JUMP-1 共通バス
パケット送受仕様説明書

JUMP-1 設計グループ

Revision History

1.0	'97 09/04	H. Nakashima	第一次仕様作成
1.1	'97 09/10	H. Nakashima	*control_read_req* 除去など。

ver. 1.0 ('97 09/04, H. Nakashima)

第一次仕様作成。

ver. 1.1 ('97 09/10, H. Nakashima)

(1) 9 章の *control_read_req* に関する節を除去。

(2) 9 章に BMP アクセスのための *control_write_req* に関する記述を追加。

(3) 11.10 節, 11.12 節, 11.13 節に `did` = all の例を追加。

目次

1 概説 **1**

1.1 本資料の構成 . . . 2

1.2 用語 . . . 3

1.2.1 ハードウェア ・モジュール / 資源 . . . 3

1.2.2 その他の用語 . . . 4

1.3 memory-command . . . 8

1.3.1 memory-command の決定情報 . . . 8

1.3.2 cachability . . . 8

1.3.3 coherence-policy . . . 9

1.3.4 memory-command の機能 . . . 10

1.4 キャッシュの状態 . . . 11

1.4.1 L2C の状態 . . . 11

1.4.2 cluster-memory の状態 . . . 11

1.5 凡例 . . . 13

1.5.1 memory-command . . . 13

1.5.2 送出される要求パケット . . . 14

1.5.3 他の L2C の動作 . . . 15

1.5.4 MMC/MBPL の動作 . . . 15

1.5.5 応答受領時の動作 . . . 15

1.5.6 想定される用法 . . . 15

1.5.7 パケットの流れと状態遷移 . . . 16

1.6 パケット一覧 . . . 19

2 L2-cachable load **22**

2.1 *read_req* ← **normal-read**, **s-read**, **f-read** (coherence-policy(r) = normal) . . . 24

2.2 *allread_req* ← **normal-read**, **s-read**, **f-read** (coherence-policy(r) = all-read) . . . 31

2.3 *rdinv_req* ← **normal-read** (coherence-policy(r) = read-invalidate) . . . 39

2.4 *n_c_rd_req* ← **normal-read** (coherence-policy(r) = no-coherent) . . . 47

2.5 *rdinv_q_req* ← **q-read**, **p-read**, **q-check** (cache-hit) . . . 50

2.6 *rdinv_p_req* ← **q-read**, **p-read**, **q-check** (cache-miss) . . . 53

2.7 *bq_read_req* ← **bq-read**, **bq-check** (normal) . . . 57

2.8 *bq_nop_req* ← **bq-read**, **bq-check** (migration) . . . 61

3 L2-cachable store **63**

3.1 *read_req* ← **normal-write**, **s-write** (coherence-policy(r) = normal, coherence-policy(w) ≠ invalidate, cache-miss) . . . 65

3.2 *allread_req* ← **normal-write**, **s-write** (coherence-policy(r) = all-read, cache-miss) . . . 66

3.3 *rdinv_req* ← **normal-write** (coherence-policy(r) = read-invalidate ∨ coherence-policy(w) = invalidate, cache-miss), **bq-write** (cache-miss) . . . 67

3.4 *n_c_rd_req* ← **normal-write**(coherence-policy(r) = no-coherent, cache-miss) . . . 68

3.5 *invalidate_via_home_req* ← **normal-write** (coherence-policy(w) = invalidate, shared-hit), **normal-write** (coherence-policy(w) ∈ {update, update-direct}, ¬all-bytes, shared-hit) . . . 69

3.6 *update_via_home_req* ← **normal-write** (coherence-policy(w) = update, all-bytes, shared-hit) 73

3.7 *update_direct_req* ← **normal-write**, **s-write** (coherence-policy(w) = update-direct, all-bytes, shared-hit) . . . 76

3.8 *update_direct_s_req* ← **s-write** (cache-hit, w-wait) . . . 80

3.9 *no_alloc_write_req* ← **s-write** (dummy-page, cache-miss) . . . 83

3.10 *update_via_home_q_req* ← **q-write** . . . 87

3.11 *bq_write_req* ← **bq-write** (cache-hit, last-byte) . . . 89

4 L2-cachable swap **91**

4.1 *rdinv_req* ← **normal-swap** (coherence-policy(s) = invalidate, ¬exclusive-hit) . . . 92

4.2 *swap_req* ← **normal-swap**(coherence-policy(s) = update, ¬exclusive-hit) . . . 95

4.3 *n_c_rd_req* ← **normal-swap** (coherence-policy(s) = no-coherent, cache-miss) . . . 103

5 L2-cachable-prefetch/flush **104**

5.1 *n_c_write_m_req* ← **pf/cf**(M) . . . 106

6 non-cachable load **107**

6.1 *read_req* ← **normal-read**, **s-read**, **f-read** . . . 109

6.2 *rdinv_q_req* ← **q-read** . . . 111

6.3 *rdinv_p_req* ← **p-read** . . . 114

6.4 *rdinv_f_req* ← **q-check** . . . 115

7 non-cachable store **118**

7.1 *write_req* ← **normal-write** . . . 120

7.2 *update_direct_s_req* ← **normal-write** . . . 122

7.3 *update_via_home_q_req* ← **q-write** . . . 124

7.4 *n_c_write_m_req* ← **t-write**, **m-write**, **pf/cf**(M) . . . 126

8 non-cachable swap **127**

8.1 *swap_req* ← **normal-swap** . . . 128

8.2 *swap_m_req* ← **m-swap** . . . 130

9 control-space access **132**

9.1 *control_write_req* ← CSA-write(write-external) 133

10 pseudo-reply **135**

10.1 *writeback_reply* 136

10.2 *nop* 138

11 MMC/MBPL からの要求パケット **139**

11.1 *read_req* 140

11.2 *allread_req* 144

11.3 *rdinv_req* 145

11.4 *invalidate_req* 149

11.5 *update_req* 152

11.6 *no_alloc_write_req* 154

11.7 *swap_req* 157

11.8 *cflush_req* 161

11.9 *injection_req* 164

11.10 *demap_req* 166

11.11 *control_read_req* 168

11.12 *interrupt_req* 170

11.13 *control_write_req* 172

11.14 *error_reply* 174

1 概説

本資料はJUMP-1のクラスタ・バスであるCBUSを介して送受されるパケットについて，以下の事項を説明したものである。

(1) 2次キャッシュ(L2C)が送出する要求パケットが，要素プロセッサ(CPU)のどのような操作に基づいて生成されるか。また，その要求パケットが，他のL2C，メモリ・コントローラ(MMC)，MBP-light(MBPL)において，どのように処理される（と想定される）か。

(2) MMC/MBPLが送出する要求パケットは，どのような状況において送出されると想定されているか。また，その要求パケットが，L2Cにおいてどのように処理されるか。

また本資料はJUMP-1クラスタの実際の実装に基づいたものであり，「あるべき仕様」を定めたものではない。したがって以下の点に注意されたい。

- L2C
  キャッシュ・コントローラ・チップ(CC)をはじめとするLSI等のハードウェア実装に基づいて記述されており，L2Cは本資料に示すパケット送受を行なう。したがって，ごく一部の不具合を含め，本資料に示したパケット送受動作は「実仕様」として容認されなければならない。
  なおL2Cの全体的なハードウェアに関する仕様書は未完成である。またCPUが行なう操作がパケット送受を伴わない場合や，複数のパケット送受を伴う場合については，本資料には記載されていない[†1]。
- MMC
  MMCのパケット送受に関する記述は資料[1]に基づいているが，一部はハードウェア実装に基づき修正を加えている。ただしMMCの設計が完了していないことや，実装に不明な点もあることから，本資料の記述に正確ではない部分がある可能性がある。
- MBPL
  本資料の目的はMBPLのハードウェア/ソフトウェア（ファームウェア）の仕様を定めるものではない。したがって本資料に記載されているMBPLのパケット処理，特にクラスタ間の通信に関する事項は，あくまで「想定」である。ただし，この想定に大きく反するような実装，例えば特定の要求パケットに対して本資料で示すものとは異なる応答を行なうように実装した場合，クラスタ（特にL2C）の動作が不正なものとなることが極めて強く予想される。

なお，本資料に関連する資料としては以下のものがある。

参考文献

[1] JUMP-1設計グループ. JUMP-1メモリコントローラ仕様説明書(ver. 2.0), October 1995.

[2] JUMP-1設計グループ. JUMP-1メモリ・コマンド仕様(ver. 2.1), September 1997.

[3] JUMP-1設計グループ. JUMP-1共通バス仕様書(ver. 4.1), June 1996.

[4] JUMP-1設計グループ. JUMP-1 2次キャッシュ・コントローラ仕様書. (未完).

[5] JUMP-1設計グループ. JUMP-1 2次キャッシュ内部資源仕様書(ver. 2.0), September 1997.

[6] JUMP-1設計グループ. Doubleword-Queue仕様書. (未完).

[7] 中島, 五島, 福島, 森, 富田. Block Queueの実装. 並列・分散処理研究推進機構 カテゴリー1並列分散処理アーキテクチャ技術 成果報告書, 第3.1章. 並列・分散処理研究推進機構, March 1997.

[8] SPARC Technology Business. *SuperSPARC & MultiCache Controller User's Manual*, 1994.

[†1] 本資料に加筆して，インタフェースに関する仕様書を作成する予定である。

1.1 本資料の構成

本資料の 2 章以下では，以下の構成にしたがってパケット送受に関する説明を行なう。

2. L2-cachable load

3. L2-cachable store

4. L2-cachable swap
L2C にキャッシュ可であるような (L2-cachable) アドレスに対して CPU が行なう read （2 章），write （3 章），swap （4 章）の結果，どのようなパケットが送受されるかを示す。

5. L2-cachable prefetch/flush
L2C にキャッシュ可であるような (L2-cachable) アドレスに対して CPU が行なう write の中には，プリフェッチ (prefetch) あるいはキャッシュの無効化 (flush) を行なうものがある。5 章ではこの prefetch/flush 操作の結果，どのようなパケットが送受されるかを示す。

6. non-cachable load

7. non-cachable store

8. non-cachable swap
L2C キャッシュにキャッシュ不可であるような (non-cachable) アドレスに対して CPU が行なう read （6 章），write （7 章），swap （8 章）の結果，どのようなパケットが送受されるかを示す。

9. pseudo-reply
L2C が送出する疑似的な応答 (pseudo-reply) について説明する。

10. MMC/MBPL からの要求パケット
MMC/MBPL が送出する要求パケットのうち L2C が受理するものについて，L2C の動作などを説明する。

1.2 用語

1.2.1 ハードウェア・モジュール / 資源

BMP
: CBUS のパケット送受の際に参照される L2C の「裏」ディレクトリ (back map)。L2C の各ラインの状態は BMP と FMP (front map) の双方に重複あるいは分割して格納される。読出要求に対する応答義務 (ownership) の有無は，BMP だけが保持している。

CBUS
: JUMP-1 のクラスタ・バス。CBUS を構成する信号は `CBUS:x_crply` のように表記し[†1]，本資料では特に以下の信号を参照する。

  - `x_crply`
    読出要求パケットに対して，いずれかの L2C が応答意志を表明すると LO (0^T) になる。
  - `x_msh`
    要求 / 応答パケットの対象ラインを所有する L2C が存在すると LO (0^T) になる。なお，要求パケットを送出した L2C 自身はラインを所有していても $\texttt{x_msh} = 0^T$ とはしないが，応答パケットを送出した L2C 自身は (応答送出時にラインを所有していれば) $\texttt{x_msh} = 0^T$ とする。

CC
: 2 次キャッシュ・コントローラ。

CPU
: 要素プロセッサ SuperSPARC。

DAR
: L2C のデータ・アレイ。

DTAG
: クラスタ・メモリ (cluster-memory) の各 dw に付された 2 ビットのタグ。`DTAG:{wwait|rwait}` は，I-structure に対する書込 (`wwait`) または読出 (`rwait`) を待つようなスレッドの存在を意味する。

FMP
: CPU からのアクセスの際に参照される L2C の「表」ディレクトリ (front map)。L2C の各ラインの状態は FMP と BMP (back map) の双方に重複あるいは分割して格納される。FMP は，読出要求に対する応答義務 (ownership) の有無以外の情報を，全て保持している。FMP の各エントリのフィールドは `FMP:state` のように表記し，本資料では以下を参照する。

  - `state.{v|x|l|d}`
    ラインの状態について，valid, exclusive, local, dirty をそれぞれ示す (1.4.1 節参照)。
  - `sync[3:0]`
    ラインの各 dw に付された同期ビット。
  - `ww`
    ライン中の 4 dw の中に，$\texttt{DTAG:wwait} = 1$ であるような dw が存在していることを示す。すなわち $\texttt{FMP:ww} = \sum \texttt{DTAG:wwait}$ であり，`DTAG:wwait` の完全な情報を保持してはいないことに注意せよ。また，FMP は `DTAG:rwait` に関する情報を保持しない。
  - `tag[31:20]`
    ラインに対応するアドレスのビット $[31:20]$。

L1C
: CPU が持つ 1 次キャッシュ。

L2C
: 2 次キャッシュ。CC, DAR, FMP, BMP, および本資料では言及しない CB (cache buffer) からなる。

[†1] `x_crply` などの制御信号線は，CBUS への入出力に対応して `x_crply_i` と `x_crply_o` のように対になっているが，本資料の説明のためには入出力の区別が不要であるので省略する。

L2CCTL
: L2C の内部レジスタ。L2CCTL のフィールドは L2CCTL.ctype のように表記し，本資料では以下を参照する。

  - ctype[1:0]
    4 種のメモリ ・コマンド ・セットのいずれを用いるかを定める。1.3.1 節参照。
  - l2ccbl
    一部のメモリ ・コマンドに対する L2C のキャッシュ可否を定める。1.3.2 節参照。
  - wacks, wackq, wackbq
    メモリ ・コマンド **s-write** (wacks), **q-write** (wackq), **bq-write** (wackbq) について，1 であれば操作完了を STBAR で確認できることを示す。

MBPL
: (チップではなくモジュールとしての) MBP-light。

MMC
: クラスタ ・メモリ ・コントローラ。

STAG
: cluster-memory の各ラインに付加されたラインの状態情報。STAG は以下のビットからなり，STAG:valid のように表記する。

  - valid
    ラインが有効であることを示す。
  - global
    ラインがクラスタ間で共有されていることを示す。
  - wr_trap
    MMC によるラインへの書込時に，MBPL への割出しを行なうことを示す。
  - rd_trap
    MMC によるラインの読出時に，MBPL への割出しを行なうことを示す。

fe
: ラインの各 dw に付された同期ビット。

VBUS
: CPU と L2C を結合するバス。VBUS を構成する信号は VBUS:addr のように表記し，本資料では特に以下の信号を参照する。

  - addr[35:0]
    物理アドレス。
  - data[63:0]
    データ。

1.2.2 その他の用語

$0^F/1^T$
: 正論理のビット値。0^F が偽，1^T が真である。

$0^T/1^F$
: 負論理のビット値。0^T が真，1^F が偽である。

all-bytes
: update 系の書込において，ライン中の全ての word について，全 byte が更新されたかあるいは全 byte が更新されていないかの，いずれかであることを示す。すなわち一部の byte のみが更新されたような word が存在しないことを示す。

all-read
: キャッシュの一貫性維持管理ポリシー (coherence-policy) の一種。cache-miss の際に他のキャッシュにもラインを注入 (injection) する。1.3.3 節参照。

block-load
: cache-miss などに伴うラインの取得。

block-queue
: 一つの要素の大きさがラインであるようなキュー。JUMP-1 では任意のラインを block-queue とすることができる。1.3.4 節参照。

bus-error
: L2C が CPU に通知する例外の一種。一般に，メモリ・アクセスに関する各種の規約に違反した場合に通知される。

cachability
: キャッシュ可否。

cache-hit
: (L2C の) ヒット。

cache-miss
: (L2C の) ミス。

clean
: ラインが除去される際に cluster-memory に対する書き戻しが不要であることを意味する。なお，必ずしもラインの値が cluster-memory の値と一致しているとは限らない。1.4.1 節参照。

coherence-policy
: キャッシュの一貫性維持管理ポリシー。1.3.3 節参照。

control-space access
: control-space と呼ばれる特別なメモリ空間に対する CPU のアクセス。

CSA-read
: control-space に対する読出。

CSA-write
: control-space に対する書込。

dequeue
: キュー (dw-queue, block-queue) の先頭要素の除去操作。

dirty
: L2C が所有しているラインの値が cluster-memory の値と一致しておらず，ラインが除去される際には cluster-memory に対する書き戻しが必要であることを意味する。なお「書き戻し義務」を持たない場合は，cluster-memory と一致していなくても dirty とは呼ばない。1.4.1 節参照。

dummy-page
: 実体が cluster-memory には存在しないようなページ。物理アドレスのビット 29 が 1^T である。L1C, L2C にはキャッシュすることができる。

dw
: doubleword。

dw-queue
: 一つの要素の大きさが dw であるようなキュー。JUMP-1 では任意の dw を dw-queue とすることができる。1.3.4 節参照。

empty
: I-structure が保有するデータが有効ではない，または dw-queue, block-queue が空であることを意味する。

enqueue
: キュー (dw-queue, block-queue) の末尾へ要素を付加する操作。

exclusive
: 特定の L2C だけがラインを所有していて，かつその事実を認識している状態。1.4.1 節参照。

exclusive-hit
: exclusive なラインに対する cache-hit。

full
: I-structure が保有するデータが有効である，または dw-queue, block-queue が空でないことを意味する。1.3.4 節参照。

global-writeback
: 他のクラスタ (一般には home-cluster) の cluster-memory に対するラインの書き戻し。

globally-shared
: ラインがクラスタ間で共有されている (可能性がある) 状態。1.4.1 節参照。

home-cluster
: ラインが属するページの共有情報を管理するクラスタ。

hw
: halfword。

I-structure
: データの他に，その有効性を示す同期ビットが付加された同期構造体。JUMP-1 では任意の dw を I-structure とすることができる。

injection
: L2C に対してラインを注入する操作。

invalidate
: coherence-policy の一種。書込時に他のコピーを無効化する。1.3.3 節参照。

L1-cachable
: L1C にキャッシュ可であること。

L2-cachable
: L2C にキャッシュ可であること。

locally-shared
: ラインがクラスタ間で共有されておらず，かつその事実を認識している状態。1.4.1 節参照。

memory-command
: 物理アドレスの上位 4 ビットなどで指定されるメモリ・アクセスの方法。I-structure, dw-queue, block-queue などに対する高機能アクセスや，coherence-policy が指定される。1.3 節参照。

no-coherent
: coherence-policy の一種。一貫性維持を全く行なわない。1.3.3 節参照。

non-cachable
: L1C, L2C にキャッシュ不可であること。

owner
: あるラインに対して CBUS からの読出要求に対する応答義務 (ownership) を有するような L2C, またはそのような L2C が保有しているラインの状態。1.4.1 節参照。

ownership
: あるラインに対して CBUS からの読出要求に対する応答義務。1.4.1 節参照。

pseudo-reply
: 要求に対応する応答ではないが，L2C が自発的に CBUS に送出するパケットの中で応答として取り扱われるもの。

read-invalidate
: coherence-policy の一種。読出が cache-miss した際に，他のコピーを無効化する。1.3.3 節参照。

shared
: ラインが他の L2C またはクラスタに共有されている (可能性がある) 状態。1.4.1 節参照。

shared-hit
: exclusive なラインに対する cache-hit。

uncorrectable-error
: L2C が CPU に通知する例外の一種。一般に，cluster-memory の 2 ビット ･エラーや，FMP のパリティ ･エラーなどの際に，read/swap に対する応答として通知される。

undefined-error
: L2C が CPU に通知する例外の一種。一般に，empty であるような I-structure, dw-queue, block-queue の読出への応答として通知される。

update
: coherence-policy の一種。書込時に home-cluster 経由で他のコピーを更新する。なお update-direct と総称して update 系と呼ぶことがある。1.3.3 節参照。

update-direct
: coherence-policy の一種。書込時に home-cluster を経由せず他のコピーを更新する。1.3.3 節参照。

writeback
: dirty なラインが L2C から除去される際に行なわれる cluster-memory への書き戻し操作。

1.3 memory-command

本節では memory-command について表 1.1 に基づいて概説する。詳細については [2] を参照せよ。

1.3.1 memory-command の決定情報

memory-command は表 1.1 に示した以下の情報によって定まる。

(1) mcmd
物理アドレスの上位 4 ビット ([35 : 32])。memory-command の基本的な枠組を定める。

(2) type
L2CCTL.ctype[1:0] の値。4 種類のセットの中のいずれを選択するかを定める。

(3) load, store, swap
CPU が行なうアクセスの種類。

(4) D, W, H, B
CPU が行なうアクセスの幅であり，D(oubleword), W(ord), H(alfword), B(yte) のいずれかである。なお swap のアクセス幅は word または byte のみであり，どちらも同じ memory-command となる。

1.3.2 cachability

表 1.1 の cache 欄は memory-command の L2-cachability を示し，page table entry に指定された L1-cachability と関連して以下の意味を持つ。

表 1.1: memory-command 一覧

mcmd	type	cache	coh-pol	load				store				swap
				D	W	H	B	D	W	H	B	
0000	0–3	no		N	N	N	N	N	N	N	N	N
0001	0–3	no		—	—	—	RG	M	T	—	RG	M
0010	0–3	no(S)	read-inv	Q	Q	QC	Q	Q	Q	Q	Q	—
0011	0–3	no(S)	read-inv	P	P	QC	P	Q	Q	Q	Q	—
0100	0,1	no(S)	update-drct	S0	S0	F0	S0	S0	S0	PF	—	—
	2	no		S0	S0	F0	S0	S0	S0	PF	—	—
	3	y/n(S)	update-drct	S0	S0	S0	S0	S0	S0	PF	—	—
0101	0,1	no(S)	update-drct	S1	S1	F1	S1	S1	S1	PF	—	—
	2	no		S1	S1	F1	S1	S1	S1	PF	—	—
	3	y/n(S)	update-drct	S1	S1	S1	S1	S1	S1	PF	—	—
0110	0,1	no		P	P	QC	P	Q	Q	Q	Q	—
	2	no(S)	allread-up-drct	S0	S0	F0	S0	S0	S0	PF	—	—
	3	y/n(S)	update-drct	F0	F0	F0	F0	S0	S0	PF	—	—
0111	0,1	y/n	allread-up	N	N	N	N	N	N	PF	N	N
	2	no(S)	allread-up-drct	S1	S1	F1	S1	S1	S1	PF	—	—
	3	y/n(S)	update-drct	F1	F1	F1	F1	S1	S1	PF	—	—
1000	0–3	yes	read-inv	BQ	BQ	BQ	BQ	BQ	BQ	BQ	BQ	—
1001	0–2	y/n	invalidate	N	N	N	N	N	N	N	N	N
	3	no(Q)	read-inv	BC	BC	BC	BC	—	—	—	—	—
1010	0–3	y/n	invalidate	N	N	N	N	N	N	PF	N	N
1011	0–3	y/n	read-inv	N	N	N	N	N	N	PF	N	N
1100	0–3	y/n	no-coherent	N	N	N	N	N	N	N	N	N
	3	y/n	no-coherent	N	N	N	N	N	N	PF	N	N
1101	0–3	y/n	rdinv-up	N	N	N	N	N	N	PF	N	N
1110	0–3	y/n	update-drct	N	N	N	N	N	N	N	N	N
1111	0–3	y/n	update	N	N	N	N	N	N	N	N	N

(1) yes
page table の指定が L1-cachable でなければならず，また無条件に L2-cachable である。

(2) y/n
page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば (またその時に限り) L2-cachable である。

(3) y/n(S)
page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば L2-cachable である。L1-cachable でなければ，L2-cachability は `L2CCTL.l2ccbl` に従う。

(4) no
page table の指定が L1-cachable であってはならず，また無条件に L2-non-cachable である。

(5) no(S)
page table の指定は L1-cachable であってはならず，L2-cachability は `L2CCTL.l2ccbl` に従う。

(6) no(Q)
page table の指定は L1-cachable であってはならないが，常に L2-cachable と解釈される。

1.3.3 coherence-policy

表 1.1 の coh-pol 欄は，L2-cachable であるときの coherence-policy を示す。coherence-policy には読出，書込，swap の 3 種類があり，表 1.1 の表記との対応を表 1.2 に示す。

(1) read (coherence-policy(r))
主として読出が cache-miss した際のラインの取得方法を意味する。

  (a) normal
  付随的な操作を行なわない。

  (b) all-read
  他の L2C やクラスタに対する injection を行なう。

  (c) read-invalidate
  他のコピーを無効化する。

  (d) no-coherent
  一貫性を無視してラインを取得する。

(2) write (coherence-policy(w))
主として書込が shared-hit した際の更新操作を意味する。

  (a) invalidate
  他のコピーを無効化する。書込の cache-miss では読出の coherence-policy に関わらず read-invalidate によってラインを取得する。

  (b) update
  globally-shared であれば home-cluster を経由して他のコピーを更新する。書込の cache-miss では読出の coherence-policy にしたがってラインを取得する。

  (c) update-direct
  globally-shared であれば home-cluster を経由せずに他のコピーを更新する。書込の cache-miss では読出の coherence-policy にしたがってラインを取得する。

  (d) no-coherent
  一貫性を無視するので，他にコピーがあっても何もしない。書込の cache-miss では読出の coherence-policy (no-coherent) にしたがってラインを取得する。

(3) swap (coherence-policy(s))
swap が exclusive-hit しなかった際のライン取得 / 更新操作を意味する。

  (a) invalidate
  他のコピーを無効化する。

表 1.2: coherence-policy

coh-pol	read	write	swap
invalidate	normal	invalidate	invalidate
read-inv	read-invalidate	invalidate	invalidate
update	normal	update	update
update-drct	normal	update-direct	update
allread-up	all-read	update	update
allread-up-drct	all-read	update-direct	update
rdinv-up	read-invalidate	update	invalidate
no-coherent	no-coherent	no-coherent	no-coherent

(b) update
globally-shared であれば home-cluster を経由して他のコピーを更新する。

(c) no-coherent
一貫性を無視してラインを取得し，他にコピーがあっても何もしない。

1.3.4 memory-command の機能

表 1.1 の load, store, swap 欄の記号は，以下に示す memory-command を示す。

(1) N = {**normal-read**, **normal-write**, **normal-swap**}
付加的な機能が一切ない，通常の read, write, swap である。

(2) RG = {**rg-read**, **rg-write**}
L2C の内部資源に対するアクセスである[†1]

(3) M = {**m-write**, **m-swap**}

(4) T = **t-write**
MBPL のプログラムの起動要求である。

(5) {S0, S1} = {**s-read**, **s-write**}
I-structure に対するアクセスであり，S0 は同期ビットが 0 の場合に empty であることを，S1 は 1 の場合に empty であることを意味する。**s-read** は I-structure が empty である場合，L1-cachable であれば 0 を，L1-non-cachable であれば undefined-error を，それぞれ返す。

(6) {F0, F1} = **f-read**
I-structure の full/empty を知るための読出であり，full であれば−1 を，empty であれば 0 を返す。F0 は同期ビットが 0 の場合に empty であることを，F1 は 1 の場合に empty であることを意味する。

(7) Q = {**q-read**, **q-write**}

(8) P = **p-read**
dw-queue に対するアクセスである。dw-queue が empty である場合，**q-read** は undefined-error を，**p-read** は 0 を返す。

(9) QC = **q-check**
dw-queue の full/empty を知るための読出であり，full であれば−1 を，empty であれば 0 を返す。

(10) BQ = {**bq-read**, **bq-write**}
block-queue に対するアクセスである。**bq-read** は block-queue が empty である場合 undefined-error を返す。

(11) BC = **bq-check**
block-queue の full/empty を知るための読出であり，full であれば−1 を，empty であれば 0 を返す。

[†1] **rg-read**, **rg-write** はパケットを生成しないので，本資料では触れない。

1.4 キャッシュの状態

1.4.1 L2C の状態

L2C の各ラインの状態 S は ；

$$S = \texttt{\{V|I\}\{X|S\}\{L|G\}\{D|C\}\{C|N\}}$$

の 5 ビットによって表現される。

(1) {V|I}
有効であるか (V = Valid), 無効であるか (I = Invalid) を意味する。

(2) {X|S}
他のキャッシュが共有していないことが判明しているか (X = eXclusive), そうでないか (S = Shared) を意味する。

(3) {L|G}
他のクラスタが共有していないことが判明しているか (L = Local), そうでないか (G = Global) を意味する。

(4) {D|C}
ラインの値が cluster-memory の値と一致していないためラインが除去される際には cluster-memory に対する書き戻しが必要であるか (D = Dirty), そうでないかを (C = Clean) 意味する。なお 「書き戻し義務」がなければラインの値が cluster-memory の値と一致しなくても clean である。

(5) {O|N}
CBUS からの読出要求に応答する義務があるか (O = Owner), そうでないか (N = Non-owner) を意味する。owner であるような L2C はクラスタ内に高々一つしか存在しない。また clean なラインについては owner が存在するとは限らない。

これらのビットは独立ではないため，実際に生じる状態は表 1.3 に示すものに限られる。すなわち no-coherent に例外的な VXLDN , VSGDN を除き，以下の関係が成り立つ。

- X ⇒ L, G ⇒ S
- X ⇒ O, N ⇒ S
- G ⇒ C, D ⇒ L
- D ⇒ O, N ⇒ C

1.4.2 cluster-memory の状態

cluster-memory の各ラインの状態 S は ；

$$S = \texttt{\{V|I\}\{L|G\}}$$

の 2 ビットによって表現され，以下の 3 状態をとる。

表 1.3: L2C の状態

I****	invalid
VXLDO	exclusive-dirty
VXLCO	exclusive-clean
VSLDO	locally-shared-dirty
VSLCO	locally-shared-clean-owner
VSLCN	locally-shared-non-owner
VSGCO	globally-shared-owner
VSGCN	globally-shared-non-owner
VXLDN	exclusive-dirty-non-owner
VSGDN	globally-shared-dirty

(1) I* = invalid
クラスタ内にはラインが存在しない。

(2) VL = local
クラスタ内にラインが存在し，かつ他のクラスタが共有していないことが判明している。なお，cluster-memory のラインが有効であること (すなわち clean であること) は必ずしも意味しない。

(3) VG = global
クラスタ内にラインが存在し，かつ他のクラスタが共有している可能性がある。この状態では cluster-memory のラインは必ず有効であり (すなわち clean)，また home-cluster にも有効なラインが存在する。

なおクラスタに存在するラインについて，以下の事実を効率的に (すなわち CBUS に対するパケット送受を伴わずに) MMC/MBPL が知る方法は存在しない。

- VL であるようなラインが dirty であるか clean であるか。
- V* であるようなラインを所有する L2C が存在するか否か。なお MMC/MBPL が行なったラインの読出要求に L2C が応答しない場合であっても，ラインを所有する L2C が存在することがある。

1.5 凡例

この節では 2 章以下での説明に使用する図表や記号について概説する。

1.5.1 memory-command

2 ~ 8 章の冒頭には，章中で説明するパケットを生成するような memory-command の一覧表が掲げられている。以下の表の例は表 3.1 の一部分である。

表 3.1: L2-cachable store によるパケット生成

mcmd	type	size	C	H	*1	mnem	pcmd/scmd	section
					⋮		⋮	⋮
0111	$\{0,1\}$	$\neg hw$	y/n	0^F		**normal-write** (all-read/update)	*allread_req*	3.2節
0111	$\{0,1\}$	$\neg hw$	y/n	1^T	$m = 0^F$	**normal-write** (all-read/update)	*invalidate_via_home_req*	3.5節
0111	$\{0,1\}$	$\neg hw$	y/n	1^T	$m = 1^T$	**normal-write** (all-read/update)	*update_via_home_req*	3.6節
0111	2	$\neg hw$	no(S)	0^F	$d = 0^F$	**s-write** (all-read/update-direct)	*allread_req*	3.2節
0111	2	$\neg hw$	no(S)	0^F	$d = 1^T$	**s-write** (all-read/update-direct)	*no_alloc_write_req*	3.9節
0111	2	$\neg hw$	no(S)	1^T	$w = 0^F$	**s-write** (all-read/update-direct)	*update_direct_req*	3.7節
0111	2	$\neg hw$	no(S)	1^T	$w = 1^T$	**s-write** (all-read/update-direct)	*update_direct_s_req*	3.8節
0111	3	$\neg hw$	y/n(S)	0^F	$d = 0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0111	3	$\neg hw$	y/n(S)	0^F	$d = 1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0111	3	$\neg hw$	y/n(S)	1^T	$w = 0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.8節
0111	3	$\neg hw$	y/n(S)	1^T	$w = 1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
					⋮		⋮	⋮

(1) mcmd, type, size
パケットを生成する memory-command を定める物理アドレスの上位 4 ビット (mcmd), `L2CCTL.ctype` の値 (type), およびアクセス幅 (size) を示す。この例では，mcmd $= 0111$ かつ size $\neq hw$ である場合，type $\in \{0,1\}$ であれば **normal-write**, type $\in \{2,3\}$ であれば **s-write** であることが示されている。

(2) C
1.3.2 節に示した L2-cachability。この例では type $\in \{0,1\}$ であれば y/n, type $= 2$ であれば no(S), type $= 3$ であれば y/n(S) であることが示されている。

(3) H
パケットが cache-miss の際に送出されるか (0^F), cache-hit (この場合は shared-hit) の際に送出されるか (1^T) を示す。

(4) *1
送出されるパケットを定めるその他の条件。

(5) mnem
1.3.4 節に示した memory-command の mnemonic。また必要に応じて coherence-policy を括弧内に記載し，この例では読出と書込の coherence-policy が示されている。例えば mcmd $= 0111$ で type $\in \{0,1\}$ の **normal-write** では，coherence-policy(r) が all-read, coherence-policy(w) が update である。なお coherence-policy については 2 章と 5 章では読出に関するもののみが，また 4 章では swap に関するもののみが，それぞれ示される。

(6) pcmd/scmd
送出されるパケットの mnemonic。

(7) section
送出されるパケットを説明している節。

1.5.2 送出される要求パケット

パケットを説明する各節の 「送出される要求パケット」には[†1]，その節で解説するパケットの各フリットとアドレス ・フリットの各フィールドの値が以下のように示される。なお，パケットの形式についての詳細は [3] を参照せよ。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1010/000	= *update_direct_req*
size	= 110	= masked block
ack	= 1	= with ack
did	= 111	= all
sid	= *pid*	= own ID
drty	= $\perp$	
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= VBUS.addr ($[5:4] = 0$)
wwe[0:3]	= *mdfd*	= word modified
data[0:3]	= *data*	= VBUS.data

(1) plen
アドレス ・フリットに付加されたパケット長，すなわち CBUS:x_astb $= 0^T$ のタイミングにおける CBUS:wwe の値。この例では $10 = 5$ flit であることが示さている。

(2) injmask
1^T であれば injection を行なうことを示す。この例では injection をしない (0^F) ことが示されている。device-ID が i であるような L2C は injmask[i] のみを入出力するので，一般に injmask[i] の値のみを示す。なお，この例のように L2C が送出するパケットについては，injmask[i] が他のビットにもコピーされる。すなわち L2C が送出するパケットでは injmask[3:0] $\in \{0000, 1111\}$ である。

(3) ccbl
1^T であれば cachable なアドレスを対象としたパケットであることを示す。この例では cachable (1^T) である。

(4) pcmd/scmd
パケットの種類を指定する。この例では *update_direct_req* は pcmd $= 1010$, scmd $= 000$ であることが示されている。

(5) size
パケットが操作対象とするデータ幅を示す。この例では word 単位に更新の有無が指定された masked block (110) であることが示されている。

(6) ack
1^T であれば要求の完了確認を行なうことを示す。この例では完了確認を行なう (1^T) ことが示されている。

(7) did
パケットの宛先を示す。この例では全てのデバイスが宛先である $(111 = \text{all})$ ことが示されている。

(8) sid
パケットの送信元を示す。この例では L2C 自身の device ID であることが示されている。

(9) drty
1^T であれば対象とするラインが dirty であることを示す。この例では drty に意味がないことが示されている。一般に$\perp$ は pcmd/scmd および ccbl が定めるパケットにとって，フィールドの値が本質的に無意味であるため ；

- L2C が送出する場合には，有意な値ではない。

[†1] 10 章では 「送出される疑似応答パケット」。

- L2C が受理する場合には，どのような値であってもよい。

(10) clsh
1^T であれば対象とするラインが globally-shared であることを示す。この例では FMP:state.l の論理否定であることが示されている。

(11) fe[3:0]
対象とするラインの各 dw の同期ビットの値。この例では FMP:sync であることが示されている。

(12) addr[31:0]
対象とするアドレス[†1]。この例ではパケット送出の要因となった CPU のアクセス対象アドレス (VBUS:addr) のビット $[5:4]$ を 0 とした値であることが示されている。

(13) wwe[0:3]
$0 \sim 3$ 番めのデータ・フリットの送出タイミングにおける CBUS:wwe[1:0] の値。この例では word 単位の更新の可否を示す値であることが示されている。

(14) data[0:3]
$0 \sim 3$ 番めのデータ・フリットの値。この例ではパケット送出の要因となった CPU のアクセスにおける書込データ (VBUS:data) であることが示されている。

また上記の例には現れないが，記号 dwix は，パケットが対象とする dw の index, すなわち addr[5:4] を意味する。

なおアドレス・フリットの以下のビットについては L2C は入出力しないため，L2C が送出する場合には有意な値ではなく，受理する場合にはどのような値であってもよい。

- $(reserved) = [47]$
- msh $= [40]$
- su $= [37]$
- fepol $= [36]$

1.5.3 他の L2C の動作

パケットを説明する各節の「他の L2C の動作」には[†2]，その節で解説するパケットを送信元以外の L2C が受理するか否か，また受理した場合にはどのような動作を行なうかが記述される。L2C が応答する場合には，応答パケットの値が 1.5.2 節に示した形式で示される。

1.5.4 MMC/MBPL の動作

パケットを説明する各節の「MMC/MBPL の動作」には，その節で解説するパケットを MMC が受理するか否か，また受理した場合にはどのような動作を行なうかが記述される。また MMC が MBPL に割出す場合については，MBPL でどのような操作が行なわれると想定されるかも記述される。さらに，MMC/MBPL が応答する場合には，応答パケットの値が 1.5.2 節に示した形式で示される。

1.5.5 応答受領時の動作

パケットを説明する各節の「応答受領時の動作」には，その節で解説するパケットに対する応答が存在するか否か，また存在する場合には応答を受領した L2C あるいは MMC がどのような動作を行なうかが記述される。

1.5.6 想定される用法

11 章の各節の「想定される用法」では，MMC/MBPL が送出する要求パケットが，どのような状況において，またどのような目的のために用いられることが想定されているかが記述される。

[†1] [3] では addr[31:28] をそれぞれ large, ow_ch, dummy, home としているが，これらは広義のアドレスであるので，本資料では一括して addr として表記する。

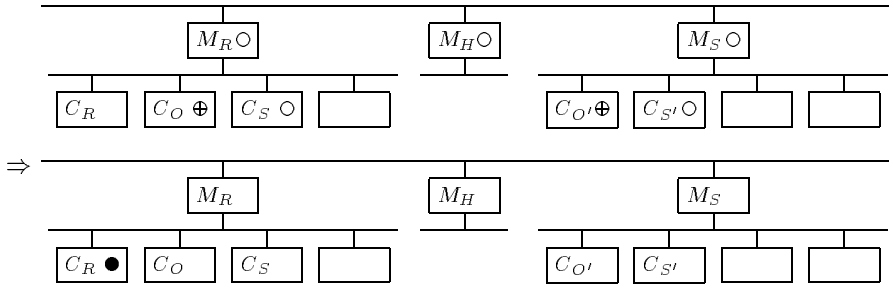
[†2] 11 章では「L2C の動作」。

1.5.7　パケットの流れと状態遷移

パケットを説明する各節の「パケットの流れと状態遷移には」，その節で解説するパケットを含むようなトランザクションについて，いくつかの典型的な例に関して以下の事項が図示される。

状態遷移

トランザクション開始時と完了時の状態を，以下の例のように図示する。



すなわち図の上半分が開始時の状態，下半分が完了時の状態であり，記号は以下の意味を持つ。

(1) M_R, M_H, M_S

トランザクションを開始したクラスタ (M_R), home-cluster (M_H), およびラインを共有しているクラスタ (M_S) の cluster-memory を意味する。なお上記の例にはないが，トランザクション開始時に M_R 以外のクラスタのみがラインを所有している場合には，記号 M_O が用いられる。

また，M_xの右に付された記号 '○' は cluster-memory のラインに有効な値が存在することを，また記号が付されていない場合には有効な値ではないことを，それぞれ意味する。

(2) C_R, C_O, C_S

トランザクションを開始した L2C (C_R), トランザクション開始時に対象ラインを所有している owner (C_O) および non-owner (C_S) であるような L2C を意味する。なお上記の例のように複数の owner や non-owner が存在する場合には，C_R とは異なるクラスタに属するものを $C_{O'}$ や $C_{S'}$ のように表記する。また上記の例にはないが，ラインを所有していない L2C を C_U として示すことがある。

また C_xの右に付された記号は，以下の意味を持つ。

- ● ... ラインの状態が dirty である。
- ⊕ ... ラインの状態が clean-owner である。
- ○ ... ラインの状態が non-owner である。

記号が付されていない場合は，ラインを所有していないことを意味する。

したがって上記の例は，以下のことを意味する。

- トランザクション開始時においてラインが globally-shared であって，C_R はラインを所有しておらず，C_O と $C_{O'}$ が owner として，また C_S と $C_{S'}$ が non-owner として，それぞれラインを所有している。
- トランザクション完了時においてラインは locally-shared になり，C_R のみが dirty-owner としてラインを所有している。

パケットの流れ

トランザクションの進行中に，どのようなパケットがクラスタ内 / クラスタ間で送受され，その結果として L2C や cluster-memory のラインの状態が変更されるか（あるいはされないか）を，以下の例のように図示する。

$$
\begin{array}{lcl}
C_R:\texttt{I****} & \xrightarrow{L:\mathit{rdinv_req}} & M_R:\texttt{VG} \\
 & & C_O:\texttt{VSGCO} \\
 & & \quad\Rightarrow\texttt{I****} \\
 & & C_S:\texttt{VSGCN} \\
 & & \quad\Rightarrow\texttt{I****}
\end{array}
\qquad
\begin{array}{lcl}
M_R:\texttt{VG} & \xrightarrow{G:\mathit{rdinv_req}} & M_H:\texttt{VG} \xrightarrow{G^m:\mathit{invalidate_req}} \\
\Rightarrow\texttt{VL} & & \Rightarrow\texttt{VXLDO}
\end{array}
$$

$$
\begin{array}{lcl}
M_S:\texttt{VG} & \xrightarrow{L:\mathit{invalidate_req}} & M_S:\texttt{VG} \\
 & & \quad\Rightarrow\texttt{I*} \\
 & & C_{O'}:\texttt{VSGCO} \\
 & & \quad\Rightarrow\texttt{I****} \\
 & & C_{S'}:\texttt{VSGCN} \\
 & & \quad\Rightarrow\texttt{I****}
\end{array}
\qquad
\begin{array}{lcl}
 & & M_H:\texttt{VG} \xrightarrow{G:\mathit{rdinv_reply}} \\
 & & \Rightarrow\texttt{I*}
\end{array}
$$

右矢印 ($\longrightarrow$) の上にはパケット名と，それがクラスタ内 (L)，あるいはクラスタ間 (G) で送受されるものであるかが示されている。パケット名に続いて (X, L, D) のような表記がある場合には，パケット送出時点での CBUS:x_msh の値と，パケットの clsh と drty の値を，以下の記法で示す。

$$
\texttt{CBUS:x_msh} = \begin{cases} 1^F \to X \\ 0^T \to S \end{cases}
$$

$$
\texttt{clsh} = \begin{cases} 0^F \to L \\ 1^T \to G \end{cases}
$$

$$
\texttt{drty} = \begin{cases} 0^F \to C \\ 1^T \to D \end{cases}
$$

また，クラスタ間のパケットがラインの全所有者に対する multicast である場合には G^m ，全所有者からの応答 (acknowledgment) の収集であれば G^c と，それぞれ表記している。なお，クラスタ間のパケットについてはあくまで想定である。

右矢印の両端には，パケットを送受する L2C あるいは cluster-memory が，1.4 節で述べたラインの状態とともに示される。複数のデバイスが関与する場合には，最上段のデバイスがパケットの送出者である[†1]。またパケットの受理 (あるいは送出) によって状態遷移が生じる場合には，新たな状態が⇒ に続いて示される。したがって上記の例は，以下のようなトランザクションの進行を意味している。

(1) C_R の状態は I**** であり，CBUS に $\mathit{rdinv_req}\,(1)$ を送出する。

(2) $\mathit{rdinv_req}\,(1)$ は M_R, C_O, C_S に受理され，C_O は VSGCO から I**** へ，C_S は VSGCN から I**** へ，それぞれ状態遷移を行なう。M_R は状態遷移を行なわず，M_H に対して $\mathit{rdinv_req}\,(2)$ を送出する。

(3) $\mathit{rdinv_req}\,(2)$ は M_H に受理される。M_H は状態遷移を行なわず，ラインの全所有者に対して $\mathit{invalidate_req}\,(3)$ を multicast する。

(4) $\mathit{invalidate_req}\,(3)$ はラインの所有者 (の一つ) である M_S に受理される。M_S は状態遷移を行なわず CBUS に $\mathit{invalidate_req}\,(4)$ を送出する。

(5) $\mathit{invalidate_req}\,(4)$ は $C_{O'}$ と $C_{S'}$ に受理され[†2]，$C_{O'}$ は VSGCO から I**** へ，$C_{S'}$ は VSGCN から I**** へ，それぞれ状態遷移を行なう。また M_S は VG から I* に状態遷移を行なって，M_H に対して $\mathit{invalidate_ack}\,(5)$ を送出する。

