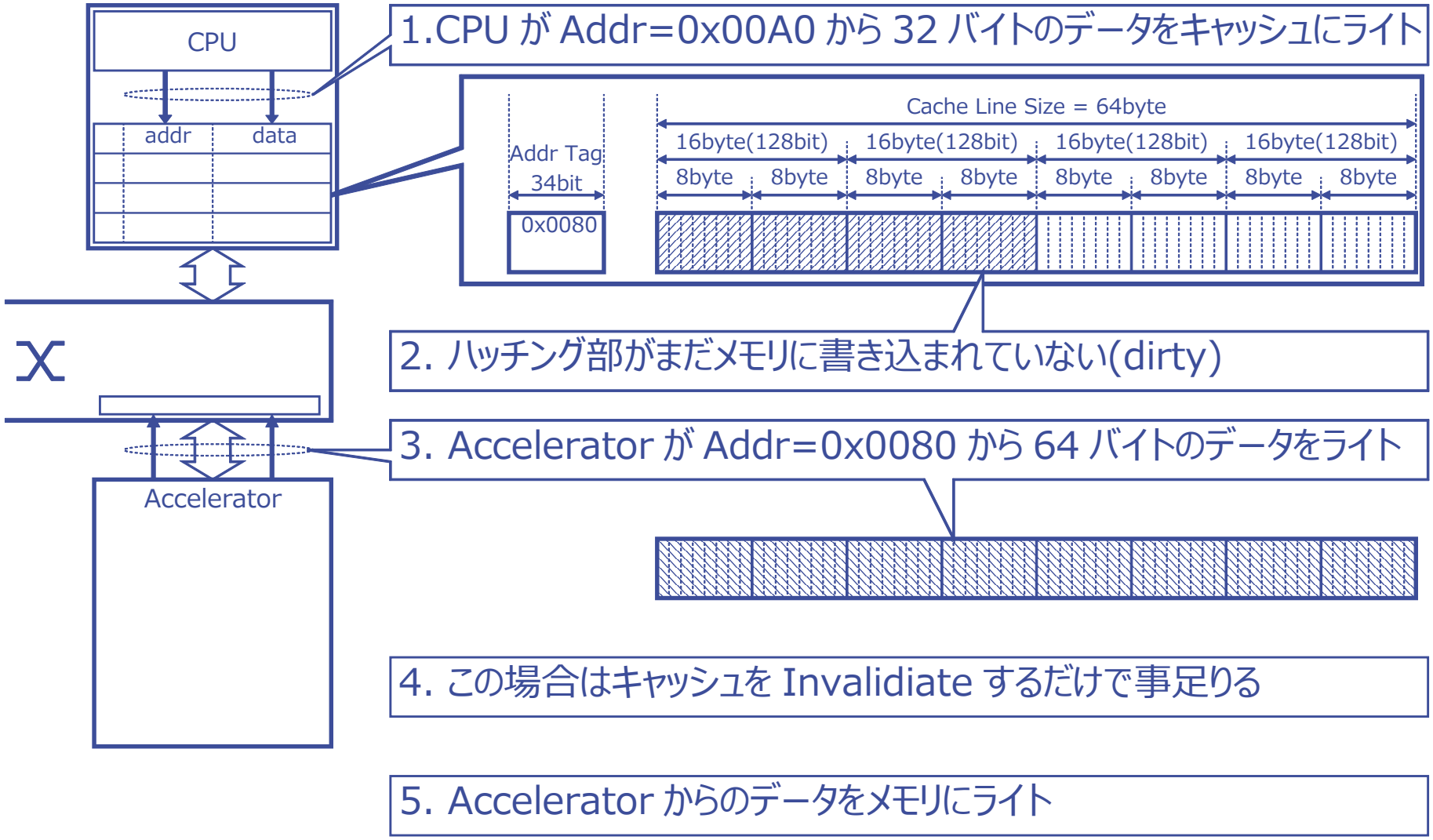
補足　キャッシュラインのアライメント問題



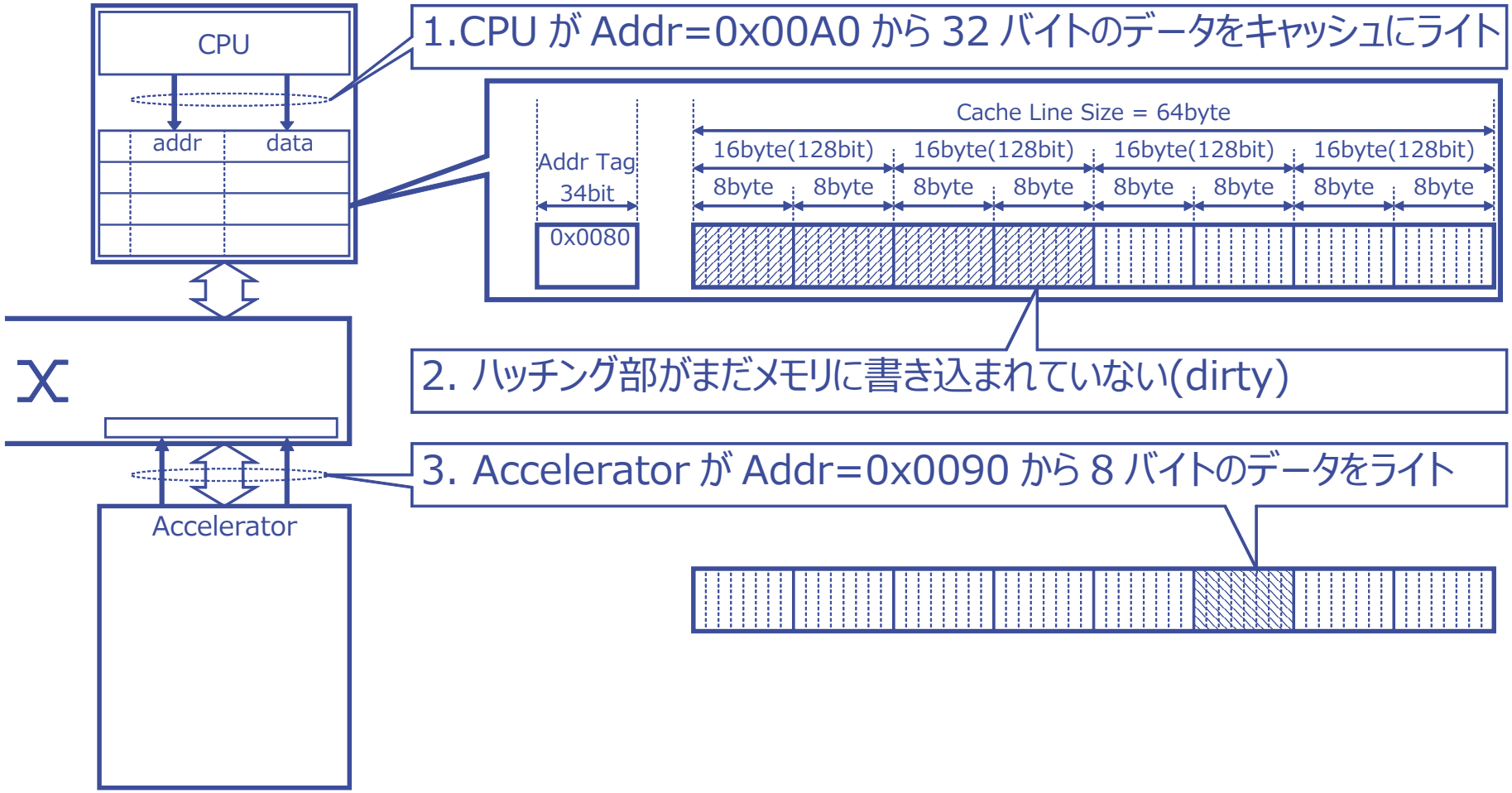
補足　キャッシュラインのアライメント問題



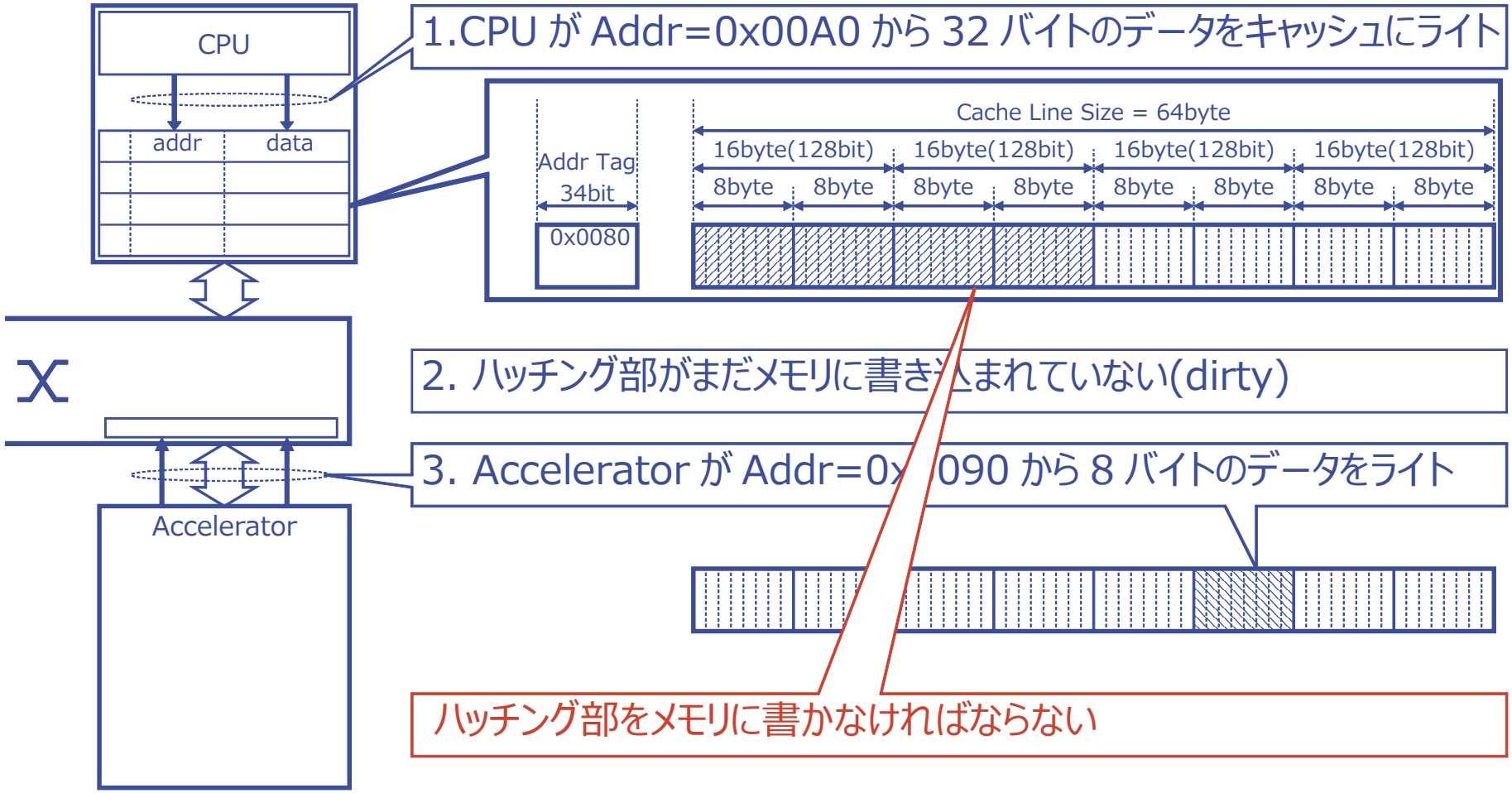
