超並列計算機 JUMP-1 の
実装と評価

鈴木 紀章
Noriaki Suzuki
開放環境科学専攻
School of Science for Open and Environmental Systems

2003 年度

論文要旨

キャッシュコヒーレントな分散共有メモリをもつマルチプロセッサシステム CC-NUMA (Cache Coherent Non-Uniform Memory Access model) は，中，大規模な並列計算機の代表的な構成方法の 1 つである．CC-NUMA は，バス結合型の小規模並列計算機に比べて多数のプロセッサを接続でき，またこれらの小規模並列計算機で開発されたプログラムを容易に移植できる利点もある．このため，SGI の Origin，Sequent の NUMA-Q など，商用化が進んでいる．

文部省重点領域研究の一環として 1994 年から開発が行われた JUMP-1 は，数千プロセッサを超す超並列計算機上に効率の良い分散共有記憶や同期，メッセージ転送を実現するためのテストベッドである．JUMP-1 では従来のものよりはるかに多数のプロセッサ上でキャッシュコヒーレントな分散共有記憶を実現するため様々な新しい手法が提案されている．

JUMP-1 は数千プロセッサ規模への拡張を可能とするために，結合網 RDT (Recursive Diagonal Torus) で相互接続されたクラスタ構造をとる．RDT は二次元トーラスの Fat-tree 状の階層構造をもつ結合網で，メッセージマルチキャストと応答収集機能をもつルータチップを用いて実現されている．分散共有メモリ管理には，専用プロセッサである MBP(Memory Based Processor) -light を用いている．MBP-light は，シンプルなコアプロセッサをもち，さらにメモリシステム，バス，ネットワークパケットを扱うハードウェアコントローラを備えている．さらに各クラスタは並列 I/O サブシステムを構成する高速シリアルリンク STAFF-Link により，ディスクおよび画像データ用 Frame Buffer と接続する．また，各クラスタに対してメンテナンス用ホスト PC を設け，システムのブートアップやモニタを行う．

本論文では，この JUMP-1 システムの実装を行った内容について触れ，実機を用いた評価結果について述べる．JUMP-1 は，2000 年 3 月に 16 クラスタ 64 プロセッサのハードウェアが完成したため，以後実機を用いて様々な評価を行っていくことになった．本研究では，完成した実機を用いて，RDT ネットワークの応答パケットの自動生成，自動収集機構に関する評価，MBP-light の命令セットアーキテクチャの評価等を行った．本論文では，これらを含めて JUMP-1 プロジェクトで得られた知見についてまとめる．

RDT ネットワークのバンド幅は，最大パケット長である 15 フリットの場合で 100MBytes/sec となることが分かった．応答パケットの自動生成機構はソフトウェア処理に比べて 1.8 倍程度，自動収集機構は 2.4 倍程度高速であることが分かった．JUMP-1 のメモリ管理プロセッサ MBP-light は，Buffer-Register Architecture をとることにより，Home クラスタで 5.64%，Remote クラスタで 6.27%の性能向上を達成している．特殊命令では，ハッシュ値を求める命令のみが有効に働いており，これにより 2.80%の性能向上を達成している．しかし，頻繁に利用されている命令は共通 RISC 命令であり，別のアーキテクチャをとることにより，より高性能なものを実装できる可能性があることが分かった．また，JUMP-1 の I/O システムは，クラスタと同数の入出力ユニットを用いた場合，read でクラスタ当たり 2.1Mbps，write では 4.5Mbps の実行転送速

度を得られ，4 クラスタ 4 入出力ユニットまで直線的に性能が向上することが分かった．JUMP-1 の DSM(Distributed Shared Memory) では，Home クラスタに有効なデータがある場合のリモートリードで 469 サイクル (50MHz 動作の場合 9.38μsec)，Home 以外のクラスタが有効なデータを保持している場合で 850 サイクル (50MHz 動作の場合 17.00μsec) となっている．行列積プログラムの実行では，4 プロセッサのシステムで 3.72 倍，8 プロセッサのシステムで 6.61 倍の台数効果が得られることが分かった．

Abstract

A Cache Coherent Non-Uniform Memory Access machine (CC-NUMA) is one of hopeful candidates for future common high performance machines. Unlike bus-connected multiprocessors, the system performance can be enhanced scalably as to the number of processors. Moreover, parallel programs developed in small multiprocessors can be transported easily. Presently, a number of commercial CC-NUMA machines have been developed.

JUMP-1 is a prototype of a massively parallel CC-NUMA developed by collaboration of 7 Japanese universities. The major goal of this project is building an efficient cache coherent distributed shared memory on a large system with more than 1000 processors. A lot of novel techniques are introduced in the DSM(Distributed Shared Memory) of JUMP-1 for this purpose.

In order to satisfy both high degree of performance and flexibility, JUMP-1 has several distinctive structures. Interconnection network called RDT (Recursive Diagonal Torus) includes both torus and a kind of fat tree structure with recursively overlaid two-dimensional square diagonal tori structure. A dedicated processor called MBP(Memory Based Processor)-light is proposed to manage the DSM of JUMP-1. MBP-light consists of a simple dedicated core processor and hardwired controllers that handle memory systems, bus and network packets. Every cluster board which consists of processors, memory etc. has a serial link called STAFF-Link(Serial Transparent Asynchronous First-in First-out Link). Each channel of STAFF-Link is connected to an independent I/O unit, and works in parallel to obtain the scalable I/O bandwidth. For efficient monitoring and maintenace of a lot of processing elements, a dedicated interface system connecting with the host called MBIF (Maintenace Bus Interface) is equipped.

This thesis describes an implementation and evaluation of JUMP-1. The first prototype of JUMP-1 with 4 clusters/16 processors started to work in 1999, and the system that provides 16 clusters/64 processors was available from 2000. The first and the second prototype are used for evaluation in this paper.

Through the evaluation, the following results were appeared. The maximum bandwidth of RDT network is 100M Bytes/sec in the case that packets consist of 15 flits. Multicasting mechanism saves 52 cycles compared with unicasting by core processor software even when it sends only one packet, and does not require extra cycles for multicasting up to 8 packets. It also contributes to resolve the network traffic congestion. The automatic acknowledge generation logic saves 40 cycles compared with generation by the software of core processor. The automatic acknowledge packet combining mechanism achieves 2.4 times performance compared with that by the software. The Buffer-Register Architecture proposed for MBP Core improves

performance with 5.64% in the home cluster and 6.27% in a remote cluster. Only a special instruction for hashing cluster address is efficient and improves the performance with 2.80%, but other special instructions are almost useless. The I/O bandwidth for reading is 2.1Mbps and that for writing is 4.5Mbps. The whole I/O bandwidth increased linearly as the increase number of clusters and I/O units. The remote read cycles with JUMP-1's DSM system is 469 cycles (9.38μsec at 50MHz) in the case that there is the valid data in the home cluster, and 850 cycles (17.00μsec at 50MHz) in the case that the valid data is not in the home cluster. The execution result of matrix multiplication program shows that, the rate of speed-up is 3.72 with 4 processors, and 6.61 with 8 processors.

目 次

第1章 緒 論 **1**
1.1 重点領域研究の目的と JUMP-1 の位置づけ 1
1.2 JUMP-1 の基本構想 2
1.3 JUMP-1 開発の経緯 3
1.4 論文の位置づけ 4
1.5 JUMP-1 の概要と本論文の構成 5

第2章 関連研究 **7**
2.1 超並列計算機プロジェクトに関する調査 7
2.1.1 CP-PACS プロジェクト 8
2.1.1.1 全体構成 8
2.1.1.2 ノードプロセッサ 8
2.1.1.3 ハイパークロスバ結合網 8
2.1.1.4 オペレーティングシステム 9
2.1.1.5 プログラミング環境 9
2.1.2 RWC-1 プロジェクト 11
2.1.2.1 プロセッサアーキテクチャ 11
2.1.2.2 相互結合網 12
2.1.2.3 入出力アーキテクチャ 13
2.1.3 まとめ 13
2.2 CC-NUMA に関する調査 14
2.2.1 Stanford 大学の DASH 14
2.2.2 Sequent 社の NUMA-Q 14
2.2.3 Tronto 大学の NUMAchine 17
2.2.4 Stanford 大学の FLASH 18
2.2.5 HAL 研の Mercury 19
2.2.6 Silicon Graphics 社の Origin2000 20
2.2.7 Silicon Graphics 社の NUMAflex(Origin3000) 22
2.2.8 まとめ 22

第3章 超並列計算機 JUMP-1 の全体設計 **25**
3.1 JUMP-1 の目標 25
3.2 JUMP-1 の設計方針 25
3.3 JUMP-1 の構成 26

3.4 要素プロセッサおよび2次キャッシュ 29
3.5 バリア同期機構 29
3.6 結合網 RDT(Recursive Diagonal Torus) 30
3.7 メインテナンスシステム 30
3.8 Disk 入出力サブシステム 30
3.9 MBP-light 30

第4章 超並列計算機 JUMP-1 の構成要素 31
4.1 SuperSPARC+ 31
4.2 2次キャッシュ 32
4.2.1 一貫性制御方式 32
4.2.2 トランザクションの逐次化 35
4.2.3 2次キャッシュの実装諸元 36
4.2.4 キャッシュコヒーレンス制御 36
4.2.5 2次キャッシュのブート 38
4.3 クラスタバス (CBus) 38
4.3.1 アービトレーション機構 38
4.4 Elastic Barrier 39
4.4.1 Elastic Barrier の概要 39
4.4.2 Elastic Barrier の実装 42

第5章 MBP-light 45
5.1 JUMP-1 における分散共有記憶管理の概要 45
5.1.1 分散共有記憶管理のための基本構想 45
5.1.2 アドレス変換 46
5.1.3 ページ管理とデータ転送 47
5.1.4 page fault 処理 47
5.1.5 page entry 48
5.1.6 page 属性 48
5.1.7 クラスタ内一貫性管理 50
5.1.8 メモリ参照の順序性と単一性 51
5.1.9 分散共有記憶の高機能化 51
5.2 設計方針 51
5.3 MBP-light の構成 53
5.3.1 MBP-light の全体構成 53
5.3.2 RDT Interface 53
5.3.2.1 RDT Interface の構成 53
5.3.2.2 Packet Buffer Register(PBR) 56
5.3.2.3 Send/Receive Unit 56
5.3.2.4 Packet Handler 57
5.3.2.5 Ack Collector 59
5.3.2.6 Ack Generator 64

5.3.2.7 shootdown と setup 67
5.3.2.8 MBP Core Interface 69
5.3.3 MMC (Main Memory Controller) 69
5.3.3.1 内部構成 69
5.3.3.2 メモリ構成 70
5.3.3.3 SBUSCTL(cluSter BUS ConTroLler) 72
5.3.3.4 MTC(Memory Timing Controller) 76
5.3.4 MBP Core 79
5.3.4.1 MBP Core の構成 79
5.3.4.2 Buffer-Register Architecture 79
5.3.4.3 パイプライン構成 83
5.3.4.4 内部メモリ 84
5.3.4.5 Table Jump と割込み 87
5.3.4.6 Timer/Monitor 機能 89
5.3.4.7 命令セットアーキテクチャ 93
5.3.5 ローカルバス 101
5.3.5.1 MBP-light のブート 101
5.4 MBP-light の実装 102
5.5 MBP-light の命令セットアーキテクチャの評価 103
5.5.1 評価環境 103
5.5.2 各命令の比率 104
5.5.3 演算種類 107
5.5.4 PBR 操作命令 107
5.5.5 特殊命令 109
5.5.6 その他の実装方法の検討 111
5.5.7 命令セットアーキテクチャの評価のまとめ 112

第 6 章 結合網 RDT(Recursive Diagonal Torus) 113
6.1 縮約階層 bitmap directory 方式 113
6.1.1 階層 bitmap directory 方式 113
6.1.2 RHBD 方式 114
6.2 RDT ネットワーク 116
6.2.1 RDT の構成 116
6.2.2 RDT における RHBD 方式の実現 119
6.2.3 RDT router chip 122
6.2.3.1 アービトレーションとクロスバの設定 124
6.2.3.2 パケットのマルチキャスト 125
6.2.3.3 multicast map の生成 125
6.2.3.4 応答パケットの収集 127
6.2.3.5 パケットの shootdown と setup 128
6.2.3.6 エラーの検出 128
6.2.3.7 RDT router の実装 129

6.3 RDT ネットワークの評価 . . . 131
6.3.1 評価に使用したシステムの構成 . . . 131
6.3.2 RDT ネットワークの最大バンド幅 . . . 131
6.3.3 マルチキャスト機構 . . . 132
6.3.3.1 マルチキャストの送信処理時間 . . . 132
6.3.3.2 ネットワークの飽和 . . . 133
6.3.3.3 マルチキャスト機構のハードウェア要求量の評価 . . . 134
6.3.4 応答パケット自動生成機構 (無駄パケット自動廃棄機構) の評価 . . . 134
6.3.4.1 性能評価 . . . 134
6.3.4.2 ハードウェア量の評価 . . . 135
6.3.5 応答パケット自動収集機構の評価 . . . 135
6.3.5.1 RDT ルータチップによる応答パケット自動収集機構の評価 135
6.3.5.2 MBP-light による応答パケット自動収集機構の評価 . . . 137
6.3.5.3 ハードウェア量の評価 . . . 138
6.3.6 RDT の評価のまとめと他のシステムとの比較 . . . 138

第 7 章 メインテナンスシステム 139
7.1 メインテナンスシステムの方針 . . . 139
7.1.1 目的 . . . 139
7.1.2 提案 . . . 140
7.1.2.1 全体構成 . . . 140
7.1.2.2 MC(Maintenance Controller) . . . 140
7.1.2.3 Maintenance Bus InterFace . . . 141
7.1.2.4 MBIF-MC 間の転送プロトコル . . . 141
7.1.3 提案のまとめ . . . 143
7.2 メインテナンスシステムの実装形態 . . . 143
7.2.1 MBIF-MC 間の実装形態 . . . 143
7.2.2 MBIF-MC 間の信号線 . . . 144
7.2.3 MBIF とローカルバスの接続 . . . 146
7.3 MC(Maintenance Controller) の実装 . . . 147
7.3.1 MC の全体構成 . . . 147
7.3.2 MC のソフトウェアプロトコル制御機構 . . . 148
7.3.3 MC の I/O アドレス可変機構 . . . 149
7.4 MBIF(Maintenance Bus InterFace) の実装 . . . 150
7.4.1 QuickLogic . . . 150
7.4.2 Verilog-HDL . . . 150
7.4.3 ステートマシンの構成法 . . . 151
7.4.4 MBIF の内部構成 . . . 152
7.4.5 Sender 部の実装 . . . 152
7.4.5.1 Sender 部の転送動作 . . . 152
7.4.6 Receiver 部の実装 . . . 155
7.4.6.1 Receiver 部の受信動作 . . . 155

7.4.7 Local Bus Interface 部の実装 157
7.4.8 Controller 部の実装 157
7.4.8.1 MC 側コマンドデコーダ 159
7.4.8.2 MC 側コマンド実行用ステートマシン 160
7.4.8.3 MBP-light 側コマンド実行回路 162
7.4.9 MBIF のレジスタ 162
7.5 メインテナンスシステムの評価 165
7.5.1 MBIF の評価 165
7.5.1.1 MBIF の使用ゲート数 165
7.5.1.2 MBIF の遅延時間 165
7.5.2 MBIF-MC 間の最大転送容量 166

第 8 章 Disk 入出力サブシステム 167
8.1 システムの概要 167
8.1.1 設計方針 167
8.1.2 STAFF-Link 167
8.1.3 入出力ネットワーク 169
8.1.4 I/O ソフトウェアの実装 169
8.1.4.1 I/O 動作の流れ 170
8.1.4.2 実装 170
8.2 I/O 性能の評価 171
8.2.1 主眼 171
8.2.2 システム 171
8.2.3 方法と結果 172
8.2.4 I/O の評価のまとめ 174
8.2.4.1 スケーラビリティ 174
8.2.4.2 オーバーヘッド 174

第 9 章 分散共有メモリ (DSM: Distributed Shared Memory) 175
9.1 分散共有メモリ管理 175
9.1.1 JUMP-1 のコヒーレンス制御 175
9.1.2 クラスタ間コヒーレンス制御 175
9.1.3 DSM 管理プログラム 177
9.1.3.1 DSM 管理プログラムの基本構造 177
9.1.3.2 ペンディング・トランザクション・テーブル (PTT) . . . 178
9.1.3.3 パケットのすれちがい及び追越しの防止 178
9.1.3.4 テーブル 180
9.1.3.5 ディレクトリ 181
9.2 JUMP-1 のバグ 182
9.3 分散共有メモリの評価 182
9.3.1 トランザクションの処理時間 182
9.3.2 行列積プログラムによる評価 183

第 10 章 JUMP-1 プロジェクトのまとめ 187
10.1 JUMP-1 の新しい技術の評価のまとめ . . . 187
10.1.1 大規模な CC-NUMA 用の分散メモリ管理方式 . . . 187
10.1.1.1 キャッシュライン管理 . . . 187
10.1.1.2 積極的なデータ共有 . . . 187
10.1.1.3 RHBD 方式によるディレクトリ管理 . . . 188
10.1.1.4 MBP-light による制御 . . . 188
10.1.1.5 MBP-core の Buffer-Register Architecture . . . 188
10.1.2 結合網 RDT と RDT ルータ . . . 189
10.1.3 メインテナンスシステム . . . 189
10.1.4 I/O . . . 189
10.2 JUMP-1 の実装遅れとサイズ縮小 . . . 190
10.3 大学間共同プロジェクトの問題点 . . . 191

第 11 章 結 論 193
11.1 本研究の成果 . . . 193
11.2 最後に . . . 195

付 録 A RDT network 用の packet format 207
A.1 multicast packet . . . 207
A.2 acknowledge packet . . . 210
A.3 shootdown/setup packet . . . 212
A.4 Ack Cache shootdown packet . . . 214
A.5 timeout packet . . . 214
A.6 command packet . . . 214
A.7 status report packet . . . 217

付 録 B MMC の内部 register 219
B.1 MMC の内部レジスタに対するメモリマップ . . . 219
B.2 MMC の内部レジスタの役割 . . . 219
B.2.1 RECV_ADR_REG(0-7) . . . 219
B.2.2 RECV_BUF_STATE . . . 219
B.2.3 SEND_BUF_STATE . . . 221
B.2.4 MTC_REQ_STATE . . . 221
B.2.5 CBC_INT_PKT_SRC+ . . . 221
B.2.6 CBC_PEND_DOWN . . . 222
B.2.7 CBC_ADR_CMP_MODE . . . 222
B.2.8 CBC_IRR_REG . . . 223
B.2.9 CBC_PEND_L_ [BMP, MASK, ACT] . . . 223
B.2.10 CBC_PEND_COUNTER(0-3) . . . 225
B.2.11 MTC_PROC_NUM . . . 226
B.2.12 MTC_DECE_REG . . . 226

B.2.13 MTC_INT+ 226
B.2.14 REF_CMP 227
B.2.15 CBC_LOCK_REG_STATE 227
B.2.16 CBC_LOCK_［BMP, MASK, ACT］(0-3) 228

付 録 C MBP Core の命令 231

C.1 WGG 命令 231
C.2 WGI 命令 231
C.3 分岐命令 231
C.4 LMA(Local Memory Access) 命令 231
C.5 BPI 命令 234
C.6 WPG 命令 235
C.7 MPP 命令 235
C.8 RDT 送受信命令 236
C.9 MMC 送受信命令 236
C.10 Table Jump 命令 (TJ) 238
C.11 割込み命令 (INT) 238
C.12 特殊命令 (SPE) 239
C.13 IMA(内部メモリ access) 命令 239

付 録 D クラスタバスの仕様 241

D.1 パケットの構成 241
D.2 制御線 247

付 録 E memory command 251

付 録 F SPARC の boot 257

F.1 SPARC の reset 後の動作について 257
F.2 SPARC boot program 257
F.2.1 reset 処理及び processor の初期化の流れ 257
F.2.2 MMU の初期化の流れ 261
F.2.3 命令キャッシュの初期化 263
F.2.4 データキャッシュの初期化 263
F.2.5 2 次キャッシュの初期化 264
F.2.6 スタックの初期化 266
F.3 まとめ 267

付 録 G MBIF のコマンド 269

付 録 H MBIF を用いた File System 273

H.1 File System 用 protocol 273
H.1.1 `open` 273
H.1.2 `read` 274

H.1.3 write 274
H.1.4 close 274
H.1.5 fflush 274
H.1.6 maintenance 274
H.2 標準関数の使用法 275
H.2.1 OPEN 275
H.2.2 READ2B 276
H.2.3 WRITE2B 276
H.2.4 WRITE32B 276
H.2.5 CLOSE 276
H.2.6 FFLUSH 277
H.2.7 MAINTENANCE 277

付 録 I メインテナンスシステムでの使用チップ 279

付 録 J 汎用並列計算機 monitor system –Pot–の使用法 281

付 録 K MBP-light 用 C コンパイラ 285
K.1 GCC の特徴 285
K.2 GCC と MBP-light との不親和性 285
K.3 C コンパイラの実装 286

図 目 次

2.1 CPPACS のスライドウィンドウ機構 9
2.2 ハイパークロスバ結合網 10
2.3 RWC-1 の構成 11
2.4 RWC-1 のプロセッサアーキテクチャ 12
2.5 DASH の構成 15
2.6 NUMA-Q の構成 16
2.7 OBIC と SCLIC の構成 16
2.8 NUMAchine の構成 17
2.9 NUMAchine のノード構成 17
2.10 FLASH の構成 18
2.11 MAGIC の構成 18
2.12 Mercury の構成 19
2.13 Mercury のノード構成 20
2.14 Origin2000 の構成 20
2.15 Hub chip の構成 21
2.16 NUMAflex の構成 22

3.1 16 クラスタ 64 プロセッサの JUMP-1 27
3.2 JUMP-1 の構成 27
3.3 JUMP-1 クラスタの構成 28
3.4 JUMP-1 クラスタボードの写真 28
3.5 JUMP-1 プロセッサボードの写真 29

4.1 ACS-CC の典型的な実装方法 34
4.2 TIS-CC の典型的な実装方法 34
4.3 主記憶における逐次化 36
4.4 memory command とアドレスの関係 37
4.5 Elastic Barrier の全体構成 39
4.6 ダミーの同期要求に伴うオーバヘッドの削減法 40
4.7 生産者・消費者の依存に伴うオーバヘッドの削減法 41
4.8 クラスタ内におけるコントローラの構成 42

5.1 JUMP-1 におけるクラスタメモリ管理の一例 46
5.2 JUMP-1 におけるアドレス変換 46
5.3 高機能な分散共有記憶による同期機構の概念図 (1) 52

5.4 高機能な分散共有記憶による同期機構の概念図 (2) 52
5.5 MBP-light の全体構成 54
5.6 RDT Interface の構成 55
5.7 RDT 送受信用バッファの構成 56
5.8 RDT と MBP-light 間のマルチキャストパケットの転送の様子 57
5.9 RDT と MBP-light 間の応答パケットの転送の様子 57
5.10 送信用バッファから RDT router へのパケット送信 58
5.11 RDT router から受信用バッファへのパケット受信 59
5.12 Ack Cache の構成 60
5.13 Compressed Address の構成 61
5.14 Ack Emergency Buffer の構成と形式 64
5.15 Net Cache の構成 65
5.16 Net Cache とクラスタアドレスの関係 66
5.17 Ackmap Cache の構成 67
5.18 Net Cache による応答パケットの body の構成 67
5.19 shoot down 時の Net Cache の状態 68
5.20 RDT Interface command/status register の構成 69
5.21 MMC の内部構成 70
5.22 JUMP-1 のメモリ構成 71
5.23 クラスタメモリのフォーマット 71
5.24 SBUSCTL でのパケット受信の様子 73
5.25 SBUSCTL でのパケット処理の様子 74
5.26 クラスタメモリのアドレスフォーマット 78
5.27 MBP Core の構成 81
5.28 GPR による PBR の参照方法 82
5.29 GPR と PBR 間の加算命令 82
5.30 MMC や RDT Interface からの PBR の参照方式 83
5.31 内部メモリのアドレスマップ 85
5.32 代表的な命令形式 93
5.33 MBP-light のレイアウト 102
5.34 MBP-light の写真 103
5.35 各命令の比率: コンパイラ出力 104
5.36 各命令の比率: クラスタ 0(ホームクラスタ) 104
5.37 各命令の比率: クラスタ 1 104

6.1 階層 bitmap directory 方式 114
6.2 縮約階層 bitmap directory 方式 116
6.3 完全 RDT の構成 117
6.4 RDT(2,4,1)/α の構成 118
6.5 RDT(2,4,1)α 上での e-cube routing 119
6.6 基本転送操作 120
6.7 階層 bitmap directory 方式で用いる 8 進木とビットマップとの対応 120

6.8 マルチキャストの範囲 121
6.9 RDT router の基本構成 123
6.10 入力バッファの構成 126
6.11 RDT router chip の写真 130
6.12 RDT backplane board の写真 130
6.13 16 クラスタにおける RDT 結線図 131
6.14 単一リンクにおける RDT ネットワークのバンド幅 132
6.15 送信処理に必要なサイクル数 133
6.16 ネットワークの飽和の比較 134

7.1 メインテナンスシステムの構成 140
7.2 MBIF から MC への転送 142
7.3 MC から MBIF への転送 142
7.4 MBIF-MC の接続 144
7.5 MBIF と外部の接続 145
7.6 MBIF-MC 間接続用コネクタのピン配置 145
7.7 MC の全体構成 147
7.8 Register&Buffer 部の構成 148
7.9 I/O アドレス可変機構 149
7.10 ワン・ホット方式のステートマシン 151
7.11 デコード方式のステートマシン 152
7.12 MBIF の内部構成 153
7.13 Sender 部ステートマシンの状態遷移図 153
7.14 Receiver 部ステートマシンの状態遷移図 156
7.15 Local Bus Interface 部ステートマシンの状態遷移図 157
7.16 デコーダの構成 159
7.17 Controller 部ステートマシンの状態遷移図 161
7.18 リセット解除コマンド実行時の遅延の測定 165
7.19 最大転送容量の測定 166

8.1 STAFF-Link の構成 168
8.2 JUMP-1 本体と入出力サブシステム 169
8.3 評価システム 171
8.4 実効転送速度の変化 173

9.1 クラスタ間でのパケットの流れ 177
9.2 ペンディング・トランザクション・テーブル (PTT) 179
9.3 更新要求のすれちがい 180
9.4 台数効果 185

A.1 multicast packet の形式 207
A.2 acknowledge packet の形式 210
A.3 shootdown/setup Packet の形式 213

A.4 Ack Cache shootdown packet の形式 214
A.5 timeout packet の形式 215
A.6 command packet の形式 215
A.7 status report packet の形式 217

C.1 MBP Core から見た MMC の address 体系 238

D.1 address flit の構成 241
D.2 data flit の構成 241
D.3 長さ 2 のパケットによる送受信の様子 248
D.4 制御線のタイミング 250

I.1 74LS273(Octal D-Type Flip-Flops with Clear) 279
I.2 74LS540(Octal Buffers And Line Drivers with 3-State Output) 279
I.3 74LS642(Octal Bus Transceivers) 280
I.4 74LS688(8-Bit Magnitude Comparator) 280

K.1 関数呼出しの RTL 287
K.2 特別な関数呼出しの RTL 288

表 目 次

2.1 超並列計算機 . . . 7
2.2 CC-NUMA システムのまとめ . . . 23

4.1 1次 cache の諸元 . . . 31
4.2 TLB の諸元 . . . 32
4.3 ASI の割り当て . . . 33
4.4 2次キャッシュの実装諸元 . . . 37

5.1 SBUSCTL の割込み条件一覧 . . . 75
5.2 MTC の割込み条件一覧 . . . 77
5.3 ECC Parity 行列 . . . 80
5.4 Jump Table . . . 88
5.5 割込みレベルとクリア方法 . . . 89
5.6 Timer/Monitor の内部 memory map . . . 90
5.7 Monitor 0 の命令と状態 . . . 91
5.8 Monitor 1 の命令と項目 . . . 91
5.9 Monitor 2 の命令と項目 . . . 92
5.10 Monitor 3 の命令と項目 . . . 92
5.11 Timer の命令 . . . 93
5.12 命令の分類 . . . 94
5.13 WGG 命令 . . . 94
5.14 WGI 命令 . . . 95
5.15 Branch 命令 . . . 95
5.16 ローカルメモリアクセス命令 . . . 95
5.17 WPI 命令 . . . 96
5.18 WPG 命令 . . . 96
5.19 MPP 命令 . . . 97
5.20 RDT に関する命令 . . . 97
5.21 MMC に関する命令 . . . 98
5.22 Table Jump 命令 . . . 98
5.23 割込みに関する命令 . . . 99
5.24 内部メモリアクセス命令 . . . 100
5.25 特殊命令 . . . 100
5.26 I/O device の address map . . . 101

5.27 MBP-light の実装諸元 102
5.28 MBP-light のゲート数 103
5.29 ローカルメモリアクセスの比較 105
5.30 Delay Slot 105
5.31 branch 命令: コンパイラ出力 106
5.32 branch 命令: クラスタ 0(ホームクラスタ) 106
5.33 branch 命令: クラスタ 1 106
5.34 演算種類: コンパイラ出力 107
5.35 演算種類: クラスタ 0(ホームクラスタ) 107
5.36 演算種類: クラスタ 1 107
5.37 PBR Manipulating Instructions: Included in the Compiled Code 108
5.38 PBR Manipulating Instructions: Executed in the Cluster 0 (Home Cluster)108
5.39 PBR Manipulating Instructions: Executed in the Cluster 1 108
5.40 Replacement of PBR Manipulating Instructions with Other Instructions . 109
5.41 Compare Required Cycles: PBR Manipulating Instructions 109
5.42 特殊命令: コンパイラ出力 110
5.43 特殊命令: クラスタ 0(ホームクラスタ) 110
5.44 特殊命令: クラスタ 1 110
5.45 Replacement of Special Instructions with Other Instructions 111
5.46 Compare Required Cycles: Special Instructions 111

6.1 RDT router における hand shake 線 124
6.2 RDT router のゲート数 129
6.3 RDT router の実装諸元 129
6.4 応答パケット返送に要するサイクル数 135
6.5 ソフトウェア処理による処理内容の内訳 135
6.6 応答パケット自動収集に要するサイクル数 (RDT ルータ) 136
6.7 ソフトウェア処理による処理内容の内訳 (Core 起動時間のみ) 136
6.8 応答パケット自動収集に要するサイクル数 (MBP-light) 137
6.9 ソフトウェア処理による処理内容の内訳 (Core 起動時間のみ) 137

7.1 MBIF-MC 間の信号 144
7.2 各信号のコネクタへの割当て 145
7.3 MBIF の信号線 (ローカルバス側) 146
7.4 MBIF の I/O アドレス 146
7.5 pASIC 1 Family 150
7.6 Sender 部の入出力信号 154
7.7 Receiver 部の入出力信号 156
7.8 Local Bus Interface 部の入出力信号 158
7.9 MBIF のレジスタ 162
7.10 MSR の構成 164
7.11 LSR の構成 164

8.1 実効転送速度の合計 172
8.2 STAFF-Link の転送速度 173

9.1 基本的なトランザクションのレイテンシ 182
9.2 読出し処理のレイテンシの内訳 184
9.3 行列積プログラムの評価 184

B.1 MMC の内部レジスタ 220
B.2 PEND lock register の構成 223
B.3 アドレスの比較範囲 224
B.4 PEND lock register での action 225
B.5 lock register の構成 228
B.6 lock register での action 229

C.1 MBP Core の命令 232
C.2 WGG 命令の動作 233
C.3 WGI 命令の動作 233
C.4 分岐命令の動作 233
C.5 LMA 命令の動作 234
C.6 BPI 命令の動作 234
C.7 WPG 命令の動作 235
C.8 MPP 命令の動作 235
C.9 RDT 送受信命令 236
C.10 MMC 送受信命令 236
C.11 Table Jump 命令 238
C.12 割込み命令 239
C.13 特殊命令 240
C.14 IMA 命令 240

D.1 `size` field とデータ幅の対応 242
D.2 `did` field とデバイスの対応 242
D.3 パケット一覧 244

E.1 memory command 一覧 252
E.2 coherence-policy 253

G.1 MBIF のコマンド (MBP 側) 269
G.2 MBIF のコマンド (MC 側，初期化モード) 270
G.3 MBIF のコマンド (MC 側，メインテナンスモード) 271

第1章　緒 論

1.1　重点領域研究の目的とJUMP-1の位置づけ

超並列計算機JUMP-1は1992年より開始された文部省重点領域研究「超並列原理に基づく情報処理基本体系」の一環として京都大学，東京大学，慶應義塾大学，神戸大学，東京工科大学，九州工業大学，岡山理科大学によって共同開発された超並列計算機プロトタイプである．

超並列原理に基づく情報処理体系は，超並列処理を今後の科学技術を支える重要な基幹技術と位置づけ，その基本技術の確立を目指して，応用から言語，OS，ハードウェアアーキテクチャに至る総合的なプロトタイプを作ることを目的としてスタートした．当時は，1980年代に学会，産業界を巻き込んで広く行われたデータフローマシン研究開発，第5世代計算機プロジェクトが終了し，RISCを中心とするマイクロプロセッサがその急速な性能向上に弾みをつけつつあった．一方で，Thinking Machine社のConnection Machine[1][2]が商品化されると共に，Stanford大学のDASH[3], MITのAlewife[4]プロジェクトが後のCC-NUMA型大規模マルチプロセッサの先駆けとしてその成果を挙げつつあった．また，国内のスーパコンピュータメーカも富士通のAP1000[5]，NECのCenju[6]など従来のベクトル型から大規模並列型に，その方向を移しつつあった．

このような状況の下，データフローマシンやLogic Programmingに専用化されたプロセッサを用いるよりも，汎用プロセッサを多数用いた超並列型計算機を開発すべきであるという機運が高まり，新情報処理開発機構の初期5年間プロジェクトによる超並列計算機RWC-1[7]，科学技術計算に特化したCP-PACS[8]，東芝のTS/1[9]など，「超並列」をキーワードとしたプロジェクトが次々にスタートした．世はまさにバブル崩壊寸前，「超並列」というキーワードは時代の雰囲気を象徴するものであった．

文部省重点領域研究は，このような時代の影響を受けつつも，ハイパフォーマンスの追求よりも，並列処理の基盤技術の確立を重要視した点で，他のプロジェクトとやや異なった性格を持っていた．プロジェクトリーダの田中英彦教授のプロジェクト概要を示す文[10]を一部省略しつつ以下に引用する．

「1980年代の一連の研究を通して並列処理の基礎技術はかなり明らかになってきたが，現在の高度なプロセッサ技術，LSI技術，コンパイラ技術をベースにこれらの基礎を踏まえて新たに将来の並列処理技術体系を作り上げることが必要である．現在，いくつかの高性能な並列コンピュータが商用化されているが，その利用は限定されており，まだまだ本来の能力を発揮しているとは言いがたく，並列処理は初期的段階にあるのが現状である．すなわち，並列処理の利用を広く進めるためには様々な技術の確立と積み重ねが必要で，その基幹技術をきちんと作り上げ，まとめあげて将来の方向を示すことが現在求められていることであろう．我々の重点領域研究は，そのような基本技術体系の確立を目指して設

立されたものである.」

重点領域研究では，応用，言語，OS，ハードウェアの4つの班に分かれて，汎用的な超並列処理を実現することを目的としていた．研究は，以下の体制で実行された．このハードウェア班の研究の一環として超並列計算機 JUMP-1 の設計及び実装が行われた．

- A 班: 超並列処理モデルの研究
  多くのアプリケーションの実装を通して，超並列応用のモデル化の方法を作る．また，ベンチマーク方式を確立する．早稲田大学の村岡洋一教授が班長となり，7名の研究者により実行.
- B 班: 超並列記述系，処理系の研究
  超並列処理に適したプログラミング記述言語とコンパイラの開発．九州大学の雨宮真人教授が班長となり，10名の研究者 を含む.
- C 班: 超並列処理体系，制御系
  超並列処理に向いたオペレーティングシステムの開発．慶應義塾大学の斉藤信男教授が班長となり，8名の研究者を含む.
- D 班: 超並列ハードウェア，アーキテクチャの研究
  プロトタイプ JUMP-1 の開発．京都大学の富田眞治教授を班長とする11名の研究者.

上記の顔ぶれを見てもこのプロジェクトが国内大学の計算機関係の研究者の総力を挙げたものであったことがわかる．しかし，予算は3年間で6億円前後であり，RWC-1, CP-PACS などに比べると遥かに少なく，しかも参加人数が多いことから，プロトタイプ JUMP-1 に投入される予算は当初から極めて限定されていた.

1.2 JUMP-1 の基本構想

JUMP-1 は，CP-PACS, RWC-1 など同時期の超並列計算機が主として性能を追求しているのに対して，スケーラブルな性能向上と広い応用分野を想定したコスト性能比を追求していた．D 班では，当初からこのような目的に適合した形式を CC-NUMA(Cache Coherent Non-Uniform Memory Architecture) であると考えた．当時，CC-NUMA の研究は Stanford 大の DASH, MIT の Alewife を中心にかなり進みつつあり，16-64 ノード程度の規模での現実性が明らかになってきた．しかし，このような方式を数千，数万という規模のノード数でコスト性能比良く実現することは困難であり，この点の解決と技術の確立を目標とした.

具体的には以下のとおりである.

- コストを重視し，汎用 RISC を実働プロセッサとして用いる.
- 大規模で効率の良いキャッシュ機構の開発．1次キャッシュから主記憶を利用した3次キャッシュまで，効率の良い分散メモリ構造の実現手法を確立するこのため，アップデイトベースの高いインテリジェンスをもつ1次キャッシュを用いる．このために，分散共有メモリ制御用プロセッサ MBP(Memory Based Processor) を用いて，通信

およびメモリ制御を実現する．また，キャッシュ制御には疑似フルマップ (RHBD 方式) を用いる．

- 超並列計算機上に分散共有メモリを実現する場合に適合する結合網の実現．このために階層構造をもつ Torus である RDT(Recursive Diagonal Torus) を提案し，メッセージマルチキャストと応答パケット収集機構をもつルータを開発する．
- 低コストのシリアルリンクを用いた分散並列 I/O とグラフィックインタフェースの開発．
- 超並列システム全体を制御しデバッグするモニタ/監視システムの開発．

1.3　JUMP-1 開発の経緯

D 班は，1992 年度から頻繁にミーティングを行った結果，以下のように担当が決まった．なお，所属，職位は当時のものである．

- 1 次キャッシュを中心とした CPU 周辺を京都大学 (富田 眞治教授，中島 浩助教授，森 真一郎助手，五島 正裕助手)
- MBP を東京大学 (平木 敬助教授，松本 尚助手)
- ネットワーク周辺を慶應義塾大学 (天野 英晴助教授)，東京工科大 (工藤 知宏助教授)
- ネットワークシミュレータを九州工業大学 (末吉 敏則助教授)
- I/O を神戸大学 (金田 悠紀夫教授，中條 拓伯助手)
- グラフィックスボードを岡山理科大 (小畑 正貴助教授)
- メインテナンスシステムを九州工業大学 (久我 守弘講師)，慶應義塾大学

このうち，I/O はシリアルインタフェース STAFF-link を用いることを早期に決定して 1993 年より開発を開始し，グラフィックスボードもこれに接続する形で開発を開始した．また，慶應大は，1993 年に接続網 RDT を提案し，翌年からルータチップの開発を開始した．ネットワークシミュレータは独立に開発が可能で 1994 年には INSIGHT が開発され，RDT の性能評価に利用された．メインテナンス系は中心部の設計後にペンディングされた．

一方，クラスタ中心部のアーキテクチャは主として東大の平木助教授，松本助手，京大の中島助教授によってグランドデザインが検討された．当初予定では，MBP を東大，キャッシュコントローラおよび MBP とコントローラを接続するバスチップを京大が開発することになっていた．しかし，実装の詳細に関して意見の一致を見ず，またチップ開発環境の劣悪さも加わって，設計は大幅に遅れることとなった．

3 年間が経過し，1995 年 3 月に重点領域研究の最終報告が行われたが，この際に D 班が展示できたのは，STAFF-link とグラフィックスのデモだけであった．しかし，当初から，3 年での実装は不可能であることが予想されており，1995 年以降は，文部省科研費，

それ以降は並列分散コンソーシアムによって，JUMP-1 の実装は続けられた．実装は以下のように進んだ．このプロジェクトは毎年 3 月にデモンストレーションを義務づけられており，これをポイントに実装の進み具合いを整理すると分かりやすい．

- 1996 年 3 月：RDT ルータチップ完成，バックプレーン展示．
- 1997 年 3 月: RDT ルータチップ稼働を開始し，これを用いて SUN WS を接続した WS クラスタを JUMP-1/3 を展示．1997年 6 月，東大は構想に対してゲート数が不足することから MBP 開発を断念，慶應大が主となり構想を大幅に縮小した MBP-light の開発に着手．
- 1998 年 3 月：1 ノードのクラスタボードを展示．この年，慶應大は MBP-light，京大はキャッシュ制御チップ，バスチップを相次いでデータインすることに成功．
- 1999 年 3 月：16 ノードのプロトタイプ稼働，京大にて分散メモリ管理用プログラム稼働．
- 2000 年 3 月: 64 ノードのプロトタイプ稼働，以後，プロジェクトはシステムの評価，ソフトウェアシステムの開発に移った．

筆者である鈴木は，1998 年からプロジェクトに参加し，メインテナンス系の開発を九州工大から引き継いで行うと共に，16 ノードプロトタイプの開発，64 ノードプロトタイプの開発に主導的な役割を果たした．さらに，稼働以降は JUMP-1 の性能評価を分析を行い，プロジェクトの最終的なまとめを受け持った．

1.4　論文の位置づけ

JUMP-1 はその構想から稼働まで 8 年を要した．この 8 年間で高性能計算機システムをめぐる状況は大幅に変化した．輝かしい成功を収めた CP-PACS を除いた超並列プロジェクトは挫折し，汎用部品を用いコスト性能比に優れた PC/WS クラスタが高性能計算の主力として発展した．この結果，CC-NUMA 型の大規模並列計算機は滅亡は免れているものの，非常に厳しい競争にさらされている．地球シミュレータ [11] に代表される巨大スーパコンピュータは，国の威信を賭けた開発が依然として行われているが，現在の研究の中心は，少数のスーパコンピュータや巨大なクラスタを高速ネットワークで接続して，効率的に利用する環境であるグリッドコンピューティングに移っている．

JUMP-1 は稼働開始時点で時代遅れのマシンであり，既にちょっとした PC クラスタにも及ばない程度の性能しか実現することができなかった．また，開発，実装中に，技術的な問題点が様々な形で露呈し，予算不足，劣悪な環境，開発技術力不足などもあって必ずしも成功に終わったマシンということはできない．特に開発の足を引っ張った大学間の対立は，大学間連合プロジェクトの問題点の典型的な例となってしまった．

日本の大学の総力を挙げたこのプロジェクトを，現時点で見て笑うことは簡単である．しかし，この開発中に明らかになった知見は多く，これらを何らかの形でまとめることこそプロジェクトの成果と言える．

Hennessy と Patterson は，著名なテキストである Computer Architecture -A Quantitative Approach-でこのように言っている．

「There is value in projects that do not affect the computer industry because of lesson that they document for future efforts. The sin is not in having a novel architecture that is not a commercial success; the sin is in not quantitatively evaluating the strengths and weaknesses of the novel ideas.」

また，JUMP-1 はいくつかの問題点からその稼働状況が制限されており，そのアーキテクチャ全体にわたる評価と検証は困難ではあるが，上記に従って JUMP-1 に対してできる限りの定量的検証を試み，今後の並列計算システムの発展に少しでも貢献できることを望む．

1.5　JUMP-1 の概要と本論文の構成

本論文では，2000 年 3 月の 16 クラスタ 64 プロセッサの JUMP-1 ハードウェアシステム完成以降行った，様々な評価結果についてまとめる．

筆者は，メンテナンスシステムと呼ばれる，JUMP-1 のブートアップや，ホスト PC との通信を行うためのシステムの設計と実装，RDT ネットワークのマルチキャスト機構，応答パケットの自動生成 (無駄パケットの自動廃棄) 機構，応答パケットの自動収集機構の基礎評価，及び分散共有メモリ管理プロセッサである MBP-light の Core プロセッサの命令セットアーキテクチャの評価を行った．

これまでに述べたように，JUMP-1 は一般的に用いられている共有メモリ型マルチプロセッサを数千プロセッサの規模にまで拡張することができるスケーラブルなアーキテクチャを目指しており，そのため RDT(Recursive Diagonal Torus) [12][13][14] ネットワーク，メモリ管理プロセッサ MBP-light 等，様々な特徴的な要素を用いて構成されている．

RDT ネットワークは，トーラス構造と階層構造を併せもつネットワークを採用している．RDT ネットワークは，DSM を効率良く管理するため，マルチキャスト機構や，応答パケットの自動収集，自動生成機構を備えている．RDT ネットワークの様々な機能は，RDT Router chip とメモリコントローラである MBP-light によって実現されている．MBP-light は内部に MBP Core と呼ばれる core processor を持ち，そして高速化を要求される部分に RDT Interface と呼ばれる hardwired-logic を設けている．MBP Core は buffer-register architecture と呼ばれる特殊なアーキテクチャを採用し，独自の命令セットを備えている．独自の命令セットにより，効率良くパケットを操作したり，分散共有メモリ管理に必要な処理を少ないサイクル数で処理したりする事ができる．また，RDT Interface 内には，Ack Generator と呼ばれる応答パケットの自動生成機構が実装され，マルチキャストに対する応答を高速に行うことができるようになっている．また，RDT Interface 内には応答パケットを自動収集するための Ack Collector も実装されている．この Ack Collector と，RDT Router chip 内の応答パケット自動収集機構によって，hardwired-logic による高速な応答パケット収集が行われる．したがって，MBP Core が応答パケットの収集のために起動する可能性が低減され，パフォーマンスが向上する．

本論文では，以後下記の内容を示す．

- 第 2 章 [関連研究]: その他の超並列計算機プロジェクトや，CC-NUMA 型並列計算

機について調査したものを述べる．

- 第3章 [超並列計算機 JUMP-1の全体設計]: JUMP-1の設計思想とシステムの全体構成について述べる．
- 第4章 [超並列計算機 JUMP-1の構成要素]: JUMP-1の構成要素のいくつかについて，それぞれの特徴について述べる．
- 第5章 [MBP-light]: JUMP-1の分散共有メモリコントローラ MBP-light の構成について述べ，内部の Core プロセッサの命令セットについての評価結果について述べる．
- 第6章 [結合網 RDT(Recursive Diagonal Torus)]: JUMP-1の相互結合網 RDT について，その構成と高速化メカニズムを紹介し，それらの評価結果について述べる．
- 第7章 [メインテナンスシステム]: JUMP-1の運転監視装置であるメインテナンスシステムについて詳述する．
- 第8章 [Disk 入出力サブシステム]: JUMP-1の I/O ユニットについて述べる．
- 第9章 [分散共有メモリ]: JUMP-1の分散共有メモリ機構について述べ，分散共有メモリ上で行列積を計算するプログラムを実行した場合の性能を示す．
- 第10章 [JUMP-1 プロジェクトのまとめ]: JUMP-1 プロジェクトで得られた知見について述べる．
- 第11章 [結論]: 本研究の成果について結論を述べる．

第2章　関連研究

本章では，関連する研究や技術に関する調査結果について述べる．まず，その他の超並列計算機について触れ，特に CP-PACS[8]，RWC-1[7] プロジェクトについて詳しく述べる．次に，現在までに研究開発が行われている代表的な CC-NUMA(Cache Coherent-Non Uniform Memory Access model) とその分散共有記憶制御機構の構成についての比較検討を行う．

2.1　超並列計算機プロジェクトに関する調査

表 2.1 超並列計算機

名称	研究開発機関	要素 PU	最大 PU 数	結合網	特徴
CM-5[2]	Thinking Machine Corp.	SPARC+FPU	1024	4 進木を基本とした FatTree	最大 128G FLOPS
AP1000[5]	富士通	SPARC+FPU	1024	2 次元トーラス+リング+階層バス	最大 8.5G FLOPS
AP1000+	富士通	SuperSPARC+	1024	2 次元トーラス+リング+階層バス	最大 51.2G FLOPS
Cenju-2	NEC	VR3000 + VR3010	256	多段接続網による直接接続	最大 6.4G FLOPS
Cenju-3[6]	NEC	R4400SC	256	多段接続網による直接接続	最大 12.8G FLOPS
SR8000[15]	日立	独自 RISC	1024	多次元クロスバ	最大 1024G FLOPS
CP-PACS	筑波大	PA-RISC 改	2048	3 次元ハイパークロスバ	最大 614G FLOPS
RWC-1	新情報処理開発機構	独自 RISC	1024	MDCE 網	

表 2.1 は，JUMP-1 プロジェクトが開始された当時の超並列計算機についてまとめたものである．その後，富士通の AP3000，NEC の Cenju-4(400G FLOPS)，日立の SR8000 モデル E1 (1228.8G FLOPS) など，後継機種が開発されていくことになる．

このように，超並列計算機には企業で開発が行われたものも多数あるが，ここでは大学や研究機関で開発が行われた，CP-PACS と RWC-1 について詳しく取り上げる．

2.1.1　CP-PACS プロジェクト

CP-PACS(Computational Physics by Parallel Array Computer System)[8] プロジェクトは，平成 4 年春に文部省の「学術の新しい展開のためのプログラム」の平成 4 年度の実施テーマの 1 つとして採択され，文部省科学研究費補助金 (創成的基礎研究費) により実施された超並列計算機プロジェクトである．CP-PACS 計画の推進母体として筑波大学内に計算物理学研究センターを設置し，研究を実施していた．

2.1.1.1　全体構成

CP-PACS は，2048 台の PU(Processing Unit) を備え，理論ピーク性能 614GFLOPS の MIMD(Multiple Instruction-streams Multiple Data-streams) 方式分散メモリ型並列計算機である．2048 台の PU と，入出力を分散処理する 128 台の IOU(Input/Output Unit) が，ハイパークロスバ (Hyper Crossbar) 結合網により 8×17×16 の 3 次元配列に結合されている．

2.1.1.2　ノードプロセッサ

CP-PACS の各 PU は，PA-RISC1.1 アーキテクチャの上位互換性を保ちつつ改良を加えた独自のスーパースカラ RISC プロセッサである．RISC プロセッサによる大規模科学技術計算で問題になる，データキャッシュの容量不足による演算性能低下を，PVP-SW(Pseudo Vector Processor based on Slide Window) と呼ばれる機能の導入により解決している．

PVP-SW では，論理レジスタウィンドウを連続的に切り替えるスライドウィンドウ機構 (図 2.1) により，多数の物理レジスタの利用を可能とする．また，擬似的にパイプライン化された主記憶に対して連続発行可能な Preload/Poststore 命令により，先行ウィンドウへのロード，後続ウィンドウからのストアを行うことができる．これらの PVP-SW 機能により，主記憶のアクセス遅延が隠蔽され，スーパースカラプロセッサでありながら，効率のよいベクトル処理が可能となっている．

2.1.1.3　ハイパークロスバ結合網

CP-PACS は，多数のクロスバスイッチを x，y，z 各方向に配列した 3 次元ハイパークロスバを結合網に用いている．PU と IOU は各方向のクロスバを繋ぐ Exchanger に接続され，最大 3 段のクロスバを経由することにより，任意のパターンの PU 間データ転送が可能となっている．データ転送はリモート DMA(Remote Direct Memory Access) 方式により行われる．この方式では，OS の介在を最小限に抑え，各 PU 上のユーザ・メモリ領域間で直接データの送受信を行うことにより，転送立ち上げレイテンシの削減と，高い転送スループット性能を実現している．

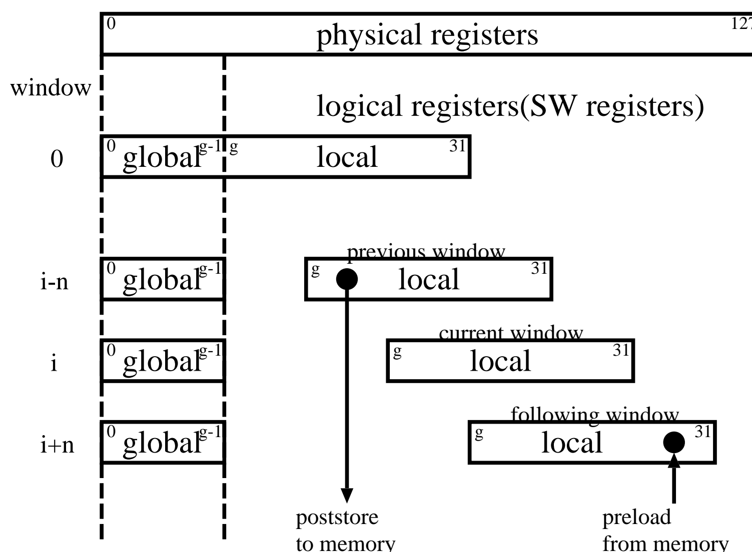


図 2.1 CPPACS のスライドウィンドウ機構

CP-PACS の 結合網は，x，y，z 各方向の中間面を境に，ハードウェアによる等分割が可能である．したがって，CP-PACS は最大 8 の部分システムによる分割運転を行うことができる．

2.1.1.4　オペレーティングシステム

CP-PACS は，OS に UNIX OSF/1 を採用している．各ノードプロセッサは，Mach 3.0 ベースのマイクロカーネルだけを搭載し，ユーザプログラムの使用可能メモリ容量と性能を確保している．カーネルは，メモリ制御，ノード間通信，プロセス制御，割込み処理，及び入出力機能を持つ．各 IOU の OS にはこのマイクロカーネルに加え，各種入出力処理用のサーバと UNIX フロントエンドを提供するサーバが加えられる．システムの全体的な制御機構は SIOU と呼ばれる IOU の 1 台に搭載され，ネットワークを通じて並列プロセスを制御する．

2.1.1.5　プログラミング環境

CP-PACS では，プログラム言語として FORTRAN90，C，C++，アセンブリを用いる事ができる．また，FORTARN 90 および C プログラムにアセンブリ記述されたサブルーチンを組み込むことも可能である．FORTRAN90 および C コンパイラは，modulo scheduling 技法と register coloring 技法を用いて，PVP-SW 機構に対応した実行モジュールを生成する．ハイパークロスバを通じてリモート DMA 転送を行う場合には，プログラム中で通信ライブラリを呼び出すことで実現可能である．

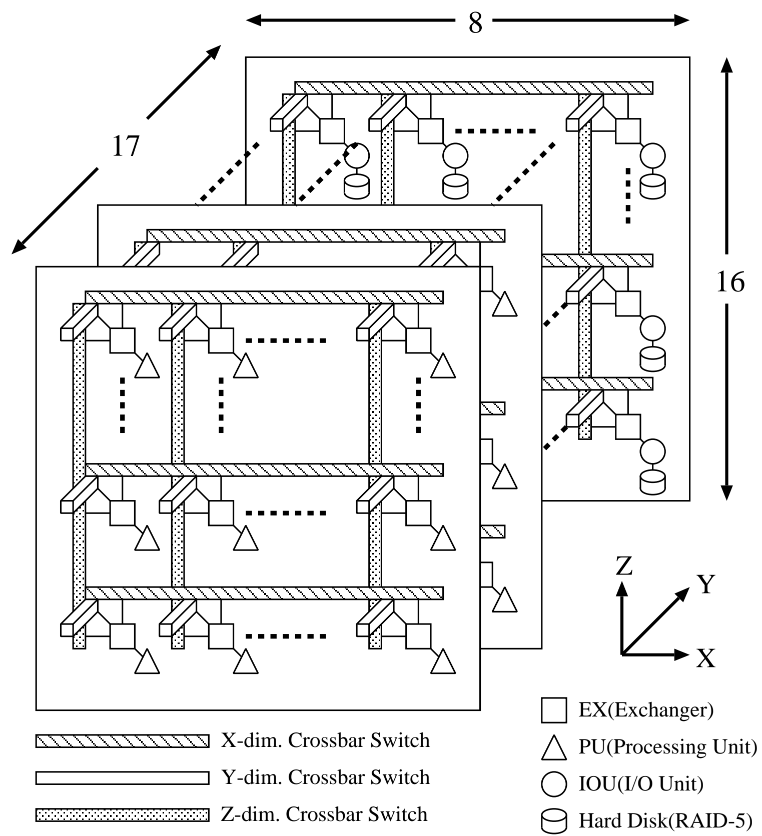


図 2.2 ハイパークロスバ結合網

2.1.2　RWC-1 プロジェクト

RWC-1[7] は，新情報処理開発機構において開発された超並列計算機である．超並列計算機 RWC-1 は，1024 個の要素プロセッサ (PE) とそれを繋ぐ相互結合網，入出力用相互結合網，大容量二次記憶系，画像・音声インタフェース，および外部とのインタフェースからなる．RWC-1 ではプロセッサアーキテクチャとして RICA (Reduced Inter-processor Communication Architecture) を採用し，通信の隠蔽を図っている．さらに，プロセッサ間結合網として MDCE(Multi-Dimensional Directed Cycles Ensemble) 網を採用することにより，高いスループットを維持しつつ遅延を削減している．

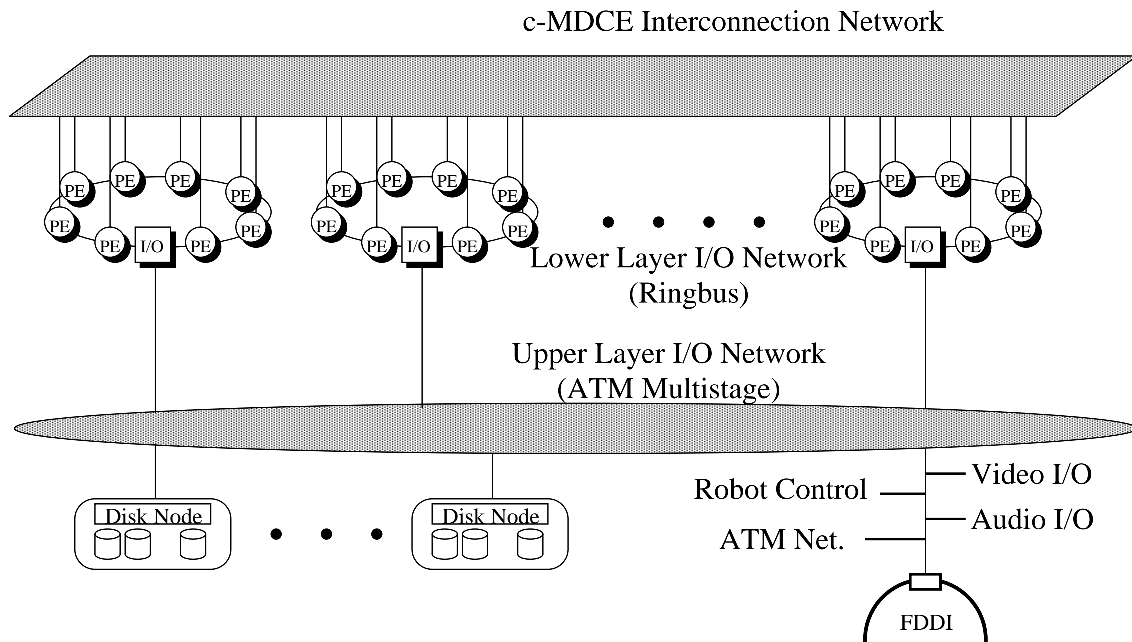


図 2.3 RWC-1 の構成

2.1.2.1　プロセッサアーキテクチャ

RWC-1 のプロセッサは，市販のプロセッサチップを用いず，独自のアーキテクチャをとる．通信と演算が高度に融合したアーキテクチャである RICA，超並列 OS の支援機能等の特徴がある．

プロセッサの構成を図 2.4 に示す．プロセッサは，BSB(Buffering and Scheduling Block)，EXB(EXecution Block)，POB(Packet Output Block)，MCB(Memory Control Block)，MAINT(MAINTenance Block) からなる．BSB，POB はそれぞれ，パケットのバッファリングと処理の起動，パケットの生成・出力をハードウェア的に行うユニットである．

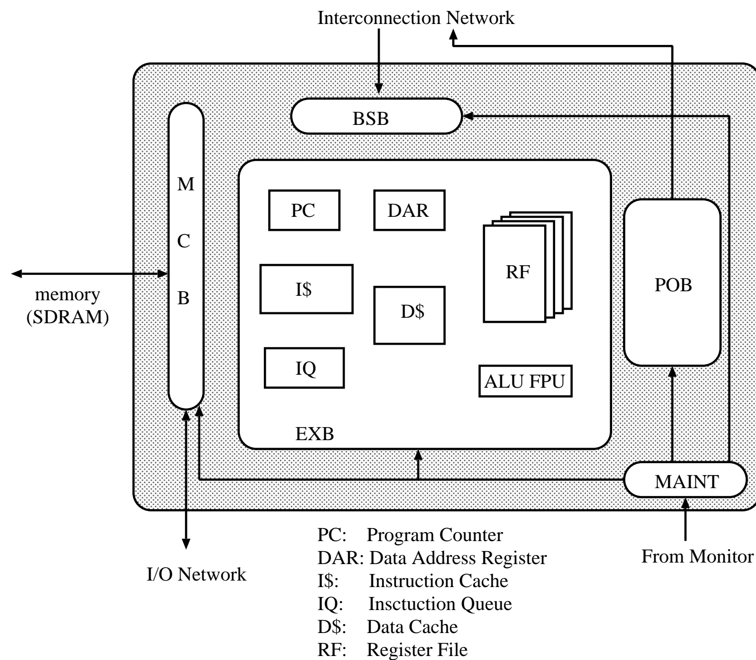


図 2.4 RWC-1 のプロセッサアーキテクチャ

2.1.2.2　相互結合網

RWC-1 の相互結合網は，MDCE(Multi-dimensional Directed Cycles Ensemble) 網を階層化して用いる．RWC-1 で必要とされる通信速度や通信機能を満たすため，ATM スイッチや Caltech chip などの市販の要素スイッチは用いず，オリジナルのチップを用いて構成している．RWC-1 の階層化 MDCE 網は，次の特徴を備える．

- 高スループット
- 低遅延
- セルフルーティングが容易
- 効率良い空間分割が容易

また，SFD(Store and Forward Deadlock) 防止のための仮想チャネル機構，優先度処理機構，時分割のためのドレイン機構などが実装されている．

2.1.2.3　入出力アーキテクチャ

図 2.3 に示すとおり，RWC-1 の入出力系は

- 2 階層の相互結合網
- 大容量 2 次記憶系
- 画像・音声インタフェース
- 外部とのインタフェース

などから成る．

入出力用の相互結合網が 2 階層であるのは，プロセス内部の局所性を利用するためと，システムの空間分割に効率的に対応するためである．下位階層の相互結合網は独自のリングバス，上位階層の相互結合網は ATM スイッチなどの市販のハードウェアを採用している．プロセッサは相互結合網やメモリとは独立した入出力インタフェースをもち，入出力と演算処理の間でメモリインタフェースを共有している．

2 次記憶系では，複数の RAID，これを制御するディスクノードを 1 モジュールとして，複数のモジュールを稼働させる構成となっている．

2.1.3　まとめ

それぞれの超並列計算機を比べてみる．プロセッサは，CP-PACS では PA-RISC を拡張したもの，RWC-1 は独自のプロセッサである．また，商用並列計算機の多くは汎用のプロセッサを用いているが，SR8000 では独自のプロセッサを採用している．また結合網は，CP-PACS では 3 次元ハイパークロスバ，RWC-1 では階層 MCDE 網を採用している．商用機では FatTree，多段接続網，クロスバ等，こちらもシステムに合わせて様々な構成方法が取られている．

このように，各々のプロジェクトは超並列計算という同一の視点に基づいているものの，採用しているシステムの構成方法は全く異なるものとなっている．

2.2　CC-NUMAに関する調査

CC-NUMA(Cache Coherent Non-Uniform Memory Access model)は大規模並列計算機を拡張性のある形で構成する一手法である．NUMA(Non-Uniform Memory Access model)とは，アクセスするアドレスによってデータが到着するまでのレイテンシや，バンド幅が異なるメモリシステムのモデルである．これに対して，レイテンシやバンド幅の差がないものをUMA(Uniform Memory Access model)と呼ぶ．CC-NUMAは，NUMAのモデルに基づくメモリシステムで，キャッシュの一貫性を保証するメカニズムを採り入れたものである．

CC-NUMAは，多数のプロセッサを接続可能であると共に，共有メモリを利用したプログラム開発も比較的容易であることから，いくつかの企業や大学により研究開発が盛んに行われてきた．本節では，以下の7つのシステムについて，分散共有メモリコントローラに焦点をあてて，調査及び比較検討を行う．

- Stanford大学のDASH [3]
- Sequent社のNUMA-Q [16]
- Tronto大学のNUMAchine [17]
- Stanford大学のFLASH [18]
- HAL研のMercury [19]
- Silicon Graphics社のOrigin2000 [20]
- Silicon Graphics社のNUMAflex(Origin3000) [21]

2.2.1　Stanford大学のDASH

Stanford大学で1992年に開発されたDASHの構成図を図2.5に示す．各ノードはSilicon Graphics社のPOWER station 4D/240で構成されており，4台のR3000プロセッサからなるバス結合型並列計算機となっている．DASHではhardwired-logicによって分散共有記憶を管理する．ディレクトリはキャッシュライン単位で管理され，各ノードにはディレクトリの管理を行うdirectory controllerが接続されている．また，結合網は2次元meshであり，デッドロックを避けるためにrequest用とreply用の2つのmeshが用意されている．DASHのキャッシュコヒーレンスプロトコルは無効化型であり，memory consistency modelとしてrelease consistency [22] を採用している．

2.2.2　Sequent社のNUMA-Q

Sequent社によって1997年に商用化されたNUMA-Qの構成を図2.6に示す．各ノードは4台のIntel Pentium Proプロセッサを P6 busで接続したバス結合型並列計算機となっている．NUMA-Qではhardwired-logicとprotocol processorの2つを使うことによって分散共有記憶を管理する．hardwired-logicはOBICと呼ばれ，分散共有記憶以外にリモー

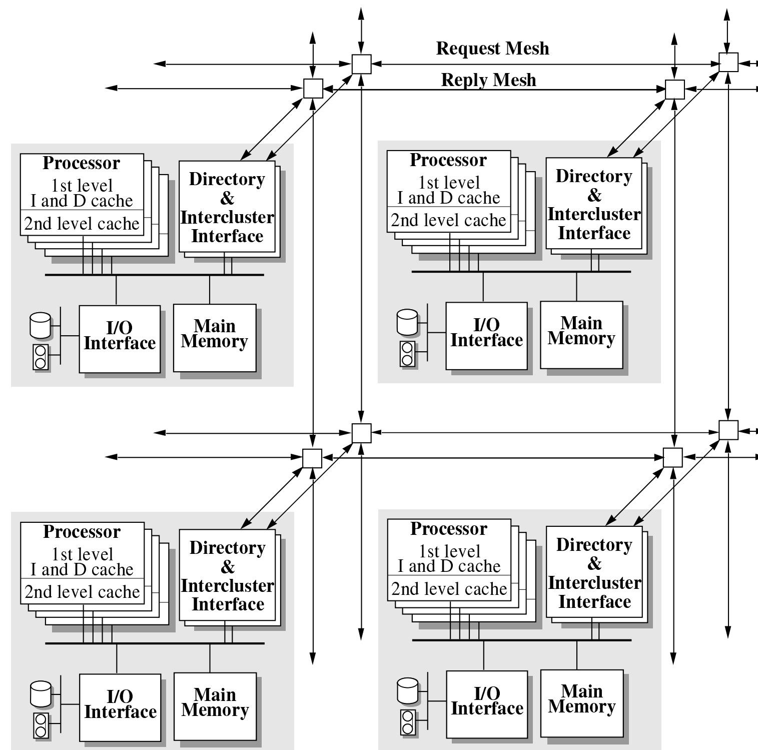


図 2.5 DASH の構成

トキャッシュも管理する．また，protocol processor は SCLIC と呼ばれ，OBIC との協調動作によって分散共有記憶を管理する．結合網としては汎用の SCI ring が用いられている．また，NUMA-Q のキャッシュコヒーレンスプロトコルは無効化型で，ディレクトリはキャッシュライン単位で管理する．

ここで，OBIC と SCLIC の構成を図 2.7 に示す．OBIC は P6 bus からのメモリ参照要求に対するキャッシュのヒット判定を行う．もしそのラインが invalid であれば，SCLIC にその要求を依頼する．一方，SCLIC は結合網や OBIC から受け取ったパケットをヘッダ部とデータ部に分割し，ヘッダ部だけ protocol engine で処理を行い，出力時にヘッダ部とデータ部とを組み合わせる．protocol engine はプロトコルに関連した処理を柔軟に行うためのプロセッサである．このプロセッサは task switch 機能をもっており，12 の独立なタスクをサポートすることができる．

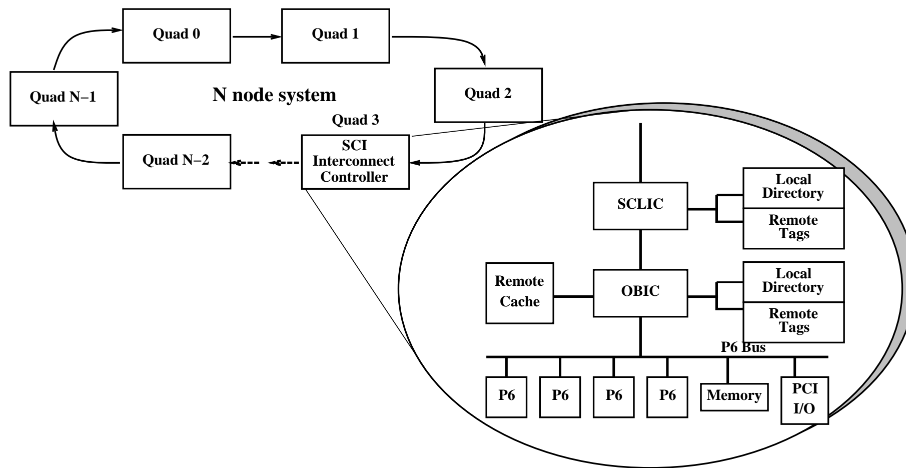


図 2.6 NUMA-Q の構成

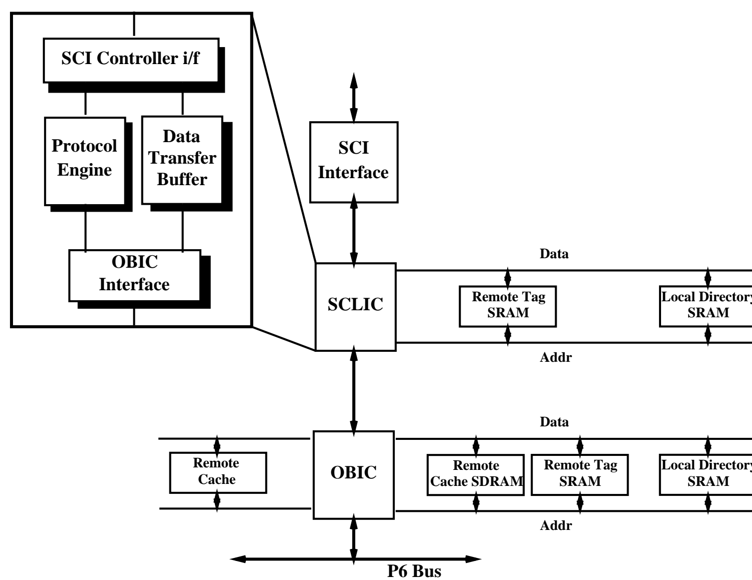


図 2.7 OBIC と SCLIC の構成

2.2.3　Tronto 大学の NUMAchine

Tronto 大学で研究された NUMAchine の構成を図 2.8 に示す．各ノードは 4 台の R4400 プロセッサからなるバス結合型並列計算機となっている．ノードの構成を図 2.9 に示す．NUMAchine では，Network Cache が明示的に存在し，分散共有記憶コントローラはメモリモジュールに内蔵されている．結合網は階層型 ring である．また，キャッシュコヒーレンスプロトコルは無効化型であり，ディレクトリはキャッシュライン単位だが縮約型で管理する．

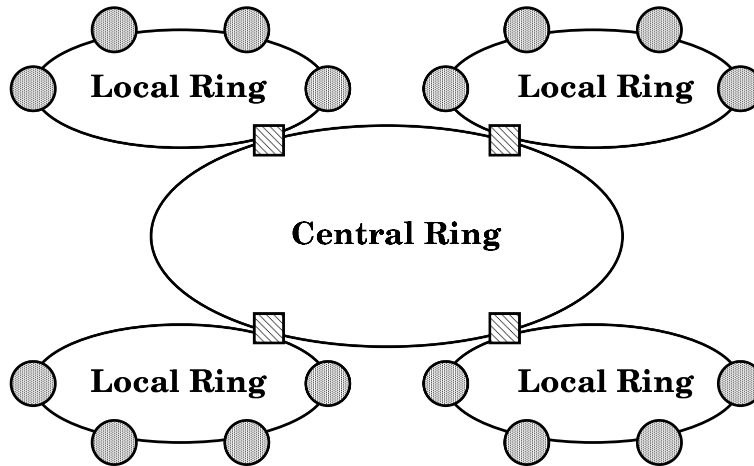


図 2.8 NUMAchine の構成

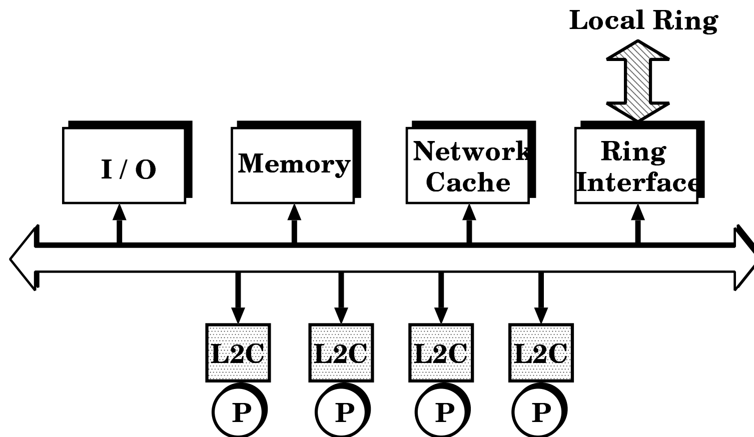


図 2.9 NUMAchine のノード構成

2.2.4　Stanford 大学の FLASH

DASH の後継機として Stanford 大学で開発が行われた FLASH の構成を図 2.10 に示す．各ノードは R10000 プロセッサからなるバス結合型並列計算機となっている．FLASH では MAGIC と呼ばれる protocol processor により分散共有記憶を管理する．結合網は DASH と同様に 2 次元 mesh であり，デッドロックを避けるために request 用と reply 用の 2 つの mesh が用意されている．また，キャッシュコヒーレンスプロトコルは無効化型であり，ディレクトリはキャッシュライン単位で管理する．

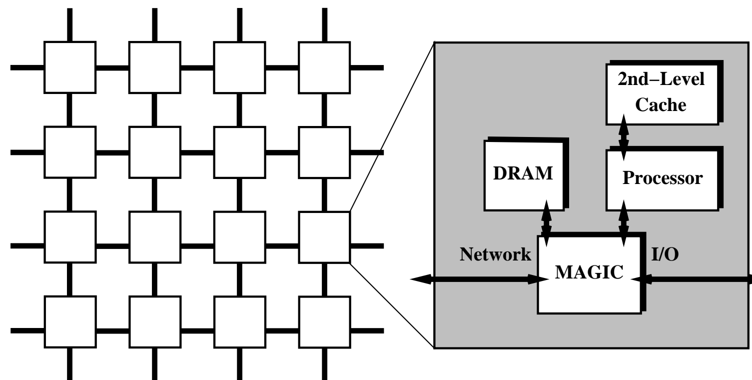


図 2.10 FLASH の構成

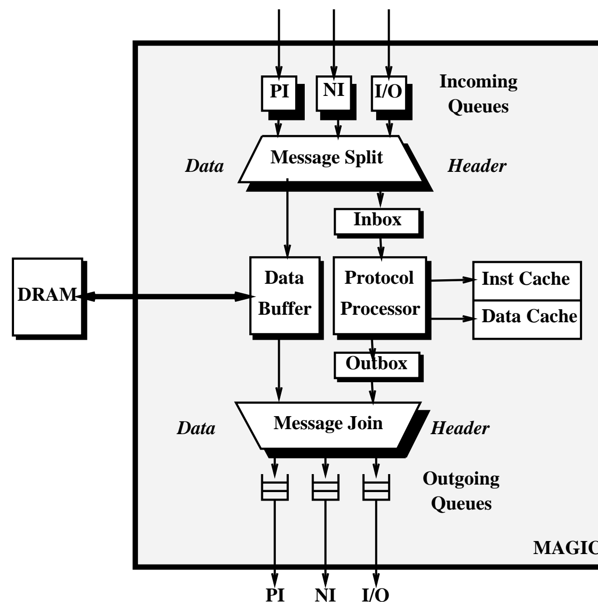


図 2.11 MAGIC の構成

ここで，MAGIC の構成を図 2.11 に示す．MAGIC では I/O，結合網，プロセッサの 3 種類から出力されるパケットを扱う．MAGIC 内ではこの 3 種類のパケットは同様に処理

される．まず，入力されたパケットはヘッダ部とデータ部に分離される．そして，ヘッダ部は protocol processor の入力 queue(Inbox) へ，データ部はデータバッファへそれぞれ渡される．ヘッダ部には，分散共有記憶の参照に伴うアドレス変換等の操作が施される．最後に protocol processor で処理されたヘッダ部とデータ部が出力時に合流し，外部へと転送されることになる．

この protocol processor は 64 bit の 2 命令同時発行スーパスカラプロセッサであり，100MHz で動作する．また，例外や割込み，pipeline clock の interlock，TLB などはハードウェアを複雑化してしまうためにサポートしていない．また，命令の競合のチェックを行わないため，全命令はコンパイラにより静的にスケジューリングされていることが必要である．命令セットは DLX[23] を基本としており，プロトコル処理に頻繁に用いられるビットフィールドの挿入/抽出や set/clear などの専用命令も備えている．

2.2.5　HAL 研の Mercury

HAL 研によって研究された Mercury(商用名は SynfinityNUMA) の構成を図 2.12 に示す．その結合網はトポロジフリーであり，通信プロトコルのみが規定されているに過ぎない．各ノードは 4 台の Pentium Pro プロセッサからなるバス結合型並列計算機である．ノードの構成を図 2.13 に示す．Mercury では MCU と呼ばれる hardwired-logic によって分散共有記憶を管理する．また，キャッシュコヒーレンスプロトコルは無効化型と推測され，ディレクトリはキャッシュライン単位だがスパースな形で管理される．

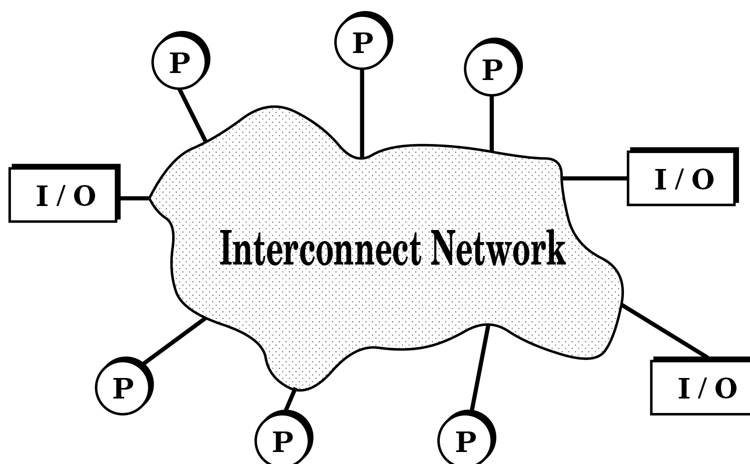


図 2.12 Mercury の構成

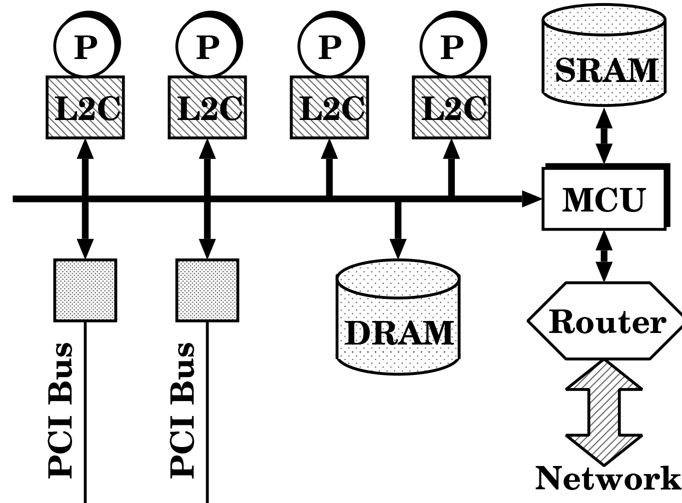


図 2.13 Mercury のノード構成

2.2.6　Silicon Graphics 社の Origin2000

Silicon Graphics 社によって 1997 年に商用化された Origin2000 の構成を図 2.14 に示す．各ノードは 2 台の R10000 プロセッサからなるバス結合型並列計算機である．しかし，一般のバス結合型のそれとは異なり，スヌープ方式でないのが特徴である．つまり，要求は直接分散共有記憶を管理する Hub chip が処理する．結合網としては，fat-bristled hypercube と呼ばれる hypercube の亜種を用いており，デッドロックを避けるために request 用と reply 用の両方が用意されている．また，キャッシュコヒーレンスプロトコルは DASH のプロトコルを基本とした無効化型であり，ディレクトリはキャッシュライン単位である．64 ノード以下のシステムでは，full bit vector でエントリを保持するが，64 ノード以上のシステムにおいては，共有ノードがホームノードの近傍の場合は full bit vector を，より遠いノードが含まれる場合は coarse bit vector で保持するというように，動的に選択して管理される．

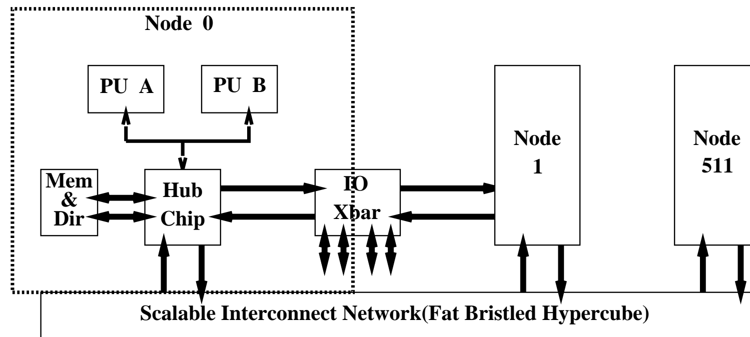


図 2.14 Origin2000 の構成

ここで，Hub chip の構成を図 2.15 に示す．Hub chip は NI(Network Interface)，II(I/O Interface)，PI(Processor Interface)，MD(Memory and Directory Interface) とクロスバから構成される．各インタフェース間の通信にはクロスバを用いる．II では I/O サブシステムとの通信や DMA のサポートを行う．NI では無効化時のパケットのマルチキャストのサポートやルーティングテーブルの制御を行う．また，MD ではディレクトリやメモリの参照を管理し，SDRAM の参照が最もバンド幅が高くなるように最適化している．Origin2000 は DASH 同様に release consistency model を採用しているため，メモリ参照間の同期のための fence 命令をサポートしている．これにはメモリ参照の履歴を表として管理する必要があり，protocol table や I/O protocol table がそれに相当する．