(6) M_H は全所有者からの $\mathit{invalidate_ack}\,(5)$ を受理すると，VG から I* に状態遷移を行なって，M_R に対して $\mathit{rdinv_reply}\,(6)$ を送出する。

(7) $\mathit{rdinv_reply}\,(6)$ は M_R に受理される。M_R は VG から VL に状態遷移を行なって，CBUS に $\mathit{rdinv_reply}\,(7)$ を送出する。このパケットの送出時点では $\texttt{CBUS:x_msh} = 1^F\ (X)$, またパケットの clsh と drty はそれぞれ $0^F\ (L)$ と $1^T\ (D)$ である。

[†1] 最上段のデバイスは左側のパケットを受理しないことがある。

[†2] この例では M_S は受理しない。

(8) $rdinv_reply(7)$ は C_R に受理される。C_R は `I****` から `VXLDO` に状態遷移を行なって，トランザクションを完了する。

1.6 パケット一覧

表 1.4 に，L2C が送受するパケットの一覧を示す。表の各欄の意味は以下の通り。

(1) mnem
パケットの mnemonic。なお cachable/non-cachable の双方が存在するものについては，cachable (C) または non-cachable (N) を末尾に付している。

(2) C → B
L2C から CBUS にパケットを送出することがあるか (∘), あるいは決して送出しないか (×) を示す。

(3) B → C
L2C が CBUS からパケットを受理することが想定されているか (∘), あるいは想定されていないか (×) を示す。受理が想定されていないパケットの `did` が，$0 \leq$ `did` ≤ 3 あるいは `did` $= 111$ (all) であるために受理を強制されると，多くの場合には CPU に不正パケット受理を割込によって通知するが，誤って受理して不正な結果を招くこともある。また，`did` $= 111$ である応答や，期待していない応答に対しては，∘ である場合でも不正パケット割込が生じる。

(4) `plen`, ..., `data`
`CBUS:wwe`, アドレス ・フリット，およびデータ ・フリットの値。記号は以下の意味を持つ。

(a) 0
L2C が送出する際には必ず 0 であり，受理する場合には 0 でなければならない[†1]

(b) 1
L2C が送出する際には必ず 1 であり，受理する場合には 1 でなければならない[†1]。

(c) ∗
L2C が送出する際には必ず有意な値であり，受理する場合には有意な値でければならない[†1]。

(d) ⊥
L2C が送出する際には有意な値ではなく，受理する場合の値は任意である。

(e) ×1, ×4
データ ・フリットが 1 または 4 フリット存在する。

(f) −
データ ・フリットが存在しない。

(5) section
パケットを解説する節番号。

なお，表に記載されていないパケットは，C→B/B→C がともに×である。特に [3] に記載されている下記のパケットが，表 1.4 には記載されていないことに注意せよ。

read_xc_req	*read_xd_req*	*read_xc_reply*	*read_xd_reply*
update_via_home_s_req	*update_s_req*	*update_q_req*	*n_c_write_req*
write_xc_req	*write_xd_req*		
write_xa_reply	*write_xb_reply*	*write_xc_reply*	*write_xd_reply*

また，表 1.5 に，パケットを解説する節の索引を示す。

[†1] 任意である場合もあるが，指定された値であれば安全である。

表 1.4: パケット一覧

mnem	C↓B	B↓C	pl en	imc msk	imc mdl	src md	dst md	size ze	ack ck	dst di	src di	dcr ty	dcl hs	fan fe	ada dr	adw we	ada ta	section
read_req(C)	○	○	00	0	1	000	0000	100	0	111	∗	⊥	⊥	⊥	∗	–	–	2.1, 3.1, 11.1
read_req(N)	○	×	00	0	0	000	0000	∗	0	100	∗	⊥	0	⊥	∗	–	–	6.1
n_c_rd_req	○	×	00	0	1	001	0000	100	0	100	∗	⊥	⊥	⊥	∗	–	–	2.4
swap_req(C)	○	○	01	0	1	011	0000	∗	0	111	∗	⊥	⊥	⊥	∗	⊥×1	∗×1	4.2, 11.7
swap_req(N)	○	×	01	0	0	011	0000	∗	0	100	∗	⊥	0	⊥	∗	⊥×1	∗×1	8.1
bq_read_req	○	○	00	0	1	100	0000	100	0	100	∗	⊥	⊥	⊥	∗	–	–	2.7
bq_nop_req	○	○	00	0	1	101	0000	100	0	100	∗	⊥	⊥	⊥	∗	–	–	2.8
read_reply(C)	○	○	10	0	1	000	0001	100	0	∗	∗	∗	∗	∗	∗	∗×4	∗×4	2.1, 11.1
read_reply(N)	×	○	01	0	0	000	0001	∗	0	∗	100	⊥	⊥	⊥	∗	⊥×1	∗×1	6.1
n_c_rd_reply	×	○	10	0	1	001	0001	100	0	∗	100	0	∗	∗	∗	∗×4	∗×4	2.4
swap_reply(C)	○	○	10	0	1	011	0001	100	0	∗	∗	∗	∗	∗	∗	∗×4	∗×4	4.2, 11.7
swap_reply(N)	×	○	01	0	0	011	0001	∗	0	∗	100	⊥	⊥	⊥	∗	⊥×1	∗×1	8.1
bq_read_reply	×	○	10	0	1	100	0001	100	0	∗	100	0	0	∗	∗	∗×4	∗×4	2.7
bq_nop_reply	×	○	00	0	1	101	0001	100	0	∗	100	⊥	⊥	∗	∗	–	–	2.7, 2.8
allread_req	○	○	00	0	1	000	0010	100	0	111	∗	⊥	⊥	⊥	∗	–	–	2.2, 3.2, 11.2
injection_req	×	○	10	∗	1	010	0010	100	⊥	111	100	0	∗	∗	∗	∗×4	∗×4	11.9
swap_m_req	○	×	01	0	0	011	0010	∗	0	100	∗	⊥	0	⊥	∗	⊥×1	∗×1	8.2
allread_reply	○	○	10	1	1	000	0011	100	0	∗	∗	∗	∗	∗	∗	∗×4	∗×4	2.2
swap_m_reply	×	○	01	0	0	011	0011	∗	0	∗	100	⊥	⊥	⊥	∗	⊥×1	∗×1	8.2
rdinv_req	○	○	00	0	1	000	0100	100	0	111	∗	⊥	⊥	⊥	∗	–	–	2.3, 3.3, 4.1, 11.3
rdinv_p_req(C)	○	×	00	0	1	100	0100	100	0	100	∗	⊥	⊥	⊥	∗	–	–	2.6
rdinv_p_req(N)	○	×	00	0	0	100	0100	∗	0	100	∗	⊥	0	⊥	∗	–	–	6.3
rdinv_q_req(C)	○	×	00	0	1	110	0100	011	0	100	∗	⊥	⊥	⊥	∗	–	–	2.5
rdinv_q_req(N)	○	×	00	0	0	110	0100	∗	0	100	∗	⊥	0	⊥	∗	–	–	6.2
rdinv_f_req	○	×	00	0	0	111	0100	∗	0	100	∗	⊥	0	⊥	∗	–	–	6.4
rdinv_reply	○	○	10	0	1	000	0101	100	0	∗	∗	∗	0	∗	∗	∗×4	∗×4	2.3, 3.5, 11.3
rdinv_p_reply(C)	×	○	10	0	1	100	0101	100	0	∗	100	1	0	∗	∗	00×4	∗×4	2.6
rdinv_p_reply(N)	×	○	01	0	0	100	0101	∗	0	∗	100	⊥	⊥	∗	∗	⊥×1	∗×1	6.3
rdinv_q_reply(C)	×	○	01	0	1	110	0101	011	0	∗	100	1	0	∗	∗	00×1	∗×1	2.5
rdinv_q_reply(N)	×	○	01	0	0	110	0101	∗	0	∗	100	⊥	⊥	∗	∗	⊥×1	∗×1	6.2
rdinv_f_reply	×	○	01	0	0	111	0101	∗	0	∗	100	⊥	⊥	∗	∗	⊥×1	∗×1	6.4
(*rdinv_reply*)	×	○	10	0	1	000	0111	100	0	∗	∗	∗	0	∗	∗	∗×4	∗×4	
update_via_home_req	○	○	10	0	1	000	1000	110	1	111	∗	⊥	∗	∗	∗	∗×4	∗×4	3.6, 11.5
write_req	○	×	01	0	0	010	1000	∗	1	100	∗	⊥	0	⊥	∗	⊥×1	∗×1	7.1
cflush_req	×	○	00	0	1	011	1000	100	⊥	111	100	⊥	⊥	⊥	∗	–	–	11.8
update_via_home_q_req	○	×	01	0	0	110	1000	∗	∗	100	∗	⊥	0	⊥	∗	⊥×1	∗×1	3.10, 7.3
update_req	×	○	10	0	1	000	1001	110	⊥	111	100	0	1	∗	∗	∗×4	∗×4	11.5
writeback_reply	○	×	10	0	1	010	1001	100	0	100	∗	∗	∗	∗	∗	∗×4	∗×4	10.1
update_direct_req	○	○	10	0	1	000	1010	110	∗	111	∗	⊥	∗	∗	∗	∗×4	∗×4	3.7, 11.5
n_c_write_m_req(C)	○	×	01	0	1	010	1010	001	1	100	∗	⊥	⊥	⊥	∗	⊥×1	∗×1	5.1
n_c_write_m_req(N)	○	×	01	0	0	010	1010	∗	1	100	∗	⊥	0	⊥	∗	⊥×1	∗×1	7.4
update_direct_s_req(C)	○	○	10	0	1	100	1010	110	∗	111	∗	⊥	∗	∗	∗	∗×4	∗×4	3.8, 11.5
update_direct_s_req(N)	○	×	01	0	0	100	1010	∗	∗	100	∗	⊥	0	∗	∗	⊥×1	∗×1	7.2
invalidate_via_home_req	○	○	00	0	1	000	1100	100	1	111	∗	⊥	∗	⊥	∗	–	–	3.5, 11.4
no_alloc_write_req	○	○	10	0	1	100	1100	110	∗	111	∗	⊥	⊥	∗	∗	∗×4	∗×4	11.6, 3.9
bq_write_req	○	×	10	0	1	101	1100	100	∗	100	∗	∗	∗	∗	∗	∗×4	∗×4	3.11, 11.4
invalidate_req	×	○	00	0	1	000	1101	100	⊥	111	100	⊥	⊥	⊥	∗	–	–	11.4
control_read_req	(○)	○	00	0	0	000	1110	∗	0	∗	∗	⊥	0	⊥	∗	–	–	11.11
demap_req	×	○	01	0	0	001	1110	⊥	⊥	∗	100	⊥	⊥	⊥	⊥	⊥×1	∗×1	11.10
control_write_req	○	○	01	0	0	010	1110	∗	1	∗	∗	⊥	0	⊥	∗	⊥×1	∗×1	9.1, 11.13
interrupt_req	×	○	01	0	0	011	1110	011	⊥	∗	100	⊥	⊥	⊥	∗	⊥×1	∗×1	11.12
control_read_reply	○	(○)	01	0	0	000	1111	∗	0	∗	∗	⊥	0	⊥	∗	⊥×1	∗×1	11.11
demap_ack	○	×	00	0	0	001	1111	⊥	0	100	∗	⊥	0	⊥	⊥	–	–	11.10
error_reply	○	×	00	0	⊥	110	1111	⊥	⊥	100	∗	⊥	⊥	⊥	∗	–	–	11.14
nop	○	×	⊥⊥	0	⊥	111	1111	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	10.2

表 1.5: パケット索引

allread_reply 2.2
allread_req 2.2, 3.2, 11.2
bq_nop_reply 2.7, 2.8
bq_nop_req 2.8
bq_read_reply 2.7
bq_read_req 2.7
bq_write_req 3.11, 11.4
cflush_req 11.8
control_read_reply 11.11
control_read_req 11.11
control_write_req 9.1, 11.13
demap_ack 11.10
demap_req 11.10
error_reply 11.14
injection_req 11.9
interrupt_req 11.12
invalidate_req 11.4
invalidate_via_home_req 3.5, 11.4
n_c_rd_reply 2.4
n_c_rd_req 2.4
n_c_write_m_req 5.1, 7.4
no_alloc_write_req 11.6, 3.9
nop 10.2
rdinv_f_reply 6.4
rdinv_f_req 6.4
rdinv_p_reply(C) 2.6
rdinv_p_reply(N) 6.3
rdinv_p_req(C) 2.6
rdinv_p_req(N) 6.3
rdinv_q_reply(C) 2.5
rdinv_q_reply(N) 6.2
rdinv_q_req(C) 2.5
rdinv_q_req(N) 6.2
rdinv_reply 2.3, 3.5, 11.3
rdinv_req 2.3, 3.3, 4.1, 11.3
read_reply(C) 2.1, 11.1
read_reply(N) 6.1
read_req(C) 2.1, 3.1, 11.1
read_req(N) 6.1
swap_m_reply 8.2
swap_m_req 8.2
swap_reply(C) 4.2, 11.7
swap_reply(N) 8.1
swap_req(C) 4.2, 11.7
swap_req(N) 8.1
update_direct_req 3.7, 11.5
update_direct_s_req(C) 3.8, 11.5
update_direct_s_req(N) 7.2
update_req 11.5
update_via_home_q_req 3.10, 7.3
update_via_home_req 3.6, 11.5
write_req 7.1
writeback_reply 10.1

2 L2-cachable load

L2-cachable load とは CPU が行なう read 要求（すなわち write でも swap でもないメモリ・アクセス要求）のうち，control-space access ではなく，かつ L2-cachable であるものをいう。したがって，load の実行や命令フェッチによる「通常」の L1C のミスは L2-cachable load となる。

L2-cachable load が許される memory-command は表 2.1 に示すものであり [2]，かつ page table entry の L1-cachability の指定によって同表の C 欄に示す以下の条件が加味される。

- yes
  page table の指定が L1-cachable でなければならず，また無条件に L2-cachable である。
- y/n
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば（またその時に限り）L2-cachable である。
- y/n(S)
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば L2-cachable である。L1-cachable でなければ，L2-cachability は `L2CCTL.l2ccbl` に従う。
- no(S)
  page table の指定は L1-cachable であってはならず，L2-cachability は `L2CCTL.l2ccbl` に従う。
- no(Q)
  page table の指定は L1-cachable であってはならないが，常に L2-cachable と解釈される。

L2-cachable load によって生成されるパケットは，表 2.1 に示すものである。一般には L2C のミスによって生じるが（同表の $H = 0^F$），いくつかの memory-command では L2C がヒットしてもパケットが生成される。

表 2.1: L2-cachable load によるパケット生成

mcmd	type	C	H	mnem	pcmd/scmd	section
0010	—	no(S)	0^F	**q-read/q-check**	*rdinv_p_req*	2.6節
0010	—	no(S)	1^T	**q-read**	*rdinv_q_req*	2.5節
0011	—	no(S)	0^F	**p-read/q-check**	*rdinv_p_req*	2.6節
0011	—	no(S)	1^T	**p-read**	*rdinv_q_req*	2.5節
0100	$\{0,1\}$	no(S)	0^F	**s-read/f-read** (normal)	*read_req*	2.1節
0100	3	y/n(S)	0^F	**s-read/f-read** (normal)	*read_req*	2.1節
0101	$\{0,1\}$	no(S)	0^F	**s-read/f-read** (normal)	*read_req*	2.1節
0101	3	y/n(S)	0^F	**s-read/f-read** (normal)	*read_req*	2.1節
0110	2	no(S)	0^F	**s-read/f-read** (all-read)	*allread_req*	2.2節
0110	3	y/n(S)	0^F	**s-read/f-read** (normal)	*read_req*	2.1節
0111	$\{0,1\}$	y/n	0^F	**normal-read** (all-read)	*allread_req*	2.2節
0111	2	no(S)	0^F	**s-read/f-read** (all-read)	*allread_req*	2.2節
0111	3	y/n(S)	0^F	**s-read/f-read** (normal)	*read_req*	2.1節
1000	—	yes	*1	**bq-read**	*bq_nop_req*	2.8節
1000	—	yes	*1	**bq-read**	*bq_read_req*	2.7節
1001	$\{0,1,2\}$	y/n	0^F	**normal-read** (normal)	*read_req*	2.1節
1001	3	no(Q)	*1	**bq-check**	*bq_nop_req*	2.8節
1001	3	no(Q)	*1	**bq-check**	*bq_read_req*	2.7節
1010	—	y/n	0^F	**normal-read** (normal)	*read_req*	2.1節
1010	—	y/n	0^F	**normal-read** (read-invalidate)	*rdinv_req*	2.3節
1100	—	y/n	0^F	**normal-read** (no-coherent)	*n_c_rd_req*	2.4節
1101	—	y/n	0^F	**normal-read** (read-invalidate)	*rdinv_req*	2.3節
1110	—	y/n	0^F	**normal-read** (normal)	*read_req*	2.1節
1111	—	y/n	0^F	**normal-read** (normal)	*read_req*	2.1節

*1 Hに関わらずパケットが生成される。パケットが CBUS に送出される以前に対象のラインが CBUS からの要求によって無効化されるという例外的事象 (block-queue の読出権移動) が生じると，*bq_nop_req* が出力される。それ以外の場合には *bq_read_req* が出力される。

2.1 *read_req* ← **normal-read, s-read, f-read** (coherence-policy(r) = normal)

read_req は，読出の coherence-policy が normal であるような **normal-read**, **s-read** または **f-read** が cache-miss したときの block-load 要求である。

送出される要求パケット

`plen`	= 00	= 1 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0000/000	= *read_req*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= 111	= all
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= ⊥	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`

他の L2C の動作

対象ラインを owner として所有している L2C があれば (S = `V***O`)，その L2C が `CBUS:x_crply` = 0^T として応答意志を示し，FMP/DAR を読出して以下の *read_reply* を応答し，S = `VS*CN` に遷移する。なお共有状態が globally-shared であっても owner であれば応答する。他のキャッシュは何もしない。

`plen`	= 10	= 5 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0001/000	= *read_reply*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= *did*	= requester's ID
`sid`	= *pid*	= own ID
`drty`	= *dirty*	= `FMP:state.d`
`clsh`	= *global*	= ¬`FMP:state.l`
`fe`	= *sync*	= `FMP:sync`
`addr`	= *addr*	= requested address
`wwe[0:3]`	= {*w-wait*, 0}∗4	= {`FMP:ww`, 0}∗4
`data[0:3]`	= *data*	= `DAR[dwix..dwix+3]`

MMC/MBPL の動作

対象ラインを owner として所有している L2C が存在しなければ (`CBUS:x_crply` = 1^F), MMC は対象ラインの STAG をアクセスする。その結果 `STAG:{rd_trap|valid}` = $\{0^F 1^T\}$ であれば，cluster-memory を読出して以下の *read_reply* を応答する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/000	= *read_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 0	= clean
clsh	= *global*	= STAG.global
fe	= *sync*	= STAG.fe
addr	= *addr*	= requested address
wwe[0:3]	= *w-wait/r-wait*	= DTAG[dwix..dwix+3]
data[0:3]	= *data*	= DRAM[dwix..dwix+3]

直接応答が不可能な場合には MBPL によって処理され，home-cluster にライン取得を要求するなどして，最終的には上記のパケットが応答される。

[注意]

MBPL が globally-shared なラインを取得して応答する場合，必ずラインの内容を cluster-memory に書込んだ上で，$\texttt{clsh} = 1^T$, $\texttt{drty} = 0^F$ として応答しなければならない。

応答受領時の動作

応答が送出された際の $\texttt{CBUS:x_msh} \equiv x$ と，応答パケットの $\texttt{clsh} \equiv \texttt{g}$ と $\texttt{drty} \equiv \texttt{d}$ に従って $S = \texttt{VxgdO}$ に遷移する。ただし ；

$$\texttt{x} = \begin{cases} \texttt{X} & x = 1^F \wedge \texttt{g} = 0^F \\ \texttt{S} & \text{otherwise} \end{cases}$$

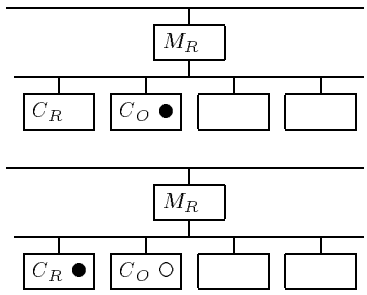
である。したがって，L2C が応答した場合には応答者自身が $\texttt{CBUS:x_msh} = 0^T$ とするので必ず shared となり，MMC/MBPL が応答した場合には他のクラスタと L2C の共有状況によって shared または exclusive となる。

また memory-command, *read_reply* の $f \equiv \texttt{fe}[\texttt{dwix}]$, および L1-cachability c に基づき，以下の値を CPU に返す。

mem. cmd.	f	c	return value
normal-read	—	—	data
s-read	1^T	—	data
	0^F	1^T	0
		0^F	undefined-error
f-read	1^T	—	-1
	0^F	—	0

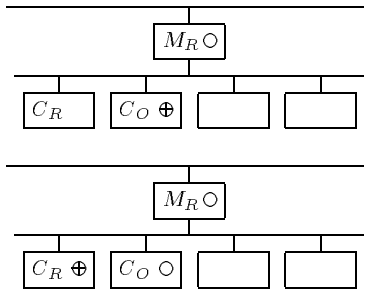
パケットの流れと状態遷移

(1)



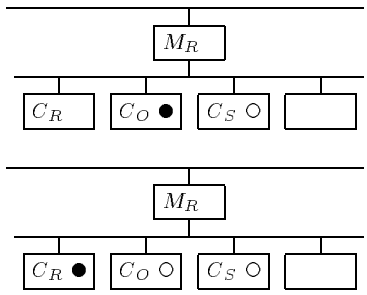
C_R :I**** $\xrightarrow{L : read_req}$ C_O :VXLDO $\xrightarrow{L : read_reply(S,L,D)}$ C_R :I****
⇒ VSLCN ⇒ VSLDO

(2)



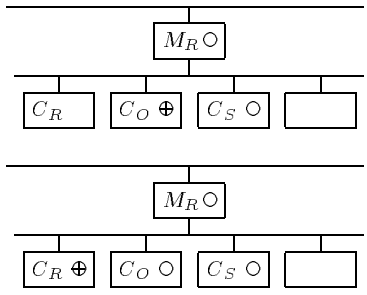
C_R :I**** $\xrightarrow{L : read_req}$ C_O :VXLCO $\xrightarrow{L : read_reply(S,L,C)}$ C_R :I****
⇒ VSLCN ⇒ VSLCO

(3)



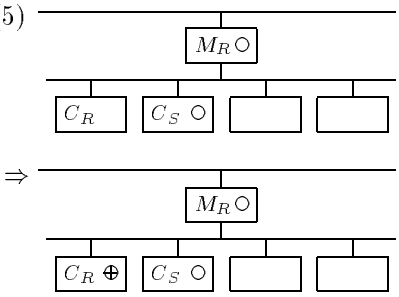
C_R :I**** $\xrightarrow{L : read_req}$ C_O :VSLDO $\xrightarrow{L : read_reply(S,L,D)}$ C_R :I****
⇒ VSLCN ⇒ VSLDO

(4)



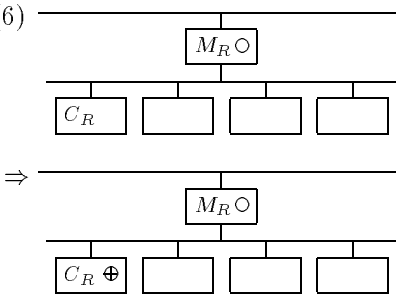
C_R :I**** $\xrightarrow{L : read_req}$ C_O :VSLCO $\xrightarrow{L : read_reply(S,L,C)}$ C_R :I****
⇒ VSLCN ⇒ VSLCO

(5)



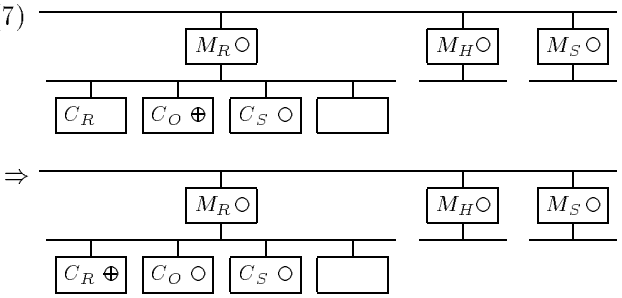
C_R :I**** $\xrightarrow{L : read_req}$ M_R:VL $\xrightarrow{L : read_reply(S, L, C)}$ C_R :I****
⇒ VSLCO

(6)



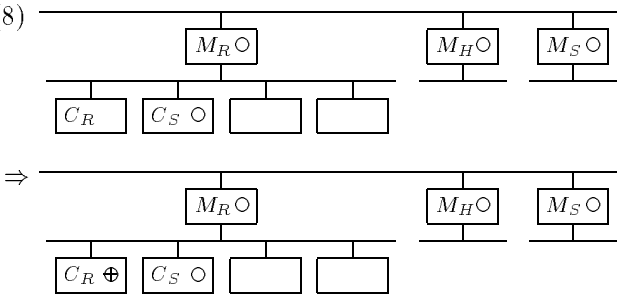
C_R :I**** $\xrightarrow{L : read_req}$ M_R:VL $\xrightarrow{L : read_reply(X, L, C)}$ C_R :I****
⇒ VXLCO

(7)



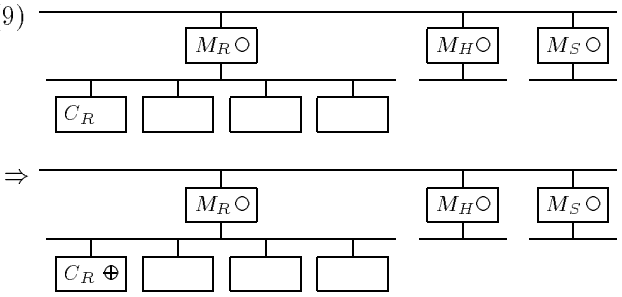
C_R :I**** $\xrightarrow{L : read_req}$ C_O :VSGCO $\xrightarrow{L : read_reply(S, G, C)}$ C_R :I****
⇒ VSGCN ⇒ VSGCO

(8)



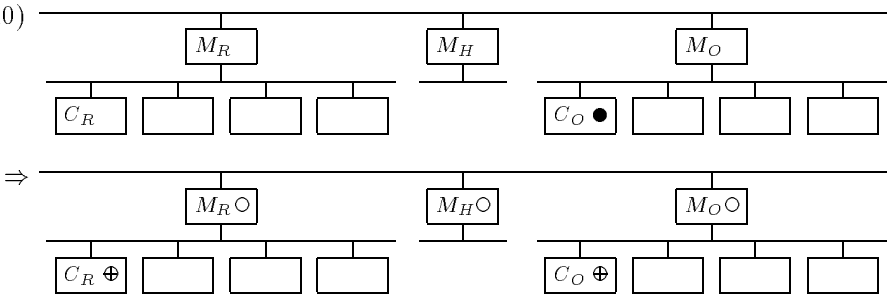
C_R :I**** $\xrightarrow{L : read_req}$ M_R:VG $\xrightarrow{L : read_reply(S, G, C)}$ C_R :I****
⇒ VSGCO

(9)



C_R :I**** $\xrightarrow{L : read_req}$ M_R:VG $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VSGCO

(10)

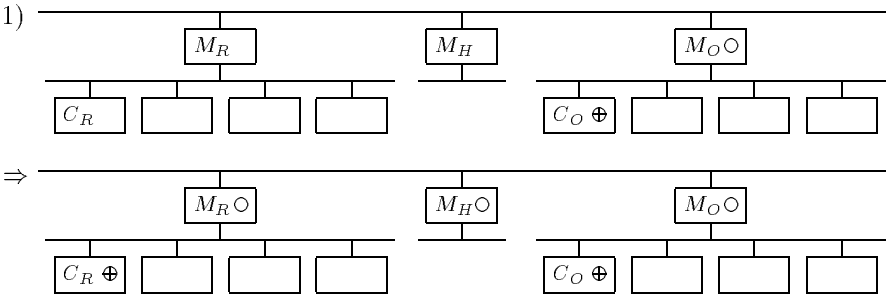


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

C_O :VXLDO $\xrightarrow{L : read_reply(X, L, D)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(11)

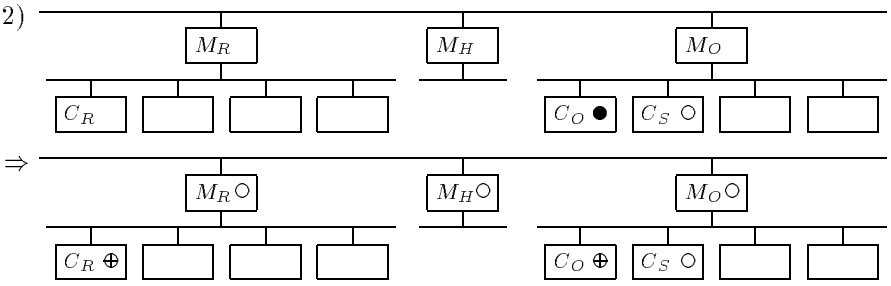


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

C_O :VXLCO $\xrightarrow{L : read_reply(X, L, C)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(12)



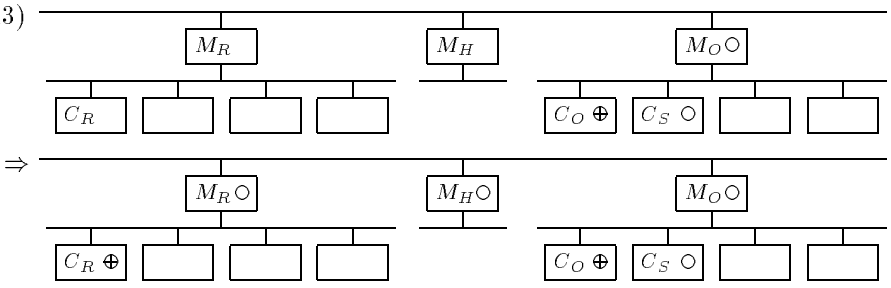
C_R : I**** $\xrightarrow{L : read_req}$ M_R : I* $\xrightarrow{G : read_req}$ M_H : I* $\xrightarrow{G : read_req}$ M_O : VL $\xrightarrow{L : read_req}$

C_O : VSLDO $\xrightarrow{L : read_reply(S, L, D)}$ M_O : VL $\xrightarrow{G : read_reply}$ M_H : I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG

C_S : VSLCN
⇒ VSGCN

M_R : I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R : I****
⇒ VG ⇒ VSGCO

(13)



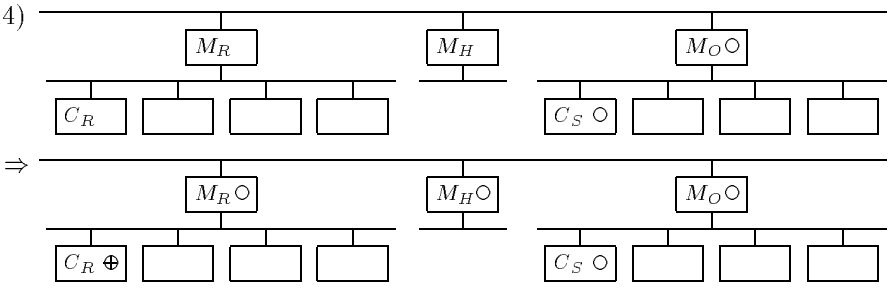
C_R : I**** $\xrightarrow{L : read_req}$ M_R : I* $\xrightarrow{G : read_req}$ M_H : I* $\xrightarrow{G : read_req}$ M_O : VL $\xrightarrow{L : read_req}$

C_O : VSLCO $\xrightarrow{L : read_reply(S, L, C)}$ M_O : VL $\xrightarrow{G : read_reply}$ M_H : I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG

C_S : VSLCN
⇒ VSGCN

M_R : I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R : I****
⇒ VG ⇒ VSGCO

(14)

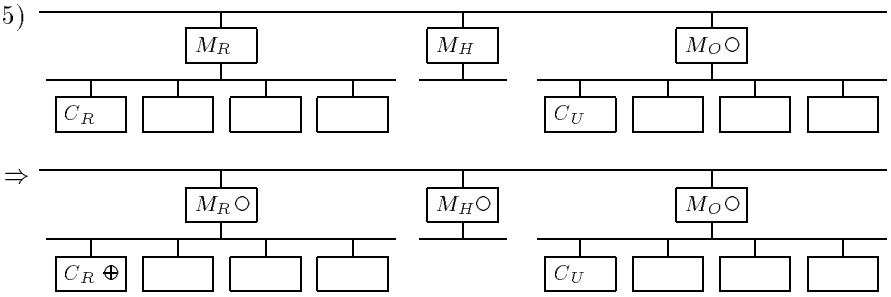


C_R : I**** $\xrightarrow{L : read_req}$ M_R : I* $\xrightarrow{G : read_req}$ M_H : I* $\xrightarrow{G : read_req}$ M_O : VL $\xrightarrow{L : read_req}$

M_O : VL $\xrightarrow{L : read_reply(S, L, C)}$ M_O : VL $\xrightarrow{G : read_reply}$ M_H : I* $\xrightarrow{G : read_reply}$
C_S : VSLCN ⇒ VG ⇒ VG
⇒ VSGCN

M_R : I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R : I****
⇒ VG ⇒ VSGCO

(15)



⇒

C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

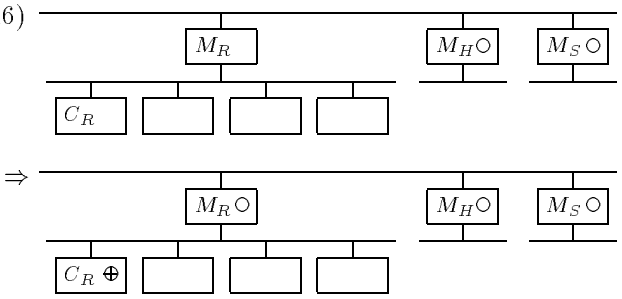
M_O:VL $\xrightarrow{L : read_reply(X, L, C)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$

⇒ VG ⇒ VG

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****

⇒ VG ⇒ VSGCO

(16)



⇒

C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:VG $\xrightarrow{G : read_reply}$

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****

⇒ VG ⇒ VSGCO

2.2 *allread_req* ← **normal-read, s-read, f-read** (coherence-policy(r) = all-read)

allread_req は，読出の coherence-policy が all-read であるような **normal-read**, **s-read** または **f-read** が cache-miss したときの，injection を伴う block-load 要求である。

送出される要求パケット

`plen`	= 00	= 1 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0010/000	= *allread_req*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= 111	= all
`sid`	= *pid*	= own ID
`drty`	= $\perp$	
`clsh`	= $\perp$	
`fe`	= $\perp$	
`addr`	= *addr*	= `VBUS.addr`

他の L2C の動作

対象ラインを owner として所有している L2C があれば (S = `V***O`)，その L2C が `CBUS:x_crply` = 0^T として応答意志を示し，FMP/DAR を読出して以下の *allread_reply* を応答し，S = `VS*CN` に遷移する。なお共有状態が globally-shared であっても owner であれば応答する。

`plen`	= 10	= 5 flit
`injmask`	= 1	= injection
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0011/000	= *allread_reply*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= *did*	= requester's ID
`sid`	= *pid*	= own ID
`drty`	= *dirty*	= `FMP:state.d`
`clsh`	= *global*	= ¬`FMP:state.l`
`fe`	= *sync*	= `FMP:sync`
`addr`	= *addr*	= requested address
`wwe[0:3]`	= {*w-wait*, 0}∗4	= {`FMP:ww`, 0}∗4
`data[0:3]`	= *data*	= `DAR[dwix..dwix+3]`

他のキャッシュは要求パケットに対しては何もしないが，応答に対しては反応する。すなわち *allread_reply* は injection 要求と解釈され，injection 操作が行なわれて S = `VSgCN` となる。ただし，対象ラインに対する要求が何らかの理由で受理できない場合には，単に *allread_reply* を無視する。また対象ラインの状態が内部レジスタ `INJLEV` が定める条件に合致しなければ，結果的に無視される (1.9 節，10.2 節参照)。

MMC/MBPL の動作

対象ラインを owner として所有している L2C が存在しなければ (`CBUS:x_crply` = 1^F), MMC は対象ラインの STAG をアクセスする。その結果 `STAG:{rd_trap|valid}` = $\{0^F 1^T\}$ であれば，cluster-memory を読出して以下の *allread_reply* を応答する。

`plen`	$= 10$	$= 5$ flit
`injmask`	$= 1$	= injection
`ccbl`	$= 1$	= cachable
`pcmd/scmd`	$= 0011/000$	$= \mathit{allread_reply}$
`size`	$= 100$	= block
`ack`	$= 0$	= nop
`did`	$= \mathit{did}$	= requester's ID
`sid`	$= 100$	= MMC/MBPL
`drty`	$= 0$	= clean
`clsh`	$= \mathit{global}$	= `STAG.global`
`fe`	$= \mathit{sync}$	= `STAG.fe`
`addr`	$= \mathit{addr}$	= requested address
`wwe[0:3]`	$= \mathit{w\text{-}wait/r\text{-}wait}$	= `DTAG[dwix..dwix+3]`
`data[0:3]`	$= \mathit{data}$	= `DRAM[dwix..dwix+3]`

直接応答が不可能な場合には MBPL によって処理され，home-cluster にライン取得を要求するなどして，最終的には上記のパケットが応答される。なおライン取得の過程で他のクラスタに対する injection がなされるか否かは，MBPL のプログラムによって定まる。

[注意]

(1) MBPL が globally-shared なラインを取得して応答する場合，必ずラインの内容を cluster-memory に書込んだ上で，`clsh` $= 1^T$, `drty` $= 0^F$ として応答しなければならない。

(2) 他のクラスタから all-read によるライン取得を要求された MBPL (すなわちクラスタ間では共有されていないラインを所有するクラスタの MBPL) は，ラインを取得するために CBUS に対して要求を送出するが，その際の要求は *allread_req* であってはならず，*read_req* でなければならない。この制約は L2C が応答する *allread_reply* の `clsh` が，*allread_req* を受理する以前のクラスタ間共有状態を反映していることに起因する。すなわち応答を送出する L2C は *allread_req* を受理した結果 $S =$ `VSGCN` に遷移するが，*allread_reply* の `clsh` は (通常) 0^F である。したがって他の L2C は locally-shared であると誤解して injection を行ない，クラスタ間共有に関する状態の矛盾が生じる。したがって，一般に MBPL は *allread_req* を CBUS に送出してはならない。

なお，上記の MBPL がクラスタ中の全ての L2C に対する injection を行なうようにするためには，*read_reply* を受理後に *injection_req* を CBUS に送出すればよい。

応答受領時の動作

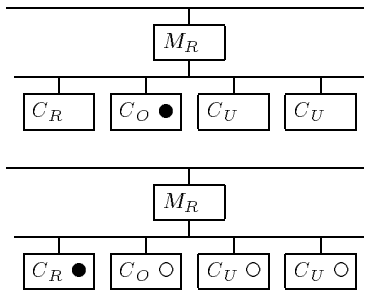
応答パケットの `clsh` $\equiv$ `g` と `drty` $\equiv$ `d` に従って $S =$ `VSgdO` に遷移する。すなわち応答送出時点での他の L2C の共有状態に関わらず，injection がなされるものと仮定して shared となる。

また memory-command と *allread_reply* の $f \equiv$ `fe`[`dwix`] に基づき，以下の値を CPU に返す。

mem. cmd.	f	return value
normal-read	—	`data`
s-read	1^T	`data`
	0^F	undefined-error
f-read	1^T	-1
	0^F	0

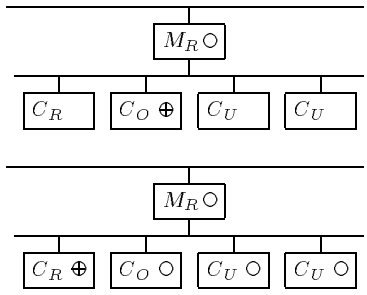
パケットの流れと状態遷移

(1)



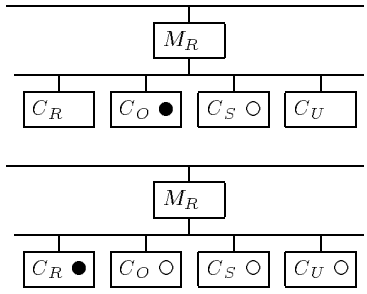
C_R :I**** $\xrightarrow{L : allread_req}$ C_O :VXLDO $\xrightarrow{L : allread_reply(S,L,D)}$ C_R :I****
⇒ VSLCN
⇒ VSLDO
C_U :I****
⇒ VSLCN

(2)



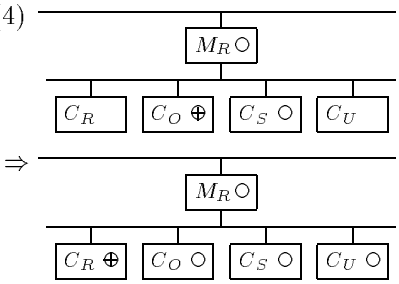
C_R :I**** $\xrightarrow{L : allread_req}$ C_O :VXLCO $\xrightarrow{L : allread_reply(S,L,C)}$ C_R :I****
⇒ VSLCN
⇒ VSLCO
C_U :I****
⇒ VSLCN

(3)



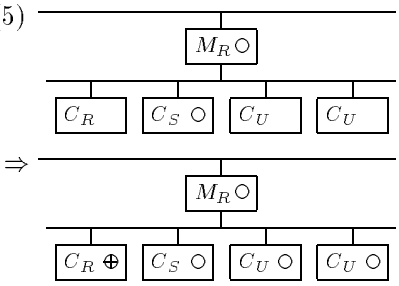
C_R :I**** $\xrightarrow{L : allread_req}$ C_O :VSLDO $\xrightarrow{L : allread_reply(S,L,D)}$ C_R :I****
⇒ VSLCN
⇒ VSLDO
C_U :I****
⇒ VSLCN

(4)



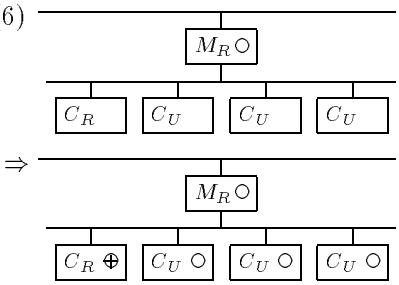
C_R :I**** $\xrightarrow{L : allread_req}$ C_O :VSLCO $\xrightarrow{L : allread_reply(S,L,C)}$ C_R :I****
⇒ VSLCN ⇒ VSLCO
C_U :I****
⇒ VSLCN

(5)



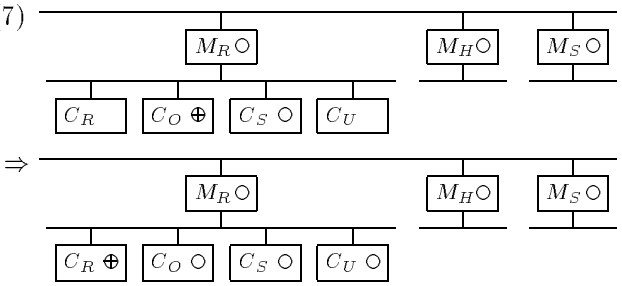
C_R :I**** $\xrightarrow{L : allread_req}$ M_R :VL $\xrightarrow{L : allread_reply(S,L,C)}$ C_R :I****
⇒ VSLCO
C_U :I****
⇒ VSLCN

(6)



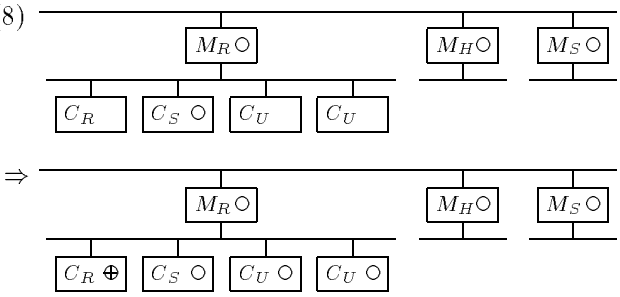
C_R :I**** $\xrightarrow{L : allread_req}$ M_R :VL $\xrightarrow{L : allread_reply(X,L,C)}$ C_R :I****
⇒ VSLCO
C_U :I****
⇒ VSLCN

(7)



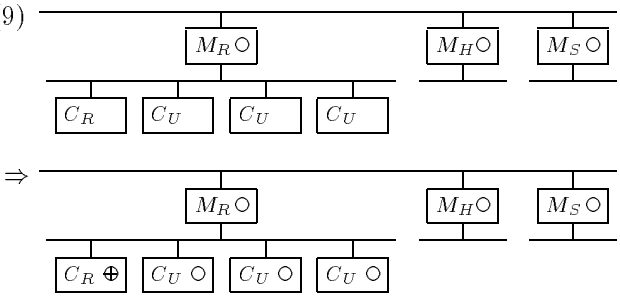
C_R :I**** $\xrightarrow{L : allread_req}$ C_O :VSGCO $\xrightarrow{L : allread_reply(S,G,C)}$ C_R :I****
⇒ VSGCN ⇒ VSGCO
C_U :I****
⇒ VSGCN

(8)



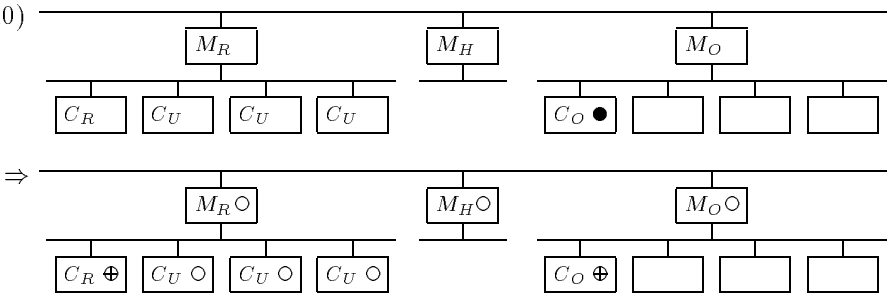
C_R :I**** $\xrightarrow{L : allread_req}$ M_R:VG $\xrightarrow{L : allread_reply(S, G, C)}$ C_R :I****
⇒VSGCO
C_U :I****
⇒VSGCN