補足　キャッシュラインのアライメント問題



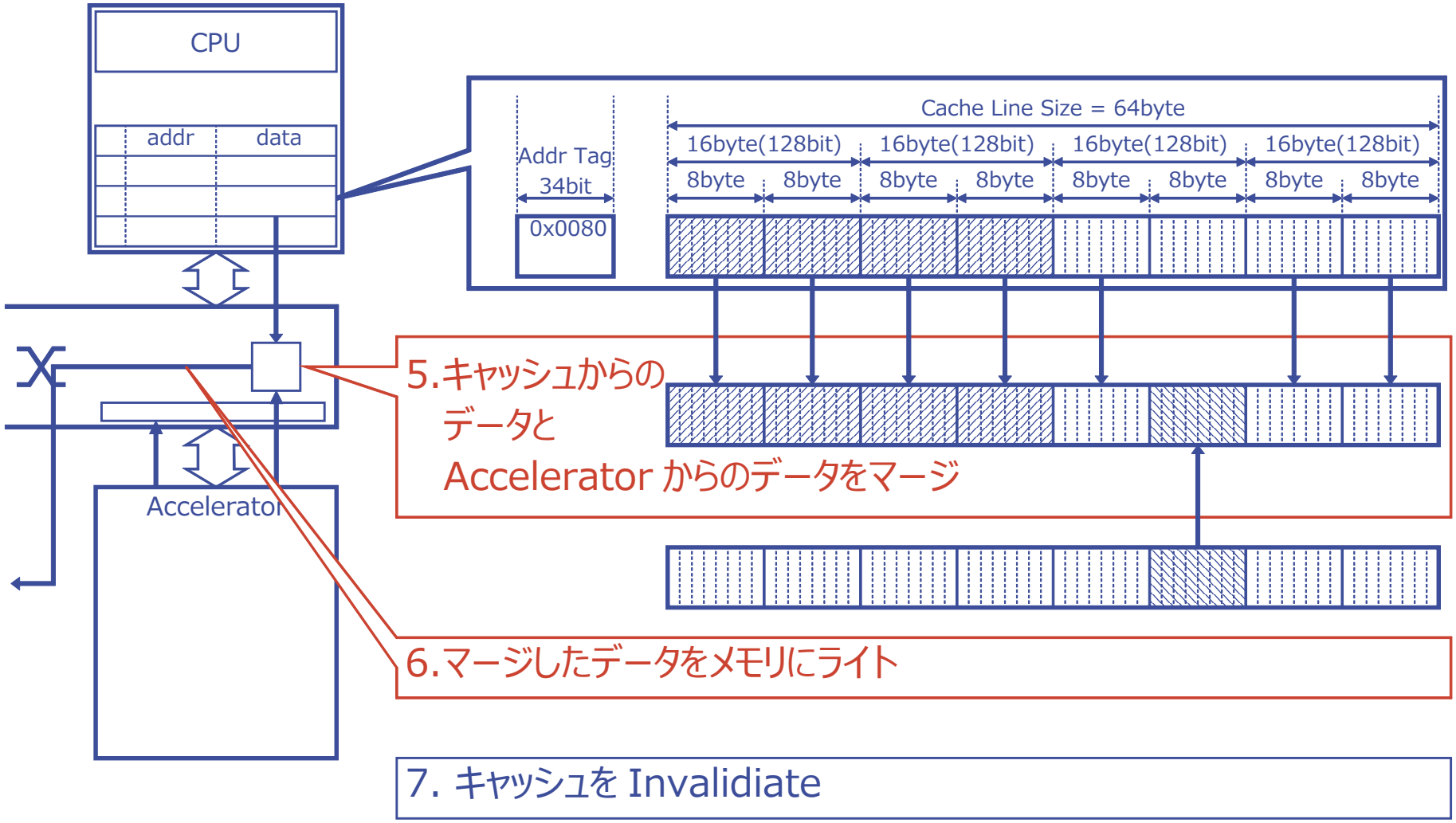
補足　キャッシュラインのアライメント問題



補足　キャッシュラインのアライメント問題



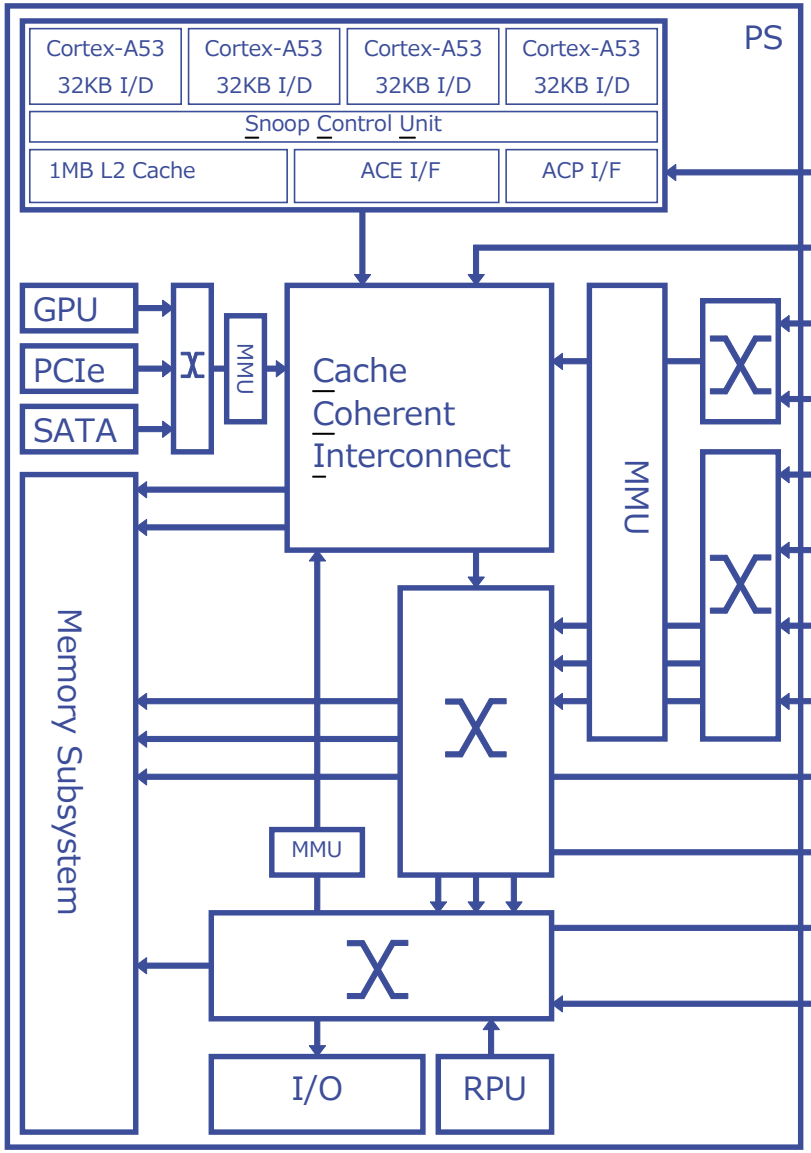
補足　キャッシュラインのアライメント問題



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
  - キャッシュラインのアライメント問題はハードで解決済み
  - 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
  - ARCACHE 信号と AWCACHE 信号を"1111"にしておくこと