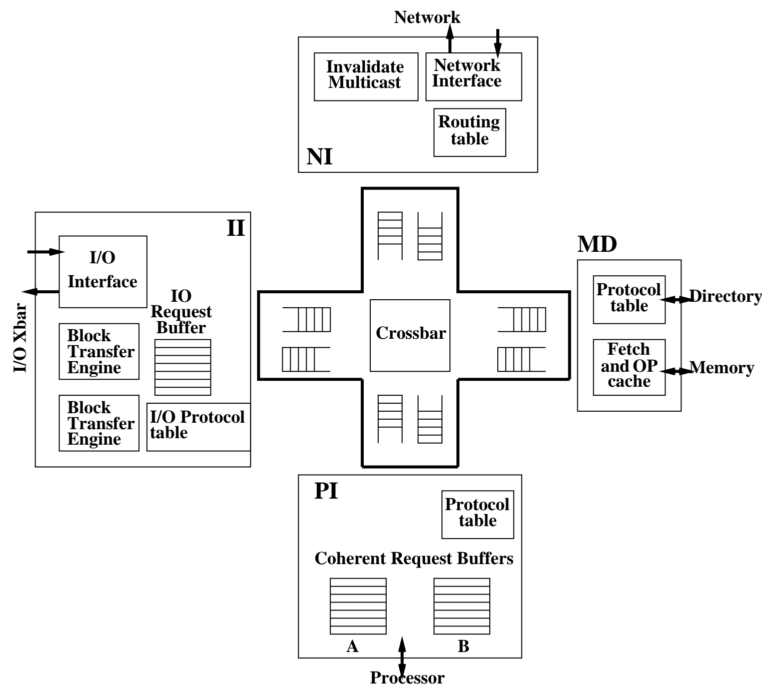


図 2.15 Hub chip の構成

2.2.7　Silicon Graphics 社の NUMAflex(Origin3000)

Silicon Graphics 社によって商用化された NUMAflex の構成を図 2.16 に示す．NUMAflex は，CPU に MIPS の R12000 プロセッサを採用した Origin3000，Onyx3000 と，Intel の IA-64 アーキテクチャによる Itanium プロセッサを採用したモデルがあるが，CPU 周辺を除いて構成は変わらない．NUMAflex では，1 ノードあたり 2〜4 プロセッサであり，128 プロセッサまでのネットワーク構造は hypercube である．それを超える場合，fat cube 構成をとることによって最大 512 プロセッサまで拡張可能である．

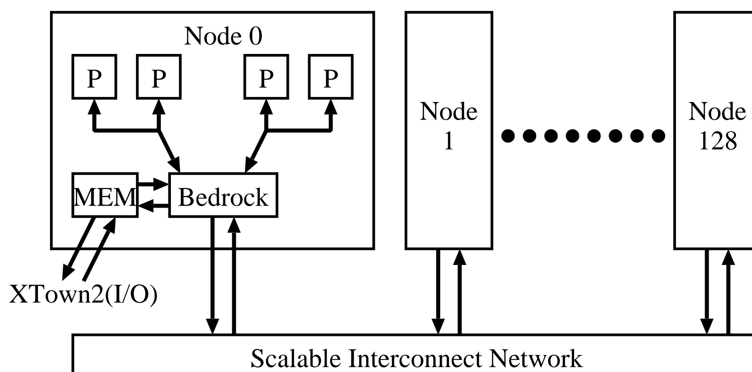


図 2.16 NUMAflex の構成

NUMAflex において，Origin 2000 の Hub Chip に当たるのが Bedrock と呼ばれる ASIC である．Bedrock は，2 チャネルの MIPS SYSAD bus をもち，バスのバンド幅は Origin 2000 の 2 倍となっている．また，メモリのバンド幅は 4 倍で，ネットワークのバンド幅は 2 倍である．また，1 ノード当たり 4 プロセッサ，1 ルータ当たり 8 リンクの構成をとることによって，プロセッサ数の増加によるホップ数の増加を最小限にとどめ，リモートメモリ参照のレイテンシを Origin 2000 の約 1/2 にすることに成功している．NUMAflex の分散共有記憶管理には Origin 2000 から大きな変更はないが，ハードウェアによる分散共有記憶のサポートがより一層強化されている．

2.2.8　まとめ

この節では，現在研究または商用化されている CC-NUMA に関してその構成と分散共有記憶コントローラに焦点を当てて調査を行った．それぞれのシステムを簡単にまとめたものを表 2.2 に示す．

まず，分散共有記憶コントローラは hardwired-logic による完全ハードウェア制御か protocol processor によるソフトウェア制御の 2 つに大きく分類される．ハードウェア制御による CC-NUMA の代表は Stanford 大学の DASH である．この節で列挙した他の CC-NUMA に先駆けて 1985 年に発表された並列計算機である．分散共有記憶管理にディレクトリ方式を導入し，memory consistency model として release consistency model を採用した．そして，その有効性を示した点において画期的といえる．しかし，ハードウェアによる制御ではプロトコルに柔軟性を欠き，キャッシュライン単位の無効化型プロトコルに制限され

表 2.2 CC-NUMA システムのまとめ

名称	研究機関	PU 数	結合網	コントローラ	特徴
DASH	Stanford 大学	4	mesh	hardwired	先駆的な研究
NUMA-Q	Sequent 社	4	ring	processor	バス側とネットワーク側の処理の分離
NUMAchine	Tronto 大学	4	ring	hardwired	明示的なリモートノード用のデータキャッシュ
FLASH	Stanford 大学	1	mesh	processor	強力な protocol processor によるサポート
Mercury	HAL 研	4	—	hardwired	トポロジフリーな結合網
Origin2000	SGI	2	hypercube	hardwired	強力なハードウェアによるサポート
NUMAflex	SGI	4	hypercube	hardwired	Origin 2000 のハードウェアを更に強化

てしまう．また，ユーザレベルの message passing において性能が劣化してしまうという欠点ももつ．この DASH を基本として種々のシステムが研究開発された．

Tronto 大学の NUMAchine では，結合網として階層型 ring を用いて，パケットのマルチキャストを効率良くサポートしている．また，リモートノードのデータ用のキャッシュである Network Cache を明示的に設けたのが画期的といえる．しかし，大学での研究のため性能はあまり高くない．

そして，HAL 研の Mercury では結合網をトポロジフリーにするという画期的な提案がなされている．しかも，強力なルータチップと分散共有記憶コントローラを設けることで，NUMA-Q 等に比べ大幅な性能向上が見られる．

さらに，商用機として開発されたのが Silicon Graphics 社の Origin2000 である．Origin2000 は Stanford 大学での DASH の研究者が主として開発したものである．各ノードはバス結合型の並列計算機であるが，スヌープ方式ではない．これはローカルノードへのメモリ参照時間とリモートノードへの参照時間の比を小さくしようとする努力の現われといえる．しかも，これを効率良くサポートするために，Mercury と同様に強力なルータチップと分散共有記憶コントローラを設けている．

また，Silicon Grafics 社は，Origin 2000 の後継機として新たに NUMAflex を商用化している．NUMAflex では，分散共有記憶管理には大きな変更はないが，Origin 2000 より更に強力なハードウェアによって，バンド幅の増加とレイテンシの短縮を実現している．

一方，ソフトウェア制御による CC-NUMA についてまとめる．まず，商用機として開発されたのが Sequent 社の NUMA-Q である．プロセッサとして Pentium-Pro を用いており，ソフトウェアの移植性を高める目的があると考えられる．しかし，P6 bus は他のメモリバスよりも複雑なアクセスサイクルを規定しているため，そのコントローラは複雑化しやすい．NUMA-Q もその例にもれず，バスコントローラである OBIC が複雑化し，SCLIC と OBIC を分割せざるを得なかったのであろう．しかし，メモリに対するネットワークとバスアクセスを機能別に分離して処理することになったため，性能に大きく影響することはない．さらに，NUMA-Q ではプロトコル管理に柔軟性をもたせるべく，SCLIC 中に protocol processor を採用している．バスアクセスの多くの処理は OBIC が行うため，protocol processor の処理能力が問われることはない．

さらに，FLASHではDASHにおいて欠点としてあげられたプロトコルの柔軟性とmessage passingの性能を改善するべく，MAGICと呼ばれるコントローラを採用した．MAGICではprotocol processorを高性能なものにし，I/Oや結合網，プロセッサからのすべての種類の処理をそれに委ねている．しかし，MAGICは一度に1つのパケットしか処理できないために，protocol processorの処理能力がMAGICの処理能力におけるボトルネックになってしまうと考えられる．

第3章　超並列計算機JUMP-1の全体設計

JUMP-1[24, 25, 26] は文部省重点領域研究「超並列原理に基づく情報処理基本体系」の一環として，1994 年より 7 大学 (京都大学，東京大学，慶應義塾大学，神戸大学，東京工科大学，九州工業大学，岡山理科大学) の共同で開発が行われた．

本章では，まず JUMP-1 の目標，設計方針を掲げ，その構成及び構成要素について述べる．

3.1　JUMP-1 の目標

JUMP-1 プロジェクトが開始された頃，超並列処理の飛躍的な性能向上を目指した研究開発が盛んに行われていた．特に，プロセッサを多数組み合わせる (Massively Parallel Processor) 方式は，次世代のスーパーコンピュータの実現方法の 1 つとして広く認知されていた．その第一世代の超並列計算機は分散記憶アーキテクチャを基礎としていた．その構成はデータ並列計算モデルに基づくアプリケーションでは高性能を発揮することができる．しかし，それ以外のアプリケーションに関しては性能が低下することが明らかになっている [27]．したがって，数値計算，特に科学技術計算に適用範囲が限定され，汎用計算目的として利用することは非常に困難である．当時，並列計算機アーキテクチャに関する研究は広く行われていたが，実際に実機を製作する段階に達するものは国内の大学の研究機関では数少なく，超並列計算機という観点では，他に文部省科研費による CP-PACS[8]，通産省の RWC-1 プロジェクト [7] の 3 つしか行われていない．このため，様々な特徴的な要素を備える JUMP-1 プロジェクトの位置付けは重要なものであった．

この JUMP-1 の目標は OS の介在するマルチユーザ，マルチタスク環境で，汎用的なアプリケーションを効率的に実行させる並列計算機環境を整えることにあった．

3.2　JUMP-1 の設計方針

目標の実現のため，基本アーキテクチャとしてはキャッシュコヒーレントな分散共有記憶を採用し，いわゆる CC-NUMA 型の大規模並列計算機の 1 つに分類される．

JUMP-1 でのプロセッサには市販の RISC プロセッサ (SuperSPARC+) を採用した．というのも，大きなプロセスコンテキスト (レジスタおよびシステムレジスタ) を導入することによって生じる逐次部分を高速に処理できるからである．また，分散共有記憶アーキテクチャにおいて特徴的に現れる細粒度並列処理を効率的に実現することを目的に，異なる要素処理装置を用いた粗粒度操作と細粒度操作からなる複数粒度による並列処理を基本とする．したがって，RISC プロセッサと専用の分散共有記憶コントローラを備える．さらに，キャッシュシステムでは，効率の良い分散共有記憶プロトコル及び共有記憶空間にお

ける語単位の同期操作を実現する必要がある．それにより，効率の良い共有記憶及び共有記憶を介したメッセージ通信やスレッド操作をプロセッサに対して提供する．また，分散共有記憶アーキテクチャでは結合網に対する負荷が増大するので，結合網に高い転送能力とバンド幅が求められる．さらに，分散共有記憶実現のために頻繁に起こるマルチキャストのサポートが必須となる．これらに加えて，データを格納するディスクシステムやデータの視覚化のための画像入出力システムを接続することで，アプリケーションの分野を広げることができる．ただし，これらは十分な入出力バンド幅を確保するため，入出力サブシステムを構成する方が望ましい．

以上の実現に向けて，JUMP-1 全体としてクラスタ化したアーキテクチャを採用した．これは次の理由による．

(1) クラスタ化によって処理に内在する局所性を利用し，高速化を可能とする．

(2) クラスタ化によって通常の数値計算に必要なネットワークのコストを低下できる．

(3) クラスタ化によってネットワークの使用効率を向上させることが可能である．

(4) 階層化された同期を含む共有記憶プロトコルを採用することにより，ネットワーク負荷を低下させることが可能である．

(5) 当時の基板実装技術では 1 クラスタを 1 つの基板上に実装することが可能である (実際には CPU はドータボード上への実装となった)．

(6) プロセッサから直接参照できる主記憶の量が増大する．

(7) クラスタ内部の結合密度が高くなると，自動並列化コンパイラによる局所並列性の抽出が容易になる．

3.3　JUMP-1 の構成

JUMP-1 は最大 1024 プロセッサ (256 ノード) を目標としたシステムである．2000 年 3 月に完成した，16 クラスタ 64 プロセッサのシステムを図 3.1 に示す．

図 3.2 に示すようにクラスタ間は結合網 RDT(Recursive Diagonal Torus)[12, 13, 14] で接続されている．JUMP-1 で用いられる RDT は 2 次元トーラスの fat-tree 状の階層構造をもつ．また，RDT router chip[28, 29] は慶應義塾大学により実装された．さらに，各クラスタは並列入出力サブシステムを構成する高速 serial link(STAFF-Link[30]) によりディスク及び画像データ用 Frame Buffer(FB)[31] と接続される．

JUMP-1 クラスタの構成を図 3.3 に示す．4 台の RISC プロセッサである SuperSPARC+ はそれぞれ京都大学により開発された 2 次キャッシュ(L2 cache) [32, 33, 34] をもつ．そして，2 次キャッシュは同様に京都大学により開発された 64 bit 幅のクラスタバス [35] を介して，分散共有記憶コントローラである MBP(MBP-light) に接続される．MBP は東京大学の松本氏 [36] によって提案され，MBP-light [37] として主に慶應義塾大学により実装された．ただし，MBP-light 内にあるメモリコントローラは東大の佐藤 充氏が担当した．

図 3.1 16 クラスタ 64 プロセッサの JUMP-1

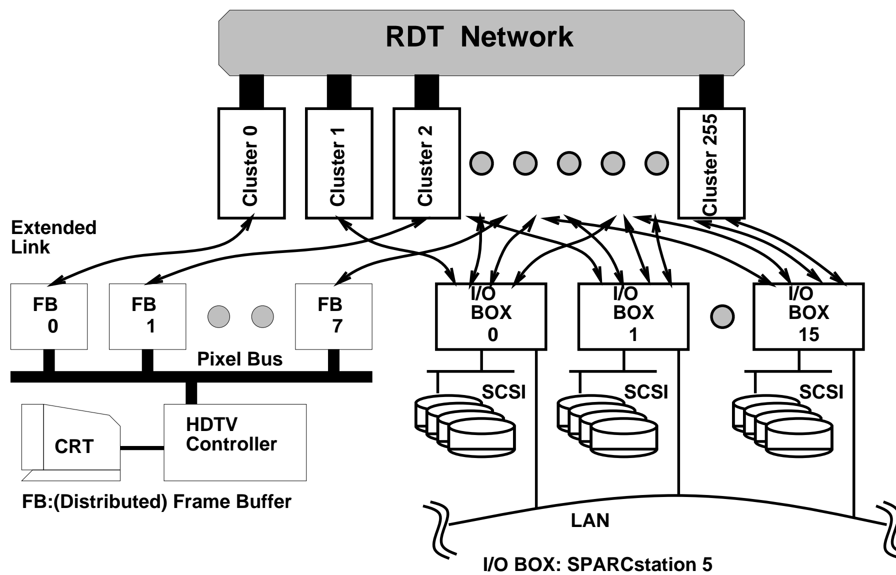


図 3.2 JUMP-1 の構成

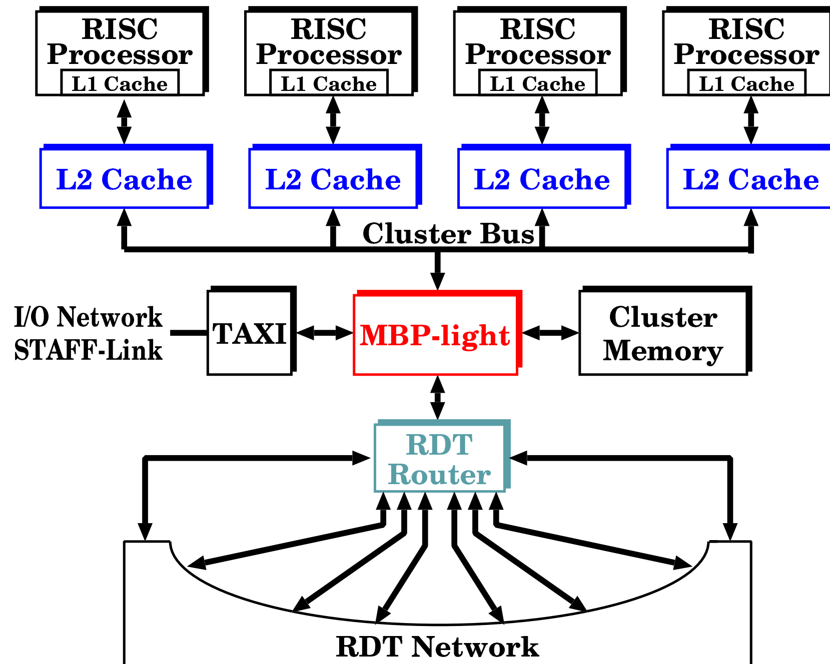


図 3.3 JUMP-1 クラスタの構成

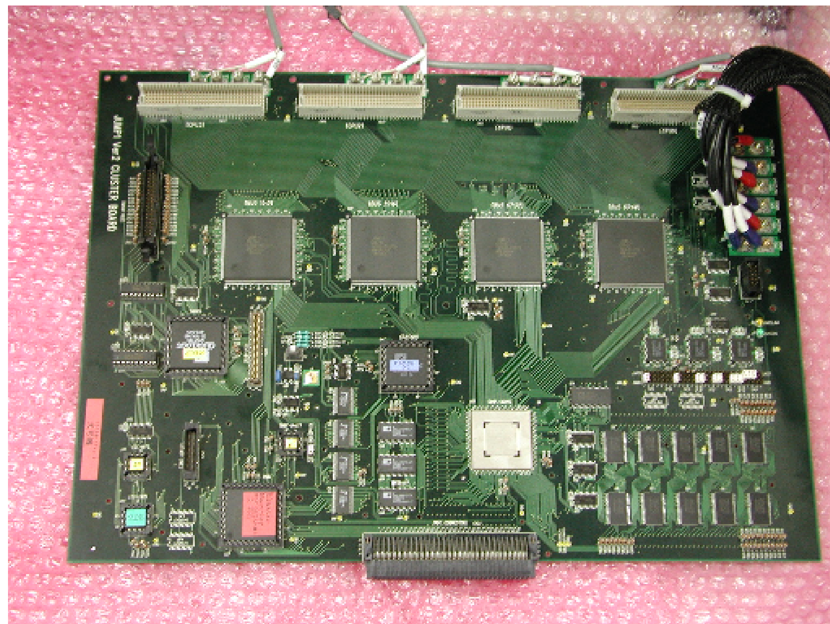


図 3.4 JUMP-1 クラスタボードの写真

MBP-light には，分散共有記憶及び 3 次キャッシュ(L3 cache) として用いられる 16Mbyte のクラスタメモリや STAFF-Link を構成する TAXI chip，RDT router chip が接続され

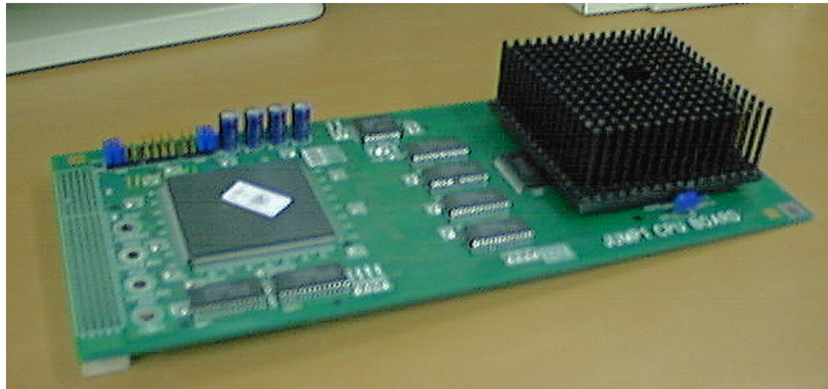


図 3.5 JUMP-1 プロセッサボードの写真

る．ほかにも，高機能な分散共有記憶を実現するために，1 つのキャッシュラインに対して 8 bit のタグを設け，512Kbyte の SRAM に格納する．同様に，64 bit データに対して 2 bit のタグを設け，512Kbyte の SDRAM に格納する．MBP-light は 2 次キャッシュとの協調動作やクラスタ間の通信，I/O とのデータ送受などの処理を行う．つまり，MBP-light が JUMP-1 のクラスタにおいて中心的な役割を担うことになる．

図 3.4 に JUMP-1 クラスタの写真を示す．4 つ並んでいるチップはクラスタバスを構成する bus chip である．そして，中央の銀色のチップが MBP-light であり，右下にはクラスタメモリがある．また，このクラスタボードの上部にあるコネクタにプロセッサおよび 2 次キャッシュを載せたボードをつなげることになる．このボードを図 3.5 に示す．

ボードの左側にあるチップが 2 次キャッシュコントローラであり，ヒートシンクの付いた右側のチップが SuperSPARC+である．さらに，これらのクラスタは RDT バックプレーンボードにつなげる．バックプレーンボードについては図 6.12 を参照されたい．

3.4　要素プロセッサおよび 2 次キャッシュ

JUMP-1 は，要素プロセッサとして SuperSPARC+プロセッサを用いており，京都大学により実装された 2 次キャッシュコントローラをもつ．2 次キャッシュコントローラは，JUMP-1 で採用された遷移的矛盾を伴う一貫性制御方式を実現するよう実装されている．これらについては第 4 章で詳述する．

3.5　バリア同期機構

JUMP-1 では，重複可能で，同期要求と同期承認を別個に扱うことができる Elastic Barrier を採用し，同期待ちによるオーバヘッドの削減を図っている．Elastic Barrier については第 4 章で詳述する．

3.6　結合網 RDT(Recursive Diagonal Torus)

RDT(Recursive Diagonal Torus) は基本のトーラス構造の上に目の粗いトーラスを 45 度ずつ傾けながら再帰的に積み上げていくことにより，トーラス構造と階層構造の両方を満足させることを狙っている．JUMP-1 の大きな特徴の 1 つは，RDT の階層性を利用し，近傍に対するマルチキャストとそれに対する応答パケット収集を高速に行う機構を持つ点である．これは，CC-NUMA のキャッシュコヒーレンス維持を高速に行うだけでなく，アプリケーションで用いるマルチキャストならびにブロードキャスト通信を高速に行う事ができる．

RDT については第 6 章で詳述する．

3.7　メインテナンスシステム

メインテナンスシステムは，JUMP-1 の運転監視装置としての役割を担うシステムであり，ホスト PC からのプログラム転送，JUMP-1 の起動や停止，JUMP-1 とホスト PC の間での小規模な通信等を行うことができる．

メインテナンスシステムについては第 7 章で詳述する．

3.8　Disk 入出力サブシステム

JUMP-1 の分散共有メモリは，1000 プロセッサを超える規模でのスケーラビリティを考慮して設計されている．したがって，計算機システム全体の処理能力の鍵の 1 つである入出力性能も同じようにスケーラビリティを持つものでなければならない．このため，全てのクラスタに入出力のためのインターフェイスを装備し，入出力サブシステムは複数の入出力ユニットで構成する．そしてこれらを並列に動作させることによって物理的アクセスを分散させるという方針をとっている．

ディスク入出力サブシステムの詳細については第 8 章 で詳述する．

3.9　MBP-light

MBP-light は，JUMP-1 クラスタの中心に位置する分散共有メモリ管理コントローラであり，プロセッサ，分散共有メモリを構成するクラスタメモリ，RDT ネットワークとの間の転送を行う．MBP-light では，基本的なトランザクションをハードウェア処理する一方，Core プロセッサを備え，複雑な例外処理をソフトウェア実行する事による回路規模の削減や，プログラムの変更によるプロトコルの柔軟性を兼ね備えている．

MBP-light については第 5 章で詳述する．

第4章　超並列計算機JUMP-1の構成要素

本章では，JUMP-1 の構成要素のうち，筆者の関与があまりないもののいくつかについて，簡単に述べる．ここでは，要素プロセッサである SuperSPARC+，2 次キャッシュ，クラスタバス，Elastic Barrier について触れる．メインテナンスシステム，結合網 RDT，MBP-light，Disk 入出力サブシステム，分散共有メモリについては，次章以降で評価も含めて述べる．

4.1　SuperSPARC+

JUMP-1 のプロセッサには，RISC である SuperSPARC+を用いる．SuperSPARC+は SPARC V8 architecture [38] を実装したものである．最大動作周波数は 50MHz であり，スーパスカラ機能を有する整数演算器をもつ．最大で 2 個までの演算と 1 個のメモリ参照命令を同時に実行することができる．また，20Kbyte の命令キャッシュと 16Kbyte のデータキャッシュを備える．詳細を表 4.1 に示す．

表 4.1 1 次 cache の諸元

	命令	データ
アドレシング	物理アドレス	
更新方式	—(read only)	write-through/-back
置換方式	LRU	
状態	Valid	Valid/Dirty/Shared
コヒーレンス制御	Invalidation	
容量	20KB	16KB
ラインサイズ	64B	32B
ウェイ数	5	4
セット数	64	128
store buffer	—	8 double words

実際の JUMP-1 では外部キャッシュ(L2 cache) を設けるために，更新方式としては write-through となる．また，SuperSPARC+には SPARC 参照記憶管理機構も含まれている．それは TLB(Translation Look aside Buffer) を用いて 32 bit の仮想アドレスを 36 bit の物理アドレスに変換する．詳細を表 4.2 に示す．

また，SPARC V8 architecture では memory consistency model として TSO と PSO が提供されている．このような種々の内部資源を参照するために，SPARC では ASI(Alternative

表 4.2 TLB の諸元

エントリ数	64(fully-associative)
ページ属性	Cacheable/Modified/Referenced, Permission={read, write, execute}for{Supervisor, User}
ページサイズ	4KB, 256KB, 16MB, 4GB

Space Identifier) というものが設けられている．これを切り換えることで資源を選択することになる．そこで，最後に SPARC V8 architecture で推奨されている ASI の割当てを表 4.3 に示す．

4.2　2 次キャッシュ

この節では，京都大学により設計及び実装された 2 次キャッシュ [32, 33, 39, 40, 41] について述べる．

4.2.1　一貫性制御方式

現在までに開発された多くの共有記憶型並列計算機においては，同一ブロックに対する一貫性制御による値更新や状態遷移はキャッシュ間で同じ順序で行われる．この一貫性制御方式を ACS-CC(Always Coherent Scheme of Cache Coherence) と呼ぶことにする．ACS-CC では，各アドレスに対して load 命令により観測できる値がどのプロセッサに対しても同じ順序で変化していくことになる．ACS-CC の典型的な実装方法を図 4.1 に示す．ただし，図中の $\circ$ はその時点でキャッシュの状態を変化させてもよいことを，$\times$ は逆に変化させてはいけないことを意味する．

一方，JUMP-1 ではすべての 2 次キャッシュにおいて状態の遷移順序が同じとは限らない．これは遷移的矛盾を伴う一貫性制御方式 (TIS-CC : Transitionally Incoherent Scheme of Cache Coherence) と呼ばれる．TIS-CC では各アドレスに対し当該アドレスに対する未完了の store がシステム内に存在しなければ，どのプロセッサから見ても同じ値が観測できることになる．よって，制約を緩和した分だけ TIS-CC は性能の点において従来の ACS-CC より良いことが知られている [39]．TIS-CC の典型的な実装は図 4.2 に示すようにして行うことができる．すなわち，従来の ACS-CC と同様に同期点を設けるものの，ACS-CC では許されていなかった同期点を通過する前にキャッシュの状態を遷移させてもよい．ただし，後に同期点を通過したパケットによって状態や値を上書きし，プロセッサ間で観測できる値の一致性を保証する．

この方式を最適化するべく，JUMP-1 では同期点を複数配置することで，トランザクションの通過経路短縮や流量の低減を図っている．その同期点は 2 次キャッシュやクラスタバス，主記憶 (MBP-light) に相当する．また，最終結果の上書きの削減を行い，負荷の低減を図っている．つまり，共有状態がクラスタ内に閉じている場合，トランザクションが終了する前に

表 4.3 ASI の割り当て

ASI	Function	Access	size
0x00–0x01	Reserved	—	—
0x02	Control Space Access	LD/ST	all
0x03	MMU Probe	LD/ST	single
0x04	MMU Registers	LD/ST	single
0x05	Reserved	—	—
0x06	MMU TLB Diagnostics	LD/ST	single
0x07	Reserved	—	—
0x08	User Instruction	LD/ST	all
0x09	Supervisor Instruction	LD/ST	all
0x0A	User Data	LD/ST	all
0x0B	Supervisor Data	LD/ST	all
0x0C	Instruction Cache Tags	LD/ST	double
0x0D	Instruction Cache Data	LD/ST	double
0x0E	Data Cache Tags	LD/ST	double
0x0F	Data Cache Data	LD/ST	double
0x10–0x1F	Reserved	—	—
0x20–0x2F	MMU Bypass	LD/ST	all
0x30	Store Buffer Tags	LD/ST	double
0x31	Store Buffer Data	LD/ST	double
0x32	Store Buffer Control	LD/ST	single
0x33–0x35	Reserved	—	—
0x36	Instruction Cache Flash Clear	ST	single
0x37	Data Cache Flash Clear	ST	single
0x38	MMU Breakpoint Diagnostics	LD/ST	double
0x09	BIST Diagnostics	LD/ST	single
0x3A–0x3F	Reserved	—	—
0x40–0x41	Emulation Temps	LD/ST	single
0x42–0x43	Reserved	—	—
0x44	Emulation Data In	LD	single
0x45	Reserved	—	—
0x46	Emulation Data Out	LD/ST	single
0x47	Emulation Exit PC	LD/ST	single
0x48	Emulation Exit nPC	LD/ST	single
0x49	Emulation Counter Value	LD/ST	single
0x4A	Emulation Counter Control	LD/ST	single
0x4B	Emulation Counter Status	LD/ST	single
0x4C	Action Register	LD/ST	single
0x4D–0xFF	Unassigned	—	—

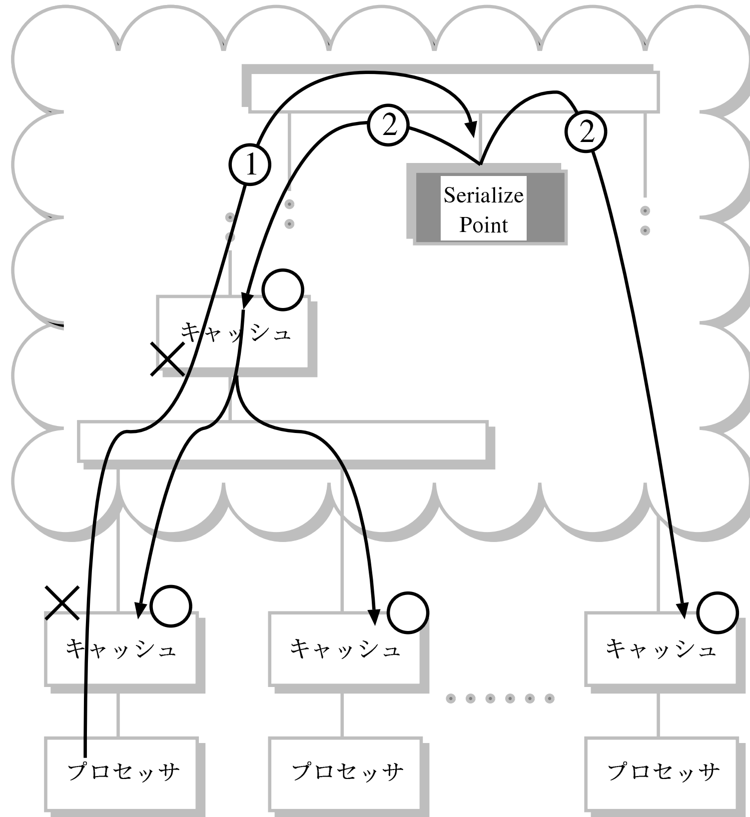


図 4.1 ACS-CC の典型的な実装方法

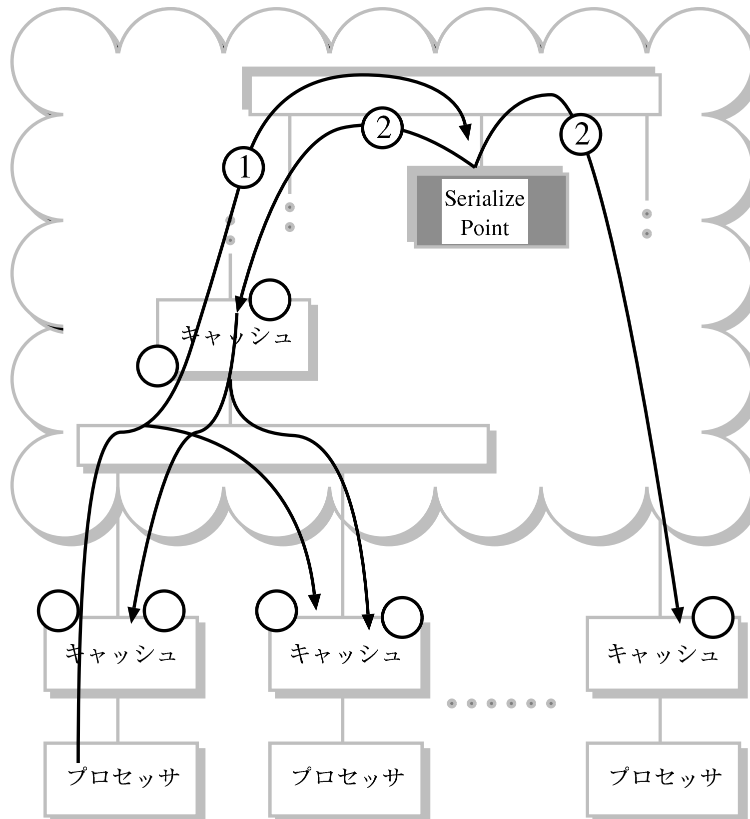


図 4.2 TIS-CC の典型的な実装方法

- 同一ラインに対して無効化要求がなされた場合は，無効化されたことを検出して，後に上書きを行う
- それ以外の要求がなされた場合は，その要求による当該ラインの状態遷移及び store を行ったアドレスの値が変化しないように保護する

方法を採用している．

4.2.2 トランザクションの逐次化

TIS-CC を実現するためには，いくつかのトランザクションの逐次化をしなければならない．そうしなければ，各プロセッサから観測される値が最終的に一致しなくなってしまう．その逐次化の機構を次に述べる．

- クラスタバスにおける逐次化

  クラスタバスにパケットが流れた時，以下の条件を満たした際にそのパケットにリトライを要求する．

  – 応答が必要なトランザクションをクラスタバスに送出した後，応答がまだ返っていない同一アドレスを有するパケットが 2 次キャッシュのどれかに存在する場合
  – 同一アドレスを有する応答パケットを送信しようとしている，または送信しなければならない場合

- 主記憶 (MBP-light) における逐次化

  応答が必要なトランザクションは主記憶を通過する際に，後続のトランザクションを停止させるために，プロセス表へ登録を行う．後続トランザクションの処理の流れを図 4.3 に示し，以下に説明する．

  (1) プロセス表を検索し，当該アドレスに対して応答が必要な未完了トランザクションが存在するかどうかを調べる．
  (2) 存在すれば，そのトランザクションを suspend buffer に登録する．
  (3) 後に先行トランザクションの応答が戻ると，suspend buffer を検索して停止されているトランザクションがあるかどうか検索する．
  (4) 存在すれば，ready queue に移動する．
  (5) ready queue に移動後，しばらく後にプロセス表を検索することから再開する．

- 結合網における逐次化

  パケットの FIFO 性を保証しなければならない．また，store に関連したトランザクションであれば，その要求を発生したクラスタも同期点からのパケットが確実に届くようにする必要がある．

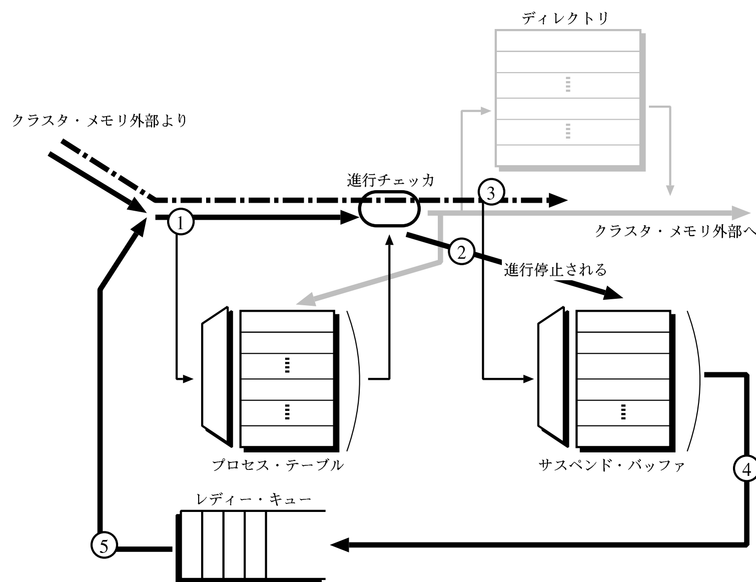


図 4.3 主記憶における逐次化

4.2.3　2 次キャッシュの実装諸元

ここで，2 次キャッシュの実装諸元について表 4.4 に示す．2 次キャッシュは東芝の 0.5μm で CMOS 2 層 metal の gate array により実装された．ピン数は 304 で，パッケージとして QFP を用いている．

ただし，write buffer は同一キャッシュラインに対するマージ機能を備える．

4.2.4　キャッシュコヒーレンス制御

JUMP-1 におけるメモリへの参照は通常のシステムと同じように load/store を用いて行われる．しかし，参照する物理アドレスによって種々の 2 次的な効果が得られる．例えば，動的かつ明示的にキャッシュの一貫性を保つプロトコルが規定されたり，同期構造への操作となったりする．この効果のことを memory command と呼ぶ．

memory command は SuperSPARC+の提供する 36 bit の物理アドレスのうち，上位 4 bit によって指定される．よって，16 種類の memory command が各々のアドレスに定義されることになる．ちなみに，JUMP-1 ではさらに 4 bit を付加的な情報として使用するため，物理アドレスは下位 28 bit によって定められる．これらの関係を図 4.4 に示す．

ここで，V_x は仮想ページを示し，P_x は SuperSPARC+が提供する物理ページを，P は実際の物理ページを示す．このように $\mathrm{V_M}$ と $\mathrm{V_N}$ は同一の物理ページを参照しているにも関わらず，memory command($\mathrm{P_M}$ と $\mathrm{P_N}$) が異なっているため，参照時の効果は違うものとなる．ちなみに，ある物理アドレスに対し最大 4 つの異なる memory command を持つことが可能である．しかし，そのような状況を扱うには処理が非常に複雑なものとなるため，2 次キャッシュは大体 2 つまでになるように制御を行う．memory command の詳細は付録 E を参照のこと．

表 4.4 2 次キャッシュの実装諸元

アドレシング	物理アドレス
更新方式	write-through/-back
状態	I/VXLDO/VXLCO/VSLDO/VSLCO VSLCN/VSGCO/VSGCN/VXLDN/VSGDN
制御方式	Invalidate/Update
容量	1Mbyte
ラインサイズ	32byte
ウェイ数	1(direct map)
セット数	32Kbyte
スヌープ機構	あり (キャッシュ間転送もあり)
プリフェッチ機構	あり (2 ライン以上)
write buffer	8 double words

V: Valid　I: Invalid　X: Exclusive　S: Shared　L: Local
G: Global　C: Clean　D: Dirty　O: Owner　N: Non Owner

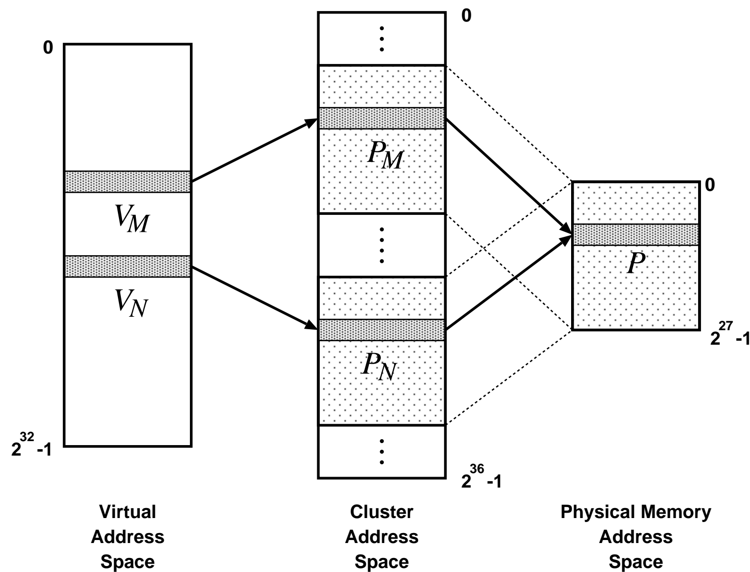


図 4.4 memory command とアドレスの関係

4.2.5　2 次キャッシュのブート

control space 内の 0x10000150(`L2CRDY`) という内部資源がある．リセット時には 0 であり，CPU のリセット割込みルーチンの実行を抑止している．2 次キャッシュを含むメモリシステムの初期化後に，MBP-light から制御用パケット (`control_write_req`) で 1 にする必要がある．これにより，CPU はリセット割込みルーチンの実行を開始することができる．ちなみに，0 を書き込むことは想定されておらず，実行の結果は保証されない．

4.3　クラスタバス (CBus)

本節では，京都大学により設計及び実装された Cluster Bus Chip [35, 34] について述べる．クラスタバスは split transaction を行う 64 bit のバスである．クラスタバスは 4 個の ASIC によって構成される．クラスタバスのパケット及び制御線の詳細については，付録 D を参照されたい．

4.3.1　アービトレーション機構

クラスタバスは 2 段階の優先度をもつ集中アービトレーションを用いる．16 bit 毎にスライスされた各 Bus Chip はそれぞれアービタをもち，同一のアービトレーション動作を行う．以下にその方式を示す．

(1) 送信可能な reply 系要求がある場合には，必ず request 系要求に先立って送信が行われる．

(2) 優先順位は MBP-light，0 番キャッシュ(PU 0 のキャッシュ，以下同様)，1 番キャッシュ，2 番キャッシュ，3 番キャッシュの順で固定とする．しかし，ある時点でサンプリングされた要求すべてに対してサービスが行われた後に，新しい要求をサンプリングする．

4.4　Elastic Barrier

この節では，東京大学の松本氏によって考案された Elastic Barrier [42, 35] について述べる．全体構成は図 4.5 に示すように階層構造をとっている．これにより，システム全体をパーティションに分割して使用することができる．

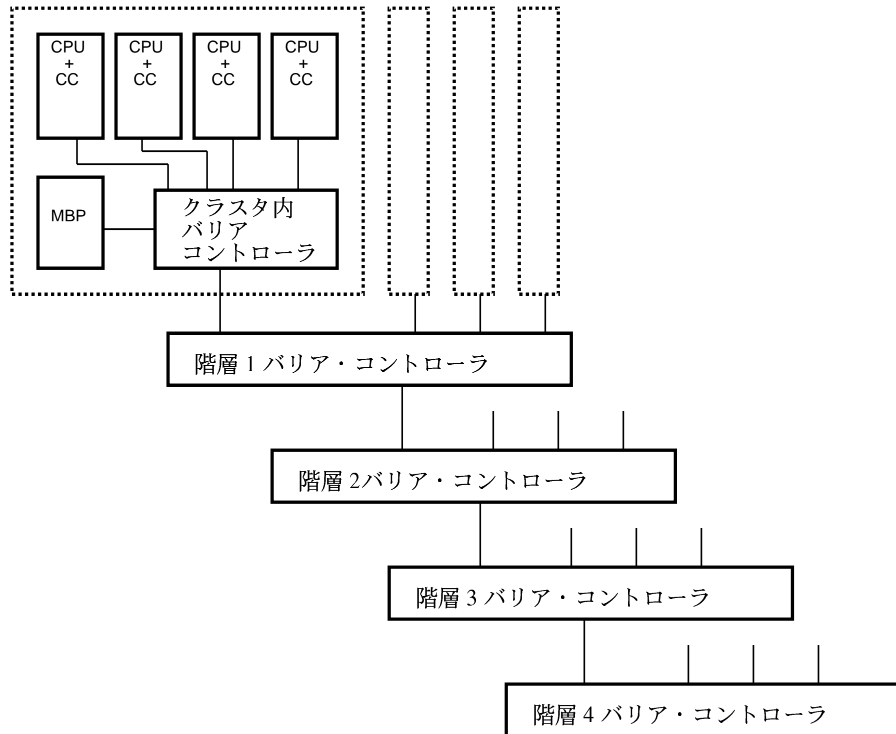


図 4.5 Elastic Barrier の全体構成

4.4.1　Elastic Barrier の概要

Elastic Barrier は重複可能で，同期要求と同期承認を別個に扱える方式のバリア同期である．単純なバリア同期とは異なり，次の 3 種類の primitive をサポートする．

(1) RREQ

同期情報を検査し，同期の成立を検出する．待合わせは検出結果による分岐で実現される．この節では，常に同期の成立を待ち合わせることとし，成立するまで命令実行中断状態になるものとする．

(2) APRV

同期情報を出力後，命令実行を続ける．ダミーの同期要求や生産者 · 消費者の関係での生産者側の場合等，他のプロセッサ上のタスクの実行終了を待つ必要のない場合の同期情報．意味は，次回の同期成立の承認．

(3) PREQ

同期情報を出力後，命令実行を続ける．RREQ の前に挿入して，後続の RREQ を予告するための同期情報．意味は，次回の同期成立の先取り．

この拡張された同期情報を使って，同期のオーバヘッドが発生する可能性を低減する．

図 4.6 は，ダミーの同期要求に伴うオーバヘッドを削減する方法を示している．× 印は RREQ が挿入されている場所を意味する．また，PU2 はダミーの同期要求を同期位置 2 の付近に挿入する必要がある．同期情報を拡張する前の機構では，PU2 での同期位置 2 を実行前に見積もり，RREQ をダミーとしてその地点 (黒丸) に挿入する．しかしながら，実行時の予測できない要因により実際の同期位置 2 がずれて，プロセッサに不必要な待ちを生じさせる可能性がある．そこで，RREQ の代わりに APRV を用いる．図の白丸の位置に挿入すれば，オーバヘッドを避けることができる．同期コントローラは同期位置 1 の同期の完了後すぐに APRV を PU2 から受けとる．これにより同期位置 2 のために同期信号線をアクティブにする．一方，PU2 は独立に命令の実行を継続する．同じ PU 上で APRV が連続することを許すために同期承認カウンタが設けられ，同期条件未成立の APRV の数を計数する．同期承認カウンタは APRV を出すたびにインクリメントされ，同期信号線がアクティブになるたびにデクリメントされる．そして，同期承認カウンタが 0 になるまでこの動作は続く．

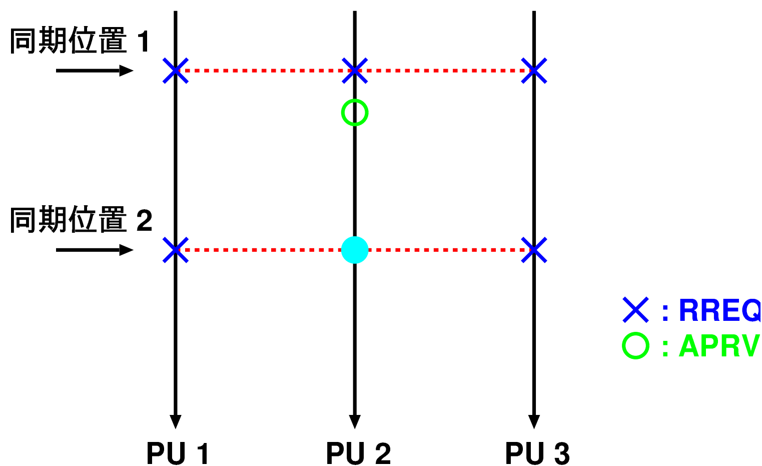


図 4.6 ダミーの同期要求に伴うオーバヘッドの削減法

図 4.7 は生産者 · 消費者の依存を守る同期に伴うオーバヘッドの削減を示している．PU1 上の APRV(白丸) が生産者に対応し，PU2 での 2 番目の RREQ(× 印) が消費者を表している．共有記憶上でデータの受渡しを行う場合，生産者は対応する消費者が処理を始めるのを待つ必要がないので，生産の完了を示す同期情報として APRV を使う．PREQ を使

用せず，PU1 の生産者が処理を先に終了した場合，この同期に対応する同期条件の成立はPU2 の 2 番目の RREQ まで待たされる．しかし，生産者 · 消費者の同期の性質から考えると，同期条件は図の領域 S のどこで成立しても構わない．そこで，図の △ 印の地点にPREQ を挿入し，同期条件が PREQ と対応する RREQ で囲まれる範囲の任意の地点で成立可能にする．同期コントローラは PREQ を受けとると同期信号線をアクティブにし，APRV のときと同様に PU2 は命令の実行を継続する．PU2 は命令の実行が RREQ に到達したときに初めて，継続する命令が実行可能かコントローラに問い合わせる．同期条件がその間に成立していれば，コントローラは命令継続の許可を PU2 に与える．成立していなければ，成立するまで待つことになる．さらに，同期条件の成立順序が保たれる範囲で，RREQ と対応する PREQ の間に他の同期情報が挿入可能なように 2 つのカウンタを用いて拡張する．カウンタの 1 つは同期条件未成立の PREQ の数を計数する同期予告カウンタである．もう一方は，PREQ によって成立したが，RREQ によって確認されていない同期条件の数を計数する同期成立カウンタである．同期予告カウンタは同期承認カウンタとまったく同じ動作を行う．同期成立カウンタは PREQ による同期条件の成立を検出したときにインクリメント，RREQ により確認されるたびにデクリメントされる．

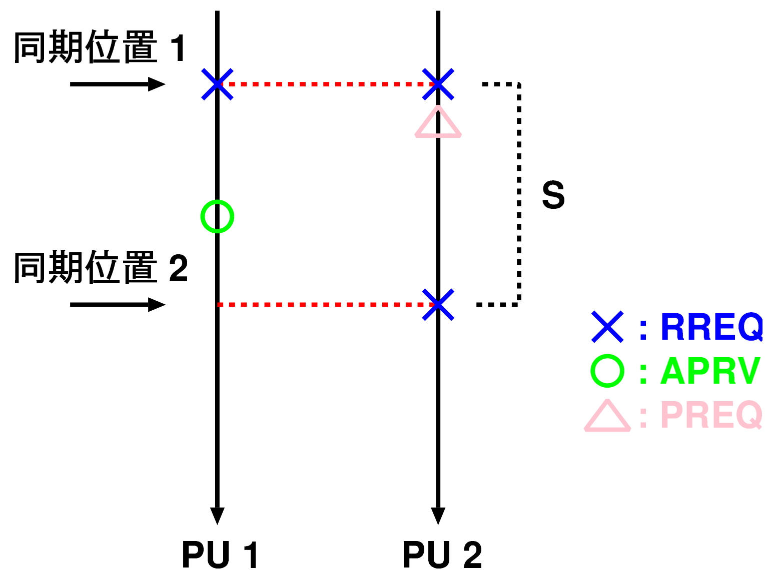


図 4.7 生産者 · 消費者の依存に伴うオーバヘッドの削減法

以上，基本的な Elastic Barrier の概要を例と共に述べた．ここで述べていない拡張等については，文献 [42] を参照されたい．

4.4.2 Elastic Barrier の実装

本節では，JUMP-1 における Elastic Barrier の実装について述べる．クラスタ内におけるコントローラを図 4.8 に示す．

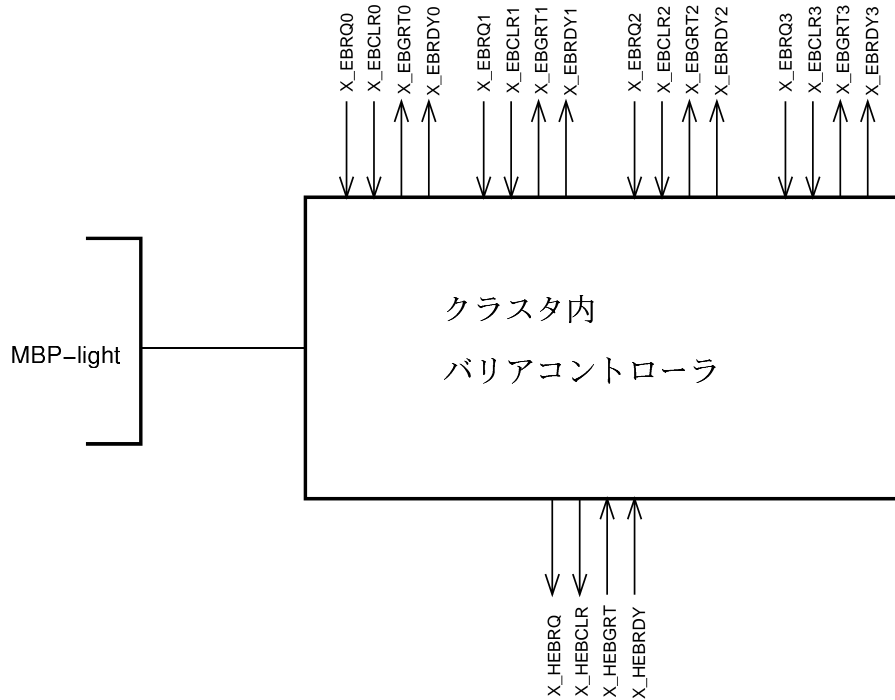


図 4.8 クラスタ内におけるコントローラの構成

信号線の意味を以下に簡単に説明する．ただし，添えられた数字はクラスタ内の 2 次キャッシュ(プロセッサ) に割り当てられた番号を表す．

- X_EBRQ

  Elastic Barrier による同期要求．

- X_EBRDY

  同期要求を出してよいことを示す．

- X_EBGRT

  Elastic Barrier が成立したことを示す．

- X_EBCLR

  コントローラの初期化．

残りの X_HEB{RQ, RDY, GRD, CLR} は次階層に対して設けられた信号線となっている．

それでは，Elastic Barrier をサポートするために，各デバイスがコントローラに対して用意している資源について説明する．まず，2 次キャッシュには Elastic Barrier に対する内部資源として 3 種類の primitive 及び 3 種類のカウンタ，clear 命令があり，Elastic Barrier をサポートしている．3 種類の primitive は次のとおりである．

(1) `EBAPRV`

任意の値を書くと Elastic Barrier の同期点報告を報告する．なお将来，その同期が成立したことを確認してはならない．

(2) `EBPREQ`

任意の値を書くと Elastic Barrier の同期点報告を報告する．なお将来，その同期が成立したことを確認しなければならない．

(3) `EBRREQ`

Elastic Barrier の未確認同期点の中で，最も古いものが成立していれば -1 を，そうでなければ 0 を返す (後述の `EBCCP` というカウンタが 0 であれば 0 になり，そうでなければ -1 になる)．

また，3 種類の 16 bit 幅のカウンタには次のものが用意されている．

(1) `EBCAP`

Elastic Barrier の未成立同期点のうち，`EBAPRV` で報告したものの数を保持するカウンタ．

(2) `EBCPR`

Elastic Barrier の未成立同期点のうち，`EBPREQ` で報告したものの数を保持するカウンタ．

(3) `EBCCP`

Elastic Barrier の成立した同期点のうち，`EBRREQ` で確認していないものの数を保持するカウンタ．

さらに，`EBCLR` という clear 命令によって 3 種類のカウンタ及び Elastic Barrir のコントローラ等の初期化が行うことができる．

次に，MBP-light とのインタフェースについて述べる．2 次キャッシュとは異なり，MBP-light には Elastic Barrier 用のインタフェースが設けられていない．よって，2 次キャッシュで用意されているカウンタは Elastic Barrier のコントローラ内に実装する．そして，それらに対する参照や 3 種類の primitive，clear 命令等は通常の I/O 空間に対してアドレスをマップして行うこととする．そのために，MBP-light から Elastic Barrier のコントローラに対しては，アドレス線の 0 bit 目から 3 bit 目と 12 bit 目から 15 bit 目が接続されている．また，データ線として下位 11 bit が，ほかにもローカルバスでの制御線が接続さ

れている．Elastic Barrier のコントローラには，MBP-light の local I/O address のうち，0x0000，0x1000 が割り当てられている．

第5章　MBP-light

本研究では，JUMP-1における分散共有記憶管理プロセッサであるMBP-light [37, 43, 44]が備えるCoreプロセッサの命令セットアーキテクチャについて，実機による評価を行うとともに，その特殊なアーキテクチャの利点，改善すべき点について詳細な検討を行った．

本章では，まずJUMP-1の分散共有記憶管理について述べ，これを実現する分散共有記憶管理プロセッサMBP-lightについて述べる．その際，本研究と関連するCoreプロセッサの命令セットについて設計思想および概要を述べる．また，第6章で評価結果を述べるMBP-lightが備える応答パケットの自動生成，収集機構についても説明する．その後，筆者が担当した命令セットアーキテクチャの評価およびアーキテクチャを検討した結果について述べる．

5.1　JUMP-1における分散共有記憶管理の概要

東京大学の松本氏によって提案された分散共有記憶の管理手法 [36, 45] を基本構想としたJUMP-1の分散共有記憶の管理手法について説明する．

5.1.1　分散共有記憶管理のための基本構想

JUMP-1において，MBP-lightは効率の良い分散共有記憶を実現するための核となる．MBP-lightは本質的に局所性のない処理を要素プロセッサから分離し，並列処理することを目的としたプロセッサである．ただし，その処理を高速化するために若干のhardwired-logicを付加するものとする．

JUMP-1では，すべてのプロセッサが単一かつグローバルなアドレス空間を持ち，分散共有記憶を介して通信を行う．各クラスタはページ (4K Byte) 単位でメモリを管理し，各プロセッサはpage tableを管理することで仮想共有記憶空間を提供する．page tableの管理はプロセッサのMMU (Memory Management Unit) で行われ，仮想アドレスから物理アドレスの変換に利用される．

JUMP-1では，MBP-lightに接続されているクラスタメモリの一部を他のクラスタメモリのキャッシュ領域として利用する．これは3次キャッシュ(L3 cache) と呼ばれ，クラスタ外へのメモリ参照時間を大幅に減らすことが可能となる．IVY [46] などで用いられている仮想共有記憶ではページ単位でデータを転送する．しかし，小さなデータ構造を共有する場合などにはfalse sharingが生じてしまい，データに対する参照時間が非常に大きなものとなってしまう．よって，JUMP-1では管理単位をページとするものの，キャッシュライン単位にタグを付加し，キャッシュライン単位でデータを転送する．

クラスタ内ではクラスタバスをスヌープすることで，2次キャッシュ間のコヒーレンス

を保っているが，クラスタ間ではスヌープ処理を用いることはできない．そこで，クラスタ間ではページの共有情報をディレクトリ方式で管理する．詳細は 6.1 節を参照されたい．

最後に，クラスタメモリ A が B のデータを共有する場合の一例を図 5.1 に示す．まず，

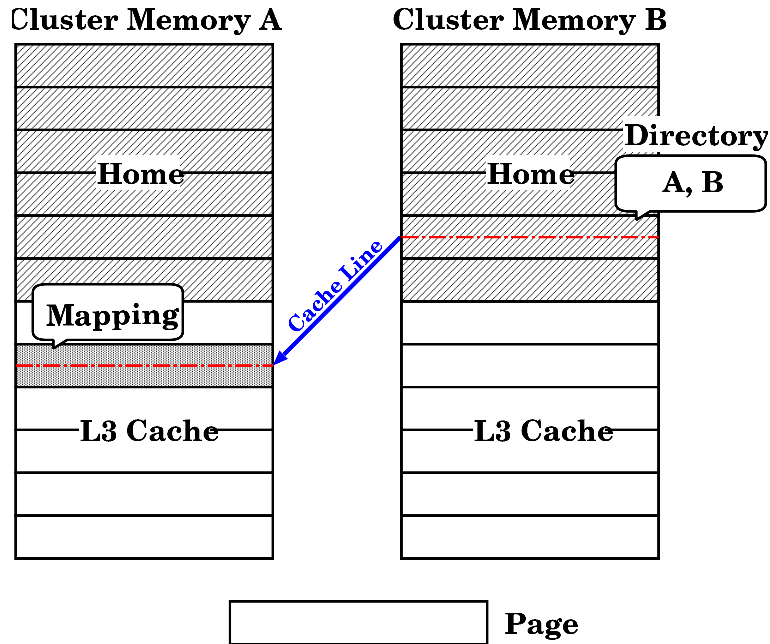


図 5.1 JUMP-1 におけるクラスタメモリ管理の一例

A は B の該当するページを自身の 3 次キャッシュ領域にマップする．そして，実際に共有するデータをキャッシュライン単位で B に要求する．B は要求を受けると，当該キャッシュラインを転送するとともに，共有情報をページ単位で付加されたディレクトリで管理する．

5.1.2　アドレス変換

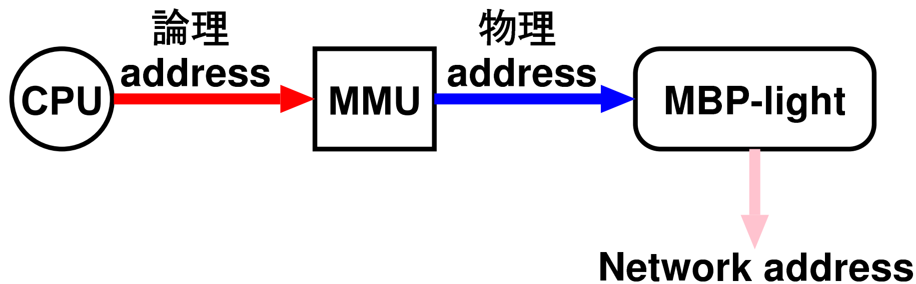


図 5.2 JUMP-1 におけるアドレス変換

JUMP-1 ではマルチユーザ/マルチタスクを実現し，その基盤として仮想共有記憶空間を提供する．JUMP-1 において行うべきアドレス変換の様子を図 5.2 に示す．仮想共有記

憶においてクラスタ間でのみで用いられるアドレス (network address) が仮想化されており，特定のクラスタと関連づけられることはない．これにより，クラスタ間でのメモリ領域の migration 等を行う際の管理が容易となる．また，クラスタメモリはキャッシュ領域としても利用されるため，network address は複数のクラスタメモリのアドレス (物理アドレス) にマップされる必要がある．network address と物理アドレス間のアドレス変換は各クラスタ内で行われる．さらに，要素プロセッサ内の仮想アドレスと物理アドレス間でも通常のページ機構によるアドレス変換が行われる．よって，JUMP-1 では仮想アドレスと物理アドレス，network address の3種類のアドレスを管理することが必要となる．network address はシステム全体で名前資源の管理が行われ，物理アドレスはクラスタ内で管理される．network address は物理アドレスと独立に管理されるが，仮想アドレスから network address を生成する方法を，例えば network address を task ID とするなど，ある程度系統的に行う方が高速である．

5.1.3　ページ管理とデータ転送

JUMP-1 では仮想共有記憶を採用し，メモリをページ単位で管理する．よって，クラスタ間の共有情報を管理するディレクトリはページ単位に設けられる．Stanford 大学の DASH 等の従来の CC-NUMA においてはディレクトリは要素プロセッサのキャッシュライン単位で設けている．しかし，ディレクトリに必要なメモリ容量が大きくなる点とディレクトリの更新頻度が高くなる点を考慮すると，超並列計算機を目指す JUMP-1 では顕著にそれらの欠点が現れることが予想されるために採用しがたい．ただし，ページ単位のディレクトリ管理を行った場合に，データの転送単位もページ単位にしてしまうと，結合網を介した要求の待ち時間が長くなることから，データの転送は基本的にキャッシュライン単位で行う方がよい．

5.1.4　page fault 処理

要素プロセッサが自クラスタ内に割り当てられていないアドレスを参照した場合は page fault が発生する．要素プロセッサは page fault が発生したアドレスを含むページの割り当てを MBP-light に対して要求する．すると，MBP-light はページに関する情報を検索し，クラスタ内に page entry として必要な属性とアドレス変換表を設定する．必要であれば，そのページに対する領域をクラスタメモリ上に確保し，page fault を起こしたキャッシュラインを読み込む．その後，要素プロセッサの内蔵 TLB に page entry を設定し，プロセッサに処理を再開させる．MBP-light がこの page fault の処理を行っている間に，要素プロセッサは他のコンテキストの実行を行うことができる．また，MBP-light に対する page prefetch 命令を要素プロセッサに用意して，MBP-light に page fault 処理を先行実行させることも可能である．

5.1.5　page entry

page entry には，page 属性や network address(グローバルな共有アドレス)，ページ内にある dirty なライン数，処理中のトランザクション数のフィールド等を付加する．よって，要素プロセッサ本来の page entry を拡張したものとなる．クラスタ内で実行中のタスクの page entry は各クラスタメモリの非共有領域内 (page entry pool) にある．この page entry を生成する責任は operating system が担う．

5.1.6　page 属性

ページ内で保持すべき page 属性について述べる．

- ページの存在に関する属性

  (1) home page
  network address 空間においてページ毎に唯一存在する．実メモリの割当てを受けたページの home はどこか唯一のクラスタに存在する．システム全体に分散した home page がシステム全体の主記憶を形成する．

  (2) copy page
  3 次 cache 用のページ．同一 network address を有する home page の copy data を保持するための領域である．

  (3) dummy page
  page entry はあるが，ページ実体 (コピー用のデータ領域) のないページ．参照元のクラスタにデータを残さず，他のクラスタとデータを転送することができる．また，クラスタメモリ上に 3 次キャッシュ用のデータ領域の確保が困難な場合に，copy page の代わりとして使用することもできる．

- consistency protocol に関する属性
  4.2 節も合わせて参照されたい．

  (1) normal
  付加的な機能を一切設けない．ただし，一貫性の維持は行うものとする．

  (2) invalidate
  書込みに対して invalidate の方針を選択する．home page の存在するクラスタ (以降，単に home と記す) に書込み要求は必ず送られる．そこで，ライン単位で書込み要求は逐次化される．home の MBP-light が他の copy page の該当ラインをすべて無効化した後に，書込み要求元でデータの書込みが行われる．

  (3) update_direct
  書込みに対して update の方針を選択する．home 以外のコピーをもつクラスタもページ単位にディレクトリを管理して，home を経由しない update を行う．また，読出しによるデータ転送と update のデータ転送が競合を起こさないよ

うに，読出し時は同期を取る必要がある．update_direct は多くの制約があるが，この制約が満たされることが保証されていれば，データは直接コピーをもつクラスタに転送することが可能となる．これにより，転送時間を最小にすることができ，ネットワーク上を一貫性管理のために伝達されるメッセージの数も少なくできる．

(4) update(_via_home)

書込みに対して update の方針を選択する．home を経由した update を行う．また，読出しによるデータ転送を必ず home から行えば，update のデータ転送との競合問題は生じない．さらに，copy page を必要とするクラスタは home の共有情報に登録し，その時点での home のデータを読み出すことで一貫性に関する問題を避けることができる．update_via_home は転送時間が update_direct よりも大きいが，使用上の制約が少ない．そして，ネットワーク上を一貫性管理のために伝達されるメッセージの数は update_direct とほぼ同程度に少ない．ただし，invalidate と同様に home の処理負荷が集中する．

(5) no_coherent(read_only)

一貫性保持動作及び cache copy の管理を行わない．

(6) read_invalidate(exclusive)

書込み時のみでなく，読出し時に排他的に (他の cache copy を無効化して) 読み出す．

(7) allread

読出し時に経路途中にある同一論理グループのクラスタにページが存在する場合，そのページにもデータを張り付ける．

- 3 次キャッシュの replacement に関する属性

3 次キャッシュ中にある copy page の replacement の可/不可を制御するこの制御は MBP-light で実現することができる．本属性は replacement の際にわかればよい．ただし，無制限に copy page を replacement 不可の属性にすると，クラスタメモリ資源が枯渇するので，operating system 等で不可にできるページ数をユーザ毎に制限するなどの制約が必要となる．その制約のもとで，不可という属性の指定を可能にして update_direct の効果的な使用や copy page 上にも構造体を形成する同期 primitive 等の多様な処理形態を可能にする．対応する home が swap out される場合には copy page も swap out する．また，タスクが終了する場合にはエントリ自体を消去することになる．

- 一貫性管理 (データ転送) 単位に関する属性

(1) double word 単位

double word 単位にデータの転送を行う．

(2) line 単位

キャッシュライン単位に一貫性管理やデータの転送を行う．

- キャッシュ可能性に関する属性

  (1) cacheable
  クラスタ内もクラスタ間もキャッシングを認める．

  (2) non-cacheable
  クラスタ内もクラスタ間もキャッシングを禁止する．

  (3) L1_non-cacheable
  プロトコルの種類の区別や同期の制御を物理アドレス上位ビットで行うページに対しては，内蔵1次キャッシュを無効化する必要がある．

  (4) L2_non-cacheable
  参照頻度が少ない場合に，1次及び2次キャッシュへのキャッシングを禁止する．そして，遅延を削減するためにクラスタ間のキャッシングのみ認める場合に使用する．

- page migration に関する属性

  (1) cluster local
  私的な変数や work 領域のためのページであることを示す．クラスタ外から参照された場合はプロセス (または，スレッド) のクラスタレベルの migration が行われたことを意味する．この場合には参照元の cluster page(私的なので必然的に home page) の migration を行う．

  (2) group local
  特定のクラスタのみ参照するページであることを示す．クラスタの他に論理的なグループ (物理的に近接した複数のクラスタが割り当てられる) という概念 (階層構造も可) を準備し，あるグループ内からしか参照され得ないデータにはこの属性を付加する．もしグループ外から参照された場合はグループレベルの migration が行われたことを意味する．

  (3) global
  どのクラスタからも参照される可能性があることを示す．

5.1.7　クラスタ内一貫性管理

クラスタ内で閉じたメモリトランザクションに関しては，通常のスヌープ方式の一貫性維持を行う．クラスタバスによってクラスタ内におけるメモリトランザクションの単一性と順次性が保証される．しかし，クラスタ間で共有されているページ (home page 及び copy page) に関しては，ライン単位の転送に基づく一貫性管理をクラスタメモリ (3次キャッシュ) において行う必要がある．そのため，クラスタメモリにはキャッシュライン毎に 8 bit の制御用のタグをもつ (5.3.3.2 節を参照されたい)．

5.1.8　メモリ参照の順序性と単一性

クラスタ内のメモリ参照はバス結合型のため，バスがメモリトランザクションの順序性と単一性を保証してくれる．しかし，クラスタ間にまたがるトランザクションでは保証されない．そのため，まずトランザクションの要求はそのアドレスの home に送られ，単一性を保証してから処理がなされる．つまり，同一ラインへの競合する複数の更新要求等は home の MBP-light で調停され，順次に更新が実行される．ただし，順序性保証の必要がないことが静的にわかるプログラムに対しては，その保証を行わないことも可能である．また，順序性保証のため，クラスタ間でメモリトランザクションやキャッシュの操作毎に acknowledgment(以下，Ack) を処理の要求元に返す必要が生じる．キャッシュコピーが多くのクラスタ上に存在する場合，キャッシュコヒーレンス制御のためのメモリトランザクションは時間がかかる処理となる．そのため，一貫性を損なうことがないような複数のトランザクションを並列に処理する機構や Ack を処理の要求元に高速に返す機構を持たせることが有効である．

5.1.9　分散共有記憶の高機能化

JUMP-1 の分散共有記憶はプロセッサ間の通信機能として I-structure や Q-structure, Block Queue といった同期機構をサポートし，高機能化を図っている．この同期構造のために，JUMP-1 では各 double word 毎にタグが設けられている (5.3.3.2 節参照のこと)．タグは full / empty を表す 1 bit とコンテキストの待ち状態を表す 2 bit の計 3 bit からなる．4.2 節で述べたように，このような高機能化された分散共有記憶は 2 次キャッシュを通じて以下の memory command によって参照することができる．

(1) I-structure: {s-read, s-write}

最も単純な生産者-消費者の通信に有効である．

(2) Q-structure: {q-read, q-write, p-read, q-check}

(3) Block Queue: {bq-read, bq-write, bq-check}

I-structure は生産者が複数である時にその通信をうまく処理することができない．そこで，queue を設けることで，複数の生産者の通信に有効となる．

図 5.3 と図 5.4 に以上の同期機構の概念図を示す．

5.2　設計方針

まず，MBP-light に対する要件を列挙する．

(1) クラスタ内でのメモリ参照や縮約階層 bitmap directory 方式の性能を十分発揮するために，応答パケットの生成や収集をできるだけ高速に行う必要がある．また，この処理が他のパケットの処理をなるべく妨げないような構成をとり，負荷分散する必要がある．

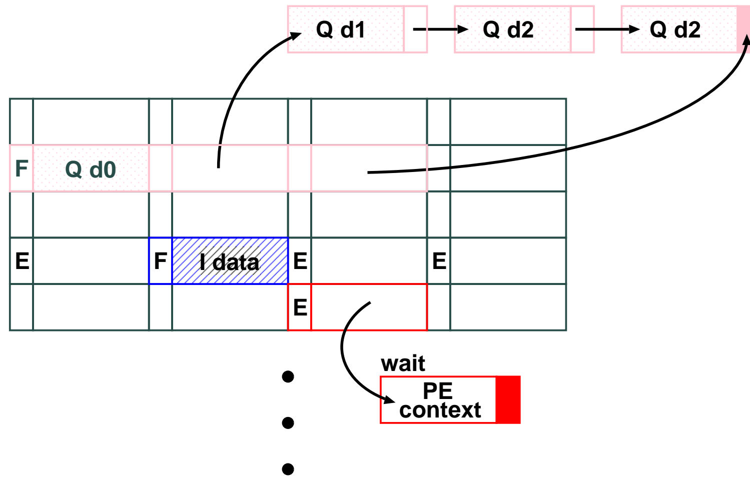


図 5.3 高機能な分散共有記憶による同期機構の概念図 (1)

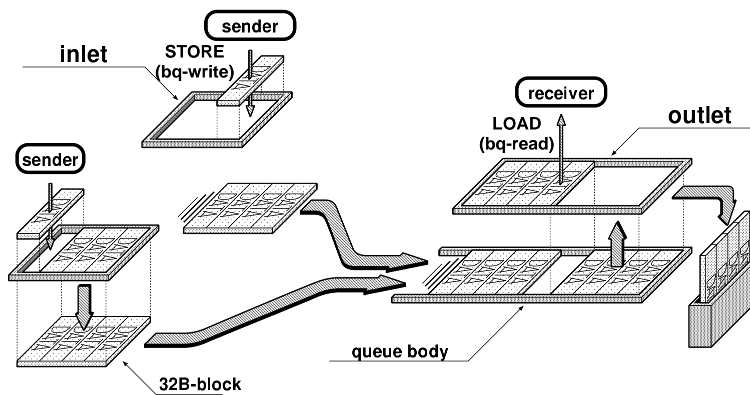


図 5.4 高機能な分散共有記憶による同期機構の概念図 (2)

(2) JUMP-1 では多様なキャッシュ一貫性プロトコルをサポートしており，必要とされるパケットの付加情報が多く，制御も複雑となる．よって，パケットに関する情報の管理やパケット制御などは柔軟に管理する必要がある．ハードウェア量から考えると，hardwired-logic で実現するのではなく Core プロセッサで実現した方が望ましい．

(3) 分散共有記憶上で I-structure や Q-structure といった同期機構を実現する必要がある．よって，パケットのヘッダ部とデータ部とを一体にした処理をせねばならない．しかし，68 bit 幅の packet buffer を直に扱うことはハードウェア量から考えると現実的でない．よって，これをうまく解決するようなアーキテクチャが必要となる．

これらの要件を満たすべく，MBP-light では次の設計方針を採用した．

(1) 応答パケットの生成や収集は専用の hardwired-logic で処理する方が望ましい．よって，CPU 側とネットワーク側のそれぞれに設けることとする．

(2) 柔軟性やハードウェア量を考慮した結果，RISC 型の Core プロセッサを内蔵する．

(3) packet buffer を特殊なレジスタとして扱う *buffer-register architecture* を用いる．これにより，少なくとも packet buffer を内部メモリに格納する一般の RISC プロセッサよりも性能向上を図る．

5.3　MBP-light の構成

5.3.1　MBP-light の全体構成

上記の方針に基づく MBP-light の全体構成を図 5.5 に示す．MBP-light は専用 hardwired-logic である RDT Interface と Main Memory Controller(MMC)，Core プロセッサである MBP Core から構成される．RDT Interface は主に RDT router chip の制御を行い，MMC はクラスタバスやクラスタメモリの制御を行う．単純なプロトコル処理はこの 2 つのモジュールによって処理される．もし処理できない場合には，割込みにより MBP Core を起動し，複雑なプロトコルをソフトウェアにより柔軟に制御する．以下にそれぞれのモジュールについて詳細に説明する．

5.3.2　RDT Interface

5.3.2.1　RDT Interface の構成

RDT Interface の構成を図 5.6 に示す．RDT Interface はパケットの送受信を行う Packet Handler，応答パケットの収集を行う Ack Collector，応答パケットの自動生成を行う Ack Generator，Send/Receive Unit および MBP Core と RDT Interface 間の送受信に用いられる packet buffer から構成される．RDT router chip とのパケットの送受信は Send/Receive Unit によって行われる．また，送受信するパケットの制御は他の 3 つのモジュールがすべて独立に動作する．

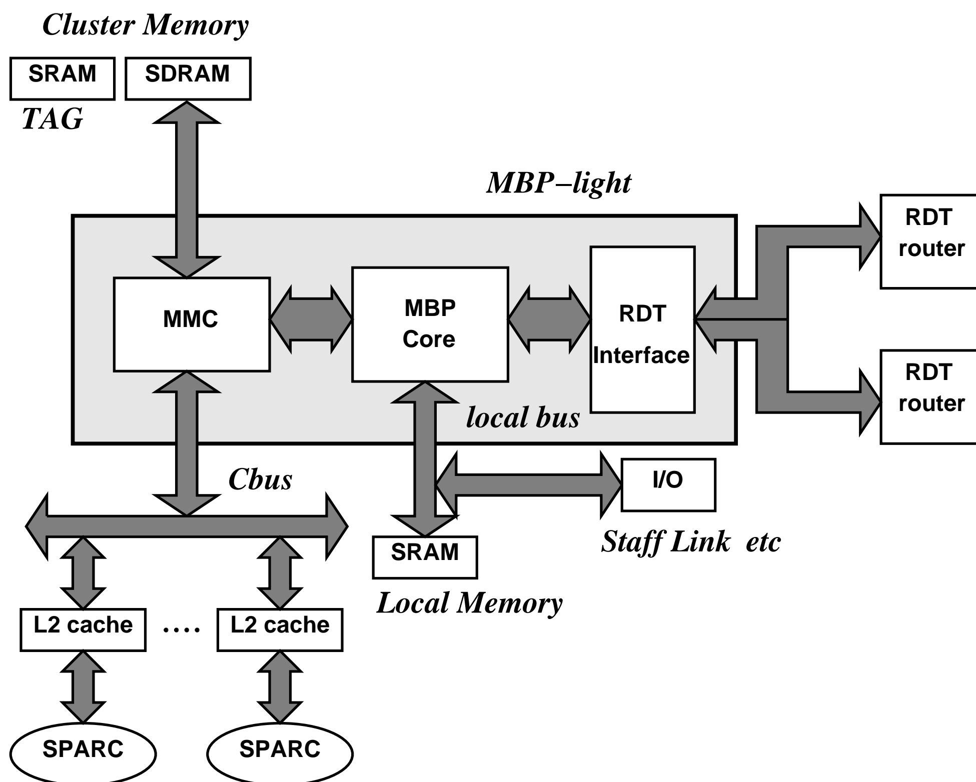


図 5.5 MBP-light の全体構成

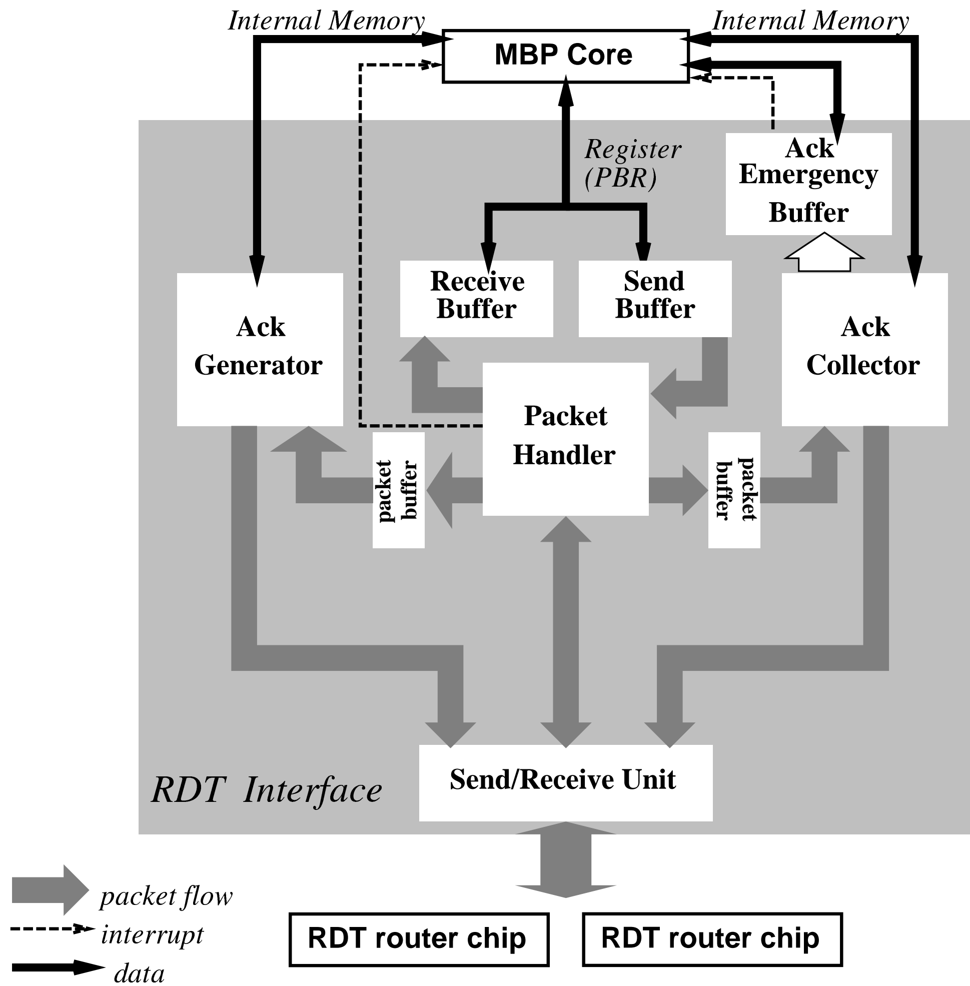


図 5.6 RDT Interface の構成

5.3.2.2 Packet Buffer Register(PBR)

packet buffer は 68 bit 幅の Packet Buffer Register (PBR) からなる．packet buffer として RDT パケットを 1 パケット分格納するために PBR を 8 個必要する．そして，それぞれ 3 パケット分の PBR が送信用と受信用の 2 系統の cyclic buffer を構成する．以降，特に明記しない限り，送受信用バッファとは MBP Core のそれを意味する*．PBR の構成図を図 5.7 に示す．RDT router 側と MBP Core 側の両方にポインタが存在し，両側のポインタが同じ packet buffer を指していた場合は空であることになる．この構成をとることにより，MBP Core でパケットを生成かつ処理する操作と，Send/Receive Unit がパケットを送受信する操作を互いに別々の buffer に対しての操作として扱うことができる．

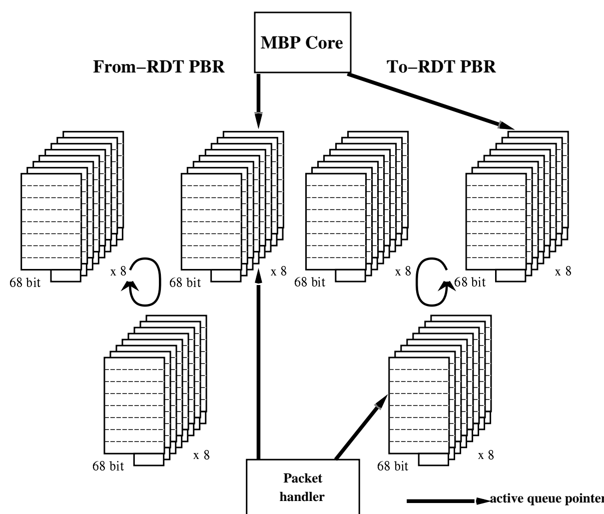


図 5.7 RDT 送受信用バッファの構成

5.3.2.3 Send/Receive Unit

Send/Receive Unit は MBP-light と RDT router 間のパケットの送受信を行うハードウェアである．主な機能は，RDT router と MBP-light との間のパケット送受信時のハンドシェークやアービトレーション，Ack Collector，Packet Handler，Ack Generator で生成されるパケット間の送信時のアービトレーションである．

RDT router と MBP-light 間の通信におけるハンドシェークは request 信号である Mreq (MBP request) と Rreq (RDT request)，ready 信号である Mready (MBP ready) と Rready (RDT ready)，および応答パケットを扱う際の rqex (request exclusive) 信号によって行われる．RDT と MBP-light 間でパケット送受信時に競合が起きた際には，初期化後の最初の送信では MBP-light の送信が優先権をもつ．その後はパケット転送のたびに優先権が移動し，次に転送要求が競合した際には，その前に送信を行った側が送信要求を取り下げる．通常のマルチキャストパケット転送時の送受信の様子を図 5.8 に示す．この転送サイ

