MBP-light マニュアル

1997.7.5 慶應 天野、安生、井上が MBP-light の構成案を基に作成
1997.7.8 慶應 天野、佐藤さんによる MMC の改造にインタフェースを合わせる
1997.7.11 慶應 安生，RDTi/f コマンドに parity enable を，Net cache tag に packet discard を付加
1997.7.15 慶應 天野，タイマと割り込みの部分を付加
1997.8.15 慶應 天野，MMC のアドレスマップを変更
1997.8.16 慶應 井上，安生，MMC のアドレスマップの図の修正，命令の誤字修正
1997.8.20 慶應 安生，SSR 命令の GPRs フィールドの訂正
1997.8.21 慶應 天野 LUDR, SUDR 命令の追加
1997.8.28 慶應 井上、誤字及び、内部レジスタアドレスマップ修正
1997.11.10 慶應 安生、MEMBAR の付加, LIMP format の変更
1997.12.2 慶應 井上、BIT 系命令の削除、WWE の変更、割り込みの修正
1997.12.5 慶應 井上、MEMBAR について補足
1997.12.12 慶應 安生、GRDT の記述の訂正
1997.12.17 慶應 安生、PBP, TJ の記述の変更
1997.12.26 慶應 井上、BITS, BITR の記述復刻

1 ユーザマニュアル

MBP-Core の基本的な骨格は、命令 21bit, データ 16bit の RISC プロセッサである。最初に、普通の RISC としての見え方を解説する。

1.1 16bit RISC としての MBP Core

16bit 幅の GPR を 16 個と、アドレス空間、I/O 空間がそれぞれ 16bit のワード (データ 16bit, 命令 21bit) アドレッシングのローカルメモリを持つ若干変わった RISC である。データ / アドレスの表現形式は、JUMP-1 全体の方式に従い、SPARC Architecture に基づく Big Endian である。基本 RISC としては、16bit しか扱わないので、単に Big Endian で MSB が 15bit 目である。Local Memory は恐ろしいことにバイトアドレッシングではなく、16bit(21bit) ワードアドレッシングある。
この基本 RISC としての命令体系は、PICO, SIMPL などの教育用 RISC プロセッサのごとく簡単である。基本命令は

- WGG 命令: レジスタ間演算命令
- WGI 命令: レジスタ-イミーディエイト命令
- Branch 命令

に分類される。

WGG 命令: 最初の 5bit は OPE code で次に 3 アドレッシングでレジスタを指定する。GPR0 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 は基本的にこれで数値演算を行なおうと思っていないので、Arithmatic Shift は持っていない。
GPR2, GPR3 はソース、GPR1 はディスティネーションである。

Instruction	Instruction Code	Meaning	Call
SUBGG	11110 | GPR1GPR2GPR3 | 0000	$GPR1 \leftarrow GPR2 - GPR3$	Sub
ADDGG	11110 | GPR1GPR2GPR3 | 0001	$GPR1 \leftarrow GPR2 + GPR3$	Add
SLLGG	11110 | GPR1GPR2GPR3 | 0010	$GPR1 \leftarrow GPR2 << GPR3$	Shift Left logical
SRLGG	11110 | GPR1GPR2GPR3 | 0011	$GPR1 \leftarrow GPR2 >> GPR3$	Shift Right logical
ANDGG	11110 | GPR1GPR2GPR3 | 0100	$GPR1 \leftarrow GPR2 \wedge GPR3$	AND
ORGG	11110 | GPR1GPR2GPR3 | 0101	$GPR1 \leftarrow GPR2 \vee GPR3$	OR
XORGG	11110 | GPR1GPR2GPR3 | 0110	$GPR1 \leftarrow GPR2 \oplus GPR3$	Exclusive OR

WGI 命令:　これも同様に基本的な作り方である。

Instruction	Instruction Code	Meaning	Call
SUBGI	01000 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 - Im_7{}^8\#\#Im$	Sub Imm.
ADDGI	01001 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 + Im_7{}^8\#\#Im$	Add Imm.
SLLGI	01010 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 << Im_7{}^8\#\#Im$	Shift Left Logical Imm.
SRLGI	01011 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 >> Im_7{}^8\#\#Im$	Shift Right Logical Imm.
ANDGI	01100 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 \wedge Im_7{}^8\#\#Im$	AND Imm.
ORGI	01101 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 \vee Im_7{}^8\#\#Im$	OR Imm.
XORGI	01110 | GPR1GPR2 | #Im.8bit	$GPR1 \leftarrow GPR2 \oplus Im_7{}^8\#\#Im$	Exclusive OR Imm.
LHI	01111 | GPR1 ---- | #Im.8bit	$GPR1 \leftarrow Imm\#\#0^8$	Load High Imm.