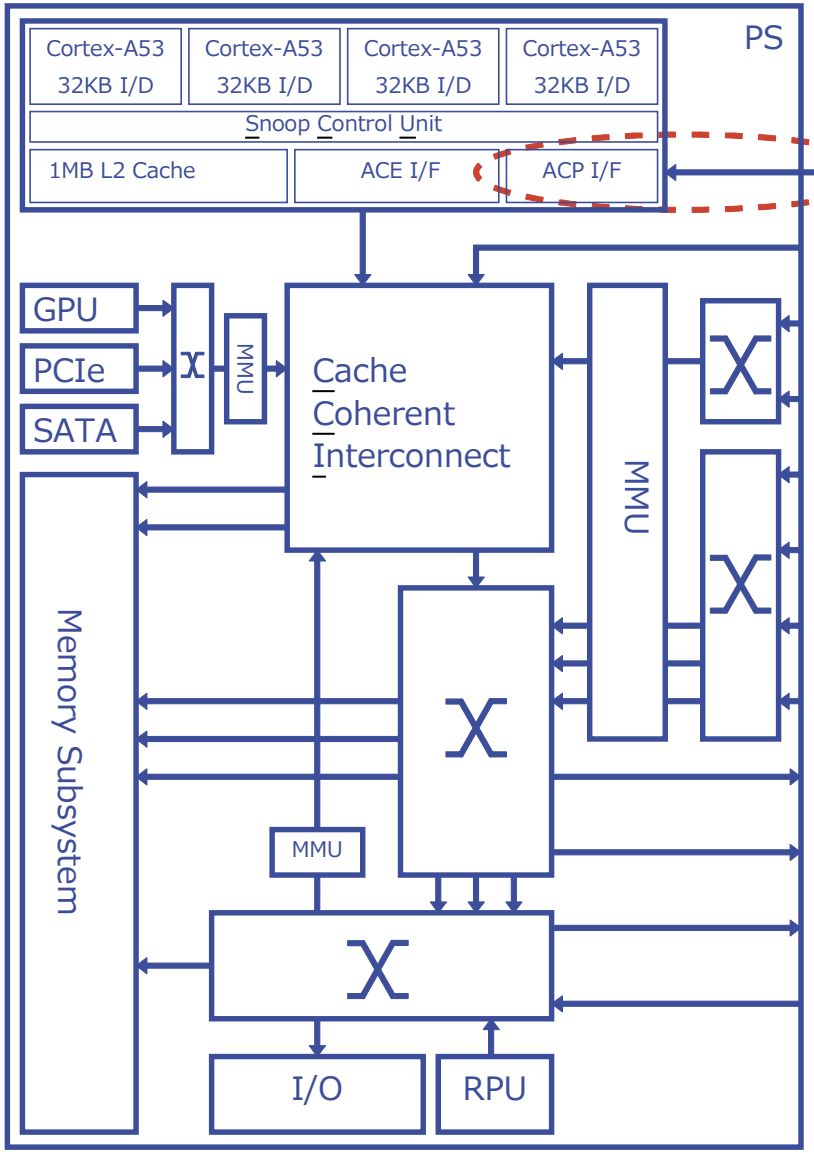
ZynqMP(Zynq Ultrascale+ MPSoC) PS-PL Interface



PL

Interface Name	Addr Width	Data Width	MMU	Cache Coherence
S_AXI_ACP_FPD	40	32 or 64 or 128	No	I/O
S_AXI_ACE_FPD				Full
S_AXI_HPC0_FPD	49		Yes	I/O
S_AXI_HPC1_FPD				
S_AXI_HP0_FPD				No
S_AXI_HP1_FPD				
S_AXI_HP2_FPD				
S_AXI_HP3_FPD				
M_AXI_HPM0_FPD	40			
M_AXI_HPM1_FPD				
M_AXI_HPM0_LPD	32			
S_AXI_LPD	49		Yes / No	I/O / No

ZynqMP の ACP(Accelerator Coherency Port) の制約



ACP Limitations

The ACP accepts only the following (cache-line friendly) transactions.

. 64-byte aligned (of 64-byte) read/write INCR transactions. All write-byte strobes must be same (either enabled or disabled).

. 16-byte aligned (of 16-byte) read/write INCR transactions. Write-byte strobes can have any value.