*Ack Collector や Ack Generator にも送信バッファは存在する．

クルでは RDT router から MBP-light に転送しているが，逆の転送も同様となる．

パケット受信時では，最後のフリットを受け取る際にリクエスト信号を 1 クロックだけアサートすることで，正しく受信したことを送信側に知らせる．ここで，応答パケット転送の様子を図 5.9 に示す．マルチキャストの頻度が高くなると，応答パケット収集の際にルータと MBP-light 間がボトルネックになる [47]．よって，RDT router から MBP-light に対する応答パケットの転送を優先させる必要がある．まず，rqex 信号をアサートすることで応答パケットの転送であることを示す．そして，RDT router からの応答パケットの転送に対しては，MBP-light からのパケット送信要求との競合が起こった場合に必ず優先することを保証する．

パケットを送受信する際には，マルチキャストパケットであればパケット長が可変なため，2 フリット目の Length field を調べる．そして，その長さに 1 を加えたもの (parity flit 分) をパケット長とする．送信する際には，その長さで RDT router に対してパケットを送出する．また，送信する際には最終フリットに合わせて，request 信号をアサートする．その他のパケットであればパケット長は固定であるため，それぞれのパケット長に対応して送受信の制御を行う．

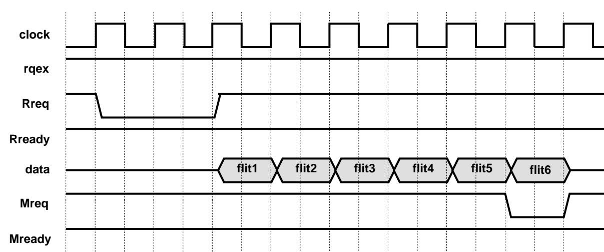


図 5.8 RDT と MBP-light 間のマルチキャストパケットの転送の様子

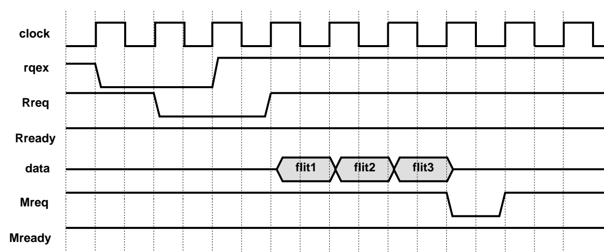


図 5.9 RDT と MBP-light 間の応答パケットの転送の様子

5.3.2.4　Packet Handler

Packet Handler は送信部と受信部に分けられる．以下に，送信部と受信部の機能について説明する．

(1) Packet Handler 送信部

Packet Handler の送信部では，MBP Core で生成されたパケットに 1 フリット分の parity flit を付加する．そして，最小 7 フリット ∼ 最大 16 フリットのパケットとして RDT router に送信する．これに対し，応答パケットには parity flit は存在せず，3 フリットのまま送信される．MBP Core はパケットを送信用バッファに格納する．そして，送信命令を実行するのを受けて，RDT Interface はパケットの送信を開始する．送信用バッファは 3 パケット分のサイクリックバッファを構成している．よって，Packet Handler が送信処理を行っている間でも，MBP Core は別の送信用バッファにパケットを生成することができる．

1 つの送信用バッファは，68 bit の PBR 8 本で構成される．フリット番号との対応を図 5.10 に示す．RDT パケットの先頭 4 フリットは 2 つの RDT router に共通であるため，図 5.10 に示すように，4 フリット分を確保する．そして，Packet Handler 送信部で router 2 個分に増幅して送信することになる．図中の PBR の offset 8 の領域は 4 bit ある．この 4 bit を各フリットに分割して送信し，各フリットに対して図 5.10 に示すように 16 bit 目に割り当てる．図では 1 ∼ 4 フリット目しか示していないが，他の PBR に対するタグは上位ビットからそれぞれ第 i U，第 i L，第 $(i+1)$ U，第 $(i+1)$L フリットに 1 bit ずつ図 5.10 に示す方法で付加する．

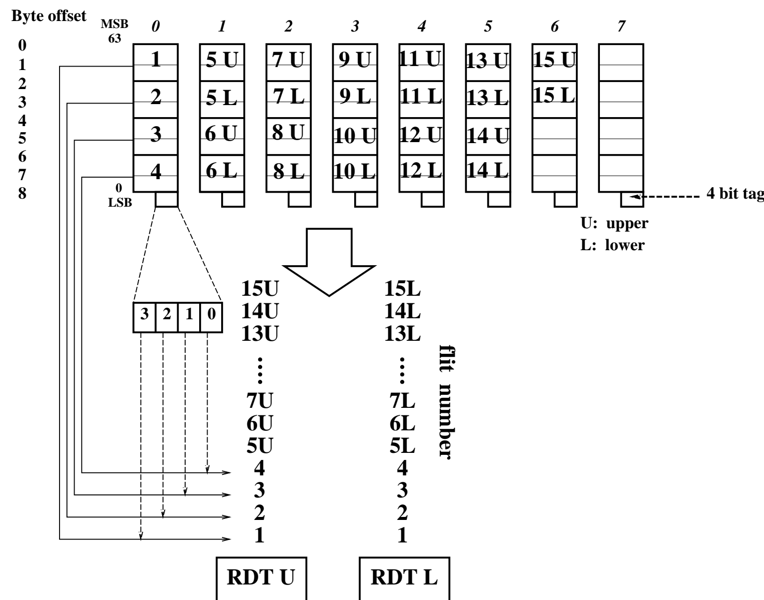


図 5.10 送信用バッファから RDT router へのパケット送信

(2) Packet Handler 受信部

Packet Handler 受信部では，RDT router から受信したマルチキャストパケットおよび到着ノードである応答パケットを RDT 受信用バッファに格納する．マルチキャストパケットの受信時の各フリットと受信用バッファの対応を図 5.11 に示す．受信時は送信時とは逆に各フリットの 16 bit 目を集めて，対応する PBR のオフセット 8 の領域に格納する．受信用バッファは，送信用バッファと同様に 3 パケット分の

cyclic buffer を構成する．また，送信部と同様に先頭4フリットが共通なものであるため，17 bit×4 フリット分のみ受信用バッファに格納する．またパリティは受信中に検査し，誤りがあればMBP Core に割込みをかけることも可能である．

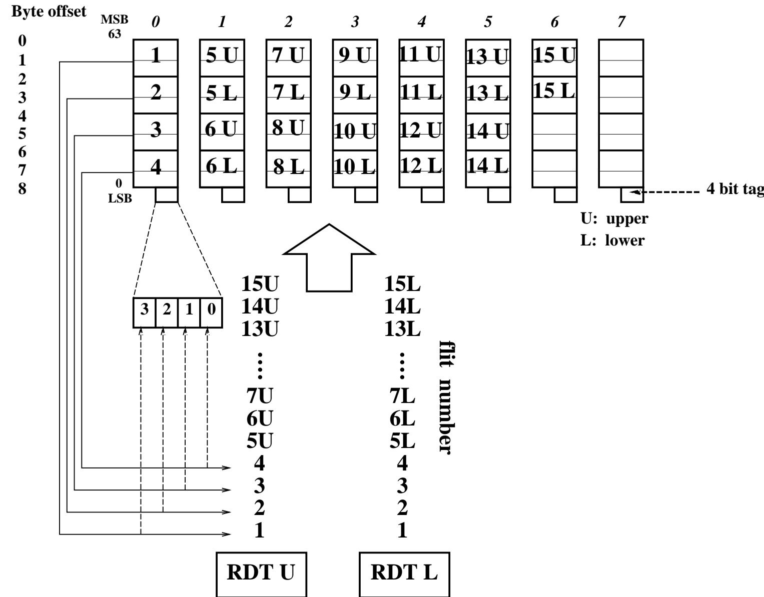


図 5.11 RDT router から受信用バッファへのパケット受信

5.3.2.5　Ack Collector

- Ack Collector の構成

  マルチキャストパケットを発生したノードはその終了の確認のために，応答パケットを収集しなければならない．RDT router chip は基本トーラス (rank 0) 上の応答パケットを1エントリ分だけ収集する機能をもつ．しかし，router chip 内の収集用エントリが溢れた場合の基本トーラスでの収集と上位トーラスでの収集は，MBP-light 中の RDT Interface の役目となる．

  その目的を果たすため，Ack Collector は基本トーラス用と上位 rank 用に Ack Cache と呼ばれるキャッシュを保持している．それぞれは 128 エントリと 2 エントリからなる表を持ち，全体として特殊な 2 way で set associative なキャッシュとして機能する．これは，このような構成をとることにより，ハードウェア量の増加を抑えるためである．そして，このキャッシュを用いることで，それぞれの階層における応答パケットの収集を完全にハードウェア制御で行う．また，Ack Collector は Ack Emergency Buffer(Ack 緊急バッファ) と呼ばれる緊急用のバッファをもっており，これを通じて MBP Core に Ack Collector から緊急パケットを送ることができる．Ack Cache の構成を図 5.12 に示す．Ack Cache のエントリの種類とその内容について以下に述べる．

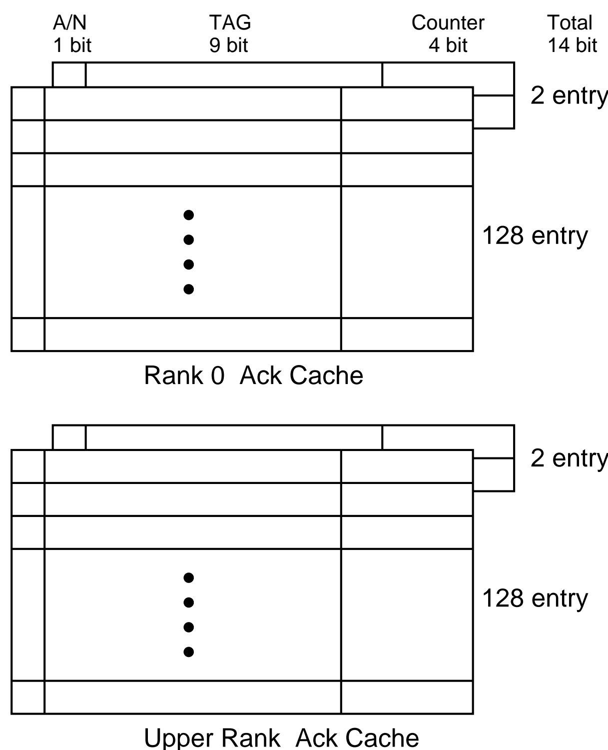


図 5.12 Ack Cache の構成

A/N	上位階層に返すパケットを Ack にするか Nack にするかを示す．1 であれば Ack を表す．1 つでも Ack を受信すると Ack を上位階層に返すことになる．
Tag	図 5.13 の tag に相当する．このフィールドを用いて一致検出を行う．
Counter	収集すべき応答パケットの数を表す．0 であれば空きエントリである．

- 応答パケット収集用エントリの作成

ここで応答パケットの収集用エントリの作成方法について述べる．応答パケット収集用のエントリは基本的にマルチキャストパケットを送信するときに作成される．ブロードキャストに対応する収集では，ブロードキャストを行ったノードへ最初に到着した応答パケットを受信するときに作成される．このエントリは RDT router や Ack Collector，MBP Core の順にいずれかで作成される．ただし，ブロードキャストの場合のエントリは RDT router には作成されない．

エントリ作成は，マルチキャストパケットの Eack field(Ack 収集要求ビット) が 1 の場合，または応答パケットが到着した際に対応する収集用エントリが存在しなかった場合に行われる．また，登録を行うノードはパケットが実際にマルチキャストを起こすノード (木構造の根または節にあたるノード) である．登録の際にはマルチキャストパケットおよび応答パケットに付加されている Compressed Address と呼ばれる

ネットワーク内で唯一の鍵を用いてエントリを作成する．この Compressed Address の構成を図 5.13 に示す．Compressed Address は 16 bit で構成され，10 bit の node ID と 5 bit の version 番号，1 bit の RDT 拡張ビットで構成される．下位 8 bit が 256 クラスタ構成での node ID となり， 拡張 node ID の 2 bit を付加して 1024 クラスタ構成での node ID とする．このように 8 bit と 2 bit に分かれている理由は 5.3.2.6 節の Ackmap Cache に関する記述を参照されたい．また，version 番号は，マルチキャストの開始ノード (木構造の根にあたるノード) がそのノードから現在発行されているトランザクションを区別するために付加する番号である．これにより，各ノードはそれぞれ 5 bit 分，つまり 32 個の応答パケットの収集を伴うマルチキャストパケットを同時に発行することができる．そして，rank 2 以上のノードを Root Rank とした応答パケットの収集を伴うマルチキャストを行う際には，RDT 拡張ビットを 1 に設定し，さらに 4 と 5 フリットの両方に Compressed Address を付加する必要がある．それ以降のフリットは 1 フリット分ずつ後ろに繰り下がる．

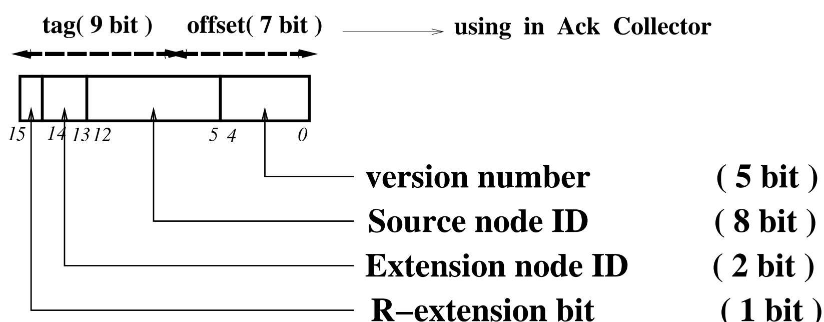


図 5.13 Compressed Address の構成

- マルチキャスト収集用エントリの作成

(1) RDT router

Eack field(付録 A 参照) が 1 であるようなマルチキャストを行った際には，どのランクの Ack 収集をする必要があるかどうかを MBP-light に伝える．そのパケットは受信パケットの Message for MBP のフィールドを適切に設定したものである．

(2) Ack Collector

RDT Interface では図 5.13 に示される 9 bit のタグと 7 bit のオフセットを用いて Ack Collector 中の基本トーラス (rank 0) 用か上位 rank 用の Ack Cache 中にエントリを保持する．ただし，これは 128 エントリをもつ way の場合であって，2 エントリをもつ way の場合にはオフセットは下位 1 bit に読み代える必要がある．エントリは RDT router とは異なり，パケット数に加えて Compressed Address の tag(9 bit) から構成される．エントリ作成の際には，そのオフセット

でのエントリのパケット数を検査する．パケット数が 0 であれば，そのエントリは収集済みであり，新たなエントリを作成する．エントリが作成できなかった場合は，Ack Emergency Buffer を用いて MBP Core 自身が収集する必要があることを伝える．しかし，該当オフセットに対応する応答パケットが後続として到着し，エントリに空きがあった場合は，ブロードキャストに対する応答パケットであると誤認されてしまう．よって，エントリが作成できなかった場合には，Ack Cache の該当するオフセットをロックする．このロックにより broadcast entry の作成を禁止することができる．このロックは MBP Core が設定および解除することができる．

(3) MBP Core

MBP Core で収集する場合にも Ack Collector と同様に収集用の表を何らかのメモリ上に確保する必要がある．MBP Core で収集中のエントリにロックをかけていない場合は，応答パケットが Ack Collector から MBP Core に渡されない．可能性がある．

- broadcast entry の作成

ブロードキャストを行うマルチキャストパケット対して，収集用のエントリの作成は送信時に行わない．そのために，マルチキャスト時に行っていたような MBP-light に対する登録用のパケットも送信されない．ブロードキャスト時の収集用エントリは最初に到着した応答パケットによって作成される．最初に到着したブロードキャストに対する応答パケットのエントリのパケット数は 0 であり，エントリの作成時に対応するエントリの未収集パケットの数を 7 とする．もしエントリがロックされていた場合には，その応答パケットは MBP Core で収集中の可能性がある．よって，Ack Emergency Buffer を通じて MBP Core に転送される．ブロードキャストに対応した応答パケットは RDT router にエントリを作成することはできない．また，その応答パケットのたどりつくべき木構造の節または根に対応したノードでは，必ず RDT router から MBP-light に転送される．mutlicast entry の作成時と同様に，エントリの登録に失敗した場合にはロックをかける必要がある．そして，後続の応答パケットが MBP Core に転送されるようにする．

- 応答パケットの収集手法

応答パケットを収集するのは RDT router か MBP-light のどちらかである．応答パケットの S1 と S0 field を検査することで，応答パケットの状態を判別することができる．S1 と S0 は応答パケットに対しルータ間でパケット制御に用いる状態で，MBP-light から送信する際には 00 で送信する必要がある．

- 00: rank 0 で収集が必要な応答パケットであることを示す (状態 A)．
- 01: 上位 rank で収集が必要な応答パケットであることを示す (状態 B)．
- 10: RDT router から MBP-light に対して送信するパケットにしか使用されず，RDT router で rank 0 の収集が終わっているパケットということを示す (状態 B)．

- 11: root で収集済みのパケットで Root Rank で示す rank の router から Dest Rank で示される rank にルーティングされることを示す (状態 C).

Ack Collector での応答パケット収集のアルゴリズムを以下に示す.

(1) パケットが状態 C であれば，そのノードが宛先であるため，収集は終了する．そして，パケットを Packet Handler に受け渡す (実際には一度 RDT router を介する).

(2) パケットが状態 A ならば基本トーラス，状態 B ならば上位 rank の Ack Cache を検索する．Compressed Address のオフセットに対応するエントリが空であるかどうか，そのタグと Compressed Address のタグが一致するかどうかを検査する．検索の結果，

  (a) エントリは空である (かつ，タグが一致しない) 場合は，ブロードキャストに対して最初に到着した応答パケットと判断する．そして，カウンタを 7 にしてエントリを作成する.

  (b) エントリが空でなく，タグも一致しない場合は，該当エントリにロックをかける．そして，パケットを Ack Emergency Buffer を通じて MBP Core に転送する.

  (c) エントリが空でなくタグが一致する場合は，該当エントリのカウンタをデクリメントする．その結果 0 となれば，(3) の処理を行う.

(3) パケットが状態 A でその Root Rank が 0 ならば，それはローカルマルチキャストに対する応答パケットと認識される．よって，(2) で収集済みの応答パケットの宛先はこのノードとなり，収集処理は終了する．そして，パケットを Packet Handler に受け渡す.

(4) パケット中の M0 → M1 → M2 → M3 の順に従ってパケットを転送する.

(5) パケットが状態 B であり，M0 が 000(Myself) であれば，このノードで rank 1 での収集が必要なパケットとなる．そして (1) に戻り，上位 rank の収集処理を行う．そうでなければ，上位 rank で収集すべきパケットであるので，S1 と S0 を 01(状態 B) として RDT router に転送する.

(6) パケットの Root Rank とこのノードの rank が一致すれば，最終宛先ノードに対する処理が必要となる．よって，S1 と S0 を 11(状態 C) として RDT router に転送する.

(7) パケットの Dest Rank とこのノードの rank が一致すれば，収集処理は終了する．そして，パケットを Packet Handler に受け渡す.

- Ack Emergency Buffer

  Ack Emergency Buffer は Ack Collector から MBP Core に対して用いられる緊急用のパケットバッファである．Ack Emergency Buffer は緊急性を要する処理に用いられる．Ack Emergency Buffer が使用されるのは次の場合である.

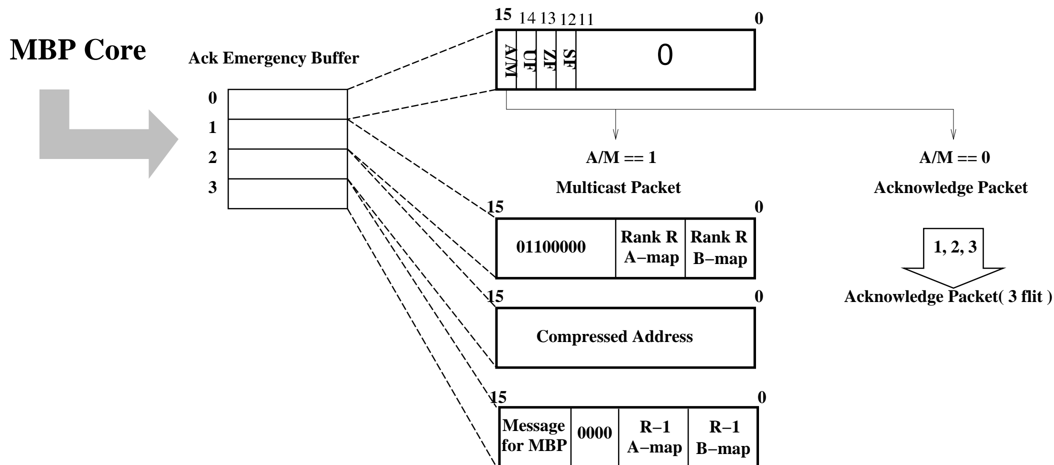


図 5.14 Ack Emergency Buffer の構成と形式

– マルチキャストパケットを受信し，3 フリット目の Message for MBP の Ack Cache 登録要求ビット (R0R と URR) が 1 であると，Ack Cache にエントリを作成しようとする．しかし，エントリに空きがなかったため，その登録用パケットを MBP Core に受け渡す．
– 応答パケットを受信し，該当するエントリを検索するが存在せず，しかも空きエントリもないため，broadcast entry の作成もできない．結果，応答パケットを MBP Core に受け渡す．

Ack Emergency Buffer は MBP Core から内部メモリとして参照することができ，その構成を図 5.14 に示す．ただし，図中の UF とは上位 rank 用の Ack Cache に空きがなかったことを，ZF は rank 0 用の Ack Cache に空きがなかったことを，SF は Ack Collector 内の送信バッファが満杯であったことを意味する．

このバッファの容量は小さいため，Ack Collector が書き込もうとし，なおかつ MBP Core で使用中の場合には，これ以上パケットを受信することは許されない．すると，Ack Collector は Send/Receive Unit に指示して，RDT router に対して ready 信号をネゲートする．MBP Core で Ack Emergency Buffer の clear 命令が実行された場合に Ack Emergency Buffer の内容は消去される．

5.3.2.6　Ack Generator

マルチキャストに対応した応答パケットの収集は，少しでも短時間で終了した方がよい．そのために，応答パケット収集が必要なマルチキャストパケットを受け取ったノードは速やかに応答パケットを生成し，送信する必要がある．Ack Generator では Net Cache と Ackmap Cache の 2 つのキャッシュを用いて，ハードウェアにより応答パケットの自動生成機能を実現する．

- Net Cache

Ack Generator では応答パケットの自動生成のために，512 エントリの Net Cache を 2 組備えている．クラスタ内のキャッシュラインの状態に従って，Net Cache の状態を作成しておく．そして，マルチキャストパケット到着時に Net Cache を調べ，マルチキャストパケット中のアドレスがヒットした場合，登録しておいた状態を応答パケットに付加することができる．ただし，この Net Cache に対するエントリの作成や追出しなどの管理は MBP Core 上のソフトウェアで行う必要がある．Net Cache の構成を図 5.15 に示す．また，図 5.16 に RDT パケットの 5 フリット目と図 D.1 での `addr` field との関係および `addr` field と Net Cache のオフセット，key との関係を示す．そして，一致検出はこのオフセットに対するエントリの key が等しいかどうかで判別される．もし一致するならば，同じエントリ内のタグに従って処理を行うことになる．Net Cache のタグは 4 bit で構成されており，その種類と内容について以下に述べる．

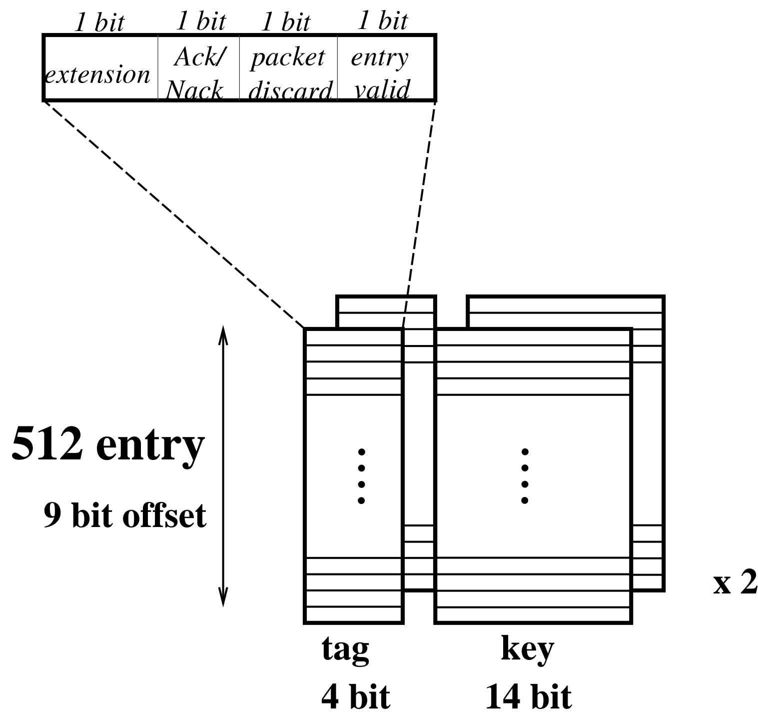


図 5.15 Net Cache の構成

Entry Valid	Net Cache のエントリが有効であることを示す．
Packet Discard	このビットを 0 にした場合，応答パケットを自動生成した場合に対応する要求パケットを破棄し，MBP Core に転送しないようにする．この機能により MBP Core が不要なパケットのために起動してしまうことを防ぐことができる．
Ack/Nack	自動生成する応答パケットが Ack か Nack かを示す．1 なら Ack となる．
Extension	ハードウェアで未定義なため，ソフトウェアで使用することができる．

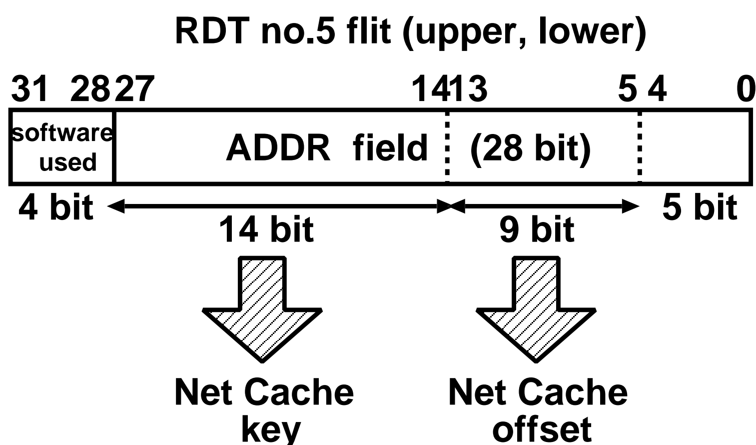


図 5.16 Net Cache とクラスタアドレスの関係

- Ackmap Cache

  Ack Generator では，Ackmap Cache を用意して，応答パケットに返送する情報として Ackmap(マルチキャストのどのリンクに対応した応答パケットかを示す情報) をもたせる．マルチキャスト到着時には，パケットの 4 フリット目にある図 5.13 での Compressed Address 中の source node ID をオフセットとして Ackmap Cache を検索する．そして，タグ中の extension node ID が一致するかどうかを調べる．もし一致して，かつ有効である場合には，そのエントリを読み出して応答パケット用の情報を設定する．Ackmap Cache の構成を図 5.17 に示す (フィールドの詳細は付録 A を参照されたい)．Ackmap Cache は 256 ノード分 (8 bit) のエントリを確保しており，256 ノード以内の場合には必ず Ackmap Cache にヒットする．ただし，256 ノード以上のシステム構成では，MBP Core のソフトウェアで Ackmap Cache の登録や追出しなどの管理を行う必要がある．

- エントリの作成手順

  応答パケットを自動生成するためには，Ackmap Cache のエントリが存在し，かつ Net Cache のエントリも存在することが必要である．どちらかが欠けている場合に

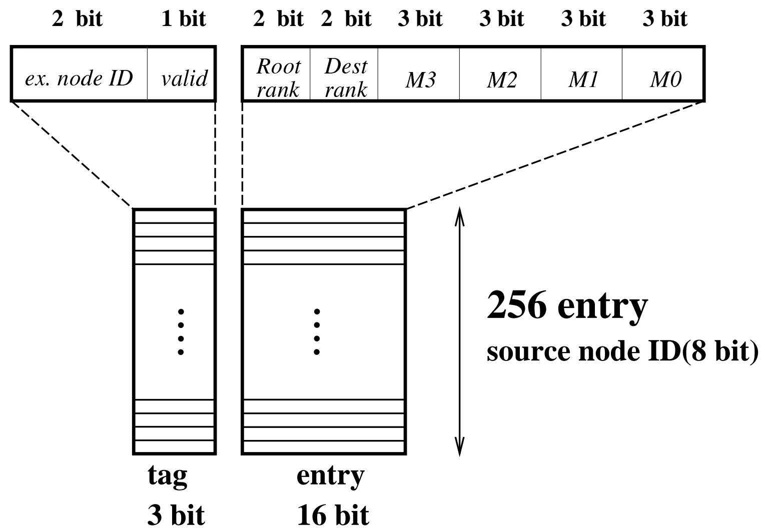


図 5.17 Ackmap Cache の構成

は応答パケットは生成されない．Net Cache と Ackmap Cache ともエントリの登録は MBP Core で行う必要がある．特に，Ackmap Cache では登録操作の最中にそのアドレスに対するマルチキャストパケットが到着することを想定する必要がある．よって，MBP Core ではエントリを設定してから，Valid bit を 1 にしなければならない．この順序を逆にすると，誤った応答パケットを自動生成してしまう可能性がある．最後に，応答パケットの自動生成の際に応答パケットの 3 フリット目中の body がどのように構成されるかを図 5.18 に示す．

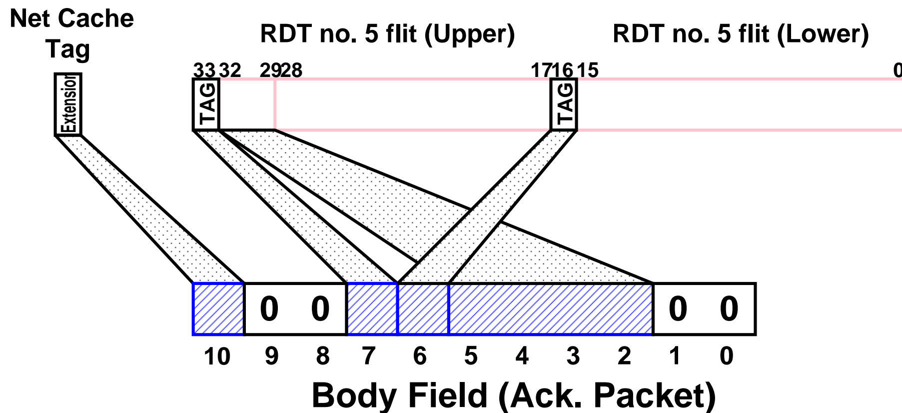


図 5.18 Net Cache による応答パケットの body の構成

5.3.2.7　shootdown と setup

RDT router は router 内の channel buffer に存在するパケットの状態を MBP-light に報告することが可能となっている．これを shootdown 機能という．shootdown は緊急時やデ

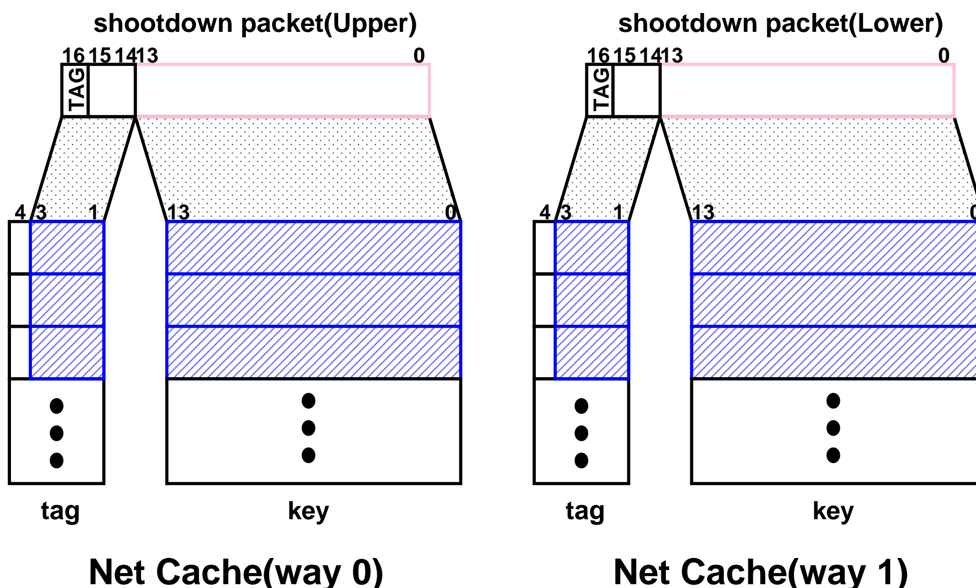


図 5.19 shoot down 時の Net Cache の状態

バッグ時，operating system のタスク切換時に使用することを想定している．shootdown は基本的に MBP Core からの要求によって引き起こされる．ただし，RDT router の設定によってはエラー発生時に RDT router が動作させることもある．さて，shootdown を起こすためには command packet の mode を示す M1 と M0 を 01 (shootdown mode) に設定して，RDT router に送信すればよい．RDT router は shootdown を要求する command packet を受け取ると，パケットが存在する場合には shootdown packet を送信する．また，router 内の Ack Cache にエントリが存在する場合には，Ack Cache shootdown packet を送信する．そして，RDT Inteface も shootdown mode になる．shootdown mode になると，shootdown packet や Ack Cache shootdown packet を RDT router は連続して MBP-light に送信する．その際，RDT Interface は受け取ったパケットをそのまま Net Cache にオフセット 0 から順番に格納する．また，shootdown 時にはそれら以外のパケットを MBP-light と RDT router 間で送受信することはできない．そして，shootdown 終了の際に特別の status report packet が RDT router から送信される．shootdown packet 受信時の Net Cache へのパケット格納の形式を図 5.19 に示す．ただし，パケットがどこまで格納されたかは MBP Core のソフトウェアにより検査されなければならない．

RDT router に shootdown をさせた後，RDT router を shootdown を起こす前の状態に戻したい場合には，RDT router の setup を行う必要がある．ただし，RDT router の Ack Cache を shootdown の前の状態に設定することはできないため，Ack Cache shootdown packet に示されるカウンタの値と Compressed Address のエントリを MBP-light で管理する必要がある．setup を行う際には，shootdown によって届いた shootdown packet をそのまま RDT router に転送する必要がある．ただし，Ack Cache shootdown packet と status report packet は送信してはならない．そして，setup を開始するには command packet の M1 と M0 を 10(setup mode) に設定して，RDT router に送信する必要がある．その後，

必要な数だけ setup packet (shootdown されてきたパケット) を送信する．再び通常の状態で RDT router を動かすには，command packet の M1 と M0 を 00(Normal Mode) にして送信すればよい．

5.3.2.8　MBP Core Interface

- 割込み

  RDT Interface から MBP Core に対して割込みをかけることができる．割込みに関しては 5.3.4.5 節で詳しく述べる．

- command status register

  MBP Core から RDT Interface に対して制御を行うレジスタ．その他にも状態を知ることができる．レジスタの構成を図 5.20 に示し，以下に説明する．

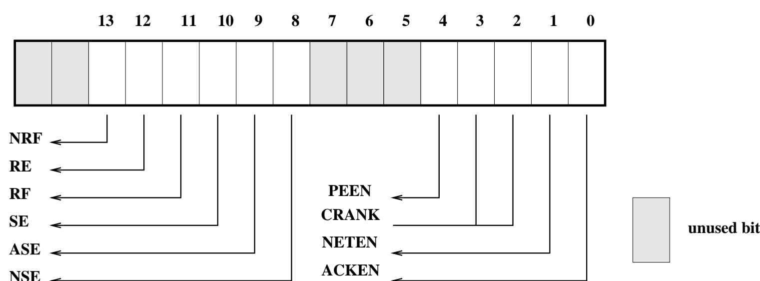


図 5.20 RDT Interface command/status register の構成

NRF(Net Cache Receive Full)	R	Net Cache 内の受信バッファが満杯になった
RE(Receive Buffer Empty)	R	受信バッファが空
RF(Receive Buffer Full)	R	受信バッファが満杯
SE(Send Buffer Empty)	R	送信バッファが空
SF(Send Buffer Full)	R	送信バッファが満杯
ASE(Ack Cache Send Buffer Empty)	R	Ack Cache 内の送信バッファが空
NSE(Net Cache Send Buffer Empty)	R	Net Cache 内の送信バッファが空
PEEN(Parity Error Enable)	W	parity 誤りの検査を行うかどうか指定
CRANK(Current Rank)	W	このノードの rank を設定 (2 bit)
NETEN(Net Cache Enable)	W	Ack Generator の機能を利用
ACKEN(Ack Cache Enable)	W	Ack Collector の機能を利用

5.3.3　MMC (Main Memory Controller)

5.3.3.1　内部構成

MMC は，MBP Core やクラスタバスの両者からの要求を高速に処理する hardwired-logic である．MMC の構成を図 5.21 に示す．MMC は，主に東京大学の佐藤 充氏によっ

て設計された [48].

MMC は大きく分けて，SBUSCTL(cluSter BUS ConTroLler) と入出力バッファ(I/O buffer)，MTC(Memory Timing Controller) の 3 つから構成される．また，MTC は主に次のものを内部にもつ.

- Address Controller
- SDRAM Controller
- SRAM Controller
- Refresh Controller

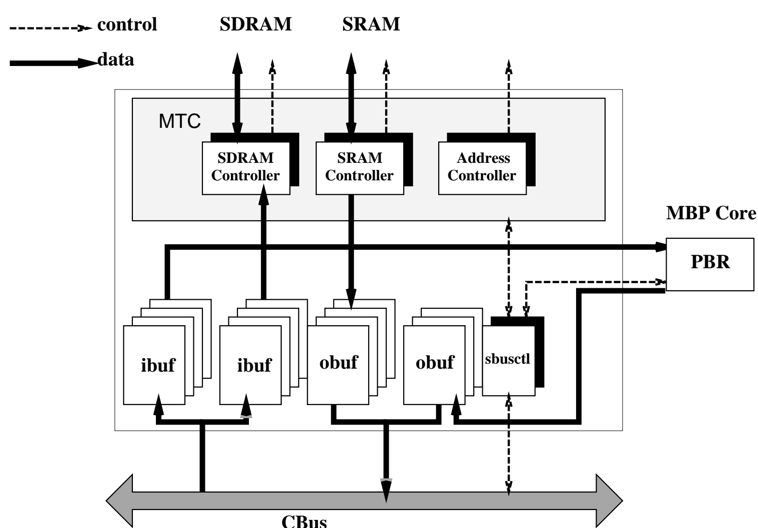


図 5.21 MMC の内部構成

MMC は，種々の内部レジスタを MBP Core に提供しており，詳細は付録 B を参照されたい.

5.3.3.2　メモリ構成

JUMP-1 のメモリ構成は図 5.22 のようにタグに用いられる 512Kbyte の SRAM と，データに用いられる 16Mbyte の SDRAM，タグに用いられる 512Kbyte の SDRAM，ECC (Error Correcting Code) に用いられる 512Kbyte の SDRAM からなる．データとタグ，ECC 用の SDRAM はインターリーブされている.

また，クラスタメモリのフォーマットは図 5.23 のようになっている．まず，SRAM のタグについて説明する．SRAM tag はキャッシュライン単位に設けられており，それぞれのビットの意味は次のとおりである.

- Full/Empty

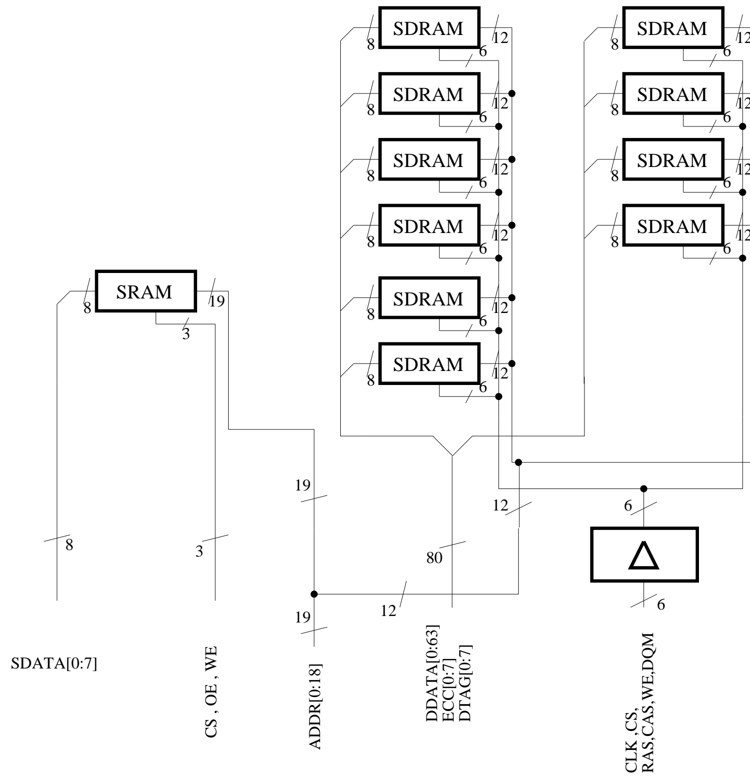


図 5.22 JUMP-1 のメモリ構成

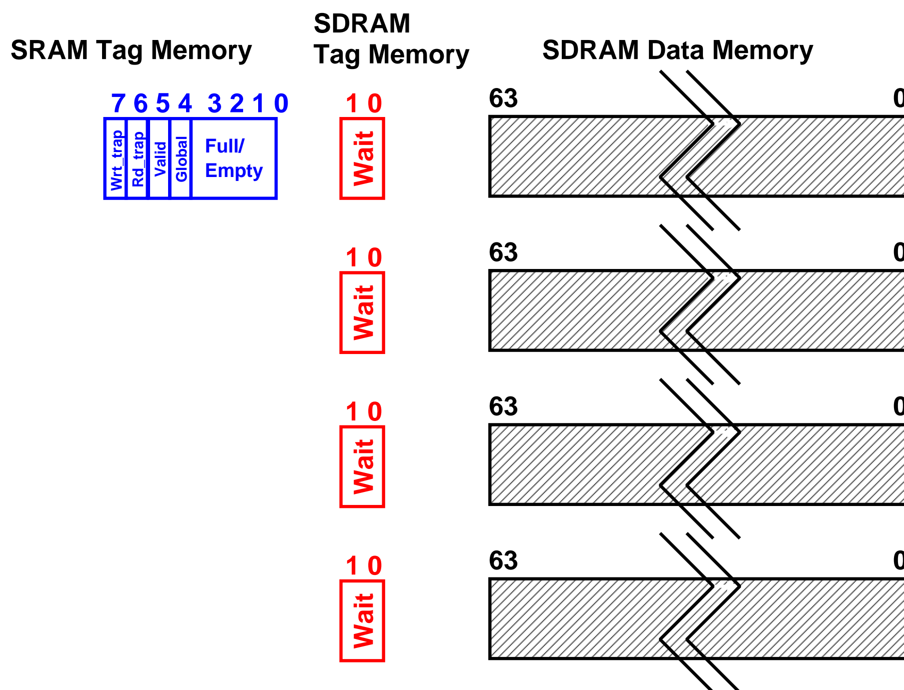


図 5.23 クラスタメモリのフォーマット

block 内の各 double word の full/empty を表す．0 bit 目が block 中最も小さいアドレスをもつ double word，3 bit 目が block 中最も大きいアドレスをもつ double word の状態を表す．この極性はパケット中の fepol field によって決定される．

- Global

  当該 block がクラスタ間で共有されていることを示す．このときにメモリ上のデータが有効であることも保証される．

- Valid

  当該 block がクラスタ内で有効であることを示す．このときにメモリ上のデータが有効でないこともありうる．

- Rd_trap と Wrt_trap

  当該 block への trap 用ビット．要求パケットに応じて MBP-light へ割込みをかける．詳細は表 5.2 を参照されたい．

次に，SDRAM のタグについて説明する．SDRAM tag は各 double word 毎に設けられており，Wait は当該 double word の待合わせ状況を保持する同期タグとなっている．その値によって以下の状態をもつ．

- ‘00’: 非 wait 状態
- ‘01’: read wait 状態
- ‘10’: write wait 状態
- ‘11’: read 及び write wait 状態

それぞれメモリ上での I-structure や Q-structure といった同期機構の実現に利用することができる．

5.3.3.3　SBUSCTL(cluSter BUS ConTroLler)

SBUSCTL は MMC の中でクラスタバスを制御するユニットである．クラスタバスから MMC に向けて送信されたパケットは入力バッファに格納される．入力バッファは request 系パケット用に 4 つ，reply 系パケット用に 4 つ用意されている．さらに，MBP Core からのメモリ参照要求を受けとるために 1 つ存在している．実際にはそれぞれの系統で 2 個のパケットを受信すると，MMC はバスに busy 信号を送出するので，バッファが全て使われることはほとんどありえない．busy 信号は MBP Core によって強制的にアサートすることも可能となっている．

また，出力バッファは MMC がハードウェア的に生成するパケット用に 4 つ，MBP Core から送信する用に request 系と reply 系の 2 つが用意されている．ただ，MBP Core が同時にパケットを格納することはありえない．MBP Core が送信を行う場合に，バスの信号線 `X_BREQ` が ‘01’ である状態が 2 クロック続いた後に `X_BGRT` を与えられると，MMC が誤動作をすることが分かっている．これを解決するため，`X_BRETRY_I` の代わりに各 CC chip

や MMC 自身からの X_BRETRY_O 信号の論理和を直接とって，先に延べた条件が満たされたときにこの信号を与えなければならない．さらに，パケット送信中にバスの grant 線がアクティブになっていると，SBUSCTL が誤動作するというバグが存在することが分かっている．これを解決するため，MMC の X_ASTB_O を見て MMC がパケットの送信を始めたら，5 クロック (最長パケットを送出し終えるまで) MMC に対する X_BGRT をカットするという方法を用いている．また，MMC が処理することができず，MBP Core へ割込みを引き起こすパケットはその要因をクリアするまでバッファを占有する．

さて，SBUSCTL でのパケット受信の様子を図 5.24 に簡単に示す．

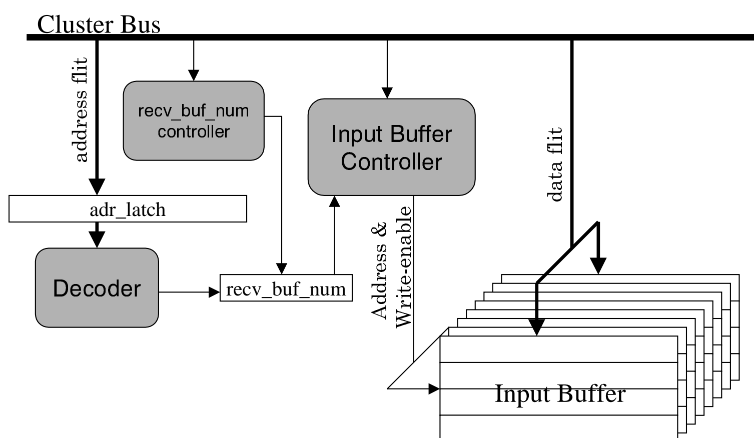


図 5.24 SBUSCTL でのパケット受信の様子

SBUSCTL は自分がパケットを受信できる状態にあるときには，常にアドレスフリットを adr_latch に受けとりつつ，request 信号を監視する．request を検出すると，アドレスフリットの内容に応じて即座にデータを入力するべきバッファ番号を設定する．このバッファ番号は recv_buf_num に蓄えられ，データ受信中その値をずっと保持する．データ受信部では recv_buf_num に設定された番号に従ってデータを入力バッファに格納する．

続いてパケット処理の様子を図 5.25 に簡単に示す．

クラスタバスから受信したパケット全てが MTC で処理されるわけではない．MBP Core を起動するパケットや単なる reply 系パケットも受信する可能性がある．したがって，受信したパケットをそのまま MTC に渡すわけにはいかない．また，クラスタバスでは request 系パケットのリトライが可能となっている．このリトライは一度送出したパケットが無効化されるので，内部バッファのクリアや内部要求の取消し等の複雑な手続きが必要となる．また，キャッシュがリプライを返すときも同様の手続きが必要である．そこで，これらの問題を解決するために，パケットの受信状態を記録するレジスタ (RECV_BUF_STATE) とメモリ参照要求状態を記録するレジスタ (MTC_REQ_STATE) を備えている．パケットを受信すると，まず RECV_BUF_STATE の当該ビットを設定する．そして，バッファを予約すると同時に retry controller を起動して，リトライまたはキャッシュのリプライの監視を行う．そして，retry controller によってリトライまたはキャッシュのリプライがないことが確定すると，パケットのデコード状態に応じて MTC_REQ_STATE が設定される．ここで初めて MTC に対してのメモリ参照要求が出される．もしリトライやキャッシュのリプライがあった場

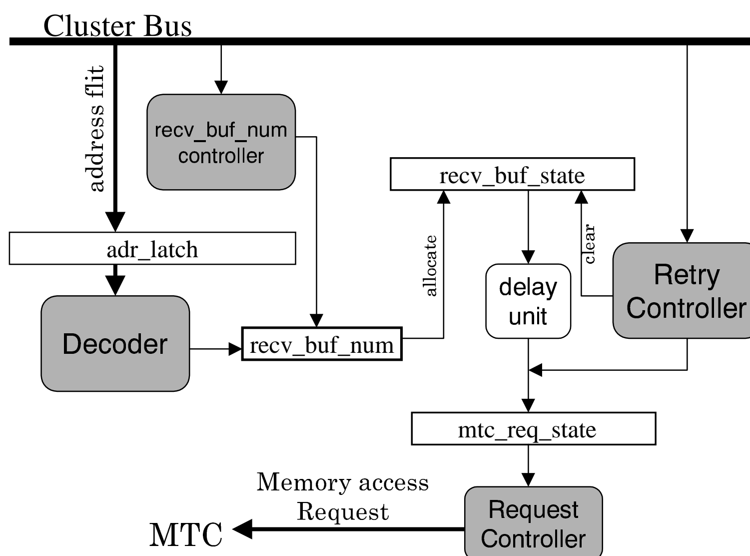


図 5.25 SBUSCTL でのパケット処理の様子

合には，retry controller によって `RECV_BUF_STATE` がクリアされ，入力バッファに蓄えられたパケットは破棄される．

SBUSCTL の機能を以下に示す．

- アドレス比較機能

  同一アドレスの逐次化を行うために用意された機能である．現在の入出力バッファにあるパケットと受信しようとしているパケットが同一アドレスとみなされると，受信しようとしてるパケットをリトライする．この同一アドレスとみなす範囲としての次に示す 5 種類をサポートしている．

  – 全て
  – block
  – page (4 KByte)
  – large page (256 KByte)
  – 16 MByte
  – なし

  標準の場合はブロックに設定されており，MBP Core から変更可能である

- 割込み機能

  以下の条件 (表 5.1 参照) を満たすパケットを受信したときに SBUSCTL は MBP Core へ割込みをかける．

  なお，SBUSCTL が受信して何もせず棄却するのは次の場合である．

  (1) nop

表 5.1 SBUSCTL の割込み条件一覧

packet 名	SBUSCTL Interrupt
read_req	did_MBP && Eaddr
n_c_rd_req	did_MBP && Eaddr
allread_req	did_MBP && Eaddr
swap_req	did_MBP && (Eaddr || !ccbl)
rdinv_req	did_MBP && Eaddr
update_via_home_req	did_MBP && (Eaddr || clsh || !ccbl)
write_req	did_MBP && (Eaddr || clsh || s_mblock)
writeback_reply	did_MBP && (Eaddr || !s_block)
update_direct_req	did_MBP && (Eaddr || clsh || !ccbl)
n_c_write_req	did_MBP && (Eaddr || clsh || s_mblock)
invalidate_via_home_req	did_MBP && (Eaddr || clsh)
injection_req	did_mbp
update_req	did_mbp
update_S_req	did_mbp
update_Q_req	did_mbp
invalidate_req	did_mbp
demap_req	did_mbp
interrupt_req	did_mbp
cflush_req	did_mbp
その他	did_MBP

(did_mbp || did_all) を did_MBP と略記．large_address || dummy || ow_ch を Eaddr と略記．
s_hoge は size field が hoge であるということを意味し，!はそのビットが 0 であることを意味する．

(2) `clsh` が 0 である update_via_home_req

(3) `clsh` が 0 である invalidate_via_home_req

(4) `clsh` が 0 である update_direct_req

(5) `did` が all である injection_req

(6) `did` が all である update_req

(7) `did` が all である update_S_req

(8) `did` が all である update_Q_req

(9) `did` が all である invalidate_req

(10) `did` が all である demap_req

(11) `did` が all である interrupt_req

(12) `did` が all である cflush_req

- lock register 機能

  トランザクションを排他的に処理するために用意された機能である．lock register は 4 個用意されており，それぞれに比較するパターンや比較する部分，一致した場合の動作を指定することができる．クラスタバスから MMC が受信するパケットが lock register で指定されたものと一致すると，そのパケットのリトライか MBP Core への割込みを生じさせることができる．ただし，送信元が MBP-light であるようなパケットはたとえ一致したとしてもその動作は無視される．

- PEND counter

  16 bit の PEND counter は 4 つの CPU のそれぞれに対して保持されている．そして，現在発行中の `ack` field が 1 であるパケットの数を保持する．SuperSPARC+では，PEND counter に保持されている値が 0 になるまで，STBAR 命令による待ち状態が続く．そして，PEND counter の値が 0 になると，後続の命令を実行しはじめる．MMC から各 SuperSPARC+には，X_PEND 信号と呼ばれる PEND counter が 0 かどうかを示す信号が発行されており，SuperSPARC+はこれを用いて PEND counter が 0 かどうかを知る．

  もし `ack` field が 1 であるパケットが割込みを起こした場合，トランザクションの処理の終了と共に MBP Core が明示的にデクリメントする必要がある．このために内部レジスタが用意されている．

  ただし，MTC で割込みを起こしたパケットの場合，MMC が誤動作して投機的に PEND counter をデクリメントしてしまう．すると，まだ処理中のトランザクションがあるのにもかかわらず，SuperSPARC+はそのトランザクションの処理が終了したと誤って認識する．これは，PSO でのプログラム順序の保証ができないことを意味する．よって，外部にレジスタを設け，MMC から SuperSPARC+への完了信号を MBP Core が明示的に操作するなどの回避策が必要となる．

- PEND lock register

  PEND lock register は PEND counter がオーバーフロー (0xFFFC 以上) したときに使用されるレジスタである．MBP Core がオーバーフローによる割込み処理を行っている間の lock register の役割を果たす．

5.3.3.4　MTC(Memory Timing Controller)

MTC は，SBUSCTL によってメモリへの参照が必要であると判断された場合にのみ起動される．その際には，SRAM Controller と SDRAM Controller，Address Controller を起動し，タグ用の SRAM とデータ用の SDRAM を同時に参照することになる．これによりタグ用の SRAM 参照時間を SDRAM の setup 時間で隠蔽することが可能である．そのタグが表 5.2 の条件を満たす場合には MTC は MBP Core へ割込みをかけることになる．

また，SDRAM を参照する可能性があるのは次の場合である．

表 5.2 MTC の割込み条件一覧

Command	MTC Interrupt
read_req	!valid || rd_trap
n_c_rd_req	!valid || rd_trap
allread_req	!valid || rd_trap
swap_req	global || !valid || rd_trap || wrt_trap
rdinv_req	global || !valid || rd_trap
update_via_home_req	global || !valid
write_req	!valid || global || wrt_trap || udword
writeback_reply	!valid || global || wrt_trap
update_direct_req	global || !valid || udword
n_c_write_req	!valid || global || wrt_trap || udword
invalidate_via_home_req	global || !valid
control_read_req	発生せず
control_write_req	発生せず
その他	無条件

tag は図 5.23 を参照されたい．udword とは double word 未満のサイズを表す．
また，!はそのビットが 0 であることを示す．

(1) read_req

(2) n_c_rd_req

(3) allread_req

(4) swap_req

(5) rdinv_req

(6) write_req

(7) writeback_reply

(8) n_c_write_req

(9) `home` が 0 な control_read_req

(10) `home` が 0 な control_write_req

- Address Controller

  Address Controller はメモリに対するアドレス線の制御を行っている．メモリのアドレスフォーマットについて図 5.26 に示す．MBP-light ではピン数の問題から SRAM と SDRAM の両方に対するアドレス線を共通化している．そのため，メモリ参照

のタイミングによって制御する必要がある．メモリの参照開始時はパケットのアドレスに従って，SRAMへのアドレス及びSDRAMのRAS(Row Address Strobe)を出力する．しばらく後，SDRAM ControllerからCAS(Column Address Strobe)出力要求がくるので，Address ControllerはCASを出力する．また，リセット時にはSDRAMのmode registerを書込み，リフレッシュ時にはSDRAMの両bankを選択する．

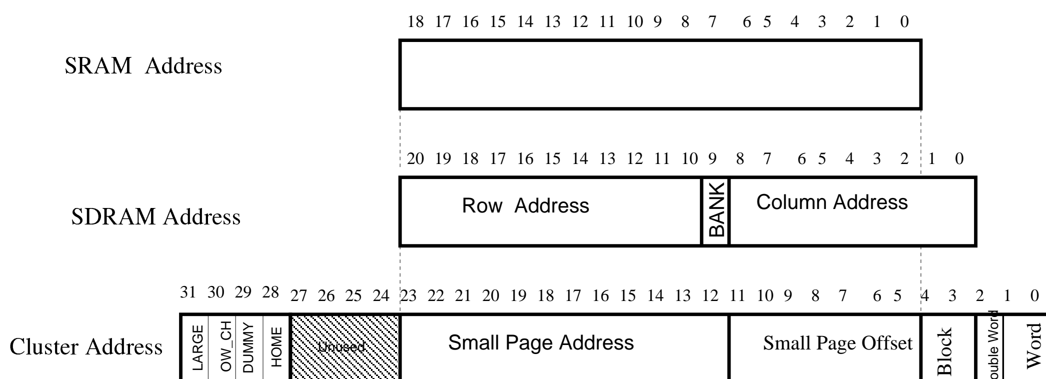


図 5.26 クラスタメモリのアドレスフォーマット

- SRAM Controller

  SRAM ControllerはSRAMの制御を行う．通常時にはSRAMは読出し専用であり，OE(Output Enable)の制御を行ってデータを取り込むだけである．ただし，MBP Coreからの書込み要求が到着すると，SRAMのOEやWE(Write Enable)の信号線を制御しデータを書き込む．

- SDRAM Controller

  SDRAMの制御を行う．SDRAMの参照手順にしたがってAddress ControllerにRASやCAS出力の指示を送り，データの読み書きを行う．また，double word毎に存在するI-Structure等の同期機構を構成するタグの読み書きも同時に行う．さらに，メモリのエラーチェックも行うことができる．これはMBP Coreによって制御することになる．その場合，データ書込み時にはECC(Error Correcting Code)を付加し，データ読出し時には誤り検査を行う．ECC parity行列を表5.3に示す．このECC codeは2 bitの誤り検出と1 bitの誤り訂正を行うことができる．ただし，自動誤り訂正の機能はないので，MBP Coreにより誤り訂正を行うことになる．64 bitのデータに対する具体的な手順を以下に示す．

  (1) 表5.3の列毎に‘1’が立っているビットのデータに対して排他的論理和をとり，その結果を ECC_{now} とする (逆に，書込み時にはこの結果をECCとして書き込むことを明記しておく).
  (2) MBP Coreが明示的にECC格納メモリから ECC_{old} を読み出す．
  (3) ECC_{new} と ECC_{old} との排他的論理和をとり，その結果をsyndromeと呼ぶ．

(4) syndrome と表 5.3 の行を比較し，一致すればその 1 bit を訂正すればよい．また，そうでなければどこかの 2 bit が誤りを起こしたことを検出することができる．

読出し時にどの 64 bit のデータが誤りを起こしたかどうかは，付録 B で説明する内部レジスタの MTC_INT+で特定できる．

- Refresh Controller

  クロックをカウントし，一定値になると SDRAM Controller に対してリフレッシュ要求を行う．この値は MBP Core によって変更可能である．

5.3.4 MBP Core

単純なプロトコル処理はすべて RDT Interface と MMC の hardwired-logic で処理される．そして，これらが処理できないような複雑なプロトコル処理が生じると，割込みを通じて MBP Core が起動される．MBP Core の主な仕事はキャッシュラインの転送やキャッシュの一致制御，パケットの生成や分解などである．

5.3.4.1 MBP Core の構成

図 5.27 に MBP Core の構成を示す．MBP Core は命令長が 21 bit で，データ長が 16 bit の 4 段パイプラインをもつ RISC プロセッサである．16 bit 幅の GPR(General Purpose Register) が 16 個と packet buffer としての役割をもつ 68 bit 幅の PBR(Packet Buffer Register) を 112 個有する．アドレス空間は I/O 空間とメモリ空間が分離しており，LIO と呼ばれる信号線でこれを切り換えることになる．MBP Core のローカルバス (後述の 5.3.5 節) 上には命令とデータ格納用の 21 bit ×64 K のローカルメモリが接続される．また，I/O 空間として LED や RS-232C，MBIF，STAFF-Link などが接続される．data/address の指定方式は JUMP-1 全体の方式に従い，Big-Endian である．さらに，MBP Core は 256 エントリ ×16 bit の内部メモリをもつ．この内部メモリは種々の目的に使用し，割込みや RDT Interface などの制御を行うことができる．

5.3.4.2 Buffer-Register Architecture

MAGIC[18] や SCLIC[16] ではプロトコル処理を高速化するために，パケットのヘッダ部を GPR として直接扱うことが可能である．しかし，JUMP-1 ではパケットのヘッダ部が大きく，種類によって可変である．また，データ部に含まれる同期タグもメモリ上での同期構造を実現するためにも扱う必要がある．したがって，データ部も含めた packet buffer 全体を扱えることが望ましい．すると，packet buffer は 68 bit 幅となり，これをプロセッサの GPR として扱うには膨大なハードウェア量が必要となってしまう．

そこで，68 bit 幅の PBR という特殊なレジスタで代用することを考える．PBR は同時に読み書き可能な 2 port memory で構成されている．そして，RDT パケット送信用と受信用がそれぞれ 24 個，汎用が 64 個で合計 112 個存在する．PBR は GPR の値によって 3

表 5.3 ECC Parity 行列

ECC	Data	7	6	5	4	3	2	1	0	# of 1
	63	1	1	0	0	1	0	0	0	3
	62	1	1	0	0	0	1	0	0	3
	61	1	1	0	0	0	0	1	0	3
	60	1	1	0	0	0	0	0	1	3
	59	1	1	1	1	0	1	0	0	5
	58	1	0	0	0	1	1	1	1	5
4		0	0	0	1	0	0	0	0	1
3		0	0	0	0	1	0	0	0	1
	57	1	1	1	0	0	0	0	0	3
	56	1	0	1	1	0	0	0	0	3
	55	0	0	0	0	1	1	1	0	3
	54	0	0	0	0	1	0	1	1	3
	53	1	1	1	1	0	0	1	0	5
	52	0	0	0	1	1	1	1	1	5
5		0	0	1	0	0	0	0	0	1
2		0	0	0	0	0	1	0	0	1
	51	1	0	0	0	1	1	0	0	3
	50	0	1	0	0	1	1	0	0	3
	49	0	0	1	0	1	1	0	0	3
	48	0	0	0	1	1	1	0	0	3
	47	0	0	1	1	1	0	0	0	3
	46	0	0	1	1	0	1	0	0	3
	45	0	0	1	1	0	0	1	0	3
	44	0	0	1	1	0	0	0	1	3
	43	1	0	1	0	1	0	0	0	3
	42	1	0	1	0	0	1	0	0	3
	41	1	0	1	0	0	0	1	0	3
	40	1	0	1	0	0	0	0	1	3
	39	1	0	0	1	1	0	0	0	3
	38	1	0	0	1	0	1	0	0	3
	37	1	0	0	1	0	0	1	0	3
	36	1	0	0	1	0	0	0	1	3
	35	0	1	0	1	1	0	0	0	3
	34	0	1	0	1	0	1	0	0	3
	33	0	1	0	1	0	0	1	0	3
	32	0	1	0	1	0	0	0	1	3
	31	1	0	0	0	1	0	1	0	3
	30	0	1	0	0	1	0	1	0	3
	29	0	0	1	0	1	0	1	0	3
	28	0	0	0	1	1	0	1	0	3
	27	1	0	0	0	1	0	0	1	3
	26	0	1	0	0	1	0	0	1	3
	25	0	0	1	0	1	0	0	1	3
	24	0	0	0	1	1	0	0	1	3
	23	1	0	0	0	0	1	0	1	3
	22	0	1	0	0	0	1	0	1	3
	21	0	0	1	0	0	1	0	1	3
	20	0	0	0	1	0	1	0	1	3
	19	1	0	0	0	1	1	0	0	3
	18	0	1	0	0	1	1	0	0	3
	17	0	0	1	0	1	1	0	0	3
	16	0	0	0	1	1	1	0	0	3
	15	0	1	1	0	1	0	0	0	3
	14	0	1	1	0	0	1	0	0	3
	13	0	1	1	0	0	0	1	0	3
	12	0	1	1	0	0	0	0	1	3
	11	1	1	1	1	1	0	0	0	5
	10	0	1	0	0	1	1	1	1	5
7		1	0	0	0	0	0	0	0	1
0		0	0	0	0	0	0	0	1	1
	9	0	1	1	1	0	0	0	0	3
	8	1	1	0	1	0	0	0	0	3
	7	0	0	0	0	0	1	1	1	3
	6	0	0	0	0	1	1	0	1	3
	5	1	1	1	1	0	0	0	1	5
	4	0	0	1	0	1	1	1	1	5
6		0	1	0	0	0	0	0	0	1
1		0	0	0	0	0	0	1	0	1
	3	1	0	0	0	0	0	1	1	3
	2	0	1	0	0	0	0	1	1	3
	1	0	0	1	0	0	0	1	1	3
	0	0	0	0	1	0	0	1	1	3

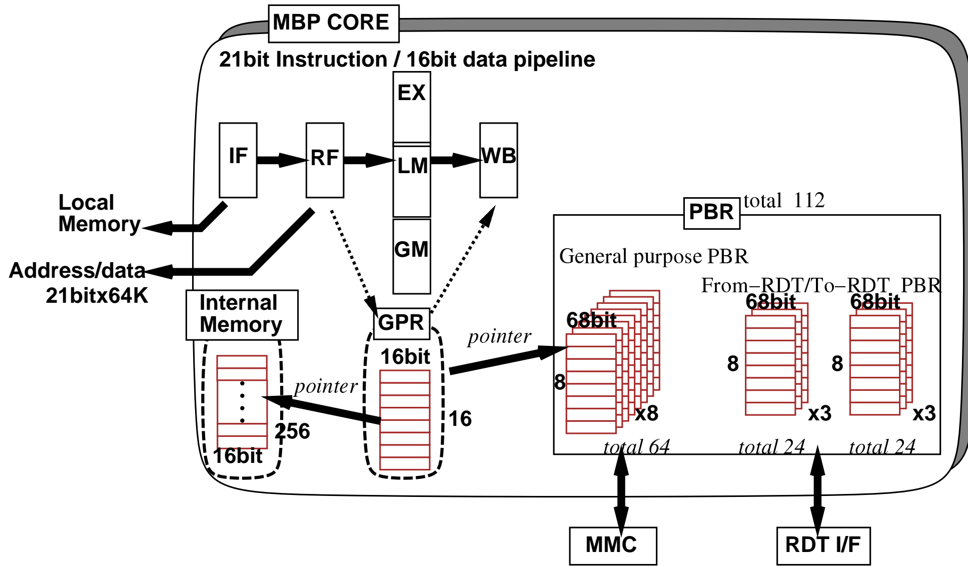


図 5.27 MBP Core の構成

種類のいずれかの PBR を切り換えて参照することができる．図 5.28 に PBR への参照方法を示す．

PBR は 8 bit を単位とした 8 つのオフセットと 4 bit のタグから形成される．そして，GPR をポインタとして任意のオフセットを参照することができる．その際に 1 つのオフセット分の 8 bit か，連続した 2 つのオフセット分の 16 bit かを命令で指定することが可能となっている．例えば，8 bit 直値演算や GPR への転送，GPR との演算などが挙げられる．ほかにも，PBR への 8 bit 転送や 16 bit 転送，68 bit 転送も可能である．演算がバッファとレジスタ間で行われることから，この構成を buffer-register architecture と呼ぶ．buffer-register architecture では PBR は単なる内部メモリではなく，パイプライン中で GPR 同様にストールを伴うことなく扱うことができる．代表的なものとして GPR と PBR 間の加算命令を図 5.29 に示す (`addpg r1 r2(0)`)．

図 5.30 に示すように，MMC と RDT Interface の構造上の相違から PBR の参照方法は異なっている．図中の破線は MMC や RDT Interface からパケットの流れを，太線は MBP Core からの PBR の参照を示す．

各 RDT パケット送受信用の 24 個の PBR は RDT パケットの 1 つ分の大きさ (8 個) を単位とし，それぞれ 3 パケット分で cyclic buffer を構成する．すると，送受信ともに 1 つのパケットを MBP Core から扱うことができるため，送受信の際には RDT Interface に制御を移して直接パケット転送を行う．ただし，リセット解除された後ではパケットを 1 つ受信したのと同じ状態となっており，リセット解除時にも RDT パケット受信用 PBR を通常の PBR と同様に使用可能である．できれば，MBP Core の初期化時に空のパケット受信命令を発行し，この状態を抜けた方が望ましい．

一方，クラスタメモリの参照やクラスタバスにパケットを転送するためには，MMC 中の入出力バッファとの間でパケットの送受信を行う必要がある．転送には RDT パケット送受信用や汎用のいずれの PBR の任意の場所を用いてもよいが，処理の終了まで 10 クロッ

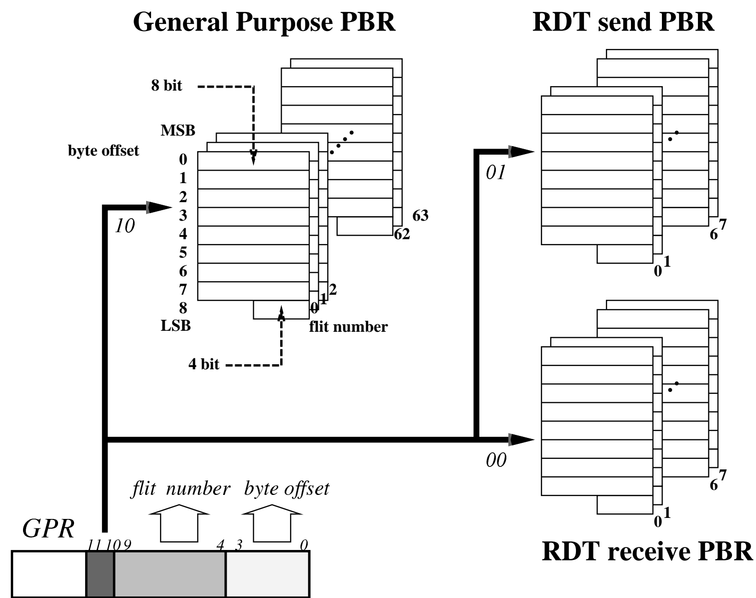


図 5.28 GPR による PBR の参照方法

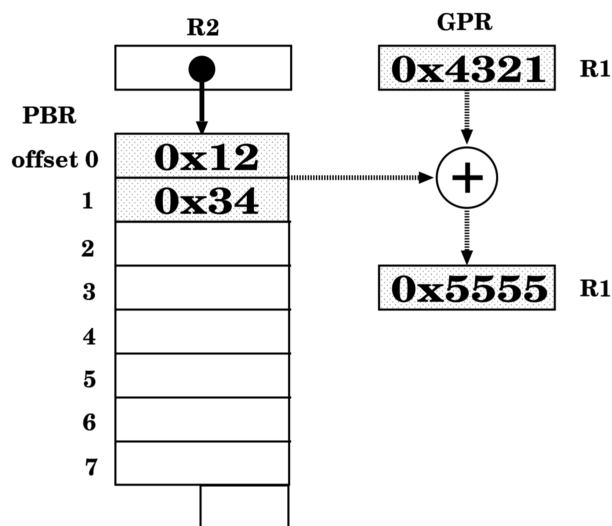


図 5.29 GPR と PBR 間の加算命令

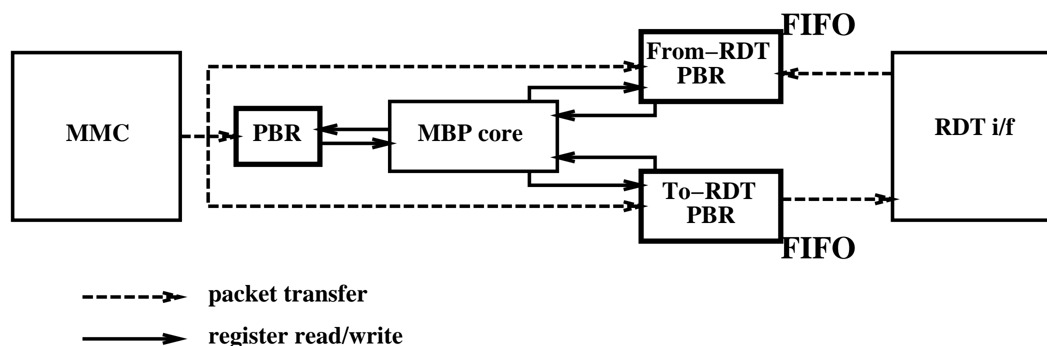


図 5.30 MMC や RDT Interface からの PBR の参照方式

ク以上要する．MBP Core がこの命令の完了を完全に待ってから後続の処理を行うと，性能を大きく劣化させてしまう．そこで，後続の実行で転送に使用中の PBR と同種の PBR を参照しない限り，現在行われている転送の終了を待たなくてもよいような構造をとっている．同種の PBR を参照しない命令を埋め込むことができれば，MMC とのパケット転送時間を隠蔽することができる．もし同種の PBR を参照する必要があるならば，メモリの転送同期命令を実行し，転送の終了を待つ必要がある．

5.3.4.3　パイプライン構成

図 5.27 中に示すように MBP Core は 4 段のパイプライン構成を取る．GPR は 16 bit の 3 ポートメモリで構成されている．また，演算命令で扱うデータは基本的に 16 bit 以内であり，displacement の範囲は 4 bit とする．MBP-light は SuperSPARC+のクロックに同期して動作するため，50MHz で動作しなければならない．今回の実装に用いる ASIC のゲート遅延を考えると，1 クロック以内に GPR の読出しと displacement 演算，あるいは PBR の読出しと 16 bit 演算を行うことが可能である．このため，1 ステージでこれらの処理を行うことにより，ステージ数を減らし，ステージ間のフォワーディングを単純化した．また，パイプライン間のデータは次の場合にフォワーディングされない．

- PBR のポインタとなる GPR に書き込んだ値†
- PBR に書き込んだ値
- jump 命令で使用する GPR

パイプラインを構成する 4 つのステージでは次のとおりである．

†PBR のアクセスを行う場合は，GPR に該当する PBR 番号を格納して，GRP をポインタとしてアクセスする．

(1) IF (Instruction Fetch)

外部から命令を取ってくる．この処理は1クロックで終了する．命令フェッチは基本的にローカルメモリから行う．しかし，リセットの解除時にローカルバスのwait信号をLowにしておくことで，I/O空間からのブートが可能である (詳細は5.3.5節を参照).

(2) RF (Register Fetch)

メモリ参照命令ならば，GPRを読み出して実効アドレスの計算を行う．また，分岐命令の際には，分岐条件の判断やアドレスの計算，分岐を行なう．その他の命令では命令中で指定されたGPR双方の値を読み出し，displacement演算を行う．

(3) 3段目のステージは、処理内容によって3つに分岐する

  (a) EX (EXecution)
  PBRの読出し及び16 bit演算を行う．

  (b) LM (Local Memory Access)
  ローカルメモリ及びI/Oへの参照を行う．MBP-lightより動作速度の遅いI/Oに対する参照では，ストールする．

  (c) GM (Global Memory Access)
  MMCを介してのクラスタメモリへの参照及びCBusパケットの転送を行う．後続の関連するPBRを参照しない命令は，LMやEXを通って先に進むことができる．

(4) WB (Write Back)

結果の格納をPBRやGPRに対して行う．パイプライン制御を簡単にするため，ローカルメモリへの参照命令やPBR間の転送命令は1クロックだけパイプラインをストールさせる．また，分岐命令は1クロックの遅延スロットを必要とする．

5.3.4.4　内部メモリ

MBP Coreにおける内部メモリには様々なものがマップされている．図5.31に内部メモリのメモリマップを示し，各内部メモリの仕様について述べる．

- Ack Cache

  詳細はRDT InterfaceのAck Collector(5.3.2.5節) を参照されたい．

- Ack Emergency buffer

  詳細は，RDT InterfaceのAck Emergency buffer(5.3.2.5節) を参照されたい．

Address	Content
0000 – 007f	Ack Cache for rank–0 (way–0)
0080 – 0081	Ack Cache for rank–0 (way–1)
0100 – 017f	Ack Cache for upper–rank (way–0)
0180 – 0181	Ack Cache for upper–rank (way–1)
0200 – 0203	Ack emergency buffer
0300 – 037f	Ack Cache lock bit
0800 – 09ff	Net Cache entry(way–0)
0a00 – 0bff	Net Cache tag(way–0)
0c00 – 0dff	Net Cache entry(way–1)
0e00 – 0fff	Net Cache tag(way–1)
1000 – 10ff	Ack map cache tag
1100 – 11ff	Ack map cache entry
1800	RDT interface command/status register
2000	Jump table mask bit
2001	Jump table priority bit
2002	Interrupt mask bit
2003	Interrupt enable bit(read only)
3000	Interrupt request bit
3001	Reset bit
4000 – 407f	Jump table
6000 – 6003	Clock counter 0
6004 – 6007	Clock counter 1
6008 – 600b	Event counter 0
600c – 600f	Event counter 1
6010 – 6012	Timer
9000 – 902f	MMC internal register

図 5.31 内部メモリのアドレスマップ

- Ack Cache lock bit

  Ack Cache lock bit は基本的に Ack Collector によって設定される．しかし，そのクリアは MBP Core の役割である．よって，Ack Cache の各アドレス (上位，下位共通) に 1 bit 用意されている．

- Net Cache

  詳細は，RDT Interface の Ack Generator(5.3.2.6 節) を参照されたい．

- Ackmap Cache

  詳細は，RDT Interface の Ack Generator(5.3.2.6 節) を参照されたい．

- RDT Interface command/status register

  詳細は，RDT Interface の MBP Core Interface(5.3.2.6 節) を参照されたい．

- Jump table mask bit

  詳細は，Table Jump と割込みの節 (5.3.4.5 節) を参照されたい．

- Jump table priority bit

  詳細は，Table Jump と割込みの節 (5.3.4.5 節) を参照されたい．

- Interrupt mask bit

  詳細は，Table Jump と割込みの節 (5.3.4.5 節) を参照されたい．

- Interrupt enable bit

  詳細は，Table Jump と割込みの節 (5.3.4.5 節) を参照されたい．

- Interrupt request bit

  詳細は，Table Jump と割込みの節 (5.3.4.5 節) を参照されたい．

- Reset bit

  RDT router 及びクラスタバス，2 次キャッシュ，SuperSPARC+をリセットするためのビット．これは書込み専用であり，リセット解除時にはネゲートされている．よって，リセットする際には明示的に行う必要がある．クラスタバスと 2 次キャッシュ，SuperSPARC+のリセットポートが 0 bit 目，RDT rouer のリセットポートが 1 bit 目である．そのビットに対して 1 を書き込むとリセットがかかり，0 を書き込むと解除される．

- Jump table

  詳細は，Table Jump と割込みの節 (5.3.4.5 節) を参照されたい．

- Clock/Event counter と Timer

  詳細は，Timer/Monitor 機能 (5.3.4.6 節) を参照されたい．

- MMC internal register

  詳細は，MMC 内部 register(付録 B) を参照されたい．

5.3.4.5　Table Jump と割込み

MBP Core は MMC や RDT Interface，外部 I/O device からの割込みをサポートしている．割込み要因は MMC や RDT Interface 内部でキューに保持され，すべての割込み要因が満足されるまで連続して割込みがかかることになる．

- 割込みレベル

  割込み (または，Table Jump) を処理する際には，内部メモリにある Jump table mask や Interrupt mask の割込みレベルに対応する各ビットを設定しなければならない．例えば，割込みレベルが 3 であれば 3 bit 目を 1 にする必要がある．割込みレベルは表 5.4 を参照されたい．Table Jump はポーリングすることによって生じるため，割込みとは性質が異なる．よって，別々の mask register が用意してある．ちなみに，この mask はリセット時にすべて 0 になる．

- Jump Table

  割込みが起こった場合，または割込み付き Table Jump 命令 (TJRQ) が実行された場合は，予約された Jump Table から値を読み出してその番地に飛ぶ． 表 5.4 に Jump Table の予約されている番地と優先度，その番地が利用される場合について示す．

  ここで，“MMC に起きた予期せぬこと” とは，2 次キャッシュから緊急トラップ信号を受けとったこと，パケットの 47 bit 目を 1 にして MBP Core が MMC にメモリ参照要求を出したこと，または付録 C の図 C.1 において

  – CNTL field が 0
  – CNTL field が 1 な control_read_req
  – CNTL field が 1 な control_write_req

  以外のアドレスフリットを用いてメモリ参照要求を出したことを意味する．

- 割込み優先度制御

  割込みの優先度は固定であり，何も設定しない場合は mask register のビット番号の大きい方が優先度が高い．優先度を変更する機構として，優先度レジスタである Jump table priority bit により変更することができる．優先度を高くしたいビットに 1 を設定すると，0 が設定されているレジスタより優先度を高くすることができる．これにより，簡単にある程度の優先度の変更であれば行うことが可能となる．しかし，自由に優先度を設定することはできない．現実に，SBUSCTL でのリクエストパケットによる割込みが MTC でのリクエストパケットによる割込みよりも優先度が低いとスタベーションを起こす．よって，MTC のリクエストパケットによる割込みの優先度の方を高くする必要がある．

表 5.4 Jump Table

Address	割込み レベル	意味
0x4000	4	MMC の SBUSCTL でリクエストパケットが割込みを起こした．
0x4001	4	MMC の SBUSCTL でリプライパケットが割込みを起こした．
0x4002	4	PEND counter が溢れている間，あらかじめ設定されていた PEND lock register にマッチするパケットを受信した．
0x4003	4	lock register にマッチするパケットを受信した．
0x4004	5	MMC の MTC でリクエストパケットが割込みを起こした．
0x4005	5	MMC の MTC でリプライパケットが割込みを起こした．
0x4006	5	MMC に予期せぬことが起きた．
0x4007	5	DRAM で ECC 誤りが発生した．
0x4008	3	この node が宛先であるマルチキャストパケットを受信した．
0x4009	3	この node が宛先である応答パケットを受信した．
0x400a	3	通常の status report packet か timeout packet のいずれかを受信した．
0x400b	3	マルチキャストパケットにパリティ誤りが発生した．
0x400c	1	Ack Emergency buffer へパケットを受信した．
0x400d	0	RDT router からの shootdown が終了した．
0x400e	2	RDT 送信用 PBR が空いている．
0x4018	6	PEND counter が overflow を起こした．
0x4019	7	PEND counter が underflow を起こした．
0x401b	8	内蔵 timer がタイムアウトを起こした．
0x401c	9	STAFF-Link から割込みがあった．
0x401d	10	MBIF から割込みがあった．
0x401e	11	Elastic Barrier から割込みがあった．
0x401f	12	外部の I/O device から割込みがあった．

- 割込み要因のクリア

  外部からの割込み要因に対してはその処理が終了した後にその要因をクリアしなければならない．表 5.5 に割込みレベルとクリアの方法対応を示す．

表 5.5 割込みレベルとクリア方法

level	方法
0	CSM 命令を発行
1	CAB 命令を発行
2	PRDT 命令を発行
3	GRDT 命令を発行
4	CCB 命令を発行
5	CMT 命令を発行
6	COV 命令を発行
7	CUF 命令を発行
8	自動的に clear
9–12	I/O 空間への参照によりデバイスのクリア命令を実行‡

5.3.4.6　Timer/Monitor 機能

MBP-light は性能計測用の Monitor 及び Timer を内蔵している．Monitor は 32 bit のカウンタを 4 つ保持しており，同時に 4 つのイベントまで計測可能となっている．Timer はタイムアウトする時間を $2^4 \sim 2^{27}$ まで変化させることができ，その時間を割込みを用いて知ることができる．MBP Core が Timer や Monitor を扱うのに必要な内部メモリのアドレスとその割当てを表 5.6 に示す．

- Monitor

  Monitor 用のカウンタは 2 種類ある． Monitor 0 と 1 は clock counter で，Monitor 2 と 3 は event counter になっている．clock counter は信号がアクティブになっているクロック数を計測する．一方，event counter は信号が Low から High に変化した時に計測される．カウンタは 32 bit であるため，上位と下位の 16 bit に分けて内部メモリを参照する必要がある．また，計測できる項目は制限されている．計測する際には，参照する項目を選び，その後その項目に対する命令を書き込まなければならない．表 5.7 と表 5.8 に Monitor 0 と 1 の命令と項目を示す．また，表 5.7 と表 5.8 に Monitor 2 と 3 の命令と項目を示す．ただし，表中の線より上が命令を，下が計測する項目を表すものとする．

‡デバイス毎に作成する必要がある．

表 5.6 Timer/Monitor の内部 memory map

Address	Meaning
0x6000	Monitor 0 Upper 16 bit [31:16]
0x6001	Monitor 0 Lower 16 bit [15:0]
0x6002	Monitor 0 command [2:0]
0x6003	Monitor 0 overflow [0]
0x6004	Monitor 1 Upper 16 bit [31:16]
0x6005	Monitor 1 Lower 16 bit [15:0]
0x6006	Monitor 1 command [2:0]
0x6007	Monitor 1 overflow [0]
0x6008	Monitor 2 Upper 16 bit [31:16]
0x6009	Monitor 2 Lower 16 bit [15:0]
0x600a	Monitor 2 command [2:0]
0x600b	Monitor 2 overflow [0]
0x600c	Monitor 3 Upper 16 bit [31:16]
0x600d	Monitor 3 Lower 16 bit [15:0]
0x600e	Monitor 3 command [2:0]
0x600f	Monitor 3 overflow [0]
0x6010	Timer Upper 16 bit [31:16]
0x6011	Timer Lower 16 bit [15:0]
0x6012	Timer command [2:0]

表 5.7 Monitor 0 の命令と状態

command	action
000	カウンタをクリアする．
001	カウンタをホールドする．
010	クロック数を数える．
011	RDT 受信バッファが満杯状態
100	RDT 受信バッファが空状態
101	Ack Emergency buffer が満杯状態
110	Packet Handler から Ack Collector へのパケット受渡しバッファが満杯状態
111	クラスタバスからの要求がある状態

表 5.8 Monitor 1 の命令と項目

command	action
000	カウンタをクリアする．
001	カウンタをホールドする．
010	クロック数を数える．
011	RDT 送信バッファが満杯状態
100	RDT 送信バッファが空状態
101	Ack Emergency buffer が満杯状態であるのにもかかわらず，さらに書込み要求がある状態
110	Net Cache にヒットするにもかかわらず，応答パケットを生成するバッファが満杯状態
111	未使用

表 5.9 Monitor 2 の命令と項目

command	action
0000	カウンタをクリアする.
0001	カウンタをホールドする.
0010	回数を数える.
0011	RDT router への送信回数
0100	RDT router への応答パケットの送信回数
0101	MBP Core から RDT router への送信回数
0110	rank 0 用 Ack Cache にマッチした回数
0111	Ack Cache への登録要求があった回数
1000	Ack 収集要求があった回数
1001	Net Cache に要求があった回数
others	未使用

表 5.10 Monitor 3 の命令と項目

command	action
0000	カウンタをクリアする.
0001	カウンタをホールドする.
0010	回数を数える.
0011	RDT router からの受信回数
0100	RDT router からの応答パケットの受信回数
0101	RDT router からのマルチキャストパケットの受信回数
0110	上位 rank 用 Ack Cache にマッチした回数
0111	Ack Cache への登録が失敗した回数
1000	Ack Cache から上位階層へ応答パケットを生成した回数
1001	Net Cache にマッチした回数
others	未使用

- Timer

  Timer の命令を以下の表 5.11 に示す．Timer はクリアすると止まり，タイムアウト時間を設定すると再びスタートする．一定時間が過ぎてタイムアウトが起こった際には，割込みが発生する．

表 5.11 Timer の命令

command	action
000	Timer をクリア
001	2^4 クロックでタイムアウト
010	2^{11} クロックでタイムアウト
011	2^{15} クロックでタイムアウト
100	2^{17} クロックでタイムアウト
101	2^{19} クロックでタイムアウト
110	2^{23} クロックでタイムアウト
111	2^{27} クロックでタイムアウト

5.3.4.7　命令セットアーキテクチャ

本節では，MBP Core の命令セットの設計思想や概要について示すが，詳細については付録 C を参照されたい．

まず，図 5.32 に MBP Core の代表的な命令形式を示す．GPR は 3 ポートメモリで構成されているため，GPR 同士の演算は 3 アドレシング方式で行われる．なお，GPR の 0 番は常に 0 となっている．また，PBR は 2 ポートメモリで構成されており，演算は 2 アドレシング方式で行われる．MBP Core の命令の分類を表 5.12 に示す．

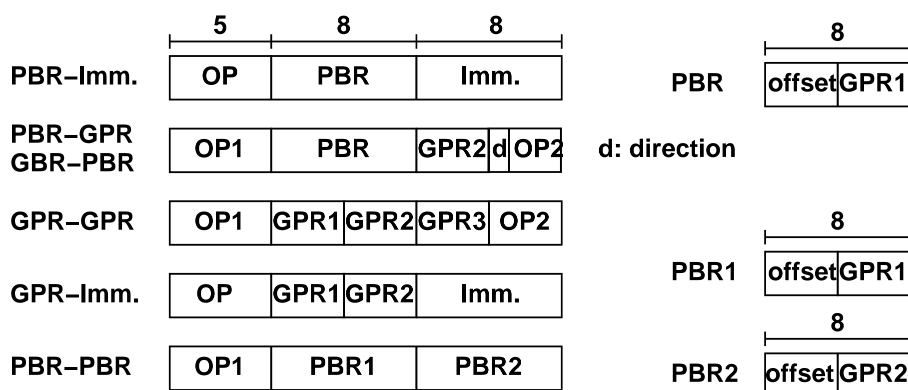


図 5.32 代表的な命令形式

表 5.12 命令の分類

Class	Type
WGG	operate between GPR and GPR
LMA	access to local memory
BRANCH	branch
WGI	operate between GPR and Imm.
BPI	operate between PBR and Imm.
RDT	control RDT Interface
MPP	transmit from PBR to PBR
WPG	operate between PBR and GPR
MMC	control of MMC
TJ	table jump
INT	control interrupt
IMA	access to internal memory
SPE	special instructions
NOP	no operation

WGG 命令 (レジスタ間演算命令): 3アドレッシングでレジスタを指定する．GPR2, GPR3はソース，GPR1はデスティネーションである．GPRのうち，R0は常に0である．このため，レジスタ間転送命令や比較命令は存在しない．MBPは数値演算を行う事は想定していないので，算術シフトは実装していない．

表 5.13 WGG 命令

Mnemonic	Meaning
SUBGG	Sub
ADDGG	Add
SLLGG	Shift Left logical
SRLGG	Shift Right logical
ANDGG	AND
ORGG	OR
XORGG	Exclusive OR

WGI 命令 (レジスタ-直値間演算命令): 8ビットの直値とGPRの間で演算を行う．

Branch 命令: パケット制御に関しては主にテーブルジャンプを用いるため，MBPのBranch命令は基本的なもののみを備えている．FlagはCarry flag(C), Zero flag(Z)の2つを備える．これらのFlagは演算命令によってセットされ，これをチェックして分岐する．条件分岐命令はプログラムカウンタ相対ジャンプだが，その他のジャンプ(BRA)は絶対ジャンプである．サブルーティンコールは戻り番地をGPR15に入れて絶対ジャンプする．

表 5.14 WGI 命令

Mnemonic	Meaning
SUBGI	Sub Imm.
ADDGI	Add Imm.
SLLGI	Shift Left Logical Imm.
SRLGI	Shift Right Logical Imm.
ANDGI	AND Imm.
ORGI	OR Imm.
XORGI	Exclusive OR Imm.
LHI	Load High Imm.