Branch 命令:　後述するが、パケット制御に関してはほとんどテーブルジャンプを用いるため、MBP の Branch 命令は極めて基本的なものである。Flag は Carry flag(C), Zero flag(Z) の二つしかなく、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 PC 相対だが、ただのジャンプ (BRA) は絶対ジャンプである (これは最初アセンブラがないころ便利だったが、直してもいい、、、)。サブルーティンコールは戻り番地を GPR15 につっこんで絶対ジャンプする。変則命令コードの利点を生かして絶対ジャンプはすべてのアドレス領域をカバーする。リターンは BR を用いる。

注意 **1:**　すべての分岐命令は **delay slot 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なわれない。

いいわけ **0:**　BR で使うレジスタに対してフォワーディングすると長大なクリティカルパスが発生して動作周波数の低下を招く。

Instruction	Instruction Code	Meaning	Call
BOZ	1000000 | #Im.14bit	$if(Z)\ PC \leftarrow PC + Im^2_{14}\#\#Im.$	Branch on Zero
BNZ	1000001 | #Im.14bit	$if(\bar{Z})\ PC \leftarrow PC + Im^2_{14}\#\#Im.$	Branch not Zero
BGT	1000010 | #Im.14bit	$if(\bar{C} \wedge \bar{Z})\ PC \leftarrow PC + Im^2_{14}\#\#Im.$	Branch greater than
BLE	1000011 | #Im.14bit	$if(C \vee Z)\ PC \leftarrow PC + Im^2_{14}\#\#Im.$	Branch Less equal
BAL	10001 | #Im.16bit	$PC \leftarrow Im, GPR15 \leftarrow PC$	Branch and Link
BRA	10010 | #Im.16bit	$PC \leftarrow Im$	Branch
BR	11101 | ---- GPR1 ---- | 0001	$PC \leftarrow GPR1$	Branch Register

ローカルメモリアクセス:　何度も繰り返すがローカルメモリは 16bit 幅 (21bit 幅)、16bit アドレッシングである。バイト単位のアクセスは考えていない。最初は単一の 64K × 16bit のアドレス空間を考えたが、途中でなんだか足りないような気がしてきて、足りないメモリをさらに I/O で食い荒すとまずいと考え、64K の I/O 空間を付け足した。外部的には I/O 信号で区別される。高速メモリを用いて 1 クロックでアクセスすることを考えているが、wait 信号によりアクセスを遅延することができる。ピン数の不足により命令メモリとデータメモリを分離しておらず、もちろんキャッシュなんぞ持っていないため、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。しかし、データハザードによるストールは起こらない。ローカルメモリは、下位 16bit 分が GPR との間で転送される。上位 5bit は、命令以外ではめったに使われないので、UDR(Upper Data Register) という 5bit の特殊レジスタとの間でデータのやりとりが行なわれる。すなわち、LLM, LLI 命令でロードされたデータの上位 5bit は UDR に入り、SLM, SLI 命令では

メモリの上位5bitに対してUDRの値が書き込まれる。UDRとGPRはLUDR(Load UDR), SUDR(Store UDR)命令でデータのやりとりを行なう。(その他の命令参照)
アドレッシングは7bit displacement付き間接レジスタ指定である。

Instruction	Instruction Code	Meaning	Call
LLM	10100 | GPR1 | GPR2 | 0 #Im.7bit	$GPR1 \leftarrow M[GPR2 + Im_7{}^9\#\#Im]$	Load Local Memory
LLI	10100 | GPR1 | GPR2 | 1 #Im.7bit	$GPR1 \leftarrow I[GPR2 + Im_7{}^9\#\#Im]$	Load I/O
SLM	10101 | GPR1 | GPR2 | 0 #Im.7bit	$M[GPR2 + Im_7{}^9\#\#Im] \leftarrow GPR1$	Store Local Memory
SLI	10101 | GPR1 | GPR2 | 1 #Im.7bit	$I[GPR2 + Im_7{}^9\#\#Im] \leftarrow GPR1$	Store I/O