『 Zynq UltraScale+ MPSoC TRM UG1085 (v1.0) November 24,2015』826

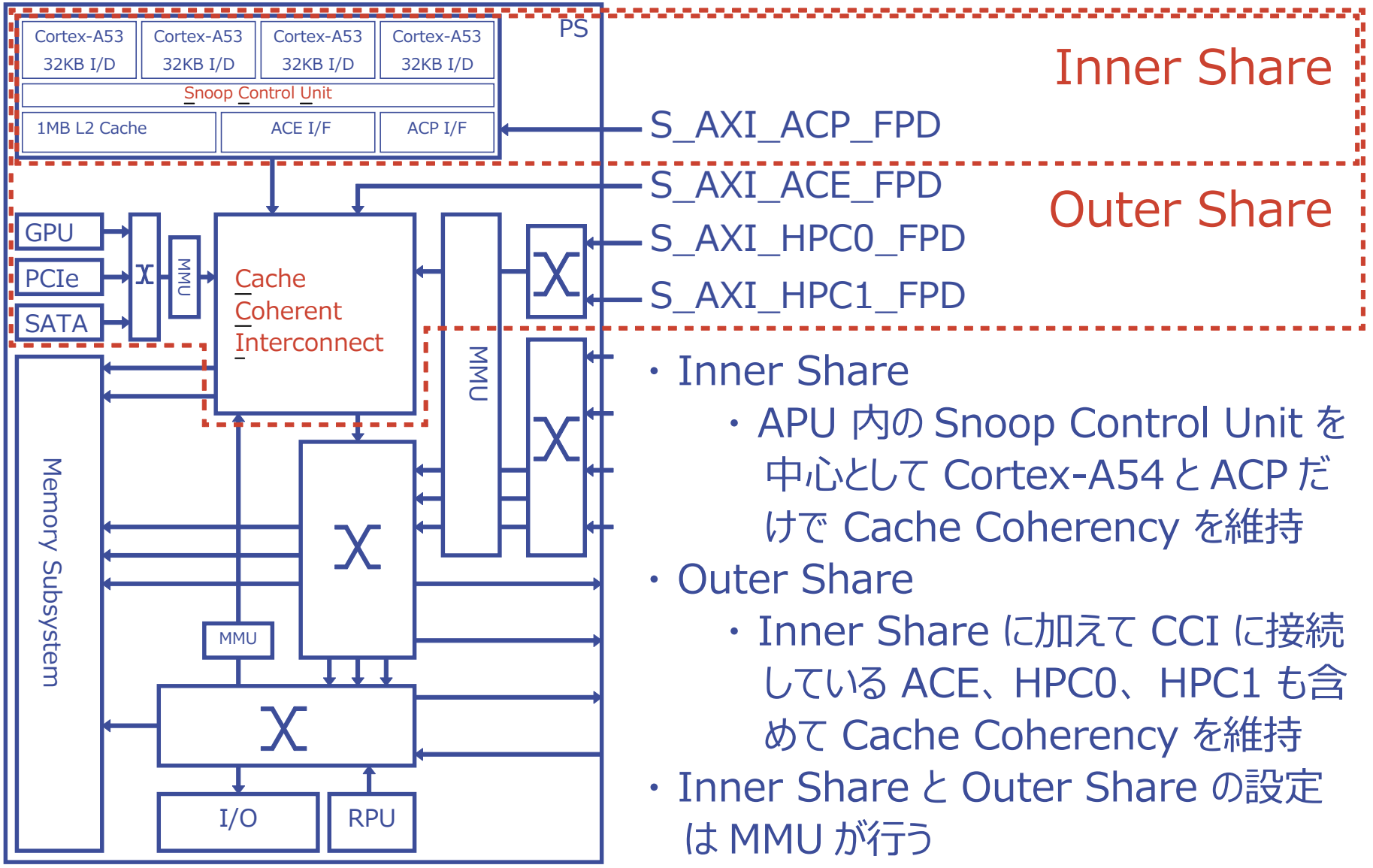
ACP Coherency

The PL masters can also snoop APU caches through the APU’ s accelerator coherency port (ACP). The ACP accesses can be used to (read or write) allocate into L2 cache. However, the ACP only supports aligned 64-byte and 16-byte accesses. All other accesses get a SLVERR response.

『 Zynq UltraScale+ MPSoC TRM UG1085 (v1.0) November 24,2015』228

キャッシュのアライメント問題は PL 側で解決しなければならない

ZynqMP の Inner Share と Outer Share

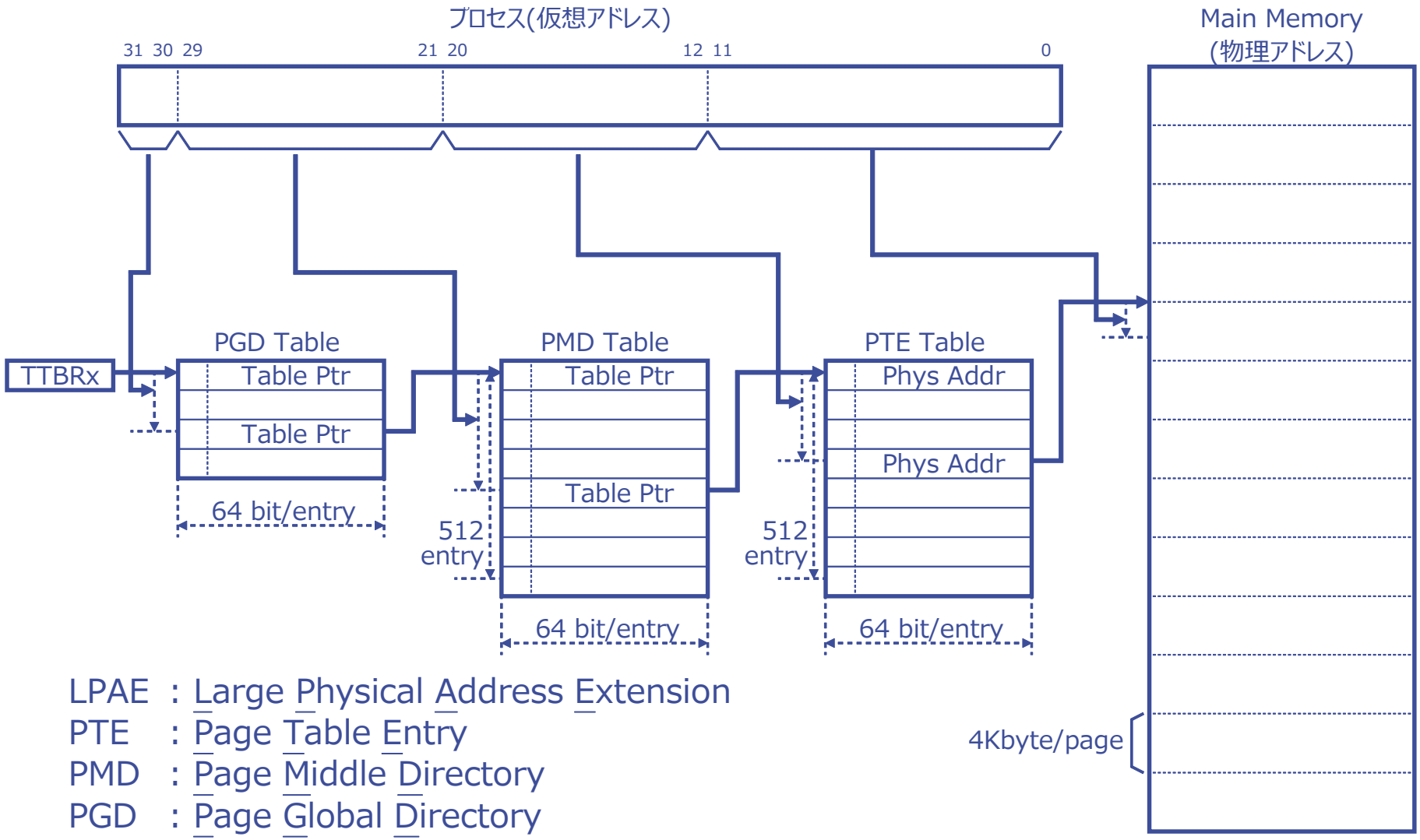


- Inner Share
  - APU 内の Snoop Control Unit を中心として Cortex-A54 と ACP だけで Cache Coherency を維持
- Outer Share
  - Inner Share に加えて CCI に接続している ACE、HPC0、HPC1 も含めて Cache Coherency を維持
- Inner Share と Outer Share の設定は MMU が行う

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

AArch64 PTE(Page Table Entry) の Shareable attribute

PXN : Private Execution permission
UXN : User Execution permission
AF : Access Flag
SH : Shareable attribute
- 00: Non shareable
- 10: Outer shareable
- 11: Inner shareable

AP : Access Permission
NS : the security bit, but only at EL3 and Secure EL1
Indx : the index into the Memory Attribute Indirection Register MAIR_ELn