変則命令コードの利点を生かし，絶対ジャンプはすべてのアドレス領域をカバーする．リターンには BR 命令を用いる．

すべての分岐命令はディレイスロット 1 を持つ遅延分岐である．BR で使うレジスタに対してはフォワーディングが行われない．

表 5.15 Branch 命令

Mnemonic	Meaning
BOZ	Branch on Zero
BNZ	Branch not Zero
BGT	Branch greater than
BLE	Branch Less equal
BAL	Branch and Link
BRA	Branch
BR	Branch Register

ローカルメモリアクセス: ローカルメモリは 21 bit 幅，16 bit アドレシングである．また，64K の I/O 空間を持ち，外部的には I/O 信号で区別される．これらのバイト単位のアクセスは想定していない．ピン数の不足により命令メモリとデータメモリの分離は行っておらず，キャッシュも装備していないため，ローカルメモリアクセスは 1 クロックのストールを伴う．アドレッシングは 7 bitdisplacement 付き間接レジスタ指定である．

表 5.16 ローカルメモリアクセス命令

Mnemonic	Meaning
LLM	Load Local Memory
LLI	Load I/O
SLM	Store Local Memory
SLI	Store I/O

PBR に関する命令:　この PBR は GPR を用いて 4 bit オフセット (displacement) 付きのレジスタ間接指定でアクセスされる．命令中で PBR はポインタとなる GPR(4 bit) + オフセット (4 bit) = 8 bit 長で指定される．PBR は，このアドレスにより 8 bit 単位，16 bit 単位でアクセスされるが double word(68 bit) 境界を越えたアクセスは不可能である．

RDT, Cbus パケットヘッダはフィールド長が 8 bit 以内で収まることが多く，長くても 16 bit 以内で収まることが多いため，PBR は 8 bit 単位，16 bit 単位に扱うのに便利な命令体系になっている．

- WPI 命令: 8 bit の直値データと指定された PBR の 8 bit フィールドとの間で演算を行う命令．ヘッダのフィールドの効率よい操作が可能である．

表 5.17 WPI 命令

Mnemonic	Meaning
SUBPI	Sub PBR Imm.
ADDPI	Add PBR Imm.
SLLPI	Shift Left PBR Imm.
SRLPI	Shift Right PBR Imm.
ANDPI	And PBR Imm.
ORPI	OR PBR Imm.
XORPI	Exclusive OR PBR Imm.
LPI	Load PBR Imm.
CPI	Compare PBR Imm.

PBR は 2 アドレッシングであるので，Compare 命令，直値をセットする命令が存在する．シフト操作はその 8 bit フリット内で行われる．

- WPG 命令: PBR の指定された 16 bit と GPR 間の演算命令 (MVBPG は 8 bit データ) である．GPR，PBR のどちらをデスティネーションにとることもできる．

表 5.18 WPG 命令

Mnemonic	Meaning
SUBPG	Sub PBR-GPR
ADDPG	Add PBR-GPR
MVBPG	Move byte PBR-GPR
MVPG	Move PBR-GPR
ANDPG	AND PBR-GPR
ORPG	OR PBR-GPR
XORPG	Exculsive OR PBR-GPR

- MPP 命令:

  PBR 間の転送命令で 8 bit, 16 bit, 68 bit 単位の転送が可能である．68 bit の移動時 (MVLPP) はオフセットが使えないので，PBR1, PBR2 は 4 bit(つまりポインタとなる GPR) で表される．

表 5.19 MPP 命令

Mnemonic	Meaning
MVBPP	Move Byte PBR-PBR
MVPP	Move PBR-PBR
MVLPP	Move long PBR-PBR

RDT とのやりとり　RDT に対するパケットの送出および受信は，次の命令で行う．

表 5.20 RDT に関する命令

Mnemonic	Meaning
PRDT	Push RDT
GRDT	Get RDT

PRDT は現在 MBP Core で操作している To RDT PBR を RDT インタフェースを送出状態にして，次のバッファを To RDT PBR として MBP Core に接続する．使用可能なバッファがなくなった場合，To RDT は使用不能 (書いても書けない，読んでも値は無効) となる．From RDT PBR は，パケットが到着していない状態では使用できない (書いても書けない，読んでも値は無効)．パケットバッファは割込みがかかった時点で使用可能 (MBP Core に接続される) となり，パケットの処理が終了したら GRDT を実行してバッファを pull し，次のパケットバッファを MBP Core に接続することができる．

MMC とのやりとり　MMC とのやりとりは，以下の命令で行われる．これらの命令ではオフセットは効かず，PBR はポインタとして用いる GPR の番号そのままの 4 bit で指定される．

- MEMBAR: TGM，PBP，GBPM，GBPS の動作が完了するまで，処理を停止する．TGM などの動作に使用した PBR は MEMBAR 実行後にアクセスしなければならないが，別種の PBR(PBR 間の区別は汎用 PBR, From RDT PBR, To RDT PBR) のアクセスは MEMBAR を実行することなく行うことができる．

- TGM: クラスタメモリのアクセスは，PBR に必要なデータをセットしたうえで TGM 命令を用いることにより行う．

- PBP: C-Bus に対してパケットを送る場合，PBP 命令を用いる．この命令は，TGM 同様，PBR にデータをセットして命令を実行することによりパケットを送信する．

表 5.21 MMC に関する命令

Mnemonic	Meaning
TGM	Transfer Global Memory
PBP	Put Bus Pakcet
GBPS	Get Bus Packet from SBUS
GBPM	Get Bus Packet from MTC
MMCP	MMC Put
MMCG	MMC Get
MEMBAR	Memory Barrier

- GBPS/GBPM: 本来 PBP と対になるはずだが，MMC からのパケットは，1. Cbus から直接 MBP Core 行きと判定されるものと，2. タグアクセスの結果，MBP Core が起動されるものの 2 種類ある．前者に対応するのが GBPS，後者が GBPM である．
- MMCP/MMCG: MBP Core は MMC の内部バッファを直接読み書きすることは原則としてしないが，インタフェースを経由すると，所定のバッファ以外はアクセスできなくなってしまうため，予備として内部バッファにアクセスする命令を設けている．通常は決して用いてはならない．

Table Jump と割込み　MBP Core は RDT や MMC の状況に応じて Table Jump したり，割込み処理を起動したりすることができる．MBP Core において Table Jump と割込みは Mask ビットを除いて同じ機構を利用しており，割込みは Table Jump が自動的にかかることにより実行される．

- Table Jump: TJRQ は外部要因によって先に示したあらかじめ設定されたテーブルジャンプを行う命令．TJ は，GPR1 によって示したローカルメモリの番地の内容にジャンプするもので，プログラムによって作ったテーブルでジャンプするときに使う．

表 5.22 Table Jump 命令

Mnemonic	Meaning
TJRQ	Table Jump with Requests
TJ	Table Jump

- 割込み: 割込みが起きるためには，Interrupt Mask レジスタの対応するビットが 1 となっている要因が発生する前に，Interrupt enable レジスタに 1 を書き込んでおく必要がある．このレジスタは Mask レジスタ同様に内部アドレスにマップされているが，INTE 命令でセットすることができる．割込みによって自動的にテーブルジャンプが起こると，Interrupt enable レジスタは 0 になり，割込み禁止になる．し

たがって，割込み処理が終わったら，下の CSM–CMT 命令で要因をリセットし，また CSM では shootdown mode(6.2.3.5 節参照)，CAB は Ack 収集緊急バッファをクリアする場合に用いる．

さて，その割込み要因を取り除いた後，下の RFI 命令を実行する．RFI 命令は PC とフラグを復帰する命令で，一種のジャンプ命令である．このディレイスロットで INTE 命令を実行すれば，リターン直後に割込み可能になる．INTD は割込みを禁止する命令である．MBP Core は，多重割込みをサポートする最小限の機能をもっている．LIPA(Load Interrupt Address) は,IPA(InterruPt Address register) 中の割込みの戻り番地を GPR にロードする．また，LIPS(Load Interrupt Status) は，IPS(InterruPt Status register) 中のフラグ 2 bit を GPR の LSB 2 bit にロードする．この 2 つの命令を使って戻り番地とステータスをセーブしておき，割込み許可にすることにより多重割込みが可能になる．しかし，最初の戻り番地に戻るためには，セーブした番地とステータスを PC に戻す必要がある．番地を戻すためには BR を用いるが，これに先立ちステータスを戻す．このために SSR(Set Status Register) が用意されている．本来，IPA, ISR に値を戻す命令を用意しておくべきだが，これはパイプラインの流れを大きく乱すため，用意していない．したがって，多重割込みのプログラムを作る場合は，レジスタのどれかを復帰 PC 格納用にセーブしておく必要がある．

表 5.23 割込みに関する命令

Mnemonic	Meaning
RFI	Return From Interrupt
INTE	Interrupt Enable
INTD	Interrupt Disable
CSM	Clear Shootdown
CAB	Clear Ack em. Buffer
CCB	Dequeue C-Bus
COV	Clear Pending overflow
CUF	Clear Pending underlow
CMT	Clear MTC
LIPA	Load Interrupt Address
LIPS	Load Interrupt Status
SSR	Set Status

内部メモリアクセス命令　MBP Core はテーブルジャンプや，ちょっとしたレジスタの退避用に内部メモリを持つと共に，MMC や RDT を制御するためのレジスタやバッファ等も内部メモリとしてマッピングされる．この内部メモリは 68 bit 単位にアドレスが振ってあるが，実際にアクセスするデータ幅は 16 bit が主である．16 bit アクセスした場合は LSB16 bit が転送の対象となる．PBR の指定はオフセットは使えない．

内部メモリは以下の命令で読み書きする．その際，4 bit のオフセットを指定することが出来る．

表 5.24 内部メモリアクセス命令

Mnemonic	Meaning
LIMG	Load Internal memory to GPR
SIMG	Store Internal memory from GPR
LIMP	Load Internal memory to PBR
SIMP	Store Internal memory to PBR

内部メモリはアクセスにストールを伴わない点がローカルメモリより有利である．

その他の命令　分散共有メモリ管理では，管理用のタグやインデックスを生成する目的でハッシュ値を用いたり，タグやインデックスのビット操作などを行う事が多い．MBP Core は分散共有メモリ管理に特化した専用プロセッサなので，ハッシュ値を求める命令，ビット操作命令などの特殊命令をもっている．また，MBP-light 固有のハードウェア制御命令も備える．

表 5.25 特殊命令

Mnemonic	Meaning
HSHC	Hash Cluster Address
HSHG	Hash GPR
HSHN	Hash Net Address
GUN	Get Unique Number
BITS	Bit set
BITR	Bit reset
BITC8	Bit Count (8 bit)
TJRI	Table Jump interrupt
CRDT	Clear RDT interface
LUDR	Load Upper Data Register
SUDR	Store Upper Data Register

CRDT は，RDT インタフェースに対してクリア信号を発生する命令である．HSHC，HSHG，HSHN はハッシュ命令，GUN，BITC8 は一種のビット操作命令でそれぞれ次の機能を持つ．

- HSHC: PBR 上の 27–5 bit 目を 8 bit 単位に折り畳んで，Exclusive OR をして 8 bit 幅にして GPR の下位にセットする．上位は 0 にする．
- HSHG: GBRs 上の 16 bit を 8 bit 単位に折り畳んで Exclusive OR をして 8 bit 幅にして GPRd の下位にセットする．上位は 0 にする．

- HSHN: PBR 上の 41–5 bit 目を 8 bit 単位に折り畳んで Exclusive OR をして 8 bit 幅にして GPRd の下位にセットする．上位は 0 にする．
- GUN: GPRs 上で LSB から検索していって始めて当たった 0 の bit 番号を得る．これは Unique Number を得るのに使う．
- BITS: GPR1 上の GPR2 番目の bit をセットして GPRd に格納する．
- BITR: GPR1 上の GPR2 番目の bit をリセットして GPRd に格納する．
- BITC8: GPRs 上の下位 8 bit 中の 1 の数を数えて GPRd に格納する．

TJRI 命令は，割込み処理のため，ハードウェアが自動的に挿入する命令なので，プログラマは使わない．

LUDR と SUDR は，UDR(Upper Data Register) と GPR 間の転送命令である．UDR は，ローカルメモリの上位 5 bit をアクセスするためのレジスタで，ローカルメモリアクセス命令 (LLM, SLM, LLI,SLI 命令) で用いられる．

5.3.5　ローカルバス

ローカルバスにはローカルメモリと I/O デバイスが接続されている．各デバイスに対する現在のアドレスマップは表 5.26 のに示すとおりである．

表 5.26 I/O device の address map

デバイス	アドレスマップ
Elastic Barrier	0x0000, 0x1000
MBIF	0x2000(Data), 0x3000(Control)
LED	0x4000
Ethernet	0x5000
STAFF-link	0x6000, 0x7000
RS-232C	0x8000(Data), 0xA000(Control)
PATCH	0xc000,0xd000,0xe000,0xf000

MBP-light は LIO という信号線によってローカルメモリ空間と I/O 空間を別々に参照することができる．そして，それぞれに専用の参照命令を設けている．詳細は付録 C を参照されたい．

5.3.5.1　MBP-light のブート

リセットがアクティブでない場合のバスマスタは MBP-light であり，バスを手放すことはない．そうでない場合にはバスを解放し，I/O デバイスからローカルメモリへの DMA を可能としている．また，外部デバイス (ROM) によって I/O 空間からブートすることも可能である．それにはリセット信号を解除する際に X_LWAIT 信号を Low にしなければ

ならない．この結果，MBP-light は常に LIO を High にし，命令フェッチが I/O 空間より行われることになる．ただし，1 クロックでデータを読むことができない場合にはクロック周波数を落とす必要がある．また，I/O 空間からブートした後に再びローカルメモリ空間から命令フェッチを行う場合には，LIO を MBP-light が明示的に Low にする必要がある．この目的のために，MBP-light は特殊なレジスタ (UDR) を用意している．詳細は付録 C を参照されたい．

5.4　MBP-light の実装

MBP-light の記述はハードウェア記述言語 VHDL を用いて記述し，論理レベルの設計段階では Mentor Graphics 社の QuickHDL を用いて設計を行った．利用したデバイスは東芝 0.4μm CMOS embedded array の TC203E340 T3S35 という 197,000 ゲート相当の ASIC を用いる．電源は 3.3V/5V が混在しており，その他の実装諸元を表 5.27 に示す．また，MBP-light のレイアウトを図 5.33 に示す．

表 5.27 MBP-light の実装諸元

最大周波数	50 MHz
プロセス規則	0.4 μm
ランダムロジック	106,905 ゲート
内部メモリ	44,848 ビット
面積使用率	38.3 %
ピン数	352
パッケージ	TBGA[§]
消費電力	3.1 W

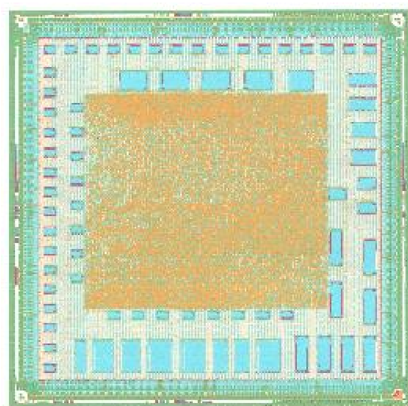


図 5.33 MBP-light のレイアウト

[§]Tape Ball Grid Array

図 5.33 からランダムロジック部が中央のブロックに，PBR や Ack Cache，Net Cache などの RAM が周辺部に配置されている様子がわかる．また，MBP-light の外観の写真を図 5.34 に示す．

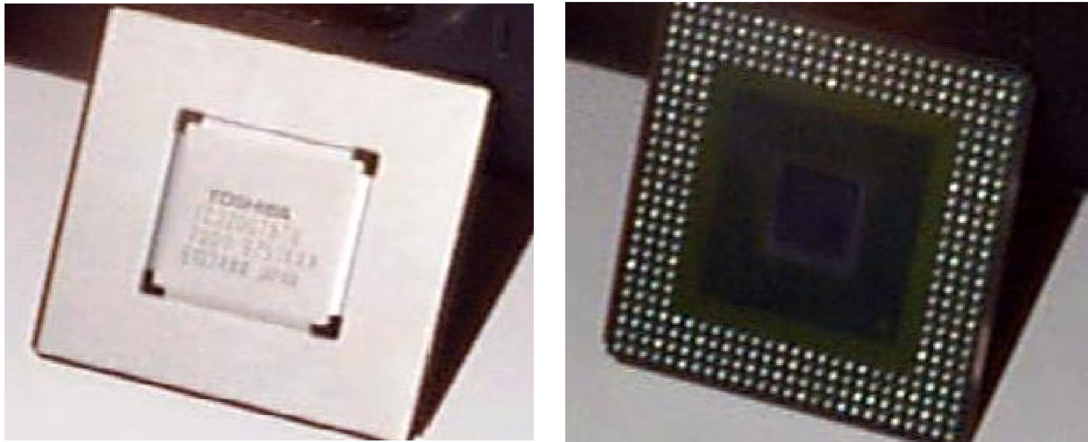


図 5.34 MBP-light の写真

最後に，MBP-light のハードウェア量を評価した結果を表 5.28 に示す．表中でメモリ量のゲート数換算が困難であったため，ビット数で示している．MBP-light では総ゲート数 197,000 のうちランダムロジック部が約半分を占めており，残りは内部メモリに使用されている．

表 5.28 MBP-light のゲート数

ブロック名	ゲート数	メモリ量 (bit)
MBP Core	35,734	6144
RDT i/f	19,832	38704
MMC	43,203	0
総計換算	98,769	44848

5.5　MBP-light の命令セットアーキテクチャの評価

本節では，著者が担当した MBP-light の命令セットアーキテクチャの評価結果について述べる．

5.5.1　評価環境

現在，JUMP-1 のファームウェアとして DSM 管理プログラムが実装されており [49]，MBP-light がこのプログラムを実行する．MBP-light の命令セットの評価の際，DSM 管理プログラムのコンパイルによって出力される命令と，256×256 の行列積を計算するプ

ログラムを実行した場合に MBP-light が実行する命令について検討する．なお，行列積の実行は要素プロセッサである SPARC が行うので，MBP-light が実行するのは DSM 管理プログラムのみである．

コンパイラ出力による検討 MBP Core で実行される共有メモリ管理プログラム全体を MBP-light 用 C コンパイラでコンパイルし，アセンブリ出力に含まれる命令から検討を行う．

行列積プログラム実行時の評価 4 クラスタの JUMP-1 システム上でプログラムを実行した際に，MBP Core がどの命令を実行するかをカウント．

共有メモリ管理プログラムは大規模で多数の条件分岐を内包しており，完全な形で実行される命令をカウントすることは非常に困難であるため，ここでは行列積プログラム実行の際に発生するパケット処理の種類と数をカウントし，各パケットの処理ルーチン内においては例外的な処理が行われないものとして実行する命令を数えることによって命令数のカウントを行った．

また，ローカルメモリの容量の関係からプログラム全体のパケットのログを取ることはできないため，最後の 100 パケットについてのデータを使用した．

5.5.2　各命令の比率

この節では，各命令の比率について示す．図 5.35 は，コンパイラの出力結果である．行列積プログラムの実行時の結果は，図 5.36，図 5.37 に示す．

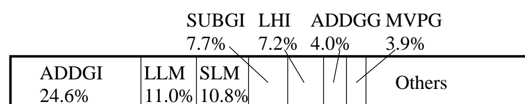


図 5.35 各命令の比率: コンパイラ出力

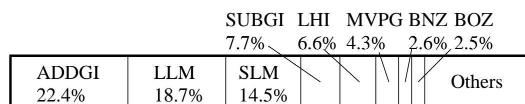


図 5.36 各命令の比率: クラスタ 0(ホームクラスタ)

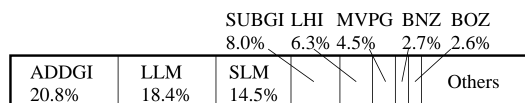


図 5.37 各命令の比率: クラスタ 1

これらの結果から，PBR を扱う命令や特殊命令に比べて，標準 RISC 命令の比率が高いことが分かる．MVPG(Move PBR from/to GPR) は特殊命令の中では比較的利用されているが，その他の命令の比率はかなり低いことが分かる．

また，最も比率が大きいのは，ADDGI 命令である．前述のとおり，GPR に関する命令は 3 アドレシングであり，move 命令の代わりに ADDGI 命令を利用する (片方のソースレジスタに，常に値が 0 である R0 を用いる)．さらに，GPR に 16 ビットの値をセットするときには，LHI 命令と ADDGI 命令を組み合わせて使用するため，ADDGI 命令の割合が高くなっている．

表 5.29 ローカルメモリアクセスの比較

	1 Clock Stall	Without Stall	Improvement
Home	64471	48394	33.2%
CL1	43130	32448	32.9%

また，ローカルメモリアクセス命令 (LLM, SLM) の比率も高く，ホームクラスタ，クラスタ 1 とも 30 これは，DSM 管理プログラムが，ローカルメモリ上に配置されたソフトウェア管理のパケットキューを扱う頻度が高いことによる．

5.3.4.7 節で示したとおり，MBP-light の命令メモリとデータメモリはパッケージのピン数の関係で分離されておらず，1 クロックのストールを伴う．表 5.29 は，ローカルメモリアクセスで 1 クロックのストールを伴う場合と，ストールなしでアクセスできるよう改良した場合のサイクル数を比較したものである．この結果から，データメモリと命令メモリを分離できた場合，ホームクラスタで 33.2%，リモートクラスタで 32.9%性能向上することが分かる．

表 5.30 Delay Slot

	Num. of Branch	No Operation	Effective Ins.	Rate
home	5187	1468	3719	72.0%
CL1	3604	1183	2421	67.2%

一方，分岐命令の割合はそれほど高くなく，全体の 10%程度となった．これらの大部分は，マッチングに用いられる BNZ(Branch Not Zero) と BOZ(Branch On Zero) である．これは，DSM プログラムが複雑な条件判断を主に行っており，単純なループ等がほとんど存在しないためである．また，MBP-light では 1 クロックの遅延分岐を用いているが，表 5.30 は有効に利用されているディレイスロットの割合を示したものである．この結果から，ディレイスロットの多くは有効に利用されており，1 クロックの遅延分岐によって生じる性能低下はローカルメモリアクセスによるものと比べて小さいことが分かる．

これまでの結果から，ホームクラスタとクラスタ 1 では，傾向に大きな違いがないことが分かる．DSM プログラムによるパケット管理は，ホームクラスタとリモートクラスタで共通点が多いため，このような結果となっている．

表 5.31 branch 命令: コンパイラ出力

命令	個数	命令	個数	命令	個数
BOZ	317	BLE	744	BR	256
BNZ	419	BAL	435		
BGT	80	BRA	700		

表 5.32 branch 命令: クラスタ 0(ホームクラスタ)

命令	個数	命令	個数	命令	個数
BOZ	1233	BLE	198	BR	782
BNZ	1240	BAL	674		
BGT	274	BRA	777		

表 5.33 branch 命令: クラスタ 1

命令	個数	命令	個数	命令	個数
BOZ	855	BLE	85	BR	535
BNZ	863	BAL	411		
BGT	243	BRA	612		

5.5.3　演算種類

演算命令である WGG，WGI，WPG，WPI の各命令の数を，演算種類である ADD，SUB，SLL，SRL，AND，OR，XOR に分類した結果を表 5.34，表 5.35，表 5.36 に示す．

表 5.34 演算種類: コンパイラ出力

命令	個数	命令	個数	命令	個数
ADD	6698	SRL	451	XOR	0
SUB	2305	AND	993		
SLL	502	OR	204		

表 5.35 演算種類: クラスタ 0(ホームクラスタ)

命令	個数	命令	個数	命令	個数
ADD	11608	SRL	562	XOR	0
SUB	4910	AND	1910		
SLL	339	OR	461		

表 5.36 演算種類: クラスタ 1

命令	個数	命令	個数	命令	個数
ADD	7290	SRL	369	XOR	0
SUB	3359	AND	330		
SLL	286	OR	365		

この結果から，JUMP-1 の DSM 管理において XOR 演算は必要ないことが分かる．また，ADD 演算の数が多いが，これは別の目的にも利用される ADDGI 命令が多いことに起因する．

5.5.4　PBR 操作命令

表 5.37 にコンパイラ出力コード中の，表 5.38，表 5.39 にホームクラスタとリモートクラスタで実行された PBR 操作命令の数をそれぞれ示す．PBR 操作命令はその特殊性のために，コンパイラが自動で命令を出力することはなく，プログラマが特殊関数呼出の形でプログラム中に記述する必要があるため，利用されている PBR 操作命令の数は限られている．PBR 操作命令の中では，Move 命令とビット操作命令が比較的利用されているが，その他の命令の利用頻度は小さい．

表 5.40 は，次に示す条件を仮定して，PBR 操作命令を通常の命令に置き換えた場合に必要となるサイクル数を示したものである．

表 5.37 PBR Manipulating Instructions: Included in the Compiled Code

Ins.	Num.	Ins.	Num.	Ins.	Num.
ADDPI	0	SRLPI	0	XORPI	0
SUBPI	0	ANDPI	10	LPI	40
SLLPI	0	ORPI	0	CPI	0
ADDPG	9	ORPG	10	MVPG	909
SUBPG	0	XORG	0		
ANDPG	41	MVBPG	252		
MVBPP	46	MVPP	144	MVLPP	158

表 5.38 PBR Manipulating Instructions: Executed in the Cluster 0 (Home Cluster)

Ins.	Num.	Ins.	Num.	Ins.	Num.
ADDPI	0	SRLPI	0	XORPI	0
SUBPI	0	ANDPI	0	LPI	52
SLLPI	0	ORPI	0	CPI	0
ADDPG	56	ORPG	0	MVPG	2099
SUBPG	0	XORG	0		
ANDPG	454	MVBPG	185		
MVBPP	2	MVPP	134	MVLPP	298

表 5.39 PBR Manipulating Instructions: Executed in the Cluster 1

Ins.	Num.	Ins.	Num.	Ins.	Num.
ADDPI	0	SRLPI	0	XORPI	0
SUBPI	0	ANDPI	38	LPI	3
SLLPI	0	ORPI	0	CPI	0
ADDPG	42	ORPG	39	MVPG	1476
SUBPG	0	XORG	0		
ANDPG	330	MVBPG	319		
MVBPP	0	MVPP	166	MVLPP	205

表 5.40 Replacement of PBR Manipulating Instructions with Other Instructions

Ins.	Cycles	Ins.	Cycles
ANDPI	3	MVBPG	1
LPI	2	MVPG	1
ADDPG	2.5	MVBPP	2
ANDPG	2.5	MVPP	2
ORPG	2.5	MVLPP	10

- パケットバッファは内部メモリとして参照される．
- 内部メモリへのバイト単位のアクセスが可能である．
- 内部メモリへのアクセスは 1 クロックで終了する．

ADDPG，ANDPG，ORPG は，デスティネーションが PBR であるか GPR であるかによって，置き換えた場合のサイクル数が異なる．GPR がデスティネーションの場合は 2 サイクルであるが，PGR の場合は 3 サイクル必要である．ここでは，簡単のために 2.5 サイクルとしている．

表 5.41 Compare Required Cycles: PBR Manipulating Instructions

	With PBR	Without PBR	Improvement
home	64471	68106	5.64%
CL1	43130	45836	6.27%

表 5.41 は，これらの仮定を基に PBR 操作命令が存在する場合としない場合について，実行サイクル数を比較したものである．この結果から，PBR 操作命令によって，ホームクラスタで 5.64%，クラスタ 1 で 6.27%性能が向上していることが分かる．

Buffer-Register Architecture をとることによるハードウェアコストの増加は，オンチップメモリと DMA を用いた手法より小さく抑えられているが，DSM 管理プログラムの性能向上への寄与はある程度限定されたものとなっている．

5.5.5　特殊命令

表 5.42，表 5.43，表 5.44 に，コンパイラ出力コードに含まれる，またはホームクラスタやクラスタ 1 で実行された特殊命令の数を示す．

特殊命令は，コンパイラが直接扱うことができないため，特殊命令に対応する関数を使用することで特殊命令を使用する．したがって，特殊命令の数は使用した関数の数によって決まる．表 5.42 に示すように，今回の実装では，HSHC，GUN，BITS，BITR，BITC8 を使用しているが，HSHC(Hash Cluster Address) 命令以外はあまり実行されていないと

表 5.42 特殊命令: コンパイラ出力

命令	個数	命令	個数	命令	個数
HSHC	35	BITS	2	LUDR	0
HSHG	0	BITR	2	SUDR	0
HSHN	0	BITC8	12		
GUN	3	CRDT	0		

表 5.43 特殊命令: クラスタ 0(ホームクラスタ)

命令	個数	命令	個数	命令	個数
HSHC	279	BITS	0	LUDR	0
HSHG	0	BITR	0	SUDR	0
HSHN	0	BITC8	4		
GUN	0	CRDT	0		

表 5.44 特殊命令: クラスタ 1

命令	個数	命令	個数	命令	個数
HSHC	204	BITS	0	LUDR	0
HSHG	0	BITR	0	SUDR	0
HSHN	0	BITC8	0		
GUN	0	CRDT	0		

いう結果となっている．これは，特殊命令が利用される状況は，扱っているパケットのアドレスと待ちパケットのアドレスのコンフリクトが発生した場合等の，例外的なケースに限られていることが大きな要因である．

表 5.45 Replacement of Special Instructions with Other Instructions

Ins.	Cycles	Reg. O.H.	Ins.	Cycles	Reg. O.H.
HSHC	7	0	BITS	3	0
HSHG	4	0	BITR	4	0
HSHN	9	0	BITC8	34	2
GUN	5-84	2			

表 5.45 は，ハッシュ値を求める命令やビット操作命令を，標準 RISC 命令で置き換えた場合に必要となるサイクル数を示したものである．「Reg. O.H.」は，命令を置き換えた場合に余分に必要となる汎用レジスタの数を示す．

表 5.46 Compare Required Cycles: Special Instructions

	With Sp. Ins.	Without Sp. Ins.	Improvement
Home	64471	66281	2.80%
CL1	43130	44354	2.83%

表 5.46 は，表 5.45 を元に特殊命令が存在する場合と存在しない場合のサイクル数を比較したものである．この結果，ホームクラスタで 2.80%，リモートクラスタで 2.83%の性能向上となっていることが分かる．

この結果から，少なくとも現在の DSM プログラムの実装においては，HSHC 命令を除く特殊命令の効果は小さいものになっていることが分かる．

5.5.6　その他の実装方法の検討

命令セットの評価結果より，次のことが分かった．

- DSM 管理プログラムは，ローカルメモリに配置されたパケットキューの処理に多くの時間を費している．そのため，ローカルメモリアクセス命令や標準 RISC 命令の利用率が高くなっている．
- PBR 操作命令のうち，算術，論理演算命令はあまり効果的に利用されていない．
- 特殊命令は，ハッシュ生成命令である HSHC 命令を除いてあまり効果的に利用されていない．
- これらの傾向は，ホーム，リモートクラスタとも共通である．

現在の DSM 管理プログラムは，より効果的に MBP Core の命令セットを利用するように改良することができる．しかし，MBP Core は DSM 管理プログラムの実装前に設計されたため，実際の DSM 管理プログラムの特徴を考慮して設計されていないことが問題となっている．DSM 管理プログラムは事前に想定していたものよりも複雑であり，MBP-light はプログラムを効率良く実行することはできない．

MBP-light は，チップ面積の制約から大きなオンチップメモリは搭載しておらず，またピン数にも制限がある．しかし，事前に DSM 管理プログラムの特徴を把握することができていれば，次に挙げるような別の実装方法を取ることも可能であったと思われる．

- Buffer-Register Archigecture をとる代わりに，パケットバッファとしても利用可能な複雑なキャッシュシステムを搭載する方が効果的である．Buffer-Register Architecture による 6.27%の性能向上が得られなくなるとしても，命令メモリとデータメモリを分割して 33%の性能向上を達成することで補うことができる．

- パケットヘッダを扱う特殊命令の代わりに，必要であればパケットバッファを扱う命令を備える．ただし，HSHC(Hash Cluster Address) 命令は装備する価値がある．

5.5.7　命令セットアーキテクチャの評価のまとめ

ここでは，JUMP-1 の実機を利用して，MBP-light が実行する DSM 管理プログラムを基に命令セットアーキテクチャの評価を行った．MBP-light は，Buffer-Register Architecture をとることにより，Home クラスタで 5.64%，Remote クラスタで 6.27%の性能向上を達成している．特殊命令では，ハッシュ値を求める命令により 2.80%の性能向上を達成した．しかし，その他の特殊命令は現在の実装では有効に働かないことが分かった．DSM 管理プログラムでは，MBP-light のローカルメモリ上に配置されたソフトウェアパケットキューの管理に大きなコストがかかっていることが分かった．このため，共通 RISC 命令，中でも Load/Store 命令が利用される頻度が高くなっている．MBP-light は命令メモリとデータメモリは分離していないが，これらを分離することができた場合，33%の性能向上を得られることが分かった．したがって，Buffer-Register Architecture をとるかわりに，命令メモリとデータメモリを分離したオンチップキャッシュの搭載や，パケットキューを管理する特殊命令を備える等の手法を採用した場合，より高性能なメモリ管理プロセッサを構成することができる可能性がある．

また，これらの手法は，CC-NUMA 型並列計算機に限らず，並列分散システムの通信制御コントローラへの応用が期待できると考えられる．

第6章　結合網RDT(Recursive Diagonal Torus)

本研究では，JUMP-1 の結合網である RDT(Recursive Diagonal Torus) ネットワークに実装された様々な高速化メカニズムについて，実機を用いた評価を行った．

本章では，まず JUMP-1 で用いられているディレクトリ方式である縮約階層 bitmap directory 方式について述べ，JUMP-1 の結合網である RDT の構成および RDT router chip について述べる．その後，筆者が担当した，RDT に実装された様々なメカニズムの評価結果について述べる．

6.1　縮約階層 bitmap directory 方式

本節では，JUMP-1 のディレクトリ管理に用いられている縮約階層 bitmap directory 方式 (Reduced Hierarchical Bitmap Directory 方式: RHBD 方式) [50, 51, 52] について述べる．

6.1.1　階層 bitmap directory 方式

キャッシュライン単位にディレクトリを管理し，無効化型プロトコルを用いる場合を考える．すると，無効化メッセージの宛先は通常 1 ないし 2 がほとんどであるといわれている [53]．この結果を受けて，limited directory 方式 [53, 4] や chained directory 方式 [54, 55] などのディレクトリ構成方式が考案された．前者は共有しているプロセッサ数が一定の数まで，それらのプロセッサを指し示すディレクトリをキャッシュのページもしくはラインに持たせるものである．そして，共有するプロセッサの数がディレクトリ数を超えないようにするか，超えた場合にはブロードキャストに切り替える方法である．また，後者はディレクトリをリスト構造で持ち，任意の数のプロセッサを示すことができるようにしている．

これに対して，ページ単位でディレクトリを管理し，更新型のプロトコルを使用することのできる JUMP-1 では，ページを共有するノード数が大幅に増加する可能性があり，必要となるディレクトリ数も増えると考えられる．ここでは fullmap 方式等に比べて拡張性に富み，COMA machine などで用いられている階層 bitmap directory 方式 [56] について説明する．この方式が JUMP-1 のディレクトリ管理に用いられている RHBD 方式の基礎となっている．

まず，クラスタが葉に相当する木構造のネットワークを想定する．ここで，木の根からパケットを供給して，同一のパケットを複数のクラスタにマルチキャストする．木構造のネットワークであるから，各節において送信先の葉が末端にある枝にのみパケットを送れ

ば，必要なクラスタにのみパケットが届くことになる．図 6.1 に三進木の根から **d** で記された葉にパケットを送る場合を示す．このとき，各節に 3 bit のビットマップを与えることにより送信先を完全に指定できる．階層 bitmap directory はマルチキャストを行うためのビットマップであり，これによって同一の経路を同一のパケットが複数回通ることがなくなる．

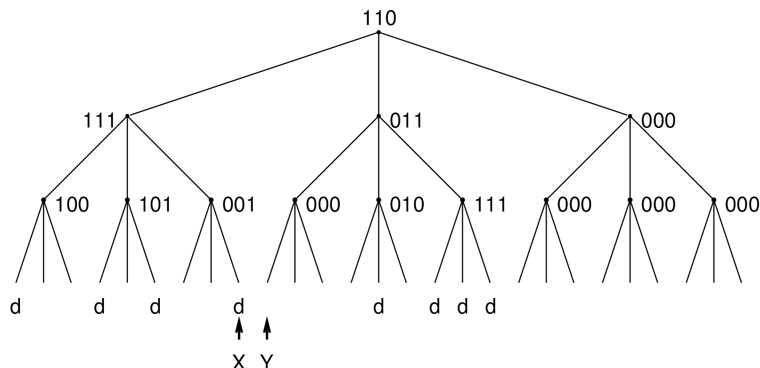


図 6.1 階層 bitmap directory 方式

なお，単純な木構造では木の根付近のトラフィックが大きくなってしまうため，結合網は fat-tree 構造を内包することが望ましい．また，パケットは実際にはいずれかのクラスタから送信されるため，送信元のクラスタから木の根に容易にパケットを送信できなければならない．後に説明するように，JUMP-1 の結合網はこれらの性質を満たしており，階層 bitmap directory 方式に適した構造を持っている．

6.1.2　RHBD 方式

階層 bitmap directory において必要なビット数は m 階層の n 進木において

$$\sum_{k=1}^{m} n^k$$

で与えられる．しかし，超並列計算機ではクラスタ数が非常に多い．よって，単純に階層 bitmap directory 方式を用いると，ディレクトリに必要なメモリ量が膨大になる．もちろん fullmap をもとに階層 bitmap を作成することは可能である．しかし，これには結合網のトポロジによって複雑な操作を必要とし，fullmap に必要なビット数も同様に膨大となる．階層 bitmap directory を用いた際に，階層毎にディレクトリを引き直すと，場合によっては大きな時間を要する．この問題を解決するために，東京大学の松本氏らが提案したのが疑似 fullmap 方式 [36] である．擬似 fullmap 方式は階層化マルチキャスト機構を前提としたディレクトリ方式である．この方式では，通信距離が近いクラスタは局所性を生かすために共有ページの存在を細かく管理し，通信距離が離れるにつれて粗く (マルチキャストの階層単位で) 管理する．この方式を基本として，ビット数を縮約する方法を以下に示す．

(1) いくつかのクラスタをグループ化して扱う．

(2) 各節において，その節で適用するディレクトリを管理し，パケットがその節に到達したときにディレクトリを引く．こうすれば，n 進木を用いたとき，各節において 1 つのディレクトリにつき，n bit のビットマップを管理すればよい．

JUMP-1 では前者について

- ある節以下はブロードキャストとする
- 複数の節で同一のビットマップを用いる

のいずれか，もしくは両方の組合せにより，階層ごとに n bit(n 進木で）のビットマップを 1 つだけ持つことにする．すると，m 階層の n 進木においてディレクトリごとに必要なビット数が $m \times n$ bit となる．さらに，ビットマップはパケットのヘッダ中に持つことができるので，完全に router 内のみでマルチキャストが可能である．また，外部のディレクトリも参照する必要がない．では，JUMP-1 で採用された 3 つのディレクトリ縮約方式について以下に説明する．

- **SM 法**

  各階層ごとに，その階層のすべての節の縮約前のビットマップの論理和をとり，その階層のすべての節で用いる．

- **LPRA 法**

  松本らが文献 [36] で提案した方法である．パケットは木の根から供給されるが，元々はいずれかの葉に相当するクラスタから送られたものである．根からこの送信元の葉に至る経路をその他の経路と区別して扱う．パケットは根から下位の節へビットマップに従って順にマルチキャストされる．しかし，ある節で送信元を含まない経路の枝にパケットが送られると，その枝の下の部分にはパケットがブロードキャストされる．すなわち，根から送信元の葉への経路上の節に各階層のビットマップが適用され，それ以外の節では上位の節からパケットが来ればブロードキャストを行う．文献 [36] では，パケットは送信元の葉から木を遡りながらマルチキャストを行うことになっているが，本質的に相違はない．

- **LARP 法**

  LPRA と同様に，根から送信元に至る経路をその他の経路とを区別して扱う．LPRA とは逆にある節で

  – 送信元を含む枝
  – 送信元を含まない枝のうちのいずれか

  の両方にパケットが送られると，送信元を含む枝の下の部分全体にパケットがブロードキャストされる．それ以外の枝では，同一階層のすべての節でそれらの節の縮約前のビットマップの論理和を用いる．

図 6.2 に 3 進木を用いた摸式図を示す．この図で **s** が送信元のクラスタ，**d** が本来の送り先，• が結果的にパケットが送り付けられるクラスタである．したがって，**d** でない • の

数が少ないほど無駄にパケットを受け取るクラスタ数が少ないことになる．この縮約方式を採り入れた階層 bitmap directory 方式を縮約階層 bitmap directory 方式 (RHBD 方式) と呼ぶ．

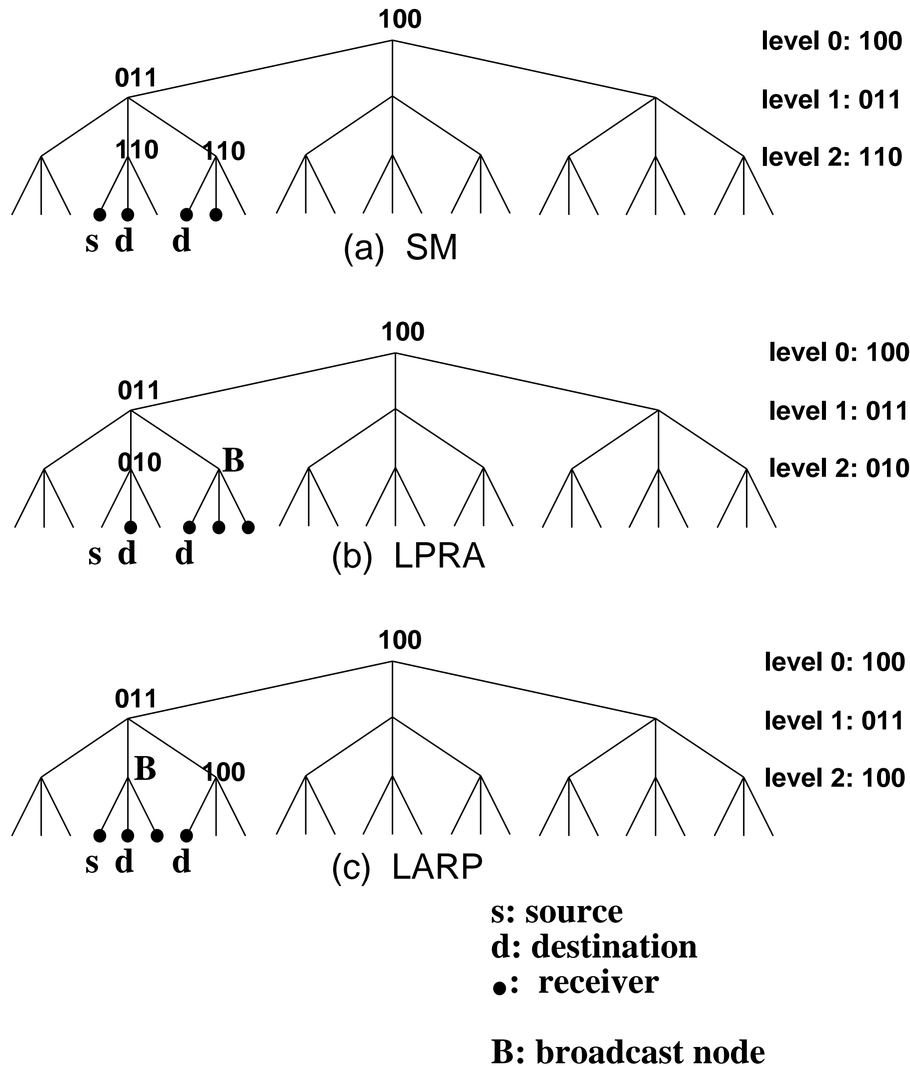


図 6.2 縮約階層 bitmap directory 方式

6.2　RDT ネットワーク

6.2.1　RDT の構成

RDT は基本のトーラス構造の上に目の粗いトーラスを 45 度ずつ傾けながら再帰的に積み上げていくことにより，トーラス構造と階層構造の両方を満足させることを狙っている．図 6.3 に示すように，基本となるトーラス上に上下に ±2 ずつ離れた点同志を結んだ rank 1 のトーラスを作る．さらに，この rank 1 のトーラス上で上下に ±2 ずつ離れた点同士を

結んで rank 2 のトーラスを作る．結果として，これは 8 ずつ離れたトーラスとなる．この操作を全ての点について上位トーラスが形成できないようになるまで繰り返す．このようにして形成されたものを完全 RDT と呼ぶ．しかし，完全 RDT は rank 数の 4 倍のリンク数をノード当りに必要とすることから，現実的な結合網ではない．そこで，各ノードは基本となるトーラスのほかにもつことのできる上位トーラスの数を 1 と定める．JUMP-1 では各 rank に対し平等で，どの rank へも 1 ステップの移動で利用可能な割付けを採用している．JUMP-1 設計時に考えられていたノード数 (16384 ノード) に対応した RDT を RDT(2,4,1)/α と呼び，この構成を図 6.4 に示す．

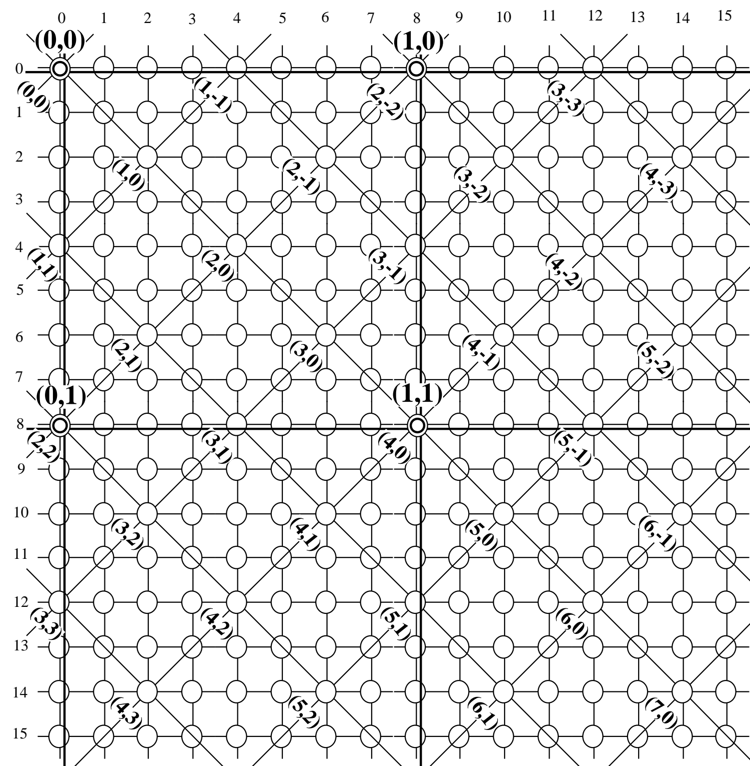


図 6.3 完全 RDT の構成

RDT の非常に大きな特徴は，理論的な結合網である完全 RDT と実装上の結合網を最初から分離して考えている点にある．このことは，結合網がわかりにくくなるという欠点を持っているが，実装上の構成が柔軟になる点では利点となる．現在目標としている JUMP-1 の構成は 1024 プロセッサ (256 ノード) である．すると，さきほどの RDT(2,4,1)/α の割当てに対し，例えば rank 4 をもつノードを rank 2 か rank 1 をもつノードへ，また rank 3 をもつノードを rank 1 をもつノードへと再割当てすることで対応可能となる．もしシステムを拡張する場合でも同様な再割当てを行うことにより，リンクの無駄は存在しない．そして，数百から数万と広いノード数の範囲でかなり効率の良い構成を取ることができる．

最後に RDT(2,4,1)/α 上での決定ルーティングの一つである e-cube routing について説明する．詳細は文献 [13, 57] を参照されたい．また，RHBD 方式を用いた場合のルーティングアルゴリズムは次節で述べる．

RDT(2,4,1)/α での e-cube routing では 4 種類の仮想 channel (CF, CW, BH, BL) を使

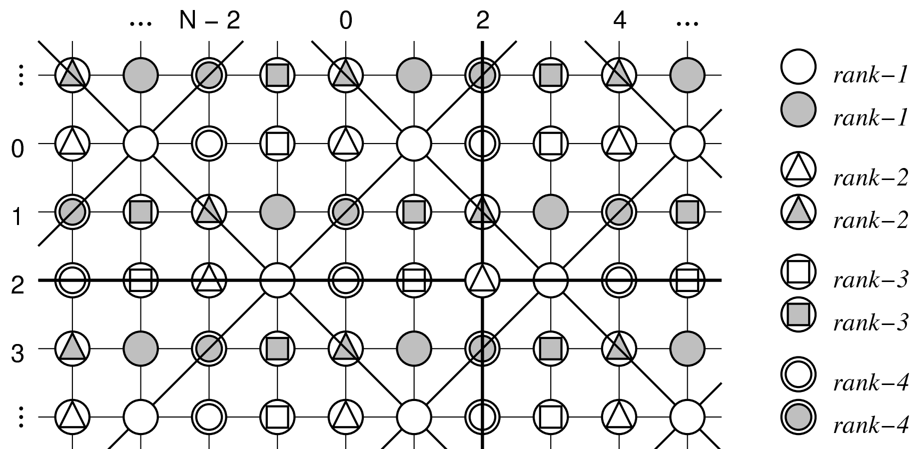


図 6.4 RDT(2,4,1)/α の構成

う．CF 及び CW の 2 種類の virtual channel の使い方は 2 次元トーラスにおける e-cube routing と全く同じである．つまり，各次元方向において通常は CF を使用し，round trip loop を用いる場合に CW を使用する．これに対し，BH 及び BL は rank を移動するために使う．BH は rank が上昇する方向のみに，BL は rank が下降する方向のみに用意されている．以上をふまえるとルーティングアルゴリズムは次のようになる．

(1) 現在のノードがもつ上位 rank がパケットのもつディレクトリの中の最大 rank でなければ，BH channel を使う．そして，その最大 rank をもつノードに到着するまで移動する．

(2) 最大 rank のトーラスにおいて，x 方向から y 方向の順にルーティングを行う．最大 rank のトーラスでのルーティング終了後，宛先に到着していなかったら，パケットがもつディレクトリの中で，次に最も rank が高いものと同じ rank をもつノードへ BL channel を使って移動する．ただし，次に高い rank が 0 の場合は rank の下降を行う必要はない．

(3) 最大 rank の場合と同様に，その rank のトーラス上で x 方向から y 方向の順にルーティングを行う．その rank でのルーティング終了後，宛先に到着していなかったら，さらに rank を下降させる．そして，(3) を繰り返す．

この様子を図 6.5 に示す．

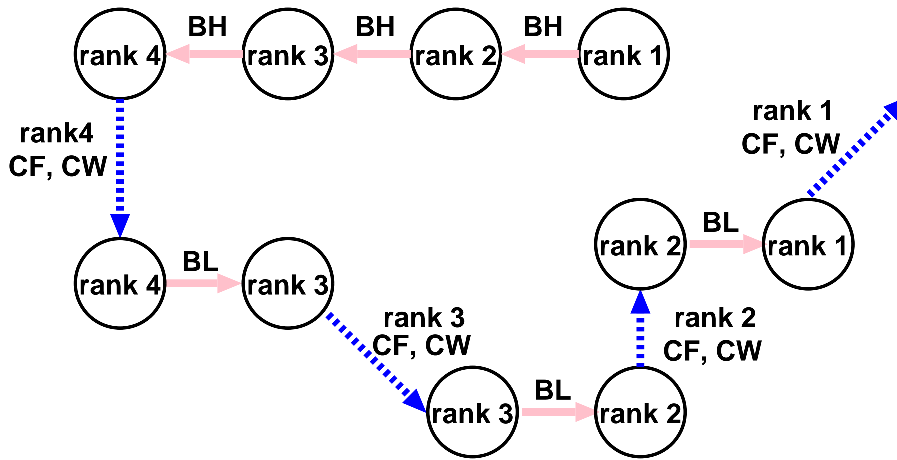


図 6.5 RDT(2,4,1)α 上での e-cube routing

6.2.2　RDT における RHBD 方式の実現

RHBD 方式を実現するために，RDT の持つ木構造のエミュレーション機能を利用する．以下，特に指定しない場合は完全 RDT を対象にアルゴリズムの解説を行うが，実際に JUMP-1 で用いる RDT(2,4,1)/α 用に拡張することは容易である．

RDT による木構造のエミュレーションの様子を図 6.6 に示す．これは，亀甲形の領域に対してのマルチキャストにより実現される．その操作は

(1) 4 隣すべてのノードにデータを送る．

(2) 1 つのノードが 3 つの特定方向にデータを送る．

という順に行う．つまり，転送を開始したノードを含めた疑似的な 8 進木を，2 段階で構成している．

ある rank i のトーラスを持つすべてのノードにおいて，この操作で転送を行った場合，この rank のトーラスを持つノードすべてにデータを送ることが可能となる．このパケットを受け取ったノードは，さらに rank $i-1$ のトーラスに対して同様な領域にマルチキャストを行う．このことにより各 rank に対して 8 進木を形成していくことができる．RDT は上位トーラスを複数持つため，結合網全体として fat-tree 構造 [58] が形成される．しかも，RDT(2,4,1)/α においては各ノードは 1 ステップで一番目の粗いトーラス (最大 rank) へ移動できる．つまり，木の根に対して直接パケットを転送することができる．このため，マルチキャストの開始時に木を遡る必要がない．この手法はトーラスやメッシュに限らず，様々な形態の結合網にも同様に適用できる [59]．

RDT(2,4,1)/α におけるマルチキャストは疑似的な 8 進木により行われるため，階層ビットマップを実現には各階層で 8 bit のビットマップが必要となる．ここで，基本転送操作に対し図 6.7 に示すビットマップを対応させる．

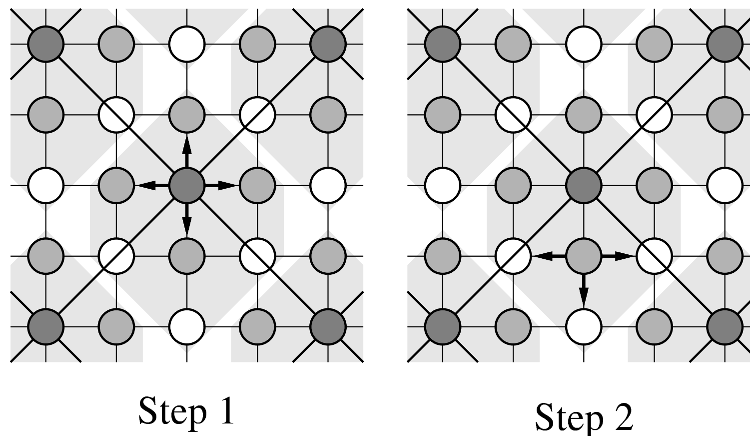


図 6.6 基本転送操作

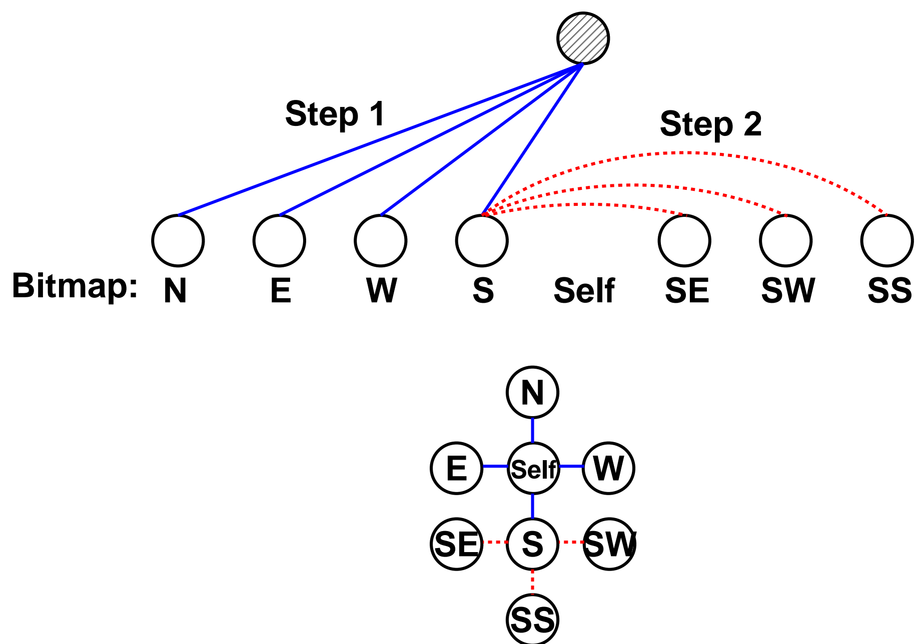


図 6.7 階層 bitmap directory 方式で用いる 8 進木とビットマップとの対応

あるノードが図 6.7 に示す操作で転送するとき，どの rank からマルチキャストするかによってパケットが届く範囲が定まる．図 6.8 に rank 1 の範囲を示す．rank が大きいほど範囲は広がっていき，マルチキャストを行うノードを中心とした不規則な同心円が形成される．

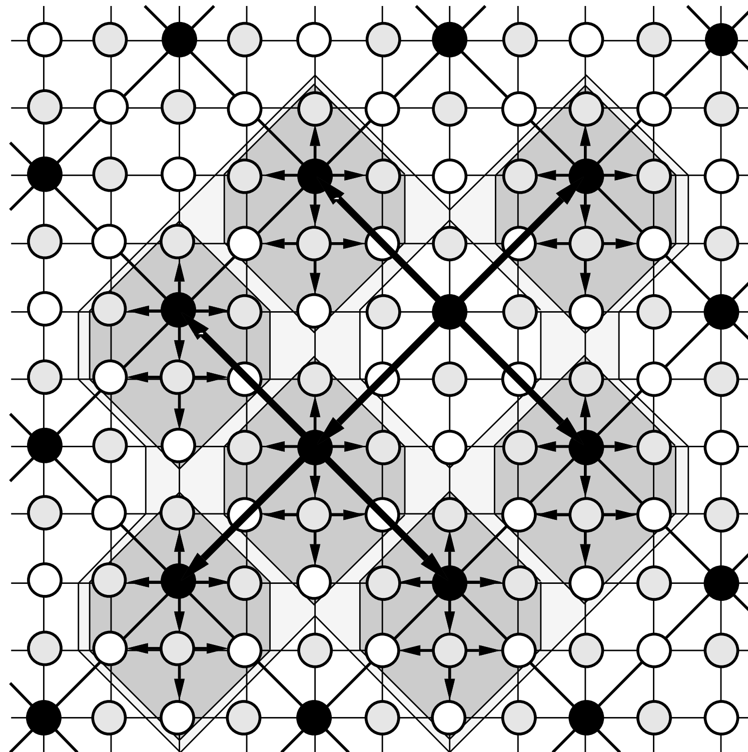


図 6.8 マルチキャストの範囲

この方法には単純な木構造や fat-tree 構造を持つ RDT に対して以下の利点がある．

- 上位トーラスが複数個存在するため，単純な木構造で問題になる木の根での混雑は起こらない．もし複数ノードでマルチキャストが起きても，通信負荷は自然に分散される．
- マルチキャストの範囲は送信ノードを中心とする同心円になる．よって，送信ノードの位置が変わると共にそれは移動する．このため，単純な木構造や fat-tree 構造のような近接ノード間の距離が極端に大きくなるといった問題は起きない．

しかし，この方法ではマルチキャストの範囲の形状が不規則となり，マルチキャストのためのビットマップを作ることが困難である．そのため，宛先ノードとの相対位置によりビットマップを格納した表を用意した方がよい．RDT(2,4,1)/α での計算法を以下に述べる．

まず，1 対 1 転送でのビットマップを作成することを基本とする．それには，

(1) 最大 rank 4 の完全 RDT でのビットマップを生成する．これは 8 進木をたどるだけでよいので，単純に行える．

(2) 送信元のノードが rank 1 の上位トーラスを持つのであれば，(1) のビットマップをそのまま使う．そうでなければ，そのノードを rank の移動を行ったときに到達する rank 1 の上位トーラスを持つノードと置き換えて考える．そして，置き換えたノードに対して (1) で計算されたビットマップを送信元のノードで使う．

といった手順で計算される．そして，マルチキャスト用のビットマップを生成するために，先ほど述べた 3 種類のいずれかの縮約を行う．

(1) SM 法

すべての宛先に対する 1 対 1 転送用のビットマップを各 rank 毎に論理和をとる．

(2) LPRA 法

それぞれの宛先に対する 1 対 1 転送用ビットマップに図 6.7 中の Self があるかどうかを最大 rank から検索する．もしあれば，Self である rank の次の rank 以降の 1 対 1 転送用ビットマップを消去する (0 にする)．この作業を経た後にできたすべての宛先に対する 1 対 1 転送用ビットマップを各 rank 毎に論理和をとる．

(3) LARP 法

(a) マルチキャスト用のビットマップを最初に作るときであれば，その 1 対 1 転送用ビットマップをマルチキャスト用のビットマップとする．

(b) そうでなければ，マルチキャスト用のビットマップに図 6.7 中の Self があるかどうかを最大 rank から検索する．その rank を r とする (なければ，0 となる)．また，1 対 1 転送用のビットマップに対しても同様のことを行う．その rank を k とする．

(c) もし $r < k$ であれば，rank k での 1 対 1 転送用のビットマップと rank k でのマルチキャスト用のビットマップの論理和をとる．それから，rank $k-1$ から 0 までの 1 対 1 転送用のビットマップを対応するマルチキャスト用のビットマップに代入する．

(d) もし $r > k$ であれば，rank r でのマルチキャスト用のビットマップを Self に設定する．

(e) もし $r = k$ であれば，rank k から 0 までの 1 対 1 転送用のビットマップと対応するマルチキャスト用のビットマップとの論理和をとる．

(f) あとはこれをすべての宛先に対して繰り返す．

6.2.3　RDT router chip

JUMP-1 で用いられている RDT router chip の構成を図 6.9 に示す．以下，RDT router と呼ぶことにする．詳細は文献 [28, 29] を参照のこと．

RDT router には rank 0 に 4 本，上位 rank4 本，JUMP-1 の設計当時に MBP はクラスタ内に 2 つあったためにそれぞれに 1 本ずつで 2 本のリンク必要である．よって，全体は 10 入力 10 出力のクロスバが基本になる．さらに，後に述べる応答パケット収集機能が加

わるため，クロスバは11入力10出力となる．また，転送のビット幅は1 chip当たり各リンクについて18 bitである．JUMP-1では2個のrouter chipをbit sliceに用いることにより，全体で36 bit幅を実現する．転送周波数はプロセッサに同期した最大50MHzであり，システム全体が単一のクロックで動作する．ピンを有効に利用するため，JUMP-1の通信はすべて双方向線により行う．

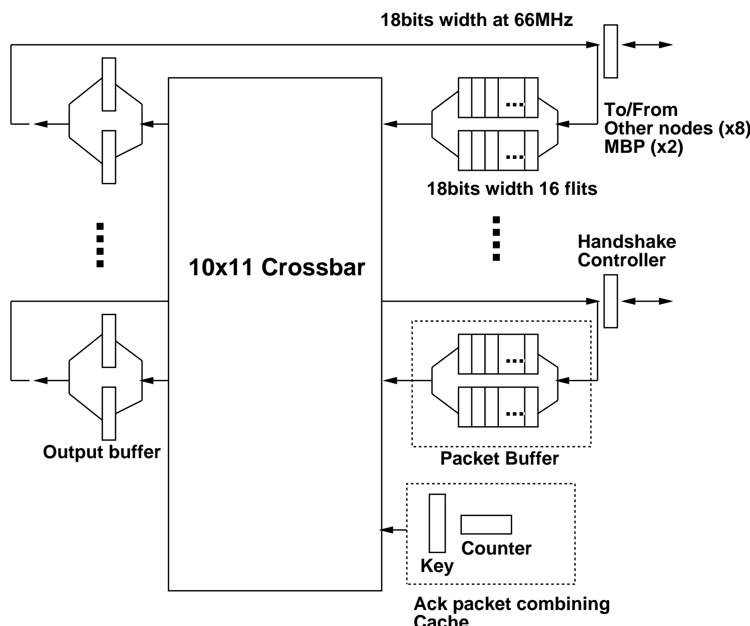


図 6.9 RDT router の基本構成

chipには日立の0.5μm BiCMOS gate arrayであるHG22S125を利用している．このgate arrayは高速である上，ECL入出力により転送線を直接駆動することができる．

JUMP-1でのネットワークパケットは最大16フリットであり，3フリットは2つのchipに対して共通の内容のヘッダとなる．また，通常用いられるパケットは最大16フリットの可変長のマルチキャストパケット(multicast packet)となる．他にも，3フリットのみの応答パケット(acknowledge packet)やMBP-lightとrouter間の制御に用いられるcommand packet，status packet，shootdown packet，setup packet等が存在する．マルチキャストパケットのヘッダにはそれぞれの階層でのマルチキャスト用のビットマップが格納されている．このビットマップは，次に使用するビットマップがrouter内で常に自動的に先頭になるようにシフトされる．パケットの詳細は付録Aを参照のこと．

転送方式はasynchronous wormhole方式[60]であり，各入力にはパケット1個分を格納することのできるバッファを2組持ち，それぞれが1つのチャネルを構成する．このチャネルを使い分けることにより，deadlock-freeな階層マルチキャストを実現することができる．RDT routerでは1対1転送は階層マルチキャストの一部として実現され，固定かつdeadlock-free，FIFO性の保証された転送となる．

RDT routerの実装上の特徴を次に示す．

- クロスバと双方向線の同時 1 クロックのアービトレーションによる高速な経路スイッチ
- 可能な転送から順に行う効率の良いマルチキャスト
- SM 法や LARP 法，LPRA 法の三種類の階層マルチキャストのサポート
- 応答パケットのルーティングと収集用キャッシュ
- FIFO 性を保証した shootdown と setup の装備

これらの点に着目して，RDT router の構造について述べる．

6.2.3.1　アービトレーションとクロスバの設定

RDT router はクロスバのアービトレーションと双方向線のアービトレーションを同時に行うことによって高速化を実現している．各線は以下の表 6.1 に示すようなハンドシェーク線をもつ．

表 6.1 RDT router における hand shake 線

hand shake 線	意味
reqout	自分の転送要求が存在する
reqin	相手の転送要求が存在する
chan0rdyout	自分の入力チャネル 0 が空いている
chan1rdyout	自分の入力チャネル 1 が空いている
chan0rdyin	相手の入力チャネル 0 が空いている
chan1rdyin	相手の入力チャネル 1 が空いている

パケットが存在する入力バッファはチャネル 0 と 1 の ready 信号を監視し，空いている場合のみ reqout をアサートする．この判断に 1 クロックを要する．同時に，クロスバに対してもアービトレーション要求を出す．そして，次のクロックの立ち上がりでクロスバのアービトレーションに成功する．また，相手からの reqin がアサートされていなければ (優先順位が高い場合はアクティブになっていても)，即座にクロスバを設定する．そして，reqout をネゲートすると共に，そのクロックで転送を開始してしまう．クロスバのアービトレーションに負けた場合でも，他の入力からのパケットが線を通過するため，転送線の制御自体に矛盾を生じることはない．

相手チャネルのバッファが空いていない場合はそもそも要求自体発生されない．よって，要求を出したのに受け付けられない状況は，転送線またはクロスバのアービトレーションに敗れた場合以外ない．クロスバの優先順位は擬似 round robin であり，転送線の優先順位は転送毎に切り替わる．要求は短時間の待ち時間で満足されるはずである．したがって，パケットの存在する入力バッファは相手バッファが塞がれない限り，要求が満足されるまで要求信号を出しつづける．転送が終了する 1 つ前のクロックで現在転送中のパケットはクロスバ及び転送線を開放し，このクロックで次の転送が決定する．このことにより，パケット間に生じる転送の空白を最小化している．

6.2.3.2　パケットのマルチキャスト

パケットのマルチキャストは送るべき相手先のバッファがすべて空くまで待って，一気に行うのが最も簡単な方法である．しかし，この方法はマルチキャストを行うことができる機会が減り，バッファとリンクの利用効率が極端に悪くなる．

そこで，RDT router は出力バッファの制御部で複数の宛て先をビットマップの形で持っている．ここで，複数の宛て先に転送を行う場合について考える．バッファが空いている相手チャネルが 1 つ以上 (複数でもよい) あれば，それらの相手に対してマルチキャストを行う．そして，転送開始と共に対応するビットを消去する．また，すべてのビットを消去できる場合は次のパケットの入力要求の転送を開始する．

図 6.10 に各バッファの構成を示す．バッファ本体は読出しと書込みが同一サイクルで可能な dual-port memory を用いている．asynchronous wormhole の実現のため，各チャネルのバッファの大きさは最大パケット長である 16 フリットとなる．このため，パケットの入出力は一度開始したら，停止することなく並行に実行できる．

6.2.3.3　multicast map の生成

先に示した階層マルチキャストの 3 つの方法での無駄パケット数は稼働する並列プログラムの性質に依存して変化する．JUMP-1 は超並列計算機のプロトタイプとして様々な実験に用いることが期待される．よって，RDT router はこの 3 種類の階層マルチキャストをすべてサポートすることが要求される．RDT router ではパケットのヘッダのビットにより階層マルチキャストの機能を切り替えることができるようになっている．

図 6.10 に示すように，バッファに入力する前段部に bitmap generator が設置されている．パケットが入力されると，ヘッダ内のマルチキャスト用ビットマップと入力ポート番号及び router 自体の位置情報から，この router で行われるマルチキャストに必要なビットマップが生成される．そして，各バッファの出力 bitmap register に送られる．この際，ヘッダの階層マルチキャスト機能を指定するビットが検査される．そこで，もしマルチキャストする場合にはすべての出力ポートに対するビットが設定される．そうでない場合にはパケット中のマルチキャスト用のビットを利用し，ビットマップを生成する．

この生成部は 3 種類の方式を実現するため複雑である．しかし，完全に組合せ回路で実装されており，タイミング的な問題や制御上の複雑さは存在しない．RDT router の特徴は，順序回路的な複雑度の高い hand shake やアービトレーション制御及びマルチキャスト制御と，組合せ回路的な複雑度の高いビットマップ生成部を完全に分離したことにある．この設計方針により，両者を独立に検査することができ，全体で複雑度の高いシステムを確実に検証することができた．

RDT router では 3 種類の方式の転送の他に，周辺ノードへの local multicast 機能や各階層のマルチキャスト開始ノードで MBP-light を介してビットマップ表を引き直す機能がサポートされている．前者は，科学技術計算でよく用いられるメッシュ構造の周辺のみに同一データを送る転送で有効となる．また，後者は 3 種類の方式のどれを用いても無駄パケットが多過ぎる場合には枝苅りを行うことができ，階層 chained directory との併用が可能になる．

ビットマップの生成は基本的に 1 クロックで行われる．しかし，header の 2 フリット目

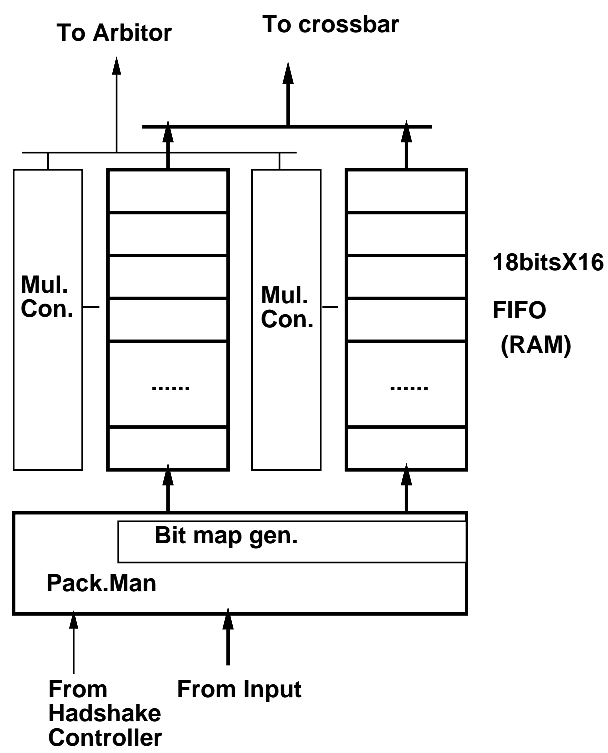


図 6.10 入力バッファの構成

や 3 フリット目の情報を必要とする場合はそれらの到着を待つために，2 クロックや 3 クロックを必要とする．このため，パケットの通過時間は相手側のバッファの空きを判断する 1 クロック，アービトレーション用の 1 クロック，クロスバを通過して出力バッファへ入るまでの 1 クロック，ラインを通過し接続されている相手ルータの入力に転送されるのに要する 1 クロックをすべて足し合わせると，最小で 5 クロック，最大で 7 クロックとなる．先に述べたように，ヘッダのマルチキャスト用ビットマップは常に次に用いるビットマップが先頭に来るようにルータ内でシフトされる．よって，全体で 7 クロックを要する事態はきわめてまれとなる．

6.2.3.4　応答パケットの収集

キャッシュの一貫性の維持のために階層マルチキャストを行った場合を考える．一貫性制御プロトコルによっては，すべてのノードがそのパケットを受けとったことを送信元のノードは知る必要がある．JUMP-1 においては，この操作は応答パケットを送信元に送り返すことによって行う．しかし，送られたノードが一対一通信で応答パケットを送信元に返すと，ネットワークの負荷が増大してしまう．そこで，マルチキャストした階層をそのまま利用して，応答パケットを収集する方法が提案されている [45, 36].