(9)



C_R :I**** $\xrightarrow{L : allread_req}$ M_R:VG $\xrightarrow{L : allread_reply(X, G, C)}$ C_R :I****
⇒VSGCO
C_U :I****
⇒VSGCN

(10)

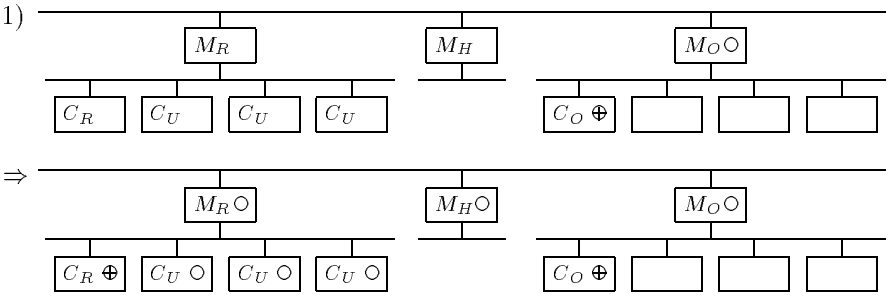


C_R :I**** $\xrightarrow{L : allread_req}$ M_R:I* $\xrightarrow{G : allread_req}$ M_H:I* $\xrightarrow{G : allread_req}$

M_O:VL $\xrightarrow{L : read_req}$ C_O :VXLDO
⇒VSGCO
$\xrightarrow{L : read_reply(X, L, D)}$ M_O:VL
⇒VG
$\xrightarrow{G : allread_reply}$

M_H:I*
⇒VG
$\xrightarrow{G : allread_reply}$ M_R:I*
⇒VG
$\xrightarrow{L : allread_reply(X, G, C)}$ C_R :I****
⇒VSGCO
C_U :I****
⇒VSGCN

(11)

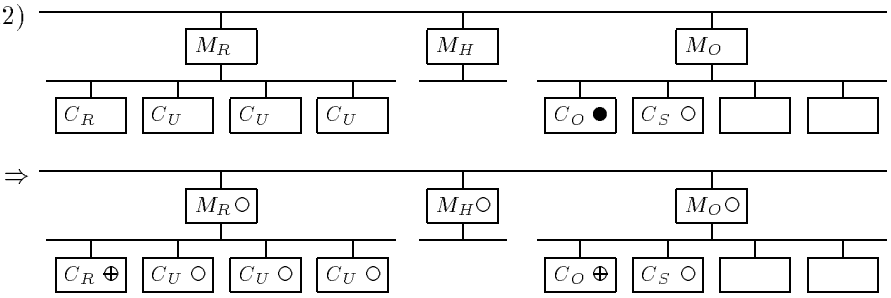


C_R : I**** $\xrightarrow{L : allread_req}$ M_R : I* $\xrightarrow{G : allread_req}$ M_H : I* $\xrightarrow{G : allread_req}$

M_O : VL $\xrightarrow{L : read_req}$ C_O : VXLCO $\xrightarrow{L : read_reply(X, L, C)}$ M_O : VL $\xrightarrow{G : allread_reply}$
⇒ VSGCO ⇒ VG

M_H : I* $\xrightarrow{G : allread_reply}$ M_R : I* $\xrightarrow{L : allread_reply(X, G, C)}$ C_R : I****
⇒ VG ⇒ VG ⇒ VSGCO
C_U : I****
⇒ VSGCN

(12)

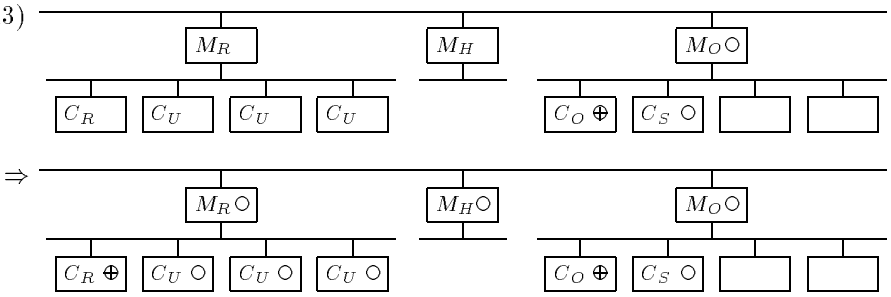


C_R : I**** $\xrightarrow{L : allread_req}$ M_R : I* $\xrightarrow{G : allread_req}$ M_H : I* $\xrightarrow{G : allread_req}$

M_O : VL $\xrightarrow{L : read_req}$ C_O : VSLDO $\xrightarrow{L : read_reply(S, L, D)}$ M_O : VL $\xrightarrow{G : allread_reply}$
⇒ VSGCO ⇒ VG
C_S : VSLCN
⇒ VSGCN

M_H : I* $\xrightarrow{G : allread_reply}$ M_R : I* $\xrightarrow{L : allread_reply(X, G, C)}$ C_R : I****
⇒ VG ⇒ VG ⇒ VSGCO
C_U : I****
⇒ VSGCN

(13)

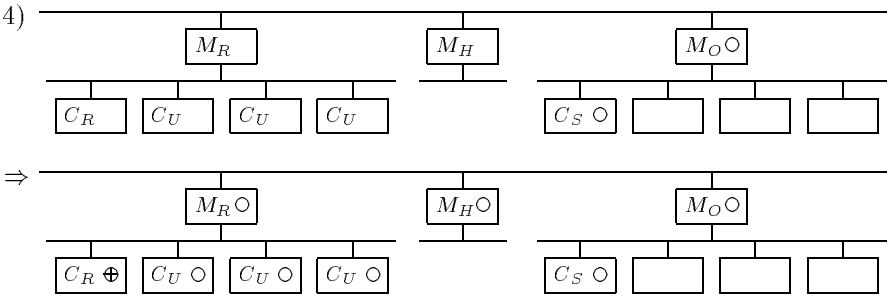


C_R : I**** $\xrightarrow{L : allread_req}$ M_R : I* $\xrightarrow{G : allread_req}$ M_H : I* $\xrightarrow{G : allread_req}$

M_O : VL $\xrightarrow{L : read_req}$ C_O : VSLCO $\xrightarrow{L : read_reply(S, L, C)}$ M_O : VL $\xrightarrow{G : allread_reply}$
$\Rightarrow$ VSGCO $\Rightarrow$ VG
C_S : VSLCN
$\Rightarrow$ VSGCN

M_H : I* $\xrightarrow{G : allread_reply}$ M_R : I* $\xrightarrow{L : allread_reply(X, G, C)}$ C_R : I****
$\Rightarrow$ VG $\Rightarrow$ VG $\Rightarrow$ VSGCO
C_U : I****
$\Rightarrow$ VSGCN

(14)

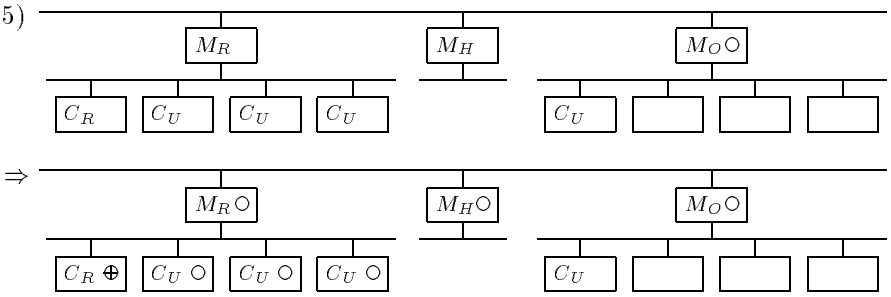


C_R : I**** $\xrightarrow{L : allread_req}$ M_R : I* $\xrightarrow{G : allread_req}$ M_H : I* $\xrightarrow{G : allread_req}$

M_O : VL $\xrightarrow{L : read_req}$ M_O : VL $\xrightarrow{L : read_reply(S, L, C)}$ M_O : VL $\xrightarrow{G : allread_reply}$
C_S : VSLCN $\Rightarrow$ VG
$\Rightarrow$ VSGCN

M_H : I* $\xrightarrow{G : allread_reply}$ M_R : I* $\xrightarrow{L : allread_reply(X, G, C)}$ C_R : I****
$\Rightarrow$ VG $\Rightarrow$ VG $\Rightarrow$ VSGCO
C_U : I****
$\Rightarrow$ VSGCN

(15)

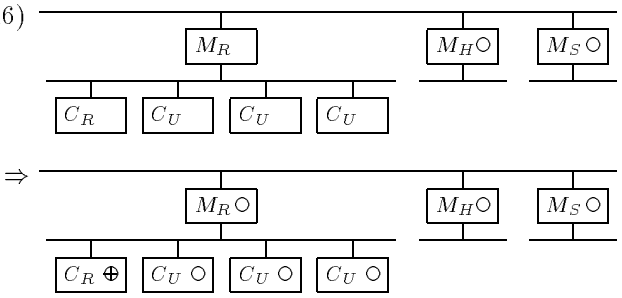


C_R :I**** $\xrightarrow{L : allread_req}$ M_R:I* $\xrightarrow{G : allread_req}$ M_H:I* $\xrightarrow{G : allread_req}$

M_O:VL $\xrightarrow{L : read_req}$ M_O:VL $\xrightarrow{L : read_reply(X, L, C)}$ M_O:VL ⇒VG $\xrightarrow{G : allread_reply}$

M_H:I* ⇒VG $\xrightarrow{G : allread_reply}$ M_R:I* ⇒VG $\xrightarrow{L : allread_reply(X, G, C)}$ C_R :I**** ⇒VSGCO, C_U :I**** ⇒VSGCN

(16)



C_R :I**** $\xrightarrow{L : allread_req}$ M_R:I* $\xrightarrow{G : allread_req}$ M_H:VG $\xrightarrow{G : allread_reply}$

M_R:I* ⇒VG $\xrightarrow{L : allread_reply(X, G, C)}$ C_R :I**** ⇒VSGCO, C_U :I**** ⇒VSGCN

2.3 *rdinv_req* ← **normal-read** (coherence-policy(r) = read-invalidate)

rdinv_req は，読出の coherence-policy が read-invalidate であるような **normal-read** が cache-miss したときの，無効化を伴う block-load 要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0100/000	= *rdinv_req*
size	= 100	= block
ack	= 0	= nop
did	= 111	= all
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= VBUS.addr

他の L2C の動作

対象ラインを owner として所有している L2C があり，かつ共有状態が globally-shared でなければ (S = V*L*O)，その L2C が CBUS:x_crply = 0^T として応答意志を示し，FMP/DAR を読出して以下の *rdinv_reply* を応答し，対象ラインを無効化する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/000	= *rdinv_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= *pid*	= own ID
drty	= *dirty*	= FMP:state.d
clsh	= 0	= ¬(FMP : state.l = 1)
fe	= *sync*	= FMP:sync
addr	= *addr*	= requested address
wwe[0:3]	= {*w-wait*, 0}*4	= {FMP : ww, 0}*4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

対象ラインを所有しているが上記の条件を満たさない L2C は，*rdinv_req* を無効化要求と解釈し，対象ラインの無効化を行なう。なおこの操作は共有状態が globally-shared であっても行なわれるので，要求元クラスタに対する無効化は省略することができる (実施しても構わない)。

MMC/MBPL の動作

対象ラインが globally-shared であるか owner として所有している L2C が存在しなければ (CBUS:x_crply = 1^F), MMC は対象ラインの STAG をアクセスする。その結果 STAG:{rd_trap|valid|global} = $\{0^F 1^T 0^F\}$ であれば，cluster-memory を読出して以下の *rdinv_reply* を応答する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/000	= *rdinv_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 0	= clean
clsh	= 0	= STAG.global = 0
fe	= *sync*	= STAG.fe
addr	= *addr*	= requested address
wwe[0:3]	= *w-wait/r-wait*	= DTAG[dwix..dwix+3]
data[0:3]	= *data*	= DRAM[dwix..dwix+3]

直接応答が不可能な場合には MBPL によって処理され，home-cluster にライン取得と他のクラスタに対する無効化を要求するなどして，最終的には上記のパケットが応答される。ただし drty = 1^T とするのが望ましい。

[注意]

(1) globally-shared の場合，STAG:valid = 1^T であっても無効化を home-cluster で serialize する必要があり，かつ L2C には serialize された時点でのラインの値を返す必要がある。したがって，MBPL は globally-shared の場合には常に home-cluster に対して *rdinv_req* を送出してラインの値を得る必要がある。

(2) MBPL が globally-shared なラインを取得して応答する場合，必ず clsh = 0^F としなければならない。また drty は $0^F/1^T$ のいずれでもよいが，drty = 0^F とする場合には cluster-memory にも取得したラインの内容を書込む必要がある。*rdinv_req* は coherence-policy が invalidate であるような store が cache-miss した際にも送出される（3.3 節参照）。この store による *rdinv_req* の方が一般的であるが，その場合には取得したラインを L2C が更新するので cluster-memory への書込は無意味である。MBPL は L2C からの *rdinv_req* が load/store のいずれによるものかを判別できないので，常に cluster-memory の書込を行なわず drty = 1^T として応答するのが望ましい。

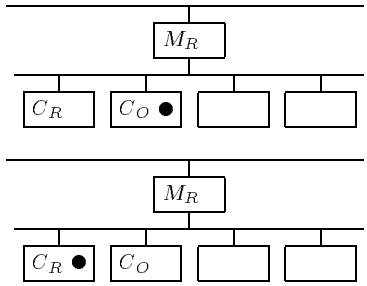
応答受領時の動作

read_reply と同様の機構によって S = VxgdO に遷移するが，必ず clsh $\equiv$ g = 0^F であり，また応答送出時には他の L2C がラインを所有していることはないため，結果的に S = VXLdO となる。

また常に data を CPU に返す。

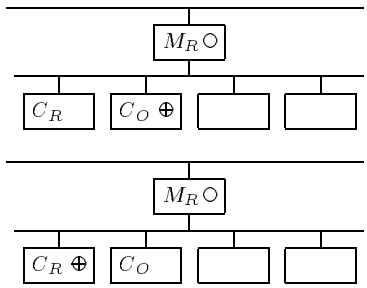
パケットの流れと状態遷移

(1)



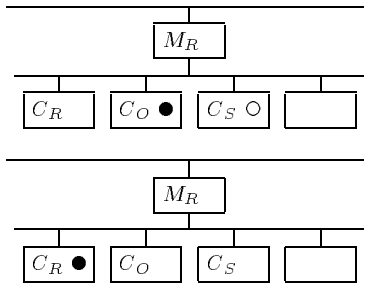
C_R :I**** $\xrightarrow{L : rdinv_req}$ C_O :VXLDO $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :I****
⇒ I**** ⇒ VXLDO

(2)



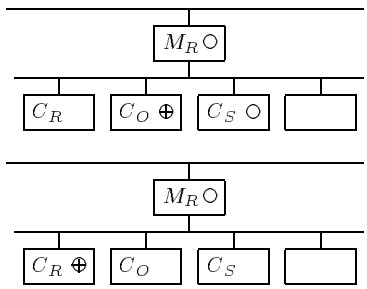
C_R :I**** $\xrightarrow{L : rdinv_req}$ C_O :VXLCO $\xrightarrow{L : rdinv_reply(X, L, C)}$ C_R :I****
⇒ I**** ⇒ VXLCO

(3)



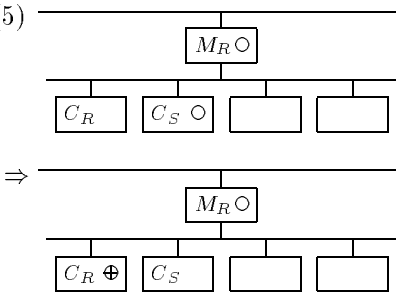
C_R :I**** $\xrightarrow{L : rdinv_req}$ C_O :VSLDO $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :I****
⇒ I**** ⇒ VXLDO
C_S :VSLCN
⇒ I****

(4)



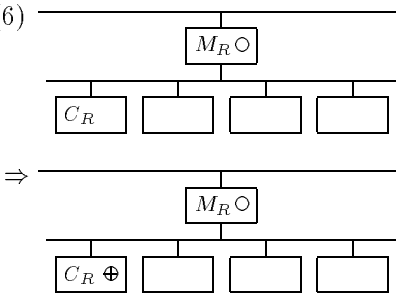
C_R :I**** $\xrightarrow{L : rdinv_req}$ C_O :VSLCO $\xrightarrow{L : rdinv_reply(X, L, C)}$ C_R :I****
⇒ I**** ⇒ VXLCO
C_S :VSLCN
⇒ I****

(5)



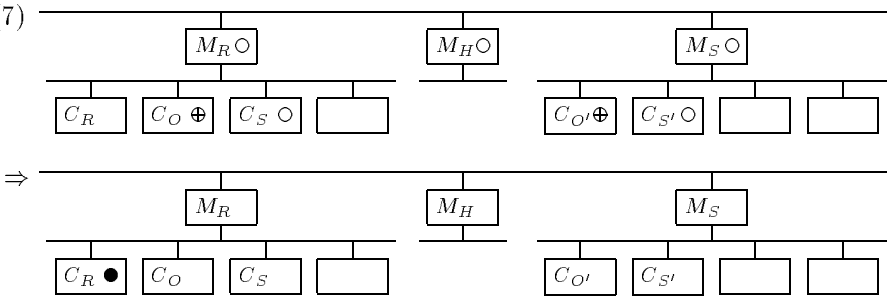
C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : VL $\xrightarrow{L : rdinv_reply(X, L, C)}$ C_R : I****
C_S : VSLCN ⇒ VXLCO
⇒ I****

(6)



C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : VL $\xrightarrow{L : rdinv_reply(X, L, C)}$ C_R : I****
⇒ VXLCO

(7)

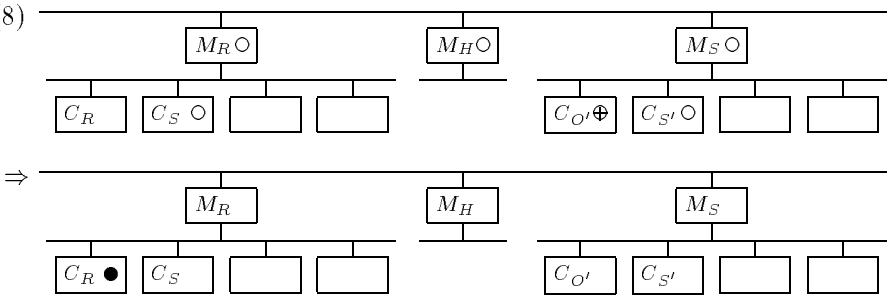


C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : VG $\xrightarrow{G : rdinv_req}$ M_H : VG $\xrightarrow{G^m : invalidate_req}$
C_O : VSGCO ⇒ I****
C_S : VSGCN ⇒ I****

M_S : VG $\xrightarrow{L : invalidate_req}$ M_S : VG ⇒ I* $\xrightarrow{G^c : invalidate_ack}$ M_H : VG ⇒ I* $\xrightarrow{G : rdinv_reply}$
$C_{O'}$: VSGCO ⇒ I****
$C_{S'}$: VSGCN ⇒ I****

M_R : VG ⇒ VL $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : I**** ⇒ VXLDO

(8)



C_R : I**** $\xrightarrow{L\,:\,rdinv_req}$ M_R : VG $\xrightarrow{G\,:\,rdinv_req}$ M_H : VG $\xrightarrow{G^m\,:\,invalidate_req}$

C_S : VSGCN ⇒ I****

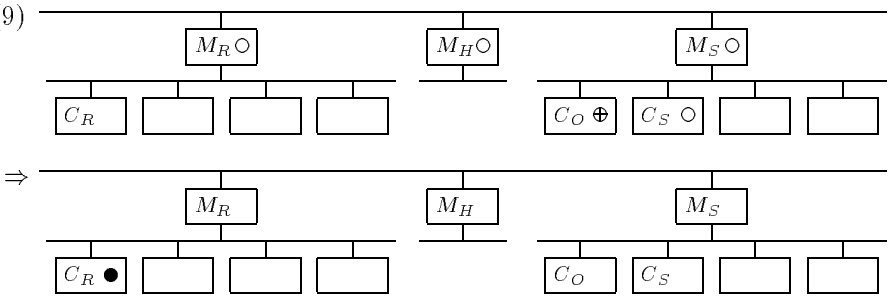
M_S : VG $\xrightarrow{L\,:\,invalidate_req}$ M_S : VG ⇒ I* $\xrightarrow{G^c\,:\,invalidate_ack}$ M_H : VG ⇒ I* $\xrightarrow{G\,:\,rdinv_reply}$

$C_{O'}$: VSGCO ⇒ I****

$C_{S'}$: VSGCN ⇒ I****

M_R : VG ⇒ VL $\xrightarrow{L\,:\,rdinv_reply(X,L,D)}$ C_R : I**** ⇒ VXLDO

(9)



C_R : I**** $\xrightarrow{L\,:\,rdinv_req}$ M_R : VG $\xrightarrow{G\,:\,rdinv_req}$ M_H : VG $\xrightarrow{G^m\,:\,invalidate_req}$

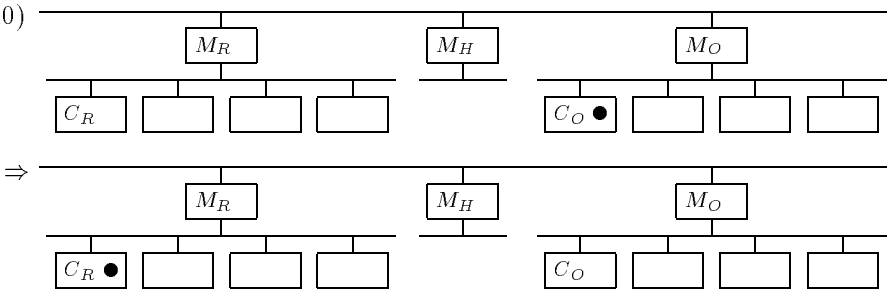
C_O : VSGCO ⇒ I****

C_S : VSGCN ⇒ I****

M_S : VG $\xrightarrow{L\,:\,invalidate_req}$ M_S : VG ⇒ I* $\xrightarrow{G^c\,:\,invalidate_ack}$ M_H : VG ⇒ I* $\xrightarrow{G\,:\,rdinv_reply}$

M_R : VG ⇒ VL $\xrightarrow{L\,:\,rdinv_reply(X,L,D)}$ C_R : I**** ⇒ VXLDO

(10)



C_R :I**** $\xrightarrow{L : rdinv_req}$ M_R:I* $\xrightarrow{G : rdinv_req}$ M_H:I* $\xrightarrow{G : rdinv_req}$ M_O:VL $\xrightarrow{L : rdinv_req}$

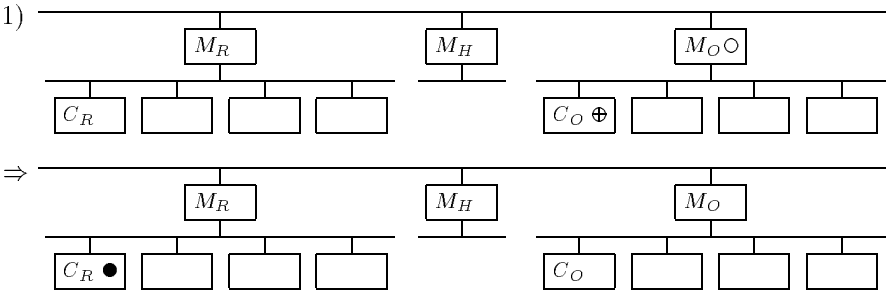
C_O :VXLDO $\xrightarrow{L : rdinv_reply(X, L, D)}$ M_O:VL $\xrightarrow{G : rdinv_reply}$ M_H:I* $\xrightarrow{G : rdinv_reply}$

⇒ I**** ⇒ I*

M_R:I* $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :I****

⇒ VL ⇒ VXLDO

(11)



C_R :I**** $\xrightarrow{L : rdinv_req}$ M_R:I* $\xrightarrow{G : rdinv_req}$ M_H:I* $\xrightarrow{G : rdinv_req}$ M_O:VL $\xrightarrow{L : rdinv_req}$

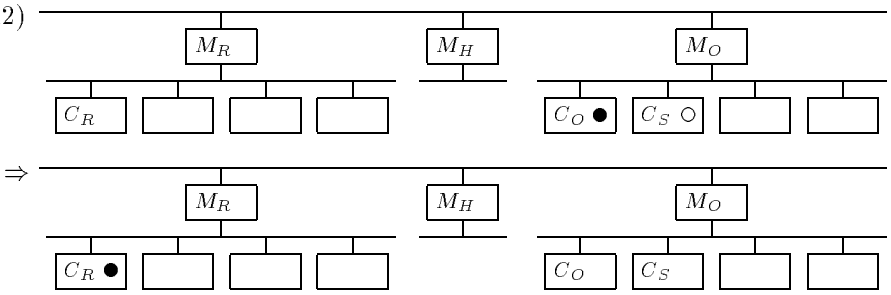
C_O :VXLCO $\xrightarrow{L : rdinv_reply(X, L, C)}$ M_O:VL $\xrightarrow{G : rdinv_reply}$ M_H:I* $\xrightarrow{G : rdinv_reply}$

⇒ I**** ⇒ I*

M_R:I* $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :I****

⇒ VL ⇒ VXLDO

(12)



C_R :I**** $\xrightarrow{L : rdinv_req}$ M_R:I* $\xrightarrow{G : rdinv_req}$ M_H:I* $\xrightarrow{G : rdinv_req}$ M_O:VL $\xrightarrow{L : rdinv_req}$

C_O :VSLDO $\xrightarrow{L : rdinv_reply(X, L, D)}$ M_O:VL $\xrightarrow{G : rdinv_reply}$ M_H:I* $\xrightarrow{G : rdinv_reply}$

⇒ I**** ⇒ I*

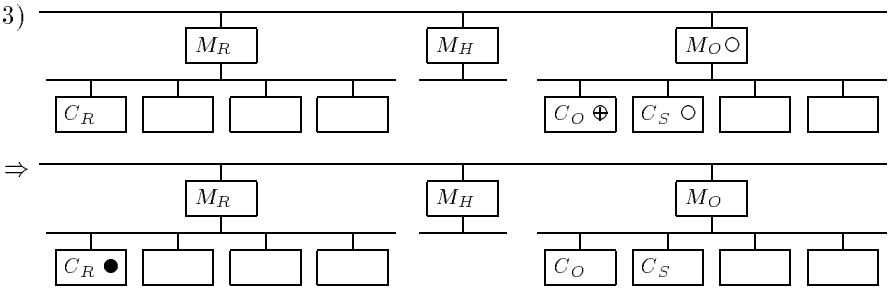
C_S :VSLCN

⇒ I****

M_R:I* $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :I****

⇒ VL ⇒ VXLDO

(13)



C_R : I**** $\xrightarrow{L:\ rdinv_req}$ M_R : I* $\xrightarrow{G:\ rdinv_req}$ M_H : I* $\xrightarrow{G:\ rdinv_req}$ M_O : VL $\xrightarrow{L:\ rdinv_req}$

C_O : VSLCO $\xrightarrow{L:\ rdinv_reply(X,L,C)}$ M_O : VL $\xrightarrow{G:\ rdinv_reply}$ M_H : I* $\xrightarrow{G:\ rdinv_reply}$

$\Rightarrow$ I**** $\Rightarrow$ I*

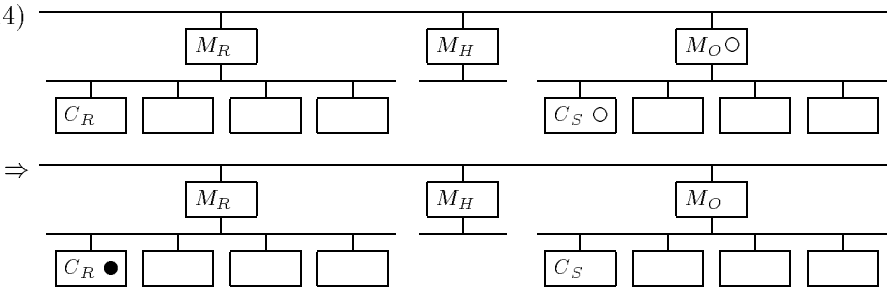
C_S : VSLCN

$\Rightarrow$ I****

M_R : I* $\xrightarrow{L:\ rdinv_reply(X,L,D)}$ C_R : I****

$\Rightarrow$ VL $\Rightarrow$ VXLDO

(14)



C_R : I**** $\xrightarrow{L:\ rdinv_req}$ M_R : I* $\xrightarrow{G:\ rdinv_req}$ M_H : I* $\xrightarrow{G:\ rdinv_req}$ M_O : VL $\xrightarrow{L:\ rdinv_req}$

M_O : VL $\xrightarrow{L:\ rdinv_reply(X,L,C)}$ M_O : VL $\xrightarrow{G:\ rdinv_reply}$ M_H : I* $\xrightarrow{G:\ rdinv_reply}$

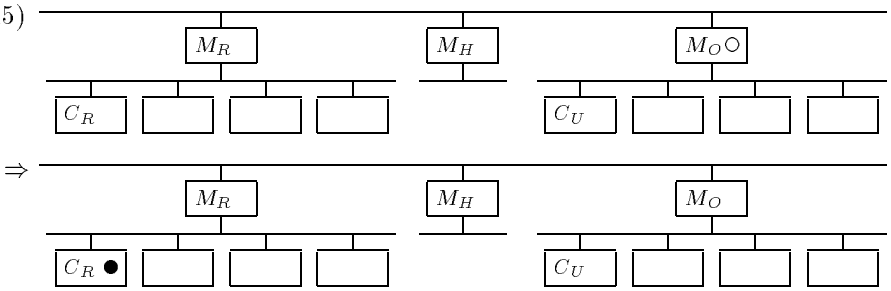
C_S : VSLCN $\Rightarrow$ I*

$\Rightarrow$ I****

M_R : I* $\xrightarrow{L:\ rdinv_reply(X,L,D)}$ C_R : I****

$\Rightarrow$ VL $\Rightarrow$ VXLDO

(15)



C_R : I**** $\xrightarrow{L:\ rdinv_req}$ M_R : I* $\xrightarrow{G:\ rdinv_req}$ M_H : I* $\xrightarrow{G:\ rdinv_req}$ M_O : VL $\xrightarrow{L:\ rdinv_req}$

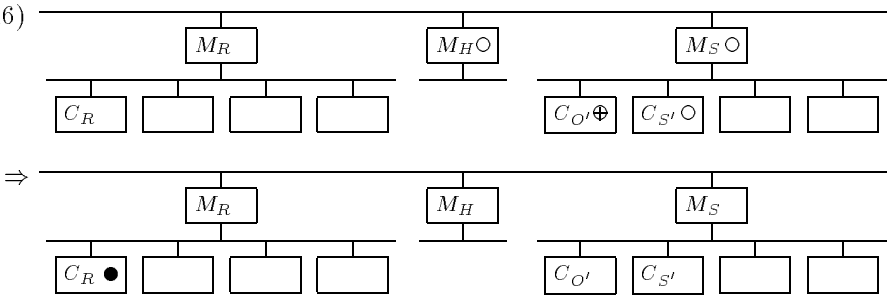
M_O : VL $\xrightarrow{L:\ rdinv_reply(X,L,C)}$ M_O : VL $\xrightarrow{G:\ rdinv_reply}$ M_H : I* $\xrightarrow{G:\ rdinv_reply}$

$\Rightarrow$ I*

M_R : I* $\xrightarrow{L:\ rdinv_reply(X,L,D)}$ C_R : I****

$\Rightarrow$ VL $\Rightarrow$ VXLDO

(16)



C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : I* $\xrightarrow{L : rdinv_req}$ M_H : VG $\xrightarrow{G^m : invalidate_req}$

M_S : VG $\xrightarrow{L : invalidate_req}$ M_S : VG ⇒ I* $C_{O'}$: VSGCO ⇒ I**** $C_{S'}$: VSGCN ⇒ I**** $\xrightarrow{G^c : invalidate_ack}$ M_H : VG ⇒ I* $\xrightarrow{G : rdinv_reply}$

M_R : VG ⇒ VL $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : I**** ⇒ VXLDO

2.4 $n_c_rd_req \leftarrow$ **normal-read** (coherence-policy(r) = no-coherent)

$n_c_rd_req$ は，読出の coherence-policy が no-coherent であるような **normal-read** が cache-miss したときの，スヌープを伴わない block-load 要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0000/001	= $n_c_rd_req$
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= pid	= own ID
drty	= $\perp$	
clsh	= $\perp$	
fe	= $\perp$	
addr	= $addr$	= VBUS.addr

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。したがって，対象ラインが複数の L2C に所有されている場合，その coherence は保証されない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (CBUS:x_crply = 1^F), MMC は対象ラインの STAG をアクセスする。その結果 STAG:{rd_trap|valid} = $\{0^F1^T\}$ であれば，cluster-memory を読出して以下の $n_c_rd_reply$ を応答する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/001	= $n_c_rd_reply$
size	= 100	= block
ack	= 0	= nop
did	= did	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 0	= clean
clsh	= $global$	= STAG.global
fe	= $sync$	= STAG.fe
addr	= $addr$	= requested address
wwe[0:3]	= $w\text{-}wait/r\text{-}wait$	= DTAG[dwix..dwix+3]
data[0:3]	= $data$	= DRAM[dwix..dwix+3]

直接応答が不可能な場合には MBPL によって処理され，home-cluster にライン取得を要求するなどして，最終的には上記のパケットが応答される。

[注意]

home-cluster にラインが存在しない状況は想定しがたい。またそのような状況で owner であるクラスタにおいて MBPL が CBUS に対して $n_c_read_req$ を送出すると，L2C は不正パケット割込を生起する。

応答受領時の動作

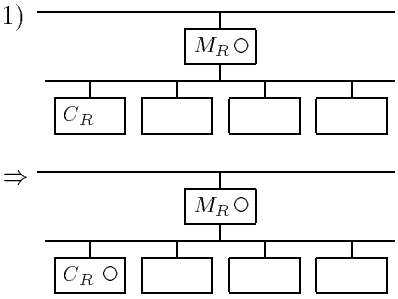
状態遷移は $read_reply$ と同様の機構によるが，owner となるのは危険であるため (万一共有が発生すると owner が複数になる) S = VxgdN に遷移する。ただし drty $\equiv$ d $= 0^F$ であり，また応答送出時には他の L2C がラインを所有していることはないため，結果的に S = VXLCN または S = VSGCN となる。

また常に data を CPU に返す。

[注意]
MMC が CBUS に $n_c_rd_reply$ を応答した時点で，対象ラインを共有しているような L2C が万一存在した場合，要求元 L2C の状態は $S = \texttt{VSgCN}$ となる。しかしこの状態で no-coherent な store を行なっても，L2C は CBUS に store に関するパケットを送出しない。

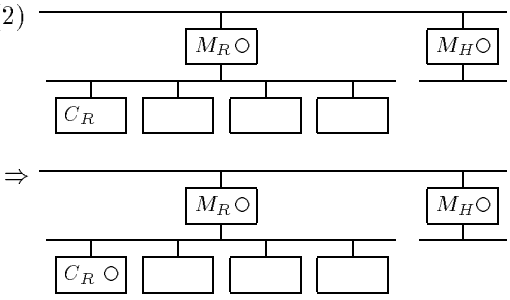
パケットの流れと状態遷移

(1)



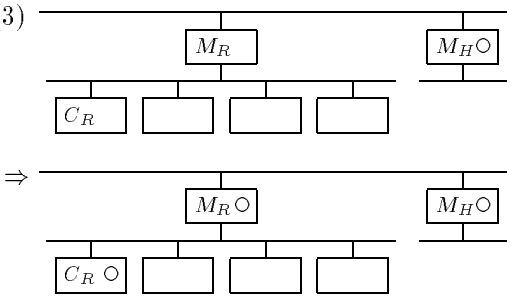
C_R :I**** $\xrightarrow{L : n_c_read_req}$ M_R:VL $\xrightarrow{L : n_c_read_reply(X, L, C)}$ C_R :I****
⇒VXLCO

(2)



C_R :I**** $\xrightarrow{L : n_c_read_req}$ M_R:VG $\xrightarrow{L : n_c_read_reply(X, G, C)}$ C_R :I****
⇒VSGCO

(3)



C_R :I**** $\xrightarrow{L : n_c_read_req}$ M_R:I* $\xrightarrow{G : n_c_read_req}$ M_H:VG $\xrightarrow{G : n_c_read_reply}$

M_R:I* $\xrightarrow{L : n_c_read_reply(X, G, C)}$ C_R :I****
⇒VG ⇒VSGCO

2.5 *rdinv_q_req* ← q-read, p-read, q-check (cache-hit)

rdinv_q_req は，**q-read**, **p-read**, **q-check** が cache-hit したときの，dw-queue に対する読出または prefetch 要求である。ただし **q-check** では，対象 dw が empty である場合にのみ prefetch 要求を行なう。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0100/110	= *rdinv_q_req*
size	= 011	= doubleword
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= VBUS.addr

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。一般に L2C から *rdinv_q_req* が送出される時点では対象ラインが他の L2C に共有されていることは想定されておらず，万一共有されている場合には dw-queue の動作は保証されない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (CBUS:x_crply = 1^F), MMC が *rdinv_q_req* を受理して無条件に MBPL に割出す。

MBPL による処理の詳細は資料 [6] に示すが，概ね以下の通りである。

(1) *rdinv_q_req* の addr をキーとして，dw-queue の制御情報を得る。制御情報の中には，queue が空か否かと，queue 本体を構成する (ring) buffer の先頭が含まれる。

(2) queue が空でなければ先頭のデータを取り出して (dequeue), 以下の *rdinv_q_reply* を応答する。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/110	= *rdinv_q_reply*
size	= 011	= doubleword
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 1	= dirty
clsh	= 0	= local
fe	= 1111	= full
addr	= *addr*	= requested address
wwe[0]	= 00	
data[0]	= *data*	= queue top

(3) queue が空であれば以下の *rdinv_q_reply* を応答する。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/110	= *rdinv_q_reply*
size	= 011	= doubleword
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 1	= dirty
clsh	= 0	= local
fe	= 0000	= empty
addr	= *addr*	= requested address
wwe[0]	= 00	
data[0]	= $\perp$	

なお上記の操作は常に queue の home-cluster で行なう。

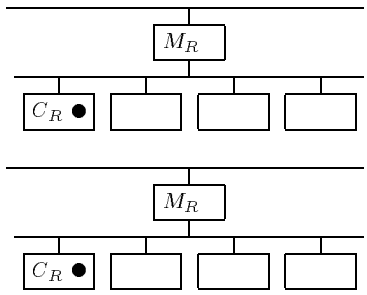
応答受領時の動作

read_reply と同様の機構によって $S =$ VxgdO に遷移するが，必ず clsh $\equiv$ g $= 0^F$, drty $\equiv$ d $= 1^T$ であり，また応答送出時には他の L2C がラインを所有していることはないため，結果的に $S =$ VXLDO となる。

また prefetch 要求でない場合，すなわち **q-read** であって要求送出時点での L2C の状態が empty である場合，*rdinv_q_reply* の fe $= 1111$ であれば応答された data を，また fe $= 0000$ であれば undefined-error をそれぞれ CPU に返す。

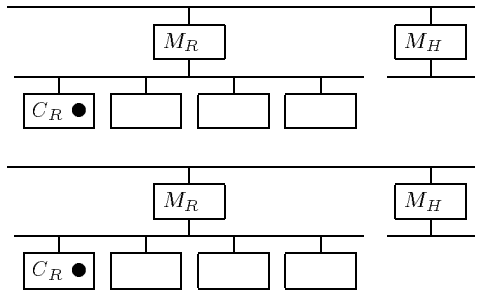
パケットの流れと状態遷移

(1)



C_R :VXLDO $\xrightarrow{L : rdinv_q_req}$ M_R:VL $\xrightarrow{L : rdinv_q_reply(X, L, D)}$ C_R :VXLDO

(2)



C_R :VXLDO $\xrightarrow{L : rdinv_q_req}$ M_R:I* $\xrightarrow{G : rdinv_q_req}$ M_H:VG $\xrightarrow{G : rdinv_q_reply}$

M_R:I* $\xrightarrow{L : rdinv_q_reply(X, L, D)}$ C_R :VXLDO

2.6 *rdinv_p_req* ← q-read, p-read, q-check (cache-miss)

rdinv_p_req は，**q-read**, **p-read**, **q-check** が cache-miss したときの，block-load 要求である。

送出される要求パケット

`plen`	= 00	= 1 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0100/100	= *rdinv_p_req*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= 100	= MMC/MBPL
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= ⊥	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。ただし，ある L2C が dw-queue の読出権を所有している場合（すなわち queue top をキャッシュしている場合），MBPL の指令によって対象ラインが MBPL を経由して要求元の L2C に渡される。この状況以外では，対象ラインが他の L2C に共有されていることは想定されておらず，万一共有されている場合には dw-queue の動作は保証されない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (`CBUS:x_crply` = 1^F), MMC が *rdinv_p_req* を受理して無条件に MBPL に割出す。

MBPL による処理の詳細は資料 [6] に示すが，概ね以下の通りである。

(1) *rdinv_p_req* の `addr` をキーとして，dw-queue の制御情報を得る。制御情報の中には，queue が空か否かと，queue 本体を構成する (ring) buffer の先頭，および queue の読出権を所有している L2C の ID が含まれる。

(2) *rdinv_p_req* の要求元が queue の読出権を所有していなければ，CBUS に対して（通常の）*rdinv_req* を送出し，*rdinv_reply* によって対象ラインを得る。そうでなければ，cluster-memory に対象ラインが存在する[†1]。このラインを L とする。

(3) L の対象 dw に対応する同期ビット s と，queue が空か否かの情報 f (= 0^F if empty) により，L の対象 dw を以下のように変更する。

(a) $s = 1^T \vee f = 0^F$
変更しない。

(b) $s = 0^F \wedge f = 1^T$
先頭のデータを取り出して (dequeue) 対象 dw を置き換え，$s \leftarrow 1^T$ とする。

(4) L を以下の *rdinv_p_reply* により応答する。

[†1] 常に *rdinv_req* によって対象ラインを得てもよい。

`plen`	$= 10$	$= 5$ flit
`injmask`	$= 0$	= nop
`ccbl`	$= 1$	= cachable
`pcmd/scmd`	$= 0101/100$	$=$ *rdinv_p_reply*
`size`	$= 100$	= block
`ack`	$= 0$	= nop
`did`	$=$ *did*	= requester's ID
`sid`	$= 100$	= MMC/MBPL
`drty`	$= 1$	= dirty
`clsh`	$= 0$	= local
`fe`	$=$ *sync*	$= L$.`fe`
`addr`	$=$ *addr*	= requested address
`wwe[0:3]`	$= \{00\}*4$	
`data[0:3]`	$=$ *data*	$= L$

なお上記の操作は常に home-cluster で行なう。また dw-queue の読出権が home-cluster 以外にあれば，読出権を保有するクラスタから対象ラインを得る。

応答受領時の動作

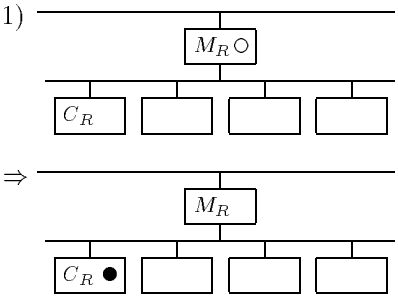
read_reply と同様の機構によって $S =$ `VxgdO` に遷移するが，必ず `clsh` $\equiv$ `g` $= 0^F$, `drty` $\equiv$ `d` $= 1^T$ であり，また応答送出時には他の L2C がラインを所有していることはないため，結果的に $S =$ `VXLDO` となる。

また memory-command と *rdinv_p_reply* の $f \equiv$ `fe`[`dwix`] に基づき，以下の値を CPU に返す。

mem. cmd.	f	return value
q-read	1^T	`data`
	0^F	undefined-error
p-read	1^T	`data`
	0^F	0
f-read	1^T	-1
	0^F	0

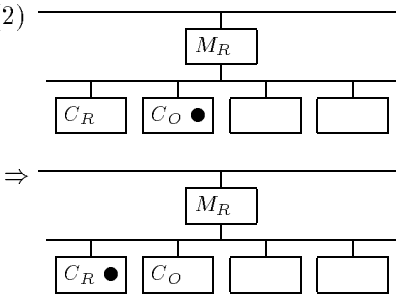
パケットの流れと状態遷移

(1)



C_R :I**** $\xrightarrow{L : rdinv_p_req}$ M_R:VL $\xrightarrow{L : rdinv_p_reply(X, L, D)}$ C_R :I****
⇒ VXLDO

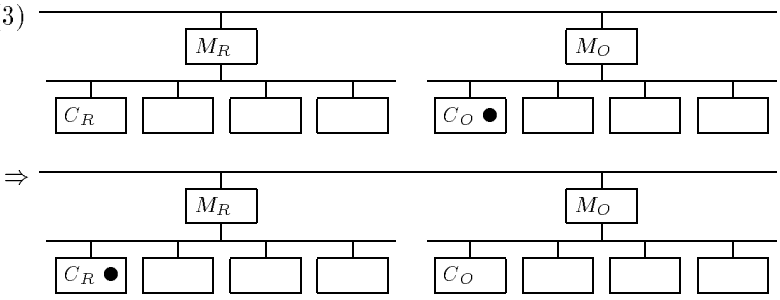
(2)