Linux Kernel 4.14 の AArch64 MMU 対応

arch/arm64/include/asm/pgtable-hwdef.h

```
          :
        (中略)
          :
/*
 * Level 3 descriptor (PTE).
 */
#define PTE_TYPE_MASK           (_AT(pteval_t, 3) << 0)
#define PTE_TYPE_FAULT          (_AT(pteval_t, 0) << 0)
#define PTE_TYPE_PAGE           (_AT(pteval_t, 3) << 0)
#define PTE_TABLE_BIT           (_AT(pteval_t, 1) << 1)
#define PTE_USER                (_AT(pteval_t, 1) << 6)     /* AP[1] */
#define PTE_RDONLY              (_AT(pteval_t, 1) << 7)     /* AP[2] */
#define PTE_SHARED              (_AT(pteval_t, 3) << 8)     /* SH[1:0], inner shareable */
#define PTE_AF                  (_AT(pteval_t, 1) << 10)    /* Access Flag */
#define PTE_NG                  (_AT(pteval_t, 1) << 11)    /* nG */
#define PTE_DBM                 (_AT(pteval_t, 1) << 51)    /* Dirty Bit Management */
#define PTE_CONT                (_AT(pteval_t, 1) << 52)    /* Contiguous range */
#define PTE_PXN                 (_AT(pteval_t, 1) << 53)    /* Privileged XN */
#define PTE_UXN                 (_AT(pteval_t, 1) << 54)    /* User XN */
#define PTE_HYP_XN              (_AT(pteval_t, 1) << 54)    /* HYP XN */
          :
        (後略)
          :
```

SH[1:0] は Inner shareable と Non sharable の 2 択しかない

Linux Kernel 4.14 では Outer Share は未サポート

MMU を使わないで Outer Share でも Cache Coherency 転送する方法

・「 Zynq UltraScale MPSoC Cache Coherency」より
http://www.wiki.xilinx.com/Zynq+UltraScale+MPSoC+Cache+Coherency

5.2 Broadcasting Inner Shareable

This method alters a register of MPSoC to enable inner shareable transactions to be broadcast. The brdc_inner bit of the lpd_apu register in the LPD_SLCR module must be written while the APU is in reset. The requirement to alter the register while the APU is in reset can be accomplished in multiple manners.

(意訳)

・lpd_apu レジスタの brdc_inner ビットを 1 にセットすることにより、Inner shareable なトランザクションを Outer Shareable 領域にもブロードキャストするようになる。要は(強制的に) Outer Shareable 領域も Inner Shareable 領域にしてしまう。

・ただし APU がリセット中にこのレジスタの値を変更する必要がある。つまり、U-Boot や Linux が起動した後ではこのレジスタの値を変更しても効果が無い。

MMU を使わないで Outer Share でも Cache Coherency 転送する方法

PMU(Power Management Unit) から lpd_apu レジスタを設定する

- PMU は Xilinx の Microbraze という CPU 内蔵
- MPSoC のリセット直後はまず PMU が起動する
- PMU が外部 ROM から boot.bin の FSBL を内部 RAM にロード
- FSBL ロード後に PMU が　APU のリセットを解除する
- boot.bin 生成時に PMU が実行するスクリプトっぽいのを埋め込む

詳しい情報は以下の URL 参照

- 「 Zynq UltraScale MPSoC Cache Coherency」 wiki.xilinx
  http://www.wiki.xilinx.com/Zynq+UltraScale+MPSoC+Cache+Coherency

- 『 UltraZed 向け Debian GNU/Linux で AXI HPC port を使う』@Qiita
  https://qiita.com/ikwzm/items/79170301e215b21b14aa

ちなみに「 Zynq UltraScale+ MPSoC Technical Reference Manual」には何も書いていない

Linux Kernel 4.14 を Outer Share 出来るようにパッチ

arch/arm64/include/asm/pgtable-hwdef.h

```
          :
        (中略)
          :
/*
 * Level 3 descriptor (PTE).
 */
#define PTE_TYPE_MASK       (_AT(pteval_t, 3) << 0)
#define PTE_TYPE_FAULT      (_AT(pteval_t, 0) << 0)
#define PTE_TYPE_PAGE       (_AT(pteval_t, 3) << 0)
#define PTE_TABLE_BIT       (_AT(pteval_t, 1) << 1)
#define PTE_USER            (_AT(pteval_t, 1) << 6)     /* AP[1] */
#define PTE_RDONLY          (_AT(pteval_t, 1) << 7)     /* AP[2] */
#define PTE_AF              (_AT(pteval_t, 1) << 10)    /* Access Flag */
#define PTE_NG              (_AT(pteval_t, 1) << 11)    /* nG */
#define PTE_DBM             (_AT(pteval_t, 1) << 51)    /* Dirty Bit Management */
#define PTE_CONT            (_AT(pteval_t, 1) << 52)    /* Contiguous range */
#define PTE_PXN             (_AT(pteval_t, 1) << 53)    /* Privileged XN */
#define PTE_UXN             (_AT(pteval_t, 1) << 54)    /* User XN */
#define PTE_HYP_XN          (_AT(pteval_t, 1) << 54)    /* HYP XN */

#define PTE_SHARED_MASK     (_AT(pteval_t, 3) << 8)
#define PTE_INNER_SHARED    (_AT(pteval_t, 3) << 8)     /* SH[1:0], inner shareable */
#define PTE_OUTER_SHARED    (_AT(pteval_t, 2) << 8)     /* SH[1:0], outer shareable */
#define PTE_SHARED          PTE_INNER_SHARED
          :
        (後略)
          :
```