この収集は各階層のマルチキャストの開始元の MBP-light が行う．しかし，マルチキャストが頻繁だと応答パケットの収集により，router と MBP-light 間がボトルネックになる [47]．このため，RDT router では応答収集のためにパケット 1 個分のキャッシュを持つ．このキャッシュは基本のトーラスに対する応答パケットのみに機能する．

基本のトーラスに対するマルチキャストを行うときにキャッシュが空いていれば，そのパケットの識別子をキャッシュの鍵とする．そして，マルチキャスト用ビットマップ中の 1 の数を数えて，それをキャッシュ中のカウンタに設定する．これに対応する応答パケットはパケットの識別子を鍵としてキャッシュを検索する．もしヒットすれば，カウンタをデクリメントする．この検索及びカウンタの減算は同時にパケットが到着した場合でも，遅れなしで並列に実行することができる．カウンタの値が 0 になると，応答パケットの収集キャッシュは上位階層のノードに対して応答パケットを送る．そして，そのエントリを空にする．

次に，識別子がヒットしなかった場合やエントリが空でないのに新たなマルチキャストが行われた場合について考える．応答パケット及びマルチキャストパケットは MBP-light に送られ，応答パケットの収集は MBP-light で行われる．RDT router はハードウェアの制約によって基本トーラスにのみ，それも 1 パケット分しか応答パケットの収集キャッシュを持たない．しかし，応答パケットはマルチキャストパケット到着時に通常即時に返されるので，この方法は少量のハードウェアで最も大きな効果が期待される．なお，この効果を確認するため，RDT router ではパケット単位で応答パケットの収集キャッシュを用いるかどうかを選択することができる．よって，実際の並列プログラムの動作時にこの点に関して評価をとることも可能となっている．

6.2.3.5　パケットの shootdown と setup

shootdown と setup はタスクの切替えやエラーの検出，デバッグの際に RDT router 内のバッファから強制的にパケットを排出または挿入する機能である．

RDT router は MBP-light から shootdown 要求を受けるか，転送エラーを検出した場合，shootdown 信号が出力される．また，RDT router は全ノードの RDT router をカスケード接続する簡単なバリア同期線を持っている．よって，このバリア同期線を用いても shootdown を起動することができる．この信号を検出した RDT router は router 内のすべての転送が終了して一定の遅延をおいた後，shootdown mode になる．このモードに入ると，各バッファは現在のバッファの位置とマルチキャストのビットマップを先頭に付加した shootdown 用パケットを生成する．そして，それを MBP-light に対して転送する．また，すべてのバッファが空になると，RDT router は今度は setup mode になる．このモードでは，MBP-light を除く各リンクの出力は RDT router 内部で自分の入力に繋がれる．このフィードバックを介して，shootdown 用パケットが順番に MBP-light より RDT router へ挿入される．最後に MBP-light が setup mode を解除すると，バッファの内容は shootdown 時と全く同じ状態から転送が再開される．以上の方法で，FIFO 性を守った shootdown と setup が可能である．

6.2.3.6　エラーの検出

RDT router は転送制御に必要なヘッダの 3 フリットのみについてパリティ検査を行う．転送性能を阻害しないため，検査はそのヘッダが実際に用いられる場合にのみ行われる．パリティ誤りを検出した場合，RDT router は shootdown 状態となり，MBP-light に対して status packet を送る．これで，パリティ謝りの発生を MBP-light に通知することができる．同時に，バリア同期線を用いて他のノードの RDT router を shootdown させることも可能である．パケットの他の部分の転送エラーは miss routing を生じない．ここでのエラー発生は，宛先と送信元の MBP-light 相互間の検査により検出し，再送を行う．

RDT router は bit slice で用いられ，2 つの router chip が同一動作する．このことを確認するため，RDT router 内の全バッファの状態のパリティを生成して，常に相互比較を行っている．2 つの router chip のパリティが異なった状態を示したときに，一致エラーを発生する．そして，パリティ誤り同様に shootdown を引き起こすことができる．

さらに，RDT router はタイムアウト機能を持っている．本来，RDT router はチャネルの設定を適切に行えば，デッドロックは生じない．しかし，設定時のエラー以外に，何らかの原因でデッドロックが生じる可能性がある．このことを防ぐため，すべてのバッファはタイムアウト機能を持っている．そして，一定時間以上バッファに滞在する場合，自動的に MBP-light に対してパケットを shootdown させることができる．この場合に，shootdown されるパケットはタイムアウトしたバッファのものだけであり，他のバッファのパケットは通常通りの転送が行われる．

以上の誤り制御機能はすべて MBP-light からの command packet により on/off することができる．そして，shootdown を引き起こすかどうかも選択可能である．

6.2.3.7　RDT router の実装

RDT router は制御が簡単で，高速性を要求されるクロスバ部やアービタ部は schematic editor によりゲートレベルの設計を行い，他の部分はハードウェア記述言語 (VHDL) で記述している．総ゲート数は約 9 万程度であり，表 6.2 にその内訳を示す．そして，実装諸元を簡単に表 6.3 にまとめる．

表 6.2 RDT router のゲート数

部分名称	ゲート数	個数	総計	記述
クロスバ	2927	1	2927	Schematic
アービタ	2736	1	2736	Schematic
マルチキャスト制御	1558	10	15580	VHDL
出力制御	397	10	3970	VHDL
ビットマップ制御	2288	10	22880	VHDL
応答パケット収集キャッシュ	2009	1	2009	VHDL
buffer RAM	2021	20	40420	RAM

表 6.3 RDT router の実装諸元

最大周波数	60 MHz
消費電力	19.4 W
ゲート使用率	63 %
ピン数	299
パッケージ	PGA

最後に，RDT router chip の写真を図 6.11 に，RDT backplane board の写真を図 6.12 に示す．図 6.12 には 1 クラスタに 2 個の RDT router chip が用いられ，全部で 8 個の RDT router chip が載ることになる．このボード内には rank 0 の 2 つのリンクを用いて，4 ノードからなるメッシュが実装されている．そして，残りの rank 0 の 2 つのリンクと上位 rank の 4 つのリンクのためにコネクタが用意されており，それらをケーブルでつなぐという構成になっている．

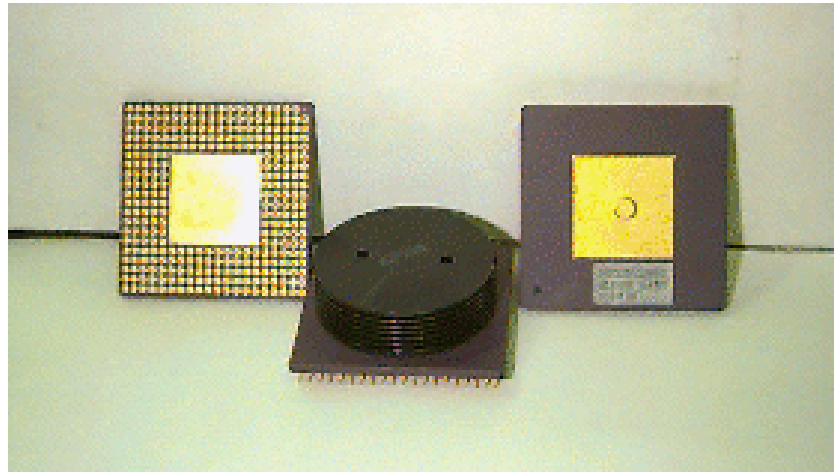


図 6.11 RDT router chip の写真

図 6.12 RDT backplane board の写真

6.3　RDT ネットワークの評価

本節では，本研究において筆者が担当した，RDT の様々な機構についての評価結果を示す．評価項目は下記のとおりである．

- RDT ネットワークの最大バンド幅
- RDT ネットワークの特徴を利用した，マルチキャスト機構の評価
- 応答パケットの自動生成機構 (無駄パケット自動廃棄機構) の評価
- 応答パケットの自動収集機構の評価

6.3.1　評価に使用したシステムの構成

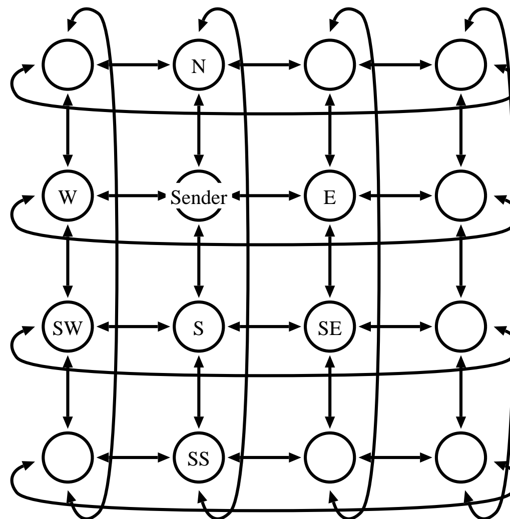


図 6.13 16 クラスタにおける RDT 結線図

評価に用いたのは，16 クラスタ (ノード)64 プロセッサ構成のシステムである．このシステムでは，rank 0 の基本となるトーラスに加えて，rank 1 の上位トーラスが形成されている．

図 6.13 では rank 0 のトーラスのみが描かれているが，rank 0 トーラスにおいて縦に 2 つ，横に 2 つ離れたクラスタ間に、rank 1 のネットワークが存在する．

6.3.2　RDT ネットワークの最大バンド幅

特殊機構の評価に先立ち，JUMP-1 の転送バンド幅の絶対性能の評価を行った．図 6.14 は，RDT ネットワークの単一リンクにおいて，最大バンド幅をパケット長を変化させて測定した結果を示す．ここでのバンド幅は，パケットのヘッダ部等は含まず，有効なデータの転送量から得られる実効バンド幅である．

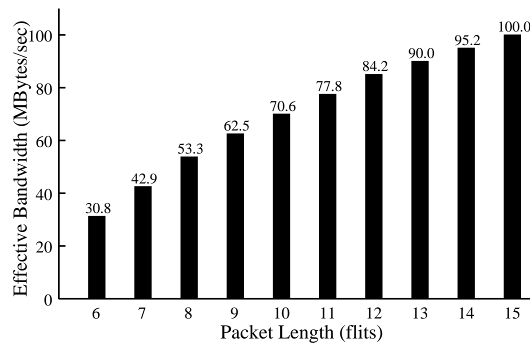


図 6.14 単一リンクにおける RDT ネットワークのバンド幅

バンド幅の測定では，送信側クラスタから隣接する単一のクラスタに対して多数のパケットを送信する．この際，送信バッファに 1 つでも空きがあればパケットを投入し，常にリンクが通信状態になるようにする．この状態で，パケット 1 個あたりの送信に要するサイクル数を測定し，このサイクル数とパケット中の実効データの割合から，バンド幅を求めている．

この結果，最大のパケット長である 15 フリット*のパケットの場合，100M Byte/sec のバンド幅となる．6 フリットのパケットの場合は，最大バンド幅の 3 割程度である 30.8M Byte/sec となる．

6.3.3　マルチキャスト機構

6.3.3.1　マルチキャストの送信処理時間

ここでは，複数の宛先に対してパケットを送信する際に，マルチキャストを用いる場合とユニキャストを複数回行う場合についての処理時間を比較する．この測定には 16 クラスタのシステムを使用した．

RDT のマルチキャストパケットは，JUMP-1 のディレクトリ構成方式である RHBD のビットマップをそのまま宛先ビットマップとして利用することができる．そのため，マルチキャストを行う場合はパケット固有の情報とディレクトリのビットマップをそのまま登録することで送信している．

一方，ユニキャストを行う場合はディレクトリのビットをそれぞれ検査し，セットされていた場合には該当する宛先に対応するビットマップを生成し，送信を行うという作業を繰り返している．

今回の評価では，このような作業を行う場合に MBP-light の Core が必要とするサイクル数を測定した．

マルチキャストを行った場合は一度パケットを送信するのみなので，サイクル数は宛先数によらず一定で 17 サイクルである．ユニキャストによる場合のサイクル数を図 6.15 に示す．

まず宛先が 1 か所である場合について検討する．先に述べたように，マルチキャストパ

*うち 4 フリットはヘッダ部と識別子．1 フリットの有効データ長は 32 bit(4 Byte).

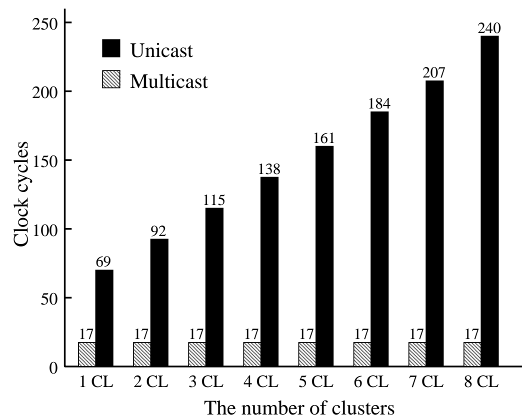


図 6.15 送信処理に必要なサイクル数

ケットの宛先ビットマップと，JUMP-1 のディレクトリのビットマップは一致するため，マルチキャストを行う場合はそのままパケットのヘッダに登録すればよく，17 サイクルで処理が終了している．一方，ユニキャストの場合には，ディレクトリの各ビットがセットされているかどうかを判断し，セットされていた場合にその宛先に対応する宛先ビットマップを生成し，パケットのヘッダに登録するという作業が必要であるため，処理には 69 サイクルかかっている．

また，更に宛先が増加した場合でも，マルチキャストの場合は 17 サイクルで処理が終了するが，ユニキャストの場合は更にサイクル数が増加していくことになる．

6.3.3.2　ネットワークの飽和

図 6.16 は，4 クラスタのシステムにおいて，マルチキャストを行った場合とユニキャストによって複数のクラスタに同一のパケットを送信した場合で，ネットワークが飽和する度合にどのような違いがあるか測定したものである．

この測定では，各クラスタが他の 3 つのクラスタ宛てに，E，S，SE 方向 (図 6.13 の右，下，右下方向を指す) にパケットを送信する作業を，パケットの送信間隔を変えて比較している[†]．各クラスタは 15 フリット長のパケットを 256 個送信する．そして，送信開始から実際に全てのパケットの送信が終了するまでのサイクル数を各クラスタで測定し，その合計を示したものが図 6.16 である．なお，パケットを送信する間隔は 16 の整数倍とした．今回の評価では，パケット送信の準備に要する時間の影響を避けるため，送信時にはヘッダ生成処理等を行わず，同一のヘッダを何度もコピーして使用することにより，ユニキャスト，マルチキャストともに評価の最短の送信間隔である 16 サイクル以内に送信処理が行われるようにしている．

この結果，送信間隔が 128 サイクルより大きい場合はマルチキャスト，ユニキャストともに問題なく送信できる．しかし，ユニキャストの場合は 128 サイクルよりも送信間隔を短くすると，ネットワークが飽和して実際の送信に必要となる時間は短くならない．それに対してマルチキャストの場合は，80 サイクルまでは送信間隔の減少に伴ない，その分処

[†]4 クラスタ構成 (単純なトーラス) で測定したため，各クラスタ，リンクには同数のパケットが送られる．

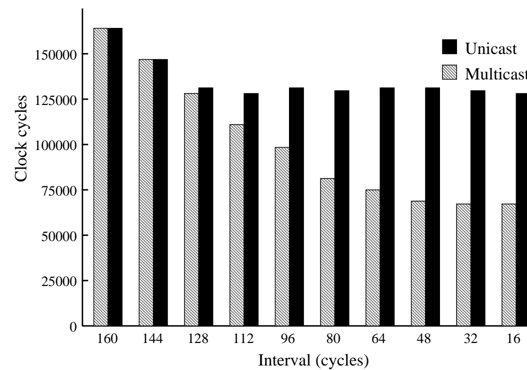


図 6.16 ネットワークの飽和の比較

理の終了までの時間が短くなる．それより送信間隔が短くなった場合でも，パケットの衝突による効果が現れてはいるものの，送信間隔 48 サイクルまでは処理時間を減少させることができる．

6.3.3.3　マルチキャスト機構のハードウェア要求量の評価

RDT ルータでは，それぞれの階層でマルチキャスト先を示すビットマップをチェックし，ビットマップが 1 になっている転送先の入力バッファが空いていれば，同時に転送を行う．終了後，対応するビットマップを 0 にし，全てのビットマップが 0 になるまで繰り返す．この操作の制御には，1558 ゲートを要しており，10 入力の全てにこの機構を持つために，全体として 15580 ゲートが必要となる．このゲート数は，ルータ全体 (90522 ゲート相当) の約 17%に相当する．

6.3.4　応答パケット自動生成機構 (無駄パケット自動廃棄機構) の評価

6.3.4.1　性能評価

応答パケット自動生成 (無駄パケット自動廃棄) 機構の効果を評価するため，自動生成機構を使用した場合と，自動生成機構と同様の操作を MBP-light の Core プロセッサによるソフトウェアで行った場合について，それぞれサイクル数を測定するという方法で行った．

評価の際，まずマルチキャストパケットを送信元のクラスタ (Sender クラスタ) から W のクラスタに向けて送信し，この送信処理開始時にクロックカウンタの動作を開始させる．受信側のノードでは応答パケットを Ack Generator で生成するか，MBP-light の Core プロセッサのソフトウェア処理によって生成し，Sender クラスタに送り返す．そして，Sender クラスタの MBP Core に応答パケットが届いたとき，カウンタの値を読み出す．

この結果を，表 6.4 に示す．また表 6.5 は，ソフトウェア処理した際に MBP Core が実行しているコードの内訳である．このように，応答パケットを自動生成した場合は，ソフトウェア処理した場合に比べて 40 サイクル短縮されており，約 1.8 倍高速に処理することができる．また，応答パケットを自動生成した場合，MBP-light の Core に対して割込みをかけないようにすることも可能であり，DSM プログラムで自動生成機構を利用した場

表 6.4 応答パケット返送に要するサイクル数

自動生成	51 サイクル
ソフトウェア処理	91 サイクル

合に MBP Core の負荷を低減するという効果も得ることができる．

6.3.4.2　ハードウェア量の評価

MBP-light 内の応答パケットの自動生成回路は，そのほとんどを Net Cache と Ackmap Cache が占める．Net Cache は，4 bit のタグと 14 bit のキーから構成されるエントリを 512 × 2 セット分持つ．全体のメモリ要求量は 18432 bit である．一方，Ackmap Cache は応答パケットの経路を保持するテーブルで 3 bit のタグと 16 bit の送り先のエントリを 256 持ち，全体のメモリ要求量は 4864 bit である．両者を合わせて MBP-light 全体の面積の約 5%に相当する．制御回路はパケット受信部に組み込まれており，分離して評価することは困難であるが，メモリの面積に対し無視できる程度に小さい．

6.3.5　応答パケット自動収集機構の評価

6.3.5.1　RDT ルータチップによる応答パケット自動収集機構の評価

16 クラスタのシステムのシステムにおいて，宛先が W クラスタ 1 か所のみの場合から，W，S，E，SW，SE，SS の最大 6 か所までのパターンのいくつかを対象とし，自動収集機構を使用した場合と，MBP-light の Core プロセッサのソフトウェアで同等の処理を行った場合について，それぞれのサイクル数を測定した．マルチキャストパケットの宛先で応答パケットを返送する際には，Ack Generator の応答パケット自動生成機構によるハードウェア処理を利用している．サイクル数のカウント開始は，Sender クラスタによるマルチキャストパケットの送信処理開始時とした．自動収集機構を利用した場合は収集が終わったことを示す応答パケットが MBP-light の Core に届いたときを処理の終了とし，ソフトウェア処理の場合はそれぞれの宛先からの応答パケットが全て届いたことを確認する処理が完了したところで処理の終了とした．

この結果を表 6.6 に示し，表 6.7 にソフトウェア処理した際に MBP Core が実行しているコードの内訳を示す．なお，表 6.7 には Core 以外のハードウェアによる処理の時間は

表 6.5 ソフトウェア処理による処理内容の内訳

レジスタ初期化等	9 サイクル
Ackmap Cache 読みだし	5 サイクル
応答パケットの作成	22 サイクル
その他	1 サイクル

表 6.6 応答パケット自動収集に要するサイクル数 (RDT ルータ)

宛先が W のみ	自動収集 ソフトウェア	51 サイクル 64 サイクル
宛先が W, S	自動収集 ソフトウェア	51 サイクル 84 サイクル
宛先が W, S, E	自動収集 ソフトウェア	51 サイクル 104 サイクル
宛先が W, S, SW	自動収集 ソフトウェア	63 サイクル 104 サイクル
宛先が W, S, E, SW	自動収集 ソフトウェア	63 サイクル 124 サイクル
宛先が W, S, E, SW, SE, SS	自動収集 ソフトウェア	71 サイクル 164 サイクル

含まない．まず，自動収集機構を利用した場合については，宛先が W クラスタ 1 か所のみ場合と W，S の 2 か所の場合，W と S，E の 3 か所の場合のように，隣接するクラスタにマルチキャストした場合ではサイクル数に差はなく，複数の宛先からの応答パケットの収集が同時に行われていることが分かる．また，51 サイクルという値は，6.3.4 節で示した応答パケットの自動返送の際のサイクル数と同一であり，オーバヘッドがないことが分かる．また，宛先が W，S，SW の 3 か所の場合，W，S，E，SW の 3 か所の場合は 63 サイクルであるが，6.3.4 節と同様のテストで，宛先を SW に変更した際には 63 サイクルとなる．これは，パケット転送のホップ数の増加のためであり，このことから，このような宛先となった場合でもオーバヘッドなしに収集処理を行うことができることが分かる．なお，宛先が W，S，E，SW，SE，SS の 6 か所の場合は少々サイクル数が増加し，71 サイクルとなる．

一方，ソフトウェア処理した場合，宛先が 1 か所のときはそれほど差がみられないが，宛先が増えるごとに 20 サイクル増加している．JUMP-1 では，各階層でのマルチキャストの最大数は 8 であるので，宛先が多い場合にはソフトウェア処理によるオーバヘッドは大きなものとなる．今回の結果によると，宛先が 6 か所になった場合では 2.3 倍程度の差が生じている．

表 6.7 ソフトウェア処理による処理内容の内訳 (Core 起動時間のみ)

パケット登録時	エントリが空かどうか エントリ登録	6 サイクル 5 サイクル
パケット収集時	エントリ読出し，一致検出 カウンタ減算，値の格納 その他	6 サイクル 7 サイクル 2 サイクル

6.3.5.2　MBP-light による応答パケット自動収集機構の評価

RDT ルータチップによる自動収集機構は効率が良いが，最下位階層 1 エントリに限定され，それを越える収集は MBP-light の自動収集機構により行われる．MBP-lignt による応答パケット自動収集機構の評価は，RDT ルータチップによるものと同様の条件 (16 クラスタシステム) で行った．ただし，MBP-light の Ack Cache は 128 エントリと 2 エントリの特殊な 2 way set associative であるが，Ack Collector の動作をソフトウェア処理するプログラムでは簡単のため，128 エントリの表のみを作成して処理している．

表 6.8 応答パケット自動収集に要するサイクル数 (MBP-light)

宛先が W のみ	自動収集	67 サイクル
	ソフトウェア	70 サイクル
宛先が W, S	自動収集	75 サイクル
	ソフトウェア	107 サイクル
宛先が W, S, E	自動収集	83 サイクル
	ソフトウェア	139 サイクル
宛先が W, S, E, N	自動収集	87 サイクル
	ソフトウェア	171 サイクル
宛先が W, S, E, N, SW	自動収集	95 サイクル
	ソフトウェア	203 サイクル
宛先が W, S, E, N, SW, SW	自動収集	103 サイクル
	ソフトウェア	235 サイクル
宛先が W, S, E, N, SW, SE, SS	自動収集	111 サイクル
	ソフトウェア	267 サイクル

表 6.9 ソフトウェア処理による処理内容の内訳 (Core 起動時間のみ)

パケット登録時	エントリが空かどうか	6 サイクル
	エントリ登録	7 サイクル
	その他	2 サイクル
パケット収集時	エントリ読出し，一致検出	12 サイクル
	カウンタ減算，値の格納	11 サイクル
	その他	2 サイクル

この場合のサイクル数を表 6.8 に示す．また，ソフトウェア処理による MBP Core が実行しているコードの内訳は表 6.9 である．なお，表 6.9 には Core 以外のハードウェアによる処理の時間は含まない．MBP-light の Ack Collector で応答パケットの自動収集を行った場合は，RDT ルータチップで収集した場合と異なりオーバヘッドが生じる．ただし，ソフトウェア処理した場合は宛先が増えると収集に要するサイクル数はかなり増加す

るが，ハードウェア処理では小規模にとどまっている．今回の結果では，宛先が 7 か所となった場合，ソフトウェア処理と自動収集した場合では約 2.4 倍の差が生じている．

また，ソフトウェア処理では 128 エントリの表のみを用いたが，実際の Ack Cache と同様の処理を行った場合は更にオーバヘッドが拡大することが予想される．

更に，MBP-light の Ack Cache は最下位階層用と上位階層用の 2 組存在するため，上位階層にも適用することができる．基本的に RDT ネットワークでは，マルチキャストや応答パケットの収集によって上位階層に行くほどパケット数が少なくなるため，実際の運用時の効果は最下位階層ほど大きくはないが，最下位階層と同様，各階層ごとに収集するパケット数に応じた性能差が生じることになる．

6.3.5.3　ハードウェア量の評価

RDT ルータチップ内の応答パケット収集回路は，1 エントリに限定したため，必要ゲートは，2009 ゲートであり，全体の 2%に過ぎない．

MBP-light 内の応答パケット収集回路は，応答パケットを収集するための Ack Cache がそのほとんどを占める．Ack Cache は 10 bit のタグと 4 bit のカウンタから成るエントリを 128+2 持つ変則 2-way 方式を取っている．全体のメモリ要求量は 1820 bit であり，MBP-light 全体の面積の 1%に満たない．

6.3.6　RDT の評価のまとめと他のシステムとの比較

JUMP-1 は，1990 年代前半の実装技術を利用しており，そのバンド幅 (100MByte/sec) はほぼ同時期に開発された CC-NUMA である Stanford FLASH が転送最大バンド幅 200-400MByte/sec を達成する [18] のに比べてやや劣っているが，MIT の Alewife[4] の 40MByte /sec よりは大きい．これはそれぞれのシステムで採用したノードプロセッサの性能 (JUMP-1: SuperSparc+/50MHz, FLASH: R10000 100MHz, Alewife SuperSparc/20MHz) を考えると，バランスが取れている値であると考えられる．一方，メッセージのマルチキャストと応答パケット収集機構に類似した機構としては，NUMA-Q[16] の SCI を利用したメッセージの巡回によるマルチキャストが挙げられる．しかし，この方法は単純なループ状のネットワークにメッセージが巡回する際のマルチキャストであり，その効果は評価されていない．また，Alewife のディレクトリの管理の一部にブロードキャストを用いる提案がなされたが，結局ソフトウェアによる方法が採用された [4]．このため，他のマシンの方法と性能を比較することはできないが，今回の評価では個々の操作における性能は大きく改善されており，これらの処理を頻繁に行うプロトコルまたはアプリケーションで有効と考えられる．また，それぞれの機構に要するハードウェア面積は，チップ全体から見れば数%内外で済んでいるものが多く，これらの機構はさほどハードウェアコストを必要としないことがわかる．また，現在では実装技術の進歩によってネットワーク性能等は大きく向上しているが，これらの機構は現在の計算機ネットワークにも適用可能であり，適用した場合に性能の向上に十分寄与することが考えられる．

第7章　メインテナンスシステム

本章では，JUMP-1の運用を行うメインテナンスシステムについて述べる．メインテナンスシステムは提案，実装を含め筆者が主体となって行ったため，実装の詳細まで含めて説明する．

7.1　メインテナンスシステムの方針

7.1.1　目的

メインテナンスシステム [61] は JUMP-1 における運転監視装置としての役割を担っており，大きく2つの機能を提供することを目的としている．

メインテナンス 電源投入後における各種資源 (メモリ，I/O，システム状態など) の初期化 (ブートアップ) およびシステム異常時における対処などの管理，監視を行う．

モニタリング (各種のデータ収集) ハードウェアに関する情報収集，ハードウェアエラーの検出，ソフトウェア開発時におけるデバッグを支援するための情報収集，性能評価に使用する動作統計データなどの情報収集および可視化を行う．

また，メインテナンスシステムはメインテナンスおよびモニタリングの機能だけでなく，JUMP-1 の開発段階におけるデバッグ支援システムとしても利用できるよう考慮しなければならない．以下にメインテナンスシステムが備えるべき機能を挙げる．

- 開発段階でのデバッグを支援できるように実装が容易かつ直ちに動作すること
- プロセッサの各種資源 (MBP-light など) にアクセス可能なこと
- 超並列計算機のスケーラビリティに対応できる実装形態を採れること
- 回路規模および価格の点で共にローコストであること
- 通常処理を妨げることなく情報収集を行うことができ，かつ可視化の機能を備えること

また，ホスト側と JUMP-1 側では，次のように要求仕様が大きく異なっている．

ホスト側: ホスト自体が動作している事が前提であり，JUMP-1 側への制御コマンドの発行が可能であればシンプルな構成でもよい．

JUMP-1 側: デバッグを行う段階では，JUMP-1 側には確実な動作が保証されているものがないことや，システム完成後であっても，ブート時には全てのシステムが停止している状態から開始するため，メインテナンスシステム自体に独立性が要求される．

7.1.2　提案

上記の目的を達成するため，7.1.2.1 節に示すような構成のメインテナンスシステムを提案した．

7.1.2.1　全体構成

メインテナンスシステムは図 7.1 に示すような構成となっている．各クラスタはクラスタボード上に配置される MBIF(Maintenance Bus InterFace) を介して，メインテナンス用の PC の ISA バスに接続されている MC(maintenance Controller) と接続される．MC は 4 つのコネクタを持ち，1 ボードで 4 クラスタまで接続することができる．JUMP-1 では，更に大規模なシステムを構築することを目的としている．そのため，更に多数のクラスタボードを接続する必要が生じてくる．その際には，メインテナンスシステムの複雑化をさけるために別の専用ハードウェアを設けることはせず，複数の PC を HUB で接続することによって対応することとする．

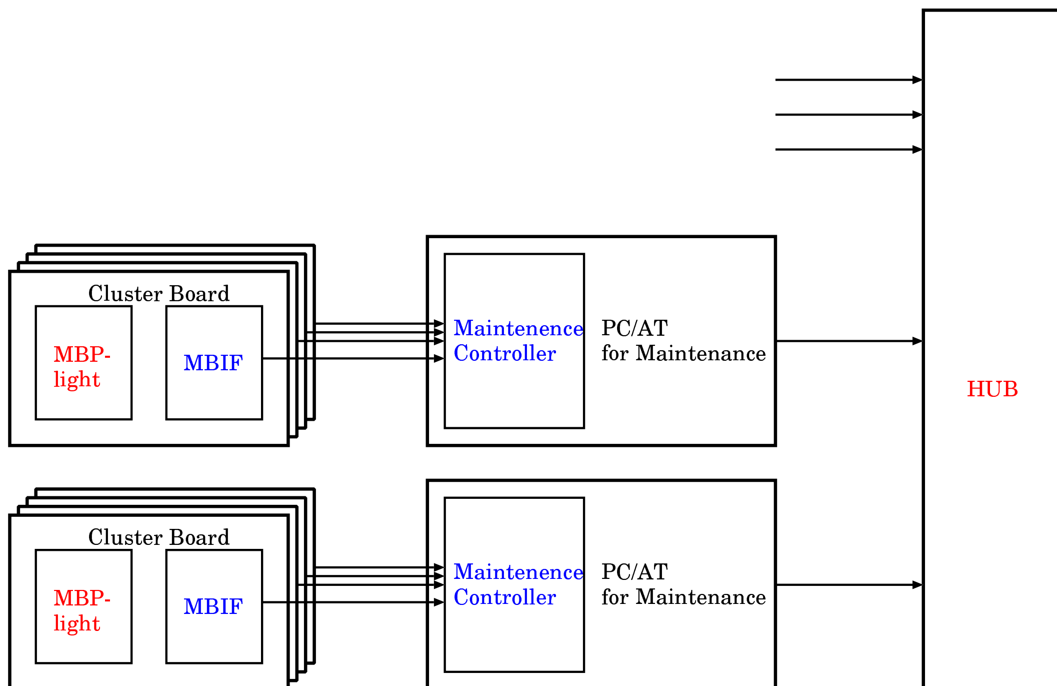


図 7.1 メインテナンスシステムの構成

7.1.2.2　MC(Maintenance Controller)

MC はメインテナンス用ホスト PC の ISA バス接続のボード上に実装される．MC はメインテナンス用 PC とのインタフェースおよび MBIF とのデータ転送を行う．MC 上には MBIF との転送プロトコルを制御する専用ハードウェアは設けず，プロトコル制御はメインテナンス用ホスト PC のソフトウェアによって制御することとする．このような構成に

することにより，MC のハードウェアを大幅に簡略化することができ，開発期間や実装コストを抑えることができる．

7.1.2.3 Maintenance Bus InterFace

MBIF は，メインテナンス用の PC と，クラスタボードを接続するためのインタフェースチップである．MBIF に要求される機能を以下に示す．

- ブート時には，JUMP-1 システムが停止しており，システムとは独立して動作する必要がある．
- MBP-light のリセット，リセット解除を行う．
- MBP-light のローカルメモリの初期化，プログラムのロードを行う．
- MBP-light 動作時に，MBP-light の I/O デバイスとして，メインテナンス用 PC との間でデータ転送を行う．
- PC 側から MBP-light への割り込みを制御する．

MBIF はこのような機能を満たすための専用ハードウェアという性質上，比較的複雑な回路を実装する必要がある．このため，クラスタボード上に FPGA を配置し，その FPGA 上に実装することとする．MBIF はクラスタボード上のクロック信号に同期して動作する．JUMP-1 のクラスタボードは 50MHz での動作を想定して実装されているので，MBIF も高速動作することを要求される．このため MBIF に用いる FPGA として，QuickLogic 社の高速 FPGA である QuickLogic pASIC-1 を用いる．

7.1.2.4 MBIF-MC 間の転送プロトコル

MBIF-MC 間の転送は非同期転送で行われる．MBIF から MC への転送手順を図 7.2 に示す．

転送手順の詳細は以下のとおりである．

(1) MBIF が X_SRDY をアクティブ (ローレベル) にする．

(2) X_SRDY がアクティブである事を確認して，MC 側が X_SSTB をアクティブ (ローレベル) にする．

(3) MBIF は X_SSTB がアクティブになると，即時に X_MDATA と X_COM の送信を開始する．

(4) MC 側は X_SSTB をアクティブにしている間に X_MDATA と X_COM のデータを取り込む．

(5) MC 側はデータの取り込みが終わると，X_SSTB を元に戻す．

(6) MBIF は X_SSTB が戻った事を確認して，データの送信を終了し，X_SRDY を戻す．

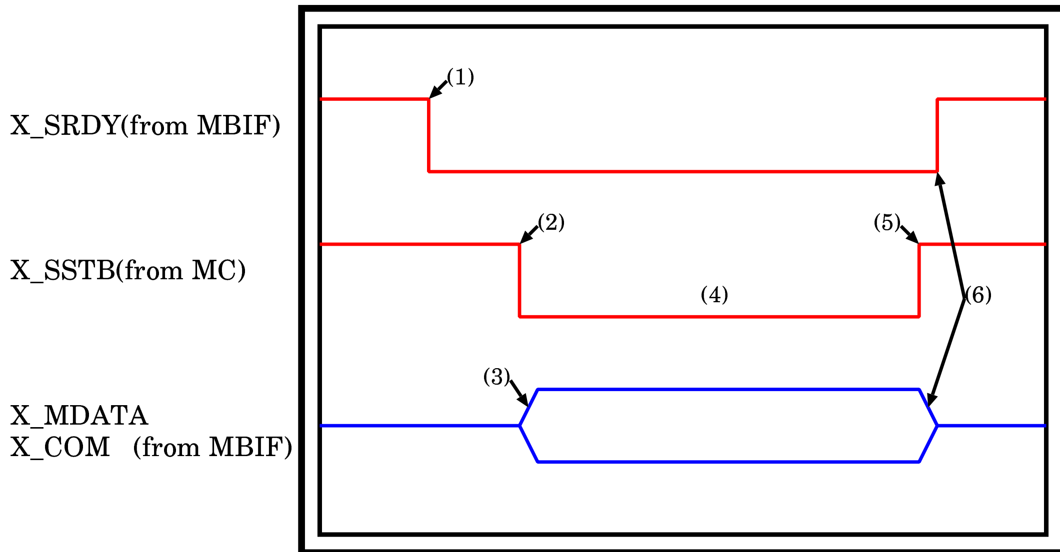


図 7.2 MBIF から MC への転送

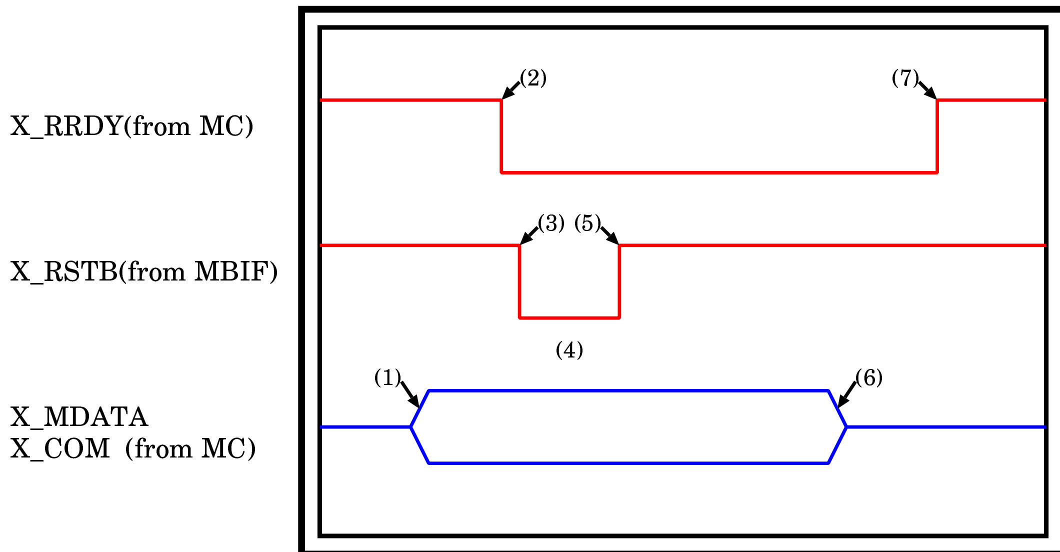


図 7.3 MC から MBIF への転送

次に，MCからMBIFへの転送手順を図7.3に示す．
転送手順の詳細は以下のとおりである．

(1) MC側は最初にX_MDATAとX_COMにデータをのせる．

(2) その後，MC側はX_RRDYをアクティブ(ローレベル)にする．

(3) X_RRDYがアクティブになると，MBIFはX_RSTBをアクティブ(ローレベル)にする．

(4) MBIFはX_RSTBをアクティブにしている間にX_MDATAとX_COMのデータを取り込む．

(5) MBIFはデータの取込みが終わると，X_RSTBを元に戻す．

(6) MC側がデータの送信を終了する．

(7) MC側がX_RRDYを戻す．

MCはプロトコル制御用の専用ハードウェアをもたず，メインテナンス用ホストPCのプログラムによって転送を行うため，MBIFからMCへの転送と，MCからMBIFへの転送では手順が少し異なっている．

7.1.3　提案のまとめ

メインテナンスシステムはJUMP-1のブートアップおよびモニタリングの機能を備えるシステムである．そしてその性質上，実装が容易であり回路規模が小さく，JUMP-1のスケーラビリティへの対応，JUMP-1の情報収集機能などが必要である．これらの機能を満たすため，メインテナンス用ホストPCにMCを，クラスタボード上にMBIFを設ける．また，拡張の際にはPCをハブで接続して対応する．MCとMBIF間の転送手順は7.1.2.4節に示した方法をとることとする．

7.2　メインテナンスシステムの実装形態

7.2.1　MBIF-MC間の実装形態

MBIFとMCは図7.4のような構成で接続される．MBIFを実装するFPGAであるQuickLogicのシンクドライブ能力は12mAであるため，直接フラットケーブルに接続するのは問題がある．このため，クラスタボード上にMBIFからの出力用のバッファとして74AS760を，MC側からMBIFへの入力用にシュミット入力レシーバ74LS541を設ける．ケーブルは34芯の標準フラットケーブル(1.27mmピッチ)を用いる．フラットケーブルには16本の信号線と，18本のGNDをそれぞれ割り当てる．

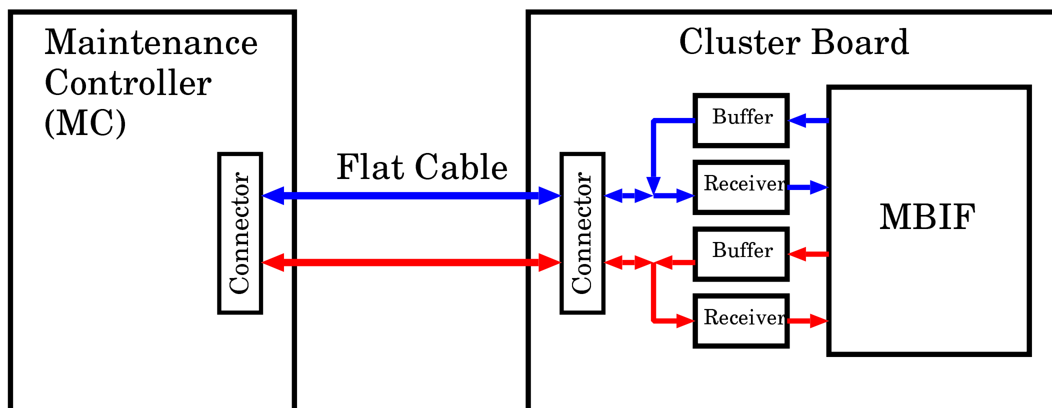


図 7.4 MBIF-MC の接続

7.2.2　MBIF-MC 間の信号線

MBIF は図 7.5 に示すような信号線によって外部と接続されている．このうち MBIF と MC 間の信号の概要を表 7.1 に，図 7.6 に示すコネクタへの割当てを表 7.2 に，それぞれ示す．MBIF と MC 間の信号線はすべて Open Collector である．なお，信号名の頭に X_ がつくものは負論理であることを示す．

表 7.1 MBIF-MC 間の信号

信号名	信号本数	入出力	論理	概要
X_MDATA	8	双方向	負論理	データライン
X_SRDY	1	出力	負論理	MBIF から MC への送信が可能
X_SSTB	1	入力	負論理	MC がデータを受信中
X_RRDY	1	入力	負論理	MC から MBIF への送信が可能
X_RSTB	1	出力	負論理	MBIF がデータを受信中
X_COM	1	双方向	負論理	コマンド及びデータ判別
X_INTRQ_MA	1	出力	負論理	未使用
X_MRESET	1	入力	負論理	MBIF のハードウェアリセット
MCLK	1	入力	負論理	12.5MHz のクロック信号

MBIF と MC 間のデータ線は 8 本で，MDATA[7..0] という信号名である．X_SRDY と X_SSTB は MBIF から MC への転送の際に使用する制御信号線で，X_RRDY と X_RSTB は MC から MBIF への転送の際に使用する．X_COM はデータ線の X_MDATA[7..0] の信号がコマンドであるときにアクティブになり，データであるときにはアクティブにならない．X_MRESET は MBIF のリセット端子に接続されており，この信号をアクティブにすることによって MBIF をハードウェアリセットすることができる．

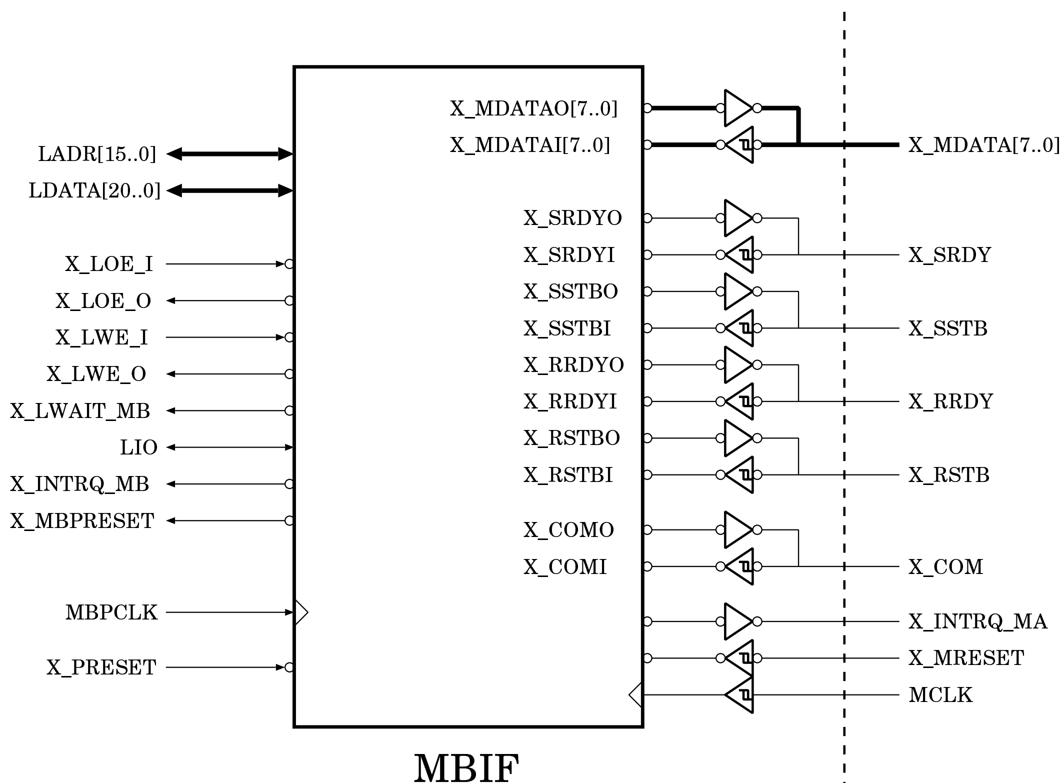


図 7.5 MBIF と外部の接続

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34

図 7.6 MBIF-MC 間接続用コネクタのピン配置

表 7.2 各信号のコネクタへの割当て

信号名	コネクタ端子番号	信号名	コネクタ端子番号
MCLK	2	X_MDATA1	22
X_MRESET	6	X_MDATA2	24
X_SRDY	8	X_MDATA2	24
X_SSTB	10	X_MDATA3	26
X_RRDY	12	X_MDATA4	28
X_RSTB	14	X_MDATA5	30
X_COM	16	X_MDATA6	32
X_INTRQ_MA	18	X_MDATA7	34
X_MDATA0	20	GND	4, 奇数番号ピン

7.2.3　MBIF とローカルバスの接続

MBIF の信号線のうち，ローカルバス側のものの概要を表 7.3 に示す．

表 7.3 MBIF の信号線 (ローカルバス側)

信号名	信号本数	入出力	論理	概要
LDATA	21	双方向	正論理	データライン
LADR	16	双方向	正論理	アドレスライン
X_LOE_I	1	入力	負論理	アウトプットイネーブル
X_LWE_I	1	入力	負論理	ライトイネーブル
X_LOE_O	1	出力	負論理	アウトプットイネーブル
X_LWE_O	1	出力	負論理	ライトイネーブル
X_LWAIT_MB	1	出力	負論理	ウェイト信号
LIO	1	入出力	正論理	I/O アクセスかローカルメモリアクセスかの判別
X_INTRQ_MB	1	出力	負論理	MBP-light への割込み信号
X_MBPRESET	1	出力	負論理	リセット信号
MBPCLK	1	入力	正論理	50MHz のクロック信号
X_PRESET	1	入力	負論理	パワーオンリセット

LDATA はローカルバスのデータ線である．MBP-light はローカルメモリ，I/O に対して 21 bit でアクセスするため，LDATA は 21 bit となっている．LADR はアドレス線である．MBIF には I/O 空間の 0x2000 番地と 0x3000 番地が割り当てられている．各アドレスに対するアクセスは表 7.4 のようになっている．

表 7.4 MBIF の I/O アドレス

アドレス	リード/ライト	概要
0x2000	リード	MC から MBP-light へのデータ転送レジスタ (LDR) のリード
0x2000	ライト	MBP-light から MC へのデータ転送用レジスタ (MDRO) へのライト
0x3000	リード	MBP-light 側のステータスレジスタ (LSR) の読込み
0x3000	ライト	MBP-light から MBIF へのコマンドの書込み

X_LOE_I 及び X_LWE_I は MBP-light のアウトプットイネーブル，ライトイネーブル信号にそれぞれ接続されている．この信号は入力信号になっていて，MBP-light が MBIF をアクセスする際に必要となる．X_LOE_O 及び X_LWE_O は MBIF からの出力信号である．この信号はローカルメモリなどのデバイスに対して MBIF がアクセスする際に使用する．X_LWAIT_MB はウェイト信号であり，MBP-light をウェイトさせるときに使用する．MBIF がこの信号を使用することも可能である．LIO は，I/O アクセス，ローカルメモリアクセスの判別に用いる．LIO がアクティブになっているとき，そのローカルバスに対す

るアクセスは I/O デバイスに対して行われることを示し，そうでないときは，ローカルメモリに対するアクセスであることを示す．X_INTRQ_MB は，出力信号で，この信号をアクティブにすることによって MBP-light に割込みをかけることができる．X_MBPRESET は，MBP-light，SPARC プロセッサなどのクラスタ上の各デバイスに対するリセット信号で，出力信号である．MBIF はメインテナンス用ホスト PC からのコマンドによって，各デバイスにリセットをかけたり，解除したりすることができる．MBPCLK は，MBP-light と MBIF 共通のクロックで，MBIF はこのクロックに同期して動作する．X_PRESET は MBIF のパワーオンリセットである．

7.3　MC(Maintenance Controller) の実装

7.3.1　MC の全体構成

MC の全体構成を，図 7.7 に示す．

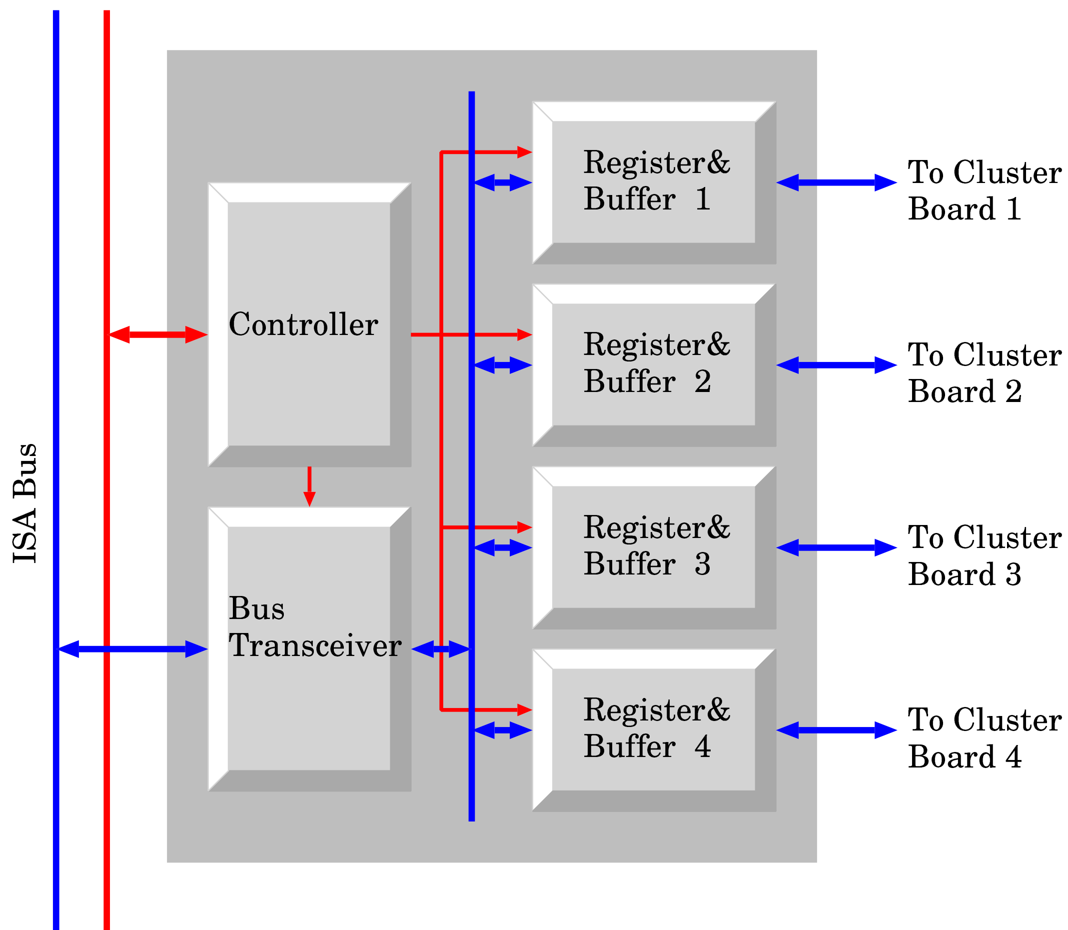


図 7.7 MC の全体構成

MC はメインテナンス用 PC の ISA バスに接続して使用する．MC には Bus Transceiver

部，Controller 部，Register&Buffer 部がある．Bus Transceiver 部は，ISA バスと MC 内部のデータの入出力を行う．Controller 部は，アドレスデコーダや ISA バスの制御信号を監視する論理回路で構成され，Bus Transceiver 部，Register&Buffer 部の制御を行う．Register&Buffer 部は送信するデータを保存する Register，外部への信号線をドライブするための Buffer，MBIF から送信されたデータを受信するための Receiver からなる．Register&Buffer 部は 4 回路が 1 ボード上に実装され，各回路が 1 つのクラスタボードと接続される．

7.3.2　MC のソフトウェアプロトコル制御機構

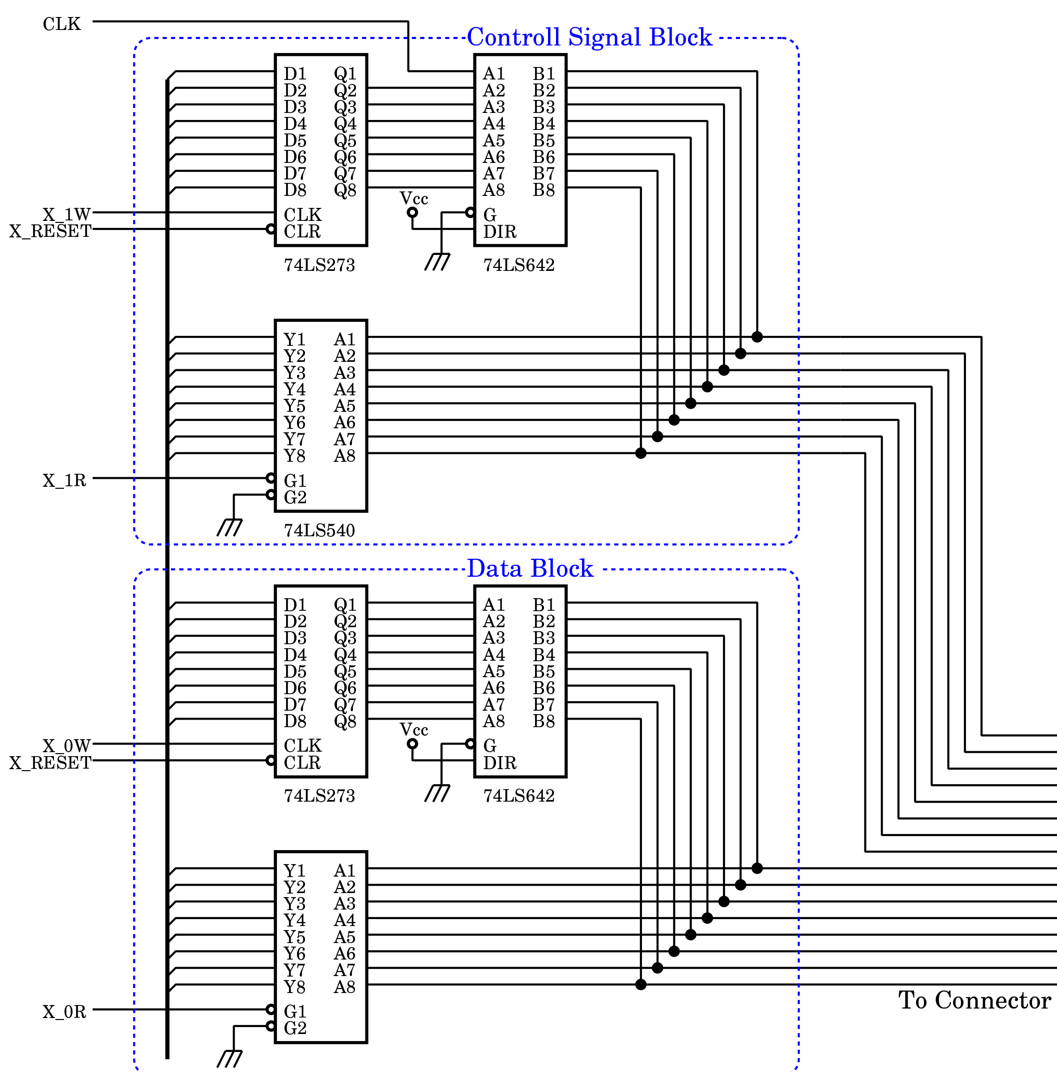


図 7.8 Register&Buffer 部の構成

MC は制御信号をコントロールするハードウェアを持たず，ソフトウェアによって転送プロトコルの制御を行う．図 7.8 は MC の Register&Buffer 部の構成である．各 IC の詳細は

付録の図 I.1 と図 I.2，図 I.3 に示した．MBIF との間でデータ転送を行う Register&Buffer 部には Controll Signal Block と Data Block が存在する．どちらの Block も回路構成は同じである．このうち，Controll Signal Block が MC-MBIF 間の転送の制御信号をコントロールする．MBIF との間でデータ転送を行う場合，MBIF からの制御信号の取得はメインテナンス用ホスト PC が Controll Signal Block に対して読出しを行うことによって行う．また，MC 側の制御信号の制御は書込みを行うことによって行っている．さらに，データの受信は Data Block を読み込むことによって，送信は書き込むことによって行っている．

7.3.3　MC の I/O アドレス可変機構

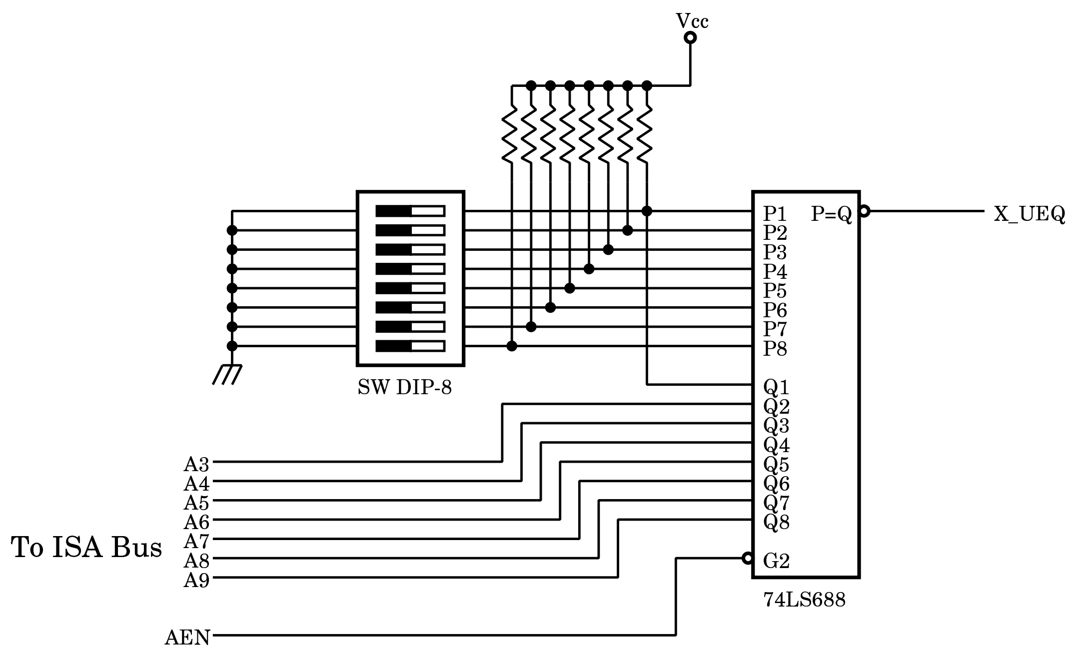


図 7.9 I/O アドレス可変機構

MC はメインテナンス用ホスト PC の ISA バス上に実装されるので，PC 上での I/O アドレスの重複を避ける必要がある．PC の各 I/O デバイスに割り当てられるアドレスは個々の PC の環境によって異なるので，MC は様々な環境に対して適応できることが必要となる．このため，MC に割り当てる I/O アドレスが変更可能な構造をとることにした．アドレスの設定は MC のボード上に実装したディップスイッチによって行う．ディップスイッチの出力と ISA Bus のアドレス線は付録の図 I.4 に示す 8-Bit Magnitude Comparator IC である 74LS688 に入力される．そして，ディップスイッチの設定と ISA バスのアドレスが一致し，更に AEN 信号 (アドレスイネーブル) がアクティブなときに MC へのアクセスが行われるようになっている．

7.4 MBIF(Maintenance Bus InterFace) の実装

7.4.1 QuickLogic

MBIF は JUMP-1 のクラスタボード上に配置され，クラスタボードのクロックに同期して動作する．JUMP-1 は，全システムが 50MHz で動作するように実装したため，MBIF も高速動作をする必要がある．このため，MBIF は QuickLogic 社の高速 FPGA である QuickLogic pASIC 1 上に実装することとした．QuickLogic は金属層と金属層を直接アンチヒューズで接続するアンチヒューズ型 FPGA である．アンチヒューズ型の構造によって，ロジックの書換えを行うことは不可能であるが，高速動作をすることは可能となる．pASIC 1 ファミリの概要を表 7.5 に示す．

表 7.5 pASIC 1 Family

Device	Logic Cells	Package Pins	Usable Gates
QL8x12B	96	44,68,100	1000
QL12x16B	192	68,84,100	2000
QL16x24B	384	84,100,144,160	4000
QL24x32B	768	44,144,208	8000

MBIF の実装には，4000 Usable Gate で 84Pin PLCC パッケージの QL16x24B-1PL84C を用いた．このチップの特徴は次のとおりである．

- 16 × 24 のアレイ上に並んだ 384 個のロジックセル
- 全ゲート数 12,000 ゲート，ゲートアレイ換算のユーザブルゲートで 4,000 ゲート
- チップ内のフリップフロップの動作周波数は 150MHz

7.4.2 Verilog-HDL

QuickLogic への MBIF の実装の際，ハードウェア記述言語 Verilog-HDL を用いる．Verilog-HDL は，1995 年に IEEE1364 として標準化された．Verilog-HDL は，ライブラリの充実，採用実績が多数あるなどの理由から，米国国防省を中心に，IEEE1076 としていち早く標準化された VHDL とともに，ASIC 開発，FPGA 開発などに広く用いられている．

Verilog-HDL の特徴を以下に挙げる

- C 言語を主体とした言語体系であり，記述が簡潔
- 構文や演算子が C 言語とほぼ同じなので，類推で記述でき，習得が容易
- シミュレーション用言語として誕生した経緯もあって，シミュレーション用記述力が充実
- ASIC の開発実績が多数あるため，シミュレーション用のライブラリやツールが豊富

7.4.3　ステートマシンの構成法

ステートマシンの構成法は，大別すると図 7.10 に示すワン・ホット方式と図 7.11 に示すデコード方式がある．ステートを保持するためには Flip Flop が必要であり，1 つのステートに対して 1 つの Flip Flop を割り当てるのが，ワン・ホット方式である．一方，デコード方式では，使用する Flip Flop の数が少なく，その出力をデコードして用いる．以下に，それぞれの方式の特徴を示す．

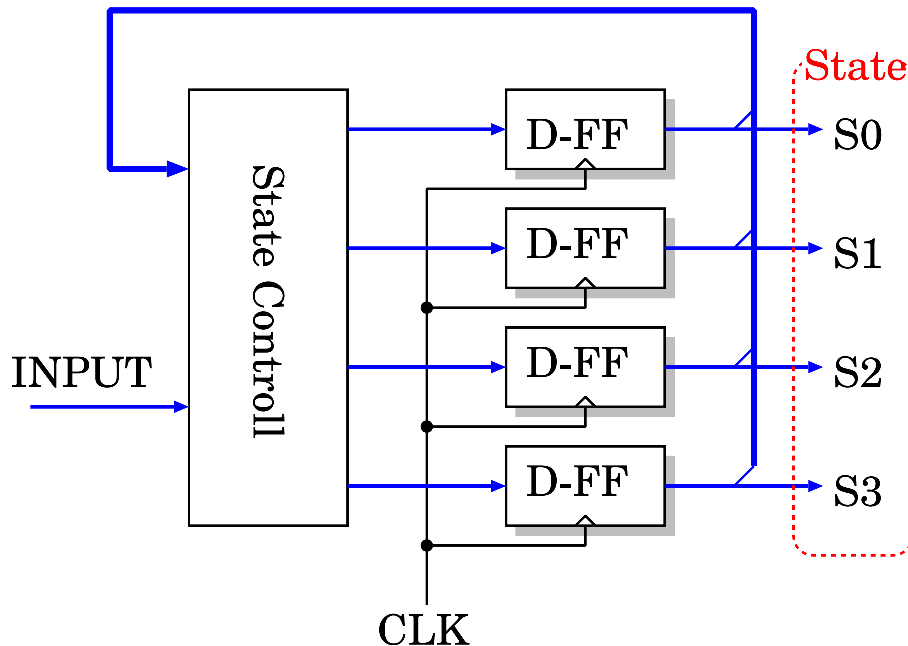


図 7.10 ワン・ホット方式のステートマシン

- ワン・ホット方式
  - 使用する Flip Flop の量が多い．
  - 動作スピードは速い．
  - 用途によっては，複数のステートが同時にアクティブにならない．ためのフェイルセーフ的な回路が必要．
- デコード方式
  - 使用する Flip Flop の量が少ない．
  - ステート数が多いと，ステートデコーダ部分の回路規模が大きくなり，動作スピードが遅くなる．
  - HDL 記述が簡潔で分かりやすい．

MBIF 内部には複数のステートマシンが存在し，様々な規模のものが存在する．そこで MBIF では，小規模のステートマシンにはデコード方式を，大規模なものにはワン・ホット方式を用いた．

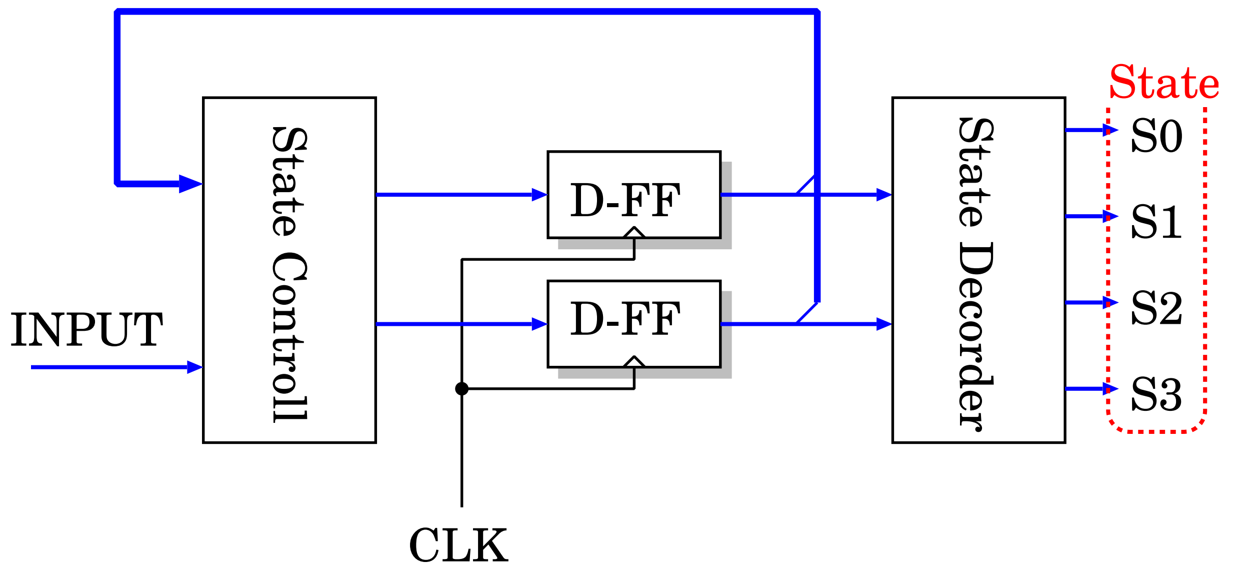


図 7.11 デコード方式のステートマシン

7.4.4　MBIF の内部構成

MBIF の内部構成を，図 7.12 に示す．MBIF は大きく分けて，BusIF，Sender，Receiver，Controller の 4 つの部分から構成される．BusIF は MBIF とクラスタのローカルバスの間のインタフェース部分である．MBIF がローカルバスをアクセスする際や MBP-light の動作時に，MBP-light が MBIF をアクセスする場合など，すべて BusIF を介してアクセスが行われる．Sender 部と Receiver 部は，MBIF と MC 間の送信及び受信を担当する．Sender 部が MBIF から MC へのデータ送信を，Receiver 部が MC から MBIF へのデータの受信を担当する．Controller 部は MBIF 全体の動作を制御している．PC や MBP-light からのコマンドもここで解析され，実行される．

7.4.5　Sender 部の実装

Sender は，MBIF から MC へのデータ転送を担当するモジュールである．このモジュールの入出力信号は，表 7.6 に示すものがある．このうち大文字の信号名を持つものは MBIF 外部と接続されており，小文字の信号名のものは MBIF 内部で使用されるものである．図 7.2 に示した転送プロトコルを実装するため，図 7.13 に示す状態遷移をするステートマシンを構成する．Sender 部のステートマシンは，ステート数が 3 ステートと少ないため，記述が容易なデコード型ステートマシンを用いている．

7.4.5.1　Sender 部の転送動作

MBIF が MC に対して転送動作を行うのは，MC からデータ転送コマンドを受信したときである．コマンドの詳細は Appendix の表 G.2，表 G.3 にそれぞれ示す．データ転送コ

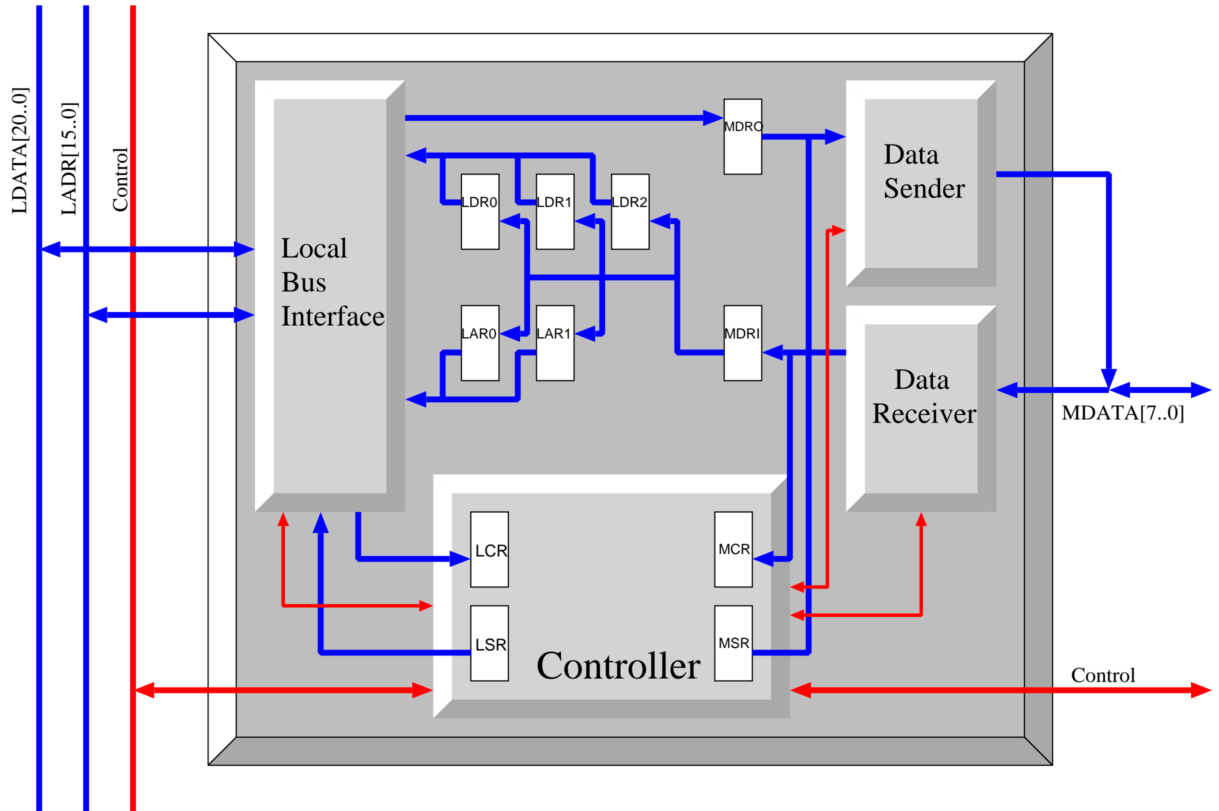


図 7.12 MBIF の内部構成

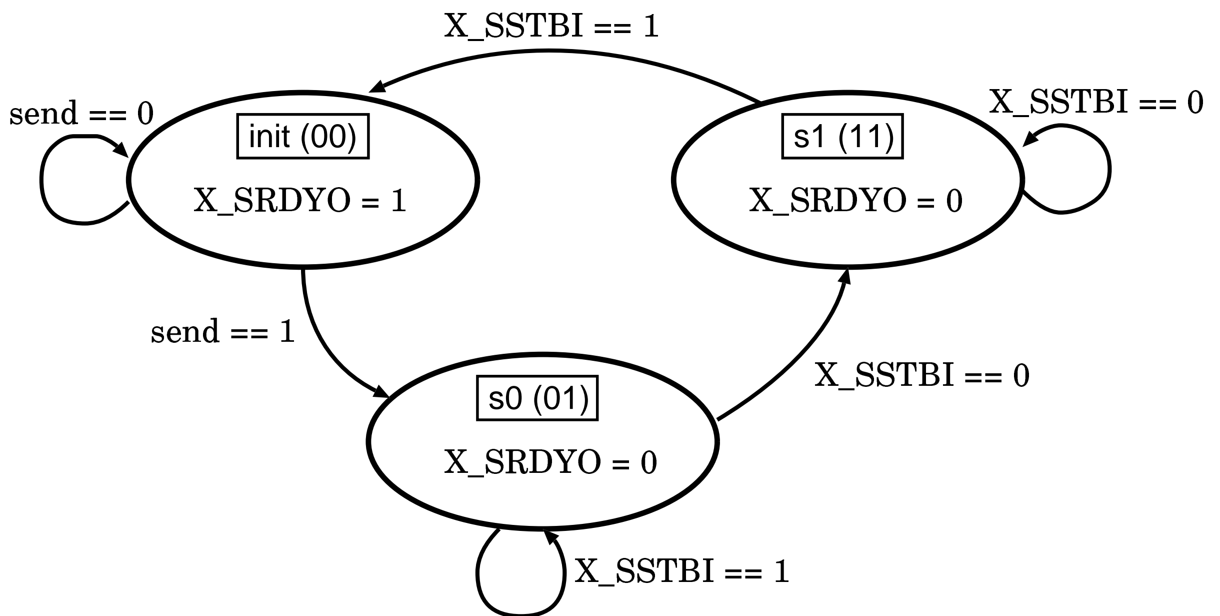


図 7.13 Sender 部ステートマシンの状態遷移図

表 7.6 Sender 部の入出力信号

信号名	概要
X_SRDYO	MBIF 外部への出力信号であり、MBIF-MC 間接続ケーブルの X_SRDY に接続する． MBIF から MC への転送プロトコル制御用．
X_SSTBI	MBIF 外部からの入力信号であり、MBIF-MC 間接続ケーブルの X_SSTB に接続する． MBIF から MC への転送プロトコル制御用．
X_MDATAO	MBIF 外部への出力信号であり、MBIF-MC 間接続ケーブルの X_MDATA に接続する． MBIF から MC への転送データを出力する．
X_COMO	MBIF 外部への出力信号であり、MBIF-MC 間接続ケーブルの X_COM に接続する． コマンドとデータの判別用．
send_data	MBIF 内部信号であり、Sender 部への入力信号である．MC へ転送するデータを入力する．
com	MBIF 内部信号であり、Sender 部への入力信号である．X_COMO への信号を入力する．
send	MBIF 内部信号であり、Sender 部への入力信号である．Sender 部は、send 信号が立ち上がると、送信を開始する．
busy	MBIF 内部信号であり、Sender 部からの出力信号である．Sender 部が動作中であることを示すのに用いる．
clk	MBIF の内部クロック．
X_rst	MBIF の内部リセット信号．

マンドを受信した後の Sender 部の転送動作を以下に示す．

(1) Controller 部が send 信号をアサートする．

(2) ステートマシンが init ステートから s0 ステートに移行し，X_SRDYO をアサート (ローレベル) する．また，このとき busy 信号がアサートされ，Sender 部が動作中であることを示す．同時に，送信データのフェッチも行う．

(3) MC が X_SSTBI をアサート (ローレベル) するのを待つ．

(4) X_SSTBI がアサートされると，ステートマシンは s1 ステートに移行し，データ出力用論理回路がデータを X_MDATAO，X_COMO に出力する．

(5) MC が X_SSTBI をネゲート (ハイレベル) するのを待つ．

(6) X_SSTBI がアサートされると，ステートマシンは init ステートに移行し，データ出力用論理回路がデータの出力を止める．同時に X_SRDYO，busy がネゲートされ，データ転送が終了する．

7.4.6　Receiver 部の実装

Receiver は MC からのデータ受信を担当するモジュールである．このモジュールの入出力信号を表 7.7 に示す．Sender 部同様，大文字の信号名を持つものは MBIF 外部と接続されており，小文字の信号名のものは MBIF の内部信号である．図 7.3 の受信プロトコルを実装するため，Receiver 部では図 7.14 に示す状態遷移をするステートマシンを実装する．Receiver 部のステートマシンはステート数が 4 ステートであり，比較的小規模である．このため記述が容易なデコード型ステートマシンを用いたが，図 7.14 に示すように，安定性の向上のためステート移動の際のビット変化は 1 bit となるように実装した．

7.4.6.1　Receiver 部の受信動作

Receiver 部は MC からのデータ転送要求に対していつでも応答できるように実装されている．以下は，Receiver 部の受信動作である．

(1) MC が X_SRDYI をアサート (ローレベル) する．

(2) ステートマシンが init ステートから s0 ステートに移行する．このステートはウェイト用である．

(3) ステートマシンが s1 ステートから s2 ステートに移行する．このとき，X_COMI がアサート (ハイレベル) されていれば mcr_set をアサートし，ネゲートされていれば mdri_set をアサートする．mcr_set は，MCR レジスタ (コマンドレジスタ) にデータをセットする信号で，mdri_set は MDRI レジスタ (データレジスタ) にデータをセットする信号である．

(4) mcr_set，mdri_set をネゲートする．

表 7.7 Receiver 部の入出力信号

信号名	概要
X_RRDYI	MBIF 外部からの入力信号であり，MBIF-MC 間接続ケーブルの X_RRDY に接続する． MC から MBIF への転送プロトコル制御用.
X_RSTBO	MBIF 外部への出力信号であり，MBIF-MC 間接続ケーブルの X_RSTB に接続する． MC から MBIF への転送プロトコル制御用.
X_MDATAI	MBIF 外部からの入力信号であり，MBIF-MC 間接続ケーブルの X_MDATA に接続する． MC からの受信データを入力する.
X_COMI	MBIF 外部からの入力信号であり，MBIF-MC 間接続ケーブルの X_COM に接続する． コマンドとデータの判別用.
r_data	MBIF 内部信号であり，Receiver 部からの出力信号である．MBIF の内部レジスタに転送する MC からの受信データを出力する.
mdri_set	MBIF 内部信号であり，Receiver 部からの出力信号である．MC からのデータを MDRI レジスタ (データレジスタ) に保存する際に使用.
mcr_set	MBIF 内部信号であり，Receiver 部からの出力信号である．MC からのデータを MCR レジスタ (コマンドレジスタ) に保存する際に使用.
clk	MBIF の内部クロック.
X_rst	MBIF の内部リセット信号.

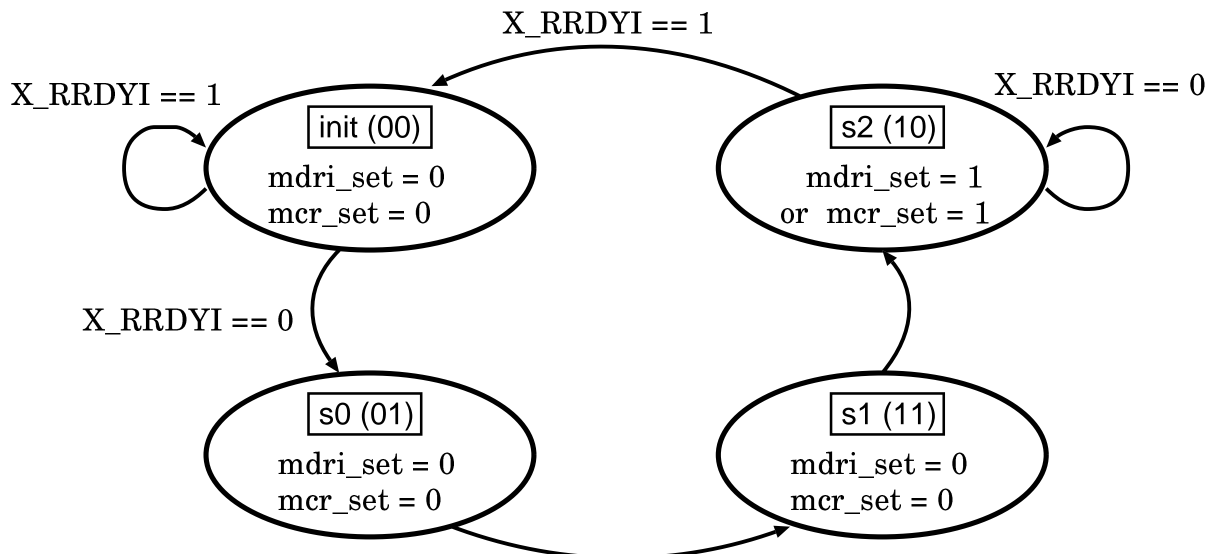


図 7.14 Receiver 部ステートマシンの状態遷移図

(5) MC が X_RRDYI をネゲート (ハイレベル) するのを待つ．

(6) X_RRDYI がネゲートされると，ステートマシンは init ステートに移行し，次の転送が行われるまで待つ (受信完了).