C_R :I**** $\xrightarrow{L : rdinv_p_req}$ M_R:VL $\xrightarrow{L : rdinv_req}$ C_O :VXLDO $\xrightarrow{L : rdinv_reply}$
⇒ I****

M_R:VL $\xrightarrow{L : rdinv_p_reply(X, L, D)}$ C_R :I****
⇒ VXLDO

(3)

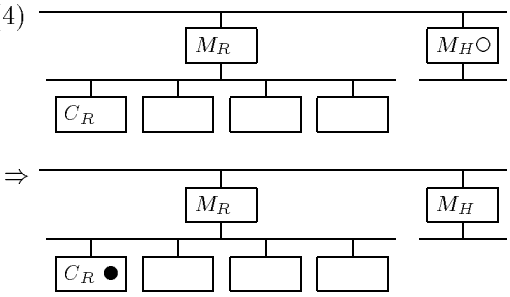


C_R :I**** $\xrightarrow{L : rdinv_p_req}$ M_R:VG $\xrightarrow{G : rdinv_req}$ M_O:I* $\xrightarrow{L : rdinv_req}$

C_O :VXLDO $\xrightarrow{L : rdinv_reply}$ M_O:I* $\xrightarrow{G : rdinv_reply}$ M_R:VG $\xrightarrow{L : rdinv_p_reply(X, L, D)}$
⇒ I**** ⇒ VL

C_R :I****
⇒ VXLDO

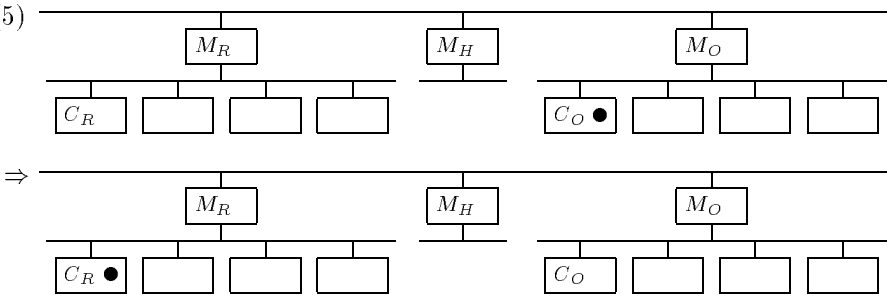
(4)



C_R :I**** $\xrightarrow{L : rdinv_p_req}$ M_R:I* $\xrightarrow{G : rdinv_p_req}$ M_H:VG $\xrightarrow{G : rdinv_p_reply}$

M_R:I* $\xrightarrow{L : rdinv_p_reply(X, L, D)}$ C_R :I****
$\Rightarrow$ VXLDO

(5)



C_R :I**** $\xrightarrow{L : rdinv_p_req}$ M_R:I* $\xrightarrow{G : rdinv_p_req}$ M_H:VG $\xrightarrow{G : rdinv_req}$

M_O:I* $\xrightarrow{L : rdinv_req}$ C_O :VXLDO $\xrightarrow{L : rdinv_reply}$ M_O:I* $\xrightarrow{G : rdinv_reply}$
$\Rightarrow$ I****

M_H:VG $\xrightarrow{G : rdinv_p_reply}$ M_R:I* $\xrightarrow{L : rdinv_p_reply(X, L, D)}$ C_R :I****
$\Rightarrow$ VXLDO

2.7 *bq_read_req* ← **bq-read, bq-check** (normal)

bq_read_req は，**bq-read**, **bq-check** による prefetch または block-load 要求である[†1]。

送出される要求パケット

`plen`	= 00	= 1 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0000/100	= *bq_read_req*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= 100	= MMC/MBPL
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= ⊥	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。ただし，ある L2C が block-queue の読出権を所有している場合（すなわち queue top をキャッシュしている場合），MBPL の指令によって読出権は MBPL を経由して要求元の L2C に渡される。この状況以外では，対象ラインが他の L2C に共有されていることは想定されておらず，万一共有されている場合には block-queue の動作は保証されない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (`CBUS:x_crply` = 1^F), MMC が *bq_read_req* を受理して無条件に MBPL に割出す。

MBPL による処理の詳細は資料 [7] に示す。応答は以下の 3 種類である。

(1) 要求元に読出権が与えられ，queue が空ではない場合[†2]。

`plen`	= 10	= 5 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0001/100	= *bq_read_reply*
`size`	= 100	= block
`ack`	= 0	= nop
`did`	= *did*	= requester's ID
`sid`	= 100	= MMC/MBPL
`drty`	= 0	= clean
`clsh`	= 0	= local
`fe`	= *sync*	= `fe` of queue top
`addr`	= *addr*	= requested address
`wwe[0:3]`	= *w-wait/r-wait*	= `DTAG`[queue top]
`data[0:3]`	= *data*	= queue top

(2) 要求元が読出権が与えられ，queue が空である場合[†3]。

[†1] *bq_read_req* は [3] では *read_xa_req* と記載されている。

[†2] *bq_read_reply* は [3] では *read_xa_reply* と記載されている。

[†3] *bq_nop_reply* は [3] では *read_xb_reply* と記載されている。また `fe` = 0000 の *bq_nop_reply* は [7] では *bq_empty_reply* と記載されている。

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/101	= *bq_nop_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= $\perp$	
clsh	= $\perp$	
fe	= 0000	= empty
addr	= *addr*	= requested address

(3) 要求元に読出権が与えられない場合[†1][†2]。

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/101	= *bq_nop_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= $\perp$	
clsh	= $\perp$	
fe	= 1111	= migrated
addr	= *addr*	= requested address

応答受領時の動作

bq_read_reply を受理すると，*read_reply* と同様の機構によって $S =$ VxgdO に遷移するが，必ず clsh $\equiv$ g $= 0^F$, drty $\equiv$ d $= 0^F$ であり，また応答送出時には他の L2C がラインを所有していることはないため，結果的に $S =$ VXLCO となる。また block-load 要求であれば CPU には応答された data を返す。

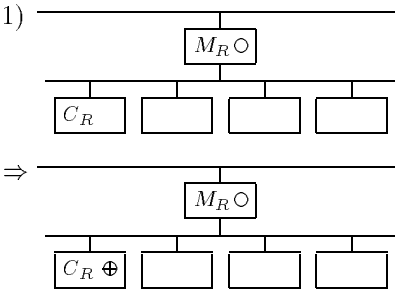
bq_nop_reply を受理すると，対象ラインを無効化し，block-load 要求であれば CPU に，fe $= 0000$ の場合は undefined-error を，fe $= 1111$ の場合は bus-error を，それぞれ返す。

[†1] *bq_nop_reply* は [3] では *read_xb_reply* と記載されている。また fe $= 1111$ の *bq_nop_reply* は [7] では *bq_migrated_reply* と記載されている。

[†2] 読出権の移動処理中は結果的に要求元に読出権が与えられる場合も，この応答が返されることがある。

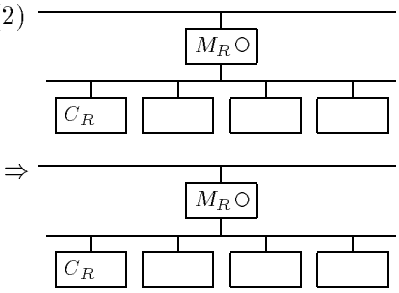
パケットの流れと状態遷移

(1)



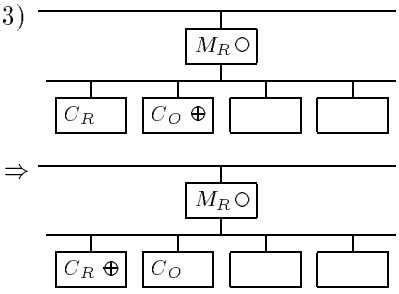
C_R :I**** $\xrightarrow{L : bq_read_req}$ M_R:VL $\xrightarrow{L : bq_read_reply(X, L, C)}$ C_R :I****
⇒ VXLCO

(2)



C_R :I**** $\xrightarrow{L : bq_read_req}$ M_R:VL $\xrightarrow{L : bq_nop_reply(empty)}$ C_R :I****

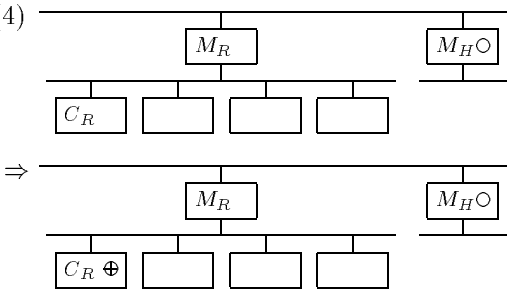
(3)



C_R :I**** $\xrightarrow{L : bq_read_req}$ M_R:VL $\xrightarrow{L : invalidate_req \times 4}$

M_R:VL $\xrightarrow{L : bq_read_reply(X, L, C)}$ C_R :I****
C_O :VXLCO ⇒ VXLCO
⇒ I****

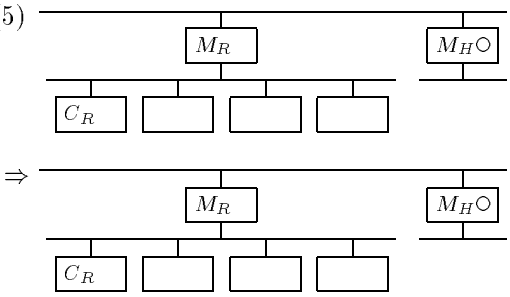
(4)



C_R :I**** $\xrightarrow{L : bq_read_req}$ M_R:I* $\xrightarrow{G : bq_read_req}$ M_H:VG $\xrightarrow{G : bq_read_reply}$

M_R:I* $\xrightarrow{L : bq_read_reply(X, L, C)}$ C_R :I****
⇒ VXLCO

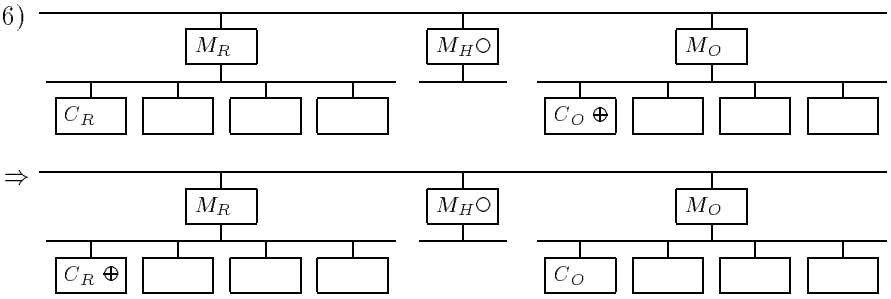
(5)



C_R :I**** $\xrightarrow{L : bq_read_req}$ M_R:I* $\xrightarrow{G : bq_read_req}$ M_H:VG $\xrightarrow{G : bq_nop_reply(empty)}$

M_R:I* $\xrightarrow{L : bq_nop_reply(empty)}$ C_R :I****

(6)



C_R :I**** $\xrightarrow{L : bq_read_req}$ M_R:I* $\xrightarrow{G : bq_read_req}$ M_H:VG $\xrightarrow{G : invalidate_req\times 4}$

M_O:I* $\xrightarrow{L : invalidate_req\times 4}$ M_R:I* $\xrightarrow{G : invalidate_ack\times 4}$ M_H:VG $\xrightarrow{G : bq_read_reply}$
C_O :VXLCO
⇒ I****

M_R:I* $\xrightarrow{L : bq_read_reply(X, L, C)}$ C_R :I****
⇒ VXLCO

2.8 *bq_nop_req* ← **bq-read, bq-check** (migration)

bq_nop_req は，**bq-read**, **bq-check** が CPU により発効されてから，対応する prefetch 要求が CBUS に送出されるまでの間に，block-queue の読出権移動によって対象ラインが無効化された場合の，prefetch 取消要求である[†1]。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0000/101	= *bq_nop_req*
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= VBUS.addr

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (CBUS:x_crply = 1^F), MMC が *bq_nop_req* を受理して無条件に MBPL に割出す。

MBPL による処理の詳細は資料 [7] に示すが，単に以下の *bq_nop_reply* を応答すればよい[†2]

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/101	= *bq_nop_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= 1111	= migrated
addr	= *addr*	= requested address

応答受領時の動作

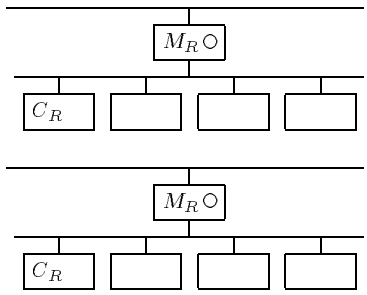
bq_nop_reply を受理すると，対象ラインを無効化し，block-load 要求であれば CPU に bus-error を返す。

[†1] *bq_nop_req* は [3] では *read_xb_req* と記載されている。

[†2] *bq_nop_reply* は [3] では *read_xb_reply* と記載されている。また fe = 1111 の *bq_nop_reply* は [7] では *bq_migrated_reply* と記載されている。

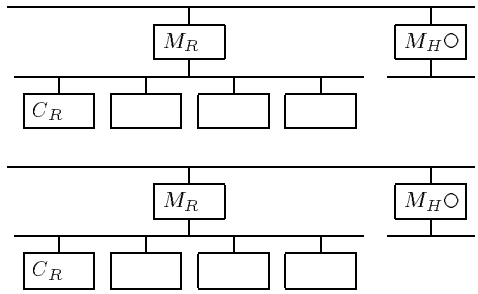
パケットの流れと状態遷移

(1)



C_R :I**** $\xrightarrow{L : bq_nop_req}$ M_R:VL $\xrightarrow{L : bq_nop_reply(migrated)}$ C_R :I****

(2)



C_R :I**** $\xrightarrow{L : bq_nop_req}$ M_R:I* $\xrightarrow{L : bq_nop_reply(migrated)}$ C_R :I****

3 L2-cachable store

L2-cachable store とは CPU が行なう write 要求 (すなわち read でも swap でもないメモリ・アクセス要求) のうち，control-space access ではなく，かつ L2-cachable であるものをいう。ただし，prefetch/flush (5 章参照) は除かれる。したがって，「通常」の store の実行は L2-cachable store となる。

L2-cachable store が許される memory-command は表 3.1 に示すものであり [2]，かつ page table entry の L1-cachability の指定によって同表の C 欄に示す以下の条件が加味される。

- yes
  page table の指定が L1-cachable でなければならず，また無条件に L2-cachable である。
- y/n
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば (またその時に限り) L2-cachable である。
- y/n(S)
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば L2-cachable である。L1-cachable でなければ，L2-cachability は `L2CCTL.l2ccbl` に従う。
- no(S)
  page table の指定は L1-cachable であってはならず，L2-cachability は `L2CCTL.l2ccbl` に従う。

L2-cachable store によって生成されるパケットは，表 3.1 に示すものである。一般には L2C のミス (同表の $H = 0^F$) あるいは shared-hit ($H = 1^T$) によって生じるが，いくつかの memory-command では exclusive-hit してもパケットが生成される。

また同表の*1 に示した条件の意味は，以下の通りである。

- d
  物理アドレスのビット 29 (dummy bit)。dummy-page に対する **s-write** が cache-miss した場合，block-load を行なわずに *no_alloc_write_req* を送出して更新操作だけを行なう。
- w
  s-write による書込待ちスレッドが存在することを示す `FMP:ww`。$w = 1^T$ であるようなラインに対する **s-write** が cache-hit すると，*update_direct_s_req* を送出してスレッド起動を伴う更新操作を要求する。なお exclusive-hit であっても *update_direct_s_req* が送出される。
- m
  要求パケット送出時点において，対象ラインのバイト i $(0 \leq i \leq 31)$ が更新されていれば $m_i = 1^T$ とする。m はラインの更新状況が word 単位か否かを示し ；

$$m^j = \begin{cases} 1^T & \text{if } \left(\prod_{k=0}^{3} m_{4j+k} = 1\right) \vee \left(\sum_{k=0}^{3} m_{4j+k} = 0\right) \\ 0^F & \text{otherwise} \end{cases}$$

$$m = \prod_{j=0}^{7} m^j$$

である。$m = 0$ であれば (すなわち更新が word 単位でなければ) coherence-policy が update 系であるような L2-cachable store が shared-hit した場合にも，*invalidate_via_home_req* により更新操作 (コピーの無効化) を行なう。

表 3.1: L2-cachable store によるパケット生成

mcmd	type	size	C	H	*1	mnem	pcmd/scmd	section
0010	—	—	no(S)	—		**q-write**	*update_via_home_q_req*	3.10節
0011	—	—	no(S)	—		**q-write**	*update_via_home_q_req*	3.10節
0100	$\{0,1\}$	$\neg hw$	no(S)	0^F	$d=0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0100	$\{0,1\}$	$\neg hw$	no(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0100	$\{0,1\}$	$\neg hw$	no(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.7節
0100	$\{0,1\}$	$\neg hw$	no(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
0100	3	$\neg hw$	y/n(S)	0^F	$d=0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0100	3	$\neg hw$	y/n(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0100	3	$\neg hw$	y/n(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.7節
0100	3	$\neg hw$	y/n(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
0101	$\{0,1\}$	$\neg hw$	no(S)	0^F	$d=0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0101	$\{0,1\}$	$\neg hw$	no(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0101	$\{0,1\}$	$\neg hw$	no(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.7節
0101	$\{0,1\}$	$\neg hw$	no(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
0101	3	$\neg hw$	y/n(S)	0^F	$d=0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0101	3	$\neg hw$	y/n(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0101	3	$\neg hw$	y/n(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.7節
0101	3	$\neg hw$	y/n(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
0110	2	$\neg hw$	no(S)	0^F	$d=0^F$	**s-write** (all-read/update-direct)	*allread_req*	3.2節
0110	2	$\neg hw$	no(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0110	2	$\neg hw$	no(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.7節
0110	2	$\neg hw$	no(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
0110	3	$\neg hw$	y/n(S)	0^F	$d=0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0110	3	$\neg hw$	y/n(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0110	3	$\neg hw$	y/n(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.7節
0110	3	$\neg hw$	y/n(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
0111	$\{0,1\}$	$\neg hw$	y/n	0^F		**normal-write** (all-read/update)	*allread_req*	3.2節
0111	$\{0,1\}$	$\neg hw$	y/n	1^T	$m=0^F$	**normal-write** (all-read/update)	*invalidate_via_home_req*	3.5節
0111	$\{0,1\}$	$\neg hw$	y/n	1^T	$m=1^T$	**normal-write** (all-read/update)	*update_via_home_req*	3.6節
0111	2	$\neg hw$	no(S)	0^F	$d=0^F$	**s-write** (all-read/update-direct)	*allread_req*	3.2節
0111	2	$\neg hw$	no(S)	0^F	$d=1^T$	**s-write** (all-read/update-direct)	*no_alloc_write_req*	3.9節
0111	2	$\neg hw$	no(S)	1^T	$w=0^F$	**s-write** (all-read/update-direct)	*update_direct_req*	3.7節
0111	2	$\neg hw$	no(S)	1^T	$w=1^T$	**s-write** (all-read/update-direct)	*update_direct_s_req*	3.8節
0111	3	$\neg hw$	y/n(S)	0^F	$d=0^F$	**s-write** (normal/update-direct)	*read_req*	3.1節
0111	3	$\neg hw$	y/n(S)	0^F	$d=1^T$	**s-write** (normal/update-direct)	*no_alloc_write_req*	3.9節
0111	3	$\neg hw$	y/n(S)	1^T	$w=0^F$	**s-write** (normal/update-direct)	*update_direct_req*	3.8節
0111	3	$\neg hw$	y/n(S)	1^T	$w=1^T$	**s-write** (normal/update-direct)	*update_direct_s_req*	3.8節
1000	—	—	yes	0^F		**bq-write**	*rdinv_req*	3.3節
1000	—	—	yes	1^T		**bq-write**	*bq_write_req*	3.11節
1001	$\{0,1,2\}$	—	y/n	0^F		**normal-write** (normal/invalidate)	*rdinv_req*	3.3節
1001	$\{0,1,2\}$	—	y/n	1^T		**normal-write** (normal/invalidate)	*invalidate_via_home_req*	3.5節
1010	—	$\neg hw$	y/n	0^F		**normal-write** (normal/invalidate)	*rdinv_req*	3.3節
1010	—	$\neg hw$	y/n	1^T		**normal-write** (normal/invalidate)	*invalidate_via_home_req*	3.5節
1011	—	$\neg hw$	y/n	0^F		**normal-write** (read-invalidate/invalidate)	*rdinv_req*	3.3節
1011	—	$\neg hw$	y/n	1^T		**normal-write** (read-invalidate/invalidate)	*invalidate_via_home_req*	3.5節
1100	{0,1,2}	—	y/n	0^F		**normal-write** (no-coherent/no-coherent)	*n_c_rd_req*	3.4節
1100	3	$\neg hw$	y/n	0^F		**normal-write** (no-coherent/no-coherent)	*n_c_rd_req*	3.4節
1101	—	$\neg hw$	y/n	0^F		**normal-write** (read-invalidate/update)	*rdinv_req*	3.3節
1101	—	$\neg hw$	y/n	1^T	$m=0^F$	**normal-write** (read-invalidate/update)	*invalidate_via_home_req*	3.5節
1101	—	$\neg hw$	y/n	1^T	$m=1^T$	**normal-write** (read-invalidate/update)	*update_via_home_req*	3.6節
1110	—	—	y/n	0^F		**normal-write** (normal/update-direct)	*read_req*	3.1節
1110	—	—	y/n	1^T	$m=0^F$	**normal-write** (normal/update-direct)	*invalidate_via_home_req*	3.5節
1110	—	—	y/n	1^T	$m=1^T$	**normal-write** (normal/update-direct)	*update_direct_req*	3.7節
1111	—	—	y/n	0^F		**normal-write** (normal/update)	*read_req*	3.1節
1111	—	—	y/n	1^T	$m=0^F$	**normal-write** (normal/update)	*invalidate_via_home_req*	3.5節
1111	—	—	y/n	1^T	$m=1^T$	**normal-write** (normal/update)	*update_via_home_req*	3.6節

3.1 *read_req* ← normal-write, s-write

(coherence-policy(r) = normal, coherence-policy(w) ≠ invalidate, cache-miss)

read_req は，読出の coherence-policy が normal であり，かつ書込の coherence-policy が invalidate ではないような (update 系の) **normal-write** または **s-write** が cache-miss したときの block-load 要求である。ただし，**s-write** については物理アドレスのビット 29 (dummy bit) が 1^T であれば，block-load を行なわずに *no_alloc_write_req* を送出する (3.9 節参照)。

生成される要求パケット，他の L2C の動作，MMC/MBPL の動作，パケットの流れと状態遷移については，2.1 節を参照せよ。

応答受領時の動作

応答 *read_reply* が送出された際の CBUS:x_msh $\equiv x$ と，応答パケットの clsh $\equiv$ g に従って S = VxgdO に遷移する。ただし ；

$$\mathtt{x} = \begin{cases} \mathtt{X} & x = 1^F \wedge \mathtt{g} = 0^F \\ \mathtt{S} & \text{otherwise} \end{cases}$$

$$\mathtt{d} = \begin{cases} \mathtt{D} & \mathtt{g} = 0^F \\ \mathtt{C} & \text{otherwise} \end{cases}$$

である。したがって，L2C が応答した場合には応答者自身が CBUS:x_msh $= 0^T$ とするので必ず shared となり，MMC/MBPL が応答した場合には他のクラスタと L2C の共有状況によって shared または exclusive となる。また globally-shared でなければ dirty となる。

続いてラインのデータを更新し，得られたラインの状態が VXLDO でなければ，書込の coherence-policy に従って，*update_via_home_req* (3.6 節)，*update_direct_req* (3.7 節)，または *update_direct_s_req* (3.8 節) を送出する。ただし **s-write** の場合，得られたラインについて ww $= 1^T$ であれば，状態に関わらず *update_direct_s_req* を送出する。

3.2 *allread_req* ← **normal-write, s-write** (coherence-policy(r) = all-read, cache-miss)

allread_req は，読出の coherence-policy が all-read であるような（update 系の）**normal-write** または **s-write** が cache-miss したときの，injection を伴う block-load 要求である。ただし，**s-write** については物理アドレスのビット 29(dummy bit) が 1^T であれば，block-load を行なわずに *no_alloc_write_req* を送出する（3.9 節参照）。

生成される要求パケット，他の L2C の動作，MMC/MBPL の動作，パケットの流れと状態遷移については，2.2 節を参照せよ。

応答受領時の動作

応答 *allread_reply* の `clsh` $\equiv$ `g` に従って $S =$ `VSgdO` に遷移する。ただし ；

$$\mathtt{d} = \begin{cases} \mathtt{D} & \mathtt{g} = 0^F \\ \mathtt{C} & \text{otherwise} \end{cases}$$

である。すなわち応答送出時点での他の L2C の共有状態に関わらず，injection がなされるものと仮定して shared となる。また globally-shared でなければ dirty となる。

続いてラインのデータを更新し，得られたラインの状態は shared であるので書込の coherence-policy に従って，*update_via_home_req*（3.6 節），*update_direct_req*（3.7 節），または *update_direct_s_req*（3.8 節）を送出する。

3.3 $rdinv_req \leftarrow$ **normal-write**

(coherence-policy(r) = read-invalidate $\vee$ coherence-policy(w) = invalidate, cache-miss),
bq-write (cache-miss)

rdinv_req は，読出の coherence-policy が read-invalidate であるか，あるいは書込の coherence-policy が invalidate であるような **normal-write**, あるいは **bq-write** が cache-miss したときの，無効化を伴う block-load 要求である。

生成される要求パケット，他の L2C の動作，MMC/MBPL の動作，パケットの流れと状態遷移については，2.3 節を参照せよ。

応答受領時の動作

応答 *rdinv_reply* を受けとると無条件に $S =$ `VXLDO` に遷移し[†1]，ラインのデータを更新する。**normal-write** の場合には要求パケットの再送出は行なわないが，**bq-write** の場合にはラインの末尾バイトが更新されていれば *bq_write_req* を再送出する (3.11 節参照)。

[†1] 応答の `clsh` は 0^F でなければならない。

3.4 $n_c_rd_req \leftarrow$ **normal-write**(coherence-policy(r) = no-coherent, cache-miss)

n_c_read_req は，読出の coherence-policy が no-coherent であるような **normal-write** が cache-miss したときの，スヌープを伴わない block-load 要求である。なお，このような **normal-write** の書込の coherence-policy は必ず no-coherent であり，cache-hit の際には単にラインを更新するだけで，CBUS への要求パケット送出は行なわない。

生成される要求パケット，他の L2C の動作，MMC/MBPL の動作，パケットの流れと状態遷移については，2.4 節を参照せよ。

応答受領時の動作

応答 *n_c_read_reply* の `clsh` に基づき ；

$$S = \begin{cases} \texttt{VXLDN} & \text{if } \texttt{clsh} = 0^F \\ \texttt{VSGDN} & \text{otherwise} \end{cases}$$

に遷移して，ラインのデータを更新する。要求パケットの再送出は行なわない。

なお上記の `VSGDN` は no-coherent であるようなラインに特有の例外的状態であり，L2C から除去される際に global-writeback を引き起こす。この状態にあるラインに対して，no-coherent ではないようなアクセスを行なった場合の結果は一切保証されない。

3.5 *invalidate_via_home_req* ←

normal-write (coherence-policy(w) = invalidate, shared-hit),

normal-write (coherence-policy(w) ∈ {update, update-direct}, ¬all-bytes, shared-hit)

invalidate_via_home_req は ；

(1) 書込の coherence-policy が invalidate であるような **normal-write** が shared-hit した場合。

(2) 書込の coherence-policy が update または update-direct であるような **normal-write** が cache-hit して一部の byte のみが更新された word が存在する場合[†1]。

のいずれかで送出される無効化要求である。なお update 系の更新要求パケットには word 単位の更新機能しかないため，(2) の場合は coherence-policy に関わらず *invalidate_via_home_req* を用いる。

locally-shared であれば，要求元 L2C は *invalidate_via_home_req* が CBUS に送出された時点で無効化が完了しているとみなし，ラインの状態は S = `VXLDO` に遷移する。

送出される要求パケット

`plen`	= 00	= 1 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 1100/000	= *invalidate_via_home_req*
`size`	= 100	= block
`ack`	= 1	= with ack
`did`	= 111	= all
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= *global*	= ¬`FMP:state.l`
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr` ([5 : 4] = 0)

他の L2C の動作

対象ラインを所有している L2C は，ラインの無効化を行なう。

MMC/MBPL の動作

MMC は `clsh` = 0^F であれば何もしない[†2]。そうでなければ大域的な操作となるので，pending counter を increment して MBPL に割出す。

MBPL は serialize のために無効化要求を home-cluster に送出し，その応答として対象ラインの値を得る。この操作は無効化要求がシステム内で並行して行なわれた場合に，ラインの最新の値を得るために必要である。したがって，globally-shared な *rdinv_req* と同じ操作をすればよい[†3]。このラインの最新値を受けとった後，MBPL は pending counter を decrement し，L2C に対して以下の *rdinv_reply* を応答する。

[†1] **s-write** には halfword, byte 単位の書込は存在しないので，*invalidate_via_home_req* が送出されることはない。

[†2] [1] によれば `clsh` = 0^F であっても STAG がアクセスされ，その結果に基づいて無視するか否かを判断するとあるが，MMC のスループット向上のために STAG のアクセスは `clsh` = 1^T に限定する実装となっている。

[†3] システム全体で無効化が完了しその応答が要求元クラスタに到着した時点で，要求元クラスタにラインの最新の値が存在していることが何らかの方法で保証できる場合には，home-cluster からラインのデータを受理する必要はない（完了応答は常に必要である）。このような最適化を可能とするため，L2C は globally-shared なラインに対する無効化を（*rdinv_req* ではなく）*invalidate_via_home_req* により要求する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/000	= *rdinv_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 1	= dirty
clsh	= 0	= STAG.global = 0
fe	= *sync*	= STAG.fe
addr	= *addr*	= requested address
wwe[0:3]	= *w-wait/r-wait*	= replied from home
data[0:3]	= *data*	= replied from home

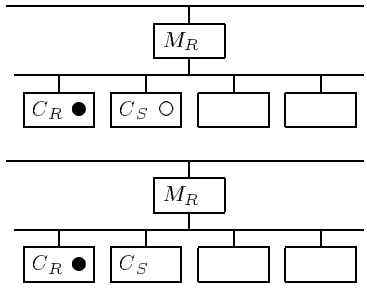
応答受領時の動作

locally-shared であれば *invalidate_via_home_req* が CBUS に送出された時点で無効化が完了しているとみなせるため，L2C は応答を期待しない。globally-shared であれば *rdinv_reply* を受けとると無条件に S = VXLDO に遷移し[†1]，ラインのデータを更新する。

[†1] 応答の clsh は 0^F でなければならない。

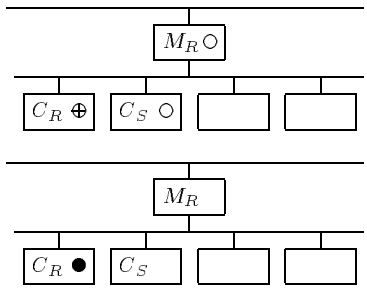
パケットの流れと状態遷移

(1)



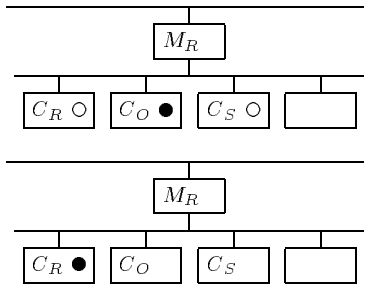
C_R :VSLDO $\xrightarrow{L : invalidate_via_home_req}$ C_S :VSLCN
⇒VXLDO ⇒I****

(2)



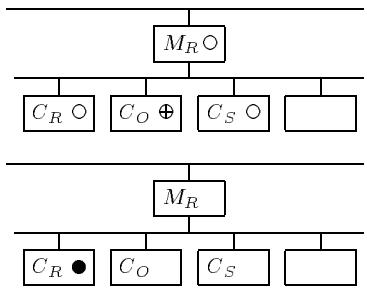
C_R :VSLCO $\xrightarrow{L : invalidate_via_home_req}$ C_S :VSLCN
⇒VXLDO ⇒I****

(3)



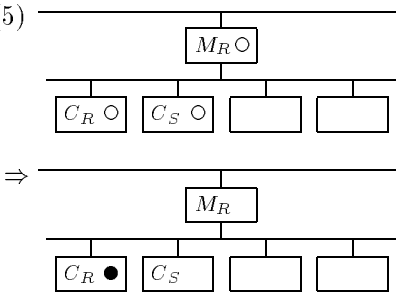
C_R :VSLCN $\xrightarrow{L : invalidate_via_home_req}$ C_O :VSLDO
⇒VXLDO ⇒I****
C_S :VSLCN
⇒I****

(4)



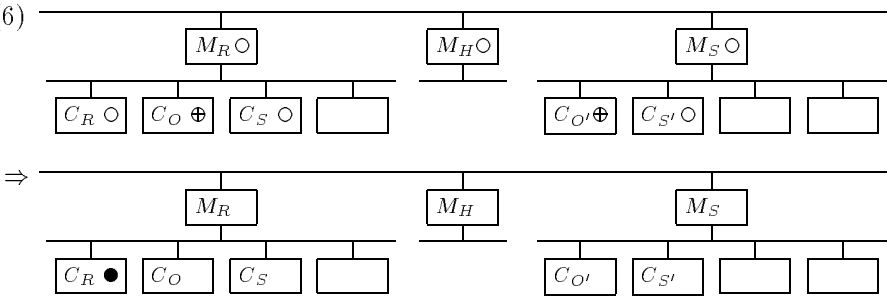
C_R :VSLCN $\xrightarrow{L : invalidate_via_home_req}$ C_O :VSLCO
⇒VXLDO ⇒I****
C_S :VSLCN
⇒I****

(5)



C_R : VSLCN $\xrightarrow{L : invalidate_via_home_req}$ C_S : VSLCN
⇒ VXLDO ⇒ I****

(6)



C_R : VSGCN $\xrightarrow{L : invalidate_via_home_req}$ M_R : VG $\xrightarrow{G : rdinv_req}$ M_H : VG $\xrightarrow{G^m : invalidate_req}$
C_O : VSGCO
⇒ I****
C_S : VSGCN
⇒ I****

M_S : VG $\xrightarrow{L : invalidate_req}$ M_S : VG $\xrightarrow{G^c : invalidate_ack}$ M_H : VG $\xrightarrow{G : rdinv_reply}$
⇒ I* ⇒ I*
$C_{O'}$: VSGCO
⇒ I****
$C_{S'}$: VSGCN
⇒ I****

M_R : VG $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : VSGCN
⇒ VL ⇒ VXLDO

3.6 *update_via_home_req* ← **normal-write**
(coherence-policy(w) = update, all-bytes, shared-hit)

update_via_home_req は，書込の coherence-policy が update であるような **normal-write** が shared-hit し，一部の byte のみが更新された word が存在しないときの，更新要求である。なお *update_via_home_req* には word 単位の更新機能しかないため，一部の byte のみが更新された word が存在する場合は *invalidate_via_home_req* を用いる（3.5 節参照）。

要求元 L2C では *update_via_home_req* を CBUS に送出した時点で処理が完了したものとみなし[†1]，対象ラインの（locally-shared であれば dirty な）owner となる。また locally-shared である場合，要求送出時にラインを共有している L2C が存在しないことが判明すれば (CBUS:x_msh = 1^F), S = VXLDO に遷移する。

送出される要求パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1000/000	= *update_via_home_req*
size	= 110	= masked block
ack	= 1	= with ack
did	= 111	= all
sid	= *pid*	= own ID
drty	= $\perp$	
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= VBUS.addr ($[5:4] = 0$)
wwe[0:3]	= *mdfd*	= word modified
data[0:3]	= *data*	= VBUS.data

他の L2C の動作

対象ラインを所有している L2C は，ラインの中で wwe[i] = 1^T であるような word のみを更新する。なお，この更新操作はラインが globally-shared である場合にも行なわれる。したがって更新要求がシステム内で非同期的に並行して行なわれた場合，遷移的に矛盾したデータが観測されることがある。

また owner であるような L2C が存在すれば，ownership を放棄する。

MMC/MBPL の動作

MMC は clsh = 0^F であれば何もしない[†2]。そうでなければ大域的な操作となるので，pending counter を increment して MBPL に割出す。

MBPL は cluster-memory の対象ラインの wwe[i] = 1^T であるような word を更新し，serialize のために更新要求を home-cluster に送出する[†3]。更新要求に対する完了確認を得た後，pending counter を decrement して完了する。

[注意]

(1) *update_via_home_req* の wwe は word 単位の更新情報であり，DTAG が保持する待ちスレッドの有無情報ではない。したがって，MBPL が cluster-memory を更新する際には，DTAG を更新してはならない。

(2) home-cluster が update 要求を multicast する際には，その対象に要求元クラスタも含めなければならない。

応答受領時の動作

ラインの状態に関わらず，L2C は応答を期待しない。

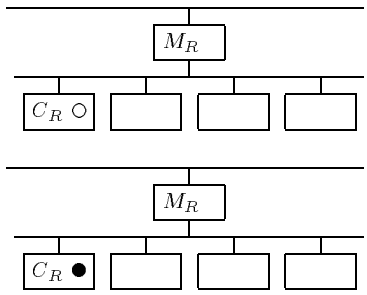
[†1] globally-shared の場合，pending store が存在することは pending counter が記憶する。

[†2] [1] によれば clsh = 0^F であっても STAG がアクセスされ，その結果に基づいて無視するか否かを判断するとあるが，MMC のスループット向上のために STAG のアクセスは clsh = 1^T に限定する実装となっている。また size が masked-block であれば MBPL に割出すとあるが，これは誤りである。

[†3] L2C が遷移的矛盾を生じるような更新操作を行なっているので，MBPL でも完了前に更新してよい。

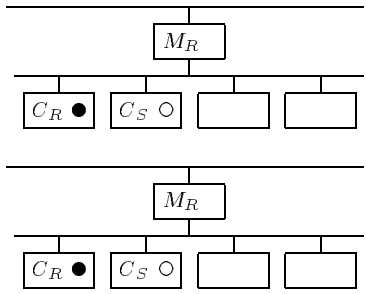
パケットの流れと状態遷移

(1)



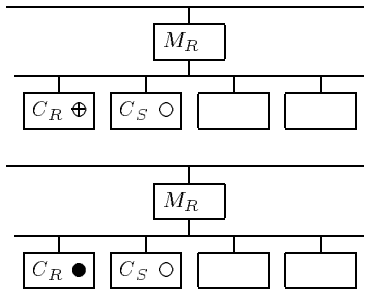
C_R :VSLCN $\xrightarrow{L : update_via_home_req}$ C_R :VSLCN
⇒ VXLDO

(2)



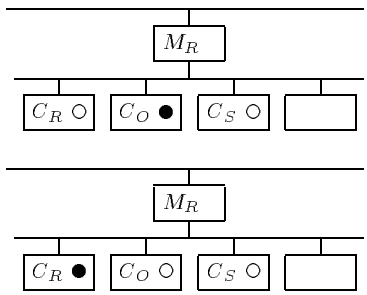
C_R :VSLDO $\xrightarrow{L : update_via_home_req}$ C_S :VSLCN
⇒ VSLDO

(3)



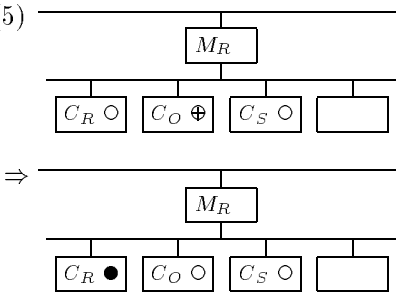
C_R :VSLCO $\xrightarrow{L : update_via_home_req}$ C_S :VSLCN
⇒ VSLDO

(4)



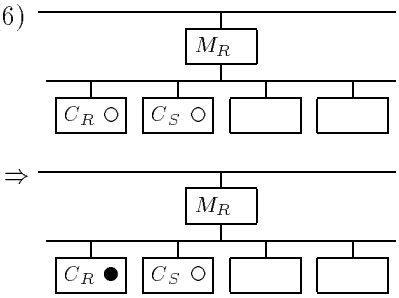
C_R :VSLCN $\xrightarrow{L : update_via_home_req}$ C_O :VSLDO
⇒ VSLDO ⇒ VSLCN
C_S :VSLCN

(5)



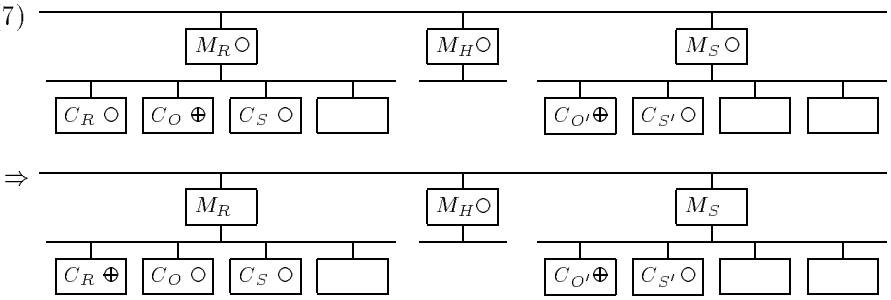
C_R : VSLCN ⇒ VSLDO $\xrightarrow{L : update_via_home_req}$ C_O : VSLCO ⇒ VSLCN, C_S : VSLCN

(6)



C_R : VSLCN ⇒ VSLDO $\xrightarrow{L : update_via_home_req}$ C_S : VSLCN

(7)



C_R : VSGCN ⇒ VSGCO $\xrightarrow{L : update_via_home_req}$ M_R : VG, C_O : VSGCO ⇒ VSGCN, C_S : VSGCN $\xrightarrow{G : update_req}$ M_H : VG $\xrightarrow{G^m : update_req}$

M_S : VG $\xrightarrow{L : update_req}$ M_S : VG, $C_{O'}$: VSGCO, $C_{S'}$: VSGCN $\xrightarrow{G^c : update_ack}$ M_H : VG $\xrightarrow{G : update_ack}$ M_R : VG

3.7 *update_direct_req* ← normal-write, s-write
(coherence-policy(w) = update-direct, all-bytes, shared-hit)

update_direct_req は，書込の coherence-policy が update-direct であるような **normal-write** または **s-write** が shared-hit し，一部の byte のみが更新された word が存在しないときの，更新要求である。なお *update_direct_req* には word 単位の更新機能しかないため，**normal-write** では一部の byte のみが更新された word が存在する場合は *invalidate_via_home_req* を用いる（3.5 節参照）。また **s-write** では，対象ラインの中に書込待ちスレッドが存在するような dw がある場合には (FMP:ww = 1^T)，*update_direct_s_req* を用いる（3.8 節参照）。

要求元 L2C では *update_direct_req* を CBUS に送出した時点で処理が完了したものとみなし[†1]，対象ラインの（locally-shared であれば dirty な）owner となる。また locally-shared である場合，要求送出時にラインを共有している L2C が存在しないことが判明すれば (CBUS:x_msh = 1^F)，S = VXLDO に遷移する。

送出される要求パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1010/000	= *update_direct_req*
size	= 110	= masked block
ack	= 1/*ack*	= 1/L2CCTL.wacks
did	= 111	= all
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= VBUS.addr ([5 : 4] = 0)
wwe[0:3]	= *mdfd*	= word modified
data[0:3]	= *data*	= VBUS.data

normal-write であれば ack = 1^T，**s-write** であれば ack = L2CCTL.wacks である。

他の L2C の動作

対象ラインを所有している L2C は，ラインの中で wwe[i] = 1^T であるような word のみを更新する。なお，この更新操作はラインが globally-shared である場合にも行なわれる。一般に，update-direct は更新要求がシステム内で非同期的に並行して行なわれることを想定しておらず，万一非同期更新がなされた場合の遷移的あるいは恒久的なデータの矛盾は許容される。

また owner であるような L2C が存在すれば，ownership を放棄する。

MMC/MBPL の動作

MMC は clsh = 0^F であれば何もしない[†2]。そうでなければ大域的な操作となるので，pending counter を increment して MBPL に割出す。

MBPL は cluster-memory の対象ラインの wwe[i] = 1^T であるような word を更新し，ラインを所有している全てのクラスタに対して更新要求を送出する[†3][†4]。要求の宛先であるような全てのクラスタから更新要求に対する完了確認を得て，ack = 1^T であれば pending counter を decrement する[†5]。

[注意]

update_direct_req の wwe は word 単位の更新情報であり，DTAG が保持する待ちスレッドの有無情報ではない。したがって，MBPL が cluster-memory を更新する際には，DTAG を更新してはならない。

[†1] globally-shared の場合，pending store が存在することは pending counter が記憶する。

[†2] [1] によれば clsh = 0^F であっても STAG がアクセスされ，その結果に基づいて無視するか否かを判断するとあるが，MMC のスループット向上のために STAG のアクセスは clsh = 1^T に限定する実装となっている。また size が masked-block であれば MBPL に割出すとあるが，これは誤りである。

[†3] 非同期更新がなされた場合の遷移的 / 恒久的なデータ矛盾は許容されているので，更新操作の完了確認前に更新してよい。

[†4] 要求元クラスタに，全ての subscribed cluster を包含するようなマップが存在することを仮定している。

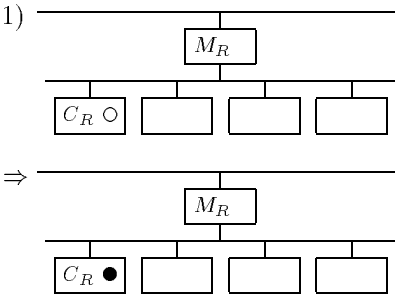
[†5] ack = 0^F であれば更新要求の完了確認をしないような最適化を行なってもよい。