PTE_SHARED_MASK と PTE_OUTER_SHARED を追加

Linux Kernel 4.14 を Outer Share 出来るようにパッチ

include/linux/mm.h

```
          :
        (中略)
          :
#if defined(CONFIG_X86)
# define VM_PAT                 VM_ARCH_1           /* PAT reserves whole VMA at once (x86) */
#if defined (CONFIG_X86_INTEL_MEMORY_PROTECTION_KEYS)
# define VM_PKEY_SHIFT          VM_HIGH_ARCH_BIT_0
# define VM_PKEY_BIT0VM_HIGH_ARCH_0             /* A protection key is a 4-bit value */
# define VM_PKEY_BIT1VM_HIGH_ARCH_1
# define VM_PKEY_BIT2VM_HIGH_ARCH_2
# define VM_PKEY_BIT3VM_HIGH_ARCH_3
#endif
#elif defined(CONFIG_PPC)
# define VM_SAO                 VM_ARCH_1           /* Strong Access Ordering (powerpc) */
#elif defined(CONFIG_PARISC)
# define VM_GROWSUP             VM_ARCH_1
#elif defined(CONFIG_METAG)
# define VM_GROWSUP             VM_ARCH_1
#elif defined(CONFIG_IA64)
# define VM_GROWSUP             VM_ARCH_1
#elif defined(CONFIG_ARM64)
# define VM_OUTER_SHARED    VM_ARCH_1
#elif !defined(CONFIG_MMU)
# define VM_MAPPED_COPY     VM_ARCH_1           /* T if mapped copy of data (nommu mmap) */
#endif
          :
        (後略)
          :
```

vm_flags のパラメータに VM_OUTER_SHARED を追加

Linux Kernel 4.14 を Outer Share 出来るようにパッチ

mm/mmap.c

```
        :
      (中略)
        :

pgprot_t protection_map[16] __ro_after_init = {
      __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
      __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
};

pgprot_t vm_get_page_prot(unsigned long vm_flags)
{
      return __pgprot(pgprot_val(protection_map[vm_flags &
                              (VM_READ|VM_WRITE|VM_EXEC|VM_SHARED)]) |
                      pgprot_val(arch_vm_get_page_prot(vm_flags)));
}
EXPORT_SYMBOL(vm_get_page_prot);
        :
      (後略)
        :
```

vm_flags から page_prot を生成する関数
- 上の protection_map[] から基本となる page_prot を得る
- アーキテクチャ毎に定義された arch_vm_get_page_prot()の戻り値を論理和 (つまりビットのセットは出来るがビットのクリアは出来ない)

このファイルは他のアーキテクチャでも使うので変更しない
- protection_map[] の中身(__P000～__S111)と arch_vm_get_page_prot() を修正する

Linux Kernel 4.14 を Outer Share 出来るようにパッチ

arch/arm64/include/asm/pgtable-prot.h

```
        :
      (中略)
        :
#define __P000  __pgprot(_PAGE_NONE             & ~PTE_SHARED_MASK)
#define __P001  __pgprot(_PAGE_READONLY         & ~PTE_SHARED_MASK)
#define __P010  __pgprot(_PAGE_READONLY         & ~PTE_SHARED_MASK)
#define __P011  __pgprot(_PAGE_READONLY         & ~PTE_SHARED_MASK)
#define __P100  __pgprot(_PAGE_EXECONLY         & ~PTE_SHARED_MASK)
#define __P101  __pgprot(_PAGE_READONLY_EXEC    & ~PTE_SHARED_MASK)
#define __P110  __pgprot(_PAGE_READONLY_EXEC    & ~PTE_SHARED_MASK)
#define __P111  __pgprot(_PAGE_READONLY_EXEC    & ~PTE_SHARED_MASK)

#define __S000  __pgprot(_PAGE_NONE             & ~PTE_SHARED_MASK)
#define __S001  __pgprot(_PAGE_READONLY         & ~PTE_SHARED_MASK)
#define __S010  __pgprot(_PAGE_SHARED           & ~PTE_SHARED_MASK)
#define __S011  __pgprot(_PAGE_SHARED           & ~PTE_SHARED_MASK)
#define __S100  __pgprot(_PAGE_EXECONLY         & ~PTE_SHARED_MASK)
#define __S101  __pgprot(_PAGE_READONLY_EXEC    & ~PTE_SHARED_MASK)
#define __S110  __pgprot(_PAGE_SHARED_EXEC      & ~PTE_SHARED_MASK)
#define __S111  __pgprot(_PAGE_SHARED_EXEC      & ~PTE_SHARED_MASK)

#define arch_vm_get_page_prot(vm_flags) ¥
        __pgprot((vm_flags & VM_OUTER_SHARED) ? PTE_OUTER_SHARED : PTE_INNER_SHARED)

#endif /* __ASSEMBLY__ */

#endif /* __ASM_PGTABLE_PROT_H */
```

__P000～__S111 の値は
SH[1:0] の部分をクリアしておく

arch_vm_get_page_prot() では vm_flags の VM_OUTER_SHARED フラグの有無により PTE_OUTER_SHARED か PTE_INNER_SHARED を返す

Udmabuf を Outer Share の設定が出来るようにパッチ

udmabuf.c

```
static int udmabuf_driver_file_mmap(struct file *file, struct vm_area_struct* vma)
{
    struct udmabuf_driver_data* this = file->private_data;

    if ((file->f_flags & O_SYNC) | (this->sync_mode & SYNC_ALWAYS)) {
        switch (this->sync_mode & SYNC_MODE_MASK) {
            case SYNC_MODE_NONCACHED :
                vma->vm_flags    |= VM_IO;
                vma->vm_page_prot = _PGPROT_NONCACHED(vma->vm_page_prot);
                break;
            case SYNC_MODE_WRITECOMBINE :
                vma->vm_flags    |= VM_IO;
                vma->vm_page_prot = _PGPROT_WRITECOMBINE(vma->vm_page_prot);
                break;
            case SYNC_MODE_DMACOHERENT :
                vma->vm_flags    |= VM_IO;
                vma->vm_page_prot = _PGPROT_DMACOHERENT(vma->vm_page_prot);
                break;
            default :
                break;
        }
    }
    else {
        if (this->outer_shared) {
            vma->vm_flags |=  VM_OUTER_SHARED;
        }
    }
    vma->vm_private_data = this;
```

outer_shared フラグが真の時は
vm_flags に VM_OUTER_SHARED を論理和する

Udmabuf を Outer Share の設定が出来るようにパッチ

udmabuf.c

```
/**
 * udmabuf_platform_driver_probe() -  Probe call for the device.
 * @pdev:        handle to the platform device structure.
 * Return:      Success(=0) or error status(<0).
 *
 * It does all the memory allocation and registration for the device.
 */
static int udmabuf_platform_driver_probe(struct platform_device *pdev)
{
          :
        (略)
          :
   /*
    * outer shared property
    */
   {
      driver_data->outer_shared = of_property_read_bool(pdev->dev.of_node, "outer-shared");
   }
          :
        (略)
          :
}
```

device-tree に outer-shared プロパティがあれば、
outer_shared フラグを真にする

ZynqMP-FPGA-Linux-Example-3-UltraZed で試してみた

- 『 UltraZed 向け Debian GNU/Linux で AXI HPC port を使う』@Qiita
  https://qiita.com/ikwzm/items/79170301e215b21b14aa
- 上の記事で使った FPGA デザインとサンプルプログラムを使う
  - 詳細は上の記事参照
- boot.bin は https://github.com/ikwzm/ZynqMP-FPGA-Linux のもの
- Linux Kernel は前述のパッチを当てたバージョン
- udmabuf は前述のパッチを当てたバージョン