7.4.7 Local Bus Interface 部の実装

このモジュールは JUMP-1 のクラスタボード上のローカルバスとのインタフェースを担当する．モジュール内部には，MBP-light が MBIF をアクセスする際の制御やローカルバスとのデータの入出力を行う組合せ論理回路部分と，MBIF がローカルメモリに対して書き込みを行う際の制御を行うステートマシンが存在する．このステートマシンの状態遷移図を図 7.15 に示す．Local Bus Interface 部のステートマシンはステート数が 5 ステートあるため，ワン・ホット方式を採用している．また，Local Bus Interface 部の信号線は表 7.8 のようになっている．

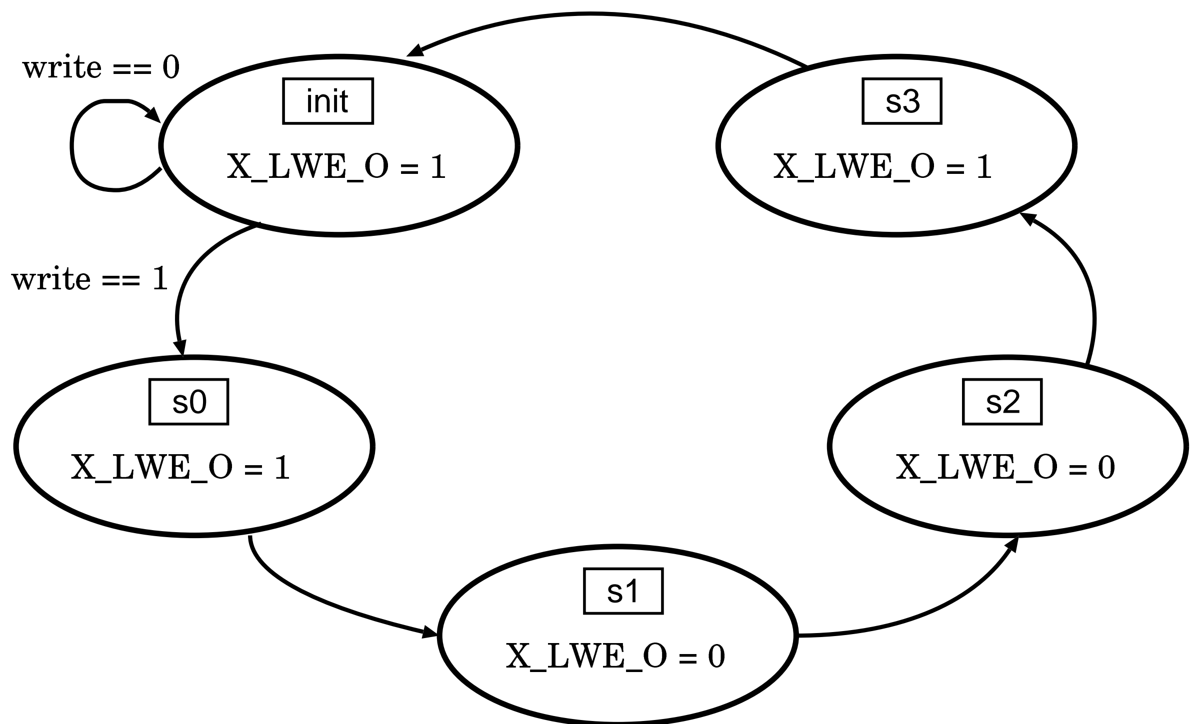


図 7.15 Local Bus Interface 部ステートマシンの状態遷移図

7.4.8 Controller 部の実装

MBIF 全体の制御の取りまとめを行う Controller モジュールは，MBIF の論理回路の大部分を占める．この内部には，MC からのコマンドのデコーダ，コマンドを処理するため

表 7.8 Local Bus Interface 部の入出力信号

信号名	概要
LADR	16 bit のアドレス線.
LDATA	21 bit のデータ線.
X_LOE_I	入力信号の，アウトプットイネーブル．MBP-light が MBIF をアクセスする際に使用する．
X_LWE_I	入力信号の，ライトイネーブル．MBP-light が MBIF をアクセスする際に使用する．
X_LOE_O	出力信号の，アウトプットイネーブル．MBIF がローカルバスをアクセスする際に使用する．
X_LWE_O	出力信号の，ライトイネーブル．MBIF がローカルバスをアクセスする際に使用する．
X_LWAIT_MB	MBP-light に対するウェイト信号.
LIO	I/O アクセスとローカルメモリアクセスの判別に用いる．
ldata_in	ローカルメモリに書込みを行う際のデータの入力．LDR2 を上位，LDR1 を中位，LDR0 を下位とする 21 bit データが入力される．
ladr_in	ローカルメモリに書込みを行う際のアドレスの入力．LAR1 を上位，LAR0 を下位とした 16 bit のアドレスが入力される．
lsr_in	LSR レジスタの内容が入力される．MBP が LSR の読出しを行う際に，データ入力を行う．
clk	MBIF 内部クロック.
X_rst	MBIF 内部リセット.
write	Controller 部がローカルメモリへの書込みを指令するために用いる．
read	Controller 部がローカルメモリの読込みを指令するために用いる．
busy	Local Bus Interface が動作中であることを示す.
mdroset	MDRO(MC への送信用データレジスタ) にデータをセットする際に使用する．
lcrset	LCR(MBP-light 側のコマンドレジスタ) にコマンドをセットする際に使用する．

のステートマシン，更に MBP-light からのコマンドのデコーダと実行用論理回路，その他組合せ論理回路が存在する．

7.4.8.1　MC 側コマンドデコーダ

MBIF はメインテナンスシステムの中核をなすデバイスであり，メインテナンス作業の大部分を処理する．このため，メインテナンス用ホスト PC から送られてくるメインテナンスコマンドの処理が大きな比重を占める．MC から送られてくるメインテナンスコマンドは，初期化モード用とメインテナンスモード用に大別される．コマンドの詳細は Appendix の表 G.2 と表 G.3 に示す．初期化モードは MBP-light やプロセッサがリセット状態にあるときのモードで，主にローカルメモリの初期化や MBP-light のマイクロプログラムのロードを行う．メインテナンスモードはリセットが解除されたときのモードで，MBIF は MBP-light の I/O デバイスとして動作することによって PC との双方向通信を実現している．初期化モード用コマンドは全部で 12 個，メインテナンス用コマンドは全部で 9 個存在し，合計 21 のコマンドをデコードするため，MC 側コマンドデコーダは大規模な回路となっている．デコーダの入力は MC から送られてくるコマンド保持用のレジスタである MCR レジスタに接続する．このデコーダは大規模なため，1 ブロックで構成すると非常に複雑な回路となってしまう．回路が複雑であると大きいファンアウトをとらなければいけないロジックが多数存在することになる．ファンアウトが大きいと動作速度の低下の原因となる．また，バッファを接続すればファンアウトの問題は解決するが，遅延時間が大きくなってしまう．そこで，デコーダを 2 ブロックに分けて構成し，機能の分化を図ることによって複雑化をさける．

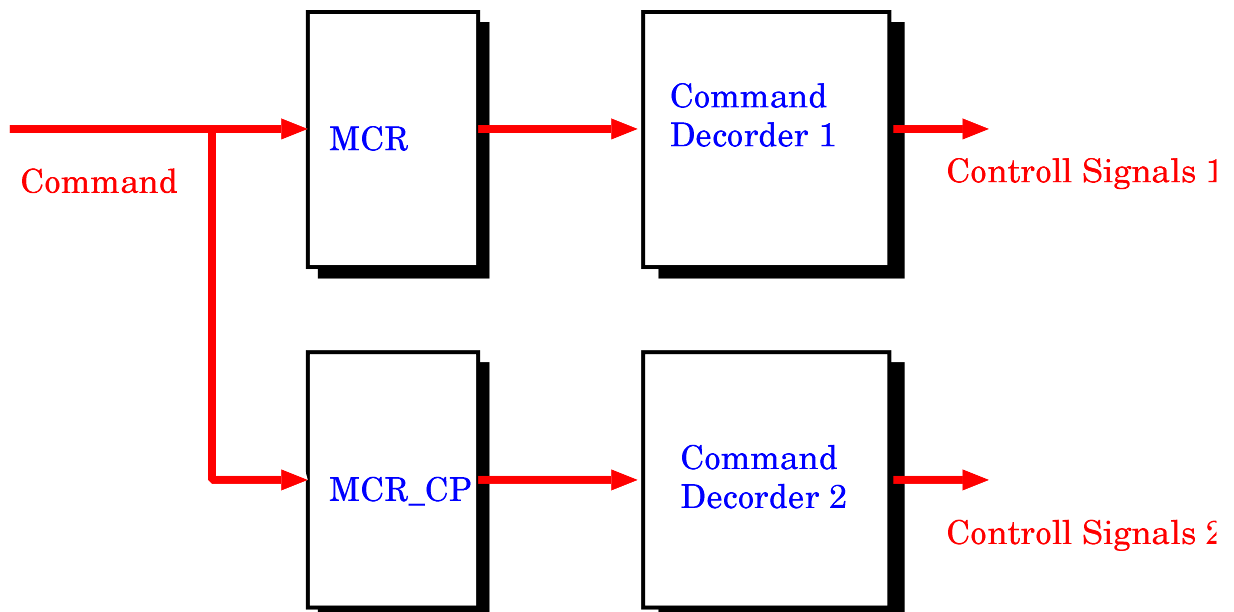


図 7.16 デコーダの構成

このデコーダ構成を図 7.16 に示す．まず，MCR レジスタと全く同じデータを保持する

複製のレジスタ (MCR_CP レジスタ) を設け，デコーダを分割してデコードするコマンドを分担することとする．MCR_CP レジスタに接続する Command Decorder 2 が LAR レジスタ (アドレスレジスタ) と LDR レジスタ (データレジスタ) 関連のコマンドを担当する．また，MCR レジスタに接続する Command Decorder 1 がそれ以外のコマンドすべてをデコードする．

7.4.8.2　MC 側コマンド実行用ステートマシン

MC から送信されてくるコマンドを実行するためのステートマシンで，全部で 20 ステート存在する．この 20 ステートのうち，12 ステートが MBP-light のリセットがかかっているときのモードである初期化モード用，残りの 8 ステートが MBP-light のリセットが解除されているときのメインテナンスモード用となっている．このステートマシンはステート数が多く，また状態遷移も複雑であるため，ワン・ホット方式のステートマシンを採用している．ステートマシンの状態遷移は図 7.17 に示すようになっている．以下，各ステートについて簡単に触れる．

init

初期化モードのコマンド待ちステートである．MBIF がリセットされた直後，ステートマシンはこのステートへと遷移する．また，メインテナンスモードにおいて初期化モードへの移行コマンドを実行することによっても，このステートへの遷移が行われる．

i_lar_w0, i_lar_w1

初期化モードにおいて，MBIF がローカルメモリをアクセスする際に使用する LAR レジスタ (アドレスレジスタ) に値を書き込むコマンドを処理する際に使用するステートである．

i_ldr_w0, i_ldr_w1, i_ldr_w2

初期化モードにおいて，MBIF がローカルメモリに書込みを行う際のデータを保持する LDR レジスタ (データレジスタ) への書込みコマンドを処理する際に使用する．

i_send0, i_send1, i_send2, i_send3

初期化モードにおいて，MBIF から MC への送信を行うコマンドを処理する際に使用する．MDRO レジスタ (MC への送信用データレジスタ) と MSR レジスタ (ステータスレジスタ) のデータが送信可能であるが，MBP-light がリセット状態であるため MDRO レジスタへのデータセットは行われないので，ほぼ MSR レジスタのデータ転送のみに用いられる．

i_write0, i_write1

初期化モードにおいて，MBP-light のローカルメモリへの書込みコマンドの処理に用いられる．

maintenance

メインテナンスモードのコマンド待ちステートである．初期化モードにおいてメインテナンスモードへの移行コマンドを実行すると，このステートに遷移する．

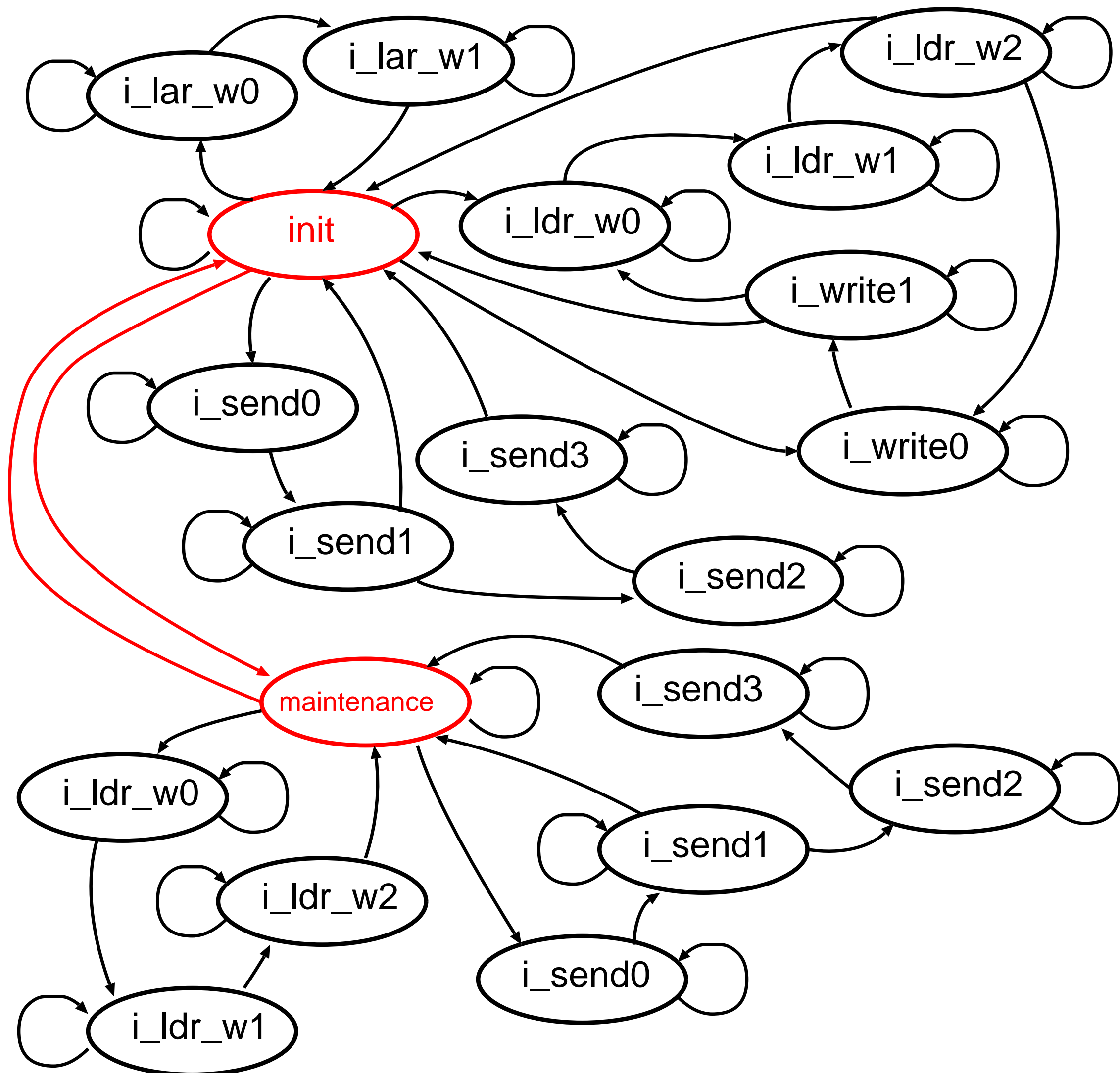


図 7.17 Controller 部ステートマシンの状態遷移図

m_ldr_w0, m_ldr_w1, m_ldr_w2

メインテナンスモードにおいて，LDR レジスタ (データレジスタ) に値を書き込むコマンドを処理する際に使用するステートである．LDR レジスタは，メンテナンスモードのときには PC から MBP-light へのデータ転送用レジスタとして使用する．

m_send0, m_send1, m_send2, m_send3

メインテナンスモードにおいて，MBIF から MC への送信を行うコマンドを処理する際に使用する．MDRO レジスタ (MC への送信用データレジスタ) と MSR レジスタ (ステータスレジスタ) のデータが送信可能であり，MDRO レジスタへ MBP-light が書き込んだデータを MC に送信することによって MBP-light からメインテナンス用ホスト PC へのデータ転送が行われる．

7.4.8.3　MBP-light 側コマンド実行回路

MBP-light からのコマンドはコマンド数も少なく，動作も簡単なため小規模な回路となっている．なお，コマンドについては Appendix の表 G.1 に示す．MBP-light からのコマンドはすべて 1 クロックで動作が完了するため，ステートマシンも存在しない．この回路はコマンドデコーダと制御信号生成回路によって構成する．

7.4.9　MBIF のレジスタ

MBIF には様々なメインテナンス用コマンドを保持するためのコマンドレジスタや MBIF の状態を表すステータスレジスタ，その他データレジスタアドレスレジスタなど，様々なレジスタが存在する．ここでは MBIF のレジスタについて簡単に触れる．レジスタの概要は，表 7.9 のとおりである．

表 7.9 MBIF のレジスタ

レジスタ名	bit 幅	概要
LDR0	8	データレジスタ．データの下位 8 bit を保持する．
LDR1	8	データレジスタ．データの中位 8 bit を保持する．
LDR2	5	データレジスタ．データの上位 5 bit を保持する．
LAR0	8	アドレスレジスタ．アドレスの下位 8 bit を保持する．
LAR1	8	アドレスレジスタ．アドレスの上位 8 bit を保持する．
MDRO	8	MBIF から MC へ送信するデータを保持する．
MDRI	8	MC から MBIF へ送られてきたデータを保持する．
MCR	8	コマンドレジスタ．MC からのコマンドを保持する．
MSR	8	ステータスレジスタ．MC 側からアクセスすることができる．
LCR	8	コマンドレジスタ．MBP-light からのコマンドを保持する．
LSR	8	ステータスレジスタ．MBP-light 側からアクセスすることができる．

LDR0, LDR1, LDR2

データレジスタで，MBP のローカルメモリの初期化の際に，ローカルメモリに書き込むデータを保存したり，MBP の動作時に，PC から MBP へデータ転送する際に，PC から転送されたデータを保存するのに用いられる．LDR0 と LDR1 は，8 bit のレジスタで，LDR2 は 5 bit である．これらの 3 つのレジスタは連続した 21 bit のデータとしてアクセスされ，LDR0，LDR1，LDR2 はそれぞれ下位，中位，上位ビットとなっている．

LAR0, LAR1

このレジスタはアドレスレジスタである．この 2 つのレジスタによって 16 bit のアドレスを保持する．LAR0 が下位ビットを，LAR1 が上位ビットを表す．このレジスタは MBIF がローカルメモリを初期化する際に使用する．

MDRO

MBP-light から PC にデータ転送する際に使用する．MBP-light が I/O アドレスの 0x2000 番地に書込みを行うと，MDRO レジスタに対して書込みが行われる．このデータを PC 側で読み出すことによって MBP から PC への転送を行っている．

MDRI

PC から送られてきたデータを一時的に蓄えておくのに用いられる．MDRI レジスタに蓄えられたデータは，その後各データレジスタやアドレスレジスタに割り振られる．

MCR

MC から送られてきたコマンドを保持する．MBP-light は，主にこの MC から送られてくるコマンドによって動作する．

LCR

MBP-light からのコマンドを保持する．MBP-light からのコマンドは，補助的なものに限られる．

MSR

MC から参照することが可能なステータスレジスタである．MSR レジスタが示す内容を表 7.10 に示す．

LSR

MBP-light から参照することが可能なステータスレジスタである．LSR レジスタは MSR レジスタとは少し異なっている．MSR レジスタは MC 側で必要な情報を，LSR は MBP-light 側で必要な情報を示すようになっている．LSR レジスタが示す内容を表 7.11 に示す．

表 7.10 MSR の構成

ビット位置	概要
第 0 bit(MDRO enable)	MDRO レジスタ (MBIF → PC の転送で使用) に新しいデータがあることを示す.
第 1 bit(Interrupt request)	未使用.
第 2 bit(LDR not empty)	LDR のデータを MBP-light がまだ読出していないことを示す.
第 3 bit(Interrupt enable)	PC から MBP-light への割込みが許可されていることを示す.
第 4 bit	未使用.
第 5 bit	未使用.
第 6 bit(M Busy)	MBIF が PC 側からのコマンドを処理中であることを示す.
第 7 bit(L Busy)	MBIF が MBP-light 側からのコマンドを処理中であることを示す.

表 7.11 LSR の構成

ビット位置	概要
第 0 bit(LDR enable)	LDR レジスタ (データレジスタ) に新しいデータがあることを示す.
第 1 bit(Interrupt request)	MBP-light に割込み要求があることを示す．INTRQ_MB 信号と同じ.
第 2 bit(LDR not empty)	LDR のデータを MBP-light がまだ読出していないことを示す.
第 3 bit(Interrupt enable)	未使用.
第 4 bit	未使用.
第 5 bit	未使用.
第 6 bit(M Busy)	MBIF が PC 側からのコマンドを処理中であることを示す.
第 7 bit(L Busy)	MBIF が MBP-light 側からのコマンドを処理中であることを示す.

7.5　メインテナンスシステムの評価

ここでは，メインテナンスシステムの各評価結果を示す．7.5.1 節は MBIF の各評価結果について，7.5.2 節は MC から MBIF への転送容量の測定結果について示す．

7.5.1　MBIF の評価

MBIF の各評価結果について示す．7.5.1.1 節は MBIF の各部の使用ゲート (モジュール) 数や全体の使用ゲート (モジュール) 数について，7.5.1.2 節はリセット解除コマンドにおける MBIF の遅延時間ついて示す．なお，MBIF は現在最大 25MHz で動作する．

7.5.1.1　MBIF の使用ゲート数

MBIF を実装した QuickLogic pASIC-1 QL16x24B-1PL84C は 16 × 24 個すなわち 384 個のロジックセルをもつ．このうち MBIF の実装によって実際に使用したセルの数は 259 セルであり，使用率は 67.4%である．

7.5.1.2　MBIF の遅延時間

MBIF がコマンドを受け付けてからコマンドが実行されるまでの遅延を測定した．このときの信号波形を，図 7.18 に示す．

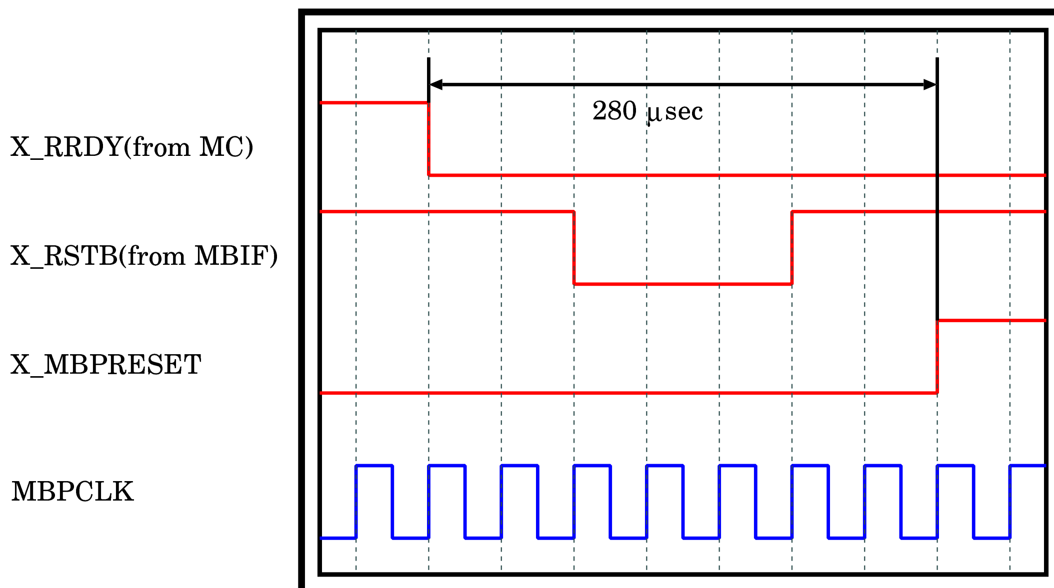


図 7.18 リセット解除コマンド実行時の遅延の測定

ここでは，MBIF のリセット解除コマンドを例として測定を行った．MBIF はリセット解除コマンドを受け取ると，MBP-light のリセット信号である X_MBPRESET をハイレ

ベルにする．図 7.18 に示した他の信号で，X_RRDY は MC が MBIF にコマンドの送信を始めるときローレベルとなる．X_RSTB は，MBIF が受信中のときローレベルとなる．

MBIF は X_RRDY の立ち下がりから 7 クロック目で X_MBPRESET をハイレベルにしている．MBIF は 25MHz で動作が可能なので，X_RRDY の立ち下がりから 280nsec でこのコマンドの実行を終了することができる．MBIF はメンテナンスを行うシステムであり，CPU のような動作は要求されないことを考慮すると，十分な応答速度が得られたと考えられる．

7.5.2　MBIF-MC 間の最大転送容量

MC から MBIF への最大転送容量の測定結果である．

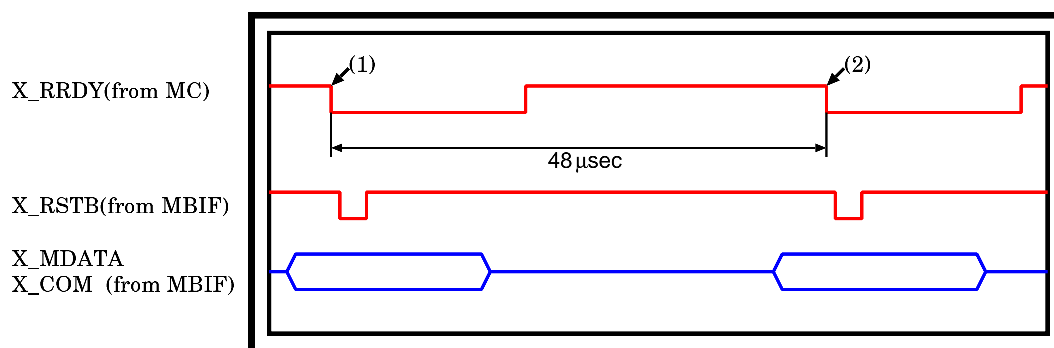


図 7.19 最大転送容量の測定

MC から MBIF へ転送を行う際，8 bit 単位で転送が行われる．図 7.19 は MC から MBIF への転送中の信号波形の測定結果である．テストにはメンテナンス用ホスト PC として CPU にインテルの 486SX(20MHz) を搭載したマシンを用いた．ここでは，あるデータが転送されてから次のデータが転送されるまでの時間を計測した．これは図 7.19 の (1) から (2) までの時間である．この場合データの転送周期は 48μsec であったため，この最大転送容量は 21KByte/sec である．MBIF の応答速度はこの周期に比べて十分高速なため，MC と MBIF 間の転送は MC の速度がボトルネックとなっていることがわかる．一方，メンテナンス用ホスト PC に更に高速の CPU を採用した PC を使用した場合は，転送容量が改善されることが予想される．しかし，現在のソフトウェア制御転送機構では，CPU が高速化されても転送容量に大きな改善は望めないため，更に高速化を行う場合には転送機構の改良を行う必要がある．

第8章　Disk入出力サブシステム

本章では，JUMP-1 の Disk 入出力システムについて述べる．本章の評価は，主に慶應義塾大学天野研究室の長名による研究成果による．

8.1　システムの概要

8.1.1　設計方針

JUMP-1 の入出力サブシステムの設計にあたっては，次の点に特に注意が払われた．

拡張性 (スケーラビリティ) JUMP-1 の分散共有メモリは，1000 プロセッサを超える規模でのスケーラビリティを考慮して設計されている．したがって，計算機システム全体の処理能力の鍵の 1 つである入出力性能も同じようにスケーラビリティをもつものでなければならない．このため，JUMP-1 では，次に示す方針をとっている．

- クラスタ本体と入出力サブシステムを分離
- 全てのクラスタに入出力のためのインターフェイスを装備
- 入出力サブシステムは複数の入出力ユニットで構成し，並列に動作させることによって物理的アクセスを分散
- 入出力ユニット間は Ethernet や STAFF-Link(後述) などで接続

柔軟性 計算機を単にベンチマークのために使うのでなく，様々なアプリケーションを実現するためには，様々な入出力機器を接続できること，また計算機本体が大きくなるため，入出力機器の設置場所が自由に選べることが求められる．したがって，

- 入出力ユニットには汎用のワークステーション (Sun の SPARCstation5 (SS5)) を使用
- クラスタ本体と入出力ユニットの間は伝送距離を長く取ることのできる高速シリアルリンクで接続

アクセス容易性 入出力のための手続きが複雑でなく，簡単にソフトウェアやハードウェアの開発ができること．

8.1.2　STAFF-Link

JUMP-1向けに設計された入出力用のPoint-to-PointリンクがSTAFF-Link(Serial Transparent Asynchronous First-in First-out)[62] である．パラレルリンクでなくシリアルリンクを採用するのは，次の理由による．

- 多数のクラスタと接続するため，ケーブルやコネクタが細く，小さいことが望ましい．シリアルリンクは線も細く，コネクタのピン数も少なくて済む．
- クラスタシステムが大きくなるため，入出力ユニットの設置場所が自由に選べる事，すなわちリンク長の制約が緩いことが望ましい．ノイズ対策等の面でパラレルリンクは不利である．
- 高速なシリアル通信を行うための専用 LSI の出現．

また，STAFF-Link は，図 8.1 に示すように両端のインターフェイスに FIFO をもっており，FIFO にシリアル—パラレル変換を行う TAXI チップなどが接続されている．STAFF-Link へのアクセスは FIFO へのアクセスとして行うことができる．これは簡単なハードウェアで，簡単な入出力ソフトウェアの実現を可能にしている．

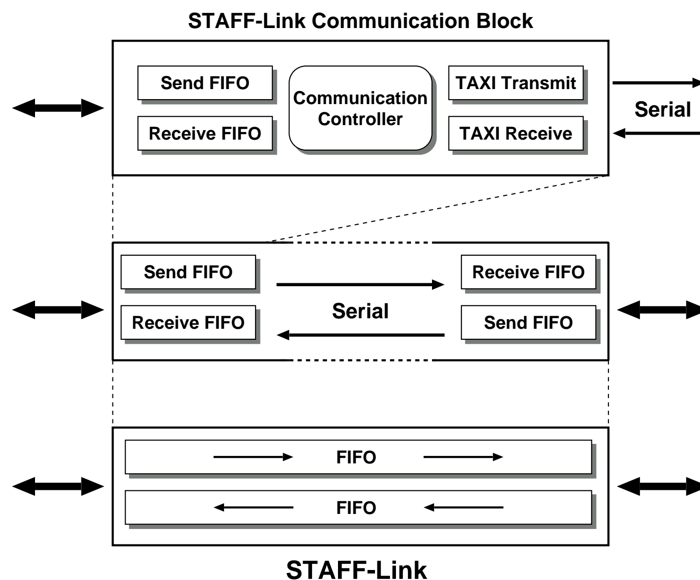


図 8.1 STAFF-Link の構成

STAFF-Link は，次のようなコンポーネントにより構成される．

- STAFF-Link ドータカード
  FIFO, TAXI, およびそれらのコントローラを搭載したカードで，ドータカード間をカテゴリ 5 のツイストペアケーブル (STP) で接続する．
- JUMP-1 STAFF-Link インターフェイス
  JUMP-1 のメモリ管理プロセッサである MBP-light のローカルバス上に接続された，STAFF-Link ドータカードを制御するためのコントローラ．小規模な PLD 2 チップで構成される単純な構成である．
- STAFF-Link マザーボード/SBus カード
  マザーボードは STAFF-Link ドータカードを 4 枚搭載することができ，SBus カードとフラットケーブルで接続する．SBus カードを，入出力ユニットである SS5 の SBus スロットに挿すことにより，SS5 から STAFF-Link へのアクセスが可能となる．

8.1.3　入出力ネットワーク

入出力ネットワーク (図 8.2) は，入出力ユニット間の負荷の分散やデータの共有のために必要である．当初，STAFF-Link での構成を前提としていたが，現在の評価システムでは Ethernet で構成している．

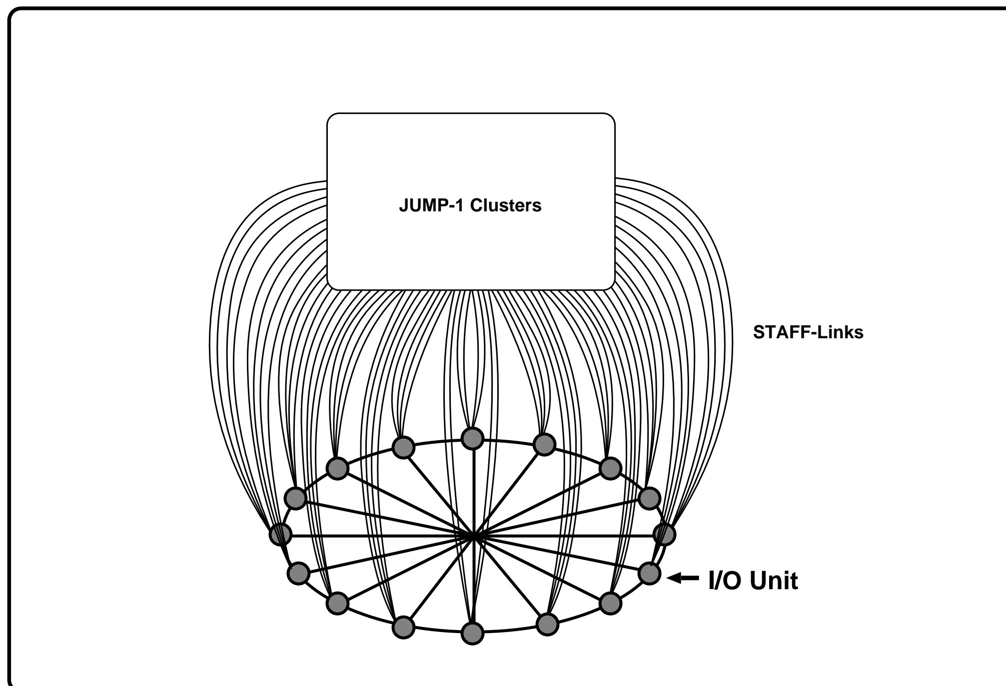


図 8.2 JUMP-1 本体と入出力サブシステム

Ethernet は広く使われているネットワークであり，入出力ユニットとして用いている SS5 でもサポートされている．Ethernet (IP) では，OS の提供する分散ファイルシステムである NFS が標準で使用できる．しかし，Ethernet はブロードキャスト型のネットワークであるため，帯域幅に限界があり，スイッチを用いればある程度解決できるものの，多くは Stored Forward 方式であるため遅延が存在し，性能に悪影響を及ぼす．

STAFF-Link は Point-to-Point のリンクであるため，Ethernet のようなバス型のトポロジをとることができない．したがって，複数の機器を接続するためにはスイッチを接続する必要がある．DSP を用いた STAFF-Link スイッチ機構が既に開発，評価されている．しかし，このスイッチを用いた場合には分散ファイルシステムを独自に実装する必要があるため，今回の実装と評価では Ethernet を用いた．

8.1.4　I/O ソフトウェアの実装

今回実装された I/O ソフトウェアは，I/O サブシステムの評価という目的もあるが，同時に他のプログラムの開発を支援するという意義ももっている．SS5 には様々な I/O デバイスを接続することができるが，評価をとることと，他のプログラムへの流用を考慮して，ファイルシステムの実装を行った．

8.1.4.1　I/O 動作の流れ

JUMP-1 の要素プロセッサである SuperSPARC+が，SS5 にアクセス要求を出すためにはいくつかの手順を経る必要がある．ここではそれについて，順を追って述べる．

(1) コマンドおよび引数の設定 (SuperSPARC+)：SuperSPARC+は，メモリ上に，入出力コマンドとそれに対する引数を，予め決められた順序に従って配置する．

(2) MBP-light の呼出し (SuperSPARC+)：SuperSPARC+は，L2 キャッシュコントローラを通じて MBP-light に要求を送信する．

(3) メモリ読出し (MBP-light)：MBP-light は，SuperSPARC+がメモリに格納したコマンドと引数を取得する必要がある．しかし実際には，これは L2 キャッシュに置かれているため，そこから読み出すことになる．そのため，MBP-light は，L2 キャッシュコントローラに対して読出し要求を送信する．

(4) SS5 へコマンド転送 (MBP-light)：L2 キャッシュコントローラが応答すると，MBP-light に割り込みがかかる．MBP-light は受信されたデータ (32 Byte 単位) をそのまま SS5 に転送する．転送後，結果がくるまで待つ．

(5) コマンド実行 (SS5)：SS5 は，受信したコマンドと引数を解釈して実行し，結果を返送する．

(6) 受信と終了通知 (MBP-light)：MBP-light は，受け取った結果をクラスタメモリに書き込み，メモリ上のフラグを通じて SuperSPARC+に終了を通知する．

8.1.4.2　実装

前節で述べたように，I/O 処理は SuperSPARC+, MBP-light, SS5 で順番に実行されるため，それぞれでのソフトウェアの実装が必要となる．ここで，それぞれの実装について述べる．

SuperSPARC+: MBP-light に要求を送るための関数を実装し，それをベースに様々なコマンドを送り，結果を受け取る入出力関数を実装した．入出力関数はメモリ上にコマンドブロック (32 Byte) とデータブロック (32 Byte の整数倍) を展開し，MBP-light にそのアドレスを渡すことによって SS5 にデータを転送する．コマンドブロックには，コマンドのほかに固定長の引数などを納めることもできる．これらの関数は，SuperSPARC+のセットアップに必要な初期化関数群とともに，JUMP-1 用の C ライブラリに納められている．

MBP-light: MBP-light は，SuperSPARC+用の実行バイナリをメモリに展開したり，また L2 キャッシュコントローラから送られてくるメモリ読出し要求などに応答する必要がある．現在の実装では，STAFF-Link 及び MBIF(Maintenance Bus InterFace) から SuperSPARC+用の実行バイナリを読み込み，実行することができる．今回は，このプログラムを拡張して，SS5 へのコマンド/データ転送を実現した．

SS5: SS5 では，Solaris7 が動作しており，この上で入出力管理プログラムを動作させる．このプログラムは常に STAFF-Link の FIFO ステータスを確認しており，コマンドが到着すると必要な処理を行う．

8.2　I/O 性能の評価

本節では，I/O 性能の評価結果について示す．本節の評価結果は，慶應義塾大学天野研究室の長名によって行われた研究による．

8.2.1　主眼

今回の評価における着目点は 2 つある．

一点目は，複数の入出力ユニットを用いることの効果がどれだけあるのかという，スケーラビリティの観点である．これは，入出力ユニット間のネットワークの転送能力の制約を受ける．もう一点は，STAFF-Link の転送能力に対して様々な入出力処理のオーバーヘッドがどれだけあるのかという点である．これらは，今後のシステムの改良に大きく関わることになる．

8.2.2　システム

評価に用いた機材は，以下に示すとおりであり，図 8.3 のように接続した．

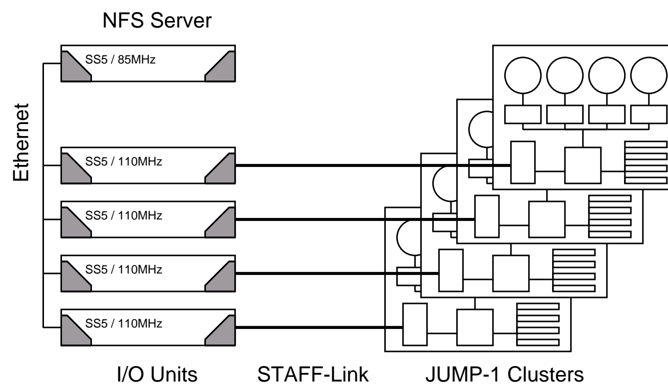


図 8.3 評価システム

- JUMP-1 クラスタ
  4 クラスタを 16MHz で稼働させた．
- SUN Microsystems SPARC Station 5(以下 SS5)
  5 台を用意した．内訳と仕様は次のとおり．

  - NFS サーバ
    MicroSPARC II 85MHz, 160MB RAM
  - 入出力ユニット
    MicroSPARC II 110MHz, 96MB RAM

- その他
  クラスタと入出力ユニットはカテゴリ 5 ツイストペアケーブル (STAFF-Link) で接続した．入出力ユニット，NFS サーバは 10Base-T Ethernet で相互接続した．

8.2.3　方法と結果

次に示す条件で，ファイルに対するシーケンシャルリード/ライトを実行し，転送速度を測定した．Local disk を用いた場合は，各クラスタは同じ入出力ユニットのディスク上の別々のファイルに，NFS を用いた場合は，同じ NFS サーバ上の別々のファイルへのアクセスである．

- Local disk を使用
  - 1 クラスタ，1 入出力ユニット
  - 2 クラスタ，1 入出力ユニット
- NFS を使用
  - 1 クラスタ，1 入出力ユニット
  - 2 クラスタ，2 入出力ユニット
  - 3 クラスタ，3 入出力ユニット
  - 4 クラスタ，4 入出力ユニット

表 8.1 実効転送速度の合計

ディスク	Cluster–SS5	read	write
Local	1 – 1	2.10	3.50
	2 – 1	1.64	0.64
NFS	1 – 1	2.10	3.51
	2 – 2	4.22	7.02
	3 – 3	6.41	10.93
	4 – 4	8.49	14.50

(Mbps)

以上について，ファイル転送の実行転送速度 (表 8.1，図 8.4) のほかに，SS5 や JUMP-1 クラスタでの処理時間を含まない，純粋な転送速度 (表 8.2) を測定した．

表 8.2 STAFF-Link の転送速度

ディスク	Cluster–SS5	`read`	`write`
Local	1 – 1	3.86	12.29
	2 – 1	2.50	12.29
NFS	1 – 1	3.86	12.29
	2 – 2	3.86	12.29
	3 – 3	3.86	12.29
	4 – 4	3.86	12.29

(Mbps/cluster)

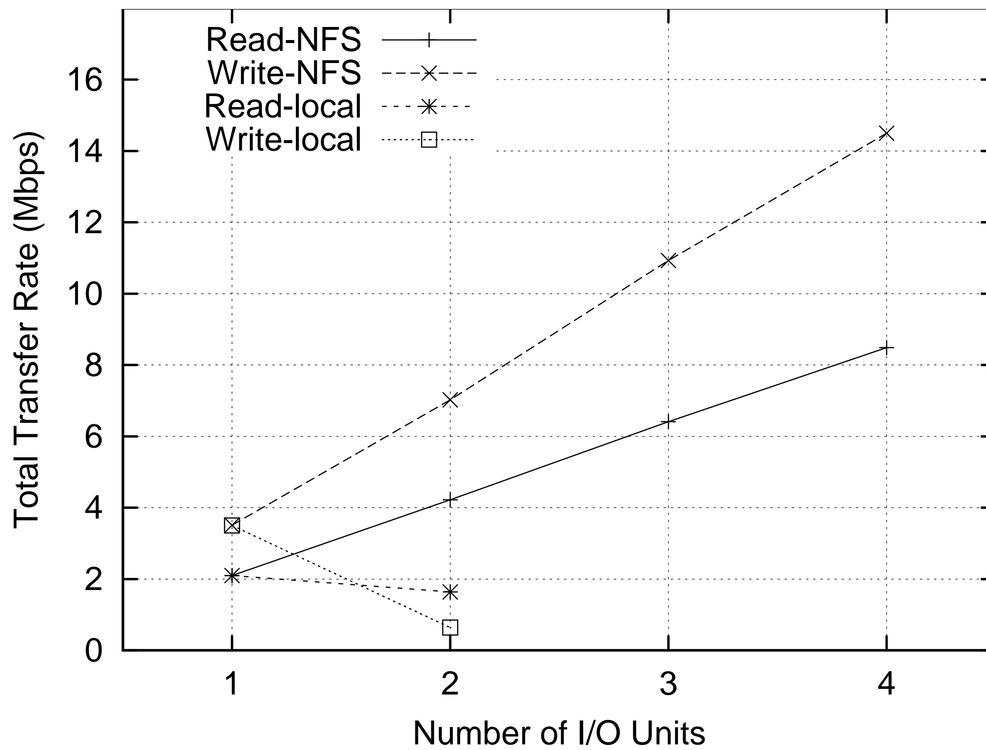


図 8.4 実効転送速度の変化

8.2.4　I/O の評価のまとめ

8.2.4.1　スケーラビリティ

図 8.4 から，単一の入出力ユニットを用いるよりも，複数の入出力ユニットを並列動作させる場合の方が明らかに性能面で有利であることがわかる．

また，入出力ユニットの OS のディスクキャッシュ機構が有効に動作しているため，今回は 4 クラスタ 4 入出力ユニットの 14.50Mbps まで直線的にバンド幅が拡大できたが，同一のファイルに対する複数のクラスタからのアクセスや，大きな単位でのアクセスの集中などに対応するための並列分散ファイルシステムを検討していく必要があると考えられる．

8.2.4.2　オーバーヘッド

表 8.2 から，STAFF-Link のリンク当たりの転送速度は，1 つの入出力ユニットに複数のクラスタを接続した場合を除いて一定であることがわかる．この一定の値を，表 8.1 と比較すると，read で 3 割程度，write で 6 割程度のオーバーヘッドの存在が明らかになる．

これは，JUMP-1 のメモリアクセスの単位は 32 Byte であり，現在の実装では入出力処理全体が 32 Byte の転送をひと区切りとして行われている．これは余計なトランザクションを増やすことになり，性能低下につながる．SS5 上でのファイルの read/write も 32 Byte 単位で行われており，このオーバーヘッドは少なくないものと考えられる．より性能を向上させるには，MBP-light から SS5 への転送ブロックサイズの拡大などを行う必要がある．

第9章　分散共有メモリ(DSM: Distributed Shared Memory)

本章では、超並列計算機 JUMP-1 の DSM(分散共有メモリ) の管理手法について述べ，MBP-light の Core プロセッサによって実行される DSM 管理プログラムの詳細について述べる．その後，分散共有メモリを利用したプログラムによる評価結果を示す．本章の評価は，主に京都大学による研究成果による．

9.1　分散共有メモリ管理

9.1.1　JUMP-1 のコヒーレンス制御

JUMP-1 は多数のプロセッサを接続することを想定しているため，コヒーレンス制御によるメッセージの数が膨大になることが予想される．このため，JUMP-1 ではコヒーレンス制御をクラスタ内とクラスタ間の 2 つに階層化し，効率化を図っている．このように階層化することにより，クラスタ内で閉じるようなデータ共有によるメッセージはネットワークに送信されることはなく，トラフィックの削減が期待できる．

まずクラスタ内でのコヒーレンス制御は，クラスタバスのスヌープによって行われる．クラスタ内では頻繁に制御要求が発生することが予想されるので，高速化のために専用ハードウェアを用いている．クラスタ内でのコヒーレンス制御は MBP-light から不可視であるため，詳細は省略する．

一方クラスタ間のコヒーレンス制御は，ディレクトリ方式で行っている．ディレクトリには縮約階層 bitmap directory[50, 51, 52] を用いることによって，クラスタ数の増大によるディレクトリのビット数の増加を抑えている．

また，クラスタ間のコヒーレンス制御には繁雑な処理が要求されるため，すべてを hardwired-logic で構成することは現実的でないため，分散共有メモリ管理プロセッサである MBP-light のソフトウェアである DSM 管理プログラムによってこれを実現する．

9.1.2　クラスタ間コヒーレンス制御

JUMP-1 では，あるアドレスに関して有効なデータをもつクラスタは 3 種類に分けられる．この 3 種類のクラスタを，次のように定義する．

- **Home クラスタ**: あるアドレスについて，他のクラスタメモリの主記憶として機能するクラスタメモリを所有するクラスタ．有効なデータが存在するクラスタを示すディレクトリを所有する．
- **Owner クラスタ**: 読出し要求に応える義務をもつ最新のデータを所有するクラスタ．
- **Renter クラスタ**: 有効なデータをもつ Home クラスタ以外のクラスタ．

Home クラスタは，アドレスごとに各クラスタに分散配置される．各クラスタには，Home クラスタとして DSM を構成するためのクラスタメモリ領域と，L3 キャッシュ(3 次キャッシュ) として，他のクラスタが Home であるアドレスのデータをキャッシングしておくためのクラスタメモリ領域が存在する．もし Home 以外のクラスタで L3 キャッシュにヒットしなかった場合，Home に要求が送られることになる．L3 キャッシュへのページの割付け及びページテーブルの管理は MBP-light によって実行される DSM 管理プログラムによって行われる．ただし，ページごとコピーすると転送量が膨大になるため，実際の転送はキャッシュライン単位で行われる．

また，JUMP-1 ではライト・バック方式でコヒーレンスをとっているため，L3 キャッシュへの書き込みが生じても Home クラスタのクラスタメモリへの書き込みは即座には行わない．更に，JUMP-1 のコヒーレンス制御はオーナシッププロトコルによって行っているため，最新のデータをもつクラスタである Owner クラスタは，そのアドレスに対する読出しに対して応答する義務をもつ．

次に，クラスタ間のメッセージの流れについて説明する．ある共有されたデータに対するクラスタ間のパケットの転送及び一連の状態遷移を**トランザクション**という．また，要求パケットを送信し，トランザクションを開始したクラスタを **Initiator クラスタ**と呼ぶ．

ここでは，トランザクションの例として読出し要求をとりあげる．まず，プロセッサからクラスタバスを通して MBP-light に送られた要求パケットをもとに，MMC がクラスタメモリアクセスを開始する．そのとき，クラスタメモリにはメモリデータの有効性及び共有状態を示すタグが付加されており，共有状態がクラスタ内で閉じていない場合は MBP Core に処理を依頼する．その後，MBP Core が Home クラスタに対して要求パケットを転送することによってクラスタ間のトランザクションが開始されることになる．

ここで，MBP Core にパケットの制御が移った後のクラスタ間のメッセージの流れを図 9.1 に示す．この例は，Home クラスタ以外のクラスタが Owner となっている場合である．この場合，以下のような流れでトランザクションが進行する．

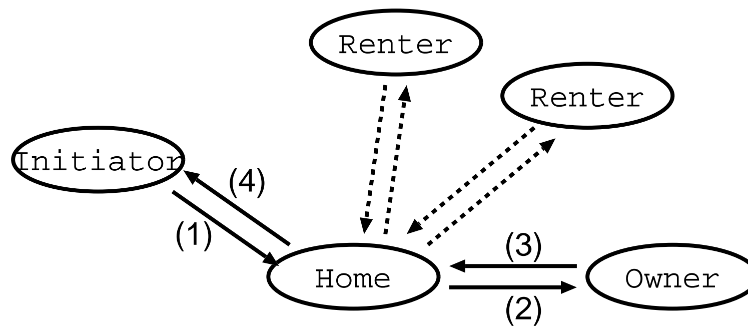


図 9.1 クラスタ間でのパケットの流れ

(1) 読出し要求が発行された Initiator クラスタは Home クラスタに対して要求パケットを送信する．

(2) Home クラスタはその要求に対して Owner クラスタに要求を送信する．

(3) Owner クラスタは応答パケットを Home クラスタに送信する．

(4) Home クラスタはその応答パケットを Initiator クラスタに送信する．

JUMP-1 では，応答パケットは必ず Home クラスタを経由することとし，三角通信は行わない．もし Home 自身が Owner である場合は，ここであげた (2)，(3) の処理は行われない．さらに，JUMP-1 では緩和された (relaxed) メモリモデルを採用しているため，各要求は Home クラスタに到着した順に反映される．

また，更新要求によるトランザクションの場合には，Initiator クラスタから Home クラスタに要求パケットが到着すると，Home クラスタは Owner クラスタだけでなく，Renter クラスタにも更新要求パケットを送信し (図中点線部)，データを更新する．

もしトランザクションが無効化要求であった場合には，各 Renter クラスタに無効化要求を送信することになる．

9.1.3　DSM 管理プログラム

JUMP-1 ではクラスタ間で共有されるデータの管理は全て MBP-light が行う．本節では，MBP-light によって実行される DSM 管理プログラムについて述べる．DSM プログラムは C 言語で記述され，MBP-light 用 C コンパイラ (付録 K) によってコンパイルが行われる．DSM 管理プログラムの実装は，開発メンバーである京都大学によって行われた．

9.1.3.1　DSM 管理プログラムの基本構造

DSM 管理プログラムはキャッシュラインの転送や状態の更新を行う．

プログラムは MBP Core に割り込みがかかった場合に実行され，DSM プログラムは次のような動作を行う．

- パケットを PBR にロードし，ヘッダ部分からパケットの判別を行う．
- 要求パケット及び応答パケットをクラスタバス及び RDT に転送する．
- 共有状態の更新が行われた場合にはクラスタメモリのタグの変更を行う．
- 更新要求及びクラスタ間でのデータ共有を行う読出し要求はクラスタメモリへの書込み及び更新を行う．
- クラスタ間のコヒーレンス制御ディレクトリ及びページテーブルの管理を行う．

また，DSM 管理プログラムでは最終的なキャッシュデータの値が各キャッシュ間で異ならないよう実行順序に関して以下のようなことを考慮する必要がある．

- 同一アドレスに対する要求パケットの追越しが起こってはならない．
- MBP-light によるクラスタ内での処理は atomic に行わなければならない．

9.1.3.2 節では要求の追越しを防ぐために用意したペンディング・トランザクション・テーブル (PTT) について述べる．その後，9.1.3.3 節ではテーブルを利用することによりアトミックな処理を行う方法とその例外について述べる．

9.1.3.2　ペンディング・トランザクション・テーブル (PTT)

要求パケットの送信を行い，応答パケットの受信を待つ間に当該アドレスに対する他の要求パケットが到着する場合がある．その場合の競合を避けるために DSM プログラムでは以下のようなペンディング・トランザクション・テーブルを用意する．このテーブルはアドレスをキーとする連想テーブルである．以下，ペンディング・トランザクション・テーブルを PTT と記述する．

要求パケットが MBP-light に到着した場合には PTT を検索する．当該アドレスに対して処理がなされていなければ，PTT に登録を行い，パケットに対する処理を実行する．このときすでに登録があれば，待ちパケット (中断中パケット) としてリストに追加する (図 9.2)．待ちパケットはすでに登録されていた要求パケットの応答パケットが到着すると直ちに実行状態に移される．これによって MBP-light に同一アドレスに対する要求が到着した場合にも処理の順序は変わることはない．

要求到着時にすでに待ちパケットの登録がある場合には，待ちパケットの登録を再帰的に行えばよい．

PTT には，要求を発行したデバイス ID，要求の種類及び待ちパケットとして登録する場合にはパケットを格納している PBR へのポインタ等を登録する．

9.1.3.3　パケットのすれちがい及び追越しの防止

9.1.3.2 節で述べた PTT への登録により要求パケットの追越しを防ぐことが可能であることは示された．しかし，PTT は応答パケットが到着するとエントリを削除するために，更新要求などの応答パケットを必要としない要求パケットは登録されない．このような要

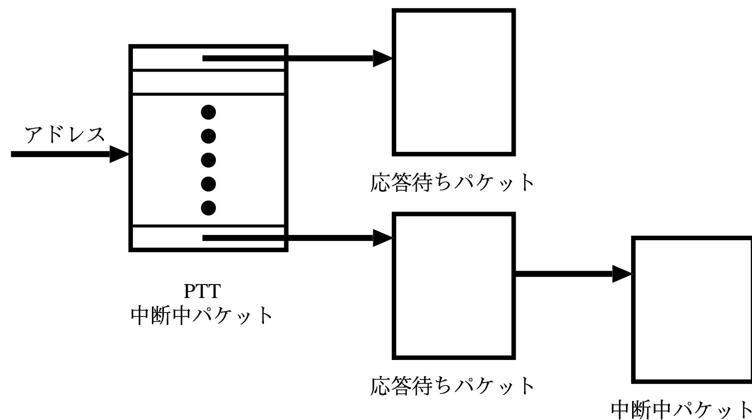


図 9.2 ペンディング・トランザクション・テーブル (PTT)

求の場合には，パケットの処理を行う前に PTT を検索し，当該アドレスに処理中のパケットの有無を確認してから処理を開始する．処理中のパケットが存在している場合には PTT に登録しておき，処理中の要求の応答パケットが到着し処理が終了した後に，PTT から削除し要求が実行される．

ところが，次のような場合には更新要求のすれちがいが起こり，クラスタ内のキャッシュとクラスタメモリの内容が異なってしまう．

図 9.3 にすれちがう場合の状態を示す．図は RDT から受信した更新要求パケットを処理している間にクラスタバスに更新要求が流れ，MMC のバッファに格納されている状態である．

(1) 他のクラスタからの更新要求が到着し，DSM 管理プログラムはパケットの処理を開始する．

(2) クラスタ内の Cache Controller(以下，CC とする) から更新要求が発行され MMC のクラスタバス受信バッファに要求が格納される．

(3) DSM 管理プログラムによってクラスタメモリが更新され，クラスタバスに要求を送信するために MMC の送信バッファに要求を格納する．

(4) 他のクラスタで発行された更新要求がクラスタバスに流れる．

(5) DSM 管理プログラムはクラスタバス受信バッファからパケットを取り出し処理を開始し，プログラムによってクラスタメモリの値は更新される．

CC の更新はクラスタバスに流れた順に更新される．ところが上記の例の場合であるとクラスタメモリの更新順序と異なることが分かる．

そこで，DSM プログラムでは RDT Router からの更新要求を処理する場合には，クラスタバスにロックをかける必要がある．

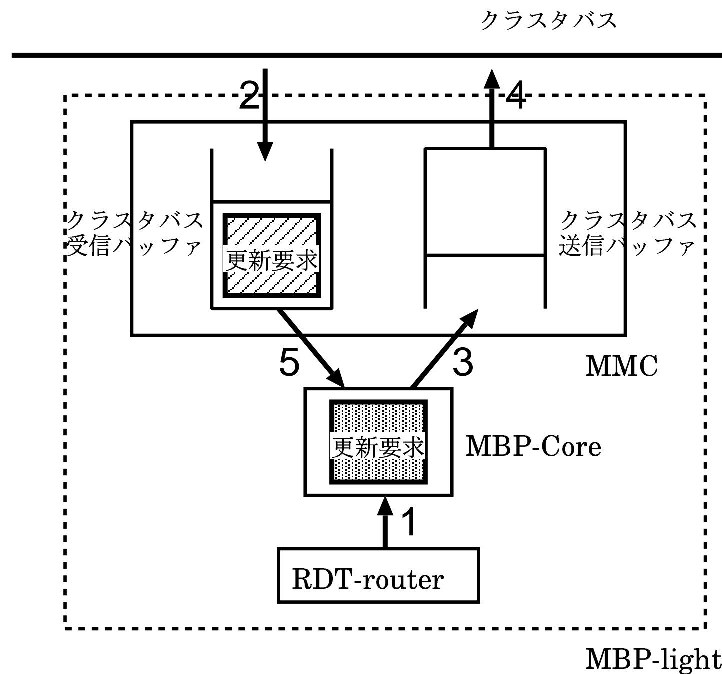


図 9.3 更新要求のすれちがい

9.1.3.4　テーブル

Core プログラムが使用する主なテーブルについて挙げる．

クラスタメモリ上　クラスタメモリ上に確保されているものは，以下の通りである．

- **ページテーブル:** 仮想 → 物理アドレス変換時に参照する．ページフォールト時のエントリの追加，更新等は SPARC によって管理しており，Core からは read のみを行う．SPARC MMU も使用するため，データ構造はその形式に従っている．
- **逆引きページテーブル:** 物理 → 仮想アドレス変換時に参照する．これも上と同様，エントリの追加，更新は SPARC によって管理され，Core からは read のみ行う．
- **ディレクトリ:** キャッシュラインの共有状態を示すディレクトリ．詳細は後述する．

ローカルメモリ上　ローカルメモリに確保されている主なテーブルは以下のとおりである．

- **ptt_array:** ライン単位のアドレスをキーとしたテーブルである．PTT として競合検出に使うのみでなく，ディレクトリ TLB としても機能する．これは，PTT 検索とディレクトリ TLB 検索を同時に行うためであう．ディレクトリのデータ構造については後述する．
- **tlb_array:** アドレス変換を高速化するため，ページテーブルの内容をキャッシングしておくテーブル．キャッシングする際，Home の情報を収集し格納しておく．Home に関するデータ構造は次のとおりである．

home_or_not: 自クラスタが Home か否かのフラグ.

to_home_map: 自クラスタ →Home へ送信する際に，RDT ヘッダに埋め込むべき宛先．自クラスタが Home なら意味をなさない.

from_home_map: Home→ 自クラスタへ送信する際に，RDT ヘッダに埋め込むべき宛先．自クラスタが Home なら意味をなさない.

実際のデータ構造では，map は rank1，rank0 と分けている.

- **reverse_tlb_entry:** 逆引きページテーブルの内容をキャッシングしておくテーブル．Home の情報などが，tlb_entry と同様にして格納されている.

9.1.3.5　ディレクトリ

ここでは，ディレクトリのデータ構造について説明する.

DSM 管理プログラムは，クラスタメモリ上にあるディレクトリを一旦ローカルメモリ上の TLB にキャッシングしてから扱う．したがってここでは，ディレクトリ TLB についてのみ説明し，クラスタメモリ上の本体とのキャッシング，書戻しの詳細については省略する.

ディレクトリ TLB は PTT と統合されている．PTT 検索が終った時点で，そのラインのディレクトリ TLB へのキャッシングと検索は済ませ，パケット実行時にすぐ参照，更新することができるようにしている.

以後，このディレクトリ TLB を単にディレクトリと呼ぶ.

ディレクトリにおける，ライン毎のエントリは以下のデータ構造を持つ.

stag: Invalid，Local，Global の状態を取る．クラスタメモリの当該ラインの stag と同じ状態になるようにしている.

owner_or_not: Home が Owner か否かのフラグ.

map: RDT ヘッダに埋め込むべき宛先．上の stag の状態によって意味が変わる.

stag=Invalid: Home→Owner への宛先.

stag=Global: そのラインを共有している全クラスタへマルチキャストする際の宛先.

stag=Local: この場合，エントリは意味をなさない.

return_num: そのラインを共有しているクラスタ数.

実際，パケット実行時に使用するときは，stag，owner_or_not を参照することで，ラインの共有状態を知り，map をそのまま RDT ヘッダに埋め込むことで RDT 側への転送を行う.

実際のデータ構造では，map は rank1，rank0 と分けている.

9.2　JUMP-1 のバグ

JUMP-1 は，以下の致命的なバグにより，アプリケーションプログラムを用いた詳細な評価を行なうことができなかった．

- MMC のバッファ内のデータが，特定の条件で上書きされる/ヘッダが入れ替わる状況が発生するため，外部でシリアライズを行って MMC のバッファが空になるまで次の処理を行わせないようにする必要がある．このため，処理がシリアライズされ，著しく性能が低下する．
- キャッシュコントローラのバグにより Update プロトコルが動作しない．

このバグは，キャッシュコントローラおよび MMC の周辺の設計が複雑すぎたこと，キャッシュコントローラ側を設計した京大，MMC 周辺を設計した東大間の意思疎通の悪さが大きな要因となっている．また，複数 CPU と MBP-light を接続した状況でのシミュレーションが不十分であった点が，設計時にバグが発見できなかった要因である．

9.3　分散共有メモリの評価

本節では，京都大学によって実装が行われた DSM 管理プログラムの評価結果について述べる．本評価結果は京都大学による研究成果から引用したものである．前述のとおり，いくつかのバグにより大規模な評価は行えないことや，Update プロトコルが利用できない等の制限があるため，無効化プロトコルを用いた限定的な評価となっている．

以後，まず基本的なトランザクションの処理時間について示し，次に行列積プログラムを実行した際の評価結果について示す．

9.3.1　トランザクションの処理時間

表 9.1 基本的なトランザクションのレイテンシ

	読出し		無効化	
	Home read	Owner read	Home read	Owner read
→ (1)	107(93/ 14)	107(93/ 14)	107(93/ 14)	107(93/ 14)
(1) → (2)	—(—/ —)	119(119/ 0)	—(—/ —)	108(108/ 0)
(2) → (3)	—(—/ —)	67(50/ 17)	—(—/ —)	104(104/ 0)
(3) → (4)	148(131/ 17)	94(80/ 14)	120(106/ 14)	147(130/ 17)
(4) →	81(64/ 17)	116(116/ 0)	81(64/ 17)	81(64/ 17)
クラスタ内 HW	45(0/ 45)	90(0/ 90)	45(0/ 45)	90(0/ 90)
クラスタ間 HW	88(0/ 88)	176(0/176)	88(0/ 88)	176(0/176)
合計	469(288/181)	850(522/328)	441(263/178)	813(499/314)

基礎評価として，表 9.1 に基本的なトランザクションの処理時間を示す．表中のカッコ内は，それぞれ (Core プログラム/それ以外のハードウェア) の処理サイクル数を示す．JUMP-1 のトランザクションの基本的な流れは次のとおりである．

(1) Initiator が要求パケットを Home へ送信

(2) Home はディレクトリを検索し，Owner/Renter へ要求パケットを送信

(3) Owner/Renter は要求に対するパケットを Home に送信

(4) Home は Initiator へ応答パケットを転送

表中の左側の覧の (1)〜(4) はこのトランザクションの各フェーズを示す．例えば，(1)→(2) は，要求パケットが Home に届いてから Owner に向けて送信するまでの MBP Core の処理を表す．表中の Home read は，Home クラスタの主記憶でヒットし，フェーズ (2) と (3) が省略できる場合である．ディレクトリ検索の結果，共有状態が global shared であり，主記憶に有効なデータが存在すると判断した場合を表す．一方，Owner read は Home でミスし，Owner が応える場合である．Owner では，要素プロセッサのキャッシュに必要なデータが保持されているため，クラスタバスに要求を転送する必要がある．表中 (3)→ は，Home からのパケットを受け取ってからクラスタバスへ転送する操作を，→(4) はクラスタバスから応答を受け取った後，Home に転送する操作を表す．クラスタ内 HW は，要素プロセッサの命令パイプラインがロード/ストア命令実行してから MBP Core に割り込みがかかるまでの時間と MBP Core が応答パケットの転送を MMC に依頼してから要素プロセッサの命令パイプラインが再び動き出すまでの時間の合計である．Owner が応える場合には，これに Core が要求をクラスタバスに送信してから，その応答が Core に割り込みをかけるまでの時間が加わる．クラスタ間 HW は，クラスタ間の RDT ネットワークのパケット転送時間を表す．

表 9.2 は，読出し要求が Home でヒットした場合のサイクル数の内訳である．このトランザクションのサイクル数は 469 サイクル，50MHz 動作時のレイテンシは 9.38μsec となる．このうち Core の処理時間は，288 サイクルとなり，それ以外のハードウェアが動作している時間に対する割合は 159.1%となる．

9.3.2　行列積プログラムによる評価

本節では，倍精度実数の 512×512 エントリの行列積を計算する場合の実行性能について述べる．ここでは，1 クラスタあたり 1 または 2 プロセッサを用い，最大 8 プロセッサまでのシステム構成における評価を行った．

なお，データは各クラスタに均等に配置されており，行列積プログラムは Owner-Computes Rule で計算を行うため，起こり得るクラスタ間トランザクションは 9.3.1 節における Home Read のみとなる．また，各階層でのキャッシュヒット率を向上させるために多重のタイリング [63] を施してある．

表 9.3 に，プログラムの実行時間，Core の稼働時間，3 次キャッシュのヒット率を示す．評価環境の PE は 1 クラスタあたりのプロセッサ数，CL はクラスタ数を示す．

表 9.2 読出し処理のレイテンシの内訳

処理内容		サイクル数
→ (1)	送信バッファの確認	8(8/ 0)
	MMC からのパケット転送	24(10/ 14)
	アドレス変換，Home の検索	34(34/ 0)
	パケット種類判別	10(10/ 0)
	ヘッダ作成	31(31/ 0)
	小計	107(93/ 14)
(1) → (4)	送信バッファの確認	13(13/ 0)
	PTT，ディレクトリ検索	34(34/ 0)
	アドレス変換	29(29/ 0)
	パケット種類判別	10(10/ 0)
	主記憶読出し	27(10/ 17)
	ヘッダ作成	35(35/ 0)
	小計	148(131/ 17)
(4) →	送信バッファの確認	8(8/ 0)
	PTT 検索，アドレス変換	34(34/ 0)
	パケット種類判別	10(10/ 0)
	ヘッダ作成，送信	10(10/ 0)
	MMC へのパケット転送	19(2/ 17)
	小計	81(64/ 17)
クラスタ内 HW		45(0/ 45)
クラスタ間 HW		90(0/ 90)
合計		469(288/181)

表 9.3 行列積プログラムの評価

システム構成 (PE×CL)	実行時間 ($10^6 cycle$)	Core 稼働時間 ($10^6 cycle$)	3rd hit(%)
1×1	676.63	0	100(%)
2×1	338.73	0	100(%)
1×2	351.06	13.216	99.953(%)
2×2	181.64	13.386	99.905(%)
1×4	188.41	15.914	99.858(%)
2×4	102.37	16.527	99.716(%)
1×8	108.90	17.134	99.669(%)

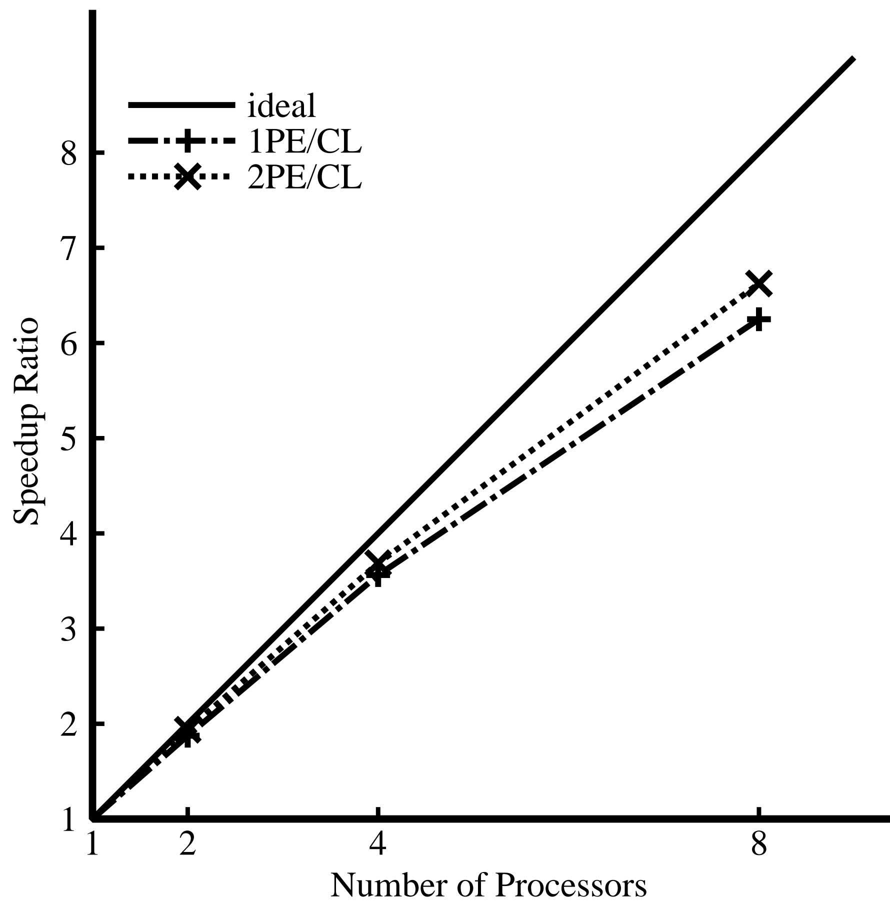


図 9.4 台数効果

図 9.4 は，行列積プログラムを実行した場合の台数効果についてまとめたものである．これは，表 9.3 における 1 プロセッサ (1PE × 1CL) の実行時間を 1 として，そのスピードアップの割合をプロットしたものである．

表 9.3 に示すとおり，3 次キャッシュのヒット率が高いため，プロセッサ数が少ない場合はリニアなスピードアップからの性能低下分は比較的小さくなっている．2 クラスタの場合を検討すると，(1PE/2CL) の場合は 1 プロセッサの場合の 1.93 倍の実行速度，Core の起動時間が実行時間に占める割合は 3.7% となっている．(2PE/2CL) の場合，3.72 倍の実行速度，Core 起動時間の割合は 7.3%である．8 プロセッサの場合はデータが細かく分断されることや，全体の実行時間が小さくなるために Core の起動時間が相対的に大きくなるために，リニアなスピードアップからの性能低下率はプロセッサ数が少ない場合に比べて大きくなる．実行速度は，1 プロセッサの場合に比べて (2PE/4CL) の場合で 6.61 倍，(1PE/8CL) の場合で 6.21 倍の性能向上を達成している．Core の起動時間の割合は，(2PE/4CL) で 16.1%，(1PE/8CL) で 15.7% となっている．

第10章　JUMP-1プロジェクトのまとめ

本章では，本研究で得られた結果を踏まえ，JUMP-1 プロジェクトを通して得られた知見についてまとめる．まず JUMP-1 の新しい技術に関する評価についてまとめ，その後，プロジェクトを進めることによって明らかになった，今回のプロジェクトで問題となった点，プロジェクト運営上注意すべきであると思われる点についても述べる．

10.1　JUMP-1の新しい技術の評価のまとめ

JUMP-1 は野心的なプロジェクトであり，様々な新しい技術の提案を含んでいた．本論文における定量的な評価を踏まえて，これらの技術に関する検証結果をまとめる．

10.1.1　大規模なCC-NUMA用の分散メモリ管理方式

10.1.1.1　キャッシュライン管理

ページ単位で管理し，転送はキャッシュライン単位で行う．主記憶の一部を L3 キャッシュとして用いる．このことにより細粒度の並列処理と粗粒度の並列処理を共に効率良く行う．

本研究ではキャッシュと MMC のバグにより大規模なアプリケーションが稼働せず，きちんとした評価は取れていないが，第 9 章に示した実アプリケーションによる評価結果，京大によるシミュレーション評価結果 [49] によるとほぼ良好な結果が得られている．主記憶の一部を他のノードの L3 キャッシュとして用いる方法は，PC クラスタの分散共有メモリ管理システムにも用いられている．JUMP-1 の設計者の一人である松本は後に SSS-Core で，PC クラスタ上のソフトウェアによりこれに近い手法を実現して，良好な成果を得ている [64]．転送の単位をより細かくする方法は，現在の PC クラスタでは実現が難しいが，田辺らはメモリバスに直接ネットワークインタフェースを接続する DIMMnet-1[65] における AOTF により，これに近い効果を実現している．ただし，AOTF は分散共有メモリではなく，細粒度のメッセージ転送である点が異なっている．DIMMnet などの PCI バスなどを介さずに CPU とネットワークインタフェースを密接続する方法が確立されれば，このような手法は一般的に用いられる可能性がある．

10.1.1.2　積極的なデータ共有

Update プロトコルを用いると共に RDT の強力なマルチキャスト，応答パケット収集能力を用いて，大規模分散共有メモリにおいて積極的にデータ共有を図る．

本研究では，キャッシュコントローラのバグにより，現在のシステムでは全く評価を取

ることができなかった．寺沢ら [66][67] のシミュレーション結果によると，小規模なシステムではこの方法はあまりうまく行っておらず，大規模における評価は現在でも行われていないが，望み薄である．この点に関しては，JUMP-1 のプロトタイプは全く評価が取れなかったという点で失敗である．

10.1.1.3　RHBD 方式によるディレクトリ管理

CC-NUMA のディレクトリ管理方式としては，RHBD 方式提案の後，複数のディレクトリ中のビットをまとめて代表させる coarse vecrtor によるディレクトリ管理法 [68] が提案され，確立された．RHBD 方式はこれらの先駆けをなすと共に，近接ノードに対する配慮がなされている点で独特であった．現在のプロトタイプではサイズが小さいため，この有効性を充分なサイズで実機確認するには至らなかったが，第 6 章での評価結果が示すように MBP-light および RDT ルータの収集機能，無駄パケットの自動廃棄機能により，ネットワークは無駄パケットで混雑しないことは実機により確認されている．なお，この手法は，スイッチ結合型計算機 SNAIL-2 のキャッシュ制御機構 MINC[69][44] に応用され，その効果と問題点が明らかになり，現在その改良手法である MINDIC[70] が提案されている．

10.1.1.4　MBP-light による制御

分散共有メモリの制御を専用プロセッサにより行うか，ハードウェアで行うかは，当時から議論が分かれていた．DASH, Alewife, Origin では，これを完全にハードウェア化し，FLASH，NUMA-Q では JUMP-1 同様，専用プロセッサを設けた．結論として，ソフトウェアの介在がもたらすアクセスレイテンシの増加が無視できず，CC-NUMA 型マルチプロセッサではハードウェア制御が主流となった．しかし一方，PC Cluster に用いられるネットワークインタフェースでは，ハードウェアによる強力な DMA 機能を専用プロセッサのソフトウェアが制御する形式が一般的となった．Myrinet における LaNai[71], QsNet における Elan[72] がこの代表である．超並列計算機 RWC-1 の後を受けて，RWCP および慶應義塾大学で開発された RHiNET[73] のネットワークインタフェースである Martini[74] は，MBP-light の反省からデータ転送の主たる操作をアドレス変換からプロテクション管理に至るまで完全にハードウェアで実現する一方，内蔵プロセッサはハードウェアを「乗っ取る」形で例外処理を実行するアーキテクチャ[75] を確立して効果を挙げている．この点で MBP-light のソフトウェアによるプロトコル制御は，CC-NUMA の制御方法としては適切でなかったが，PC Cluster 用のネットワークインタフェースの設計に対して知見を与えたと言える*.

10.1.1.5　MBP-core の Buffer-Register Architecture

第 5 章の評価結果により，このアーキテクチャは設計者の意図通りに性能に貢献していないことが明らかになった．これは，一つは京大の設計した分散共有メモリ管理プロ

*本来の MBP の構想はプロトコルの大部分をハードウェアで実行するものであり，むしろ Martini の考え方に近かった．これが MBP-light で実現できなかった主たる理由は，チップ容量の制限というよりも，プロトコルの制御方法，特にテーブルフォーマットが設計時に確定していなかったことによる．

グラムがこのアーキテクチャを使いこなしていないことが原因ではあるが，逆に考えると使い難いアーキテクチャであったのかもしれない．これはパイプラインに組み込まれた Buffer-Register の規模が中途半端で，パケットの本格的なバッファとして使うことができなかった点が大きい．パケットを格納可能なメモリとして現在のバッファ同様の広いビット幅を扱えるようにし，キャッシュの形でパイプラインに組み込む方式を採用すべきであったと言える．このことにより，第 5 章で評価したように，データと命令の衝突もなくなり，性能を向上させることができる．分散共有メモリ管理プログラムの開発と MBP-light の開発を同時期に，かつあまりコミュニケーションのない状態で行ったのが失敗の主因であろう．

10.1.2　結合網 RDT と RDT ルータ

第 6 章の評価結果により，RDT およびルータチップの性能は当初目的をほぼ達成したと考えられる．このネットワークは以下の点でいずれも世界で初めての実現であった．

(1) メッシュベースの超並列計算機用接続網の中で実機が稼働した．

(2) ソースルーティングでマルチキャストを実現した．

(3) ルータチップレベルで応答パケット収集 (Conbine 機構) を実現した．

特に，(2)，(3) は今後様々なネットワークで利用される可能性がある技術であり，その実機評価は貴重なデータであると言える．さて，RDT 以降しばらくの間超並列計算機用ネットワークの研究は盛んになったが，ハイパフォーマンスコンピューティングの主役が大規模マルチプロセッサから PC クラスタに移るにつれ，結合網の研究は SAN(System Area Network) に移って行った．SAN は，ある程度の不規則性を許容することから，トポロジ自体が問題にならなくなっている．このことから RDT の評価結果は，むしろそのトポロジの興味深さよりも，本論文で評価した，マルチキャスト能力，パケット収集能力，無駄パケット自動廃棄機構等が今後共有益であると考えられる．

10.1.3　メインテナンスシステム

第 7 章に示すように，メインテナンスシステムの主な目的は JUMP-1 の運転監視を行うことである．メインテナンスシステムでは，ホスト PC からの MBP-light や SPARC のプログラム転送，JUMP-1 の起動や制御が可能となっており，必要な機能はすべて実現できたと言える．メインテナンスシステムでは，プログラムの転送機構を利用した，ホスト PC との間のデータ送受信も可能であるが，転送速度はあまり高くない．転送速度に関しては更に改善することも可能であったが，実際のアプリケーションでの I/O 処理は STAFF-Link が担当するため，実用上問題になることはない．

10.1.4　I/O

第 8 章の評価結果により，JUMP-1 の STAFF-Link は，ディスクの並列アクセスによりスケーラブルな性能向上を実現可能にすることが明らかになった．シリアルリンクを

多数用いて I/O 用ネットワークを構築する JUMP-1 の I/O は，現在の SAN(Server Area Network) に通じる先駆的な手法と言える．ただし，近年のディスク能力，サーバの能力の増加に対応するためにはシリアルリンクでは力不足であり，強力な性能をもつ Infiniband などが普及をはじめている．STAFF-link の主要デバイスであるシリアルインタフェースである Taxi は，既に販売を終了している．

10.2　JUMP-1 の実装遅れとサイズ縮小

JUMP-1 は，1994 年に本格的に実装が開始された当初の予定では，1996 年 3 月に 64 ノード 256 プロセッサのシステムを稼働させるはずであった．しかし，現実には 2000 年 3 月に 16 ノード 64 プロセッサのシステムが稼働するに留まった．すなわち 4 年遅れてスケールが 1/4 にダウンしたことになる．この原因として以下の点が指摘された．

- 多くの大学が設計に関与したため，大学間の意見および設計方針の相違がプロジェクトの遅れにつながった．いわゆる「船頭多くして舟，山に登る」であった．
- 日本の大学の実装能力が低かった．

これらの見解は俗耳に入りやすいが，プロジェクトに関与した者としては，事実に反するとしか思えない．大学間の意見の相違および意思疎通の悪さは，設計段階での遅れにつながり，最終的なバグの原因にもなったが，深刻な意見の食い違いを起したことは少なくとも筆者の関与して以来は起きていない．また，JUMP-1 はスーパコンピュータレベルの実装技術ではなく，通常の WS/PC レベルの実装技術で賄うこと，というのが設計の大方針であった．当時，SuperSparc+の動作周波数は 50MHz であり，大学の実装能力で充分対応できるものだった．

実装の遅れの主因は，キャッシュコントローラ，バスチップ，MBP-light のチップ設計実装の遅れにあった．これは，当時のチップ開発環境がメインフレームによる社内 CAD から WS による標準 CAD への過渡期に当たり，現在の状況からは考えられない程劣悪であったことによる．これらのチップは東芝の ASIC 事業部で開発されたが，以下の CAD を用いた．

- 記述は VHDL でシミュレーションは Mentor Graphics 社の Quicksim
- 合成は Mentor Graphics 社の Autologic
- 合成後のシミュレーションとテストベクトル生成は東芝が WS 用に開発した VLCAD

これらの全ての CAD は，実際は安定動作をしておらず，結局，他の CAD に淘汰され，これから 1 年以内に使われなくなった．特に VLCAD は最終的に全機能が装備されないうちに廃棄され，東芝自体が ASIC ビジネスから撤退するに至った．

実際，チップのデータインが終了した 1998 年以降，筆者が主体になって行った基板実装，メインテナンス系の設計実装，筐体設計，組上げ，調整作業は順調であり，一年で 16 ノードのプロトタイプ 1 号機の稼働に成功し，次の一年で 64 ノードのプロトタイプ 2 号機の稼働に成功した．サイズの縮小は純粋に予算不足に起因した．技術的には 4 倍のサイ

ズの構築に全く問題はなかったが，2000年に至っては，新たな予算を確保する程の意義が見い出せなくなっていた．

また，JUMP-1プロジェクトの遅れと規模縮小は，米国の著名な大型計算機のプロジェクトと比べて必ずしも大きいとは言えない．Illinois大学のCedarプロジェクトは5年以上遅れ，サイズは32プロセッサで当初予定の1/8であった．成功したプロジェクトとして知られるStanford大学のDASHですら3年以上遅れ，最終稼働システムは16プロセッサで当初予定の1/4のサイズであった．この点を考えると，64プロセッサのシステムの稼働に成功したのは，これらのプロジェクトに比べてむしろ優れた成果と言えることがわかる．

10.3　大学間共同プロジェクトの問題点

JUMP-1プロジェクトの大学間連合における最大の問題点は，メインとなるシステム設計を東大で行うのか，京大で行うのかがはっきりしていなかったことにある．結局，東大はMBP，京大はCPUおよびキャッシュの設計という分担に決定し，バスのプロトコルをきちんと定めることで，両者の共同作業を可能にする，という方法を取った．しかし，ここでの設計方針の相違，意思疎通の悪さが，設計段階での遅れとバグの大きな原因になった．

本来，ノードアーキテクチャを2つに分割して設計することは極めて困難であり，作業量の多さから分担する必要がある場合でも，グランドデザインはどちらかの大学で行い，実装レベルで作業分担を行うべきであった．これに対して，結合網を担当した慶應大，東京工科大，I/O，グラフィックを担当した神戸大，岡山理科大は，作業分担が明解であったため，早めに設計，実装を終え，成果を挙げることができた．また，筆者がプロジェクトに関わって以来，実装の主体が慶應大に移り，指揮系統と分担関係が確立したため，急ピッチで実装が進むようになった．

しかし，このような問題点にも関わらずJUMP-1が稼働に成功したのは，東大の松本，平木ら，京大の中島，森，五島らが議論の末に確立したJUMP-1の設計目標とアプローチ自体が技術的に見て興味深いものであり，かなりの部分，正しいものであったことによる．設計当初に揉めたことも悪いことだけではなかったのである．

JUMP-1以後，大学や研究機関間の協力プロジェクトが盛んになったが，この反省を踏まえ，メインとなる部分は1つの大学が設計し，他の大学は自分の得意分野でこれを補助するという分担形式が定着した．これもJUMP-1の1つの成果と言える．

第11章　結 論

本研究では，2000年3月に完成したJUMP-1の実機を用いて，RDTネットワークの応答パケットの自動生成，自動収集機構に関する評価，MBP-lightの命令セットアーキテクチャの評価等を行った．本論文では，これらを含めてJUMP-1プロジェクトで得られた知見についてまとめた．

11.1　本研究の成果

筆者は，1998年よりJUMP-1プロジェクトに参加し，実機の実装および評価を行った．筆者が参加した1998年当時，JUMP-1のクラスタボードの試作が完了しており，クラスタボードの実機検証が開始されようとしていた．筆者は，この実機検証を可能とするべく，ホストPCとクラスタボードを接続し，JUMP-1の制御を可能とするメインテナンスシステムの設計と実装を行った．メインテナンスシステムでは，ホストPCからのMBP-lightやSPARCのプログラム転送，JUMP-1の起動や制御が可能である．このシステムの完成により，以後クラスタボードの検証や修正，およびJUMP-1システムの運用が可能となった．メインテナンスシステムでは，プログラムの転送機構を利用した，ホストPCとの間のデータ送受信も可能であるが，転送速度はあまり高くない．転送速度に関しては更に改善することも可能であったが，実際のアプリケーションでのI/O処理はSTAFF-Linkが担当するため，実用上問題になることはない．

その後，2000年に64プロセッサのJUMP-1システムが稼働し，以後ソフトウェアの開発と，実機を用いた評価が行われた．

この年，筆者はJUMP-1のネットワークであるRDTネットワークについて，様々な機構の評価を行った．RDTネットワークは，トーラス構造と階層構造を併せ持ち，小規模なシステムから大規模なシステムまで，スケーラブルに対応できるという特徴をもつ．またDSMを効率良く管理するため，マルチキャスト機構や，応答パケットの自動収集，自動生成機構を持つ．RDTネットワークの様々な機能は，RDT Router chipとメモリコントローラであるMBP-lightによって実現されている．MBP-lightは内部にMBP Coreと呼ばれるCoreプロセッサをもち，そして高速化を要求される部分にRDT Interfaceと呼ばれるhardwired-logicを設けている．RDT Interface内には，Ack Generatorと呼ばれる応答パケットの自動生成機構が実装され，マルチキャストに対する応答を高速に行うことができるようになっている．また，RDT Interface内には応答パケットを自動収集するためのAck Collectorも実装されている．このAck Collectorと，RDT Router chip内の応答パケット自動収集機構によって，hardwired-logicによる高速な応答パケット収集が行われる．従って，MBP Coreが応答パケットの収集のために起動する可能性が低減され，パフォーマンスの改善が期待される．

これらの機構について，本研究で行った評価より，以下に示すような結果を得る事ができた．

まず，RDT ネットワークのバンド幅は，最大パケット長である 15 フリットの場合で 100MBytes/sec となる事が分かった．また，RDT ネットワークに装備された，応答パケットの自動生成機構はソフトウェア処理に比べて 1.8 倍程度，自動収集機構は 2.4 倍程度高速である事が分かった．また，それぞれの機構に要するハードウェア面積は，チップ全体から見れば数%内外で済んでいるものが多く，これらの機構はさほどハードウェアコストを必要としないことがわかる．

このネットワークは以下の点でいずれも世界で初めての実現であった．

(1) メッシュベースの超並列計算機用接続網の中で実機が稼働した．

(2) ソースルーティングでマルチキャストを実現した．

(3) ルータチップレベルで応答パケット収集 (Conbine 機構) を実現した．

特に，(2)，(3) は今後様々なネットワークで利用される可能性がある技術であり，その実機評価は貴重なデータであると言える．

その後，ハイパフォーマンスコンピューティングの主役が大規模マルチプロセッサから PC クラスタに移るにつれ，結合網の研究は SAN(System Area Network) に移って行った．SAN は，ある程度の不規則性を許容することから，トポロジ自体が問題にならなくなっている．このことから RDT の評価結果は，むしろそのトポロジの興味深さよりも，本論文で評価した，マルチキャスト能力，パケット収集能力，無駄パケット自動廃棄機構等が今後共有益であると考えられる．

これらの RDT ネットワークに関する評価の実施以降，筆者は JUMP-1 のメモリ管理プロセッサである MBP-light について，Core プロセッサの命令セットアーキテクチャの評価を行った．

JUMP-1 のメモリ管理プロセッサ MBP-light は，内部に MBP Core と呼ばれる RISC プロセッサを装備しており，処理の柔軟性を実現している．また，MBP Core は，パケットバッファをレジスタとして扱える Buffer-Register Architecture をとり，ハッシュ値を求める命令やビット操作命令などの特殊命令を装備するなど，処理の高速化を図っている．

MBP Core は，Buffer-Register Architecture をとることにより，Home クラスタで 5.64%，Remote クラスタで 6.27%の性能向上を達成している．特殊命令では，ハッシュ値を求める命令のみが有効に働いており，これにより 2.80%の性能向上を達成している．しかし，このアーキテクチャは設計者の意図通りに性能に貢献していないことが明らかになった．これはパイプラインに組み込まれたバッファの規模が中途半端で，パケットの本格的なバッファとして使うことができなかった点が大きい．パケットを格納可能なメモリとして現在のバッファ同様の広いビット幅を扱えるようにし，キャッシュの形でパイプラインに組み込む方式を採用すべきであったと言える．

また，分散共有メモリの制御を専用プロセッサにより行うか，ハードウェアで行うかは，当時から議論が分かれていた．結論として，ソフトウェアの介在がもたらすアクセスレイテンシィの増加が無視できず，CC-NUMA 型マルチプロセッサではハードウェア制御が主流となった．しかし一方，PC Cluster に用いられるネットワークインタフェースでは，

ハードウェアによる強力なDMA機能を専用プロセッサのソフトウェアが制御する形式が一般的となった．超並列計算機RWC-1の後を受けて，RWCPおよび慶應義塾大学で開発されたRHiNET[73]のネットワークインタフェースであるMartini[74]は，MBP-lightの反省からデータ転送の主たる操作をアドレス変換からプロテクション管理に至るまで完全にハードウェアで実現する一方，内蔵プロセッサはハードウェアを「乗っ取る」形で例外処理を実行するアーキテクチャ[75]を確立して効果を挙げている．この点でMBP-lightのソフトウェアによるプロトコル制御は，CC-NUMAの制御方法としては適切でなかったが，PC Cluster用のネットワークインタフェースの設計に対して知見を与えたと言える．

11.2 最後に

JUMP-1の構想以来，今日までの間に高性能計算機システムをめぐる状況は大幅に変化し，現在では開発の主体はPC/WSクラスタに移っている．CC-NUMA型並列計算機は商用化が行われ，一部で利用されてきたものの，大きなマーケットを占めるまでには至らなかった．また，JUMP-1の実装も困難を伴い，得られた評価結果はある程度限定されたものではあったが，JUMP-1に実装された他に類を見ない独創的なアイデアは今日でも応用可能なものが多いと考えている．今後の並列計算システムの発展のために，本論文が貢献できれば幸いである．

謝辞

本研究の機会を与えて下さり，絶えず御指導，御助言頂いた天野 英晴教授に深く感謝致します．

共に研究にあたり，御指導頂いた

東京大学　平木 敬 教授
豊橋技術科学大学　中島 浩 教授
京都大学　五島 正裕 助手
他の重点領域 group の方々

に感謝します．

本研究を進めるにあたって多くの御助言，御協力を頂いた京都大学の秤谷 雅史，小西将人，額田 匡則各氏に感謝致します．

忙しい中査読の労を執って頂いた原田 賢一教授，野寺 隆助教授，山崎 信行専任講師に感謝の念を表します．

日頃よりたくさんのアドバイスを頂いた天野研究室の皆様に心より感謝致します．

最後に，大学院生活と研究活動を支えてくれた家族に感謝します．

2004 年 春

参考文献

[1] ダニエルヒルズ著, 喜連川優訳. コネクションマシン. パーソナルメディア, 1990.