応答受領時の動作

ラインの状態に関わらず，L2C は応答を期待しない。

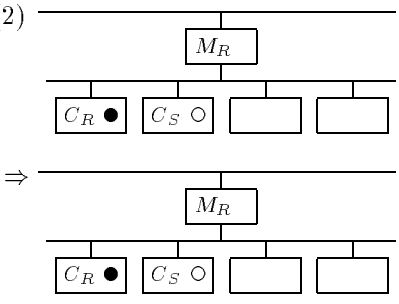
パケットの流れと状態遷移

(1)



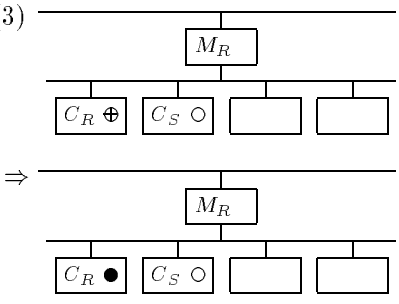
C_R :VSLCN $\xrightarrow{L : update_direct_req}$ C_R :VSLCN
⇒ VXLDO

(2)



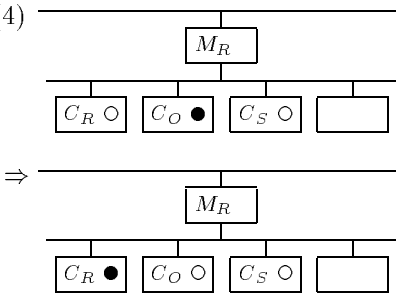
C_R :VSLDO $\xrightarrow{L : update_direct_req}$ C_S :VSLCN
⇒ VSLDO

(3)



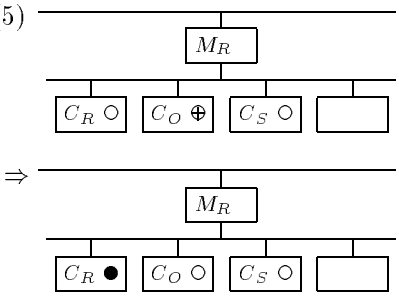
C_R :VSLCO $\xrightarrow{L : update_direct_req}$ C_S :VSLCN
⇒ VSLDO

(4)



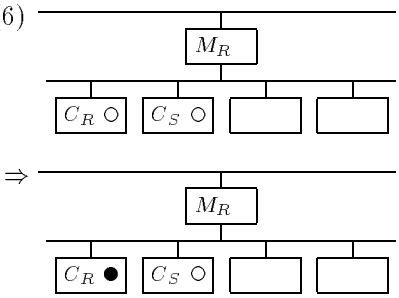
C_R :VSLCN $\xrightarrow{L : update_direct_req}$ C_O :VSLDO
⇒ VSLDO ⇒ VSLCN
C_S :VSLCN

(5)



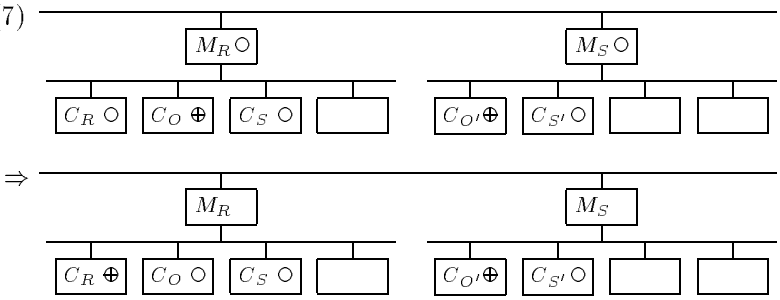
C_R : VSLCN ⇒ VSLDO $\xrightarrow{L : update_direct_req}$ C_O : VSLCO ⇒ VSLCN, C_S : VSLCN

(6)



C_R : VSLCN ⇒ VSLDO $\xrightarrow{L : update_direct_req}$ C_S : VSLCN

(7)



C_R : VSGCN ⇒ VSGCO $\xrightarrow{L : update_direct_req}$ M_R : VG, C_O : VSGCO ⇒ VSGCN, C_S : VSGCN $\xrightarrow{G^m : update_req}$ M_S : VG $\xrightarrow{L : update_req}$

M_S : VG, $C_{O'}$: VSGCO, $C_{S'}$: VSGCN $\xrightarrow{G^c : update_ack}$ M_R : VG

3.8 *update_direct_s_req* ← **s-write** (cache-hit, w-wait)

update_direct_s_req は，**s-write** が cache-hit し，かつ対象ラインの中に書込待ちスレッドが存在するような dw がある場合の ($\texttt{FMP:ww} = 1^T$)，更新要求である。

要求元 L2C では *update_direct_s_req* を CBUS に送出した時点で処理が完了したものとみなし[†1]，対象ラインの (locally-shared であれば dirty な) owner となる。また locally-shared である場合，要求送出時にラインを共有している L2C が存在しないことが判明すれば ($\texttt{CBUS:x_msh} = 1^F$), $S = \texttt{VXLDO}$ に遷移する。

送出される要求パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1010/100	= *update_direct_s_req*
size	= 110	= masked block
ack	= *ack*	= L2CCTL.wacks
did	= 111	= all
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= VBUS.addr ($[5:4] = 0$)
wwe[0:3]	= *mdfd*	= word modified
data[0:3]	= *data*	= VBUS.data

他の L2C の動作

対象ラインを所有している L2C は，ラインの中で $\texttt{wwe[i]} = 1^T$ であるような word のみを更新する。なお，この更新操作はラインが globally-shared である場合にも行なわれる。一般に，update-direct は更新要求がシステム内で非同期的に並行して行なわれることを想定しておらず，万一非同期更新がなされた場合の遷移的あるいは恒久的なデータの矛盾は許容される。

また owner であるような L2C が存在すれば，ownership を放棄する。

MMC/MBPL の動作

MMC は $\texttt{ack} = 1^T$ であれば pending counter を increment して無条件に MBPL に割出す。

MBPL は cluster-memory の対象ラインの $\texttt{wwe[i]} = 1^T$ であるような word を更新する。また $\texttt{clsh} = 1^T$ であれば，ラインを所有している全てのクラスタに対して更新要求を送出する[†2][†3]。要求の宛先であるような全てのクラスタから更新要求に対する完了確認を得て，$\texttt{ack} = 1^T$ であれば pending counter を decrement する[†4]。

$\texttt{clsh} = 1^T$ であれば上記の完了確認を得た後，また $\texttt{clsh} = 0^F$ であれば直ちに，書込待ちスレッドの起動操作を行なう[†5]。ただし，*update_direct_s_req* は対象ラインの中に書込待ちスレッドが存在する dw があることを意味しているにすぎず，そのような dw が真に更新されたとは限らない。したがって *update_direct_s_req* の wwe と DTAG:wwait がともに 1^T であるような dw がある場合にのみ，スレッド起動を行なう。またこのような dw は複数存在する可能性がある。

[注意]

update_direct_s_req の wwe は word 単位の更新情報であり，DTAG が保持する待ちスレッドの有無情報ではない。したがって，MBPL が cluster-memory を更新する際には，DTAG を更新してはならない。

応答受領時の動作

ラインの状態に関わらず，L2C は応答を期待しない。

[†1] globally-shared の場合，pending store が存在することは pending counter が記憶する。

[†2] 非同期更新がなされた場合の遷移的 / 恒久的なデータ矛盾は許容されているので，L2C が遷移的矛盾を生じるような更新操作を行なっているので，更新操作の完了確認前に更新してよい。

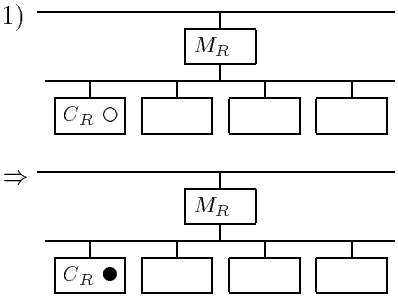
[†3] 要求元クラスタに，全ての subscribed cluster を包含するようなマップが存在することを仮定している。

[†4] $\texttt{ack} = 0^F$ であれば更新要求の完了確認をしないような最適化を行なってもよい。

[†5] 書込待ちスレッドの情報をどのようにして取得し，またどのようにして起動するかについては，本資料では一切規定しない。

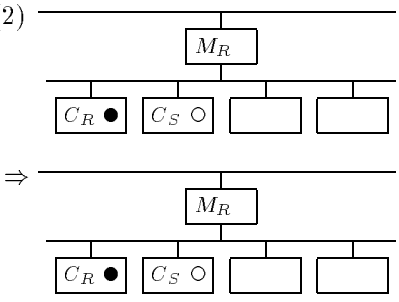
パケットの流れと状態遷移

(1)



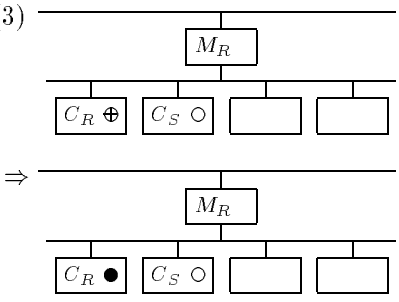
C_R :VSLCN $\xrightarrow{L : update_direct_s_req}$ C_R :VSLCN
⇒ VXLDO

(2)



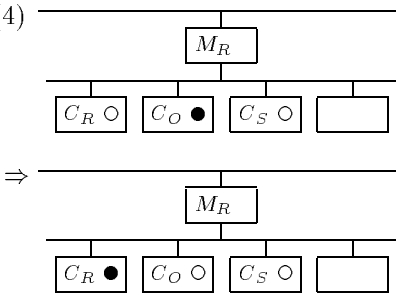
C_R :VSLDO $\xrightarrow{L : update_direct_s_req}$ C_S :VSLCN
⇒ VXLDO

(3)



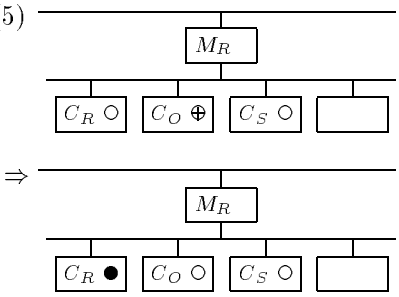
C_R :VSLCO $\xrightarrow{L : update_direct_s_req}$ C_S :VSLCN
⇒ VXLDO

(4)



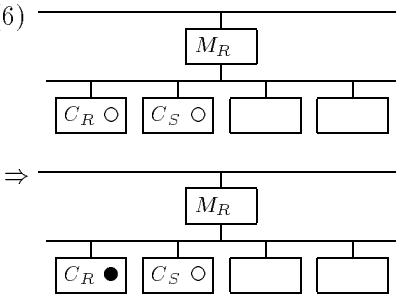
C_R :VSLCN $\xrightarrow{L : update_direct_s_req}$ C_O :VSLDO
⇒ VXLDO ⇒ VSLCN
C_S :VSLCN

(5)



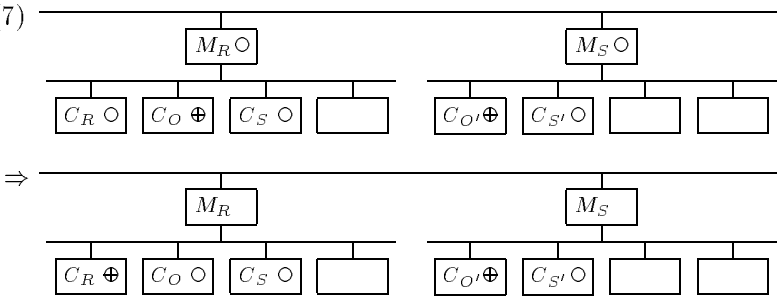
C_R : VSLCN ⇒ VXLDO $\xrightarrow{L : update_direct_s_req}$ C_O : VSLCO ⇒ VSLCN, C_S : VSLCN

(6)



C_R : VSLCN ⇒ VXLDO $\xrightarrow{L : update_direct_s_req}$ C_S : VSLCN

(7)



C_R : VSGCN ⇒ VSGCO $\xrightarrow{L : update_direct_s_req}$ M_R : VG, C_O : VSGCO ⇒ VSGCN, C_S : VSGCN $\xrightarrow{G^m : update_req}$ M_S : VG $\xrightarrow{L : update_req}$

M_S : VG, $C_{O'}$: VSGCO, $C_{S'}$: VSGCN $\xrightarrow{G^c : update_ack}$ M_R : VG

3.9 *no_alloc_write_req* ← **s-write** (dummy-page, cache-miss)

no_alloc_write_req は，dummy-page 対する **s-write** が cache-miss したときの，更新要求である[†1]。

送出される要求パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1100/100	= *no_alloc_write_req*
size	= 110	= masked block
ack	= *ack*	= L2CCTL.wacks
did	= 111	= all
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= ⊥	
fe	= *sync*	= ⊥/¬mcmd[0]
addr	= *addr*	= VBUS.addr ($[5:4]=0$)
wwe[0:3]	= *mdfd*	= word modified
data[0:3]	= *data*	= ⊥/VBUS.data

[注意]

fe の値は wwe = $\{0^F 0^F\}$ であるような dw に対して無意味である。

他の L2C の動作

対象ラインを所有している L2C は，ラインの中で wwe[i] = 1^T であるような word のみを更新する。なお，この更新操作はラインが globally-shared である場合にも行なわれる。一般に，update-direct は更新要求がシステム内で非同期的に並行して行なわれることを想定しておらず，万一非同期更新がなされた場合の遷移的あるいは恒久的なデータの矛盾は許容される。

また owner であるような L2C が存在する場合，その ownership は放棄されない。

MMC/MBPL の動作

MMC は ack = 1^T であれば pending counter を increment して無条件に MBPL に割出す。

MBPL は clsh = 0^F であれば，単に pending counter を decrement して完了する[†2]。

clsh = 1^T であれば，対象ラインは dummy-page に含まれ cluster-memory には存在しないため，単にラインを所有している全てのクラスタに対して更新要求を送出する[†3]。要求の宛先であるような全てのクラスタから更新要求に対する完了確認を得た後，ack = 1^T であれば pending counter を decrement して完了する[†4]。

[注意]

更新要求を受理したクラスタがラインを local に所有している場合は，CBUS に対して (*update_req* ではなく) *no_alloc_write_req* を送出しなければならない。また cluster-memory を更新する場合には，fe の値が wwe = $\{0^F 0^F\}$ であるような dw に対して無意味であることに注意しなければならない。

応答受領時の動作

ラインの状態に関わらず，L2C は応答を期待しない。

[†1] *no_alloc_write_req* は [3] では *write_xa_req* と記載されている。

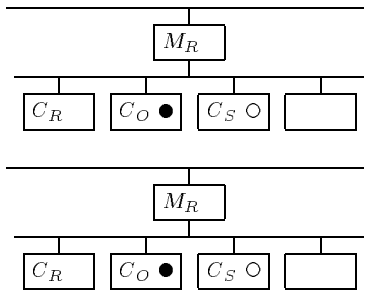
[†2] 想定は困難であるが，dummy-page のラインが locally-shared であることもあろう。

[†3] 要求元クラスタに，全ての subscribed cluster を包含するようなマップが存在することを仮定している。

[†4] ack = 0^F であれば更新要求の完了確認をしないような最適化を行なってもよい。

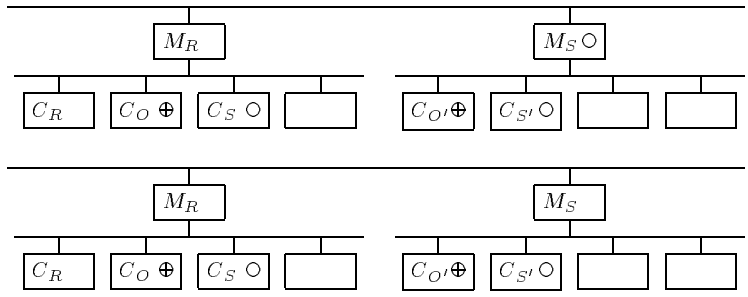
パケットの流れと状態遷移

(1)



C_R :I**** $\xrightarrow{L : no_alloc_write_req}$ C_O :VSLDO
C_S :VSLCN

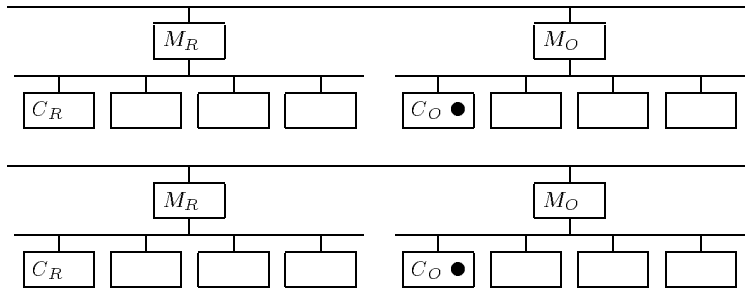
(2)



C_R :I**** $\xrightarrow{L : no_alloc_write_req}$ M_R:I* $\xrightarrow{G^m : no_alloc_write_req}$ M_S :VG $\xrightarrow{L : update_req}$
C_O :VSGCO
C_S :VSGCN

M_S :VG $\xrightarrow{G^e : no_alloc_write_ack}$ M_R:I*
$C_{O'}$:VSGCO
$C_{S'}$:VSGCN

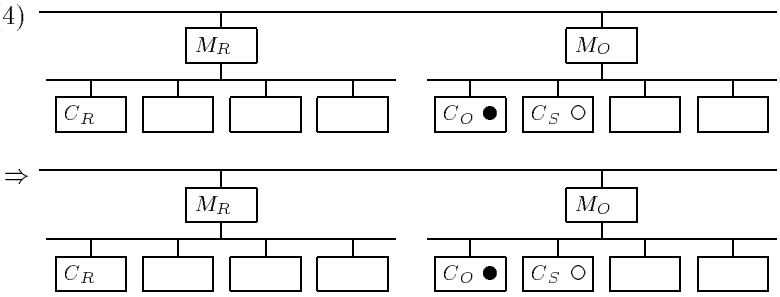
(3)



C_R :I**** $\xrightarrow{L : no_alloc_write_req}$ M_R:I* $\xrightarrow{G : no_alloc_write_req}$

M_O :VL $\xrightarrow{L : no_alloc_write_req}$ M_O :VL $\xrightarrow{G^e : no_alloc_write_ack}$ M_R:I*
C_O :VXLDO

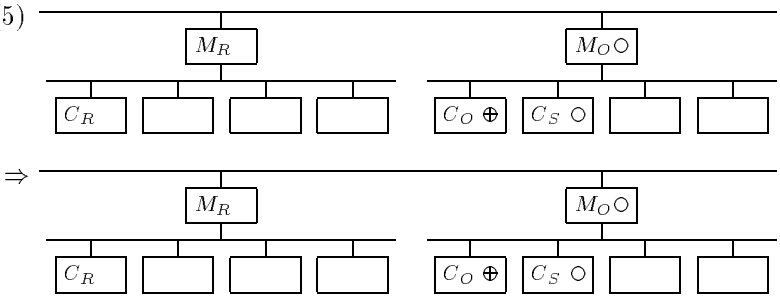
(4)



C_R : `I****` $\xrightarrow{L : no_alloc_write_req}$ M_R : `I*` $\xrightarrow{G : no_alloc_write_req}$

M_O : `VL` $\xrightarrow{L : no_alloc_write_req}$ M_O : `VL` $\xrightarrow{G^c : no_alloc_write_ack}$ M_R : `I*`
C_O : `VSLDO`
C_S : `VSLCN`

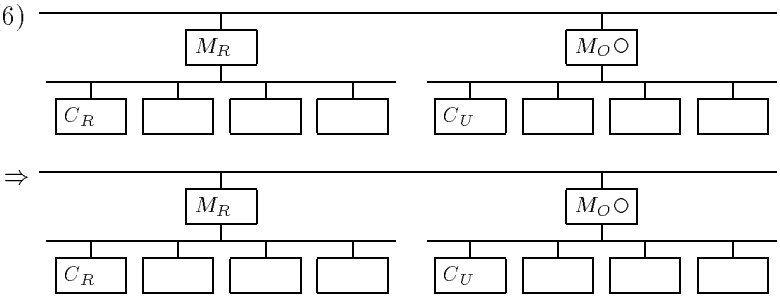
(5)



C_R : `I****` $\xrightarrow{L : no_alloc_write_req}$ M_R : `I*` $\xrightarrow{G : no_alloc_write_req}$

M_O : `VL` $\xrightarrow{L : no_alloc_write_req}$ M_O : `VL` $\xrightarrow{G^c : no_alloc_write_ack}$ M_R : `I*`
C_O : `VSLCO`
C_S : `VSLCN`

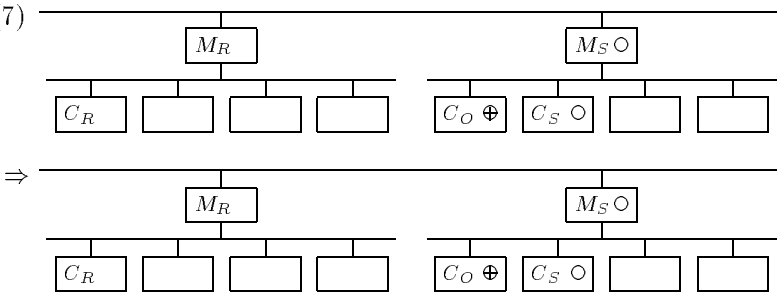
(6)



C_R : `I****` $\xrightarrow{L : no_alloc_write_req}$ M_R : `I*` $\xrightarrow{G : no_alloc_write_req}$

M_O : `VL` $\xrightarrow{L : no_alloc_write_req}$ M_O : `VL` $\xrightarrow{G^c : no_alloc_write_ack}$ M_R : `I*`

(7)



C_R : I**** $\xrightarrow{L : no_alloc_write_req}$ M_R : I* $\xrightarrow{G^m : no_alloc_write_req}$ M_S : VG $\xrightarrow{L : update_req}$
C_O : VSGCO
C_S : VSGCN

M_S : VG $\xrightarrow{G^c : no_alloc_write_ack}$ M_R : I*
C_O : VSGCO
C_S : VSGCN

3.10 *update_via_home_q_req* ← q-write

update_via_home_q_req は，**q-write** による dw-queue への enqueue 要求である。**q-write** は対応する **q-read**, **p-read**, **q-check** とは異なり，常に L2-non-cachable として取り扱われる。したがって対象ラインの状態に関わらず[†1]，**q-write** に対しては *update_via_home_q_req* が CBUS に送出される。

送出される要求パケット

`plen`	$= 01$	$= 2$ flit
`injmask`	$= 0$	= nop
`ccbl`	$= 0$	= non cachable
`pcmd/scmd`	$= 1000/110$	$=$ *update_via_home_q_req*
`size`	$=$ *size*	= accessed size
`ack`	$=$ *ack*	= `L2CCTL.wackq`
`did`	$= 100$	= MMC/MBPL
`sid`	$=$ *pid*	= own ID
`drty`	$= \perp$	
`clsh`	$= 0$	
`fe`	$= \perp$	
`addr`	$=$ *addr*	= `VBUS.addr`
`wwe[0]`	$= \perp$	
`data[0]`	$=$ *data*	= `VBUS.data`

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。一般に **q-write** の対象ラインが他の L2C に共有されていることは想定されておらず，万一共有されている場合には dw-queue の動作は保証されない。

MMC/MBPL の動作

MMC は `ack` $= 1^T$ であれば pending counter を increment して無条件に MBPL に割出す。

MBPL は，[6] に示す方法で enqueue 操作を行ない，`ack` $= 1^T$ であれば操作完了確認後に pending counter を decrement する。

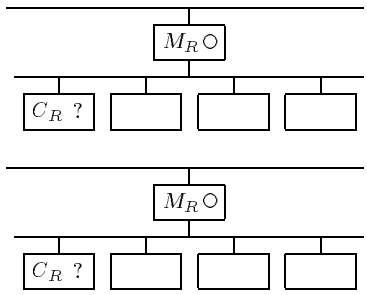
応答受領時の動作

L2C は応答を期待しない。

[†1] ラインが L2C に存在すること自体が想定されていない。

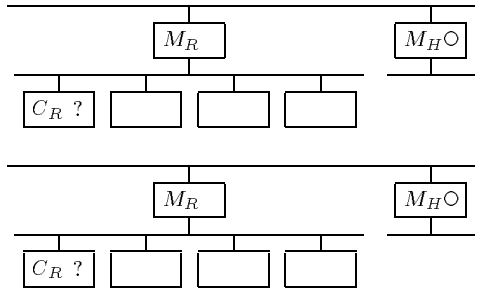
パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L : update_via_home_q_req}$ M_R:VL

(2)



C_R :????? $\xrightarrow{L : update_via_home_q_req}$ M_R:I* $\xrightarrow{G : update_via_home_q_req}$ M_H:VG

3.11 *bq_write_req* ← **bq-write** (cache-hit, last-byte)

bq_write_req は，**bq-write** が cache-hit し，かつ対象ラインの末尾バイトが更新された場合の，block-queue への enqueue 要求である[†1]。

送出される要求パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1100/101	= *bq_write_req*
size	= 100	= block
ack	= *ack*	= L2CCTL.wackbq
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= VBUS.addr
wwe[0:3]	= {*w-wait*, 0}∗4	= {FMP : ww, 0}∗4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

他の L2C の動作

対象ラインを所有している L2C は，ラインの無効化を行なう。ただし。一般に **bq-write** の対象ラインが他の L2C に共有されていることは想定されておらず，万一共有されている場合には block-queue の動作は保証されない。

MMC/MBPL の動作

MMC は ack = 1^T であれば pending counter を increment して無条件に MBPL に割出す。

MBPL は，[7] に示す方法で enqueue 操作を行ない，ack = 1^T であれば操作完了確認後に pending counter を decrement する。

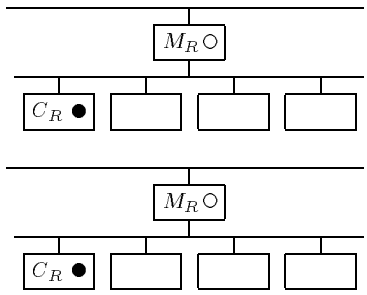
応答受領時の動作

ラインの状態に関わらず，L2C は応答を期待しない。

[†1] *bq_write_req* は [3] では *write_xb_req* と記載されている。

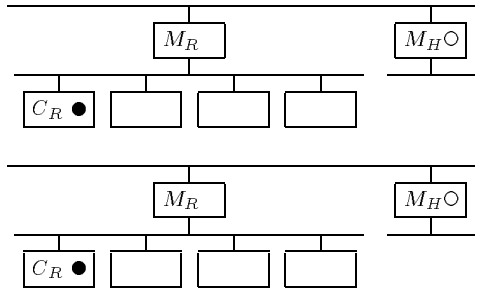
パケットの流れと状態遷移

(1)



C_R :VXLDO $\xrightarrow{L : bq_write_req}$ M_R:VL

(2)



C_R :VXLDO $\xrightarrow{L : bq_write_req}$ M_R:I* $\xrightarrow{G : bq_write_req}$ M_H:VG

4 L2-cachable swap

L2-cachable swap とは CPU が行なう swap 要求 (すなわち read でも write でもないメモリ・アクセス要求) のうち，control-space access ではなく，かつ L2-cachable であるものをいう。

L2-cachable swap が許される memory-command は表 4.1 に示すものであり [2]，page table entry の指定が L1-cachable であれば (またその時に限り) L2-cachable である。

L2-cachable swap によって生成されるパケットは，表 4.1 に示すものであり，一般には exclusive-hit でない場合に生じる。

表 4.1: L2-cachable swap によるパケット生成

mcmd	type	coh-pol	pcmd/scmd	section
0111	$\{0,1\}$	update	*swap_req*	4.2節
1001	$\{0,1,2\}$	invalidate	*rdinv_req*	4.1節
1010	—	invalidate	*rdinv_req*	4.1節
1011	—	invalidate	*rdinv_req*	4.1節
1100	—	no-coherent	*n_c_rd_req*	4.3節
1101	—	invalidate	*rdinv_req*	4.1節
1110	—	update	*swap_req*	4.2節
1111	—	update	*swap_req*	4.2節

4.1 *rdinv_req* ← **normal-swap** (coherence-policy(s) = invalidate, ¬exclusive-hit)

rdinv_req は，swap の coherence-policy が invalidate である **normal-swap** が exclusive-hit しなかったときの，無効化を伴う block-load 要求である。

生成される要求パケット，他の L2C の動作，MMC/MBPL の動作については，2.3 節を参照せよ。また cache-miss した場合のパケットの流れと状態遷移についても，要求元のラインの最終的な状態が必ず `VXLDO` となる点を除き，同節と同様である。

応答受領時の動作

要求元の L2C は，*rdinv_req* が CBUS に送出される時点でラインが locally-shared であり，かつ自身が owner であることを認知すると，応答を待たずに S = `VXLDO` に遷移して，CPU に L2C が保持する対象 word/byte の値を返した上で，それらを更新する[†1]。

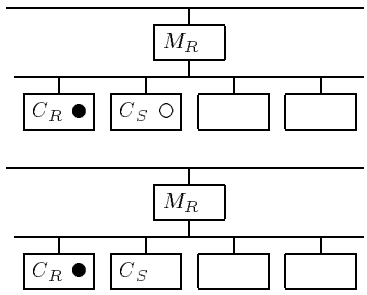
そうでなければ応答 *rdinv_reply* を受けとり，無条件に S = `VXLDO` に遷移し[†2]，CPU に応答された `data[dwix]` を返した上で，L2C の対象 word/byte を更新する。

[†1] 自身が owner でありかつ `CBUS:x_crply` $= 0^T$ とするので，他の L2C も MMC も応答することはない。

[†2] 応答の `clsh` は 0^F でなければならない。

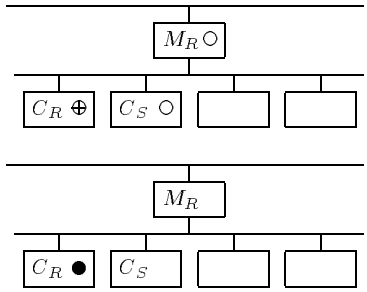
パケットの流れと状態遷移

(1)



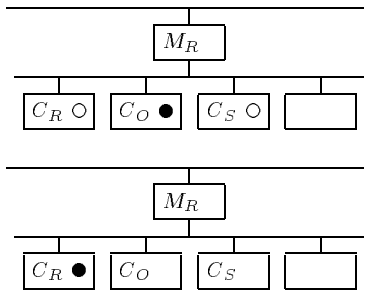
C_R :VSLDO $\xrightarrow{L : rdinv_req}$ C_R :VSLDO
⇒ VXLDO
C_S :VSLCN
⇒ I****

(2)



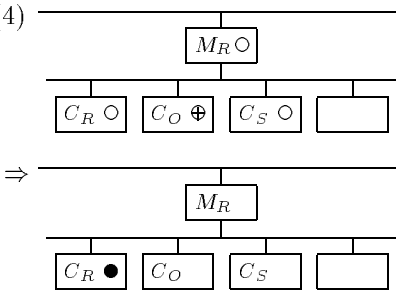
C_R :VSLCO $\xrightarrow{L : rdinv_req}$ C_R :VSLCO
⇒ VXLDO
C_S :VSLCN
⇒ I****

(3)



C_R :VSLCN $\xrightarrow{L : rdinv_req}$ C_O :VSLDO $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :VSLCN
⇒ I**** ⇒ VXLDO
C_S :VSLCN
⇒ I****

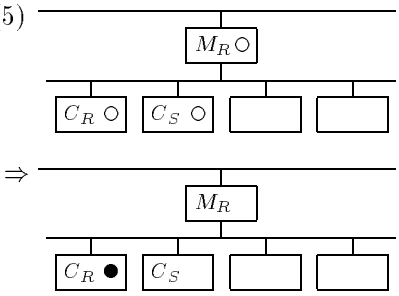
(4)



C_R :VSLCN $\xrightarrow{L : rdinv_req}$ C_O :VSLCO ⇒ I**** $\xrightarrow{L : rdinv_reply(X, L, C)}$ C_R :VSLCN ⇒ VXLDO

C_S :VSLCN ⇒ I****

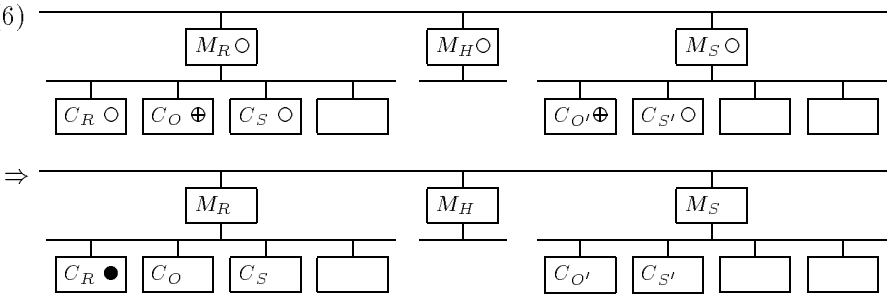
(5)



C_R :VSLCN $\xrightarrow{L : rdinv_req}$ M_R:VL $\xrightarrow{L : rdinv_reply(X, L, C)}$ C_R :VSLCN ⇒ VXLDO

C_S :VSLCN ⇒ I****

(6)



C_R :VSGCN $\xrightarrow{L : rdinv_req}$ M_R:VG $\xrightarrow{G : rdinv_req}$ M_H:VG $\xrightarrow{G^m : invalidate_req}$

C_O :VSGCO ⇒ I**** C_S :VSGCN ⇒ I****

M_S :VG $\xrightarrow{L : invalidate_req}$ M_S :VG ⇒ I* $\xrightarrow{G^c : invalidate_ack}$ M_H:VG ⇒ I* $\xrightarrow{G : rdinv_reply}$

$C_{O'}$:VSGCO ⇒ I**** $C_{S'}$:VSGCN ⇒ I****

M_R:VG ⇒ VL $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :VSGCN ⇒ VXLDO

4.2 *swap_req* ← **normal-swap**(coherence-policy(s) = update, ¬exclusive-hit)

swap_req は，swap の coherence-policy が update であるような **normal-swap** が exclusive-hit しなかったときの，swap 要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0000/011	= *swap_req*
size	= *size*	= accessed size (00:byte/10:word)
ack	= 0	= nop
did	= 111	= all
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= VBUS.addr
wwe[0]	= ⊥	
data[0]	= *data*	= VBUS.data

他の L2C の動作

対象ラインを owner として所有している L2C があり，かつ共有状態が globally-shared でなければ (S = V*L*0)，その L2C が CBUS:x_crply = 0^T として応答意志を示し，FMP/DAR を読出して以下の *swap_reply* を応答し，対象の word または byte を更新して，S = VSLCN に遷移する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/011	= *swap_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= *pid*	= own ID
drty	= *dirty*	= FMP:state.d
clsh	= 0	= ¬(FMP : state.l = 1)
fe	= *sync*	= FMP:sync
addr	= *addr*	= requested address
wwe[0:3]	= {*w-wait*, 0}*4	= {FMP : ww, 0}*4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

対象ラインを所有しているが上記の条件を満たさない L2C は，*swap_req* を update 系プロトコルによる更新要求と解釈し，対象の word または byte を更新する。なお，この更新操作はラインが globally-shared である場合にも行なわれる。したがって更新要求がシステム内で非同期的に並行して行なわれた場合，遷移的に矛盾したデータが観測されることがある。ただしシステム内で並行して行なわれる swap については，その旧値 / 新値のシーケンスに矛盾が生じることはない。

また globally-shared なラインの owner は，ownership を放棄する。

MMC/MBPL の動作

対象ラインが globally-shared であるか owner として所有している L2C が存在しなければ (CBUS:x_crply = 1^F)，MMC は対象ラインの STAG をアクセスする。その結果 STAG:{valid|global} = $\{1^T 0^F\}$ であれば，cluster-memory を読出して以下の *swap_reply* を応答する[†1][†2]。

[†1] [1] では無条件に MBPL に割出すとあるが，これは誤りである。

[†2] rd_trap / wr_trap のチェックに関しては，どのように実装されているか定かではない。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/011	= *swap_reply*
size	= 100	= block
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= 0	= clean
clsh	= *global*	= STAG.global
fe	= *sync*	= STAG.fe
addr	= *addr*	= requested address
wwe[0:3]	= *w-wait/r-wait*	= DTAG[dwix..dwix+3]
data[0:3]	= *data*	= DRAM[dwix..dwix+3]

直接応答が不可能な場合には MBPL によって処理され，home-cluster に swap 操作と他のクラスタに対する更新を要求するなどして，最終的には上記のパケットが応答される。

[注意]

(1) globally-shared の場合，STAG:valid $= 1^T$ であっても swap 操作を home-cluster で serialize する必要があり，かつ L2C には serialize された時点でのラインの値を返す必要がある。したがって，MBPL は globally-shared の場合には常に home-cluster に対して *swap_req* を送出してラインの値を得る必要がある。

(2) MBPL が globally-shared なラインを取得して応答する場合，必ずラインの内容を cluster-memory に書込んで対象の word または byte を更新した上で，clsh $= 1^T$, drty $= 0^F$ として応答しなければならない。

(3) *swap_req* の fe と wwe は無意味であるため，MBPL が cluster-memory を更新する際に STAG:fe と DTAG を更新してはならない。

応答受領時の動作

要求元の L2C は，*swap_req* が CBUS に送出される時点でラインが locally-shared であり，かつ自身が owner であることを認知すると，応答を待たずに $S =$ VXLDO に遷移し，CPU に L2C が保持する対象 word/byte の値を返した上で，それらを更新する[†1]。

そうでなければ応答 *swap_reply* を受けとり，応答が送出された際の CBUS:x_msh $\equiv x$ と，応答パケットの clsh $\equiv$ g に従って $S =$ VxgdO に遷移し，CPU に応答された data[dwix] を返した上で，L2C の対象 word/byte を更新する。ただし ；

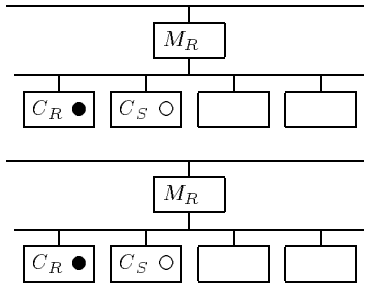
$$\mathtt{x} = \begin{cases} \mathtt{X} & x = 1^F \wedge \mathtt{g} = 0^F \\ \mathtt{S} & \text{otherwise} \end{cases}$$
$$\mathtt{d} = \begin{cases} \mathtt{D} & \mathtt{g} = 0^F \\ \mathtt{C} & \text{otherwise} \end{cases}$$

である。したがって，L2C が応答した場合には応答者自身が CBUS:x_msh $= 0^T$ とするので必ず shared となり，MMC/MBPL が応答した場合には他のクラスタと L2C の共有状況によって shared または exclusive となる。また globally-shared でなければ dirty となる。

[†1] 自身が owner でありかつ CBUS:x_crply $= 0^T$ とするので，他の L2C も MMC も応答することはない。

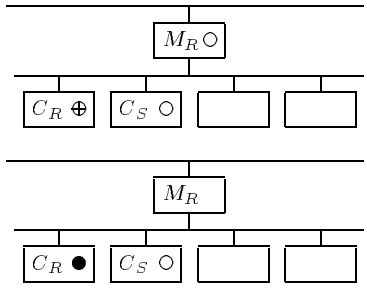
パケットの流れと状態遷移

(1)



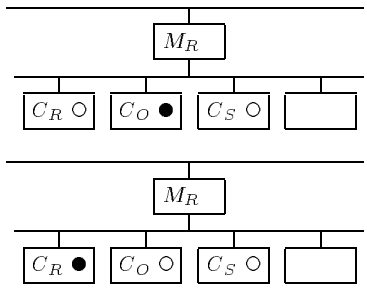
$$C_R : \texttt{VSLDO} \xrightarrow{L : swap_req} \begin{array}{l} C_R : \texttt{VSLDO} \\ \Rightarrow \texttt{VXLDO} \\ C_S : \texttt{VSLCN} \end{array}$$

(2)



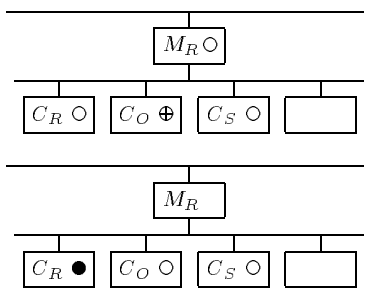
$$C_R : \texttt{VSLCO} \xrightarrow{L : swap_req} \begin{array}{l} C_R : \texttt{VSLCO} \\ \Rightarrow \texttt{VXLDO} \\ C_S : \texttt{VSLCN} \end{array}$$

(3)



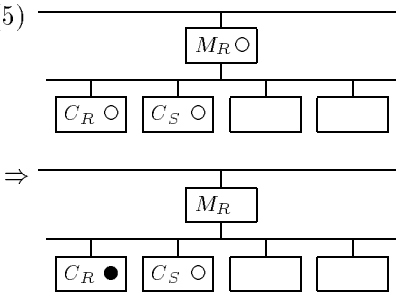
$$C_R : \texttt{VSLCN} \xrightarrow{L : swap_req} \begin{array}{l} C_O : \texttt{VSLDO} \\ \Rightarrow \texttt{VSLCN} \\ C_S : \texttt{VSLCN} \end{array} \xrightarrow{L : swap_reply(X, L, D)} \begin{array}{l} C_R : \texttt{VSLCN} \\ \Rightarrow \texttt{VXLDO} \end{array}$$

(4)



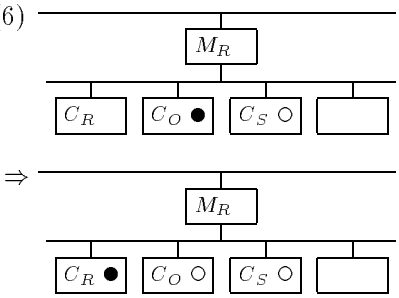
$$C_R : \texttt{VSLCN} \xrightarrow{L : swap_req} \begin{array}{l} C_O : \texttt{VSLCO} \\ \Rightarrow \texttt{VSLCN} \\ C_S : \texttt{VSLCN} \end{array} \xrightarrow{L : swap_reply(X, L, C)} \begin{array}{l} C_R : \texttt{VSLCN} \\ \Rightarrow \texttt{VXLCO} \end{array}$$

(5)



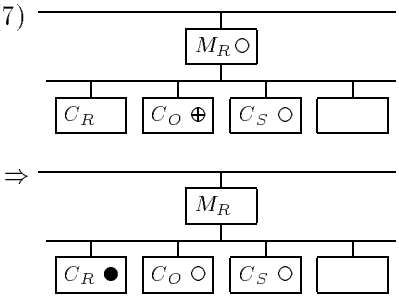
C_R : `VSLCN` $\xrightarrow{L : swap_req}$ M_R : `VL` $\xrightarrow{L : swap_reply(X, L, C)}$ C_R : `VSLCN` ⇒ `VXLDO`
C_S : `VSLCN`

(6)



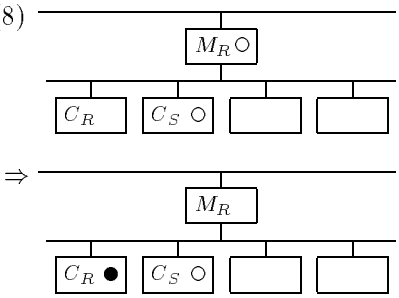
C_R : `I****` $\xrightarrow{L : swap_req}$ C_O : `VSLDO` ⇒ `VSLCN` $\xrightarrow{L : swap_reply(S, L, D)}$ C_R : `I****` ⇒ `VSLDO`
C_S : `VSLCN`

(7)



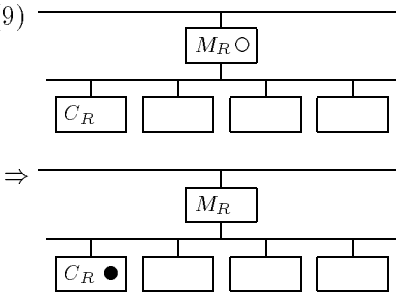
C_R : `I****` $\xrightarrow{L : swap_req}$ C_O : `VSLCO` ⇒ `VSLCN` $\xrightarrow{L : swap_reply(S, L, C)}$ C_R : `I****` ⇒ `VSLDO`
C_S : `VSLCN`
M_R : `VL`

(8)



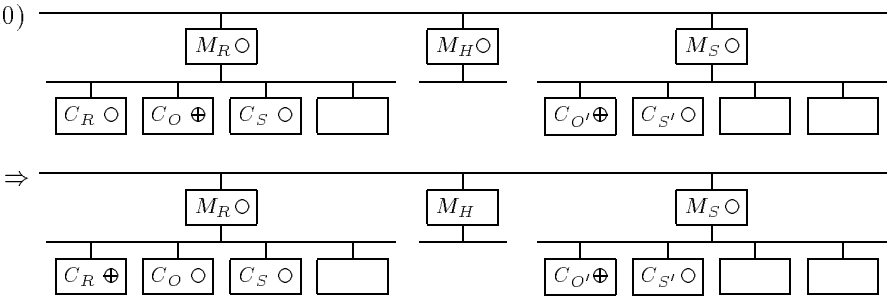
C_R : `I****` $\xrightarrow{L : swap_req}$ M_R : `VL` $\xrightarrow{L : swap_reply(S, L, C)}$ C_R : `I****` ⇒ `VSLDO`
C_S : `VSLCN`

(9)



C_R : I**** $\xrightarrow{L : swap_req}$ M_R : VL $\xrightarrow{L : swap_reply(X, L, C)}$ C_R : I****
$\Rightarrow$ VXLDO

(10)