ZynqMP-FPGA-Linux-Example-3-UltraZed で試してみた

・device tree の udmabuf に outer-shared プロパティを追加

negative2-outershared.dts

```
/dts-v1/;/plugin/;
/ {
        fragment@0 {
                target-path = "/amba_pl@0";
                #address-cells = <2>;
                #size-cells = <2>;
                __overlay__ {
                        :
                        (中略)
                        :
                        negative-udmabuf4 {
                                compatible  = "ikwzm,udmabuf-0.10.a";
                                device-name = "udmabuf4";
                                size = <0x00100000>;
                                outer-shared;
                        };

                        negative-udmabuf5 {
                                compatible = "ikwzm,udmabuf-0.10.a";
                                device-name = "udmabuf5";
                                size = <0x00100000>;
                                outer-shared;
                        };
                };
        } ;
} ;
```

ZynqMP-FPGA-Linux-Example-3-UltraZed で試してみた

- negative2.py を実行してみた

```
root@debian-fpga:/home/fpga/examples/negative2# tbocfg.rb -i negative --dts negative2-outershared.dts
[  100.601124] udmabuf udmabuf4: major number   = 242
[  100.607631] udmabuf udmabuf4: minor number   = 0
[  100.617776] udmabuf udmabuf4: phys address   = 0x0000000070100000
[  100.628796] udmabuf udmabuf4: buffer size    = 1048576
[  100.633913] udmabuf udmabuf4: dma coherent   = 0
[  100.638511] udmabuf udmabuf4: outer shared   = 1
[  100.643113] udmabuf amba_pl@0:negative-udmabuf4: driver installed.
[  100.650801] udmabuf udmabuf5: major number   = 242
[  100.655536] udmabuf udmabuf5: minor number   = 1
[  100.660126] udmabuf udmabuf5: phys address   = 0x0000000070200000
[  100.666205] udmabuf udmabuf5: buffer size    = 1048576
[  100.671325] udmabuf udmabuf5: dma coherent   = 0
[  100.675924] udmabuf udmabuf5: outer shared   = 1
[  100.680530] udmabuf amba_pl@0:negative-udmabuf5: driver installed.
root@debian-fpga:/home/fpga/examples/negative2# python3 negative.py
total:1.435[msec] setup:0.847[msec] xfer:0.044[msec] cleanup:0.544[msec]
total:1.100[msec] setup:0.620[msec] xfer:0.020[msec] cleanup:0.460[msec]
total:1.092[msec] setup:0.615[msec] xfer:0.020[msec] cleanup:0.458[msec]
total:1.094[msec] setup:0.615[msec] xfer:0.020[msec] cleanup:0.459[msec]
total:1.091[msec] setup:0.614[msec] xfer:0.020[msec] cleanup:0.457[msec]
total:1.093[msec] setup:0.615[msec] xfer:0.020[msec] cleanup:0.458[msec]
total:1.117[msec] setup:0.638[msec] xfer:0.021[msec] cleanup:0.459[msec]
total:8.966[msec] setup:0.602[msec] xfer:7.904[msec] cleanup:0.460[msec]
total:1.093[msec] setup:0.614[msec] xfer:0.020[msec] cleanup:0.459[msec]
average_setup_time  :0.642[msec]
average_cleanup_time:0.468[msec]
average_xfer_time   :0.899[msec]
throughput          :291.656[MByte/sec]
np.negative(udmabuf4) == udmabuf5 : OK
```

ZynqMP-FPGA-Linux-Example-3-UltraZed で試してみた

- ちなみに outers-shared プロパティ無しのデバイスツリーをロードして実行してみると

```
root@debian-fpga:/home/fpga/examples/negative2# dtbocfg.rb -i negative --dts negative2.dts
[  534.262992] udmabuf udmabuf4: major number   = 242
[  534.267995] udmabuf udmabuf4: minor number   = 0
[  534.278140] udmabuf udmabuf4: phys address   = 0x0000000070100000
[  534.289763] udmabuf udmabuf4: buffer size    = 1048576
[  534.300443] udmabuf udmabuf4: dma coherent   = 0
[  534.310597] udmabuf udmabuf4: outer shared   = 0
[  534.320752] udmabuf amba_pl@0:negative-udmabuf4: driver installed.
[  534.334268] udmabuf udmabuf5: major number   = 242
[  534.339009] udmabuf udmabuf5: minor number   = 1
[  534.343602] udmabuf udmabuf5: phys address   = 0x0000000070200000
[  534.349671] udmabuf udmabuf5: buffer size    = 1048576
[  534.354794] udmabuf udmabuf5: dma coherent   = 0
[  534.359392] udmabuf udmabuf5: outer shared   = 0
[  534.363995] udmabuf amba_pl@0:negative-udmabuf5: driver installed.
root@debian-fpga:/home/fpga/examples/negative2# python3 negative2.py
total:8.051[msec] setup:0.128[msec] xfer:7.909[msec] cleanup:0.014[msec]
total:0.129[msec] setup:0.077[msec] xfer:0.041[msec] cleanup:0.011[msec]
total:0.105[msec] setup:0.076[msec] xfer:0.019[msec] cleanup:0.010[msec]
total:0.102[msec] setup:0.073[msec] xfer:0.019[msec] cleanup:0.010[msec]
total:0.103[msec] setup:0.073[msec] xfer:0.019[msec] cleanup:0.010[msec]
total:0.102[msec] setup:0.073[msec] xfer:0.019[msec] cleanup:0.010[msec]
total:0.102[msec] setup:0.073[msec] xfer:0.018[msec] cleanup:0.010[msec]
total:0.102[msec] setup:0.073[msec] xfer:0.019[msec] cleanup:0.010[msec]
total:0.101[msec] setup:0.073[msec] xfer:0.019[msec] cleanup:0.010[msec]
average_setup_time  :0.080[msec]
average_cleanup_time:0.011[msec]
average_xfer_time   :0.898[msec]
throughput          :291.949[MByte/sec]
np.negative(udmabuf4) == udmabuf5 : NG
NC Count:7024
```