[2] 日本シンキングマシンズ (株). コネクションマシン CM-5. 1991.

[3] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, Vol. 25, No. 3, pp. 63–79, March 1992.

[4] David Chaiken and Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 314–324, April 1994.

[5] 石畑宏明, 稲野聡, 堀江健志, 清水俊幸, 池坂守夫. 高並列計算機 AP-1000 のアーキテクチャ. 電子情報通信学会論文誌, Vol. J75-D-I-8, No. 8, pp. 637–645, 1992.

[6] 丸山勉, 加納健, 広瀬哲也, 中田登志之, 村松一弘, 浅野由裕, 稲村雄. 並列コンピュータ Cenju-3 のアーキテクチャとその評価. 電子情報通信学会論文誌, Vol. J78-D-I, No. 2, pp. 73–81, February 1995.

[7] Shuichi Sakai, Hiroshi Matsuoka, Kazuaki Okamoto, Takashi Yokota, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. RWC-1 Massively Parallel Architecture. In *High Performance Computing Conference '94*, pp. 33–38, 1994.

[8] 中澤喜三郎, 中村宏, 朴泰祐. 超並列計算機 CP-PACS のアーキテクチャ. 情報処理, Vol. 37, No. 1, pp. 18–28, January 1996.

[9] 田邊昇. 超並列テラフロップスマシン TS/1 における並列処理プロセッサ間チェイニングとその応用. 情報処理学会論文誌, Vol. 36, No. 3, 1995.

[10] 田中英彦. 超並列原理に基づく情報処理基本体系の概要. 情報処理, Vol. 36, No. 6, pp. 501–505, June 1995.

[11] 地球シミュレータホームページ. *http://www.es.jamstec.go.jp/esc/jp/*.

[12] Yule L. Yang, Hideharu Amano, Hidetomo Shibamura, and Toshinori Sueyoshi. Recursive Diagonal Torus: An interconnection network for massively parallel computers. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pp. 591–594, December 1993.

[13] Yule L. Yang and Hideharu Amano. Message Transfer Algorithms on the Recursive Diagonal Torus. In *Proceeding of the 1994 International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 310–317, December 1994.

[14] Yule Yang and Hideharu Amano. Message Transfer Algorithms on the Recursive Diagonal Tours. *IEICE transaction on Information and Systems*, Vol. E79-D, No. 2, pp. 107–116, February 1996.

[15] Yoshiko Tamaki, Naonobu Sukegawa, Masanao Ito, Yoshikazu Tanaka, Masakazu Fukagawa, Tsutomu Sumimoto, and Nobuhiro Ioki. Node Architecture and Performance Evaluation of the Hitachi Super Technical Server SR8000. *Proc. of 12th International Conference on Parallel and Distributed Computing Systems*, pp. 487–493, August 1998.

[16] Tom Lovett and Russel Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 308–317, May 1996.

[17] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine Multiprocessor. Technical report, The University of Toronto, April 1995.

[18] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 302–313, April 1994.

[19] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[20] James Laudon and Daniel Lenoski. The SGI Origin 2000: A CC-NUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 241–251, June 1997.

[21] John R. Mashey. NUMAflex Modular Design Approach. *News*, 2000.

[22] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, May 1990.

[23] John L Hennessy and David A Patterson, editors. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., Second edition, 1996. ISBN 1-55860-329-8.

[24] Hieharu Tanaka, Yoichi Muraoka, Makoto Amamiya, Nobuo Saito, Shinji Tomita, and Hidehiko Tanaka, editors. *The Massively Parallel Processing System JUMP-1*. Ohmsha, 1996. ISBN4-274-90083-5.

[25] Kei Hiraki, Hideharu Amano, Morihiro Kuga, Toshinori Sueyoshi, Tomohiro Kudoh, Hiroshi Nakashima, Hironori Nakajo, Hideo Matsuda, Takashi Matsumoto, and Shin ichiro Mori. Overview of the JUMP-1, an MPP Prototype for General-Purpose Parallel Computations. In *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 427–434, December 1994.

[26] Takashi Matsumoto, Hideharu Amano, Tomohiro Kudoh, Katsunobu Nishimura, Kei Hiraki, and Hidehiko Tanaka. Distributed Shared Memory Architecture for JUMP-1: A General-Purpose MPP Prototype. In *Proceedings of the IEEE 1996 International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 131–137, June 1996.

[27] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 302–313, May 1993.

[28] Hiroaki Nishi, Katsunobu Nishimura, Kenichiro Anjo, Hideharu Amano, and Tomohiro Kudoh. The JUMP-1 Router Chip: A Versatile Router for Supporting a Distributed Shared Memory. In *Proceedings of the 15th International Phoenix Conference on Computers and Communications*, pp. 158–164, March 1996.

[29] Hiroaki Nishi, Ken ichiro Anjo, Tomoiro Kudoh, and Hideharu Amano. The RDT Router Chip: A Versatile Router for Supporting a Distributed Shared Memory. *IEICE transaction on Information and Systems*, Vol. E80-D, No. 9, pp. 854–862, September 1997.

[30] 中條拓伯, 中條拓伯, 中野智行, 松本尚, 小畑正貴, 松田秀雄, 平木敬, 金田悠紀夫. 分散共有メモリ型超並列計算機 JUMP-1 におけるスケーラブル I/O サブシステムの構成. 情報処理学会論文誌, Vol. 37, No. 7, pp. 1429–1439, July 1996.

[31] 小畑正貴, 中條拓伯. 超並列計算機 JUMP-1 におけるハイビジョン画像表示システム. 情報処理学会研究報告, 計算機アーキテクチャ研究会, pp. 17–23, October 1994.

[32] 五島正裕, 岡田智明, 細見岳生, 森眞一郎, 中島 浩眞治. 細粒度プロセッサ間通信をサポートする高機能キャッシュシステム. 情報処理学会研究報告, 計算機アーキテクチャ研究会, pp. 121–128, August 1993.

[33] Masahiro Goshima, Shin ichiro Mori, Hiroshi Nakashima, and Shinji Tomita. The Intelligent Cache Controller of a Massively Parallel Processor JUMP-1. In *Proceedings of Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 116–124, October 1997.

[34] JUMP-1 設計グループ. JUMP-1 共通バスパケット送受仕様説明書, 第 1.0 版, September 1997.

[35] JUMP-1 設計グループ. JUMP-1 共通バス仕様書, 第 4.1 版, June 1996.

[36] 松本尚, 平木敬. Memory-Based Processor による分散共有メモリ. 並列処理シンポジウム, pp. 245–252, May 1993.

[37] 安生健一朗. 超並列計算機 JUMP-1 における分散共有メモリ管理プロセッサに関する研究. 修士論文, 慶應義塾大学大学院, 1997.

[38] SPARC International Inc. *The SPARC Architecture Manual Version 8*. Prentice-Hall Inc., January 1991.

[39] 福島直人. 遷移的矛盾を許容するメモリ・モデルに基づく一貫性制御方式. 修士論文, 京都大学大学院, 1996.

[40] JUMP-1 設計グループ. JUMP-1 メモリ・コマンド仕様書, 第 2.1 版, September 1997.

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

[42] 松本尚. Elastic Barrier: 一般化されたバリア型同期機構. 情報処理学会論文誌, Vol. 32, No. 7, pp. 886–896, July 1991.

[43] 安生健一朗, 井上浩明, 佐藤充, 工藤知宏, 天野英晴, 平木敬. 超並列計算機 JUMP-1 における分散共有メモリ管理プロセッサ MBP-light. 情報処理学会論文誌, Vol. 39, No. 6, pp. 1632–1643, June 1998.

[44] Inoue Hiroaki, Ken ichiro Anjo, Jun Tanabe, Katsunobu Nishimura, Mitsuru Satoh, Kei Hiraki, and Hideharu Amano. MBP-light: A Processor for Management of Distributed Shared Memory. In *the 3rd International Conference on ASIC*, pp. 199–202, October 1998.

[45] 松本尚, 平木敬. 超並列計算機上の共有メモリアーキテクチャ. 電子通信学会技術報告, コンピュータシステム研究会, pp. 47–55, August 1992.

[46] Kai Li. A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. II, pp. 94–101, August 1988.

[47] 佐藤充, 三吉貴史, 松本尚, 平木敬, 田中英彦. シミュレーションを用いた疑似フルマップの定量的評価. 情報処理学会研究報告, 計算機アーキテクチャ研究会, pp. 201–224, July 1994.

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

[49] Masahito Konishi, Masahiro Goshima, Shin ichiro Mori, and Shinji Tomita. Implementation of Distributed Shared Memory Management of the JUMP-1 Multiprocessor. *IPSJ transaction*, Vol. 42, No. 4, pp. 674–682, 2001.

[50] 工藤知宏, 好村公一, 福嶋泰仁, 西村克信, 楊愚魯, 天野英晴. 超並列計算機 JUMP-1 のクラスタ間結合網 RDT における階層マルチキャストによるメモリコヒーレンシ維持手法. 並列処理シンポジウム, pp. 257–264, May 1995.

[51] Tomohiro Kudoh, Hideharu Amano, Takashi Matsumoto, Kei Hiraki, Yulu Yang, Katsunobu Nishimura, Koichi Yoshimura, and Yasuhito Fukushima. Hierarchical Bit-map Directory Schemes on the RDT Interconnection Network for a Massively Parallel Processor JUMP-1. In *Proceedings of the International Conference on Parallel Processing*, Vol. I, pp. 186–193, August 1995.

[52] 西村克信, 工藤知宏, 西宏章, 楊愚魯, 天野英晴. 相互結合網 RDT 上での階層マルチキャストによるメモリコヒーレンシ維持手法. 情報処理学会論文誌, Vol. 37, No. 7, pp. 1367–1377, July 1996.

[53] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 280–289, 1988.

[54] David V. James, Anthony T. Laundrie, Tein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, Vol. 23, No. 6, pp. 74–77, 1990.

[55] Manu Thapar and Bruce Delagi. Distributed-Directory Scheme: Stanford Distributed-Directory Protocol. *IEEE Computer*, Vol. 23, No. 6, pp. 78–80, June 1990.

[56] David H. D. Warren and Seif Haridi. Data Diffusion Machine - A Scalable Shared Virtual Memory Multiprocessor. In *Proceedings of the International Conference Generation Computer Systems*, pp. 943–952, 1988.

[57] 天野英晴, 楊愚魯. 超並列計算機向き結合網 RDT 上でのデッドロックフリールーティング. 電子通信学会技術報告, コンピュータシステム研究会, pp. 9–16, September 1994.

[58] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, Vol. C-34, No. 10, pp. 892–901, October 1985.

[59] 工藤知宏, 福嶋泰仁, 好村公一, 天野英晴, 西村克信. 超並列計算機用結合網 Ring Tree on Mesh の提案. 電子通信学会技術報告, コンピュータシステム研究会, pp. 1–6, October 1994.

[60] 天野英晴. 並列コンピュータ. 昭晃堂, 1996. ISBN 4-7856-2045-5.

[61] 富田眞治. 相互結合網シミュレータとメンテナンスシステム. 平成 7 年度科学研究費補助金 試験研究 (A)(1) 研究成果 報告書, p. 71, March 1994.

[62] Hironori Nakajo, Satoshi Ohtani, Takashi Matsumoto, Masadi Kohta, Kei Hiraki, and Yukio Kanbeda. An I/O Network Architecture of the Distributed Shared-Memory Massively Parallel Computer JUMP-1. In *Procedings of 11th International Conference On Supercomputing (ICS97)*, pp. 253–260, 1997.

[63] 津田健, 山本孝伸, 田中利彦, 五島正裕, 森眞一郎, 富田眞治. メモリ・アクセスの局所性を最適化するループ再構成手法. 情報処理学会研究報告 99-ARC-132,99-OS-80,99-HPC-75, 計算機アーキテクチャ研究会, pp. 133–138, 1999.

[64] 松本尚, 駒嵐丈人, 渦原尚三, 平木敬. 汎用並列オペレーティングシステム: Sss-core —ワークステーションクラスタにおける実現—. 情報処理学会研究報告 96-OS-73 (SWoPP'96) 79, pp. 115–120, August 1996.

[65] 田邊昇, 濱田芳博, 山本淳二, 今城英樹, 中條拓伯, 工藤知宏, 天野英晴. Dimm スロット搭載型ネットワークインタフェース dimmnet-1 とその低遅延通信機構 aotf. 情報処理学会論文誌, Vol. 44, No. SIG01, 2003.

[66] 寺沢卓也, 天野英晴, 工藤知宏. 計算機の記憶システム-iv. マルチプロセッサの記憶システム (1). 情報処理, Vol. 34, No. 1, pp. 96–105, 1993.

[67] 寺沢卓也, 天野英晴, 工藤知宏. 計算機の記憶システム-iv. マルチプロセッサの記憶システム (2). 情報処理, Vol. 34, No. 2, pp. 96–105, 1993.

[68] 濱田 芳博, 山本淳二, 今城英樹, 中條拓伯, 工藤知宏, 天野英晴. A quantitative analysis of the performance and scalability of distributed shared memry cache coherence protocols. *IEEE Transactions on computers*, Vol. 48, No. 2, February 1999.

[69] Toshihiro Hanawa, Hideki Yasukawa, Katsunobu Nishimura, and Hideharu Amano. MINC: Multistage Interconnection Network with Cache control mechanism. In *PDCS'96*, pp. 310–317, February 1996.

[70] 緑川隆, 田辺靖貴, 天野英晴. ディレクトリキャッシュスイッチを持つキャッシュ制御用田段結合網の検討. 電子情報通信学会技術研究報告 (SWoPP2003) CPSY 2003-14, pp. PP.49–54, August 2003.

[71] Myricom 社ホームページ. *http://www.myri.com/*.

[72] A. Hoisie S. Coll F. Petrini, W. Feng and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. In *IEEE Micro*, Vol. 22(1), pp. 46–57, February 2002.

[73] Tomohiro Kudoh, Shinji Nishimura, Junji Yamamoto, Hiroaki Nishi, Osamu Tatebe, and Hideharu Amano. RHiNET: A network for high performance parallel processing using locally distributed computers. In *IWIA'99*, pp. 69–73, November 1999.

[74] Konosuke Watanabe, Junji Yamamoto, Junichiro Tsuchiya, Noboru Tanabe, Hiroaki Nishi, Tomohiro Kudoh, and Hideharu Amano. Preliminary Evaluation of Martini: a Novel Network Interface Controller Chip for Cluster-based Parallel Processing. In *AI 2002*, pp. 390–395, February 2002.

[75] Konosuke Watanabe, Hideharu Amano, Junji Yamamoto, Junichiro Tsuchiya, and Tomohiro Kudoh. Taking over mechanism: a cooperation methodology of Hardware and Software in Network Controllers. In *Proc. of International Workshop on Synthesis and System Integration of Mixed Information Technologies(SASIMI2003)*, pp. 386–393, April 2003.

[76] Arvind and R. A. Iannuchi. A Critique of Multiprocessing von Neumann Style. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 426–436, June 1983.

[77] 五島正裕, 森眞一郎, 中島浩, 富田眞治. Virtual Queue: 超並列計算機向きメッセージ通信機構. 情報処理学会論文誌, Vol. 37, No. 7, pp. 1399–1408, July 1996.

[78] Texas Instruments Inc. *SuperSPARC User's Guide*, October 1992.

[79] 美辺央希. 汎用並列計算機モニタシステム Pot. 修士論文, 慶應義塾大学大学院, 1998.

[80] Yuso Kanamori, Oki Minabe, Masaki Wakabayashi, and Hideharu Amano. Pot: A General Purpose Monitor for Parallel Computers. *IEICE Transactions on Information and Systems*, Vol. E86-D, No. 10, pp. 2025–2033, October 2003.

付 録A　RDT network用のpacket format

A.1　multicast packet

multicast packet の形式を図 A.1 に示す．

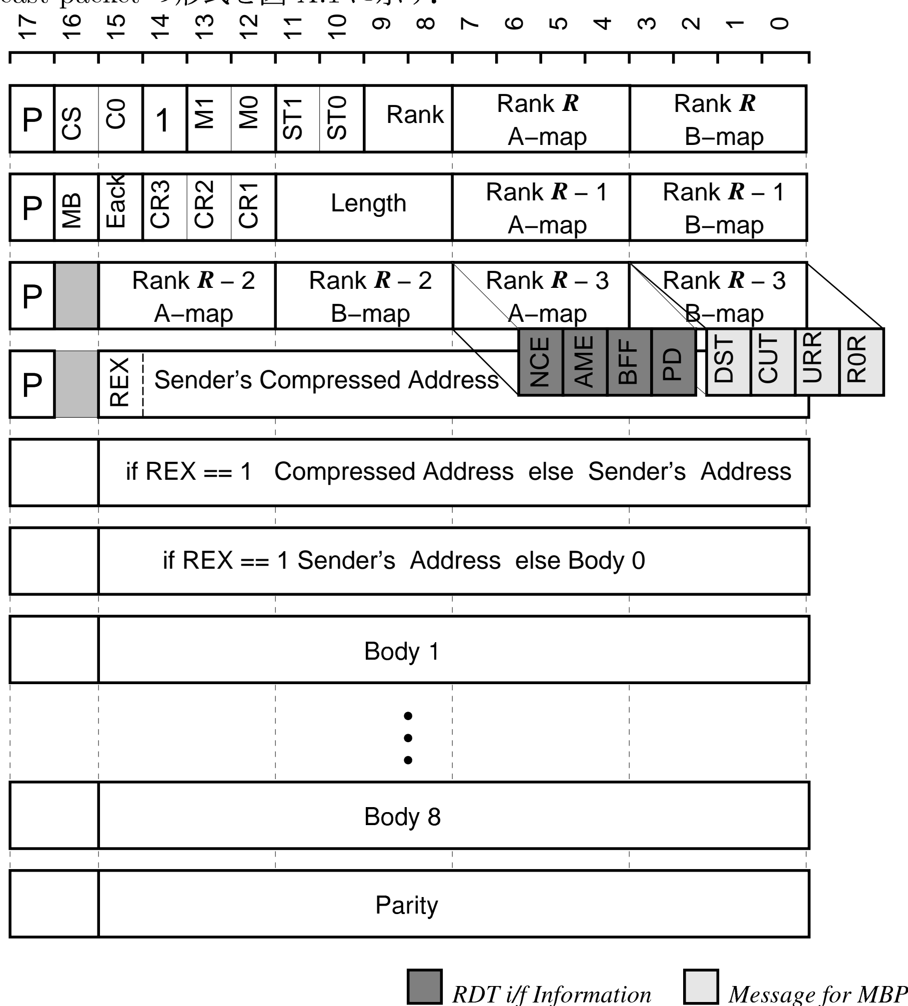


図 A.1 multicast packet の形式

P	横パリティ(偶数)
CS	special channel 識別子．MBP-light からは 0 にして送信する必要がある．

C0	channel 識別子．0 または 1 を選択できる．
M1，M0	RHBD 方式を次のように指定する． • 00: SM 法 • 01: LPRA 法 • 10: LARP 法
ST1，ST0	router 間でのパケット転送状態で，MBP-light から送信の際には 00 を指定する必要がある．
Rank	マルチキャストを開始する rank(0 ∼ 3) を示す．
Eack	router chip で応答パケットの収集を行うかどうかを示す． • 0: router で収集しない． • 1: router で収集する．
MB	宛先の MBP0，MBP1 を選択する．常に 0 でよい．
CR1，CR2，CR3	cut rank を指定することができる．cut rank では，マルチキャストの中継ノードであった場合に MBP-light を不必要だが一度経由する．CR1 や CR2，CR3 はそれぞれ rank 1 や rank 2，rank 3 での cut rank に対応したビットとなっている．
Length	Length は MBP Core からみたパケット長である．パケット長は最短 6，最長 15 である．実際に RDT router に送る際には縦パリティのフリットが付加されるため，RDT router にとっては Length はパケット長をデクリメントしたものとみなされる．
Rank R，R-1，R-2，R-3，A/B-map	Rank R はマルチキャストを開始する rank でのビットマップである．1 つ下の rank が Rank R-1，その下が Rank R-2... と降順に並べる．0 より小さい rank の map は使用されない．Rank が 0 の場合には local なマルチキャストとなり，その場合には Rank R-1 の map に従ってマルチキャストされる． • A-map: 周囲 4 方の pattern (N:E:W:S) • B-map: South 方向での pattern (Self:SE:SW:SS)

Compressed Address　Eack が 1 の場合に必要になる．マルチキャストに対する acknowledge packet の収集を行う際に鍵となって収集を行う．Compressed Address は 16 bit で構成され，最上位ビットは REX(R-extension)bit である．マルチキャスト開始 rank が 2 以上であり，acknowledge packet の収集が必要なパケットである場合には，4 と 5 フリット目両方に Compressed Address をのせて送信する必要がある．

Message for MBP　multicast packet が MBP-light に到着する際には，必ずなんらかの目的がある．その目的を示すフィールドとなる．

- DST: マルチキャストの宛先であったことを示す．
- CUT: cut rank であったことを示す．
- URR: 上位 rank の acknowledge packet の収集を行う必要がある．
- R0R: rank0 の acknowledge packet の収集を行う必要がある．

RDT Interface Information　multicast packet には，必ず 4 bit の RDT Interface におけるハードウェア情報が付加されて MBP Core に到着する．

- NCE

  Net Cache にヒットしたことを示す．

- AME

  Ackmap Cache にヒットしたことを示す．

- BFF

  Net Cache と Ackmap Cache の両方にヒットして acknowledge packet を自動生成しようとするが，そのバッファが満杯であったためにパケットが生成できなかったことを示す．

- PD

  Net Cache の tag にある Packet Discard に相当する．

Parity　パケットのヘッダ以外のフリットに対する縦パリティである．RDT Interface で付加されるため，MBP Core からは見ることができない．

Address　Address の構成については，図 D.1 と図 5.16 を参照．

Body　ハードウェアで定義されていないフィールドであり，任意に使用可能．

A.2 acknowledge packet

acknowledge packet の形式を図 A.2 に示す．

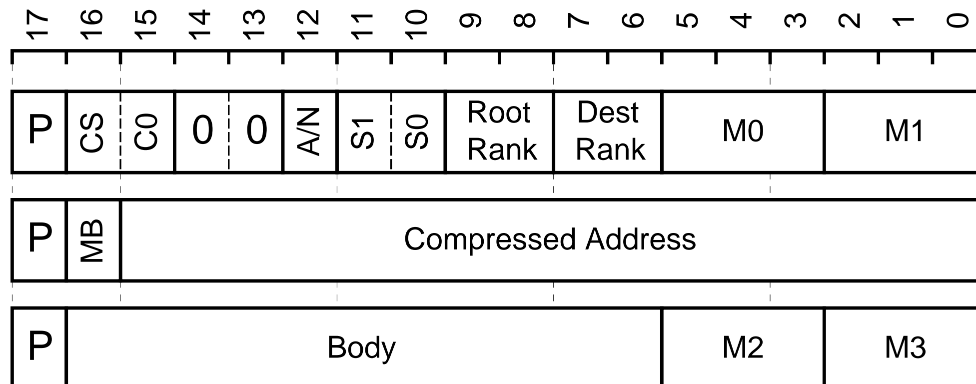


図 A.2 acknowledge packet の形式

CS　special channel 識別子．MBP-light からは 0 にして送信する必要がある．

C0　channel 識別子．0 または 1 を選択できる．

A/N　acknowledge/not-acknowledge を示す識別子．1 ならば acknowledge packet であることを示す．

S1, S0　acknowledge packet に対し router 間でパケット制御に用いる状態で，MBP-light から送信する際には 00 で送信する必要がある．

- 00: rank 0 で収集が必要な acknowledge packet であることを示す．
- 01: 上位 rank で収集が必要な acknowledge packet であることを示す．
- 10: RDT router から MBP-light に対して送信するパケットにしか使用されず，RDT router で rank 0 の収集が終わっているパケットということを示す．
- 11: root で収集済みのパケットで Root Rank で示す rank の router から Dest Rank で示される rank にルーティングされることを示す．

Root Rank　マルチキャスト開始 rank(0 ～ 3) を示す．

Dest Rank	multicast packet は，送信されたノードからマルチキャスト開始 rank の木構造の根までルーティングされ，そこからマルチキャストを開始する．Dest Rank はこの送信ノードの rank を表す．
Compressed Address	acknowledge packet は到着した multicast packet の Compressed Address をここに載せて，送信する必要がある．
MB	宛先の MBP0，MBP1 を選択する．常に 0 でよい．
M0, M1, M2, M3	acknowledge packet は木を遡るため，到着した multicast packet が 8 進木のどの枝にあたるパケットであったかを示す必要がある．M0, M1, M2, M3 はそれぞれ Rank 0∼Rank 3 のどの枝かを示す． • 000: Self(この rank では routing の必要がない) • 001: South East • 010: South West • 011: South South • 100: North • 101: East • 110: West • 111: South
Body	ハードウェアで定義されていないフィールドであり，任意に使用可能．

A.3 shootdown/setup packet

shootdown/setup packet の形式を図 A.3 に示す.

MBP-light からの shootdown 要求によって，RDT router から MBP-light に対して送信されるパケットである．shootdown が要求されると，RDT router 内のクロスバの入力バッファにパケットが存在する場合，この shootdown packet によって MBP-light にその内容が報告される．また，acknowledge packet が shootdown される場合，第 2 フリットは省略される.

また，shootdown 後に RDT router のバッファの内容を回復するために，setup を行うことができる．この場合は shootdown による shootdown packet を完全に保存しておき，setup の際に全て送信する必要がある.

PE	router 内でパリティ誤りがあった場合に 1 になる.
SC, OC	どの channel にたまったパケットであるかを示す.
Bitmap	どの出力に対してまだ送信していないかを示す．上位ビットから ACK 上位/下位，上位 rank の方位 (NEWS)，rank 0 の方位 (NEWS)，MBP0/1 となっている.
Link	どの link からの情報であるかを示す.

- 0000: MBP0
- 0001: MBP1
- 0010: South(rank-0)
- 0011: West(rank-0)
- 0100: East(rank-0)
- 0101: North(rank-0)
- 0110: South(upper rank)
- 0111: West(upper rank)
- 1000: East(upper rank)
- 1001: North(upper rank)
- 1010: Ack. Collector

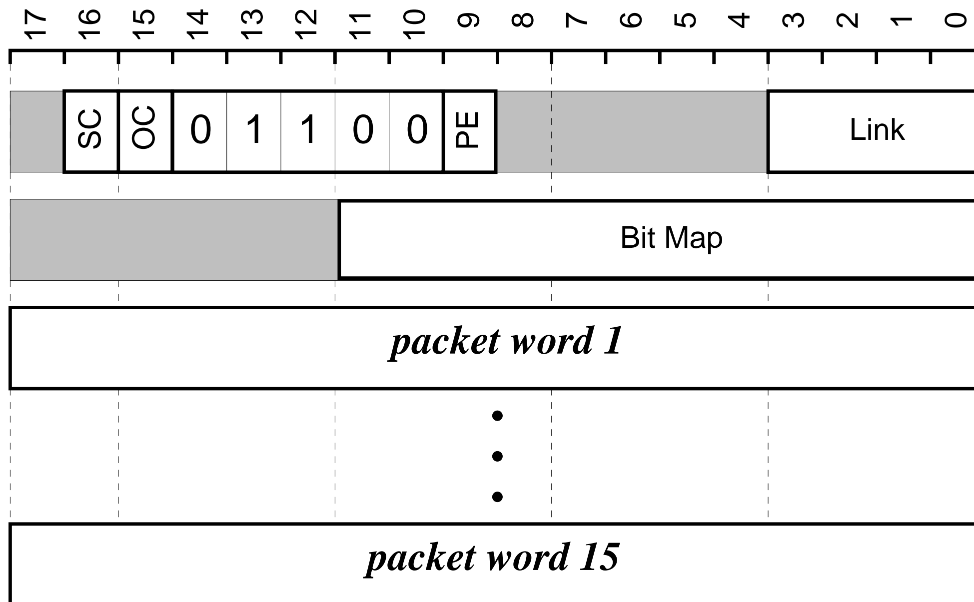


図 A.3 shootdown/setup Packet の形式

A.4　Ack Cache shootdown packet

Ack Cache shootdown packet の形式を図 A.4 に示す.

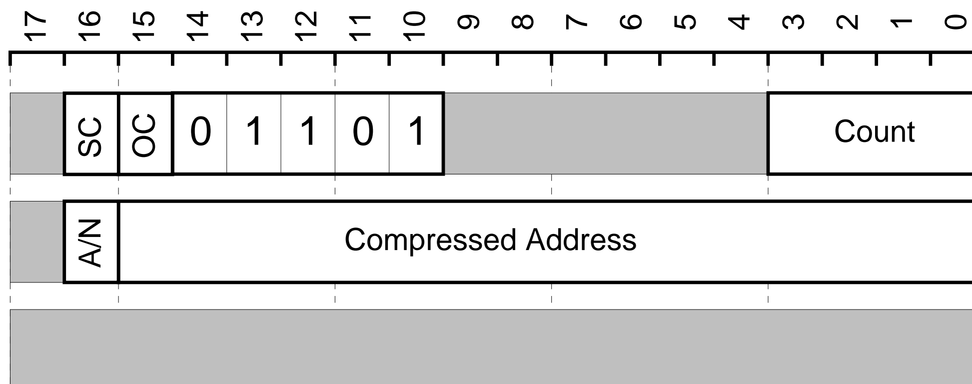


図 A.4 Ack Cache shootdown packet の形式

MBP-light からの shootdown 要求時に，shootdown packet に先だって Ack Cache からこの AckCache shootdown Packet が MBP-light に送信される．ただし，Ack Cache は回復できないため，このパケットの内容は MBP-light で保持しておく必要がある．

SC, OC　　どの channel からのパケットであるかを示す.

Count　　Ack Cache 中のまだ収集していないパケット数を示す.

Compressed Address　　Ack Cache 中の鍵となる Compressed Address を示す.

A/N　　acknowledge/not-acknowledge packet かを示す．Ack Cache では収集するパケットのどれか 1 つが acknowledge であれば，N/A は 1(acknowledge) となる．全てが not-acknowledge であった場合に，収集したパケットも not-acknowledge となる．つまり，Ack Cache の tag と同じ意味をもつ.

A.5　timeout packet

timeout packet の形式を図 A.5 に示す.

timeout packet は，timeout 発生時に RDT router から MBP-light に送られるパケットである．クロスバの入力バッファからパケットが送られる．PE が無いこととパケット識別子 (01110) が異なることを除けば，shootdown packet と同じ構成である.

A.6　command packet

command packet の形式を図 A.6 に示す.

command packet は MBP-light が RDT router に送信するもので，rank の設定や shoot-down / setup 要求などはこのパケットで行う.

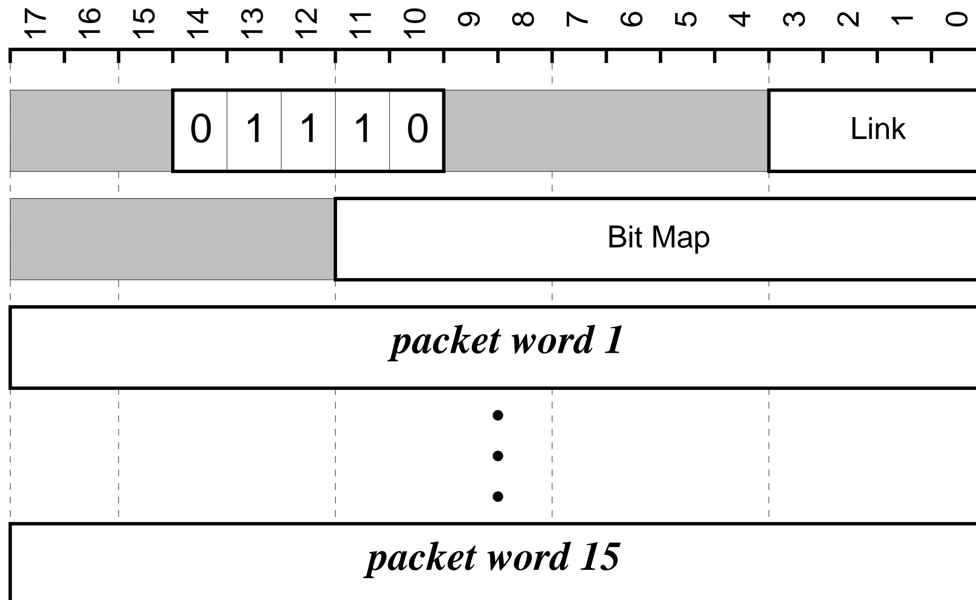


図 A.5 timeout packet の形式

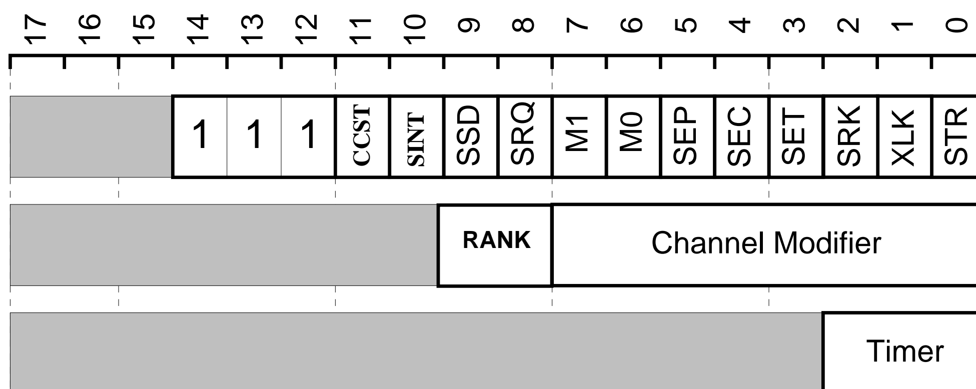


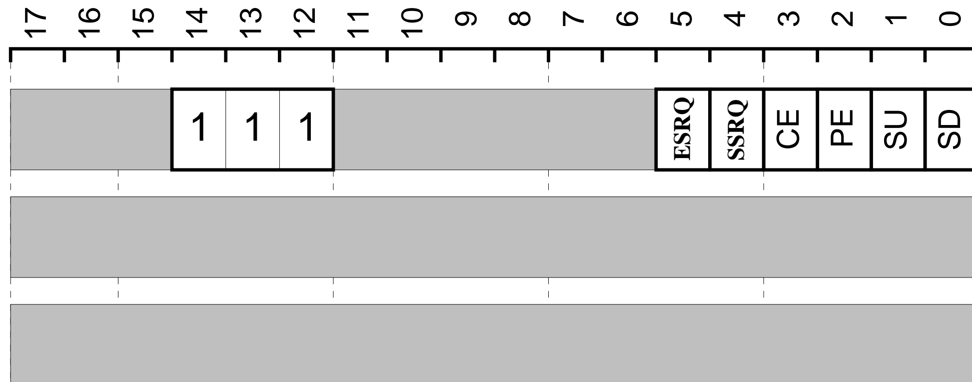
図 A.6 command packet の形式

CCST	同一 node の 2 つの router 間の動作が一致しているかどうかを検査し，error 発生時には割り込みを起こす設定を行うビット．
SINT	node 間同期要求発生時に，割り込みを起こすように設定するビット．
SSD	node 間同期要求発生時に，shootdown を引き起こすように設定するビット．
SRQ	node 間同期要求を発生する命令ビット．
M1, M0	RDT router に対する命令を示す． • 00: normal mode • 01: shootdown mode • 10: setup mode • 11: reset
SEP	parity 検査を行い，誤りが発生すると shootdown を起こすように設定するビット．
SEC	Channel Modifier に示される各出力リンクに対して channel0 と 1 の入れ替えを行うビット．このビットを設定すると，メッシュの round trip loop で使用する channel を入れ替えることが可能となり，e-cube routing をサポートすることができる．上位ビットから上位 rank の方位 (NEWS)，rank 0 の方位 (NEWS) を示す．
SET	Timer の値に従って timeout timer の設定ができるビット．
SRK	Rank を上位 rank の値として設定するビット．
STR	status report packet を送信するビット．
Channel Modifier	SEC を参照．
Timer	SET が 1 の時に timeout timer を設定する． • 000: timeout なし • 001: $2^8 \sim 2^9$clock で timeout • 010: $2^{12} \sim 2^{13}$clock で timeout • 011: $2^{16} \sim 2^{17}$clock で timeout • 100: $2^{20} \sim 2^{21}$clock で timeout • 101: $2^{24} \sim 2^{25}$clock で timeout • 110: $2^{28} \sim 2^{29}$clock で timeout • 111: $2^3 \sim 2^4$clock で timeout

XLK	クロスバの優先度を変更するビット．1 で固定優先度，0 で round robin に設定できる．
Rank	SRK を参照.

A.7 status report packet

status report packet の形式を図 A.7 に示す．



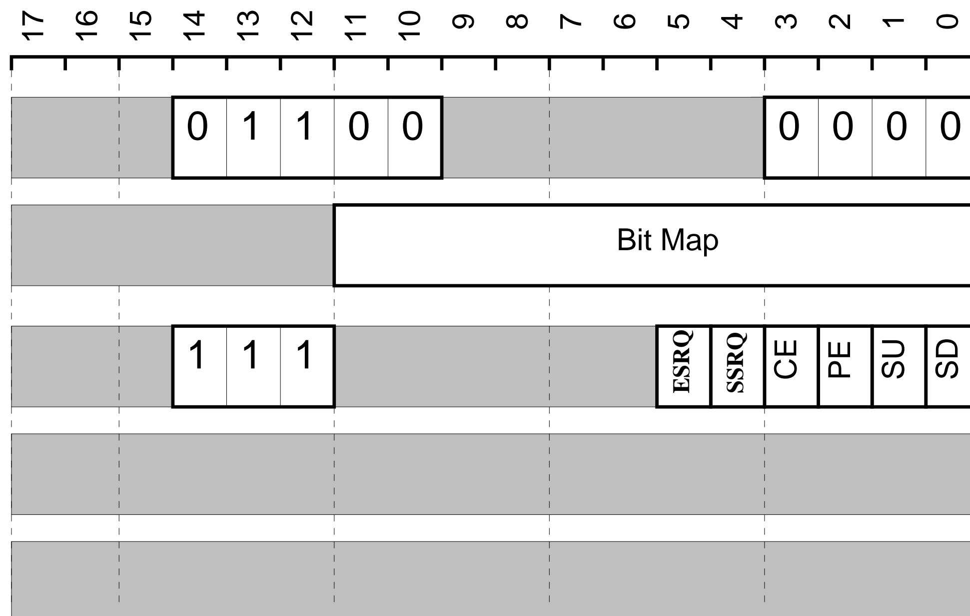


図 A.7 status report packet の形式

status report packet は command packet の STR により，status report packet が要求

された場合に MBP-light に対して送信される．こちらが図中の上のパケットに相当し，通常時のものである．また，shootdown の終了時にも RDT router から送信される．こちらは図中の下のパケットに相当し，特殊時のものである．shootdown 終了時に送信される status report packet は setup 時に RDT router に対して送信してはならない．

ESRQ	外部から同期要求が起きたことを示すビット．
SSRQ	自ら同期要求を発生したことを示すビット．
CE	2 つの router 間で consistency error が発生したことを示すビット．
PE	parity 誤りが発生したことを示すビット．
SU	setup 状態にあることを示すビット．
SD	shootdown 状態にあることを示すビット．
Bitmap	どの出力に対してまだ送信していないかを示す．上位ビットから ACK 上位/下位，上位 rank の方位 (NEWS)，rank 0 の方位 (NEWS)，MBP0/1 となっている．

ただし，以上で 17 bit 目を MBP Core が設定できないことを，そして 16 bit 目は 4 フリット分合わせて PBR の offset 8 に対応することを明記しておく．

付 録B　MMCの内部register

MBP Core からはいくつかの MMC の内部レジスタを参照可能であり，その機能を利用することができる．ここでは，その内部レジスタについて説明する．

B.1　MMC の内部レジスタに対するメモリマップ

MBP Core の内部メモリに対して MMC の内部レジスタがどのようにマッピングされているかを表 B.1 に示す．

( ) 内の数字はインタフェースとしてのビット数を示す．つまり，信号線としてはそのビット分あるが，実際にはそれだけ使われない．ただし，+は複数の信号線を組み合わせたものを意味する．

B.2　MMC の内部レジスタの役割

B.2.1　RECV_ADR_REG(0-7)

address = 0x9000 – 0x9007

offset	31 … 2	1 0
00	address[63:32]	
01	address[31: 0]	
02	—	WWE

クラスタバスから受け取ったパケットの address flit を保持しているレジスタ．入力 address flit の内容を見ることで，MBP Core は MMC がクラスタバスから受け取ったパケットの address flit を検査することができる．なお，MMC 内部では *RECV_ADR_REG0* ～ *RECV_ADR_REG3* は reply packet 用，*RECV_ADR_REG4* ～ *RECV_ADR_REG7* は request packet 用である．

B.2.2　RECV_BUF_STATE

address = 0x9008

31 … 9	8	7 … 4	3 … 0
—	C	Req.	Rep.

入力バッファの状態を表しているレジスタ．‘1’ のときには当該バッファが使用中 (パケットが入っている状態) であることを示し，‘0’ のときには未使用 (パケットが入っていない状態) であることを示す．

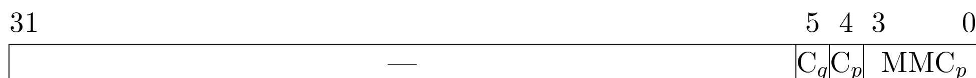
表 B.1 MMC の内部レジスタ

address	name	bit	address	name	bit
0x9000	RECV_ADR_REG0	66	0x9018	MTC_PROC_NUM	4
0x9001	RECV_ADR_REG1	66	0x9019	MTC_DECE_REG	1
0x9002	RECV_ADR_REG2	66	0x901a	MTC_INT+	8
0x9003	RECV_ADR_REG3	66	0x901b	REF_CMP	10
0x9004	RECV_ADR_REG4	66	0x901c	CBC_LOCK_REG_STATE	4
0x9005	RECV_ADR_REG5	66	0x901d	–	–
0x9006	RECV_ADR_REG6	66	0x901e	–	–
0x9007	RECV_ADR_REG7	66	0x901f	–	–
0x9008	RECV_BUF_STATE	9	0x9020	CBC_LOCK_BMP0	64
0x9009	SEND_BUF_STATE	6	0x9021	CBC_LOCK_MASK0	39
0x900a	MTC_REQ_STATE	9	0x9022	CBC_LOCK_ACT0	2
0x900b	CBC_INT_PKT_SRC+	12	0x9023	–	–
0x900c	CBC_PEND_DOWN	4(8)	0x9024	CBC_LOCK_BMP1	64
0x900d	CBC_ADR_CMP_MODE	3	0x9025	CBC_LOCK_MASK1	39
0x900e	CBC_IRR_REG	1	0x9026	CBC_LOCK_ACT1	2
0x900f	–	–	0x9027	–	–
0x9010	CBC_PEND_L_BMP	64	0x9028	CBC_LOCK_BMP2	64
0x9011	CBC_PEND_L_MASK	39	0x9029	CBC_LOCK_MASK2	39
0x9012	CBC_PEND_L_ACT	5	0x902a	CBC_LOCK_ACT2	2
0x9013	–	–	0x902b	–	–
0x9014	CBC_PEND_COUNTER0	16(32)	0x902c	CBC_LOCK_BMP3	64
0x9015	CBC_PEND_COUNTER1	16(32)	0x902d	CBC_LOCK_MASK3	39
0x9016	CBC_PEND_COUNTER2	16(32)	0x902e	CBC_LOCK_ACT3	2
0x9017	CBC_PEND_COUNTER3	16(32)	0x902f	–	–

下位ビットから順に，入力バッファの 0 番，1 番，. . .，8 番に対応している．MMC では，入力バッファの 0~3 番がクラスタバスからの reply packet 用，4~7 番がクラスタバスからの request packet 用，8 番が MBP Core 用となっている．

B.2.3　SEND_BUF_STATE

address = 0x9009



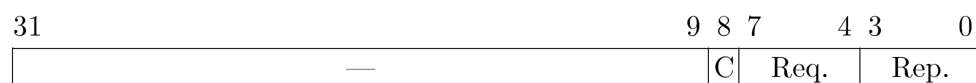
出力バッファの状態を表しているレジスタ．‘1’ のときには当該バッファが使用中 (パケットが入っている状態) であることを示し，‘0’ のときには未使用 (パケットが入っていない状態) であることを示す．

下位ビットから順に, 出力バッファの 0 番，1 番，. . .，5 番に対応している．MMC では，出力バッファの 0~3 番は，MMC がハードウェア的に生成するクラスタバスへ出力する reply packet 用*，4 番は MBP Core がクラスタバスへ出力する reply packet 用，5 番は MBP Core がクラスタバスに出力する request packet 用となっている．

MBP Coreからクラスタバスへパケットを出力するときには，自動的に *SEND_BUF_STATE* が設定される．したがって，一般には MBP Core は *SEND_BUF_STATE* に data を書き込む必要はない．MBP Core からクラスタバスへ出すパケットが実際に出力されたかどうかは，このレジスタを検査することによって知ることができる．

B.2.4　MTC_REQ_STATE

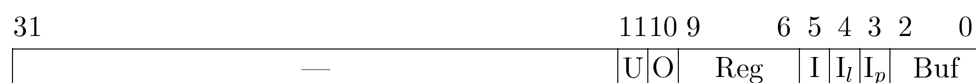
address = 0x900a



入力バッファからのメモリ参照要求の状態を表しているレジスタ．‘1’ のときには当該バッファからメモリ参照要求が出ていることを示し，‘0’ のときには当該バッファにはパケットが入っていないか，メモリ参照要求が出ないパケットが入っていることを示す．下位ビットから順に，入力バッファの 0 番，1 番，. . .，8 番に対応している．MMC では，入力バッファの 0~3 番がクラスタバスからの reply packet 用，4~7 番がクラスタバスからの request packet 用，8 番が MBP Core 用となっている．MTC はこのレジスタから出るメモリ参照要求に応じて処理する．

B.2.5　CBC_INT_PKT_SRC+

address = 0x900b



*MMC が hardware 的に生成するパケットは reply packet のみである．

割り込み packet queue の top 及び PEND lock counter のオーバーフロー，アンダーフローによる割り込みを表す．各ビットの意味は次のとおりである．

b11: ‘1’ ならば，PEND lock counter のアンダーフローによる割り込みを表す．

b10: ‘1’ ならば，PEND lock counter のオーバーフローによる割り込みを表す．

b9-6: lock register の何番のレジスタによる割り込みかを表す．原因となったレジスタに対応するビットが ‘1’ になる．b9 がレジスタの 3 番，b8 が 2 番，b7 が 1 番，b6 が 0 番に対応し，複数のレジスタによる割り込みもありうる．

b5: ‘1’ ならば，割り込みパケットを受信したことによる割り込みを表す．

b4: ‘1’ ならば，lock register による割り込みを表す．

b3: ‘1’ ならば，PEND lock register による割り込みを表す．

b2-0: 割り込みの原因となったパケットが格納されているバッファ番号を表す．

実際には複数の割り込み原因による割り込みもありうる．すなわち，b5-3 が同時に ‘1’ になる可能性もある．

CBC_INT_PKT_SRC の b9-6 は，*CBC_INT_PKT_SRC* の b2-0 で示されるパケットが何番の lock register に引っかかったか，または引っかからなかったかを示すことになる．(*cf.* *CBC_LOCK_REG_STATE*)

B.2.6 CBC_PEND_DOWN

address = 0x900c

31 — 4	3 — 0
—	D

PEND counter をデクリメントするためのレジスタ．このレジスタの対応するビットに ‘1’ を書き込むと，対応する PEND counter を 1 つ減らす (値が ‘0’ である場合はなにもしない)．PEND counter に値を直接書き込む方法と違って，この *CBC_PEND_DOWN* は，MMC による自動カウントダウンと重なっても正しく動作する．ただし，同一の clock に MMC による自動カウントダウンと *CBC_PEND_DOWN* に対する参照が重なると，PEND counter は 2 つ減らされる．

b0 が 0 番の 2 次キャッシュ用，b1 が 1 番の 2 次キャッシュ用，b2 が 2 番の 2 次キャッシュ用，そして b3 が 3 番の 2 次キャッシュ用の PEND counter に対応する．

B.2.7 CBC_ADR_CMP_MODE

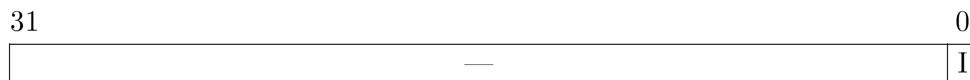
address = 0x900d

31 — 3	2 — 0
—	A

MMC は入力バッファにたまっているパケットと同じアドレスのパケットが到着するとリトライするが，この同じアドレスを認識するときに address field のどこを比較するかを指定するレジスタ． 通常は “001”(block) になっている．指定の方法は lock register の address mask と同じであるので，詳しくはそちらを参照されたい．

B.2.8　CBC_IRR_REG

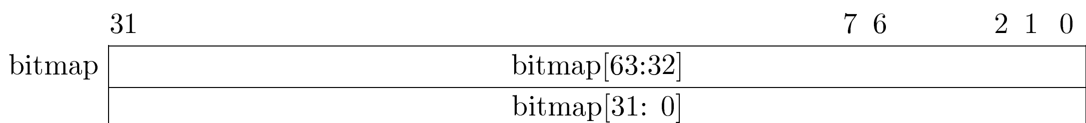
address = 0x900e

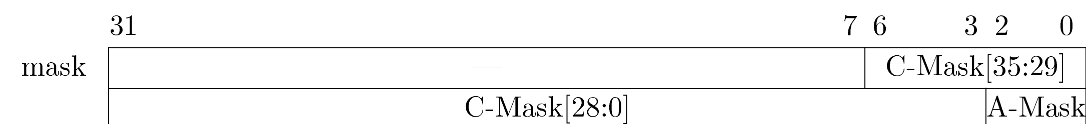


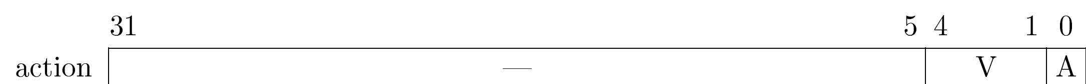
クラスタバスからのパケットの受取りを制御するレジスタ．I=‘0’ のとき，MMC はクラスタバスからのパケットに対して通常の受取り動作を行う．I=‘1’ のとき，MMC は入力バッファに空きがあってもクラスタバスに対して busy を出し，それ以上パケットを受け取らないようにする．

B.2.9　CBC_PEND_L_［BMP, MASK, ACT］

address = 0x9010[BMP], 0x9011[MASK], 0x9012[ACT]







クラスタバスからの入力パケットに対して，リトライまたは割込みを制御するためのレジスタ．PEND lock register は表 B.2 の 3 つの要素から構成される．

表 B.2 PEND lock register の構成

名前	ビット数	説明
bitmap	64	比較のためのパターン．
mask	39	どの部分を比較するかを指定する．
action	5	比較の結果一致した場合どのような動作を行うか指定する．

MMC ではクラスタバスからパケットが到着すると，その address flit(64 ビット) と bitmap を比較する．比較の結果，mask で ‘0’ になっている部分が全て一致していると，action で指定された動作を行う．なお，比較の際には入力パケットの `wwe` field は含めない．

以下に，各要素について説明する．

- bitmap

  bitmap には，比較を行うビット列を設定する．bitmap は b63 – b0 があり，これが入力パケットの address flit の b63 から b0 に対応する．また，bitmap のうち有効な部分を次に述べる mask で指定する．bitmap のうち mask で指定されなかった部分については無視される．

  ただし，PEND lock register では address flit の `sid` field(b43 – b41) に対応する部分 (b43 – b41) については，常に無視される．

- mask

  bitmap のうち，どのビットの比較結果を有効とするかを指定する．この mask を指定することによって，address flit の一部のみを検査し，action を起こすようにすることができる．

  mask は更に以下の 2 つの部分に分けられる．

  – b38 – b3: Control Mask (C-Mask)

    bitmap の b63 – b28 に対応する．値が ‘0’ のビットに関して，一致検出を行う．値が ‘1’ のビットに対応する bitmap は無視する．

    ただし，PEND lock register では address flit の `sid` field (b43 – b41) に対応する部分 (b18 – b16 / C-Mask[15:13]) に関しては，常に ‘1’ であるとして扱われる．

  – b2 – b0: Address Mask (A-Mask)

    address 部 (b27 – b0) に対応する mask．address 部に関しては，制御部のようにすべてのビットに対応した mask を指定するのではなく，表 B.3 に示すようなパターン指定を行い，対応するビットに相当する部分のみ一致検出を行う．

表 B.3 アドレスの比較範囲

b2	b1	b0	指定範囲	対応するビット
0	0	0	すべて	b27 – b0
0	0	1	block (32 Bytes)	b27 – b5
0	1	0	—	—
0	1	1	page (4 KBytes)	b27 – b12
1	0	0	—	—
1	0	1	large page (256 KBytes)	b27 – b18
1	1	0	16 MBytes	b27 – b24
1	1	1	なし	なし

- action

  action では，mask で指定された部分に関して，bitmap と入力パケットの address flit の一致検査を行った結果一致した場合，どのような動作を行うかを指定する．

表 B.4 に示すように，action は 2 つのフィールドから構成される．

表 B.4 PEND lock register での action

ビット	名前	説明
b0	action	retry / 割り込みを指定する．‘0’ のときリトライで，‘1’ のときが割込みである (ただし，reply packet は**必ず**割り込む)．
b4 – b1	valid	この PEND lock register が有効かどうかを示す．‘0’ のときは無効で，‘1’ のときが有効である．b1 が 0 番の 2 次キャッシュ，b2 が 1 番の 2 次キャッシュ，b3 が 2 番の 2 次キャッシュ，b4 が 3 番の 2 次キャッシュに対応している．また，MMC 内部の PEND counter の値がオーバーフローすると，対応する 2 次キャッシュに応じたビットが自動的に ‘1’ に設定される．

PEND counter について

PEND counter は現在メモリ処理中のトランザクション数を保持している．このカウンタがオーバーフローした場合は，カウンタの値の一貫性を保つため，何らかの処理を行う必要がある．MMC では，PEND counter がオーバーフローした場合，オーバーフローしたカウンタに関連づけられている 2 次キャッシュに対応した valid bit を設定することによって対処する．この機能を用いることによって，あらかじめ bitmap および mask に適切な値を設定しておくと，PEND counter がオーバーフローしたときに特定の 2 次キャッシュからの `ack` field が ‘1’ であるパケットをリトライするなどの処理が可能になる．

B.2.10　CBC_PEND_COUNTER(0-3)

address = 0x9014–0x9017

31 — 16	15 — 0
—	PEND counter

SPARC に送る PEND 信号を生成するためのカウンタ． クラスタバスからメモリ参照要求を発生するパケット (例えば，ACK field が ‘1’) が到着すると，このカウンタが 1 つ増やされて，メモリの参照が終了するとこのカウンタが自動的に 1 つ減らされる．PEND counter の詳細に関しては，文献 [48] を参照のこと．

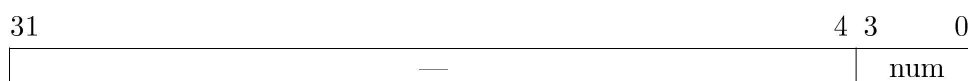
MBP Core が処理しなくてはならないメモリへの参照に対しては，MMC は PEND counter を 1 つ増やすが，減らしはしない．したがって，メモリ参照終了時に，MBP Core は明示的に PEND counter を 1 つ減らす必要がある．PEND counter を 1 つ減らす方法は，PEND counter に直接値を書き込む方法と *CBC_PEND_DOWN* を用いて PEND counter を 1 つ減らす方法の 2 つがある．しかし，PEND counter に値を直接書き込むと，MMC による自動カウントダウンと重なった場合にカウンタの値が実際のメモリ参照の数とずれてしまう可能性がある．したがって，MBP Core から PEND カウンタを減らす場合には，

*MTC_REQ_STATE*によりメモリの参照要求がないことを確認した上で，メモリの参照を処理するか，*CBC_PEND_DOWN*を用いるかどちらかの方法をとらなければならない．

なお，オーバーフローの検出はカウンタの値が 0xFFFB から 0xFFFC に変わるときであり，一方アンダーフローの検出はカウンタの値が 0x0000 から 0xFFFF に変わるときである．

B.2.11 MTC_PROC_NUM

address = 0x9018



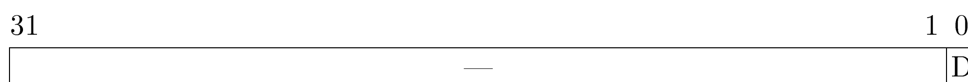
現在 MTC で処理中のパケットの入力バッファの番号を保持しているレジスタ．

MTC で割り込みが発生した場合は，割り込みの状態が保存される．したがって，MTC による割り込み発生時には，処理中バッファ番号に割り込みの元となるパケットの入力バッファの番号が保存されていることになる．

ただし，一番上のビットが ‘1’ となるのは，MBP-light からのパケットが割り込みを発生した場合である．つまり，address flit の bit 47 を ‘1’ にして，メモリアクセスを行ったときに生じる†．

B.2.12 MTC_DECE_REG

address = 0x0x9019

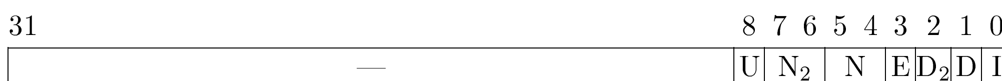


DRAM の data に付随している ECC(Error Correcting Code) による誤り検査を有効 (D = ‘1’)/無効 (D = ‘0’) にするためのレジスタ．

通常，power on では ECC memory の内容は不定なので， 読み出すたびに ECC error が起きる．そのためリセット時には，この値は ‘0’ になっており，メモリの初期化を行った後に ‘1’ にする必要がある．MBP Core から ECC memory を読む場合は，図 C.1 に示した MBP アドレスの CNTL，S/D，D/T field を ‘0’，ECC field を ‘1’ として，control_read を発行すればよい．ただし，書込みは**できない**．

B.2.13 MTC_INT+

address = 0x901a



各ビットの意味は以下の通り (ここでは 1 語を 64 ビットと考える)．

† これは隠し機能の 1 つであるが，この機能は**使用してはならない**．

b8: ‘1’ ならば MMC が予期せぬ address flit を受けとったことを示す.

b7-6: ECC 誤り検査状態で，更に ECC error の発生した語番号を示す.
(*DCTL_ERROR2_NUM*)

b5-4: ECC error の発生した語番号を示す.
(*DCTL_ERROR_NUM*)

b3: ‘1’ ならば CC からの Emergency Trap がかかっていることを表す.

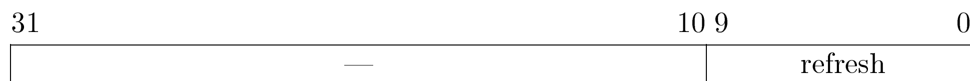
b2: ‘1’ ならば ECC 誤り検査状態で，更に ECC error が起こったことを表す.
(*DCTL_ERROR2*)

b1: ‘1’ ならば同期 DRAM で ECC error が起こったことを表す.
(*DCTL_ERROR*)

b0: ‘1’ ならば SRAM の tag による割り込みか，同期 DRAM の ECC error による割り込みが発生していることを示す．SRAM の tag による割り込みの場合，*MTC_INTERRUPT* のみが ‘1’ になり，ECC error の場合は *MTC_INTERRUPT* 及び *DCTL_ERROR* が同時に ‘1’ になる.

B.2.14　REF_CMP

address = 0x901b

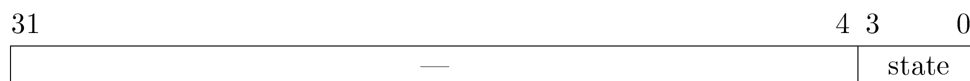


同期 DRAM のリフレッシュ間隔を設定するレジスタ．refresh counter がここで設定した値に達すると，同期 DRAM controller に対してリフレッシュ要求がでる．

μPD4516821G5-A10-7JF-A は，32ms 以内に 2048 row をリフレッシュしなくてはならない．したがって，リフレッシュの間隔は 15.6 μs になる．MMC の動作周波数は 50 MHz であるから，15.6 μs は 781 clock に相当する．50 MHz 動作の場合では少なくとも 771 以下でなくてはならない．この値は，同期 DRAM controller がリフレッシュ要求を受けたときに動作中である可能性を考えて，少なくとも 10 clock の余裕をみた結果である．よって，レジスタの初期値は “0x300”(=768) である．

B.2.15　CBC_LOCK_REG_STATE

address = 0x901c



現在の lock register のマッチング状況 (一番最近受け取ったパケットとの比較結果) を示すレジスタ．lock register が割り込みの原因となった場合には対応するビットが ‘1’ となる．b0 が 0 番目の lock register，b1 が 1 番，b2 が 2 番，b3 が 3 番にそれぞれ対応する．

(*cf. CBC_INT_PKT_SRC+*)

B.2.16　CBC_LOCK_［BMP, MASK, ACT］(0-3)

address = 0x9020 – 0x902e

31　　　7 6　　　2 1 0

bitmap
bitmap[63:32]
bitmap[31: 0]

31　　　7 6　　　3 2　　0

mask	
—	C-Mask[35:29]
C-Mask[28:0]	A-Mask

31　　　1 0

action		
—	V	A

クラスタバスからの入力パケットに対してリトライまたは，割り込みを制御するためのレジスタ．lock register は表 B.5 の 3 つの要素から構成される．

表 B.5 lock register の構成

名前	ビット数	説明
bitmap	64	比較のためのパターン．
mask	39	どの部分を比較するかを指定する．
action	2	比較の結果一致した場合どのような動作を行うか指定する．

MMC ではクラスタバスからパケットが到着すると，その address flit(64 bit) と bitmap を比較する．比較の結果，mask で '0' になっている部分が全て一致していると，action で指定された動作を行う．なお，比較の際には入力パケットの `wwe` field は含めない．また，`sid` field が MBP であるようなパケットも対象外となる．

以下に各要素について説明する．

- bitmap

  bitmap には，比較を行うビット列を設定する．bitmap は b64 – b0 があり，これが入力パケットの address flit の b64 から b0 に対応する．また bitmap のうち有効な部分を次に述べる mask で指定する．また，mask で指定されなかった部分については無視される．

- mask

  bitmap のうち，どのビットの比較結果を有効とするかを指定する．この mask を指定することによって，address flit の一部のみを検査し，action を起こすようにすることができる．

  mask は更に以下の 2 つの部分に分けられる．

– b38 – b3: Control Mask (C-Mask)
  bitmap の b63 – b28 に対応する．値が ‘0’ の field に関して，一致検出を行う．値が ‘1’ のビットに対応する bitmap は無視する．
– b2 – b0: Address Mask (A-Mask)
  address 部 (b27 – b0) に対応する mask． address 部に関しては，制御部のようにすべてのビットに対応した mask を指定するのではなく，表 B.3 に示すようなパターン指定を行い，対応するビットに相当する部分のみ一致検出を行う．

• action

  action では，mask で指定された部分に関して bitmap と入力パケットの address flit の一致検査を行った結果一致した場合， どのような動作を行うかを指定する．

  表 B.6 に示すように，action は 2 つの field から構成される．

表 B.6 lock register での action

bit	名前	説明
b0	action	リトライ / 割込みを指定する．‘0’ のときリトライで，‘1’ のときが割込みである (ただし，reply packet は**必ず**割り込む)．
b1	valid	この lock register が有効かどうかを指定する．‘0’ のときは無効で，‘1’ のときが有効である．

付 録C　MBP Coreの命令

MBP Coreに用意された命令の種類とその動作について述べる．MBP Coreの全命令と命令codeを表C.1に示す．表中で使用する記号のうちImは直値を，Im_i^jはIm中のi bit目がj bit幅続くことを意味する．また，##は結合を示す．

C.1　WGG命令

GPRとGPRの間で語単位の演算を行うレジスタ間演算命令である．GPRの0番が常に0であるため，レジスタ間転送命令や比較命令は存在しない．また，算術論理演算はサポートしていない．表C.2にWGG命令の動作を示す．

C.2　WGI命令

GPRと直値間の命令であり，8 bitの直値演算を行う．表C.3にWGI命令の動作を示す．

C.3　分岐命令

分岐命令は，条件分岐がプログラムカウンタ相対番地指定で，他の分岐命令は絶対番地指定である．条件分岐に用いるフラグはCarry(C)とZero(Z)の2つである．また，BAL命令ではGPRの15番に現プログラムカウンタを待避し，復帰するにはBR命令を用いる．全ての分岐命令は1つの遅延スロットを持つ．そのため，BAL命令の後にBR命令で復帰すると，遅延スロットは2度実行される．また，BRで使うレジスタに対してのフォワーディングはサポートしていない．表C.4に分岐命令の動作を示す．

C.4　LMA(Local Memory Access)命令

local memoryに対するデータ参照は命令フェッチと重なる．よって，構造ハザードが生じて，1クロックのストールを起こす．しかし，データハザードによるストールは生じない．アドレシングは7 bitのdisplacement付きの間接レジスタ指定である．

I/Oに対する参照命令もここに分類されるが，別命令として実装される．I/Oに対するデータ参照では，wait信号をLowにすることによって，その参照をストールさせることが可能である(詳細は5.3.5節参照)．

なお，MBP Coreでは命令が21 bit幅であるのに対し，GPRは16 bit幅である．よって，通常のメモリ参照命令では上位5 bitが操作できない．そこで，上位5 bitを扱うため

表 C.1 MBP Core の命令

Mnemonic	Instruction Code					Mnemonic	Instruction Code				
SUBGG	11110	GPR1^4	GPR2^4	GPR3^4	0000	SUBPG	11111	PBR^8		GPR ^{4}	d000
ADDGG	11110	GPR1^4	GPR2^4	GPR3^4	0001	ADDPG	11111	PBR^8		GPR^4	d001
SLLGG	11110	GPR1^4	GPR2^4	GPR3^4	0010	MVBPG	11111	PBR^8		GPR^4	d010
SRLGG	11110	GPR1^4	GPR2^4	GPR3^4	0011	MVPG	11111	PBR^8		GPR^4	d011
ANDGG	11110	GPR1^4	GPR2^4	GPR3^4	0100	ANDPG	11111	PBR^8		GPR^4	d100
ORGG	11110	GPR1^4	GPR2^4	GPR3^4	0101	ORPG	11111	PBR^8		GPR^4	d101
XORGG	11110	GPR1^4	GPR2^4	GPR3^4	0110	XORPG	11111	PBR^8		GPR^4	d110
SUBGI	01000	GPR1^4	GPR2^4	Im^8		MVBPP	10111	PBR1^8		PBR2^8	
ADDGI	01001	GPR1^4	GPR2^4	Im^8		MVPP	11100	PBR1^8		PBR2^8	
SLLGI	01010	GPR1^4	GPR2^4	Im^8		MVLPP	11101	PBR1^4	PBR2^4	----	0000
SRLGI	01011	GPR1^4	GPR2^4	Im^8		PRDT	11101	0010	0000	0000	0010
ANDGI	01100	GPR1^4	GPR2^4	Im^8		GRDT	11101	0001	0000	0000	0010
ORGI	01101	GPR1^4	GPR2^4	Im^8		TGM	11000	----	PBR1^4	----	PBR2^4
XORGI	01110	GPR1^4	GPR2^4	Im^8		PBP	11001	----	PBR1^4	----	PBR2^4
LHI	01111	GPR1^4	----	Im^8		GBPS	11010	0---	PBR1^4	----	PBR2^4
BOZ	10000	00 Im^{14}				GBPM	11010	1---	PBR1^4	----	PBR2^4
BNZ	10000	01 Im^{14}				SIMG	11101	GPRd^4	GPRs^4	Im^4	1001
BGT	10000	10 Im^{14}				SIMP	11101	PBRs^4	GPRd^4	Im^4	1011
BLE	10000	11 Im^{14}				MEMBAR	11011	0000	0000	0000	1010
BAL	10001	Im^{16}				TJRQ	11101	----	----	----	0011
BRA	10010	Im^{16}				TJ	11101	----	GPR1^4	----	0100
BR	11101	----	GPR1^4	----	0001	RFI	11101	----	----	----	0110
LLM	10100	GPR1^4	GPR2^4	0 Im^7		INTE	11101	0000	1000	0000	0010
LLI	10100	GPR1^4	GPR2^4	1 Im^7		INTD	11101	0100	0000	0000	0010
SLM	10101	GPR1^4	GPR2^4	0 Im^7		CSM	11101	0000	0100	0000	0010
SLI	10101	GPR1^4	GPR2^4	1 Im^7		CAB	11101	0000	0010	0000	0010
SUBPI	00000	PBR^8		Im^8		CCB	11101	0000	0000	0001	0010
ADDPI	00001	PBR^8		Im^8		COV	11101	0000	0000	0010	0010
ADDPI	00010	PBR^8		Im^8		CUF	11101	0000	0000	0100	0010
SRLPI	00011	PBR^8		Im^8		CMT	11101	0000	0000	1000	0010
ANDPI	00100	PBR^8		Im^8		LIPA	11011	GPRd^4	----	----	0011
ORPI	00101	PBR^8		Im^8		LIPS	11011	GPRd^4	----	----	0100
XORPI	00110	PBR^8		Im^8		SSR	11011	GPRd^4	----	----	0101
XORPI	00111	PBR^8		Im^8		HSHC	11101	GPRd^4	PBRs^4	----	1100
CPI	10110	PBR^8		Im^8		HSHN	11101	GPRd^4	PBRs^4	----	1110
HSHG	11101	GPRd^4	GPRs^4	----	1101	BITS	11011	GPRd^4	GPR1^4	GPR2^4	0001
GUN	11101	GPRd^4	GPRs^4	----	1111	BITR	11011	GPRd^4	GPR1^4	GPR2^4	0000
LUDR	11011	GPRd^4	----	----	1000	BITC8	11011	GPRd^4	GPRs^4	----	0010
LIMG	11101	GPRd^4	GPRs^4	Im^4	1000	CRDT	11101	0000	0001	0000	0010
LIMP	11101	PBRd^4	GPRs^4	Im^4	1010	SUDR	11011	----	----	GPRs^4	1001

ただし-は任意のビットを意味する

表 C.2 WGG 命令の動作

Mnemonic	Function
SUBGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} - \mathrm{GPR3}$
ADDGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} + \mathrm{GPR3}$
SLLGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} << \mathrm{GPR3}$
SRLGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} >> \mathrm{GPR3}$
ANDGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} \wedge \mathrm{GPR3}$
ORGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} \vee \mathrm{GPR3}$
XORGG	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} \oplus \mathrm{GPR3}$

表 C.3 WGI 命令の動作

Mnemonic	Function
SUBGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} - \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
ADDGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} + \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
SLLGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} << \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
SRLGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} >> \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
ANDGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} \wedge \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
ORGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} \vee \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
XORGI	$\mathrm{GPR1} \leftarrow \mathrm{GPR2} \oplus \mathrm{Im}_7^8 \#\# \mathrm{Im}^8$
LHI	$\mathrm{GPR1} \leftarrow \mathrm{Im}^8 \#\# 0^8$

表 C.4 分岐命令の動作

Mnemonic	Function
BOZ	$\mathrm{if(Z)\ PC} \leftarrow \mathrm{PC} + \mathrm{Im}^{14}$
BNZ	$\mathrm{if(\bar{Z})\ PC} \leftarrow \mathrm{PC} + \mathrm{Im}^{14}$
BGT	$\mathrm{if(\bar{C} \wedge \bar{Z})\ PC} \leftarrow \mathrm{PC} + \mathrm{Im}^{14}$
BLE	$\mathrm{if(C \vee Z)\ PC} \leftarrow \mathrm{PC} + \mathrm{Im}^{14}$
BAL	$\mathrm{PC} \leftarrow \mathrm{Im}^{16},\ \mathrm{GPR15} \leftarrow \mathrm{PC}$
BRA	$\mathrm{PC} \leftarrow \mathrm{Im}^{16}$
BR	$\mathrm{PC} \leftarrow \mathrm{GPR}$

に，6 bit 幅の UDR(Upper Data Register) という特殊なレジスタを用意している．UDR の最も上位の 1 bit は LIO という信号線を制御するもので，残りの下位 5 bit に実際にメモリ上のデータが格納される．つまり，load の際にはメモリ上のデータの上位 5 bit が UDR の下位 5 bit に格納され，store の際にはその値がメモリ上のデータの上位 5 bit に書き込まれる．UDR と GPR 間には専用の命令が用意されており，6 bit 幅のデータをやり取りすることができる (C.12 節の特殊命令参照)．

ここで，注意すべきことは，UDR の最も上位の 1 bit がそのまま LIO として出力されてしまうことである．もし，I/O 空間から MBP-light を boot しつつ，local memory 上に命令を書く場合などでは，LIO を 1 にしておかなければならない．よって，UDR を設定する場合には最も上位の 1 bit を明示的に 1 に設定する必要がある．設定されない場合，local memory 空間から急に MBP-light が命令フェッチを始めてしまい，その動作は保証されない．逆に，I/O 空間から命令フェッチするのを止めて，local memory 空間から命令フェッチしたい場合には，最も上位の 1 bit に 0 を書き込む事で設定が可能である．この場合，local memory 空間での飛び先番地へ jump する命令の遅延スロットで書込みを行うのが望ましい．表 C.5 に LMA 命令の動作を示す．

表 C.5 LMA 命令の動作

Mnemonic	Function
LLM	GPR1 $\leftarrow$ M[GPR2+ Im_6^9## Im^7]
LLI	GPR1 $\leftarrow$ I[GPR2+ Im_6^9## Im^7]
SLM	M[GPR2+ Im_6^9## Im^7] $\leftarrow$ GPR1
SLI	I[GPR2+ Im_6^9## Im^7] $\leftarrow$ GPR1

I: I/O, space, M: Memory space

C.5　BPI 命令

PBR と直値間の命令であり，8 bit の直値と PBR の 1 offset(8 bit) 間の演算を行う．シフト操作はその 8 bit 内で行われる．表 C.6 に BPI 命令の動作を示す．

表 C.6 BPI 命令の動作

Mnemonic	Function
SUBPI	$PBR^8 \leftarrow PBR - Im^8$
ADDPI	$PBR^8 \leftarrow PBR + Im^8$
SLLPI	$PBR^8 \leftarrow PBR << Im^8$
SRLPI	$PBR^8 \leftarrow PBR >> Im^8$
ANDPI	$PBR^8 \leftarrow PBR \wedge Im^8$
ORPI	$PBR^8 \leftarrow PBR \vee Im^8$
XORPI	$PBR^8 \leftarrow PBR \oplus Im^8$
LPI	$PBR^8 \leftarrow Im^8$
CPI	$PBR^8 - Im^8$

C.6　WPG 命令

PBR と GPR 間の命令及び GPR と PBR 間の命令であり，GPR と指定された PBR 16 bit 間の演算を行う．表 C.7 に WPG 命令の動作を示す．表 C.1 に示す命令コード中の d で示されるビットが 1 であれば PBR が destination，d が 0 の場合は GPR が destination になる．

表 C.7 WPG 命令の動作

Mnemonic	Function
SUBPG	$PBR^{16} \leftarrow PBR^{16} - GPR^{16}$ $GPR^{16} \leftarrow GPR^{16} - PBR^{16}$
ADDPG	$PBR^{16} \leftarrow PBR^{16} + GPR^{16}$ $GPR^{16} \leftarrow GPR^{16} + PBR^{16}$
MVBPG	$PBR^{8} \leftarrow PBR^{8}$ $GPR^{8} \leftarrow GPR^{8}$
MVPG	$PBR^{16} \leftarrow PBR^{16}$ $GPR^{16} \leftarrow GPR^{16}$
ANDPG	$PBR^{16} \leftarrow PBR^{16} \wedge GPR^{16}$ $GPR^{16} \leftarrow GPR^{16} \wedge PBR^{16}$
ORPG	$PBR^{16} \leftarrow PBR^{16} \vee GPR^{16}$ $GPR^{16} \leftarrow GPR^{16} \vee PBR^{16}$
XORPG	$PBR^{16} \leftarrow PBR^{16} \oplus GPR^{16}$ $GPR^{16} \leftarrow GPR^{16} \oplus PBR^{16}$

C.7　MPP 命令

PBR と PBR 間の命令で，PBR 間の 8 bit や 16 bit，68 bit の転送命令である．表 C.8 に MPP 命令の動作を示す．68 bit 転送命令で offset は必要ないので，PBR は GPR によるポインタのみで指定する．

表 C.8 MPP 命令の動作

Mnemonic	Function
MVBPP	$PBR1^{8} \leftarrow PBR2^{8}$
MVPP	$PBR1^{16} \leftarrow PBR2^{16}$
MVLPP	$PBR1^{68} \leftarrow PBR2^{68}$

C.8　RDT 送受信命令

RDT 送受信命令は PBR の FIFO の切換え制御を行う命令である．表 C.9 に RDT 送受信命令を示す．使用する際には，必ず使用可能であることを確認する必要があり，検査機構のサポートとして割り込みを用いることができる (詳細は 5.3.4.5 節参照)．また，これらの命令は Level 2 と Level 3 の割り込みクリア命令としての役割も果たす．

表 C.9 RDT 送受信命令

Mnemonic	Function
PRDT	RDT packet を送信
GRDT	RDT packet を受信

C.9　MMC 送受信命令

MMC とのやりとりは表 C.10 で示した命令で行われる．offset は使用せず，PBR が GPR のみのポインタで指定する．

表 C.10 MMC 送受信命令

Mnemonic	Function
PBP	クラスタバスへのパケット送信命令
GBPS	SBUSCTL から割り込みパケット受信命令
GBPM	MTC から割り込みパケット受信命令
TGM	クラスタメモリへの参照命令
MEMBAR	メモリ参照の同期命令

- PBP

  PBP は指定した PBR の内容をクラスタバス上に送信する命令である．address flit と data flit の PBR を指定することで，クラスタバスに対してパケットを送信することができる．クラスタバス上へはどのようなパケットでも送信することが可能となっている．

- GBPS 及び GBPM

  GBPS 及び GBPM は MMC からパケットを受信するための命令である．GBPS が SBUSCTL からパケットを受信するのに対し，GBPM は MTC からパケットを受信する．受信の際には，address flit 格納用の PBR と data flit 格納用の PBR を指定する必要がある．

- TGM

  TGM はクラスタメモリや tag を参照するための転送命令である．上記の命令同様，address flit と data flit の PBR を指定しなければならない．しかし，クラスタバス

上のパケットとは異なり，MMC は MBP Core からのパケットでは以下のフィールドしか参照しない．

(1) ccbl

(2) command (scmd と pcmd)

(3) b47 (常に 0 にする必要がある)

(4) address

ただし，ccbl が 1 であれば block，0 であれば double word 単位の参照を表す．つまり，`size` field の代用となっている．なお，address flit の `wwe` field は `ccbl` が 1 ならば 10 と，0 ならば 01 としなければならない．また，アドレスは MBP 向けに特殊になっており，そのアドレス体系を図 C.1 に示す．

では，実際にどのような方法で参照するかを以下に示す．

– クラスタメモリの参照

  ccbl は 0 と 1 のどちらでも可能である．command は書込み時には write_req を，読出し時には read_req を指定する．そして，図 C.1 の DADDR field に address を設定する．

  また，書込み時に data flit の `wwe` は SDRAM 上に設けてある tag(同期ビット) を更新する．一方，読出し時に PBR の offset 8 には SDRAM 上に設けてある tag (同期ビット) の値も同時に読んでくる．

– SRAM tag memory の参照

  ccbl は 0 でなければならない．command は書込み時には control_write_req を，読出し時には control_read_req を指定する．そして，図 C.1 の DADDR field に address を設定する．その他にも D/T field を 1 に，S/D filed を 0 に設定しておく．

– SDRAM tag memory の参照

  ccbl は 1 でなければならない．command は書込み時には control_write_req を，読出し時には control_read_req を指定する．そして，図 C.1 の DADDR field に address を設定する．その他にも D/T field を 1 に，S/D filed を 1 に設定しておく．

なお，MBP Core が明示的に ECC 用メモリを書き込むことはできない．

- MEMBAR

上記で説明した命令の完了には 10 クロック以上必要とする．もし MBP Core がその完了まで待つとしたら，性能を劣化させる原因となりうる．よって，MBP Core は，上記の命令が使用中の PBR の属する種類の PBR を参照しないような命令であれば，続けて実行が可能な構造をとっている．例えば，TGM が使用中の PBR の種類が汎用に属していれば，後続の命令では RDT 送信用と RDT 受信用の PBR や，GPR を参照することが可能である．

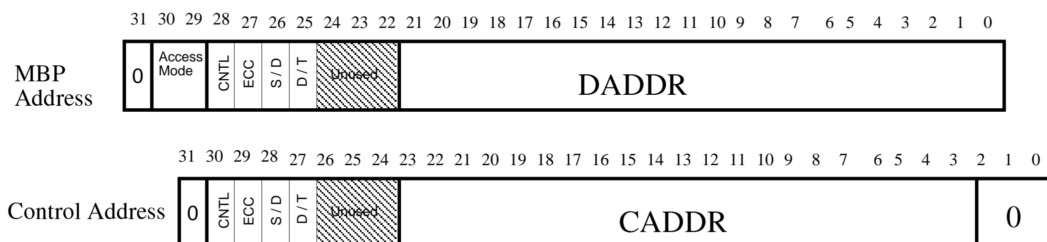


図 C.1 MBP Core から見た MMC の address 体系

しかし，この構造は 1 つの問題を生じる．その問題とは，いつ上記の命令による MMC の処理が完了したかを判別できないことである．そこで，MEMBAR が用意されている．この MEMBAR を発行することにより，その完了を MBP Core は待ち合わせることができるようになる．また，もし後続の命令で同種の PBR を参照したい場合にも MEMBAR を発行することで，正しい動作を保証することができる．ただし，完全に MMC の処理が完了していることが何らかの形でわかる場合には，MEMBAR を発行する必要はない．

C.10　Table Jump 命令 (TJ)

Table Jump 関連の命令を表 C.11 に示す．TJRQ は外部からの要因に従って予約された内部メモリの番地から値を読み出し，そこへジャンプする命令である．一方，TJ は GPR で示された内部メモリ番地に格納されている値を読み出し，その番地にジャンプする命令である．TJ も TJRQ も 1 命令分の遅延スロットをもつ．また，TJ に指定する GPR の値はフォワーディングされない．

表 C.11 Table Jump 命令

Mnemonic	Function
TJRQ	外部要因による Table Jump
TJ	Table Jump

C.11　割込み命令 (INT)

割込み関連の命令を表 C.12 に示す．

表 C.12 割込み命令

Mnemonic	Function
RFI	割込み復帰命令
INTE	割込み enable
INTD	割込み disable
CSM	Level 0 の割込み clear 命令
CAB	Level 1 の割込み clear 命令
CCB	Level 4 の割込み clear 命令
CMT	Level 5 の割込み clear 命令
COV	Level 6 の割込み clear 命令
CUF	Level 7 の割込み clear 命令
LIPA	IPA の読出し命令
LIPS	IPS の読出し命令
SSR	IPS の設定命令

割込み命令には割込みイネーブル/ディスエーブル命令 (INTE / INTD) と割込み復帰命令 (RFI) がある．INTE を実行すると，後述の割込み mask の設定に従って割込みをイネーブルにする．割込みが起こると，IPA(InterruPt Address) register と IPS(InterruPt Status) register に割込み前の状態が保存される．そして，予約された内部メモリの番地から値が読み出され，該当番地へジャンプする．この IPA や IPS は LIPA や LIPS 命令で GPR に読み出すことが可能であり，多重割込みも可能となる．保存した IPS を復帰させるには SSR 命令を用い，復帰命令には BR を用いる．また，割込み要因をクリアするための命令も設けてある．

C.12　特殊命令 (SPE)

MBP Core ではパケットのヘッダ部のフィールドを操作する処理や表管理を行う処理が多いと見込まれる．よって，その際に有効と思われる特殊命令を数多くサポートしている．表 C.13 に特殊命令を示す．

C.13　IMA(内部メモリ access) 命令

内部メモリを参照するための命令を表 C.14 に示す．アドレシングは 4 bit の offset 付きの間接レジスタ指定である．内部メモリのアドレス空間については 5.3.4.4 節を参照のこと．また，PBR と内部メモリ間の load/store 命令もサポートしている．

表 C.13 特殊命令

Mnemonic	Function
HSHC	PBRs の 27 bit 目から 5 bit 目の 23 bit を 8 bit 単位に折り畳んで，Exclusive OR を演算する．そして，結果の 8 bit を GPRd の下位 8 bit に設定する．
HSHG	GPRs 上の 16 bit を 8 bit 単位に折り畳んで，Exclusive OR を演算する．そして，結果の 8 bit を GPRd の下位 8 bit に設定する．
HSHN	PBRs の 41 bit 目から 5 bit 目までを 8 bit 単位で折り畳んで，Exclusive OR を演算する．そして，結果の 8 bit を GPRd の下位 8 bit に設定する．
GUN	GPRs 上の 1 である最下位ビットのビット番号を GPRd に設定する．
BITS	GPR1 上の GPR2 bit 目のビットを 1 にし，GPR3 に格納する．
BITR	GPR1 上の GPR2 bit 目のビットを 0 にし，GPR3 に格納する．
BITC8	GPRs 上の下位 8 bit 中の 1 の数を数えて，GPRd に格納する．
CRDT	RDT Interface を reset する命令.
LUDR	UDR を GPRd 上に読み出す命令．UDR の値はフォワーディングされない.
SUDR	UDR を GPR の下位 6 bit で示された値に設定する命令. UDR の値はフォワーディングされない.