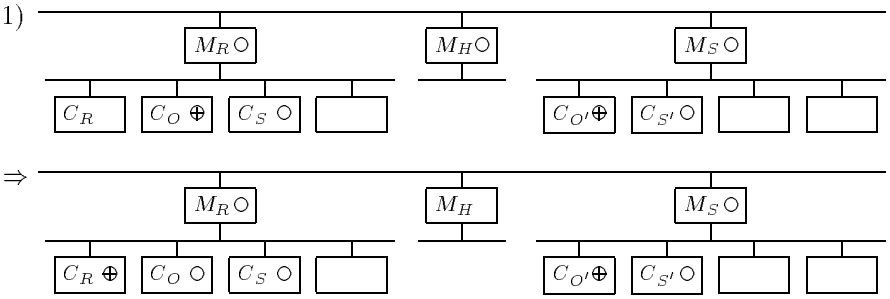


C_R : VSGCN $\xrightarrow{L : swap_req}$ M_R : VG $\xrightarrow{G : swap_req}$ M_H : VG $\xrightarrow{G^m : update_req}$
C_O : VSGCO
$\Rightarrow$ VSLCN
C_S : VSGCN
$\Rightarrow$ VSLCN

M_S : VG $\xrightarrow{L : update_req}$ M_S : VG $\xrightarrow{G^c : update_ack}$ M_H : VG $\xrightarrow{G : swap_reply}$
$C_{O'}$: VSGCO
$C_{S'}$: VSGCN

M_R : VG $\xrightarrow{L : swap_reply(X, L, D)}$ C_R : VSGCN
$\Rightarrow$ VSGCO

(11)



C_R : I**** $\xrightarrow{L : swap_req}$ M_R : VG $\xrightarrow{G : swap_req}$ M_H : VG $\xrightarrow{G^m : update_req}$

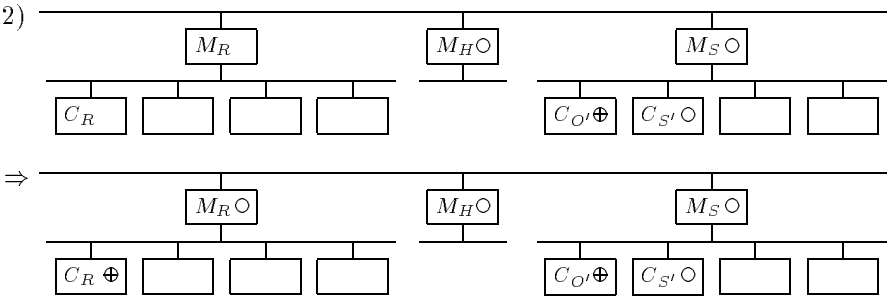
C_O : VSGCO
⇒ VSLCN
C_S : VSGCN
⇒ VSLCN

M_S : VG $\xrightarrow{L : update_req}$ M_S : VG $\xrightarrow{G^c : update_ack}$ M_H : VG $\xrightarrow{G : swap_reply}$

$C_{O'}$: VSGCO
$C_{S'}$: VSGCN

M_R : VG $\xrightarrow{L : swap_reply(X, L, D)}$ C_R : I****
⇒ VSGCO

(12)



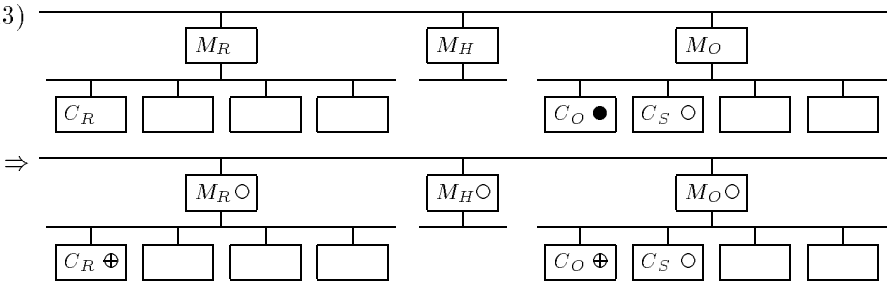
C_R : I**** $\xrightarrow{L : swap_req}$ M_R : I* $\xrightarrow{G : swap_req}$ M_H : VG $\xrightarrow{G^m : update_req}$

M_S : VG $\xrightarrow{L : update_req}$ M_S : VG $\xrightarrow{G^c : update_ack}$ M_H : VG $\xrightarrow{G : swap_reply}$

$C_{O'}$: VSGCO
$C_{S'}$: VSGCN

M_R : I*
⇒ VG
$\xrightarrow{L : swap_reply(X, L, D)}$ C_R : I****
⇒ VSGCO

(13)



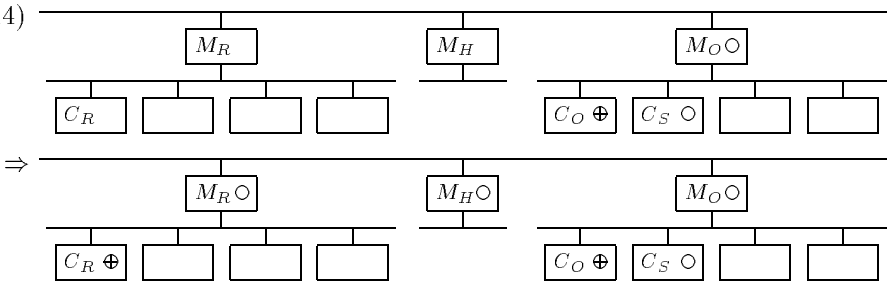
C_R :I**** $\xrightarrow{L:\,swap_req}$ M_R:I* $\xrightarrow{G:\,swap_req}$ M_H:I* $\xrightarrow{G:\,swap_req}$ M_O:VL $\xrightarrow{L:\,swap_req}$

C_O :VSLDO $\xrightarrow{L:\,swap_reply(S,L,D)}$ M_O:VL $\xrightarrow{G:\,swap_reply}$ M_H:I* $\xrightarrow{G:\,swap_reply}$
⇒VSGCO ⇒VG ⇒VG

C_S :VSLCN
⇒VSGCN

M_R:I* $\xrightarrow{L:\,swap_reply(X,G,C)}$ C_R :I****
⇒VG ⇒VSGCO

(14)



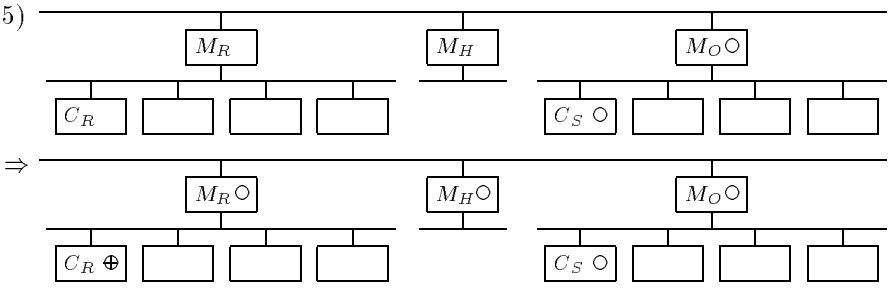
C_R :I**** $\xrightarrow{L:\,swap_req}$ M_R:I* $\xrightarrow{G:\,swap_req}$ M_H:I* $\xrightarrow{G:\,swap_req}$ M_O:VL $\xrightarrow{L:\,swap_req}$

C_O :VSLCO $\xrightarrow{L:\,swap_reply(S,L,C)}$ M_O:VL $\xrightarrow{G:\,swap_reply}$ M_H:I* $\xrightarrow{G:\,swap_reply}$
⇒VSGCO ⇒VG ⇒VG

C_S :VSLCN
⇒VSGCN

M_R:I* $\xrightarrow{L:\,swap_reply(X,G,C)}$ C_R :I****
⇒VG ⇒VSGCO

(15)

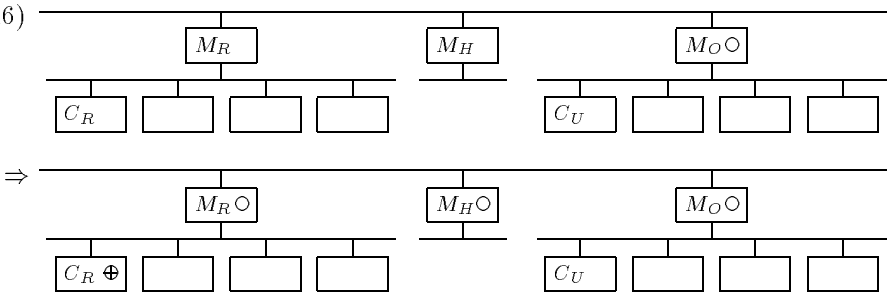


C_R :I**** $\xrightarrow{L:\,swap_req}$ M_R:I* $\xrightarrow{G:\,swap_req}$ M_H:I* $\xrightarrow{G:\,swap_req}$ M_O:VL $\xrightarrow{L:\,swap_req}$

M_O:VL $\xrightarrow{L:\,swap_reply(S,L,C)}$ M_O:VL $\xrightarrow{G:\,swap_reply}$ M_H:I* $\xrightarrow{G:\,swap_reply}$
C_S :VSLCN ⇒VG ⇒VG
⇒VSGCN

M_R:I* $\xrightarrow{L:\,swap_reply(X,G,C)}$ C_R :I****
⇒VG ⇒VSGCO

(16)



C_R :I**** $\xrightarrow{L : swap_req}$ M_R:I* $\xrightarrow{G : swap_req}$ M_H:I* $\xrightarrow{G : swap_req}$ M_O:VL $\xrightarrow{L : swap_req}$

M_O:VL $\xrightarrow{L : swap_reply(X, L, C)}$ M_O:VL ⇒VG $\xrightarrow{G : swap_reply}$ M_H:I* ⇒VG $\xrightarrow{G : swap_reply}$

M_R:I* ⇒VG $\xrightarrow{L : swap_reply(X, G, C)}$ C_R :I**** ⇒VSGCO

4.3 $n_c_rd_req \leftarrow$ **normal-swap** (coherence-policy(s) = no-coherent, cache-miss)

n_c_read_req は，swap の coherence-policy が no-coherent であるような **normal-swap** が cache-miss したときの，スヌープを伴わない block-load 要求である。

生成される要求パケット，他の L2C の動作，MMC/MBPL の動作，パケットの流れと状態遷移については，2.4 節を参照せよ。

応答受領時の動作

応答 *n_c_read_reply* の `clsh` に基づき ；

$$S = \begin{cases} \texttt{VXLDN} & \text{if } \texttt{clsh} = 0^F \\ \texttt{VSGDN} & \text{otherwise} \end{cases}$$

に遷移し，CPU に応答された `data[dwix]` を返した上で，L2C の対象 word/byte を更新する。

なお上記の `VXLDN` は no-coherent であるようなラインに特有の例外的状態であり，書き戻し義務があるが応答義務はない。また `VSGDN` も特有の例外的状態であり，L2C から除去される際に global-writeback を引き起こす。これらの状態にあるラインに対して，no-coherent ではないようなアクセスを行なった場合の結果は一切保証されない。

5 L2-cachable-prefetch/flush

L2-cachable prefetch/flush とは，CPU が行なう write 要求（すなわち read でも swap でもないメモリ・アクセス要求）のうち，control-space access ではなく，かつ L2-cachable であるものの中で，表 5.1 に示した memory-command に関する hw の store である [2]。

表 5.1 に示すものは，page table entry の L1-cachability の指定によって同表の C 欄に示す以下の条件を加味した上で，L2-cachability が定まる。

- y/n
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば（またその時に限り）L2-cachable である。
- y/n(S)
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば L2-cachable である。L1-cachable でなければ，L2-cachability は `L2CCTL.l2ccbl` に従う。
- no(S)
  page table の指定は L1-cachable であってはならず，L2-cachability は `L2CCTL.l2ccbl` に従う。

L2-cachable prefetch/flush によって生成されるパケットは，表 5.1 に示すものであり，以下の 2 種類に分類される。

- $(D = 0) = 1^T \rightarrow$ **prefetch**
  store data の上位バイトが 0 である場合。対象ラインに関する prefetch 要求であり，cache-miss の際に memory-command が定める読出の coherence-policy に従って *read_req*（2.1 節），*allread_req*（2.2 節），*rdinv_req*（2.3 節），または *n_c_rd_req*（2.4 節）が送出される。これらのパケットに関する動作などは，それぞれの節を参照せよ。
- $(D = 0) = 0^F \rightarrow$ **pf/cf**(**M**)
  store data の上位バイトの下位 7 ビットが全て 0 ではない場合。MBPL が行なう prefetch /flush 操作の起動要求と解釈され，ラインの状態に関わらず *n_c_write_m_req* が送出される。

なお store data の上位バイトが `0x80` である場合には，対象ラインの無効化が行なわれ（**cflush**），CBUS への要求パケット送出は行なわれない[†1]。

[†1] 10.2 節に示すように疑似応答パケット *nop* が送出される。

表 5.1: L2-cachable prefetch/flush によるパケット生成

mcmd	type	size	C	$D=0$	mnem	pcmd/scmd	section
0100	$\{0,1\}$	*hw*	no(S)	0^F	**prefetch** (normal)	*read_req*	2.1節
0100	$\{0,1\}$	*hw*	no(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0100	3	*hw*	y/n(S)	0^F	**prefetch** (normal)	*read_req*	2.1節
0100	3	*hw*	y/n(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0101	$\{0,1\}$	*hw*	no(S)	0^F	**prefetch** (normal)	*read_req*	2.1節
0101	$\{0,1\}$	*hw*	no(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0101	3	*hw*	y/n(S)	0^F	**prefetch** (normal)	*read_req*	2.1節
0101	3	*hw*	y/n(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0110	2	*hw*	no(S)	0^F	**prefetch** (all-read)	*allread_req*	2.2節
0110	2	*hw*	no(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0110	3	*hw*	y/n(S)	0^F	**prefetch** (normal)	*read_req*	2.1節
0110	3	*hw*	y/n(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0111	$\{0,1\}$	*hw*	no(S)	0^F	**prefetch** (all-read)	*allread_req*	2.2節
0111	$\{0,1\}$	*hw*	no(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0111	2	*hw*	y/n	0^F	**prefetch** (all-read)	*allread_req*	2.2節
0111	2	*hw*	y/n	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
0111	3	*hw*	y/n(S)	0^F	**prefetch** (normal)	*read_req*	2.1節
0111	3	*hw*	y/n(S)	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
1010	—	*hw*	y/n	0^F	**prefetch** (normal)	*read_req*	2.1節
1010	—	*hw*	y/n	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
1011	—	*hw*	y/n	0^F	**prefetch** (read-invalidate)	*rdinv_req*	2.3節
1011	—	*hw*	y/n	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
1100	3	*hw*	y/n	0^F	**prefetch** (no-coherent)	*n_c_rd_req*	2.4節
1100	3	*hw*	y/n	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節
1101	—	*hw*	y/n	0^F	**prefetch** (read-invalidate)	*rdinv_req*	2.3節
1101	—	*hw*	y/n	1^T	**pf/cf**(M)	*n_c_write_m_req*	5.1節

5.1 $n_c_write_m_req \leftarrow$ pf/cf(M)

$n_c_write_m_req$ は，**pf/cf**(M) による MBPL への処理要求である。

送出される要求パケット

`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 1010/011	= $n_c_write_m_req$
`size`	= 001	= halfword
`ack`	= 1	= with ack
`did`	= 100	= MMC/MBPL
`sid`	= pid	= own ID
`drty`	= $\perp$	
`clsh`	= $\perp$	
`fe`	= $\perp$	
`addr`	= $addr$	= `VBUS.addr`
`wwe[0]`	= $\perp$	
`data[0]`	= $data$	= `VBUS.data`

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。

MMC/MBPL の動作

MMC は pending counter を increment して無条件に MBPL に割出す。

MBPL の動作については本資料では規定しないが，以下の事項に留意せよ。

(1) MBPL は `size` = 01 (hw) により，(**m-write/t-write** ではなく) **pf/cf**(M) による起動要求であることを知ることができる[†1]。

(2) [2] では，store data の最上位ビット (`data[63]`) が 0 であれば prefetch, 1 であれば flush であると規定されている。

(3) store data は `data[63:48]` であり，例えば操作対象のバイト数などとして参照することができる[†2][†3]。

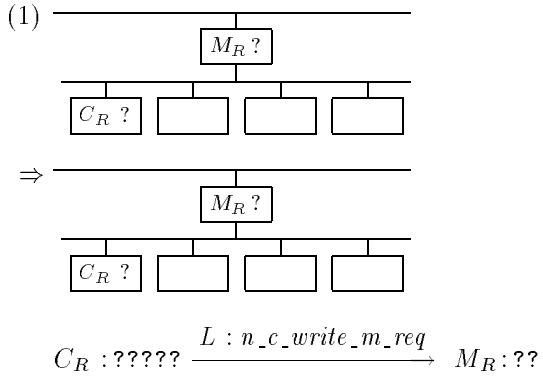
(4) MBPL が prefetch したラインを L2C に与えるためには，$injection_req$ を使用することができる (11.9 節参照)。

(5) MBPL が L2C が所有するラインの無効化を行なうには，$cflush_req$ を使用することができる (11.8 節)。

応答受領時の動作

ラインの状態に関わらず，L2C は応答を期待しない。

パケットの流れと状態遷移



[†1] `ccbl` $= 1^T$ によって判別してもよい。

[†2] dw 境界以外に対する **pf/cf**(M) は，CC によって例外 (bus-error) とされるため，store data は必ず最上位に位置する。

[†3] CC では実装の都合上，store data の下位バイトを無視するが，MBPL には下位バイトも正しく送られる。

6 non-cachable load

non-cachable load とは CPU が行なう read 要求 (すなわち write でも swap でもないメモリ・アクセス要求) のうち，control-space access ではなく，かつ L2-cachable でないものをいう。

non-cachable load が許される memory-command は表 6.1 に示すものであり [2]，かつ page table entry の L1-cachability の指定によって同表の C 欄に示す以下の条件が加味される。

- no
  page table の指定が L1-cachable であってはならず，また無条件に L2-non-cachable である。
- y/n
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば (またその時に限り) L2-cachable である。
- y/n(S)
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば L2-cachable である。L1-cachable でなければ，L2-cachability は `L2CCTL.l2ccbl` に従う。
- no(S)
  page table の指定は L1-cachable であってはならず，L2-cachability は `L2CCTL.l2ccbl` に従う。

non-cachable load によって生成されるパケットは，表 6.1 に示すものである。一般に non-cachable load の対象アドレスに対応するラインを要求元あるいは他の L2C が所有していることは想定されておらず，万一所有している場合の結果は保証されない。

また対象アドレスが L3-cachable であっても，要求パケットには coherence-policy に関する情報が反映されない。したがって MMC/MBPL は，特定の coherence-policy に限定する，あるいは物理ページごとの coherence-policy を管理するなどの方法により，適切なものを選択する必要がある[†1]。

[†1] この節の「パケットの流れと状態遷移」では invalidate を仮定している。

表 6.1: non-cachable load によるパケット生成

mcmd	type	size	C	mnem	pcmd/scmd	section
0000	—	—	no	**normal-read**	*read_req*	6.1節
0010	—	$\neg hw$	no(S)	**q-read**	*rdinv_q_req*	6.2節
0010	—	hw	no(S)	**q-check**	*rdinv_f_req*	6.4節
0011	—	$\neg hw$	no(S)	**p-read**	*rdinv_p_req*	6.3節
0011	—	hw	no(S)	**q-check**	*rdinv_f_req*	6.4節
0100	{0,1}	—	no(S)	**s-read/f-read**	*read_req*	6.1節
0100	2	—	no	**s-read/f-read**	*read_req*	6.1節
0100	3	—	y/n(S)	**s-read/f-read**	*read_req*	6.1節
0101	{0,1}	—	no(S)	**s-read/f-read**	*read_req*	6.1節
0101	2	—	no	**s-read/f-read**	*read_req*	6.1節
0101	3	—	y/n(S)	**s-read/f-read**	*read_req*	6.1節
0110	{0,1}	$\neg hw$	no	**p-read**	*rdinv_p_req*	6.3節
0110	{0,1}	hw	no	**q-check**	*rdinv_f_req*	6.4節
0110	2	—	no(S)	**s-read/f-read**	*read_req*	6.1節
0110	3	—	y/n(S)	**s-read/f-read**	*read_req*	6.1節
0111	$\{0,1\}$	—	y/n	**normal-read**	*read_req*	6.1節
0111	2	—	no(S)	**s-read/f-read**	*read_req*	6.1節
0111	3	—	y/n(S)	**s-read/f-read**	*read_req*	6.1節
1001	$\{0,1,2\}$	—	y/n	**normal-read**	*read_req*	6.1節
1010	—	—	y/n	**normal-read**	*read_req*	6.1節
1100	—	—	y/n	**normal-read**	*read_req*	6.1節
1101	—	—	y/n	**normal-read**	*read_req*	6.1節
1110	—	—	y/n	**normal-read**	*read_req*	6.1節
1111	—	—	y/n	**normal-read**	*read_req*	6.1節

6.1 *read_req* ← normal-read, s-read, f-read

read_req は，non-cachable な **normal-read**, **s-read** または **f-read** によって生成される non-cachable な読出要求である。

送出される要求パケット

`plen`	$= 00$	$= 1$ flit
`injmask`	$= 0$	= nop
`ccbl`	$= 0$	= non cachable
`pcmd/scmd`	$= 0000/000$	$= \mathit{read_req}$
`size`	$= \mathit{size}$	= accessed size
`ack`	$= 0$	= nop
`did`	$= 100$	= MMC/MBPL
`sid`	$= \mathit{pid}$	= own ID
`drty`	$= \perp$	
`clsh`	$= 0$	
`fe`	$= \perp$	
`addr`	$= \mathit{addr}$	= `VBUS.addr`

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

L2C は応答しないので ($\texttt{CBUS:x_crply} = 1^F$), MMC が対象ラインの STAG をアクセスする。その結果 $\texttt{STAG:\{rd_trap|valid\}} = \{0^F 1^T\}$ であれば，cluster-memory を読出して以下の *read_reply* を応答する。

`plen`	$= 01$	$= 2$ flit
`injmask`	$= 0$	= nop
`ccbl`	$= 0$	= non cachable
`pcmd/scmd`	$= 0001/000$	$= \mathit{read_reply}$
`size`	$= \mathit{size}$	= accessed size
`ack`	$= 0$	= nop
`did`	$= \mathit{did}$	= requester's ID
`sid`	$= 100$	= MMC/MBPL
`drty`	$= 0$	= clean
`clsh`	$= \mathit{global}$	= `STAG.global`
`fe`	$= \mathit{sync}$	= `STAG.fe`
`addr`	$= \mathit{addr}$	= requested address
`wwe[0]`	$= \mathit{w\text{-}wait/r\text{-}wait}$	= `DTAG[dwix]`
`data[0]`	$= \mathit{data}$	= `DRAM[dwix]`

直接応答が不可能な場合には MBPL によって処理され，home-cluster にライン取得を要求するなどして，最終的には上記のパケットが応答される。

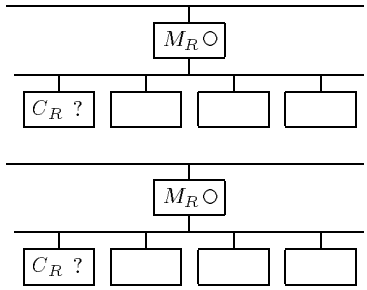
応答受領時の動作

memory-command と *read_reply* の $f \equiv \texttt{fe[dwix]}$ に基づき，以下の値を CPU に返し，L2C の状態は変更しない。

mem. cmd.	f	return value
normal-read	—	`data`
s-read	1^T	`data`
	0^F	undefined-error
f-read	1^T	-1
	0^F	0

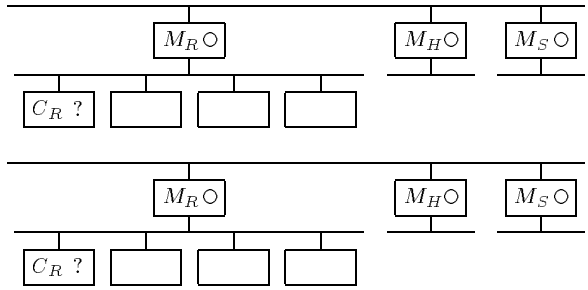
パケットの流れと状態遷移

(1)



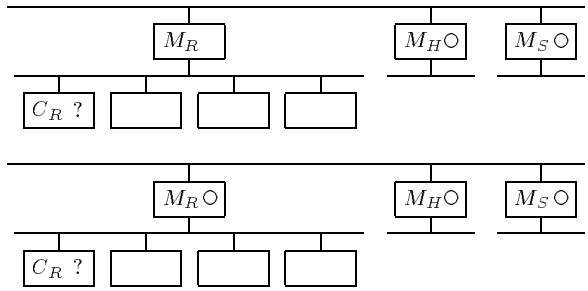
C_R :????? $\xrightarrow{L:read_req}$ M_R:VL $\xrightarrow{L:read_reply}$ C_R :?????

(2)



C_R :????? $\xrightarrow{L:read_req}$ M_R:VG $\xrightarrow{L:read_reply}$ C_R :?????

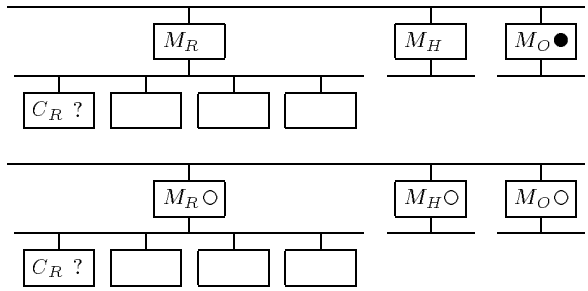
(3)



C_R :????? $\xrightarrow{L:read_req}$ M_R:I* $\xrightarrow{L:read_req}$ M_H:VG $\xrightarrow{G:read_reply}$ M_R:I* ⇒VG $\xrightarrow{L:read_reply}$

C_R :?????

(4)



C_R :????? $\xrightarrow{L:read_req}$ M_R:I* $\xrightarrow{L:read_req}$ M_H:I* $\xrightarrow{G:read_req}$ M_O:VL ⇒VG $\xrightarrow{G:read_reply}$

M_H:I* ⇒VG $\xrightarrow{G:read_reply}$ M_R:I* ⇒VG $\xrightarrow{L:read_reply}$ C_R :?????

6.2 *rdinv_q_req* ← q-read

rdinv_q_req は，non-cachable な **q-read** によって生成される non-cachable な dw-queue に対する dequeue を伴う読出要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0100/110	= *rdinv_q_req*
size	= *size*	= accessed size
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= 0	
fe	= ⊥	
addr	= *addr*	= VBUS.addr

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (CBUS:x_crply = 1^F)，MMC が *rdinv_q_req* を受理して無条件に MBPL に割出す。

MBPL による処理の詳細は資料 [6] に示すが，概ね以下の通りである。

(1) *rdinv_q_req* の addr をキーとして，dw-queue の制御情報を得る。制御情報の中には，queue が空か否かと，queue 本体を構成する (ring) buffer の先頭が含まれる。

(2) queue が空でなければ先頭のデータを取り出して (dequeue)，以下の *rdinv_q_reply* を応答する。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0101/110	= *rdinv_q_reply*
size	= *size*	= accessed size
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= 1111	= full
addr	= *addr*	= requested address
wwe[0]	= ⊥	
data[0]	= *data*	= queue top

(3) queue が空であれば以下の *rdinv_q_reply* を応答する。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0101/110	= *rdinv_q_reply*
size	= *size*	= accessed size
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= $\perp$	
clsh	= $\perp$	
fe	= 0000	= empty
addr	= *addr*	= requested address
wwe[0]	= $\perp$	
data[0]	= $\perp$	

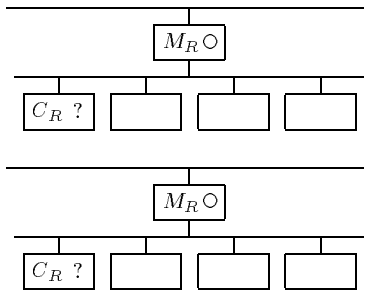
なお上記の操作は常に queue の home-cluster で行なう。

応答受領時の動作

rdinv_q_reply の fe = 1111 であれば応答された data を，また fe = 0000 であれば undefined-error をそれぞれ CPU に返し，L2C の状態は変更しない。

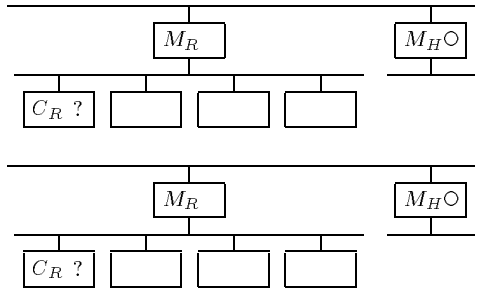
パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L : rdinv_q_req}$ M_R:VL $\xrightarrow{L : rdinv_q_reply}$ C_R :?????

(2)



C_R :???? $\xrightarrow{L : rdinv_q_req}$ M_R:I* $\xrightarrow{G : rdinv_q_req}$ M_H:VG $\xrightarrow{G : rdinv_q_reply}$

M_R:I* $\xrightarrow{L : rdinv_q_reply}$ C_R :?????

6.3 *rdinv_p_req* ← p-read

rdinv_p_req は，non-cachable な **p-read** によって生成される non-cachable な dw-queue に対する dequeue 要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0100/100	= *rdinv_p_req*
size	= *size*	= accessed size
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= 0	
fe	= ⊥	
addr	= *addr*	= VBUS.addr

q-read と **p-read** は，dw-queue が空である時の CPU に対する返値が異なるだけである。要求を送出した L2C は，その要因となった memory-command を記憶しており，応答を受理した際に memory-command に基づいて適切な返値を生成する。したがって，他の L2C の動作，MMC/MBPL の動作，パケットの流れと状態遷移については，6.2 節と全く同一である[†1]。

応答受領時の動作

rdinv_q_reply の fe = 1111 であれば応答された data を，また fe = 0000 であれば 0 をそれぞれ CPU に返し，L2C の状態は変更しない。

[†1] MMC/MBPL は *rdinv_q_reply* の代わりに *rdinv_p_reply* (pcmd/scmd = 0101/110) を応答してもよい。

6.4 *rdinv_f_req* ← q-check

rdinv_f_req は，non-cachable な **q-check** によって生成される non-cachable な dw-queue の状態チェック要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0100/111	= *rdinv_f_req*
size	= *size*	= accessed size
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= 0	
fe	= ⊥	
addr	= *addr*	= VBUS.addr

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

応答意志を表明する L2C が存在しないので (CBUS:x_crply = 1^F)，MMC が *rdinv_f_req* を受理して無条件に MBPL に割出す。

MBPL による処理の詳細は資料 [6] に示すが，概ね以下の通りである。

(1) *rdinv_f_req* の addr をキーとして，dw-queue の制御情報を得る。制御情報の中には，queue が空か否かと，queue 本体を構成する (ring) buffer の先頭が含まれる。

(2) queue が空でなければ，(dequeue 操作を行なわずに) 以下の *rdinv_f_reply* を応答する[†1]。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0101/111	= *rdinv_f_reply*
size	= *size*	= accessed size
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= 1111	= full
addr	= *addr*	= requested address
wwe[0]	= ⊥	
data[0]	= ⊥	

(3) queue が空であれば以下の *rdinv_f_reply* を応答する[†1]。

[†1] *rdinv_q_reply* または *rdinv_p_reply* であってもよい。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0101/111	= *rdinv_f_reply*
size	= *size*	= accessed size
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= $\perp$	
clsh	= $\perp$	
fe	= 0000	= empty
addr	= *addr*	= requested address
wwe[0]	= $\perp$	
data[0]	= $\perp$	

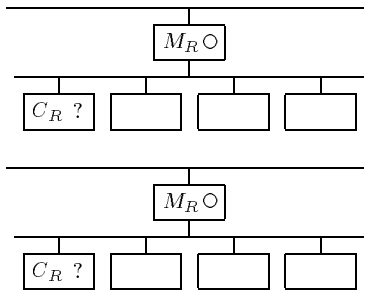
なお上記の操作は常に queue の home-cluster で行なう。

応答受領時の動作

rdinv_q_reply の fe = 1111 であれば-1 を，また fe = 0000 であれば 0 をそれぞれ CPU に返し，L2C の状態は変更しない。

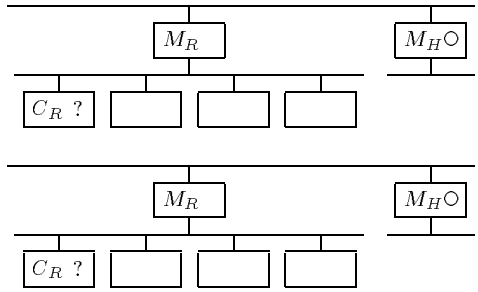
パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L : rdinv_f_req}$ M_R:VL $\xrightarrow{L : rdinv_f_reply}$ C_R :?????

(2)



C_R :???? $\xrightarrow{L : rdinv_f_req}$ M_R:I* $\xrightarrow{G : rdinv_f_req}$ M_H:VG $\xrightarrow{G : rdinv_f_reply}$

M_R:I* $\xrightarrow{L : rdinv_f_reply}$ C_R :?????

7 non-cachable store

non-cachable store とは CPU が行なう write 要求 (すなわち read でも swap でもないメモリ・アクセス要求) のうち，control-space access ではなく，かつ L2-cachable でないものをいう。

non-cachable store が許される memory-command は表 7.1 に示すものであり [2]，かつ page table entry の L1-cachability の指定によって同表の C 欄に示す以下の条件が加味される。

- no
  page table の指定が L1-cachable であってはならず，また無条件に L2-non-cachable である。
- y/n
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば (またその時に限り) L2-cachable である。
- y/n(S)
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば L2-cachable である。L1-cachable でなければ，L2-cachability は `L2CCTL.l2ccbl` に従う。
- no(S)
  page table の指定は L1-cachable であってはならず，L2-cachability は `L2CCTL.l2ccbl` に従う。

non-cachable store によって生成されるパケットは，表 7.1 に示すものである。一般に non-cachable store の対象アドレスに対応するラインを要求元あるいは他の L2C が所有していることは想定されておらず，万一所有している場合の結果は保証されない。

また対象アドレスが L3-cachable であっても，要求パケットには coherence-policy に関する情報が反映されない。したがって MMC/MBPL は，特定の coherence-policy に限定する，あるいは物理ページごとの coherence-policy を管理するなどの方法により，適切なものを選択する必要がある[†1]。

[†1] この章の「パケットの流れと状態遷移」では，**normal-write** については invalidate を，**s-write** については update-direct を仮定している。

表 7.1: non-cachable store によるパケット生成

mcmd	type	size	C	mnem	pcmd/scmd	section
0000	—	—	no	**normal-write**	*write_req*	7.1節
0001	—	word	no	**t-write**	*n_c_write_m_req*	7.4節
0001	—	dw	no	**m-write**	*n_c_write_m_req*	7.4節
0010	—	—	no(S)	**q-write**	*update_via_home_q_req*	7.3節
0011	—	—	no(S)	**q-write**	*update_via_home_q_req*	7.3節
0100	{0,1}	$\neg hw$	no(S)	**s-write**	*update_direct_s_req*	7.2節
0100	{0,1}	hw	no(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0100	2	$\neg hw$	no	**s-write**	*update_direct_s_req*	7.2節
0100	2	hw	no	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0100	3	$\neg hw$	y/n(S)	**s-write**	*update_direct_s_req*	7.2節
0100	3	hw	y/n(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0101	{0,1}	$\neg hw$	no(S)	**s-write**	*update_direct_s_req*	7.2節
0101	{0,1}	hw	no(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0101	2	$\neg hw$	no	**s-write**	*update_direct_s_req*	7.2節
0101	2	hw	no	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0101	3	$\neg hw$	y/n(S)	**s-write**	*update_direct_s_req*	7.2節
0101	3	hw	y/n(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0110	{0,1}	—	no	**q-write**	*update_via_home_q_req*	7.3節
0110	2	$\neg hw$	no(S)	**s-write**	*update_direct_s_req*	7.2節
0110	2	hw	no(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0110	3	$\neg hw$	y/n(S)	**s-write**	*update_direct_s_req*	7.2節
0110	3	hw	y/n(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0111	{0,1}	$\neg hw$	no	**normal-write**	*write_req*	7.1節
0111	{0,1}	hw	no	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0111	2	$\neg hw$	no(S)	**s-write**	*update_direct_s_req*	7.2節
0111	2	hw	no(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
0111	3	$\neg hw$	y/n(S)	**s-write**	*update_direct_s_req*	7.2節
0111	3	hw	y/n(S)	**pf/cf**(M)	*n_c_write_m_req*	7.4節
1001	{0, 1, 2}	—	y/n	**normal-write**	*write_req*	7.1節
1010	—	$\neg hw$	y/n	**normal-write**	*write_req*	7.1節
1010	—	hw	y/n	**pf/cf**(M)	*n_c_write_m_req*	7.4節
1011	—	$\neg hw$	y/n	**normal-write**	*write_req*	7.1節
1011	—	hw	y/n	**pf/cf**(M)	*n_c_write_m_req*	7.4節
1100	{0, 1, 2}	—	y/n	**normal-write**	*write_req*	7.1節
1100	3	$\neg hw$	y/n	**normal-write**	*write_req*	7.1節
1100	3	hw	y/n	**pf/cf**(M)	*n_c_write_m_req*	7.4節
1101	—	$\neg hw$	y/n	**normal-write**	*write_req*	7.1節
1101	—	hw	y/n	**pf/cf**(M)	*n_c_write_m_req*	7.4節
1110	—	—	y/n	**normal-write**	*write_req*	7.1節
1111	—	—	y/n	**normal-write**	*write_req*	7.1節

7.1 *write_req* ← normal-write

write_req は，non-cachable な **normal-write** によって生成される non-cachable な書込要求である。

送出される要求パケット

`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 0	= non cachable
`pcmd/scmd`	= 1000/010	= *write_req*
`size`	= *size*	= accessed size
`ack`	= 1	= with ack
`did`	= 100	= MMC/MBPL
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= 0	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`
`wwe[0]`	= ⊥	
`data[0]`	= *data*	= `VBUS.data`

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

MMC は対象ラインの STAG をアクセスする。その結果 `STAG:{wr_trap|valid|global}` = $\{0^F 1^T 0^F\}$ であり，かつ `size` = 11 (dw) であれば cluster-memory に `data` を書込む。

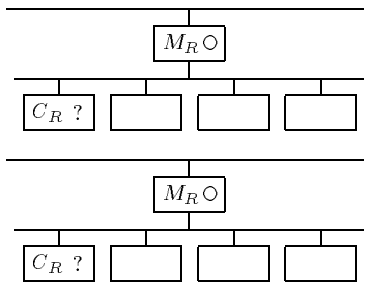
直接更新が不可能な場合，MMC は pending counter を increment して MBPL に割出す。MBPL は，`data` の有意なバイトのみに関する cluster-memory の更新，あるいは home-cluster にライン取得や coherence 維持操作を要求するなどして更新操作を行ない，操作完了後に pending counter を decrement する。

応答受領時の動作

L2C は応答を期待しない。

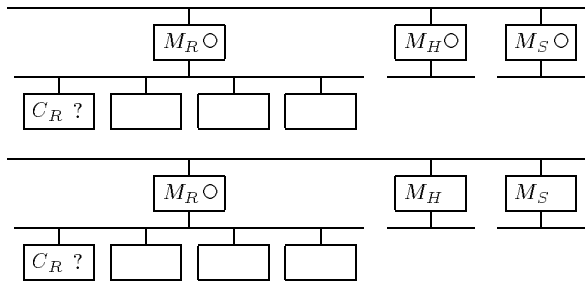
パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L : write_req}$ M_R:VL

(2)

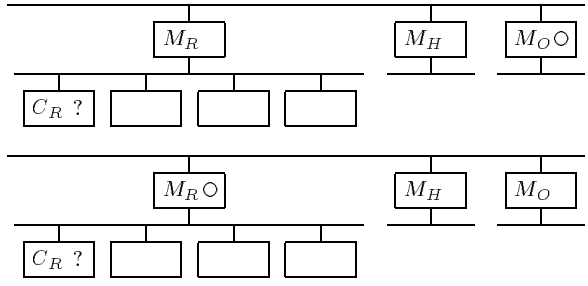


C_R :????? $\xrightarrow{L : write_req}$ M_R:VG $\xrightarrow{G : rdinv_req}$ M_H:VG $\xrightarrow{G^m : invalidate_req}$

M_S:VG $\xrightarrow{G^e : invalidate_ack}$ M_H:VG $\xrightarrow{G : rdinv_reply}$ M_R:VG

⇒I* ⇒I* ⇒VL

(3)

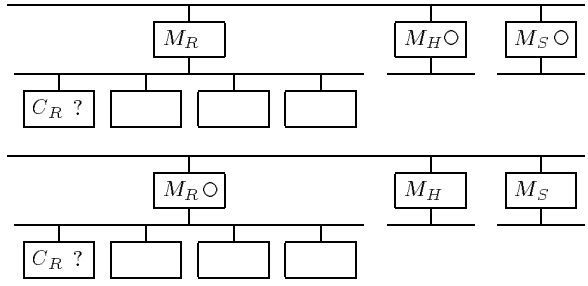


C_R :????? $\xrightarrow{L : write_req}$ M_R:I* $\xrightarrow{G : rdinv_req}$ M_H:I* $\xrightarrow{G : rdinv_req}$

M_O:VL $\xrightarrow{G : rdinv_reply}$ M_H:I* $\xrightarrow{G : rdinv_reply}$ M_R:I*

⇒I* ⇒VL

(4)



C_R :????? $\xrightarrow{L : write_req}$ M_R:I* $\xrightarrow{G : rdinv_req}$ M_H:VG $\xrightarrow{G^m : invalidate_req}$

M_S:VG $\xrightarrow{G^e : invalidate_ack}$ M_H:VG $\xrightarrow{G : rdinv_reply}$ M_R:I*

⇒I* ⇒I* ⇒VL

7.2 *update_direct_s_req* ← normal-write

update_direct_s_req は，non-cachable な **s-write** によって生成される non-cachable な書込要求である。

送出される要求パケット

`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 0	= non cachable
`pcmd/scmd`	= 1010/100	= *update_direct_s_req*
`size`	= *size*	= accessed size
`ack`	= *ack*	= `L2CCTL.wacks`
`did`	= 100	= MMC/MBPL
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= 0	
`fe`	= ⊥/*por*	= ⊥/`mcmd[0]`
`addr`	= *addr*	= `VBUS.addr`
`wwe[0]`	= ⊥	
`data[0]`	= *data*	= `VBUS.data`

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

MMC は `ack` $= 1^T$ であれば pending counter を increment して無条件に MBPL に割出す。

ラインの STAG に基づき，`data` の有意なバイトのみに関する cluster-memory の更新，あるいは home-cluster にライン取得や coherence 維持操作を要求するなどして，更新操作を行ない，`ack` $= 1^T$ であれば操作完了後に pending counter を decrement する。また，必要があれば書込待ちスレッドの起動操作を行なう。

[注意]

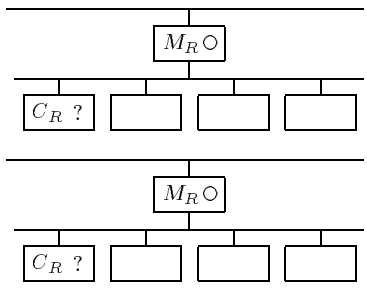
update_direct_s_req の `fepol` は無意味であり，**s-write** の同期ビット極性は `fe[0]` が示している。すなわち，`fe[0]` の値が empty を示す値であるので，更新操作完了後の対象 word/dw の `STAG:fe[dwix]` は，`fe[0]` を反転した値とすればよい。

応答受領時の動作

L2C は応答を期待しない。

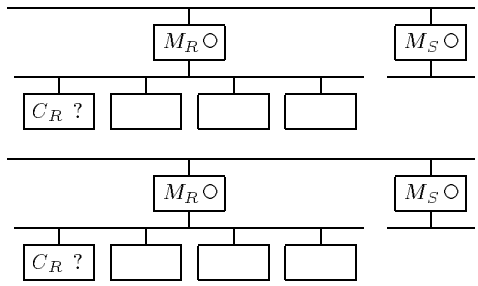
パケットの流れと状態遷移

(1)



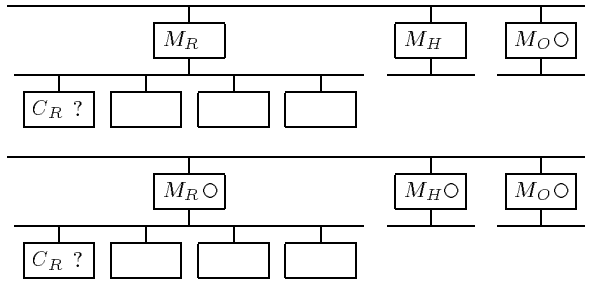
C_R :????? $\xrightarrow{L : update_direct_s_req}$ M_R:VL

(2)



C_R :????? $\xrightarrow{L : update_direct_s_req}$ M_R:VG $\xrightarrow{G^m : update_req}$ M_S:VG $\xrightarrow{G^c : update_ack}$ M_R:VG

(3)

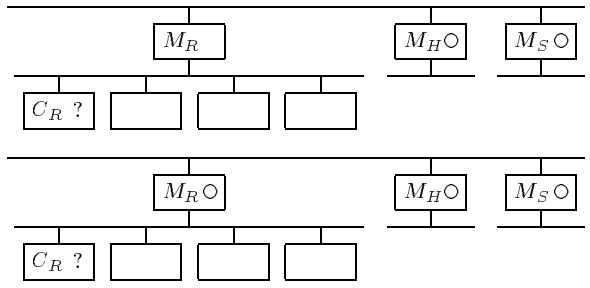


C_R :????? $\xrightarrow{L : update_direct_s_req}$ M_R:I* $\xrightarrow{G : read_and_update_req}$

M_H:I* $\xrightarrow{G : read_and_update_req}$ M_O:VL $\xrightarrow{G : read_and_update_reply}$
⇒VG

M_H:I* $\xrightarrow{G : read_and_update_reply}$ M_R:I*
⇒VG ⇒VG

(4)



C_R :????? $\xrightarrow{L : update_direct_s_req}$ M_R:VG $\xrightarrow{G : read_and_update_req}$ M_H:VG $\xrightarrow{G : update_req}$

M_S:VG $\xrightarrow{G : update_ack}$ M_H:VG $\xrightarrow{G : read_and_update_reply}$ M_R:I*
⇒VG

7.3 *update_via_home_q_req* ← q-write

update_via_home_q_req は，**q-write** による dw-queue への enqueue 要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1000/110	= *update_via_home_q_req*
size	= *size*	= accessed size
ack	= *ack*	= L2CCTL.wackq
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= $\perp$	
clsh	= 0	
fe	= $\perp$	
addr	= *addr*	= VBUS.addr
wwe[0]	= $\perp$	
data[0]	= *data*	= VBUS.data

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

MMC は ack = 1^T であれば pending counter を increment して無条件に MBPL に割出す。

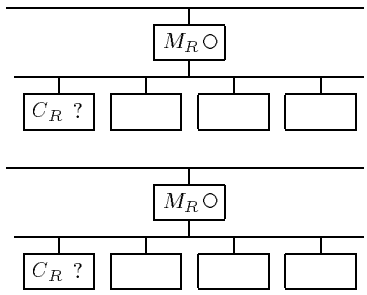
MBPL は，[6] に示す方法で enqueue 操作を行ない，ack = 1^T であれば操作完了確認後に pending counter を decrement する。

応答受領時の動作

L2C は応答を期待しない。

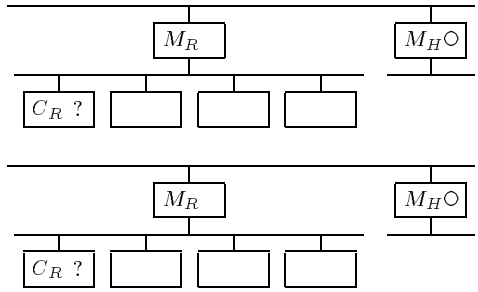
パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L : update_via_home_q_req}$ M_R:VL

(2)



C_R :????? $\xrightarrow{L : update_via_home_q_req}$ M_R:I* $\xrightarrow{G : update_via_home_q_req}$ M_H:VG

7.4 *n_c_write_m_req* ← t-write, m-write, pf/cf(M)

n_c_write_m_req は，**t-write**, **m-write**, または **pf/cf**(M) による MBPL への処理要求である。

送出される要求パケット

`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 0	= non cachable
`pcmd/scmd`	= 1010/011	= *n_c_write_m_req*
`size`	= *size*	= accessed size
`ack`	= 1	= with ack
`did`	= 100	= MMC/MBPL
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= 0	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`
`wwe[0]`	= ⊥	
`data[0]`	= *data*	= `VBUS.data`

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

MMC は pending counter を increment して無条件に MBPL に割出す。

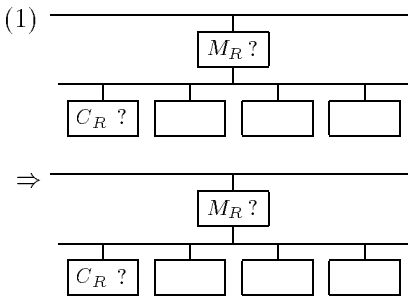
MBPL の動作については本資料では規定しない。なお **t-write**, **m-write**, **pf/cf**(M) のいずれによって *n_c_write_m_req* が送出されたかは，`size` によって以下のように判別することができる。

size	command
01 (hw)	**pf/cf**(M)
10 (word)	**t-write**
11 (dw)	**m-write**

応答受領時の動作

L2C は応答を期待しない。

パケットの流れと状態遷移



C_R :????? $\xrightarrow{L : n_c_write_m_req}$ M_R:??

8 non-cachable swap

non-cachable swap とは CPU が行なう swap 要求 (すなわち read でも write でもないメモリ・アクセス要求) のうち，control-space access ではなく，かつ L2-cachable でないものをいう。

non-cachable swap が許される memory-command は表 8.1 に示すものであり [2]，かつ page table entry の L1-cachability の指定によって同表の C 欄に示す以下の条件が加味される。

- no
  page table の指定が L1-cachable であってはならず，また無条件に L2-non-cachable である。
- y/n
  page table の指定が L1-cachable であってもなくてもよく，L1-cachable であれば (またその時に限り) L2-cachable である。

non-cachable swap によって生成されるパケットは，表 8.1 に示すものである。一般に non-cachable swap の対象アドレスに対応するラインを要求元あるいは他の L2C が所有していることは想定されておらず，万一所有している場合の結果は保証されない。

また対象アドレスが L3-cachable であっても，要求パケットには coherence-policy に関する情報が反映されない。したがって MMC/MBPL は，特定の coherence-policy に限定する，あるいは物理ページごとの coherence-policy を管理するなどの方法により，適切なものを選択する必要がある[†1]。

表 8.1: non-cachable swap によるパケット生成

mcmd	type	C	mnem	pcmd/scmd	section
0000	—	no	**normal-swap**	*swap_req*	8.1節
0001	—	no	**m-swap**	*swap_m_req*	8.2節
0111	{0,1}	y/n	**normal-swap**	*swap_req*	8.1節
1001	{0,1,2}	y/n	**normal-swap**	*swap_req*	8.1節
1010	—	y/n	**normal-swap**	*swap_req*	8.1節
1011	—	y/n	**normal-swap**	*swap_req*	8.1節
1100	—	y/n	**normal-swap**	*swap_req*	8.1節
1101	—	y/n	**normal-swap**	*swap_req*	8.1節
1110	—	y/n	**normal-swap**	*swap_req*	8.1節
1111	—	y/n	**normal-swap**	*swap_req*	8.1節

[†1] この章の「パケットの流れと状態遷移」では，**normal-swap** について invalidate を仮定している。

8.1 *swap_req* ← normal-swap

swap_req は，non-cachable な **normal-swap** によって生成される non-cachable な swap 要求である。

送出される要求パケット

`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 0	= non cachable
`pcmd/scmd`	= 0000/011	= *swap_req*
`size`	= *size*	= accessed size
`ack`	= 0	= nop
`did`	= 100	= MMC/MBPL
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= 0	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`
`wwe[0]`	= ⊥	
`data[0]`	= *data*	= `VBUS.data`

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

MMC は無条件に MBPL に割出す。

MBPL は，対象 byte/word に関する cluster-memory の更新，あるいは home-cluster にライン取得や coherence 維持操作を要求するなどして更新操作を行ない，最終的には更新される前の byte/word の値を以下の *swap_reply* によって応答する。

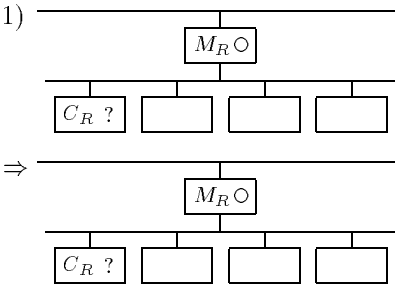
`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 0	= non cachable
`pcmd/scmd`	= 0001/011	= *swap_reply*
`size`	= *size*	= accessed size
`ack`	= 0	= nop
`did`	= *did*	= requester's ID
`sid`	= 100	= MMC/MBPL
`drty`	= 0	= clean
`clsh`	= *global*	= `STAG.global`
`fe`	= *sync*	= `STAG.fe`
`addr`	= *addr*	= requested address
`wwe[0]`	= *w-wait/r-wait*	= `DTAG[dwix]`
`data[0]`	= *data*	= `DRAM[dwix]`

応答受領時の動作

単に `data` を CPU に返し，L2C の状態は変更しない。

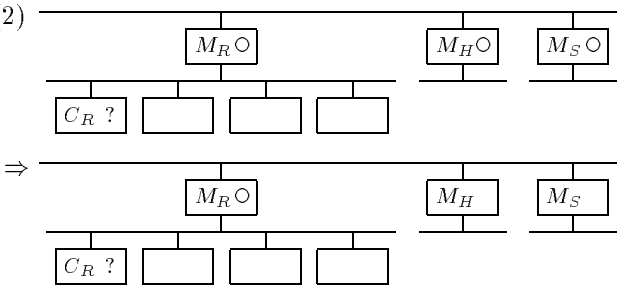
パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L:swap_req}$ M_R:VL $\xrightarrow{L:swap_reply}$ C_R :?????

(2)

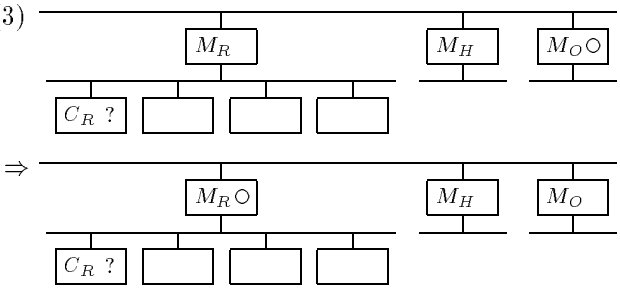


C_R :????? $\xrightarrow{L:swap_req}$ M_R:VG $\xrightarrow{G:rdinv_req}$ M_H:VG $\xrightarrow{G^m:invalidate_req}$

M_S:VG $\xrightarrow{G^e:invalidate_ack}$ M_H:VG $\xrightarrow{G:rdinv_reply}$ M_R:VG $\xrightarrow{L:swap_reply}$ C_R :?????

⇒I* ⇒I* ⇒VL

(3)

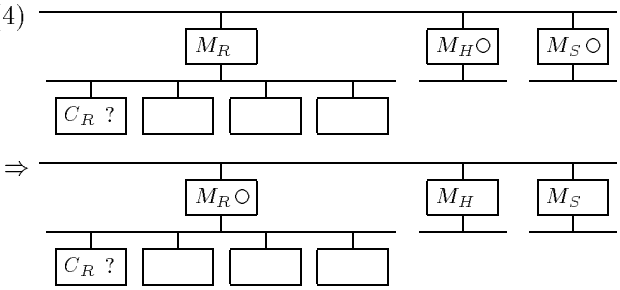


C_R :????? $\xrightarrow{L:swap_req}$ M_R:I* $\xrightarrow{G:rdinv_req}$ M_H:I* $\xrightarrow{G:rdinv_req}$

M_O:VL $\xrightarrow{G:rdinv_reply}$ M_H:I* $\xrightarrow{G:rdinv_reply}$ M_R:I* $\xrightarrow{L:swap_reply}$ C_R :?????

⇒I* ⇒VL

(4)



C_R :????? $\xrightarrow{L:swap_req}$ M_R:I* $\xrightarrow{G:rdinv_req}$ M_H:VG $\xrightarrow{G^m:invalidate_req}$

M_S:VG $\xrightarrow{G^e:invalidate_ack}$ M_H:VG $\xrightarrow{G:rdinv_reply}$ M_R:I* $\xrightarrow{L:swap_reply}$ C_R :?????

⇒I* ⇒I* ⇒VL

8.2 *swap_m_req* ← m-swap

swap_req は，**m-swap** による応答を伴う MBPL への処理要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0010/011	= *swap_m_req*
size	= *size*	= accessed size
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= 0	
fe	= ⊥	
addr	= *addr*	= VBUS.addr
wwe[0]	= ⊥	
data[0]	= *data*	= VBUS.data

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。一般に non-cachable な要求に対応するラインを L2C が所有していることは想定されていない。

MMC/MBPL の動作

MMC は無条件に MBPL に割出す。

MBPL の動作については本資料では規定しないが，最終的には以下の *swap_m_reply* によって応答する。

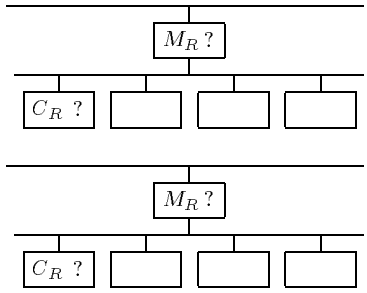
plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 0011/011	= *swap_m_reply*
size	= *size*	= accessed size
ack	= 0	= nop
did	= *did*	= requester's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= requested address
wwe[0]	= ⊥	
data[0]	= *data*	= replied data

応答受領時の動作

単に data を CPU に返し，L2C の状態は変更しない。

パケットの流れと状態遷移

(1)



C_R :????? $\xrightarrow{L : swap_m_req}$ M_R:?? $\xrightarrow{L : swap_m_reply}$ C_R :?????

9 control-space access

CSA-write により，control-space のアドレス 0x10000160 に割り当てられた CC の内部レジスタ ECSAID に，表 9.1 に示す値を STBA で書込むと，その次に行なわれる CSA-write により他の L2C や MMC/MBPL の内部資源をアクセスするための *control_write_req* が送出される [5][†1]。

また CSA-write により，control-space のアドレス 0xE0000000 ～ 0xEFFFFFFF へ任意のデータを書込むと，BMP[addr[19 : 5]] を読出してその値を内部レジスタ BMPRDR にセットするための，自身に宛てられた *control_write_req* が送出される。同様に 0xF0000000 ～ 0xFFFFFFFF へ任意のデータを書込むと，BMP[addr[19 : 5]] に addr[27 : 20] と addr[3 : 0] を書込むための，自身に宛てられた *control_write_req* が送出される。

表 9.1: ECSAID と送出されるパケット

ECSAID	pcmd/scmd(did)	section
0x8	*control_write_req*(000 = CC⟨0⟩)	9.1節
0x9	*control_write_req*(001 = CC⟨1⟩)	9.1節
0xA	*control_write_req*(010 = CC⟨2⟩)	9.1節
0xB	*control_write_req*(011 = CC⟨3⟩)	9.1節
0xC	*control_write_req*(100 = MMC/MBPL)	9.1節
0xD	(reserved)	
0xE	(reserved)	
0xF	*control_write_req*(111 = all)	9.1節

[†1] $0x0 \leq$ ECSAID $\leq 0x4$ であるときには *control_read_req* が送出されるが，その応答は要求元の L2C に受理はされるものの結果的に無視される。

9.1 *control_write_req* ← CSA-write(write-external)

control_write_req は，以下のいずれかによって送出される L2C , MMC/MBPL の内部資源アクセス要求である。

(1) `0x8` ≤ `ECSAID` ≤ `0xC` である状態での CSA-write による，他の L2C または MMC/MBPL の内部資源書込。

(2) CSA-write による，自身の BMP の読出 / 書込。

送出される要求パケット

`plen`	= 01	= 2 flit
`injmask`	= 0	= nop
`ccbl`	= 0	= non cachable
`pcmd/scmd`	= 1110/010	= *control_write_req*
`size`	= *size*	= accessed size
`ack`	= 1	= with ack
`did`	= *did*	= recipient's ID
`sid`	= *pid*	= own ID
`drty`	= ⊥	
`clsh`	= 0	
`fe`	= ⊥	
`addr`	= *addr*	= `VBUS.addr`
`wwe[0]`	= ⊥	
`data[0]`	= *data*	= `VBUS.data`

他の L2C の動作

0 ≤ `did` ≤ 3 であれば，`did` により定まる L2C が *control_write_req* を受理し，`addr` により定まる内部資源に `data` を書込む。なお `addr` と内部資源の対応については [5] を参照せよ。

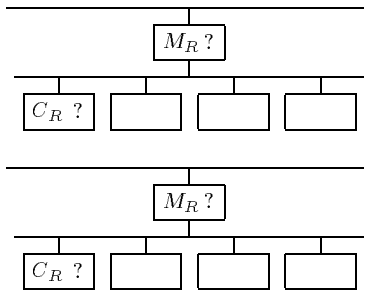
MMC/MBPL の動作

`did` = 4 であれば MMC が *control_write_req* を受理し，pending counter を increment した上で無条件に MBPL に割出す。

MBPL の動作については本資料では規定しないが，`addr` により定められる動作を行ない，完了後に pending counter を decrement する。

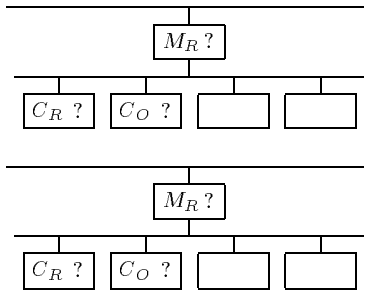
パケットの流れと状態遷移

(1)



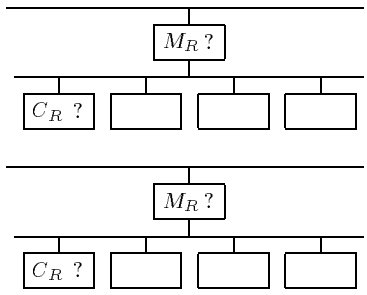
C_R :????? $\xrightarrow{L : control_write_req(C_R)}$ C_R :?????

(2)



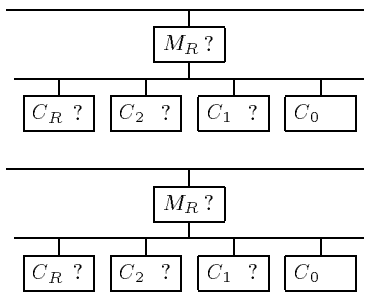
C_R :????? $\xrightarrow{L : control_write_req(C_O)}$ C_O :?????

(3)



C_R :????? $\xrightarrow{L : control_write_req(M_R)}$ M_R:??

(4)



C_R :????? $\xrightarrow{L : control_write_req(\text{all})}$ C_R :?????
C_2 :?????
C_1 :?????
C_0 :?????
M_R:??

10 pseudo-reply

L2C は以下の 2 種類のパケットを，疑似応答として送出することがある。

(1) *writeback_reply* (10.1 節)
dirty なラインが L2C から除去される際の，cluster-memory への書き戻し。

(2) *nop* (10.2 節)
BMP の更新などの内部操作。

10.1 *writeback_reply*

writeback_reply は，dirty なラインが L2C から除去される際に送出される，cluster-memory へのライン書き戻しのための疑似応答である。

送出される疑似応答パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1001/010	= *writeback_reply*
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= *dirty*	= FMP:state.d
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= FMP:tag/VBUS.addr
wwe[0:3]	= {*w-wait*, 0}∗4	= {FMP : ww, 0}∗4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

他の L2C の動作

対象ラインの状態に関わらず，他の L2C は何もしない。

MMC/MBPL の動作

MMC は *writeback_reply* を無条件に受理し，対象ラインの STAG をアクセスする。その結果 STAG:{wr_trap|valid|global} = $\{0^F 1^T 0^F\}$ であれば，data を cluster-memory に書込む。

直接書込が不可能な場合には MBPL によって処理され，home-cluster にラインの書き戻しを要求するなどの操作を行なう[†1]

[注意]

writeback_reply の wwe は，DTAG が保持すべき待ちスレッドの有無情報とは一致しない。したがって MMC は DTAG を更新しない。また MBPL が書き戻し操作を行なう場合にも，DTAG を更新してはならない。

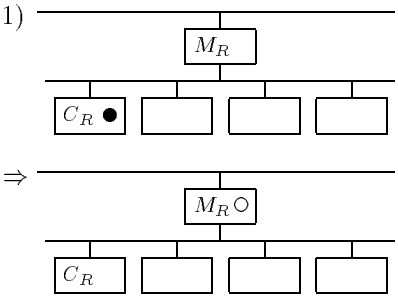
応答受領時の動作

L2C は応答を期待しない。

[†1] 例えば coherence-policy が no-coherent であるようなラインを他のクラスタから輸入している場合には，他クラスタへの書き戻し操作が必要であろう。

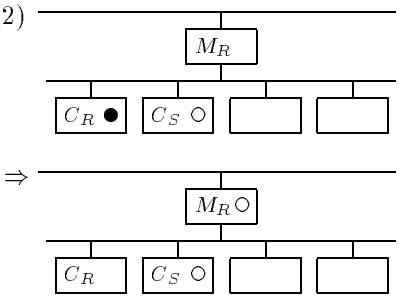
パケットの流れと状態遷移

(1)



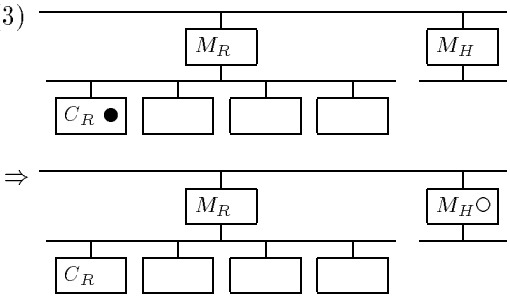
C_R :VXLDO $\xrightarrow{L : writeback_reply}$ M_R:VL
⇒I****

(2)



C_R :VSLDO $\xrightarrow{L : writeback_reply}$ M_R:VL
⇒I****

(3)



C_R :VSGDN $\xrightarrow{L : writeback_reply}$ M_R:I* $\xrightarrow{G : writeback_reply}$ M_H:I*
⇒I**** ⇒VG

10.2 *nop*

nop は，L2C が以下の理由により送出する疑似応答であり，他の L2C や MMC/MBPL での操作を引き起こすものではない。

(1) CPU の要求によるライン無効化 (**cflush**) の際に，BMP を無効状態にする。

(2) *allread_reply* (2.2 節参照) または *injection_req* (11.9 節参照) による injection 操作が，内部レジスタ `INJLEV` が定める条件に合致しない場合に，一旦は injection により更新された BMP の状態を復旧する。

(3) L2C が要求送出のために CBUS を確保した後に，対象ラインに対する操作競合によってその要求の送出ができないことが判明した場合，要求を *nop* に変換して送出する。

以下は上記の (2) の際に送出される *nop* のパケット値であるが，一般には `pcmd/scmd` 以外のフィールドの値は意味を持たない[†1]。またパケット長についても，1, 2, 5 のいずれの可能性もある。

送出される疑似応答パケット

`plen`	= 00	= 1 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 1111/111	= *nop*
`size`	= 000	= ⊥ (byte)
`ack`	= 0	= nop
`did`	= 111	= all
`sid`	= *pid*	= own ID
`drty`	= *owend*	= `BMP.state.o`
`clsh`	= *global*	= ¬`FMP:state.l`
`fe`	= *sync*	= `FMP:sync`
`addr`	= *addr*	= `FMP:tag`/requested address

他の L2C の動作

対応するラインの状態に関わらず，他の L2C は何もしない。

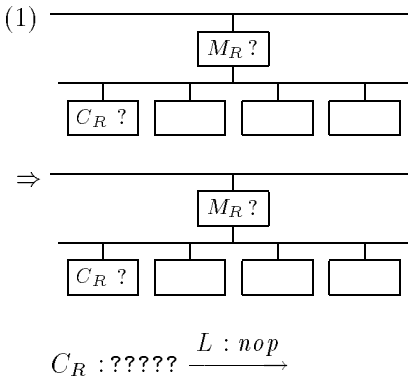
MMC/MBPL の動作

MMC は *nop* を受理しない。

応答受領時の動作

L2C は応答を期待しない。

パケットの流れと状態遷移



[†1] `injmask` $= 0^F$ は保証される。

11 MMC/MBPL からの要求パケット

L2C は，MMC/MBPL が送出する要求パケットのうち，表 11.1 に示すものを受理する。なお表の cond 欄の表記は，以下の受理条件を意味する。ただし i は L2C の device-ID である。また *error_reply* は要求ではなく応答パケットであるが，便宜上この節で説明する。

- H
  要求パケットについて $\texttt{ccbl} = 1^T$ であり，かつ対象ラインを L2C が保有していれば受理する。この受理条件が定められたパケットについて，$\texttt{did} = i$ (all) または $\texttt{did} = 100$ (MMC/MBPL) でない場合の結果は保証されない。
- I
  要求パケットについて $\texttt{injmask}[i] = 1^T$ であれば受理する。この受理条件が定められていない要求パケットについて，$\texttt{injmask}[i] = 1^T$ である場合の結果は保証されない。
- D
  要求 / 応答パケットについて $\texttt{did} = i$ であれば受理する。
- D'
  要求パケットについて $\texttt{did} = i$ あるいは $\texttt{did} = 111$ (all) であれば受理する。

また表に記載されていない要求パケットについては，$\texttt{did} = i$ あるいは $\texttt{did} = 111$ (all) であれば，不正パケットとして CPU に対する割込 (NMI) を生起する。

表 11.1: L2C が受理する MMC/MBPL からの要求パケット

pcmd/scmd	cond	section
read_req	H	11.1 節
allread_req	H	11.2 節
rdinv_req	H	11.3 節
invalidate_req	H	11.4 節
(*invalidate_via_home_req*)	H	11.4 節
(*bq_write_req*)	H	11.4 節
update_req	H	11.5 節
(*update_via_home_req*)	H	11.5 節
(*update_direct_req*)	H	11.5 節
(*update_direct_s_req*)	H	11.5 節
no_alloc_write_req	H	11.6 節
swap_req	H	11.7 節
cflush_req	H	11.8 節
injection_req	I	11.9 節
demap_req	D'	11.10 節
control_read_req	D	11.11 節
interrupt_req	D'	11.12 節
control_write_req	D'	11.13 節
error_reply	D	11.14 節

11.1 *read_req*

read_req は，付随的な操作を伴わない，ラインの読出要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0000/000	= *read_req*
size	= 100	= block
ack	= 0	= nop
did	= 111	= all
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= requested address

想定される用法

読出の coherence-policy が normal あるいは all-read であるような **normal-read**, **s-read**, **f-read**, **normal-write**, **s-write** が cache-miss し，かつ要求元ではないようなクラスタ C_O のみがラインを所有している場合に，home-cluster からのライン取得要求に応えるために C_O の MBPL が L2C に要求する。

[注意]

読出の coherence-policy が all-read である場合，本来ならば C_O の MBPL は *allread_req* によってラインを取得すべきであるが，L2C での操作に不具合があるため *read_req* を使用しなければならない（11.2 節参照）。

L2C の動作

対象ラインを owner として所有している L2C があれば (S = V***O)，その L2C が CBUS:x_crply = 0^T として応答意志を示し，FMP/DAR を読出して以下の *read_reply* を応答し，S = VSGCO に遷移する。なお共有状態が globally-shared であっても owner であれば応答し，ownership は放棄しない[†1]。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/000	= *read_reply*
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= *dirty*	= FMP:state.d
clsh	= *global*	= ¬FMP:state.l
fe	= *sync*	= FMP:sync
addr	= *addr*	= requested address
wwe[0:3]	= {*w-wait*, 0}∗4	= {FMP : ww, 0}∗4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

対象ラインを所有しているが owner ではない L2C は，単に S = VSGCN に遷移する。

[注意]

writeback_reply の wwe は，DTAG が保持すべき待ちスレッドの有無情報とは一致しない。したがって MBPL が応答値を cluster-memory に書込む際には，DTAG を更新してはならない。

MMC/MBPL の動作

対象ラインを owner として所有している L2C が存在しなければ (CBUS:x_crply = 1^F), MMC は対象ライ

[†1] globally-shared であることは上記の用法では想定されない。

ンの STAG をアクセスする。その結果 STAG:{rd_trap|valid} $= \{0^F 1^T\}$ であれば，cluster-memory を読出して以下の *read_reply* を自分自身に対して応答する。

plen	$= 10$	$= 5$ flit
injmask	$= 0$	$=$ nop
ccbl	$= 1$	$=$ cachable
pcmd/scmd	$= 0001/000$	$=$ *read_reply*
size	$= 100$	$=$ block
ack	$= 0$	$=$ nop
did	$= 100$	$=$ MMC/MBPL
sid	$= 100$	$=$ MMC/MBPL
drty	$= 0$	$=$ clean
clsh	$=$ *global*	$=$ STAG.global
fe	$=$ *sync*	$=$ STAG.fe
addr	$=$ *addr*	$=$ requested address
wwe[0:3]	$=$ *w-wait/r-wait*	$=$ DTAG[dwix..dwix+3]
data[0:3]	$=$ *data*	$=$ DRAM[dwix..dwix+3]

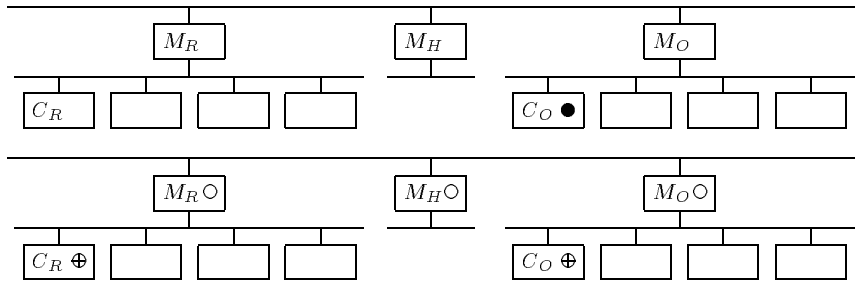
直接応答が不可能であるための MBPL への割出しは，想定が困難である。

応答受領時の動作

L2C あるいは自身からの *read_reply* を MMC が受理すると，無条件で MBPL に割出す。

パケットの流れと状態遷移

(1)

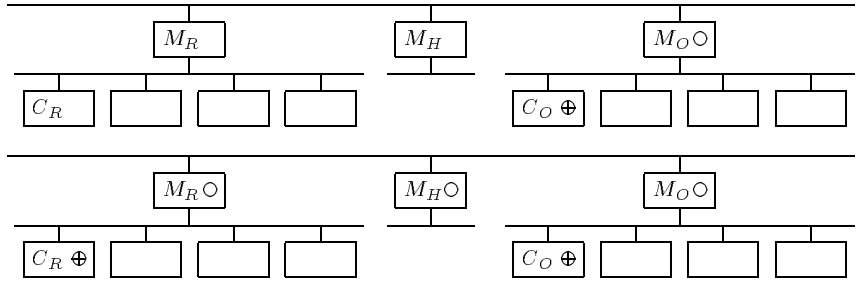


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

C_O :VXLDO $\xrightarrow{L : read_reply(X, L, D)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(2)

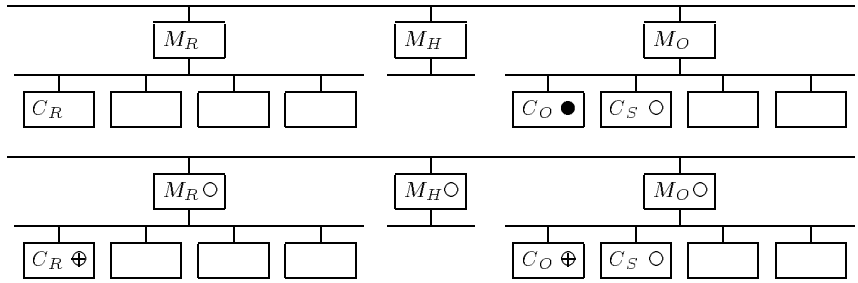


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

C_O :VXLCO $\xrightarrow{L : read_reply(X, L, C)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(3)

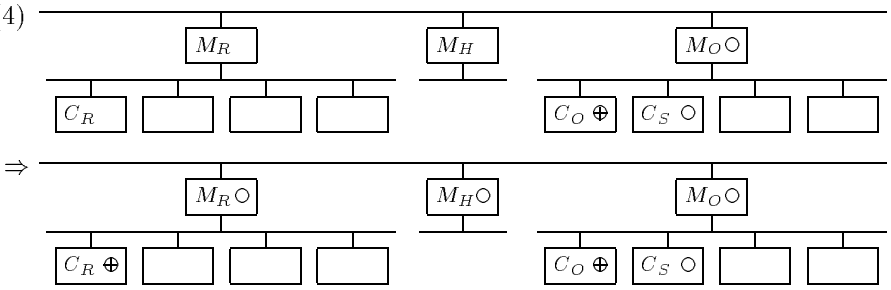


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

C_O :VSLDO $\xrightarrow{L : read_reply(S, L, D)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG
C_S :VSLCN
⇒ VSGCN

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(4)

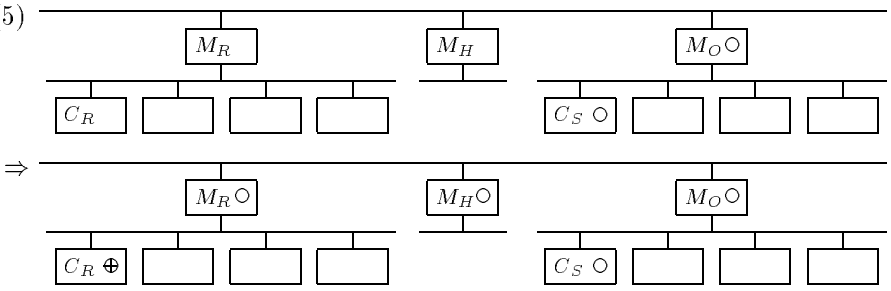


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

C_O :VSLCO $\xrightarrow{L : read_reply(S, L, C)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VSGCO ⇒ VG ⇒ VG
C_S :VSLCN
⇒ VSGCN

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(5)

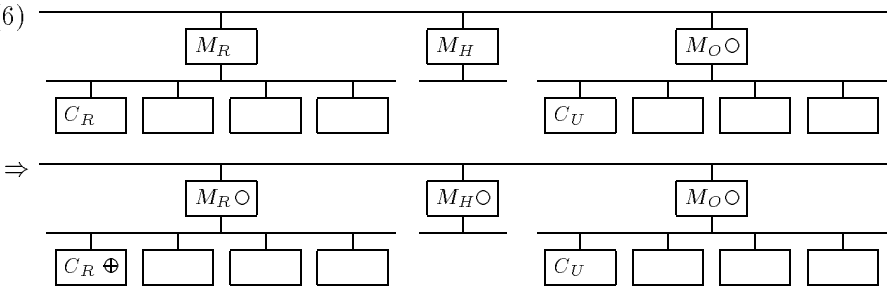


C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

M_O:VL $\xrightarrow{L : read_reply(S, L, C)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
C_S :VSLCN ⇒ VG ⇒ VG
⇒ VSGCN

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(6)



C_R :I**** $\xrightarrow{L : read_req}$ M_R:I* $\xrightarrow{G : read_req}$ M_H:I* $\xrightarrow{G : read_req}$ M_O:VL $\xrightarrow{L : read_req}$

M_O:VL $\xrightarrow{L : read_reply(X, L, C)}$ M_O:VL $\xrightarrow{G : read_reply}$ M_H:I* $\xrightarrow{G : read_reply}$
⇒ VG ⇒ VG

M_R:I* $\xrightarrow{L : read_reply(X, G, C)}$ C_R :I****
⇒ VG ⇒ VSGCO

11.2 *allread_req*

allread_req は，injection を伴う，ラインの読出要求である。

[注意]

allread_req を受理して *allread_reply* を応答する L2C は $S = \texttt{VSGCN}$ に遷移するが，応答の clsh には遷移前の状態に基づくクラスタ間共有状態を報告する（したがって通常は 0^F）。この結果，対象ラインを所有していない L2C は locally-shared であると誤解して injection を行なうため，クラスタ間共有に関する状態の矛盾が生じる。

したがって，一般に MBPL は *allread_req* を CBUS に送出してはならない。

11.3 *rdinv_req*

rdinv_req は，無効化を伴う，ラインの読出要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0100/000	= *rdinv_req*
size	= 100	= block
ack	= 0	= nop
did	= 111	= all
sid	= 100	= MMC/MBPL
drty	= $\perp$	
clsh	= $\perp$	
fe	= $\perp$	
addr	= *addr*	= requested address

想定される用法

以下のいずれかであって，かつ要求元ではないようなクラスタ C_O のみがラインを所有している場合に，home-cluster からのライン取得要求に応えるために C_O の MBPL が L2C に要求する。

(1) 読出の coherence-policy が read-invalidate であるような **normal-read** の cache-miss。

(2) **q-read**, **p-read**, **q-check** の cache-miss。

(3) 読出の coherence-policy が read-invalidate であるか，書込の coherence-policy が invalidate であるような **normal-write** の cache-miss。

(4) **bq-write** の cache-miss。

(5) swap の coherence-policy が read-invalidate であるような **normal-swap** の cache-miss あるいは shared-hit。

L2C の動作

対象ラインを owner として所有している L2C があり，かつ共有状態が globally-shared でなければ (S = V*L*O)[†1]，その L2C が CBUS:x_crply = 0^T として応答意志を示し，FMP/DAR を読出して以下の *rdinv_reply* を応答し，対象ラインを無効化する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/000	= *rdinv_reply*
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= *dirty*	= FMP:state.d
clsh	= 0	= $\neg$(FMP : state.l = 1)
fe	= *sync*	= FMP:sync
addr	= *addr*	= requested address
wwe[0:3]	= {*w-wait*, 0}*4	= {FMP : ww, 0}*4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

対象ラインを所有しているが上記の条件を満たさない L2C は，*rdinv_req* を無効化要求と解釈し，対象ラインの無効化を行なう。

[†1] globally-shared であることは上記の用法では想定されない。

[注意]

writeback_reply の wwe は，DTAG が保持すべき待ちスレッドの有無情報とは一致しない。したがって MBPL が応答値を cluster-memory に書込む際には，DTAG を更新してはならない。

MMC/MBPL の動作

対象ラインが globally-shared であるか[†1]，owner として所有している L2C が存在しなければ (CBUS:x_crply $= 1^F$), MMC は対象ラインの STAG をアクセスする。その結果 STAG:{rd_trap|valid} $= \{0^F 1^T\}$ であれば，cluster-memory を読出して以下の *read_reply* を自分自身に対して応答する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0101/000	= *rdinv_reply*
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= 100	= MMC/MBPL
drty	= 0	= clean
clsh	= *global*	= STAG.global
fe	= *sync*	= STAG.fe
addr	= *addr*	= requested address
wwe[0:3]	= *w-wait/r-wait*	= DTAG[dwix..dwix+3]
data[0:3]	= *data*	= DRAM[dwix..dwix+3]

直接応答が不可能であるための MBPL への割出しは，想定が困難である。

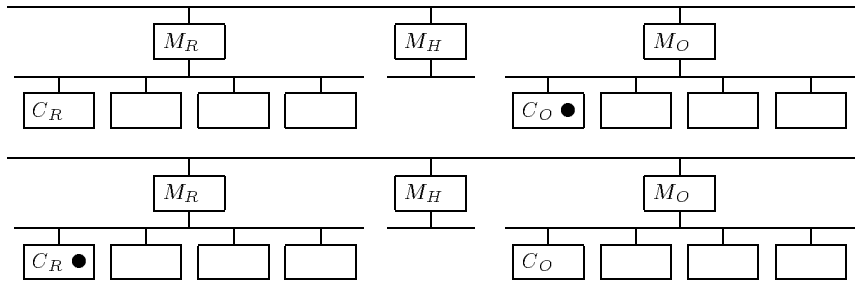
応答受領時の動作

L2C あるいは自身からの *rdinv_reply* を MMC が受理すると，無条件で MBPL に割出す。

[†1] globally-shared であることは上記の用法では想定されない。

パケットの流れと状態遷移

(1)



C_R :I**** $\xrightarrow{L:rdinv_req}$ M_R:I* $\xrightarrow{G:rdinv_req}$ M_H:I* $\xrightarrow{G:rdinv_req}$ M_O:VL $\xrightarrow{L:rdinv_req}$

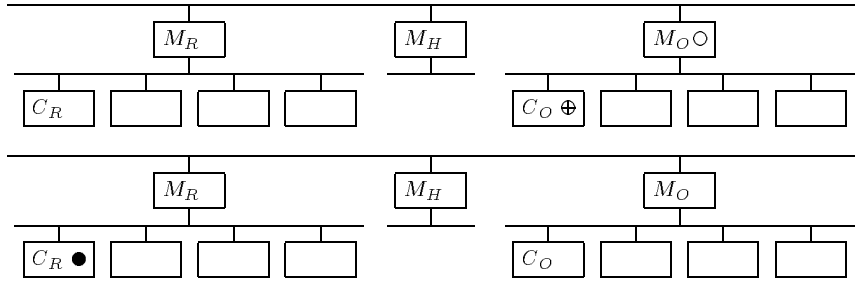
C_O :VXLDO $\xrightarrow{L:rdinv_reply(X,L,D)}$ M_O:VL $\xrightarrow{G:rdinv_reply}$ M_H:I* $\xrightarrow{G:rdinv_reply}$

⇒ I**** ⇒ I*

M_R:I* $\xrightarrow{L:rdinv_reply(X,L,D)}$ C_R :I****

⇒ VL ⇒ VXLDO

(2)



C_R :I**** $\xrightarrow{L:rdinv_req}$ M_R:I* $\xrightarrow{G:rdinv_req}$ M_H:I* $\xrightarrow{G:rdinv_req}$ M_O:VL $\xrightarrow{L:rdinv_req}$

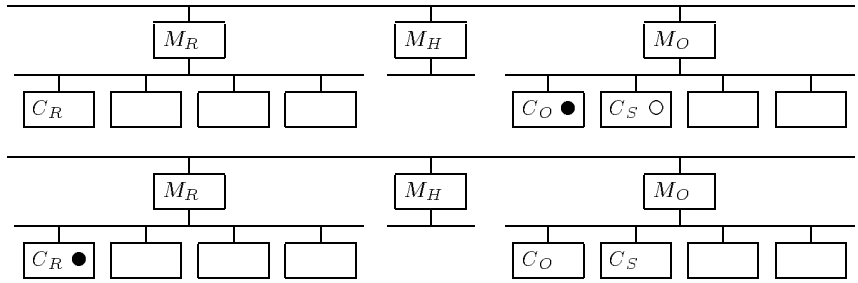
C_O :VXLCO $\xrightarrow{L:rdinv_reply(X,L,C)}$ M_O:VL $\xrightarrow{G:rdinv_reply}$ M_H:I* $\xrightarrow{G:rdinv_reply}$

⇒ I**** ⇒ I*

M_R:I* $\xrightarrow{L:rdinv_reply(X,L,D)}$ C_R :I****

⇒ VL ⇒ VXLDO

(3)



C_R :I**** $\xrightarrow{L:rdinv_req}$ M_R:I* $\xrightarrow{G:rdinv_req}$ M_H:I* $\xrightarrow{G:rdinv_req}$ M_O:VL $\xrightarrow{L:rdinv_req}$

C_O :VSLDO $\xrightarrow{L:rdinv_reply(X,L,D)}$ M_O:VL $\xrightarrow{G:rdinv_reply}$ M_H:I* $\xrightarrow{G:rdinv_reply}$

⇒ I**** ⇒ I*

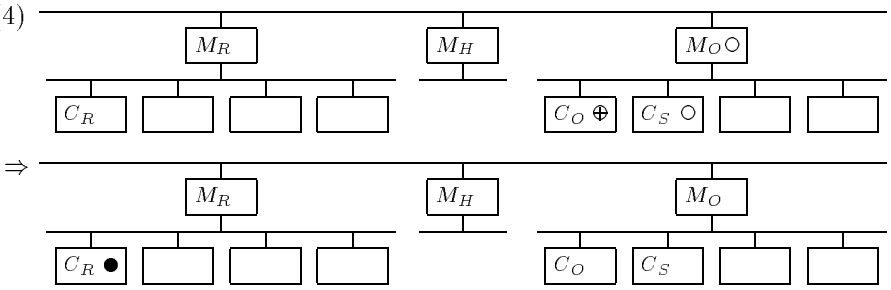
C_S :VSLCN

⇒ I****

M_R:I* $\xrightarrow{L:rdinv_reply(X,L,D)}$ C_R :I****

⇒ VL ⇒ VXLDO

(4)



C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : I* $\xrightarrow{G : rdinv_req}$ M_H : I* $\xrightarrow{G : rdinv_req}$ M_O : VL $\xrightarrow{L : rdinv_req}$

C_O : VSLCO $\xrightarrow{L : rdinv_reply(X, L, C)}$ M_O : VL $\xrightarrow{G : rdinv_reply}$ M_H : I* $\xrightarrow{G : rdinv_reply}$

$\Rightarrow$ I****

$\Rightarrow$ I*

C_S : VSLCN

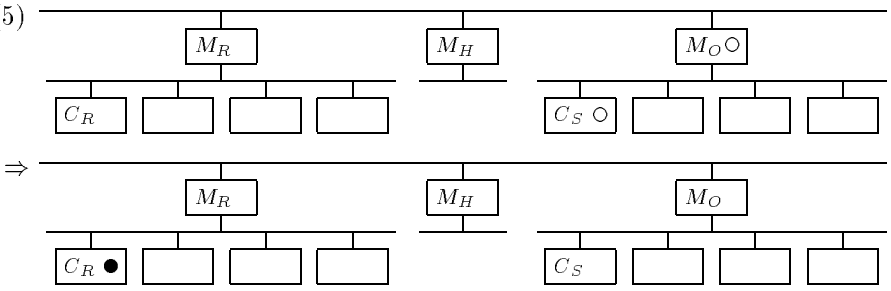
$\Rightarrow$ I****

M_R : I* $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : I****

$\Rightarrow$ VL

$\Rightarrow$ VXLDO

(5)



C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : I* $\xrightarrow{G : rdinv_req}$ M_H : I* $\xrightarrow{G : rdinv_req}$ M_O : VL $\xrightarrow{L : rdinv_req}$

M_O : VL $\xrightarrow{L : rdinv_reply(X, L, C)}$ M_O : VL $\xrightarrow{G : rdinv_reply}$ M_H : I* $\xrightarrow{G : rdinv_reply}$

C_S : VSLCN

$\Rightarrow$ I*

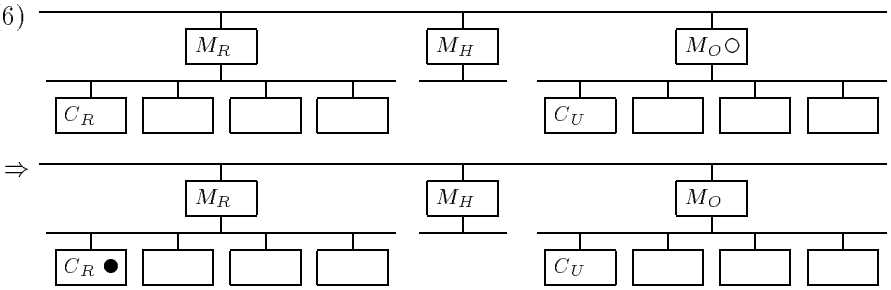
$\Rightarrow$ I****

M_R : I* $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : I****

$\Rightarrow$ VL

$\Rightarrow$ VXLDO

(6)