表 C.14 IMA 命令

Mnemonic	Function
LIMG	$\mathrm{GPRd}^{16} \leftarrow \mathrm{IM}[\mathrm{GPRs}+\mathrm{offset}^{4}]$
SIMG	$\mathrm{IM}[\mathrm{GPRd}+\mathrm{offset}^{4}] \leftarrow \mathrm{GPRs}^{16}$
LIMP	$\mathrm{PBRd}^{68} \leftarrow \mathrm{IM}[\mathrm{GPRs}+\mathrm{offset}^{4}]$
SIMP	$\mathrm{IM}[\mathrm{GBRd}+\mathrm{offset}^{4}] \leftarrow \mathrm{PBRs}^{68}$

付 録D　クラスタバスの仕様

D.1　パケットの構成

トランザクションは1つ以上のパケットの系列からなる．パケットはアドレスやデータを示す64 bitのフリットをいくつか組み合わせたものであり，連続したバスサイクルで転送される．address flitの構成を図D.1に示す．また，data flitの構成を図D.2に示す．

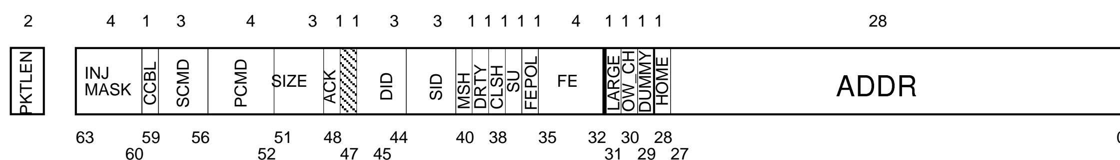


図 D.1 address flitの構成

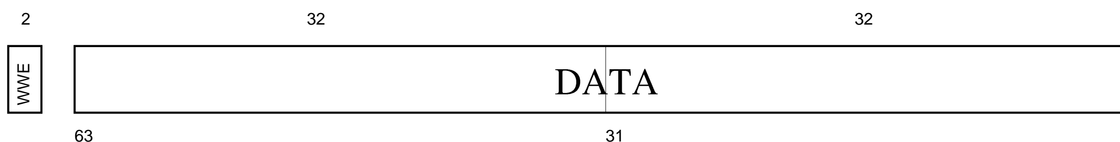


図 D.2 data flitの構成

各フィールドの意味を以下に記す．

(1) `pktlen`

address flitに付加されたパケット長を示す．0なら1フリット，1なら2フリット，2なら5フリットを意味する．`wwe`と書かれることもある．

(2) `injmask`

1であれば，injectionを行うことを示す．

(3) `ccbl`

1であれば，cacheableなアドレスを対象としたパケットであることを示す．

(4) `pcmd/scmd`

パケットの種類を指定する．

(5) `size`

パケットが操作対象とするデータ幅を示す．対応を表 D.1 に示す，

表 D.1 `size` field とデータ幅の対応

field 値	データ幅	field 値	データ幅
000	byte	100	block
001	half word	101	reserved
010	word	110	masked block
011	double word	111	none

(6) `ack`

1 であれば，要求の応答確認を行う．

(7) `did`

パケットの宛先を示す．対応を表 D.2 に示す．

表 D.2 `did` field とデバイスの対応

field 値	デバイス	field 値	デバイス
000	0 番キャッシュ	100	MBP-light
001	1 番キャッシュ	101	reserved
010	2 番キャッシュ	110	reserved
011	3 番キャッシュ	111	reserved

(8) `sid`

パケットの送信元を示す．対応は表 D.2 と同じである．

(9) `drty`

1 であれば，対象とするキャッシュラインが dirty であることを示す．

(10) `clsh`

1 であれば，対象とするキャッシュラインがクラスタ間で共有されていることを示す．

(11) `fe`

対象とするキャッシュラインの各 double word の同期ビットの値．

(12) `addr`

対象とするアドレス．

(13) `wwe`

(a) 要求パケットの data flit に付加している場合
`wwe` の 1 bit 目が上位 32 bit のデータ，0 bit 目が下位 32 bit のデータに対する書き込み mask となる．

(b) 応答パケットの data flit に付加している場合
同期待ち情報を表す (5.3.3.2 節を参照されたい)．

(14) `data`

data flit の値．

なお，`msh` や `su`，`fepol` などのフィールドは 2 次キャッシュにとって意味がなく，2 次キャッシュ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んな値でもよい．

そして，実際にバス上に流れるパケットを表 D.3 にまとめる．

ただし，$C \to B(Cache \to Bus)$ は 2 次キャッシュからクラスタバス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．$B \to C(Bus \to Cache)$ はその逆である．また，各 field の値に記された記号は以下の意味をもつ．

(1) 0

2 次キャッシュ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．

(2) 1

2 次キャッシュ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．

(3) $*$

2 次キャッシュ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なければならない．

(4) $\perp$

2 次キャッシュ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．

(5) $\times 1, \times 4$

data flit が 1 または 4 フリット存在する．

(6) –

data flit が存在しない．

各パケットの mnemonic について以下に列挙する．reply とつくパケットは，基本的に req とつく要求パケットの応答であるため，ここでは省略した．また，Home とは参照するアドレスによって唯一定まるクラスタであり，詳細は 5.1.9 節を参照されたい．

表 D.3 パケット一覧

packet mnemonic	C↓B	B↓C	plen	imsk	ccbl	scmd	pcmd	size	ack	did	sid	drty	clsh	fe	addr	wwe	data
read_req(C)	○	○	00	0	1	000	0000	100	0	111	∗	⊥	⊥	⊥	∗	−	−
read_req(N)	○	×	00	0	0	000	0000	∗	0	100	∗	⊥	0	⊥	∗	−	−
n_c_rd_req	○	×	00	0	1	001	0000	100	0	100	∗	⊥	⊥	⊥	∗	−	−
swap_req(C)	○	○	01	0	1	011	0000	∗	0	111	∗	⊥	⊥	⊥	∗	⊥ × 1	∗ × 1
swap_req(N)	○	×	01	0	0	011	0000	∗	0	100	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
bq_read_req	○	○	00	0	1	100	0000	100	0	100	∗	⊥	⊥	⊥	∗	−	−
bq_nop_req	○	○	00	0	1	101	0000	100	0	100	∗	⊥	⊥	⊥	∗	−	−
read_reply(C)	○	○	10	0	1	000	0001	100	0	∗	∗	∗	∗	∗	∗	∗ × 4	∗ × 4
read_reply(N)	×	○	01	0	0	000	0001	∗	0	∗	100	⊥	⊥	⊥	∗	⊥ × 1	∗ × 1
n_c_rd_reply	×	○	10	0	1	001	0001	100	0	∗	100	0	∗	∗	∗	∗ × 4	∗ × 4
swap_reply(C)	○	○	10	0	1	011	0001	100	0	∗	∗	∗	∗	∗	∗	∗ × 4	∗ × 4
swap_reply(N)	×	○	01	0	0	011	0001	∗	0	∗	100	⊥	⊥	⊥	∗	⊥ × 1	∗ × 1
bq_read_reply	×	○	10	0	1	100	0001	100	0	∗	100	0	0	∗	∗	∗ × 4	∗ × 4
bq_nop_reply	×	○	00	0	1	101	0001	100	0	∗	100	⊥	⊥	∗	∗	−	−
allread_req	○	○	00	0	1	000	0010	100	0	111	∗	⊥	⊥	⊥	∗	−	−
injection_req	×	○	10	∗	1	010	0010	100	⊥	111	100	0	∗	∗	∗	∗ × 4	∗ × 4
swap_m_req	○	×	01	0	0	011	0010	∗	0	100	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
allread_reply	○	○	10	1	1	000	0011	100	0	∗	∗	∗	∗	∗	∗	∗ × 4	∗ × 4
swap_m_reply	×	○	01	0	0	011	0011	∗	0	∗	100	⊥	⊥	⊥	∗	⊥ × 1	∗ × 1
rdinv_req	○	○	00	0	1	000	0100	100	0	111	∗	⊥	⊥	⊥	∗	−	−
rdinv_p_req(C)	○	×	00	0	1	100	0100	100	0	100	∗	⊥	⊥	⊥	∗	−	−
rdinv_p_req(N)	○	×	00	0	0	100	0100	∗	0	100	∗	⊥	0	⊥	∗	−	−
rdinv_q_req(C)	○	×	00	0	1	110	0100	011	0	100	∗	⊥	⊥	⊥	∗	−	−
rdinv_q_req(N)	○	×	00	0	0	110	0100	∗	0	100	∗	⊥	0	⊥	∗	−	−
rdinv_f_req	○	×	00	0	0	111	0100	∗	0	100	∗	⊥	0	⊥	∗	−	−
rdinv_reply	○	○	10	0	1	000	0101	100	0	∗	∗	∗	0	∗	∗	∗ × 4	∗ × 4
rdinv_p_reply(C)	×	○	10	0	1	100	0101	100	0	∗	100	1	0	∗	∗	00 × 4	∗ × 4
rdinv_p_reply(N)	×	○	01	0	0	100	0101	∗	0	∗	100	⊥	⊥	∗	∗	⊥ × 1	∗ × 1
rdinv_q_reply(C)	×	○	01	0	1	110	0101	011	0	∗	100	1	0	∗	∗	00 × 1	∗ × 1
rdinv_q_reply(N)	×	○	01	0	0	110	0101	∗	0	∗	100	⊥	⊥	∗	∗	⊥ × 1	∗ × 1
rdinv_f_reply(N)	×	○	01	0	0	111	0101	∗	0	∗	100	⊥	⊥	∗	∗	⊥ × 1	∗ × 1
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
write_req	○	×	01	0	0	010	1000	∗	1	100	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
cflush_req	×	○	00	0	1	011	1000	100	⊥	111	100	⊥	⊥	⊥	∗	−	−
update_via_home_q_req	○	×	01	0	0	110	1000	∗	∗	100	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
update_req	×	○	10	0	1	000	1001	110	⊥	111	100	0	1	∗	∗	∗ × 4	∗ × 4
writeback_reply	○	×	10	0	1	010	1001	100	0	100	∗	∗	∗	∗	∗	∗ × 4	∗ × 4
update_direct_req	○	○	10	0	1	000	1010	110	∗	111	∗	⊥	∗	∗	∗	∗ × 4	∗ × 4
n_c_write_m_req(C)	○	×	01	0	1	010	1010	001	1	100	∗	⊥	⊥	⊥	∗	⊥ × 1	∗ × 1
n_c_write_m_req(N)	○	×	01	0	0	010	1010	∗	1	100	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
update_direct_s_req(C)	○	○	10	0	1	100	1010	110	∗	111	∗	⊥	∗	∗	∗	∗ × 4	∗ × 4
update_direct_s_req(N)	○	×	01	0	0	100	1010	∗	∗	100	∗	⊥	0	∗	∗	⊥ × 1	∗ × 1
invalidate_via_home_req	○	○	00	0	1	000	1100	100	1	111	∗	⊥	∗	⊥	∗	−	−
no_alloc_write_req†	○	○	10	0	1	100	1100	110	∗	111	∗	⊥	⊥	∗	∗	∗ × 4	∗ × 4
bq_write_req	○	×	10	0	1	101	1100	100	∗	100	∗	∗	∗	∗	∗	∗ × 4	∗ × 4
invalidate_req	×	○	00	0	1	000	1101	100	⊥	111	100	⊥	⊥	⊥	∗	−	−
control_read_req	○	○	00	0	0	000	1110	∗	0	∗	∗	⊥	0	⊥	∗	−	−
demap_req	×	○	01	0	0	001	1110	⊥	⊥	∗	100	⊥	⊥	⊥	⊥	⊥ × 1	∗ × 1
control_write_req	○	○	01	0	0	010	1110	∗	1	∗	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
interrupt_req	×	○	01	0	0	011	1110	011	⊥	∗	100	⊥	⊥	⊥	∗	⊥ × 1	∗ × 1
control_read_reply	○	○	01	0	0	000	1111	∗	0	∗	∗	⊥	0	⊥	∗	⊥ × 1	∗ × 1
demap_ack	○	×	00	0	0	001	1111	⊥	0	100	∗	⊥	0	⊥	⊥	−	−
error_reply	○	×	00	0	⊥	110	1111	⊥	⊥	100	∗	⊥	⊥	⊥	∗	−	−
nop	○	×	00	⊥	⊥	111	1111	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

† MBP-light が送出する際には update_req で代用しなければならない

- read_req(C)

  block 単位のデータ読出し要求である．

- read_req(N)

  データの non-cacheable な読出し要求である．

- n_c_rd_req

  block 単位のデータ読出し要求であるが，キャッシュのスヌープを伴わない．

- swap_req(C)

  データのスワップ要求である．

- swap_req(N)

  データの non-cacheable なスワップ要求である．

- bq_read_req

  block を要素とする FIFO queue における block 単位での読出し要求，またはプリフェッチ要求である．

- bq_nop_req

  block を要素とする FIFO queue における block 単位でのプリフェッチ取り消し要求である．prefetch 要求がバスに送出されるまでの間に，読出し権が移動した場合に用いられる．

- allread_req

  block 単位で他のキャッシュへの injection を伴うデータ読出し要求である．

- injection_req

  MBP-light が送出する injection 要求である．

- swap_m_req

  応答を伴う MBP-light への処理 (スワップ) 要求である．

- rdinv_req

  block 単位で無効化を伴うデータ読出し要求である．

- rdinv_p_req(C)

  double word を要素とする FIFO queue における block 単位での読出し要求である．ただし，ある 2 次キャッシュがこのデータの読出し権を所有している場合に限る．

- rdinv_p_req(N)

  double word を要素とする FIFO queue における non-cacheable な dequeue 要求である．

- rdinv_q_req(C)

  double word を要素とする FIFO queue における double word 単位での読出し，またはプリフェッチ要求である．ただし，対象ラインが共有されていない場合に限る．

- rdinv_q_req(N)

  double word を要素とする FIFO queue における dequeue を伴う non-cacheable な読出し要求である．

- rdinv_f_req

  double word を要素とする FIFO queue の状態検査要求である．先頭要素は dequeue されない．

- update_via_home_req

  Home を経由したデータの更新要求である．

- write_req

  non-cacheable な書込み要求である．

- cflush_req

  MBP-light が送出する書き戻しを伴う無効化要求である．

- update_via_home_q_req

  double word を要素とする FIFO queue における enqueue 要求である．

- update_req

  MBP-light が送出する更新要求である．

- writeback_reply

  クラスタメモリへの書戻しのための擬似応答である．

- update_direct_req

  Home を経由しないデータの更新要求である．

- n_c_write_m_req(C)

  MBP-light が行う prefetch/flush 操作の起動要求である．

- n_c_write_m_req(N)

  MBP-light が行う prefetch/flush 操作の起動要求である．

- update_direct_s_req(C)

  対象 line に書込み待ちスレッドが存在するような場合の更新要求である．

- update_direct_s_req(N)

  同期機構の non-cacheable な書込み要求である．

- invalidate_via_home_req
  Home を経由したデータの無効化要求である．
- no_alloc_write_req
  dummy page に対する更新要求である．
- bq_write_req
  block を要素とする FIFO queue における enqueue 要求である．
- invalidate_req
  MBP-light が送出する無効化要求である．
- control_read_req
  バス上のデバイスの内部資源読出し要求である．
- demap_req
  CPU に対する demap 要求である．
- control_write_req
  バス上のデバイスの内部資源書き込み要求である．
- interrupt_req
  CPU に対する割り込み要求である．

具体的な処理に関しては文献 [34] を参照されたい．

D.2　制御線

簡単にクラスタバスで用いられる制御線について説明する．ここでの入出力はクラスタバスを基準として考えている．

各デバイスはクラスタバスを得るために，まず 2 bit の信号線 `X_BREQ` をアサートしなければならない．要求パケットには 0 bit 目が用いられ，応答パケットには 1 bit 目が用いられる．同様に，各デバイスがパケットを受理不可能であることを表す信号線 `X_BBUSY` も 2 bit 設けられており，パケットの種類によって使用されるビットは上記と同じである．当然，バスの使用が割り当てられたことを各デバイスへ通知する信号線 `X_BGRT` も備えられていることを明記しておく．

さて，パケットの転送の際にデバイスはパケットの先頭を表す信号線 (`X_ASTB_I`) が必要である．この信号線がアサートされている間に送出されているパケットのフリットは address flit である．また，パケットの送信を開始したことを表す基準点となる．クラスタバスは，信号線 `X_ASTB_I` を一旦ラッチした `X_ASTB_O` を通じて address flit であることをその他のデバイスへ通知する．ここで，デバイス A と B，C が長さ 2 のパケットを送受信する様子を図 D.3 に示す．

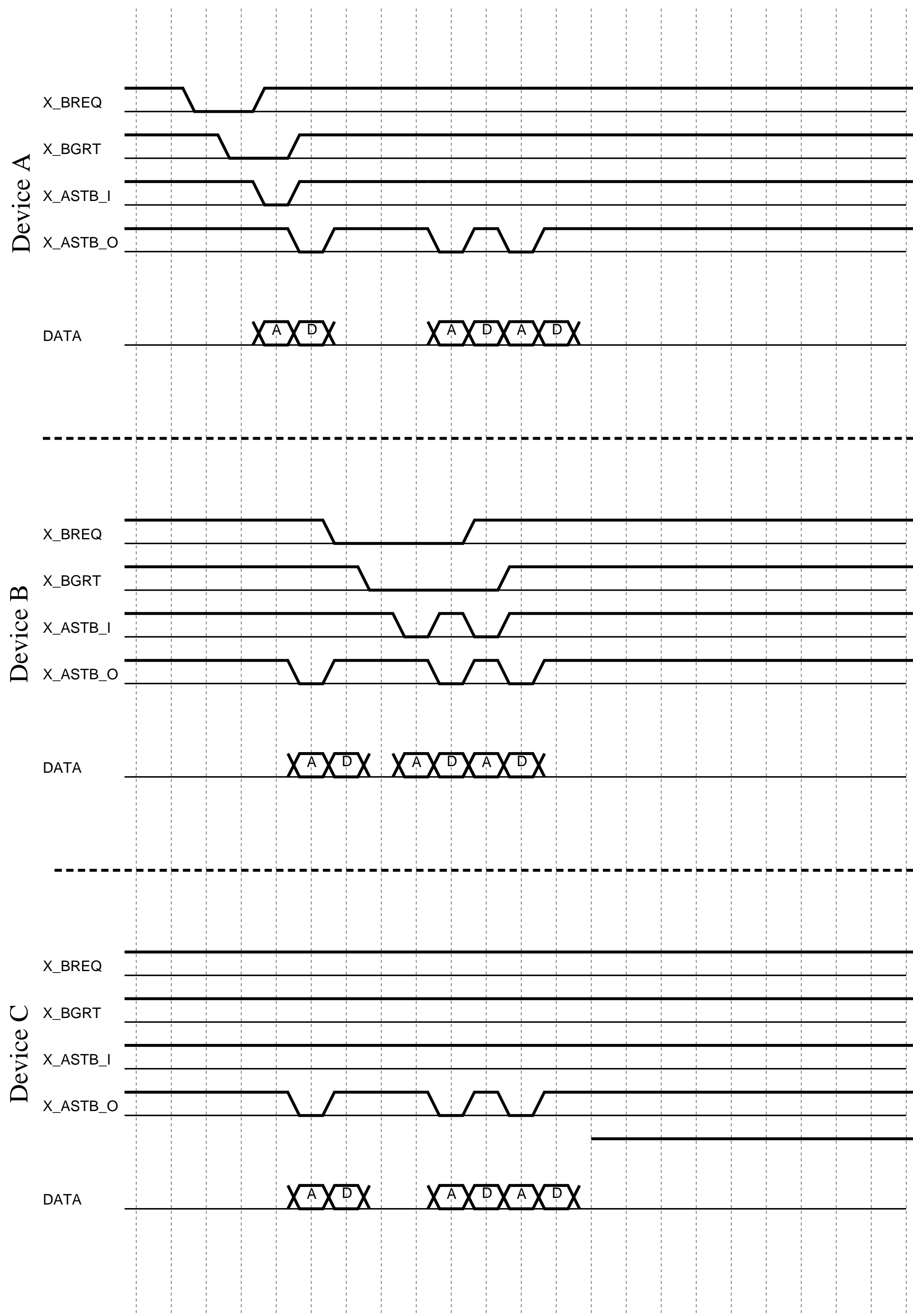


図 D.3 長さ 2 のパケットによる送受信の様子

クラスタバス上を流れる要求パケットに対してデバイスは retry をすることができる．retry された要求パケットは全く無かったものとして扱われるため，関係するデバイスは retry かどうか確定するまで種々の資源の変更を行うことはできない．また，応答パケットを retry することはできない．以下にそのタイミングについて説明する．

(1) 第 i bus clock から要求パケットの送信を開始．

(2) その他のデバイスがリトライを必要だと判断したならば，第 $i+2$ bus clock でリトライ要求 (信号線 X_BRETRY_I の assert)．

(3) 送信元のデバイスが第 $i+4$ bus clock でリトライ要求 (信号線 X_BRETRY_O のアサート) を受ければ，その送信を取り下げて再びバスを要求する．そうでなければ，その送信は受け付けられたことになる．

ただし，信号線 X_BRETRY_O は各デバイスから入力した X_BRETRY_I の論理和をとり，一旦ラッチした信号である．また，クラスタバスには 2 次キャッシュの共有情報に関する制御線も備えられている．これは次の 2 つである．

- 現在転送中または直前に転送したパケットに対してキャッシュラインを共有していることを表す信号線 (X_MSH_{I,O})

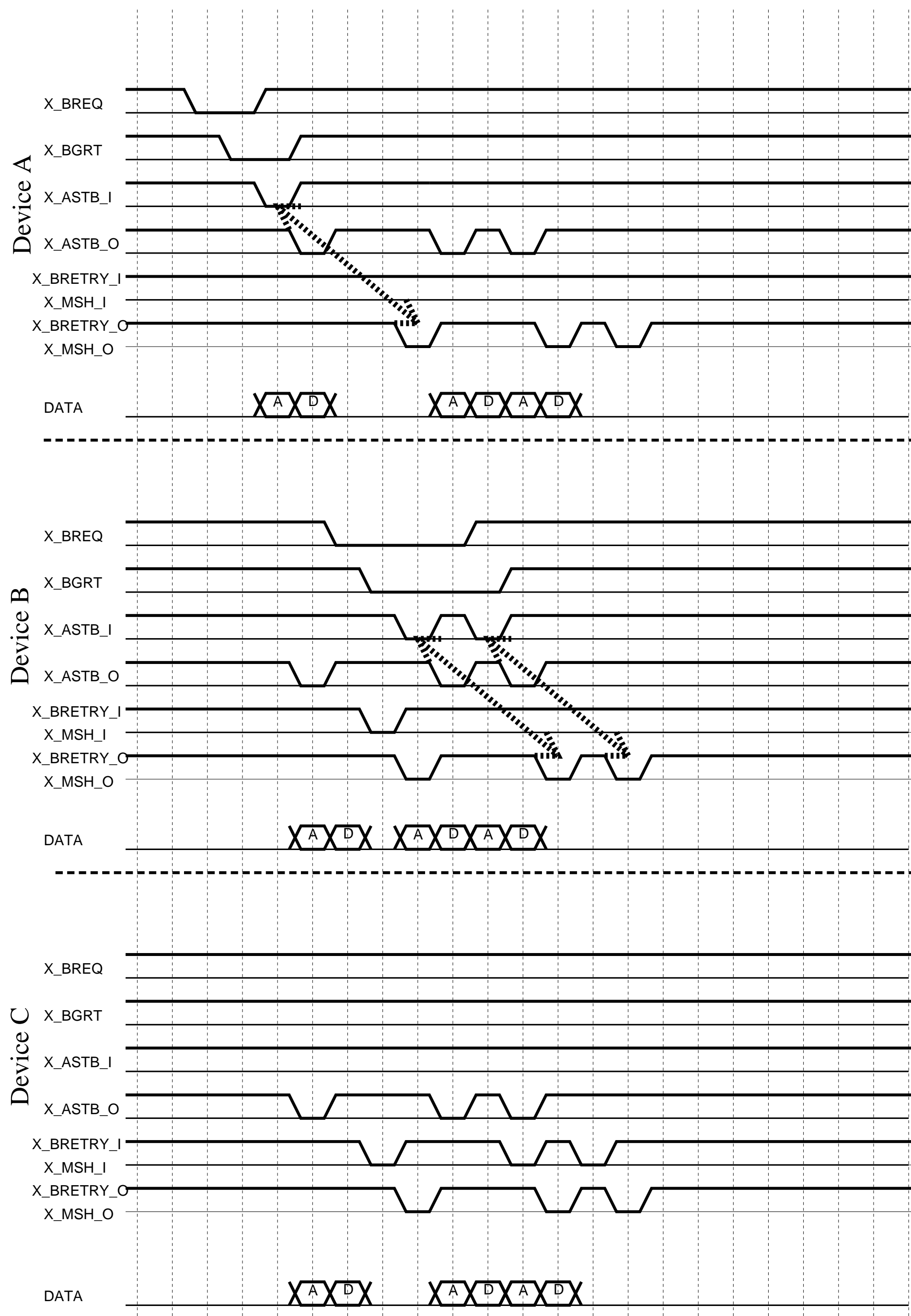
  ⇒ キャッシュラインが共有されているかどうかを判断する (キャッシュラインの状態遷移に影響を与える)．

- 現在転送中または直前に転送したパケットに対してキャッシュラインの owner が応答することを表す信号線 (X_CRPLY_{I,O})

  ⇒ MBP-light による応答動作を抑制する．

各デバイスによるタイミングは上記に述べたリトライと同じである．ここで，デバイス A と B，C による制御線のタイミングを図 D.4 に示す．

さらに，SuperSPARC+で採用された PSO において，write 間のプログラム順序を保つ STBAR と呼ばれる fence 命令を実装するために，信号線 X_PEND が用いられる．2 次キャッシュから ack フィールドが 1 である要求パケットを受理した MBP-light はそれぞれの 2 次キャッシュごとに設けられたカウンタ (詳細は 5.3.3.3 節を参照されたい) をインクリメントし，X_PEND をアサートする．そして，当該要求の処理を終えた MBP-light は送信元の 2 次キャッシュに対応するカウンタをデクリメントする．もしカウンタの残数が 0 であれば，MBP-light は X_PEND をネゲートし，全ての要求を処理したことを 2 次キャッシュに知らせる．



A: Send –> Receive –> Receive,　B:　Receive(retry) –> Send –> Send,　C: Receive(retry) –> Receive(retry) –> Receive(retry)

図 D.4 制御線のタイミング

付 録E　memory command

memory command の一覧を表 E.1 に示す．ここで，ctype 欄は 2 次キャッシュの内部資源 `L2CCTL.ctype[1:0]` の値により，4 種類のセットのいずれかが選ばれることを意味する．また，cache 欄は cacheability に関して以下の意味を持つ．

(1) yes

page table の指定が L1-cacheable でなければならず，また無条件に L2-cacheable である．

(2) y/n

page table の指定が L1-cacheable であってもなくてもよく，L1-cacheable であれば (また，そのときに限り)L2-cacheable である．

(3) y/n(S)

page table の指定が L1-cacheable であってもなくてもよく，L1-cacheable であれば L2-cacheable である．L1-cacheable でなければ，2 次 cache の cacheability はその内部資源 `L2CCTL.l1ccbl` に従う．

(4) no

page table の指定が L1-cacheable であってはならず，また無条件に L2-non-cacheable である．

(5) no(S)

page table の指定が L1-cacheable であってはならず，2 次 cache の cacheability はその内部資源 `L2CCTL.l1ccbl` に従う．

(6) no(Q)

page table の指定が L1-cacheable であってはならず，常に L2-cacheable と解釈される．

なお，cacheability の可否が定められたものに対して，それに反する指定を行って参照することは許されていない．また，cacheable な 2 つの memory command である C_1 と C_2 に対し，同一アドレスに対して C_1 による書込みを行った後で C_2 による読出しを行うと，書込み結果が反映されていない場合がある．なぜなら，JUMP-1 の物理アドレスとしては同一と解釈されるが，SuperSPARC+の内部では異なるアドレスとなる．よって，1 次キャッシュ内のそれぞれ異なるエントリにマップされてしまう可能性がある．この問題の対策として，2 次キャッシュは C_1 による書込みを検知すると，直ちに C_2 に対応する 1

表 E.1 memory command 一覧

mcmd	ctype	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	F0	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	F1	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	0-2	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

D: Double word, W: Word, H: Half word, B: Byte

次キャッシュのエントリを無効化する．しかし，SuperSPARC+には書込みをバッファリングする機構があるために，2 次キャッシュが検知する前に読出しが実行されてしまうことがある．そこで，C_1 による書込みを行った後で STBAR を実行し 2 次キャッシュに明示的に書込みを検知させた後に，C_2 による読出しを行わなければならない．

次に，coh-pol 欄は L2-cacheable であるときのコヒーレンスポリシを表す．ただし，これには load や store，swap の 3 種類があり，表 E.2 に対応を示す．

表 E.2 coherence-policy

coh-pol	load	store	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

(1) load ... 主として load がキャッシュミスした際のラインの取得方法を意味する．

(a) normal
付随的な動作を一切行わない．

(b) all-read
ミスしたキャッシュと近縁関係にあるキャッシュに対して，データの injection を行う．これにより，同じデータが必要な近縁関係にあるプロセッサはキャッシュミスを起こすことなく，データを入手できる．この injection を行うかどうかの判断は 2 次キャッシュの内部資源 `INJLEV` に従う．

(c) read-invalidate
ミスした際にもすべてのコピーを無効化する機能を付加したものである．例えば，参照と更新が pair になっているような共有データを考える．すると，これを使うことにより参照時の通信と書込み時の通信を 1 回で済ませることができる．

(d) no-coherent
一貫性を無視してラインを取得する．

(2) store ... 主として store が shared hit した際の更新操作を意味する．

(a) invalidate
すべてのコピーが無効化される．store 時のキャッシュミスでは load の coherence-policy が何であれ read-invalidate によってラインを取得する．

(b) update

クラスタ間で共有されていれば，home cluster と呼ばれる参照するアドレスによって定まる特別なクラスタを経由して他のコピーを更新する (home cluster の詳細は 5.1.9 節 を参照されたい)．すると，書込みのたびにすべてのコピーへデータが送られるので，通信が多発する傾向にある．ただし，時間的に近接した同一アドレスへの書込みは，キャッシュの write buffer によってマージされる．また，store 時のキャッシュミスでは load の coherence-policy に従ってラインを取得する．

(c) update-direct

クラスタ間で共有されていれば，home cluster を経由せずに他のコピーを更新する．これにより無駄に home cluster で逐次化する必要がなくなる．また，store 時のキャッシュミスでは load の coherence-policy に従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．また，store 時のキャッシュミスでは load の coherence-policy に従ってラインを取得する．

(3) swap ... 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

すべてのコピーが無効化される．

(b) update

クラスタ間で共有されていれば，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．

最後に，load や store，swap 欄は memory command の機能を意味しており，それぞれ以下に説明する．

(1) N = {normal-read, normal-write, normal-swap}

付加的な機能が一切ない通常の read や write，swap である．

(2) RG = {rg-read, rg-write}

2 次キャッシュの内部資源に対する参照である．

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

(4) T = t-write

MBP-light の起動要求である．

(5) $\{S0, S1\} = \{\text{s-read}, \text{s-write}\}$

JUMP-1 では I-structure [76] による通信/同期機構を備えている．これはすべての double word のデータに同期ビットを 2 bit 用意することで実現される．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-cacheable であれば 0 を返す．また，L1-non-cacheable であれば，未定義の error を返す．

(6) $\{F0, F1\} = \text{f-read}$

I-structure の full/empty を知るための読出しであり，full であれば -1 を返す．また，empty ならば 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 は同期ビットが 1 の場合に empty であることを意味する．

(7) $Q = \{\text{q-read}, \text{q-write}\}$

(8) $P = \text{p-read}$

JUMP-1 では Q-structure [45] と呼ばれる通信/同期機構により，double word を要素とする FIFO queue のサポートを行う．q-read は先頭要素を返し，その要素を queue から取り除く．もし queue が empty である場合には，未定義 error を返す．p-read は empty である場合に，0 を返すという点が q-read と異なる．そして，q-write は queue の末尾にデータを付加する．

(9) $QC = \text{q-check}$

double word を要素とする FIFO queue の full/empty を知るための読出しであり，full であれば -1 を，empty であれば 0 を返す．

(10) $BQ = \{\text{bq-read}, \text{bq-write}\}$

JUMP-1 では Block Queue [77] と呼ばれる通信/同期機構により，キャッシュラインを要素とする FIFO queue のサポートを行っている．bq-read は先頭要素の取出しと，その要素中の個々のデータの読出しを行う．もし queue が empty ある場合には，未定義 error を返す．そして，bq-write は queue の末尾に付加すべき要素中の個々のデータの書込みと実際の付加を行う．

(11) $BC = \text{bq-check}$

キャッシュラインを要素とする FIFO queue の full/empty を知るための読出しであり，full であれば -1 を，empty であれば 0 を返す．

付 録F　SPARCのboot

ここでは，典型的なSPARCのboot programについて示す．ただし，boot時には2次キャッシュの初期化等を含むために，プログラム自体は京都大学により開発された．

F.1　SPARCのreset後の動作について

reset後にSPARCはboot modeに入っている．この後，non-cacheableな`read_req`により，PA[35:0] = 0xFF0000000という番地からdouble wordだけ命令フェッチを試みる．残念ながらこのパケットはMBP-lightに対してクラスタバスからの割込みを起こす．従って，MBP-lightは適切な番地からデータを読み出し，`read_reply`を返す必要がある．なお，boot modeから抜けるには，MMU CoNTroL register(`MCNTL`)のBT (BooT mode) fieldをを0にしなければならない．

F.2　SPARC boot program

boot programは0番地である`start`から開始する．ただし，programは

```
label
        番地:  16進dump code       mnemonic code
```

と記すこととする．

F.2.1　reset処理及びprocessorの初期化の流れ

```
start()
           0:  30 80 00 70         ba,a    reset
```

delay slot無の分岐命令により，まず`reset`番地に飛ぶ．

```
reset()
        1c0:  11 00 00 00        sethi    %hi(_START_), %o0
        1c4:  90 12 23 00        or       %o0, MFSR, %o0  ! MFSR
        1c8:  d0 82 00 80        lda      [%o0] 0x04, %o0
        1cc:  13 00 00 80        sethi    %hi(0x20000), %o1
        1d0:  80 a2 00 09        cmp      %o0, %o1
        1d4:  06 80 00 0b        bl       .check_BIST_reset
        1d8:  01 00 00 00        nop
        1dc:  30 80 00 01        ba,a     .error_mode
```

MMU Fault Status Register(`MFSR`) の EM (Error Mode) field が 1 であるかどうかを調べている．これは Error Mode Reset が行われたかどうかを示している．通常の boot 時には 0 だと考えられる．よって`.check_BIST_reset` 番地のみ説明する．

```
.check_BIST_reset()
        200:  90 10 21 00        mov      256, %o0
        204:  d0 82 07 20        lda      [%o0] 0x39, %o0
        208:  80 a2 20 00        cmp      %o0, 0
        20c:  12 80 00 05        bne      .check_BIST_signature
        210:  01 00 00 00        nop
        214:  c0 a0 07 20        sta      %g0, [%g0] 0x39
        218:  00 00 00 00        unimp    0x0
        21c:  00 00 00 00        unimp    0x0
```

BIST (Buildt-In Self-Test) Diagnostic Register から 2 bit の BIST status 情報を読み出している．そして，何も BIST が走っていないのであれば (値が 0)，short BIST を起動する．そうでなければ`.check_BIST_signature` 番地へ飛ぶ．

```
.check_BIST_signature()
        220:  d0 80 07 20        lda      [%g0] 0x39, %o0
        224:  13 00 00 07        sethi    %hi(0x1c00), %o1
        228:  92 12 63 04        or       %o1, 0x304, %o1 ! 0x1f04
        22c:  30 80 00 05        ba,a     init
```

そして，31 bit の BIST signature を読み出して，`init` 番地へ飛んでいる．本来はこれを正しい期待値と比較して故障の有無や状態を診断すべきであるが，ここでは簡単化を図ったものと思われる．

```
init()
        240:  c0 a0 00 80        sta      %g0, [%g0] 0x04
        244:  81 d8 00 00        iflush   %g0
        248:  30 80 00 02        ba,a     .init_cregs
```

ここで，`MCNTL` の BT field に 0 を書込み，boot mode から抜ける．これ以降，MBP-light には cacheable な `read_req` が届くため，MBP Core が起動することはない．

```
.init_cregs()
        250:  11 00 44 00         sethi     %hi(trap_table), %o0
        254:  90 12 20 00         or        %o0, _START_, %o0
        258:  81 98 00 08         mov       %o0, %tbr
        25c:  11 00 00 00         sethi     %hi(_START_), %o0
        260:  90 12 20 01         or        %o0, INIT_WIM, %o0
        264:  81 90 00 08         mov       %o0, %wim
        268:  11 00 00 04         sethi     %hi(INV_SYNC), %o0
        26c:  90 12 20 a7         or        %o0, 0xa7, %o0  !
        270:  81 88 00 08         mov       %o0, %psr
        274:  40 00 01 8b         call      init_fpu_fsr
        278:  01 00 00 00         nop
        27c:  30 80 00 01         ba,a      .enable_superscalar_exec
```

まず，Trap Base Register (TBR) に 20 bit の trap table 用 base 番地を書き込む．また，Processor Status Register (PSR) のうち CWP (Current Window Pointer) field を 7 に，S (Supervisor) field を 1 に，EF (Enable FPU) を 1 に設定する．そして，Window Invalid Mask (WIM) には 1 を書き込む*．この後 init_fpu_fst 番地に飛ぶ．

```
init_fpu_fsr()
        820:  11 03 c0 00         sethi     %hi(0xf000000), %o0
        824:  13 00 00 07         sethi     %hi(0x1c00), %o1
        828:  92 12 63 40         or        %o1, 0x340, %o1
        82c:  d0 22 40 00         st        %o0, [%o1]
        830:  c1 0a 40 00         ld        [%o1], %fsr
        834:  81 c3 e0 08         jmp       %o7 + 8
        838:  01 00 00 00         nop
```

まず，0x1F40 番地に値 0xF000000 を書いている．この際に non-cacheable な write_req を MBP-light が処理する必要がある．そして，同一番地から値を読み出して，その値を Floting-point State Register (FSR) の TEM (Trap Exception Mask) field のうち

(1) NVM (iNValid operaion trap Mask)

(2) OFM (OverFlow trap Mask)

(3) UFM (UnderFlow trap Mask)

(4) DZM (Divide by Zero trap Mask)

を設定している．この後，0x278 番地に戻った後 .enable_superscalar_exec 番地に飛んでいる．

*SAVE 命令が発行される度に CWP はデクリメントされ，CWP が 1 になったときに trap が発生する．

```
.enable_superscalar_exec()
        280:  11 00 00 04        sethi    %hi(0x1000), %o0
        284:  d0 a0 09 80        sta      %o0, [%g0] 0x4c
        288:  11 04 00 00        sethi    %hi(CSADR0), %o0
        28c:  90 12 20 90        or       %o0, 0x90, %o0
        290:  92 10 20 03        mov      3, %o1
        294:  d2 aa 00 40        stba     %o1, [%o0] 0x02
        298:  11 04 00 00        sethi    %hi(CSADR0), %o0
        29c:  90 12 20 00        or       %o0, _START_, %o0
        2a0:  c0 aa 00 40        stba     %g0, [%o0] 0x02
        2a4:  11 04 00 00        sethi    %hi(CSADR0), %o0
        2a8:  90 12 20 08        or       %o0, 0x8, %o0   ! CSADR1
        2ac:  c0 aa 00 40        stba     %g0, [%o0] 0x02
        2b0:  11 04 00 00        sethi    %hi(CSADR0), %o0
        2b4:  90 12 20 10        or       %o0, 0x10, %o0  ! CSADR2
        2b8:  c0 aa 00 40        stba     %g0, [%o0] 0x02
        2bc:  11 04 00 00        sethi    %hi(CSADR0), %o0
        2c0:  90 12 20 18        or       %o0, 0x18, %o0  ! CSADR3
        2c4:  c0 aa 00 40        stba     %g0, [%o0] 0x02
        2c8:  11 24 00 00        sethi    %hi(0x90000000), %o0
        2cc:  90 12 21 58        or       %o0, 0x158, %o0
        2d0:  c0 aa 00 40        stba     %g0, [%o0] 0x02
        2d4:  30 80 00 03        ba,a     .init_mmu_and_caches
```

まず，Breakpoint ACTION Register の MIX (Multiple-Instruction-per-cycle eXecution mode) field を 1 に設定している．これにより，1 サイクルに複数の命令が実行可能になる．また，2 次キャッシュの内部資源である `L2CCTL` の WMQDP (Write Merge Queue DePth) filed に 3 を設定している．これが書込みデータをマージするためのキューの深さとなる．そして，2 次キャッシュの内部資源である `WMLMT` (Write Merge timeout LiMiT) の address に 0x10000000 に 0 を設定している．ライトバッファへの投入間隔が `WMLMT` を超えると，バッファ中のすべての書込み要求がバスへ送出される．この内部資源には CSADR (Control Space Access Data Register) を介して間接的に 0 を設定しなければならない．そのため，まず CSADR にすべて 0 を書き込んだ後，その内部資源に対応する番地に 0x10000000 を足した番地へ `stba` を行う必要がある†．

```
.init_mmu_and_caches()
        2e0:  40 00 01 80        call     init_mmu
        2e4:  01 00 00 00        nop
        2e8:  40 00 00 0e        call     init_icache
        2ec:  01 00 00 00        nop
        2f0:  40 00 00 1c        call     init_dcache
        2f4:  01 00 00 00        nop
        2f8:  7f ff ff 8a        call     init_ecache
        2fc:  01 00 00 00        nop
        300:  40 00 00 a0        call     set_stack
        304:  01 00 00 00        nop
```

†読込みの際は内部資源に対応する番地へ `ldba` を行った後に CSADR からデータを読むことになる．

ここから外部の module 及び stack に対する初期化を行う．以降，流れに従って順に説明する．

F.2.2　MMU の初期化の流れ

```
init_mmu()
        860:  81 e0 00 00         save
        864:  40 00 00 47         call      demap_all_TLB_entries
        868:  01 00 00 00         nop
```

まず，demap_all_TLB_entries 番地に飛ぶ．

```
demap_all_TLB_entries()
        980:  11 00 00 01         sethi     %hi(MFAR), %o0
        984:  90 12 20 00         or        %o0, _START_, %o0
        988:  c0 a2 00 60         sta       %g0, [%o0] 0x03
        98c:  81 c3 e0 08         jmp       %o7 + 8
        990:  01 00 00 00         nop
```

MMU の Probe and Demap Address Format により type field が 4 (Entire) となる address に書込みを行う．これにより，すべての PTE (Page Table Entry) が demap される．この後 0x868 番地に戻る．

```
        86c:  90 10 20 00         clr       %o0
        870:  40 00 00 34         call      set_MMU_MCTX
        874:  01 00 00 00         nop
```

%o0 を 0 に初期化して set_MMU_MCTX 番地に飛ぶ．

```
set_MMU_MCTX()
        9c0:  13 00 00 00         sethi    %hi(_START_), %o1
        9c4:  92 12 62 00         or       %o1, .check_BIST_reset, %o1
        9c8:  d0 a2 40 80         sta      %o0, [%o1] 0x04
        9cc:  81 c3 e0 08         jmp      %o7 + 8
        9d0:  01 00 00 00         nop
```

MMU ConTeXt register(MCTX) に 0 を設定する．これが現在のコンテキスト番号を示すことになる．そして，0x874 番地に戻る．

```
          878:  90 10 20 00         clr       %o0
          87c:  92 10 20 00         clr       %o1
          880:  15 3c 00 00         sethi     %hi(BMP_START), %o2
          884:  94 12 a0 fc         or        %o2, 0xfc, %o2
          888:  96 10 20 00         clr       %o3
          88c:  98 10 20 00         clr       %o4
          890:  40 00 00 44         call      set_TLB_entry
          894:  01 00 00 00         nop
```

%o0，%o1，%o3 と%o4 に 0 を，%o2 に 0xF00000FC を代入する．それから set_TLB_entry 番地へ飛ぶ．

```
set_TLB_entry()
          9a0:  99 2b 20 0c         sll       %o4, 12, %o4
          9a4:  d0 a3 00 c0         sta       %o0, [%o4] 0x06
          9a8:  98 03 21 00         add       %o4, 256, %o4
          9ac:  d2 a3 00 c0         sta       %o1, [%o4] 0x06
          9b0:  98 03 21 00         add       %o4, 256, %o4
          9b4:  d4 a3 00 c0         sta       %o2, [%o4] 0x06
          9b8:  98 03 21 00         add       %o4, 256, %o4
          9bc:  d6 a3 00 c0         sta       %o3, [%o4] 0x06
          9c0:  81 c3 e0 08         jmp       %o7 + 8
          9c4:  01 00 00 00         nop
```

MMU TLB Diagnostic Access Address により%o4 を TLBE (TLB Entry) field に設定する．そのために，12 bit だけ%o4 を最初に左シフトしている．さらに，このときに SEL (SELect) field が 0 であるために，TLB Virtual Page Number が%o0 に設定される．また，次の参照では SEL field が 1 であり，TLB Entry Context Number が%o1 に設定される．続いての参照では SEL が 2 であり，TLB Entry Physical Page Number が%o2 に設定される．そして，最後の参照により TLB Entry Lock Bit が%o3 になる．以上で述べた設定を列挙する．

```
TLB[0].VPN (Virtual Page Number)  = 0x00000
TLB[0].CTX (ConTeXt)              = 0x0000
TLB[0].PPN (Physical Page Number) = 0xF00000
TLB[0].Cacheable.Modified.Valid
TLB[0].ACC (Access permissions)   = Read/Write/Execute for Supervisor
TLB[0].LVL (LeVeL)                = 4 GB
TLB[0].LOCK                       = Unlocked
```

PPN field の先頭が 0xF であることから，memory command によると cache の protocol として更新型が選ばれたことになる[‡]．本来，ほかにも TLB entry を設定すべきであるが，ここではこの entry のみしか登録していない．これも簡単化を図ったためと考えられる．これで MMU の初期化は終了となり，次の命令キャッシュの初期化を行う．

[‡]0xA であれば無効化型となる．

F.2.3　命令キャッシュの初期化

```
init_icache()
        320:  c0 a0 06 c0        sta       %g0, [%g0] 0x36
        324:  11 02 00 00        sethi     %hi(0x8000000), %o0
        328:  c0 a2 06 c0        sta       %g0, [%o0] 0x36
        32c:  81 c3 e0 08        jmp       %o7 + 8
        330:  01 00 00 00        nop
```

まず，最初の参照は Address Cache Flash Clear Address Format により type field が 1 であるため，すべての Ptag 及び Stag 内の Valid と MRU (Most Recently Used)field がクリアされる．また，次の参照ですべての Stag の lock bit がクリアされる．これで命令 cache の初期化は終了となり，次のデータキャッシュの初期化を行う．

F.2.4　データキャッシュの初期化

```
init_dcache()
        360:  c0 a0 06 e0        sta       %g0, [%g0] 0x37
        364:  11 20 00 00        sethi     %hi(0x80000000), %o0
        368:  c0 a2 06 e0        sta       %g0, [%o0] 0x37
        36c:  81 c3 e0 08        jmp       %o7 + 8
        370:  01 00 00 00        nop
```

命令 cache の初期化と同様の処理を Data Cache Flash Clear Address Format に従ってデータキャッシュに行う．これでデータキャッシュの初期化は終了となり，次の 2 次キャッシュの初期化を行う．

F.2.5　2 次キャッシュの初期化

```
init_ecache()
        120:  81 e0 00 00         save
        124:  11 04 00 00         sethi     %hi(CSADR0), %o0
        128:  90 12 20 00         or        %o0, _START_, %o0 ! CSADR0
        12c:  c0 aa 00 40         stba      %g0, [%o0] 0x02
        130:  11 04 00 00         sethi     %hi(CSADR0), %o0
        134:  90 12 20 08         or        %o0, 0x8, %o0   ! CSADR1
        138:  c0 aa 00 40         stba      %g0, [%o0] 0x02
        13c:  11 04 00 00         sethi     %hi(CSADR0), %o0
        140:  90 12 20 10         or        %o0, 0x10, %o0  ! CSADR2
        144:  c0 aa 00 40         stba      %g0, [%o0] 0x02
        148:  11 04 00 00         sethi     %hi(CSADR0), %o0
        14c:  90 12 20 18         or        %o0, 0x18, %o0  ! CSADR3
        150:  c0 aa 00 40         stba      %g0, [%o0] 0x02
        154:  11 2c 00 00         sethi     %hi(FMP_START), %o0
        158:  90 12 20 00         or        %o0, _START_, %o0
        15c:  13 3c 00 00         sethi     %hi(BMP_START), %o1
        160:  92 12 60 00         or        %o1, _START_, %o1
        164:  94 02 21 40         add       %o0, 320, %o2
        168:  c0 aa 00 40         stba      %g0, [%o0] 0x02
        16c:  c0 aa 40 40         stba      %g0, [%o1] 0x02
        170:  90 02 20 20         add       %o0, 32, %o0
        174:  92 02 60 20         add       %o1, 32, %o1
        178:  c0 aa 00 40         stba      %g0, [%o0] 0x02
        17c:  c0 aa 40 40         stba      %g0, [%o1] 0x02
        180:  80 a2 00 0a         cmp       %o0, %o2
        184:  06 bf ff fb         bl        0x170
        188:  01 00 00 00         nop
        18c:  11 2c 00 00         sethi     %hi(FMP_START), %o0
        190:  90 12 20 00         or        %o0, _START_, %o0
        194:  90 02 21 40         add       %o0, 320, %o0
        198:  13 3c 00 00         sethi     %hi(BMP_START), %o1
        19c:  92 12 60 00         or        %o1, _START_, %o1
        1a0:  92 02 61 40         add       %o1, 320, %o1
        1a4:  15 2c 03 ff         sethi     %hi(0xb00ffc00), %o2
        1a8:  94 12 a3 e0         or        %o2, 0x3e0, %o2
        1ac:  17 00 00 11         sethi     %hi((MAT+9368)), %o3
        1b0:  96 12 e3 81         or        %o3, 0x381, %o3
        1b4:  30 bf ff bb         ba,a      init_ecache_kernel
```

2 次 cache の内部資源である FMP と BMP を 0 に初期化している．さきほど述べた WMLMT と同様に CSADR を用いて間接的に書き込む必要がある．C 言語風に書くと以下のようになる．

```
/* 0x10000000 for store is added to each address */
FMP = 0xB00000000; BMP = 0xF00000000;
for(i = 0; i<=0x140; i += 32)
    FMP[i] = BMP[i] = 0;
```

そして，%o0 に 0xA0000140 を，%o1 に 0xF0000140 を，%o2 に 0xB00FFFE0 を，%o3 に 0x4781 を代入して `init_ecache_kernel` 番地に飛ぶ．

```
init_ecache_kernel()
        a0:  d6 a0 00 80        sta       %o3, [%g0] 0x04
        a4:  81 d8 00 00        iflush    %g0
        a8:  90 02 20 20        add       %o0, 32, %o0
        ac:  92 02 60 20        add       %o1, 32, %o1
        b0:  c0 aa 00 40        stba      %g0, [%o0] 0x02
        b4:  c0 aa 40 40        stba      %g0, [%o1] 0x02
        b8:  80 a2 00 0a        cmp       %o0, %o2
        bc:  06 bf ff fb        bl        0xa8
        c0:  01 00 00 00        nop
        c4:  11 00 00 00        sethi     %hi(_START_), %o0
        c8:  90 12 20 20        or        %o0, BMP_STEP, %o0
        cc:  13 00 00 04        sethi     %hi(INV_SYNC), %o1
        d0:  92 12 60 50        or        %o1, 0x50, %o1  ! 0x1050
        d4:  d4 8a 40 40        lduba     [%o1] 0x02, %o2
        d8:  95 2a a0 05        sll       %o2, 5, %o2
        dc:  90 02 00 0a        add       %o0, %o2, %o0
        e0:  92 10 20 01        mov       1, %o1
        e4:  11 00 00 00        sethi     %hi(_START_), %o0
        e8:  90 12 20 20        or        %o0, BMP_STEP, %o0
        ec:  92 10 20 01        mov       1, %o1
        f0:  11 00 00 00        sethi     %hi(_START_), %o0
        f4:  90 12 20 20        or        %o0, BMP_STEP, %o0
        f8:  92 02 20 20        add       %o0, 32, %o1
        fc:  81 c7 e0 08        ret
       100:  81 e8 00 00        restore
```

まず，MCNTL の SE (Snoop Enable)，SB (Store Buffer enable)，IE (Instruction cache Enable)，DE (Data cache Enable)，PSO と EN (mmu ENable) を 1 に初期化する．ストアバッファをクリアした後に，2 次キャッシュの内部資源である FMP と BMP のさきほどの残りすべてを 0 に初期化している．残りはデバッグ用コードで，ここでは意味をなしていない．これで 2 次キャッシュの初期化は終了となり，次のスタックの初期化を行う．

F.2.6　スタックの初期化

```
set_stack()
       580:  13 00 00 04        sethi     %hi(INV_SYNC), %o1
       584:  92 12 60 50        or        %o1, 0x50, %o1  ! 0x1050
       588:  d0 8a 40 40        lduba     [%o1] 0x02, %o0
       58c:  91 2a 20 0a        sll       %o0, 10, %o0
       590:  13 00 00 04        sethi     %hi(INV_SYNC), %o1
       594:  92 12 63 00        or        %o1, MFSR, %o1  ! STACK
       598:  92 02 40 08        add       %o1, %o0, %o1
       59c:  9c 10 00 09        mov       %o1, %sp
       5a0:  bc 10 00 09        mov       %o1, %fp
       5a4:  81 c3 e0 08        jmp       %o7 + 8
       5a8:  01 00 00 00        nop
```

まず，2 次キャッシュの内部資源である PEID を読み出す．この値を peid とし，スタック

の先頭番地を stack_adr とする．これを C 言語風に書くと以下のようになる．

```
%sp /* stack pointer */ =
%fp /* frame pointer */ = (peid << 10 + stack_adr);
```

ただし，この 10 という数字は任意にとってよい．

F.3　まとめ

以上に述べた SuperSPARC+の各内部レジスタの詳細は文献 [78] を参照されたい．JUMP-1 に特有な初期化部分は 2 次キャッシュの内部資源を参照している部分であり，他は通常のシーケンスとほとんど変わらない．

付 録G　MBIFのコマンド

MBIF のコマンドには，MBP-light から送られるものと MC から送られるものが存在する．ここではこれらのコマンドの概要を示す．MBP-light から送られるコマンドを表 G.1 に，MC から送られるものを表 G.2，表 G.2 にそれぞれ示す．

表 G.1 MBIF のコマンド (MBP 側)

コマンド名	ビットパターン	概要
割り込みの不許可	1110 0000	MBIF から MBP-light への割り込みを禁止する (ポーリングモード).
割り込みの許可	1110 0001	MBIF から MBP-light への割り込みを許可する (割り込みモード).
拡張用	1110 0010	未使用.
拡張用	1110 0011	未使用.
初期化モードへの移行	1111 1110	MBP-light をリセットして初期化モードに移行する.

表 G.2 MBIF のコマンド (MC 側，初期化モード)

コマンド名	ビットパターン	概要
アドレスレジスタクリア	0000 0000	ローカルバス用のアドレスレジスタ (LAR) をクリア (all 0) する．
アドレスレジスタセット	0000 0001	ローカルバス用のアドレスレジスタ (LAR) を任意の値にセットする．セットするデータは下位バイト，上位バイトの順にコマンドに引き続き 2 Byte 転送する．
アドレスレジスタ+1	0000 0010	ローカルバス用のアドレスレジスタ (LAR) を +1 する．
データレジスタクリア 1	0000 0100	ローカルバス用のデータレジスタ (LDR) をクリア (all 0) する．
データレジスタクリア 2	0000 0101	ローカルバス用のデータレジスタ (LDR) をクリア (all 0) する．また，LSR の LDR enable を有効にする．
データレジスタセット 1	0000 0110	ローカルバス用のデータレジスタ (LDR) を任意の値にセットする．セットするデータは下位バイト，中位バイト，上位バイトの順にコマンドに引き続き 3 Byte 転送する．なお，上位バイトは下位 5 bit のみが有効 (データは 21 bit であるため)．
データレジスタセット 2	0000 0111	ローカルバス用のデータレジスタ (LDR) を任意の値にセットする．セットするデータは下位バイト，中位バイト，上位バイトの順にコマンドに引き続き 3 Byte 転送する．なお，上位バイトは下位 5 bit のみが有効 (データは 21 bit であるため)．また，LSR の LDR enable を有効にする．
ローカルメモリライト	0000 1000	LAR が保持するアドレスのローカルメモリに，LDR の内容を書き込む．
ローカルメモリライト+1	0000 1001	LAR が保持するアドレスのローカルメモリに，LDR の内容を書き込む．その後，LAR を +1 する．
MSR のリード	0100 0000	MSR の内容を PC 側に転送する．
MDRO のリード	0100 0001	MDRO の内容を PC 側に転送する．
メインテナンスモードへの移行	0111 11110	MBP-light のリセットを解除して，メンテナンスモードに移行する．

表 G.3 MBIF のコマンド (MC 側，メインテナンスモード)

コマンド名	ビットパターン	概要
データレジスタクリア 1	1000 0100	ローカルバス用のデータレジスタ (LDR) をクリア (all 0) する．
データレジスタクリア 2	1000 0101	ローカルバス用のデータレジスタ (LDR) をクリア (all 0) する．また，LSR の LDR enable を有効にする．
データレジスタセット 1	1000 0110	ローカルバス用のデータレジスタ (LDR) を任意の値にセットする．セットするデータは下位バイト，中位バイト，上位バイトの順にコマンドに引き続き 3 Byte 転送する．なお，上位バイトは下位 5 bit のみが有効 (データは 21 bit であるため)．
データレジスタセット 2	1000 0111	ローカルバス用のデータレジスタ (LDR) を任意の値にセットする．セットするデータは下位バイト，中位バイト，上位バイトの順にコマンドに引き続き 3 Byte 転送する．なお，上位バイトは下位 5 bit のみが有効 (データは 21 bit であるため)．また，LSR の LDR enable を有効にする．
MSR のリード	1100 0000	MSR の内容を，PC 側に転送する．
MDRO のリード	1100 0001	MDRO の内容を，PC 側に転送する．
MBP-light への割り込み要求	1110 0010	MBP-light に割り込みを要求する．
MBP-light への割り込み要求	1110 0011	MBP-light への割り込み要求を解除する．
初期化モードへの移行	1111 1110	MBP-light をリセットして初期化モードへ移行する．

付 録H　MBIFを用いたFile System

MBP-lightはMBIFを通じてホストであるPC上にファイルシステムをもつことが可能である．そのために，openやread，writeといった標準関数が提供されている [79]．よって，ホスト上のPCとのprotocolや標準関数の使用法について説明する．

H.1　File System用protocol

では，PCとの間に必要となるprotocolを順次説明していく．

(1) open

(2) read

(3) write

(4) close

(5) fflush

(6) maintenance

H.1.1　open

openは通常のUNIXで提供されているopen(2)と同様の動作を行う．ただし，' で囲まれていた場合，その文字のASCIIコードを送ることを意味する．なお，NULL文字は\0とする．また，添えられている()内の数字はそのバイト数を表す．

形式: 'o'(1), filename(∞), '\0'(1), flags(2) , mode(2)

応答: 成功したら1 Byteのファイル記述子を得る．ただし，それが0であったならば失敗である．

ここで，flagsは次の意味をもつ．

O_RDONLY	→	0x000 ,	O_WRONLY	→	0x001
O_RDWR	→	0x002 ,	O_APPEND	→	0x008
O_CREAT	→	0x200 ,	O_TRUNC	→	0x400
O_EXCL	→	0x800 ,			

また，modeはPC上にできるファイルのpermissionを指定する．

H.1.2 `read`

`read` は通常の UNIX で提供されている `read(2)` と同様の動作を行う．

形式: ’r’(1), ファイル記述子 (1), バイト数 (2)

応答: 成功したら，バイト数とそのデータを 2 Byte 単位で返す．ただし，バイト数が 0 ならば EOF か失敗である．

H.1.3 `write`

`write` は通常の UNIX で提供されている `write(2)` と同様の動作を行う．

形式: ’w’(1), ファイル記述子 (1), byte 数 (2), data(–)

応答: 無し

処理の簡単化のために応答は無いものとした．

H.1.4 `close`

`close` は通常の UNIX で提供されている `close(2)` と同様の動作を行う．

形式: ’c’(1), ファイル記述子 (1)

応答: 無し

H.1.5 `fflush`

`fflush` は通常の UNIX で提供されている `fflush(3)` と同様の動作を行う．

形式: ’f’(1), ファイル記述子 (1)

応答: 無し

H.1.6 `maintenance`

これは特殊な命令であり，MBP-light が MBIF の所有権を放棄することをホストの PC 側に伝えるものである．というのも，MBIF が PC か MBP-light のどちらか一方からのデータ転送しかサポートしていないからである．もし MBP-light が MBIF の所有権を持っている状態で，MBIF を解放したいならば，明示的にこの system call を発行する必要がある．これにより，MBIF を通じた PC 側からの maintenance 作業を行うことが可能となる．なお，初期時には PC にその所有権がある．

形式: ’m’(1)

応答: 無し

H.2　標準関数の使用法

前節で述べたプロトコルを利用して，m4 macro で標準関数を実装した．以下に，MBP-light 用プログラムでの使用法を説明する．

(1) OPEN

(2) READ2B

(3) WRITE2B

(4) WRITE32B

(5) CLOSE

(6) FFLUSH

(7) MAINTENANCE

H.2.1　OPEN

OPEN はファイル記述子を得るために用いることができる．ここで，以降 gpr_t 型は general purpose register を，pbr_t は packet buffer register であることを意味する．

形式: OPEN(gpr_t fd, char *filename, int *flags, char *mode)

応答: open protocol で得られたファイル記述子が fd に格納される

なお，mode は permission 情報を 8 進数で表したときの文字列を渡さなければならない．以下に例を示す．

```
OPEN(‘r1’, ‘norio.dat’, ‘O_RDWR | O_CREAT’, ‘700’)
```

この例では，PC 上に permission が 700 である “norio.dat” という読み書き可能なファイルを open し，そのファイル記述子を r1 に格納する．

H.2.2　READ2B

READ2B は指定されたファイル記述子から 2 Byte のデータを読むことができる．

形式: READ2B(gpr_t fd, gpr_t data)

応答: fd に格納されているファイル記述子から read protocol を用いて 2 Byte のデータを読み，data に格納する．

以下に例を示す．

```
READ2B('r1', 'r2')
```

この例では，r1 で指定されたファイル記述子からデータを r2 に格納する．

H.2.3　WRITE2B

WRITE2B は指定されたファイル記述子に 2 Byte のデータを書くことができる．

形式: WRITE2B(gpr_t fd, gpr_t data)

応答: fd に格納されているファイル記述子へ write protocol を用いて，data に格納されている 2 Byte のデータを書く．

以下に例を示す．

```
WRITE2B('r1', 'r2')
```

この例では，r1 で指定されたファイル記述子へ r2 のデータを書く．

H.2.4　WRITE32B

WRITE32B は指定されたファイル記述子に 32 Byte のデータを書くことができる．

形式: WRITE32B(gpr_t fd, pbr_t data)

応答: fd に格納されているファイル記述子へ write protocol を用いて，data に格納されている 32 Byte のデータを書く．

以下に例を示す．

```
WRITE32B('r1', 'r2')
```

この例では，r1 で指定されたファイル記述子へ r2 という PBR のデータを書く．

H.2.5　CLOSE

CLOSE はファイル記述子を閉じるために用いる．

形式: CLOSE(gpr_t fd)

応答: fd というファイル記述子を close protocol で閉じる.

以下に例を示す.

```
CLOSE('r1')
```

この例では，r1 で指定されたファイル記述子を閉じる.

H.2.6 FFLUSH

FFLUSH はファイル記述子をフラッシュするために用いる.

形式: FFLUSH(gpr_t fd)

応答: fd というファイル記述子を fflush protocol でフラッシュする.

以下に例を示す.

```
FFLUSH('r1')
```

この例では，r1 で指定されたファイル記述子をフラッシュする.

H.2.7 MAINTENANCE

MAINTENANCE は maintenance protocol そのものである.

形式: MAINTENANCE

応答: 無し

以下に例を示す.

```
MAINTENANCE
```

付 録I　メインテナンスシステムでの使用チップ

ここでは，メインテナンスシステムで利用したチップの概要を示す．

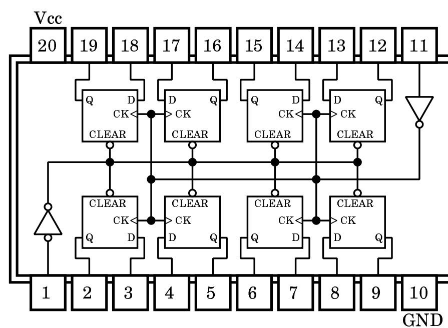


図 I.1 74LS273(Octal D-Type Flip-Flops with Clear)

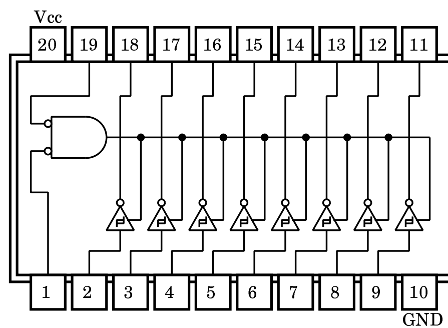


図 I.2 74LS540(Octal Buffers And Line Drivers with 3-State Output)

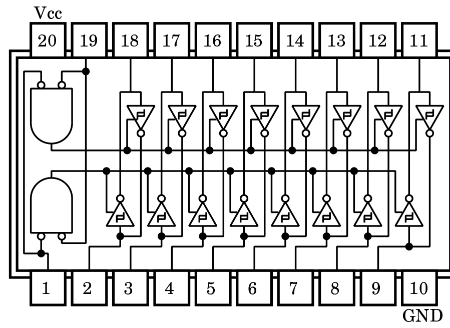


図 I.3 74LS642(Octal Bus Transceivers)

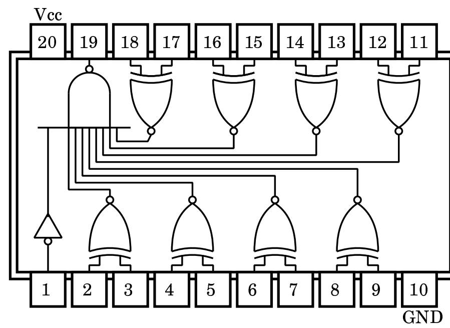


図 I.4 74LS688(8-Bit Magnitude Comparator)

付 録J　汎用並列計算機monitor system –Pot–の使用法

慶應義塾大学の美辺氏によって実装された汎用並列計算機 monitor system –Pot–について説明する．ただし，ここでは MBP-light 版に用意されている命令に限定して説明する．詳細は文献 [80]，[79] を参照されたい．

用意されている命令を以下に示す．ただし，以降の *length* とは 3 Byte を 1 単位とする．

- `cd`，`chdir`

  構文: `cd` *dir*, `chdir` *dir*

  意味: 現在のディレクトリから *dir* へ移動．

- `debug`

  構文: `debug` *level*

  意味: *level* に応じたデバッグ情報を見る．

- `echo`

  構文: `echo` [*string* ...]

  意味: echo packet を送信し，その応答 (*string* ...) を表示する．

- `exit`，`q`，`quit`

  構文: `exit`，`q`，`quit`

  意味: Pot を終了する．

- `go`

  構文: `go`

  意味: リセットを解除する．

- `help`

  構文: `help` [*command* ...]

  意味: *command* に対して簡易版のヘルプを見ることができる．指定されなかったときは全ての命令を対象とする．

- `load`

  構文: `load file` [*address*]

  意味: `file` の内容を *address* 番地からメモリに書きこむ．*address* が指定されない場合は 0 番地からとする．

- `number`

  構文: `number` [*num*]

  意味: 対象とするクラスタを *num* に切り換える．*num* が指定されない場合は現在対象としているクラスタを表示する．

- `passive`

  構文: `passive`

  意味: MBIF の所有権を放棄し，MBP-light からのデータ転送を受け付ける．MBP-light が `maintenance` protocol を発行するまでこの状態を保持する．

- `pwd`

  構文: `pwd`

  意味: 現在のディレクトリを表示する．

- `rawsend`

  構文: `rawsend` *string*

  意味: *string* を生のまま送る．

- `recvmbp`

  構文: `recvmbp` *length*

  意味: MBP-light から *length* だけデータを受け取る．

- `reset`

  構文: `reset`

  意味: リセットを行う．

- `run`

  構文: `run` *file*

  意味: *file* を `load` し `go` する．

- `save`

  構文: `save` *file address length*

  意味: メモリの *address* 番地から *length* 分だけ *file* に書き込む.

- `sendmbp`

  構文: `sendmbp`

  意味: 標準入力からのデータを MBP-light へ転送する.

- `sendmbprs`

  構文: `sendmbprs` *length*

  意味: 標準入力からのデータを *length* 分だけ RS-232C 用のフォーマットで MBP-light へ転送する．まず，転送 word 数 (*length*) を `[0-9A-F]` という 4 文字で送信する．次に，1 Byte のデータを `[0-9A-F]` という 2 文字で表現し，転送 word 数だけ送信を行う.

- `version`

  構文: `version`

  意味: 現在の Pot の version 番号を表示する.

- `write`

  構文: `write` *address data* [*data* ...]

  意味: メモリの *address* 番地から *data* を 1 つずつ書き込む.

- `interrupt`

  構文: `interrupt`

  意味: MBP-light に割込みをかける.

付 録K　MBP-light用Cコンパイラ

ここでは，DSM管理プログラム開発に使用したMBP-lightを対象にしたCコンパイラの実装について述べる．Cコンパイラの実装は京都大学によって行われ，GNU C Compiler(GCC)を移植する方法により実装されている．

K.1　GCCの特徴

GCCでは中間言語として，Register Transfer Language(RTL)というリスト形式の言語を用い，最適化等の処理を行う．

GCCのMBP-lightへの移植は，次の2つのファイルを記述することで行う．

Target Description Macro 対象となるプロセッサのハードウェア構成について記述する他，汎用レジスタの使い方，スタックの構成法，アセンブリファイルの出力の方法などの設定をする．

caller saveレジスタの設定や，引数の格納の規則を設定することができる．

Machene Description File(MDファイル) 使用する命令を，その命令の行う操作を表すRTLと一対一に対応させて記述する．

MDファイルの記述においては，標準型と呼ばれる操作に関しての記述を行う必要がある．標準型の操作は，値の格納，比較，条件分岐，関数呼出しなどであり，基本的な操作である．標準型は，構文解析の結果からRTLを生成する段階で使われる．標準型を表すRTLと命令の対応に，決められた名前を付けることで設定できる．

標準型を表す操作が複数命令でないと表現できない場合は，それぞれの命令に対応する複数のRTLの生成を行うことも可能である．

RTLから命令列を出力する段階では，すべての対応が使われ一対一でアセンブリ命令に変換される．

またMDファイルにはdelay slotの設定も可能となっている．

この2つのファイルはコンパイルの処理中に常に参照される．

K.2　GCCとMBP-lightとの不親和性

MBP-lightはメモリコントローラとして開発された特殊なプロセッサであり，GCCを移植するには次のような問題点がある．

複数のメモリに対するアクセス MBP Core からアクセスすることのできるメモリはローカルメモリ，内部メモリである．また PBR も汎用レジスタ間接でアクセスすることから，プログラムを記述する段階ではメモリと考えられる．

この 3 つは，アドレス領域によって区別されるのではなく，命令によって区別される．したがって，これらのメモリアクセスの RTL 表現において，アドレスを示すレジスタがどのメモリに対するものなのかは区別できなくなり，ローカルメモリアクセスの命令しか出力できない．

MBP-light 特有の命令 MMC 制御命令，RDT 制御命令，特殊命令などは，直に C 言語で表現することは困難である．同様に RTL でその操作を表現することも困難である．

このような問題に対して，今回は基本 RISC 命令の部分に関して GCC の移植を行い，それ以外の命令は単純にソースオペランドを引数とする関数呼出しで表現して，コンパイラによって対応した命令を出力させることで対応した．

K.3　C コンパイラの実装

前節で述べたように，MBP-light 用 C コンパイラでは基本 RISC 命令以外の命令は関数呼出しで表現し，コンパイラによって対応した命令を出力させるという方法で実装されている．

MD ファイルにおいて，RTL と命令を一対一で対応させることで，RTL からアセンブリ命令への変換が行われるが，RTL に含まれる情報によって出力される命令を変えることが可能である．関数呼出しを表現する RTL には，その情報として関数名が含まれている．したがって，特定の名前をもつ関数呼出しの RTL を，対応した命令に変換させることが可能である．

以降，特定の命令を C 言語で記述するために使う関数を,「特別な関数」と記述する．しかし基本 RISC 命令以外の命令に関して，関数呼出しで表現することによって起こる問題がある．以下，その問題点とその解決法について述べる．

オペランドの最適なレジスタ割当て 関数呼出しでは，引数及びその返り値は関数間のインタフェースをとるため，定められた規則でレジスタに格納される．そのため関数呼出しで表現される命令は特定のレジスタしかオペランドとして使えないことになる．

引数及び返り値を格納するレジスタに関しては，Target Description Macro で設定する．この Macro では，返り値のレジスタに関しては関数の名前によって設定を変更することが可能であったため，最適なレジスタを割り付けることができた．しかし，引数を格納するレジスタは，Macro では関数の名前によって変更することが不可能であった．

関数呼出しに当たって，構文解析後生成される RTL は図 K.1 のようになる．この図は RTL を簡略化したものであり，実際はもう少し複雑である．例は，引数 2 個の関数の場合であり，引数はレジスタ 2・3 に格納する，と設定してあるものとする．

ここで，argument と書いてある部分には直値，もしくはレジスタが当てはまる．

```
(1) (set (register2)(argument1))  ←  引数1をレジスタ2に格納
(2) (set (register3)(argument2))  ←  引数2をレジスタ3に格納
(3) ((call ("function")           ←  レジスタ2、3を引数として
        (use (register2))              関数functionを呼ぶ
          (use(register3)))
```

図 K.1 関数呼出しの RTL

レジスタ 2・3 の部分が最適化によって変わることはない．

これが特別な関数の場合，argument にレジスタが当てはまるなら，そのレジスタをそのままオペランドとして扱えるにも関わらず，引数用レジスタに値を格納してから，関数に対応した命令が出力されることになる．この格納命令は無駄である．

また，引数用レジスタの値を，当該命令の前後に渡って使いたいとき，その値の退避・復帰を行う命令を出力することになり，この操作も無駄である．

しかし，図 K.1 における (1)(2) の RTL 表現は，(3) の表現を MD ファイルの関数呼出しの標準型の記述によって出力しようとするときには，変更することができない．

そこで，特別な関数に関しては構文解析後，図 K.2 のような RTL が生成されるように MD ファイルを記述した．

ここで，pseudo register と書いてある部分は最適化の段階で適当なレジスタに置換される．

すなわち，図 K.1 における (3) の表現として，レジスタ 2・3 を別のレジスタに格納してから (点線部)，そのレジスタを引数として関数呼出しをする，という RTL を生成するように標準型の記述がされている．

このようにしておけば最適化の段階でレジスタ 2・3 への格納命令は無駄であるとして省かれ，pseudo register に最適なレジスタが割り当てられる．場合によっては点線部の格納命令も省かれる．この方法により関数呼出しで表現される命令のソースオペランドに対しても最適なレジスタが割り当てられることになる．

直値オペランドの問題 基本 RISC 命令以外の命令において，オペランドに直値を必要とするものに次のようなものがある．

- PBR のアドレスのオフセット
- 内部メモリアクセス命令のオフセット
- PBR-直値間命令

C 言語で記述する際，これらの命令は関数呼出しで表現することにしたが，その引数として直値で出力したい部分を指定してしまうと，その RTL 表現は図 K.2 のように直値で表現したい部分も pseudo register として表現される．

```
(set (register2)(argument1))
(set (register3)(argument2))

(set (pseudo register1)(register2))
(set (pseudo register2)(register3))

((call ("special_function")
  (use (pseudo register1))
    (use(pseudo register2)))
```

図 K.2 特別な関数呼出しの RTL

pseudo register の部分には，直値も許容する記述が可能であるが，レジスタを禁止することはできないために，アセンブリに変換するときにはこの部分の RTL 表現がどちらになるかわからない．

pseudo register の部分が，直値でもレジスタでも対応する命令が存在する場合には間違いのない命令が出力される．PBR-直値命令においては一部そのような命令が存在したため，それについてはサポートできた．PBR-直値命令においては，実際に使われる命令は限られているが，サポートできた命令で十分である．

しかし PBR，内部メモリのオフセットに関しては，RTL 表現でレジスタが現れると出力する命令が存在しない．

そこで PBR に関しては，そのオフセットに使える数が 0 から 8 までであることから，PBR の命令 1 個につき，9 種類の関数名を用意して，それによってオフセットの部分を出力するという方法をとっている．

内部メモリに関しては，そのアクセス用途は限られており，よってアドレスもプログラマに自明なものばかりであるので，オフセットは常に 0 としている．

MBP-light の命令列特有の制限事項 MBP-light では，命令列には以下のような制限事項がある．

(1) PBR に格納した値は 1 スロットの間使用できない．

(2) クラスタメモリアクセスに使用した PBR 及びその PBR を指す GPR は，そのアクセスの結果が反映されるまで使用することができない．

MD ファイルでは，ある命令によって格納された値は何スロットの間アクセスすることができない，という設定を行うことができる．

しかし (1) に関して，MD ファイルで設定することはできない．その理由は，PBR 命令は関数呼出しで表現しているため，コンパイラはその命令が PBR に値を格納する操作である，という判断ができないためである．この問題に関しては，コンパイ

ラは間違った命令列を出力する可能性がある．したがって，今回は出力された命令列を一度走査して，問題となる部分には，間に意味のない命令，NOP を挿入することで対処した．

(2) に関しても，クラスタアクセス命令を関数呼出しで表現しているため，同じ理由で MD ファイルで設定することはできない．しかしこの場合は，C 言語でプログラムを記述する段階で，メモリバリア命令を出力する関数をクラスタメモリアクセス命令の直後に呼ぶことにより，間違った命令を出力しないようにする．

その他の問題 その他に問題となったことは，DSM 管理プログラムにおいては，プロセス管理を行う必要があることである．プロセス管理においては，スタックポインタの切替えやコンテキスト切替え時の全レジスタの保存などの処理を行う．

これらの処理も C 言語では直には表現できない処理である．

今回は，これらも関数呼出しで表現し，特定の命令列を出力させることで対応している．

以下，MBP-light 用 C コンパイラの特徴について挙げる．

- 基本 RISC 命令以外の命令は関数呼出しで行う．これらはほぼアセンブリ命令と一対一に対応している．PBR-直値間命令はサポートしていないものもある．
- その他，C 言語では表現が困難な動作，例えばプロセス管理におけるコンテキスト切替えにおける全レジスタ保存の処理等も関数呼出しにして，特定の命令列を出力させる．
- inline 定義により，関数のインライン展開を行うことができ，関数呼出しのオーバヘッドをなくすことができる．プログラマは，インライン展開をコンパイラ任せにし，このことを意識せずにプログラムを開発することが可能である．
- クラスタメモリアクセスを行った場合は，直後にメモリバリア命令を表現する関数を呼ばなければならない．

  またこのことにより，クラスタメモリアクセスのレイテンシを隠蔽することが不可能になっている．