C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : I* $\xrightarrow{G : rdinv_req}$ M_H : I* $\xrightarrow{G : rdinv_req}$ M_O : VL $\xrightarrow{L : rdinv_req}$

M_O : VL $\xrightarrow{L : rdinv_reply(X, L, C)}$ M_O : VL $\xrightarrow{G : rdinv_reply}$ M_H : I* $\xrightarrow{G : rdinv_reply}$

$\Rightarrow$ I*

M_R : I* $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : I****

$\Rightarrow$ VL

$\Rightarrow$ VXLDO

11.4 *invalidate_req*

invalidate_req は，ラインの無効化要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1101/000	= *invalidate_req*
size	= 100	= block
ack	= ⊥	
did	= 111	= all
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= requested address

想定される用法

以下のいずれかであって，かつ対象ラインが globally-shared である場合に，home-cluster からの無効化要求に応えるためにラインを所有しているクラスタの MBPL が L2C に要求する。

(1) 読出の coherence-policy が read-invalidate であるような **normal-read** の cache-miss。

(2) **bq-read** の cache-miss に伴う読出権の移動。

(3) 読出の coherence-policy が read-invalidate であるような **normal-write** の cache-miss, または書込の coherence-policy が invalidate であるような **normal-write** の cache-miss あるいは shared-hit。

(4) swap の coherence-policy が read-invalidate であるような **normal-swap** の cache-miss あるいは shared-hit。

[注意]

(a) L2C は *invalidate_via_home_req* および *bq_write_req* に対しても，*invalidate_req* と全く同じ動作をするが，これらのパケットに対しては MMC が不要な割出しを行なうことがあるので，*invalidate_req* を使用することが望ましい。

(b) dirty-owner に対して *invalidate_req* が要求されることは想定されておらず，書き戻しが必要な無効化操作は *cflush_req* で行なう必要がある。

L2C の動作

対象ラインを所有している L2C は，ラインの無効化を行なう。

MMC/MBPL の動作

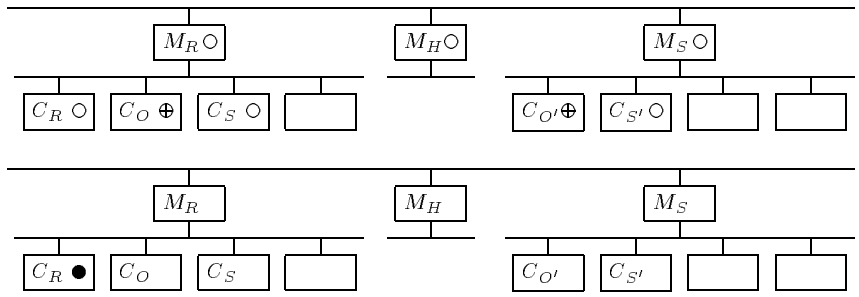
MMC は *invalidate_req* を受理しない。

応答受領時の動作

MBPL は応答を期待してはならない。

パケットの流れと状態遷移

(1)



C_R : VSGCN $\xrightarrow{L : invalidate_via_home_req}$ M_R : VG $\xrightarrow{G : rdinv_req}$ M_H : VG $\xrightarrow{G^m : invalidate_req}$

C_O : VSGCO ⇒ I****

C_S : VSGCN ⇒ I****

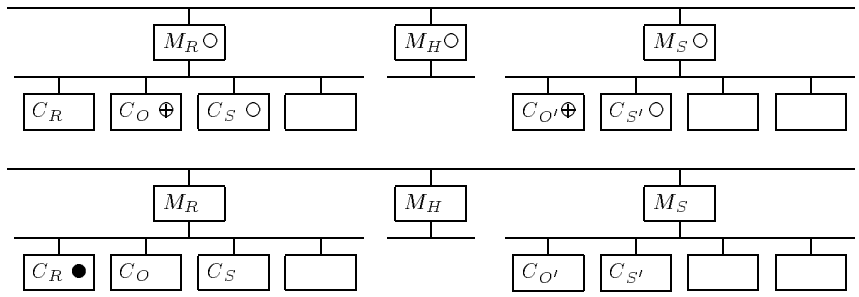
M_S : VG $\xrightarrow{L : invalidate_req}$ M_S : VG ⇒ I* $\xrightarrow{G^c : invalidate_ack}$ M_H : VG ⇒ I* $\xrightarrow{G : rdinv_reply}$

$C_{O'}$: VSGCO ⇒ I****

$C_{S'}$: VSGCN ⇒ I****

M_R : VG ⇒ VL $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : VSGCN ⇒ VXLDO

(2)



C_R : I**** $\xrightarrow{L : rdinv_req}$ M_R : VG $\xrightarrow{G : rdinv_req}$ M_H : VG $\xrightarrow{G^m : invalidate_req}$

C_O : VSGCO ⇒ I****

C_S : VSGCN ⇒ I****

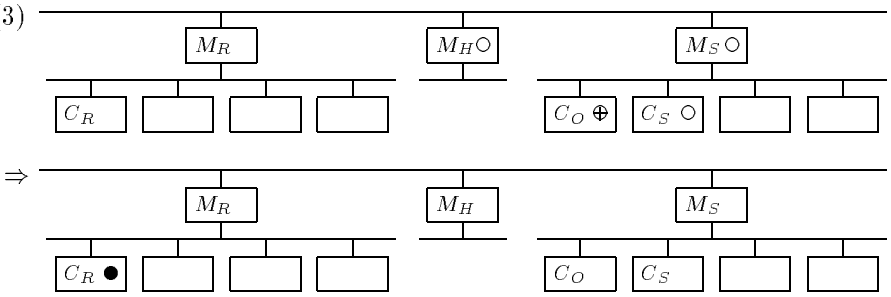
M_S : VG $\xrightarrow{L : invalidate_req}$ M_S : VG ⇒ I* $\xrightarrow{G^c : invalidate_ack}$ M_H : VG ⇒ I* $\xrightarrow{G : rdinv_reply}$

$C_{O'}$: VSGCO ⇒ I****

$C_{S'}$: VSGCN ⇒ I****

M_R : VG ⇒ VL $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R : I**** ⇒ VXLDO

(3)



C_R :I**** $\xrightarrow{L : rdinv_req}$ M_R:I* $\xrightarrow{L : rdinv_req}$ M_H:VG $\xrightarrow{G^m : invalidate_req}$

M_S :VG $\xrightarrow{L : invalidate_req}$ M_S :VG ⇒ I* C_O :VSGCO ⇒ I**** C_S :VSGCN ⇒ I**** $\xrightarrow{G^c : invalidate_ack}$ M_H:VG ⇒ I* $\xrightarrow{G : rdinv_reply}$

M_R:VG ⇒ VL $\xrightarrow{L : rdinv_reply(X, L, D)}$ C_R :I**** ⇒ VXLDO

11.5 *update_req*

update_req は，ラインの更新要求である。

送出される要求パケット

`plen`	= 10	= 5 flit
`injmask`	= 0	= nop
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 1001/000	= *update_req*
`size`	= 110	= masked block
`ack`	= ⊥	
`did`	= 111	= all
`sid`	= 100	= MMC/MBPL
`drty`	= 0	= clean
`clsh`	= 1	= global
`fe`	= *sync*	= new value
`addr`	= *addr*	= requested address
`wwe[0:3]`	= *mdfd*	= word modified
`data[0:3]`	= *data*	= new value

想定される用法

以下のいずれかであって，かつ対象ラインが globally-shared である場合に，home-cluster からの更新要求に応えるためにラインを所有しているクラスタの MBPL が L2C に要求する。

(1) 書込の coherence-policy が update または update-direct であるような **normal-write** または **s-write** の shared-hit。

(2) swap の coherence-policy が update であるような **normal-swap** の shared-hit。

[注意]

(a) L2C は *update_via_home_req*, *update_direct_req* および *update_direct_s_req* に対しても，*update_req* と全く同じ動作をするが，これらのパケットに対しては MMC が不要な割出しを行なうことがあるので，*update_req* を使用することが望ましい。

(b) *update_req* の `addr`$[5:4]$ は必ず 00 でなければならない。この制約は L2C から送出される update 系の要求パケットについて遵守されている。

L2C の動作

対象ラインを所有している L2C は，ラインの中で `wwe`$[\mathtt{i}] = 1^T$ であるような word のみを更新し，$S =$ `VSGC*` に遷移する。なお owner であるような L2C は，ownership を放棄しない。

MMC/MBPL の動作

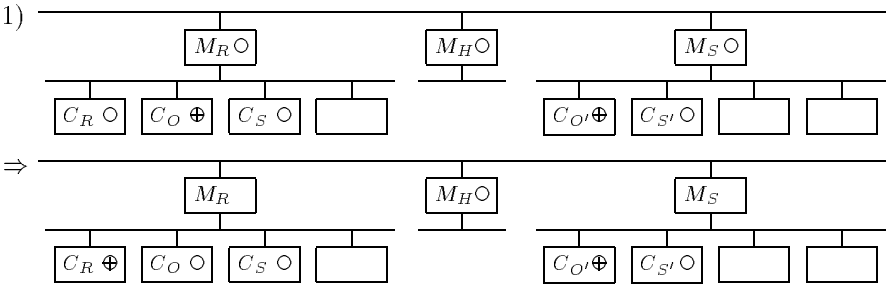
MMC は *update_req* を受理しない。

応答受領時の動作

MBPL は応答を期待してはならない。

パケットの流れと状態遷移

(1)



C_R : VSGCN $\xrightarrow{L : update_via_home_req}$ M_R : VG $\xrightarrow{G : update_req}$ M_H : VG $\xrightarrow{G^m : update_req}$

⇒ VSGCO

C_O : VSGCO

⇒ VSGCN

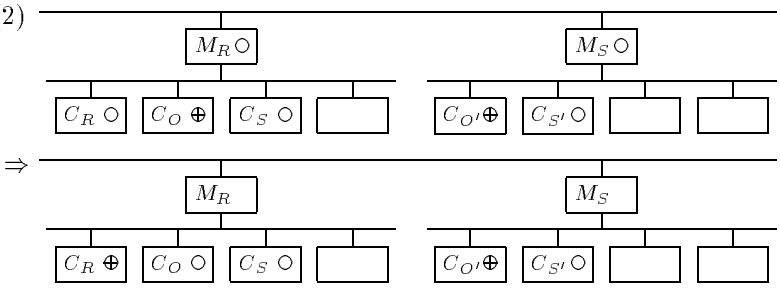
C_S : VSGCN

M_S : VG $\xrightarrow{L : update_req}$ M_S : VG $\xrightarrow{G^c : update_ack}$ M_H : VG $\xrightarrow{G : update_ack}$ M_R : VG

$C_{O'}$: VSGCO

$C_{S'}$: VSGCN

(2)



C_R : VSGCN $\xrightarrow{L : update_direct_req}$ M_R : VG $\xrightarrow{G^m : update_req}$ M_S : VG $\xrightarrow{L : update_req}$

⇒ VSGCO

C_O : VSGCO

⇒ VSGCN

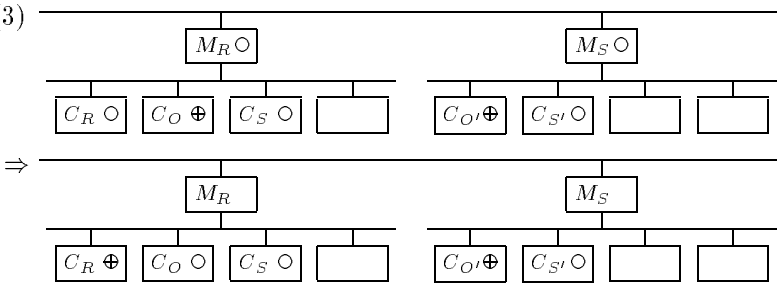
C_S : VSGCN

M_S : VG $\xrightarrow{G^c : update_ack}$ M_R : VG

$C_{O'}$: VSGCO

$C_{S'}$: VSGCN

(3)



C_R : VSGCN $\xrightarrow{L : update_direct_s_req}$ M_R : VG $\xrightarrow{G^m : update_req}$ M_S : VG $\xrightarrow{L : update_req}$

⇒ VSGCO

C_O : VSGCO

⇒ VSGCN

C_S : VSGCN

M_S : VG $\xrightarrow{G^c : update_ack}$ M_R : VG

$C_{O'}$: VSGCO

$C_{S'}$: VSGCN

11.6 *no_alloc_write_req*

no_alloc_write_req は，状態遷移を伴わない，ラインの更新要求である[†1]。

送出される要求パケット

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1100/100	= *no_alloc_write_req*
size	= 110	= masked block
ack	= $\perp$	
did	= 111	= all
sid	= 100	= MMC/MBPL
drty	= $\perp$	
clsh	= $\perp$	
fe	= *sync*	= new value
addr	= *addr*	= requested address
wwe[0:3]	= *mdfd*	= word modified
data[0:3]	= *data*	= new value

想定される用法

dummy-page に対する **s-write** が cache-miss し，かつ要求元ではないようなクラスタ C_O のみがラインを所有している場合に，home-cluster からのライン更新要求に応えるために C_O の MBPL が L2C に要求する。

[注意]

no_alloc_write_req の addr$[5:4]$ は必ず 00 でなければならない。この制約は L2C から送出される *no_alloc_write_req* について遵守されている。

L2C の動作

対象ラインを所有している L2C は，ラインの中で wwe[i] $= 1^T$ であるような word のみを更新するが，状態遷移は行なわない。

MMC/MBPL の動作

MMC は無条件に MBPL に割出す。

MBPL は cluster-memory の対象ラインの wwe[i] $= 1^T$ であるような word を更新する[†2]

応答受領時の動作

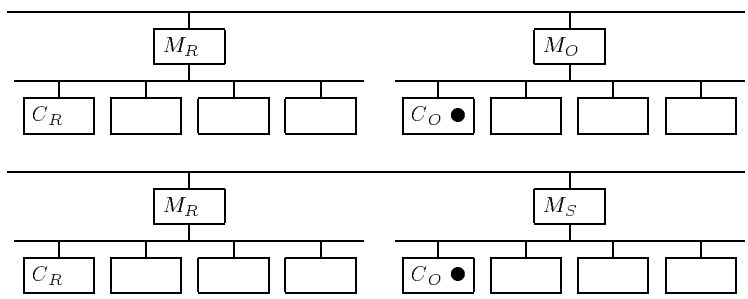
MBPL は応答を期待してはならない。

[†1] *no_alloc_write_req* は [3] では *write_xa_req* と記載されている。

[†2] dirty-owner であるような L2C が存在すればこの操作は無意味であるが，それを判別する方法はない。

パケットの流れと状態遷移

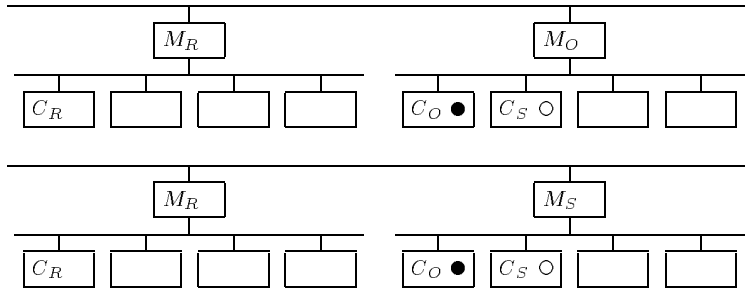
(1)



C_R : I**** $\xrightarrow{L : no_alloc_write_req}$ M_R : I* $\xrightarrow{G : no_alloc_write_req}$

M_O : VL $\xrightarrow{L : no_alloc_write_req}$ M_O : VL $\xrightarrow{G^e : no_alloc_write_ack}$ M_R : I*
C_O : VXLDO

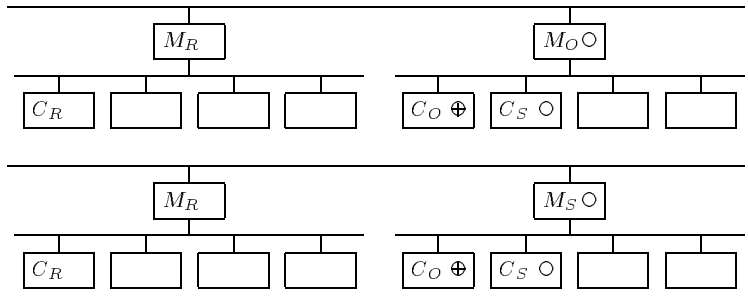
(2)



C_R : I**** $\xrightarrow{L : no_alloc_write_req}$ M_R : I* $\xrightarrow{G : no_alloc_write_req}$

M_O : VL $\xrightarrow{L : no_alloc_write_req}$ M_O : VL $\xrightarrow{G^e : no_alloc_write_ack}$ M_R : I*
C_O : VSLDO
C_S : VSLCN

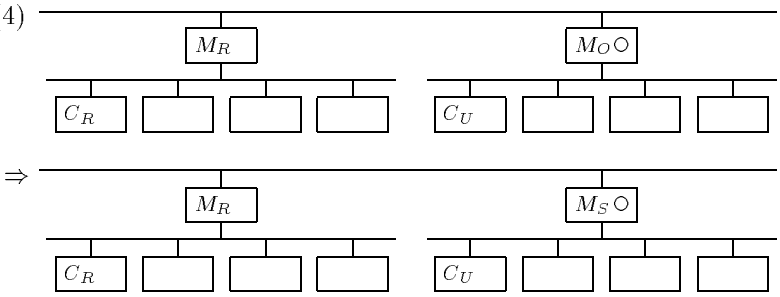
(3)



C_R : I**** $\xrightarrow{L : no_alloc_write_req}$ M_R : I* $\xrightarrow{G : no_alloc_write_req}$

M_O : VL $\xrightarrow{L : no_alloc_write_req}$ M_O : VL $\xrightarrow{G^e : no_alloc_write_ack}$ M_R : I*
C_O : VSLCO
C_S : VSLCN

(4)



$\Rightarrow$

C_R :I**** $\xrightarrow{L : no_alloc_write_req}$ M_R:I* $\xrightarrow{G : no_alloc_write_req}$

M_O:VL $\xrightarrow{L : no_alloc_write_req}$ M_O:VL $\xrightarrow{G^e : no_alloc_write_ack}$ M_R:I*

11.7 *swap_req*

swap_req は，swap を伴う，ラインの読出要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0000/011	= *swap_req*
size	= *size*	= accessed size (00:byte/10:word)
ack	= 0	= nop
did	= 111	= all
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= requested address
wwe[0]	= ⊥	
data[0]	= *data*	= new value

想定される用法

swap の coherence-policy が update であるような **normal-swap** が cache-miss し，かつ要求元ではないようなクラスタ C_O のみがラインを所有している場合に，home-cluster からの swap を伴うライン取得要求をに応えるために C_O の MBPL が L2C に要求する。

L2C の動作

対象ラインを owner として所有している L2C があり，かつ共有状態が globally-shared でなければ (S = V*L*O)[†1]，その L2C が CBUS:x_crply = 0^T として応答意志を示し，FMP/DAR を読出して以下の *swap_reply* を応答し，対象の word または byte を更新する。

plen	= 10	= 5 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 0001/011	= *swap_reply*
size	= 100	= block
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= *dirty*	= FMP:state.d
clsh	= 0	= ¬(FMP : state.l = 1)
fe	= *sync*	= FMP:sync
addr	= *addr*	= requested address
wwe[0:3]	= {*w-wait*, 0}∗4	= {FMP : ww, 0}∗4
data[0:3]	= *data*	= DAR[dwix..dwix+3]

対象ラインを所有しているが上記の条件を満たさない L2C は，*swap_req* を update 系プロトコルによる更新要求と解釈し，対象の word または byte を更新する。

[注意]

writeback_reply の wwe は，DTAG が保持すべき待ちスレッドの有無情報とは一致しない。したがって MBPL が応答値を cluster-memory に書込む際には，DTAG を更新してはならない。

MMC/MBPL の動作

対象ラインが globally-shared であるか[†1]owner として所有している L2C が存在しなければ (CBUS:x_crply =

[†1] globally-shared であることは上記の用法では想定されない。

1^F), MMC は対象ラインの STAG をアクセスする。その結果 `STAG:{valid|global}` $= \{1^T 0^F\}$ であれば，cluster-memory を読出して以下の *swap_reply* を自分自身に対して応答する[†1][†2]。

`plen`	$= 10$	$= 5$ flit
`injmask`	$= 0$	$=$ nop
`ccbl`	$= 1$	$=$ cachable
`pcmd/scmd`	$= 0001/011$	$=$ *swap_reply*
`size`	$= 100$	$=$ block
`ack`	$= 0$	$=$ nop
`did`	$= 100$	$=$ MMC/MBPL
`sid`	$= 100$	$=$ MMC/MBPL
`drty`	$= 0$	$=$ clean
`clsh`	$=$ *global*	$=$ `STAG.global`
`fe`	$=$ *sync*	$=$ `STAG.fe`
`addr`	$=$ *addr*	$=$ requested address
`wwe[0:3]`	$=$ *w-wait/r-wait*	$=$ `DTAG[dwix..dwix+3]`
`data[0:3]`	$=$ *data*	$=$ `DRAM[dwix..dwix+3]`

直接応答が不可能であるための MBPL への割出しは，想定が困難である。

応答受領時の動作

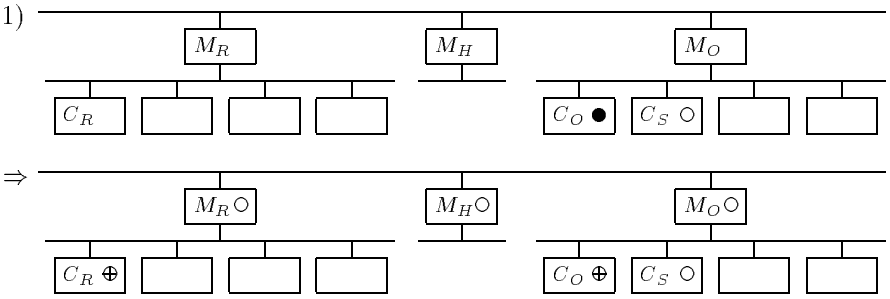
L2C あるいは自身からの *swap_reply* を MMC が受理すると，無条件で MBPL に割出す。

[†1] [1] では無条件に MBPL に割出すとあるが，これは誤りである。

[†2] `rd_trap` / `wr_trap` のチェックに関しては，どのように実装されているか定かではない。

パケットの流れと状態遷移

(1)

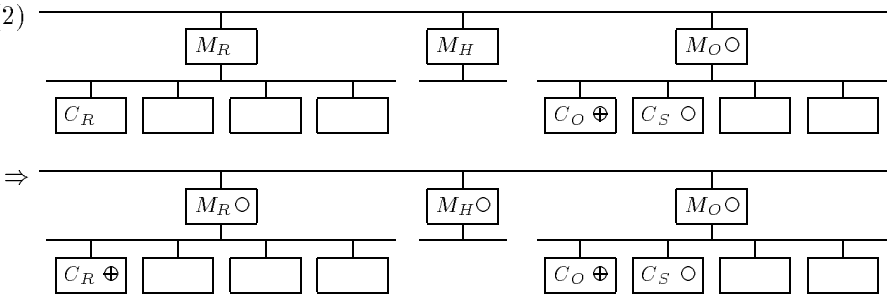


C_R :I**** $\xrightarrow{L:swap_req}$ M_R:I* $\xrightarrow{G:swap_req}$ M_H:I* $\xrightarrow{G:swap_req}$ M_O:VL $\xrightarrow{L:swap_req}$

C_O :VSLDO $\xrightarrow{L:swap_reply(S,L,D)}$ M_O:VL $\xrightarrow{G:swap_reply}$ M_H:I* $\xrightarrow{G:swap_reply}$
⇒ VSGCO ⇒ VG ⇒ VG
C_S :VSLCN
⇒ VSGCN

M_R:I* $\xrightarrow{L:swap_reply(X,G,C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(2)

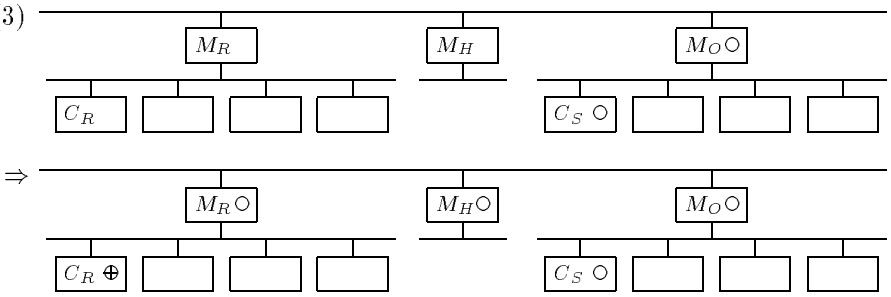


C_R :I**** $\xrightarrow{L:swap_req}$ M_R:I* $\xrightarrow{G:swap_req}$ M_H:I* $\xrightarrow{G:swap_req}$ M_O:VL $\xrightarrow{L:swap_req}$

C_O :VSLCO $\xrightarrow{L:swap_reply(S,L,C)}$ M_O:VL $\xrightarrow{G:swap_reply}$ M_H:I* $\xrightarrow{G:swap_reply}$
⇒ VSGCO ⇒ VG ⇒ VG
C_S :VSLCN
⇒ VSGCN

M_R:I* $\xrightarrow{L:swap_reply(X,G,C)}$ C_R :I****
⇒ VG ⇒ VSGCO

(3)

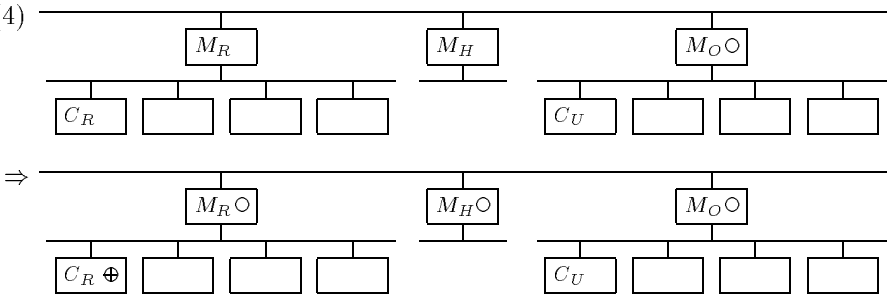


C_R :I**** $\xrightarrow{L:swap_req}$ M_R:I* $\xrightarrow{G:swap_req}$ M_H:I* $\xrightarrow{G:swap_req}$ M_O:VL $\xrightarrow{L:swap_req}$

M_O:VL $\xrightarrow{L:swap_reply(S,L,C)}$ M_O:VL $\xrightarrow{G:swap_reply}$ M_H:I* $\xrightarrow{G:swap_reply}$
C_S :VSLCN ⇒VG ⇒VG
⇒VSGCN

M_R:I* $\xrightarrow{L:swap_reply(X,G,C)}$ C_R :I****
⇒VG ⇒VSGCO

(4)



C_R :I**** $\xrightarrow{L:swap_req}$ M_R:I* $\xrightarrow{G:swap_req}$ M_H:I* $\xrightarrow{G:swap_req}$ M_O:VL $\xrightarrow{L:swap_req}$

M_O:VL $\xrightarrow{L:swap_reply(X,L,C)}$ M_O:VL $\xrightarrow{G:swap_reply}$ M_H:I* $\xrightarrow{G:swap_reply}$
⇒VG ⇒VG

M_R:I* $\xrightarrow{L:swap_reply(X,G,C)}$ C_R :I****
⇒VG ⇒VSGCO

11.8 *cflush_req*

cflush_req は，書き戻しを伴うラインの無効化要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 1	= cachable
pcmd/scmd	= 1000/011	= *cflush_req*
size	= 100	= block
ack	= ⊥	
did	= 111	= all
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= requested address

想定される用法

CPU が要求した複数ラインに対する **cflush** の実現などのための使用が想定される。

[注意]

同様の機能は *rdinv_req* を用いても実現できるが，*cflush_req* では dirty なラインの書き戻しが自動的に行なわれるため効率的である。

L2C の動作

対象ラインを所有している L2C は，ラインの無効化を行ない，必要であれば writeback も行なう。

MMC/MBPL の動作

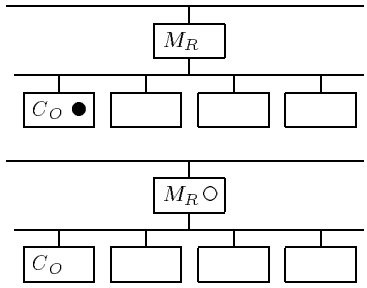
MMC は *cflush_req* を受理しない。

応答受領時の動作

MBPL は応答を期待してはならない。

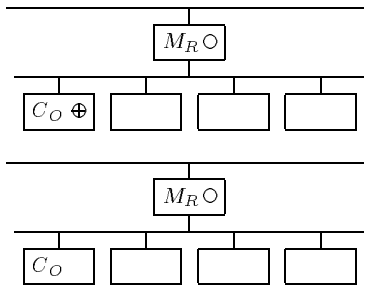
パケットの流れと状態遷移

(1)



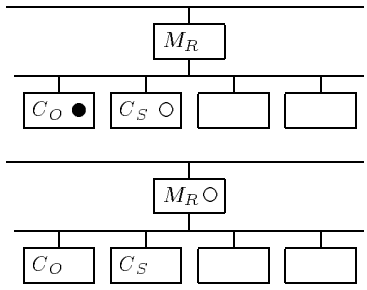
M_R : VL $\xrightarrow{L : cflush_req}$ C_O : VXLDO $\xrightarrow{L : writeback_reply}$ M_R : VL
⇒ I****

(2)



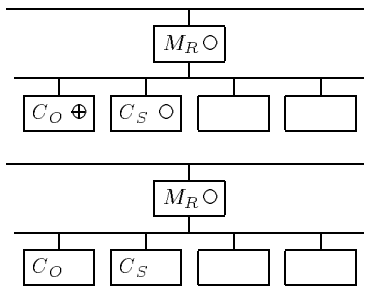
M_R : VL $\xrightarrow{L : cflush_req}$ C_O : VXLCO
⇒ I****

(3)



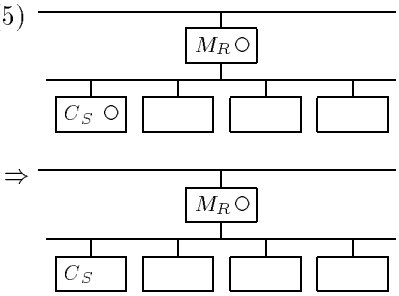
M_R : VL $\xrightarrow{L : cflush_req}$ C_O : VSLDO $\xrightarrow{L : writeback_reply}$ M_R : VL
⇒ I****
C_S : VSLCN
⇒ I****

(4)



M_R : VL $\xrightarrow{L : cflush_req}$ C_O : VSLCO
⇒ I****
C_S : VSLCN
⇒ I****

(5)



$\Rightarrow$

M_R : VL $\xrightarrow{L : cflush_req}$ M_R : VL $\quad$ M_R : VL

C_S : VSLCN

$\Rightarrow$ I****

11.9 *injection_req*

injection_req は，ラインの injection 要求である。

送出される要求パケット

`plen`	= 10	= 5 flit
`injmask`	= 1/0	= injection/not
`ccbl`	= 1	= cachable
`pcmd/scmd`	= 0010/010	= *injection_req*
`size`	= 100	= block
`ack`	= ⊥	
`did`	= 111	= all
`sid`	= 100	= MMC/MBPL
`drty`	= 0	= clean
`clsh`	= *global*	= `STAG.global`
`fe`	= *sync*	= `STAG.fe`
`addr`	= *addr*	= requested address
`wwe[0:3]`	= *w-wait/r-wait*	= `DTAG[dwix..dwix+3]`
`data[0:3]`	= *data*	= `DRAM[dwix..dwix+3]`

想定される用法

MBPL が行なった複数ラインの prefetch 結果の L2C への注入，あるいはクラスタ間での all-read 操作の実現のための使用が想定される。

L2C の動作

自身の device-ID を i としたとき，`injmask`$[i] = 0$ であるような L2C は何もしない。また，すでに対象ラインを所有している L2C も何もしない。

対象ラインを所有していない L2C は，injection によって除去されるラインの状態 S_v と，内部レジスタ `INJLEV` の値 L に基づいて injection の可否を以下のように判断する。

(1) $L = 0$
$S_v =$ `I****` であり，かつ直前に有効であった時に同じアドレスのラインが存在したならば可。

(2) $L = 1$
$S_v =$ `I****` であれば可。

(3) $L = 2$
$S_v =$ `I****` あるいは $S_v =$ `V**C*` であれば可。

(4) $L = 3$
S_v に関わらず可。

injection が可であると判断した場合，ラインを *injection_req* の内容に置き換え，必要ならば writeback を行なう。また `clsh` ≡ `g` に基づきラインの状態を $S =$ `VSgCN` とする。

[注意]

上記の条件に関わらず，L2C は *injection_req* を無視することがある。

MMC/MBPL の動作

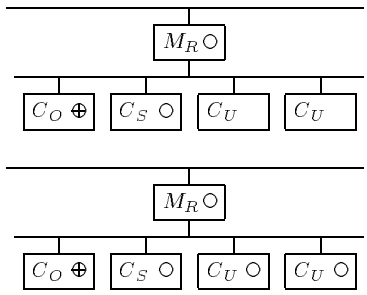
MMC は *injection_req* を受理しない。

応答受領時の動作

MBPL は応答を期待してはならない。

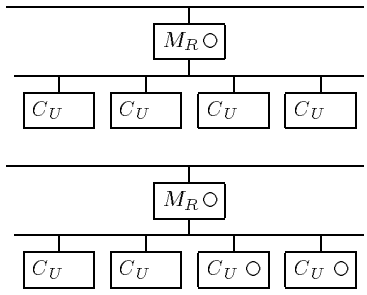
パケットの流れと状態遷移

(1)



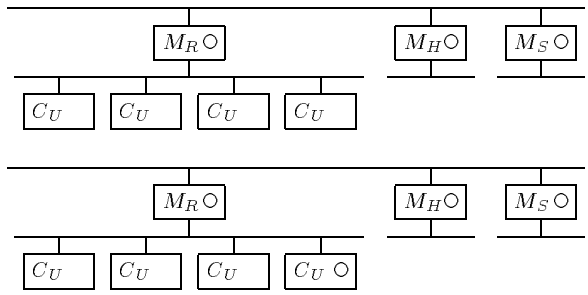
M_R : VL $\xrightarrow{L : injection_req(1111, S, L, C)}$ M_U : I****
⇒ VSLCN

(2)



M_R : VL $\xrightarrow{L : injection_req(0011, X, L, C)}$ M_U : I****
⇒ VSLCN

(3)



M_R : VG $\xrightarrow{L : injection_req(0001, X, G, C)}$ M_U : I****
⇒ VSGCN

11.10 *demap_req*

demap_req は，CPU に対する demap 要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1110/001	= *demap_req*
size	= ⊥	
ack	= ⊥	
did	= *did*	= recipient's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= ⊥	
wwe[0]	= ⊥	
data[0]	= *data*	= demap info

想定される用法

大域的な page table の更新の際に，CPU が保持するアドレス変換情報を無効化するために使用することが想定される。

L2C の動作

自身の device-ID を i としたとき，did = i あるいは did = 111 (all) であれば *demap_req* を受理し，CPU に demap 要求を伝える。その後，CPU が demap 操作完了を報告すると，以下の *demap_ack* により操作完了を応答する。

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1111/001	= *demap_ack*
size	= ⊥	
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= 0	
fe	= ⊥	
addr	= ⊥	

[注意]

demap_req が保持する情報は，一切 *demap_ack* には反映されない。したがって，未応答の *demap_req* が複数存在する場合，特定の *demap_req* の完了を知ることはできない。

MMC/MBPL の動作

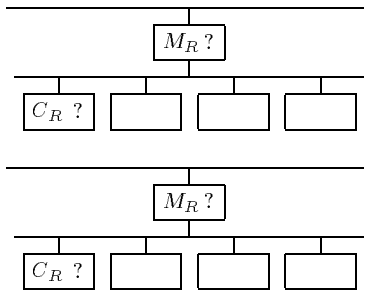
MMC は *demap_req* を受理しない。

応答受領時の動作

L2C からの *demap_ack* を MMC が受理すると，無条件で MBPL に割出す。

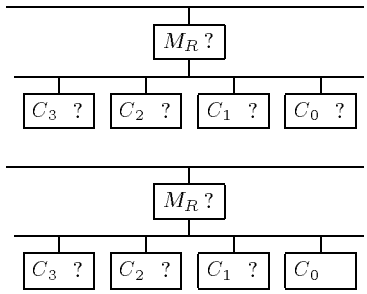
パケットの流れと状態遷移

(1)



$$M_R : ?? \xrightarrow{L : demap_req(C_R)} C_R : ????? \xrightarrow{L : demap_ack} M_R : ??$$

(2)



$$M_R : ?? \xrightarrow{L : demap_req(\text{all})} \begin{array}{l} C_3 : ????? \\ C_2 : ????? \\ C_1 : ????? \\ C_0 : ????? \end{array} \xrightarrow{L : demap_ack \times 4} M_R : ??$$

11.11 *control_read_req*

control_read_req は，L2C の内部資源読出要求である。

送出される要求パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1110/000	= *control_read_req*
size	= 011	= doubleword
ack	= 0	= nop
did	= *did*	= recipient's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= resource addr

想定される用法

L2C の初期化あるいはデバッグに使用されることが想定される。

L2C の動作

$0 \leq$ did ≤ 3 であれば，did により定まる L2C が *control_read_req* を受理し，addr により定まる内部資源の値を以下の *control_read_reply* により応答する。なお addr と内部資源の対応については [5] を参照せよ。

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1111/000	= *control_read_reply*
size	= 011	= doubleword
ack	= 0	= nop
did	= 100	= MMC/MBPL
sid	= *pid*	= own ID
drty	= ⊥	
clsh	= 0	
fe	= ⊥	
addr	= *addr*	= requested address
wwe[0]	= ⊥	
data[0]	= *data*	= register contents

MMC/MBPL の動作

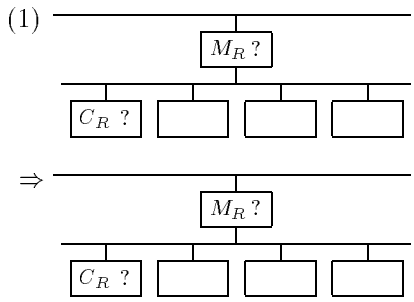
MMC は L2C に宛てられた *control_read_req* を受理しない[†1]。

応答受領時の動作

L2C からの *control_read_reply* を MMC が受理すると，無条件で MBPL に割出す。

[†1] did = 100 である *control_read_req* は MMC に受理されて MBPL に割出されるが，そのような要求を MMC/MBPL 自身が発するのは無意味である。

パケットの流れと状態遷移



$$M_R\text{:??} \xrightarrow{L : control_read_req} C_R\text{ :?????} \xrightarrow{L : control_read_reply} M_R\text{:??}$$

11.12 *interrupt_req*

interrupt_req は，CPU に対する割込要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1110/011	= *interrupt_req*
size	= 011	= doubleword
ack	= ⊥	
did	= *did*	= recipient's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= 0x10000110	= INTSET
wwe[0]	= ⊥	
data[0]	= *data*	= interrupt flag select

想定される用法

CPU に対する外部割込が想定される。

L2C の動作

$0 \leq$ did ≤ 3 であれば did により定まる L2C が，また did $= 111$ (all) であれば全ての L2C が，*interrupt_req* を受理し，data$[7:0]$ と内部レジスタ INTST の論理和を INTST にセットする。INTST$[i]$ $(0 \leq i \leq 7)$ はレベル $i+8$ の割込に対応する。INTST の詳細については [5] を参照せよ。

[注意]

(1) L2C は *interrupt_req* と *control_write_req* を完全に同一視するので，割込要求のためには *interrupt_req* の addr が 0x10000110 でなければならない。

(2) *interrupt_req* が *control_write_req* と解釈されることと，MMC が *interrupt_req* を受理しないことを利用して，全ての L2C の内部資源に対する同一操作を，did $= 111$ (all) であるような *interrupt_req* を用いて行なうことができる (11.13 節参照)。

MMC/MBPL の動作

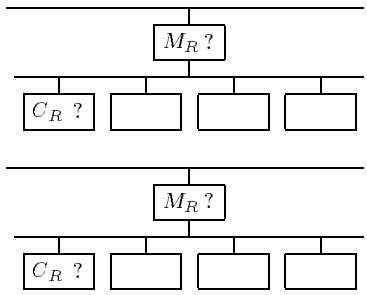
MMC は *interrupt_req* を受理しない。

応答受領時の動作

MBPL は応答を期待してはならない。

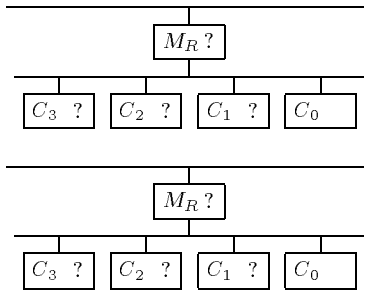
パケットの流れと状態遷移

(1)



M_R : ????? $\xrightarrow{L : interrupt_req(C_R)}$ C_R : ??

(2)



M_R : ?? $\xrightarrow{L : interrupt_req(\text{all})}$ C_3 : ?????
C_2 : ?????
C_1 : ?????
C_0 : ?????

11.13 *control_write_req*

control_write_req は，L2C の内部資源書込要求である。

送出される要求パケット

plen	= 01	= 2 flit
injmask	= 0	= nop
ccbl	= 0	= non cachable
pcmd/scmd	= 1110/010	= *control_write_req*
size	= 011	= doubleword
ack	= ⊥	
did	= *did*	= recipient's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= resource addr
wwe[0]	= ⊥	
data[0]	= *data*	= VBUS.data

想定される用法

L2C の初期化あるいはデバッグに使用されることが想定される。

L2C の動作

$0 \leq$ did ≤ 3 であれば did により定まる L2C が，また did = 111 (all) であれば全ての L2C が，*control_write_req* を受理し，addr により定まる内部資源に data を書込む。なお addr と内部資源の対応については [5] を参照せよ。

[注意]

did = 111 (all) であるような *control_write_req* は，全ての L2C に対して同じ操作を行なうために使用することができるが，MMC もそのような要求を受理し，かつ無条件に MBPL に割出すことに注意しなければならない。そこで，L2C が *interrupt_req* と *control_write_req* を同一視することと，MMC が *interrupt_req* を受理しないことを利用して，このような操作を *interrupt_req* を用いて行なうこともできる (11.12 節参照)。

MMC/MBPL の動作

MMC は L2C に宛てられた *control_read_req* を受理しないが[†1]，did = 111 (all) であれば受理して無条件に MBPL に割出す。

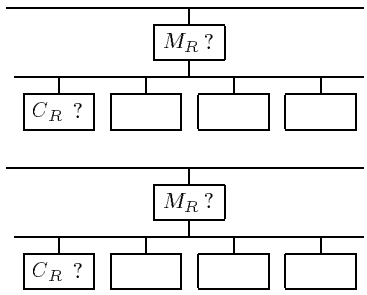
応答受領時の動作

MBPL は応答を期待してはならない。

[†1] did = 100 である *control_read_req* は MMC に受理されて MBPL に割出されるが，そのような要求を MMC/MBPL 自身が発するのは無意味である。

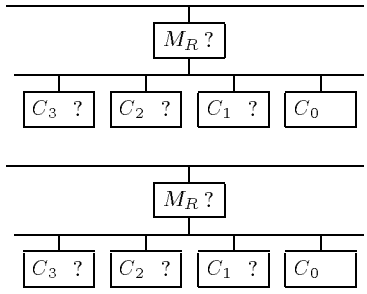
パケットの流れと状態遷移

(1)



M_R : ????? $\xrightarrow{L : control_write_req}$ C_R : ??

(2)



M_R : ?? $\xrightarrow{L : control_write_req(\text{all})}$ C_3 : ?????
C_2 : ?????
C_1 : ?????
C_0 : ?????
M_R : ??

11.14 *error_reply*

error_reply は要求ではなく，MMC/MBPL が何らかの理由で正しい応答ができなかった場合の応答である。

送出される応答パケット

plen	= 00	= 1 flit
injmask	= 0	= nop
ccbl	= ⊥	
pcmd/scmd	= 1111/110	= *error_reply*
size	= ⊥	
ack	= ⊥	
did	= *did*	= recipient's ID
sid	= 100	= MMC/MBPL
drty	= ⊥	
clsh	= ⊥	
fe	= ⊥	
addr	= *addr*	= error addr

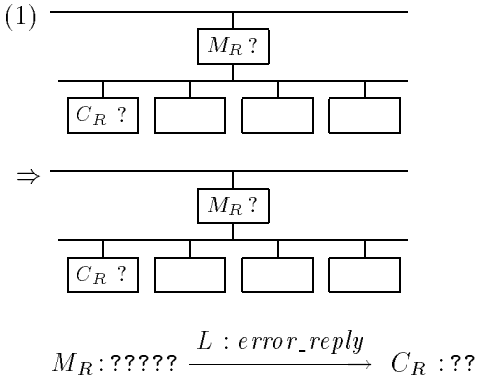
想定される用法

cluster-memory における 2 ビット ・エラー検出などによって正しい応答ができない場合に，本来の応答の代わりに用いることが想定される。

L2C の動作

did により定まる L2C が受理し，応答に対応する要求を生じる要因となった CPU のアクセスが read または swap であれば，CPU に uncorrectable-error を通知する。そうでなければ不正パケット割込が生じる。

MMC/MBPL の動作

MMC が *error_reply* を受理することは想定されていない。

パケットの流れと状態遷移