1.2 PBR(Packet Buffer Register)に関する命令

PBRはパケットを扱うための68bit幅の専用レジスタであり、汎用64本、From RDT 8本、To RDT 8本の3種類に分かれる。このPBRは、GPRを用いて4bit offset(displacement)付きのレジスタ間接指定でアクセスされる。ポインタとするGPRの11,10bit目の2bitでバッファの種類を指定し、後は8bit単位にバイトアドレッシングされている。アドレッシングは図1に示すように、基本的にはJUMP-1の原則に従ったBig Endianだが、0000 ~ 0111がパケット本体，1000がタグ領域へのアクセスとなり，1001–1111は無効となる点がちょっと変わっている。PBRは、このアドレスにより8bit単位、16bit単位でアクセスされるがDouble Word(68bit)境界を越えたアクセスは不可能である。

- From RDT-PBR: 68bit × 8 flit × 3パケット分 GPR: XXXX00XXXXXXXXXX)
  RDTIFから受け取るRDTパケット。
- To RDT-PBR: 68bit × 8 flit × 3パケット分 (GPR: XXXX01XXXXXXXXXX)
  RDTIFへ送るRDTパケット。
- PBR: Packet buffer 68bit × 64flit (GPR: XXXX10XXXXXXXXXX)
  汎用のPBR

PBRの役割は、1. クラスタメモリのアクセス、2. C-busとのパケットのやりとり、3. RDTとのパケットのやりとりの3つである。このため、PBR上でパケットの生成、分解を行なうことになる。クラスタメモリ、C-bus, RDTとのやりとりは68bit X flit長の単位で行なわれるので、PBR上での操作の目的は主としてパケットのヘッダを作ることである。RDT, Cbusパケットヘッダはフィールド長が8bit以内で収まることが多く、長くても16bit以内で収まることが多いため、PBRは8bit単位、16bit単位に扱うのに便利な命令体系になっている。命令中でPBRはポインタとなるGPR(4bit)+Offset(4bit)=8bit長で指定される。

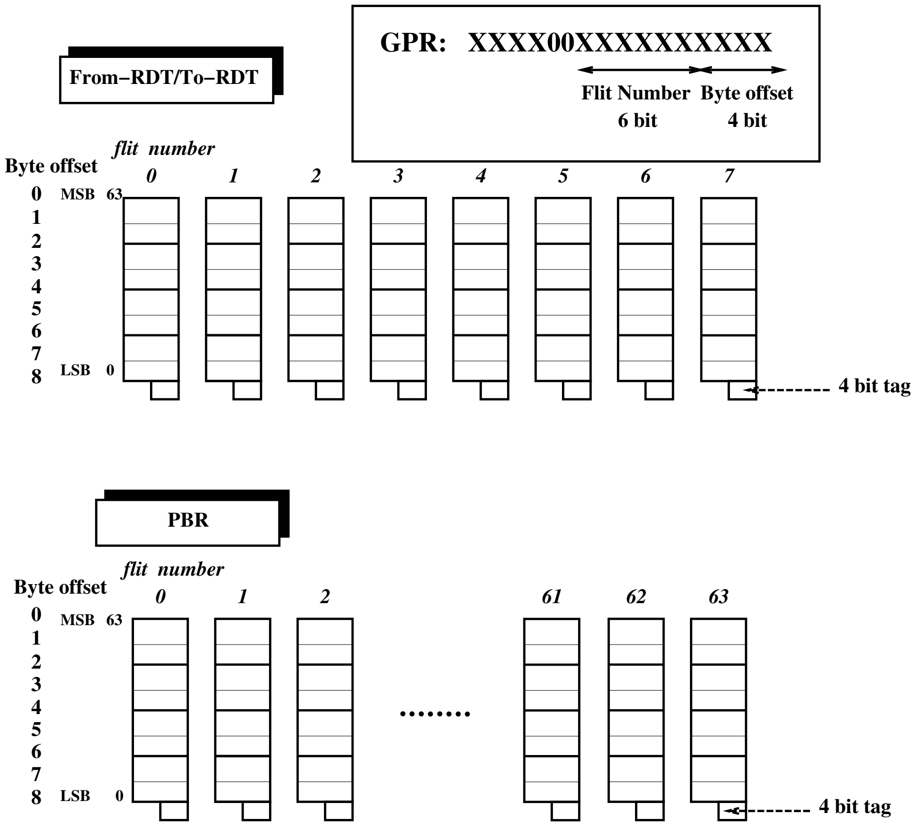
From−RDT/To−RDT
GPR: XXXX00XXXXXXXXXX
Flit Number
6 bit
Byte offset
4 bit
Byte offset
flit number
0
1
2
3
4
5
6
7
8
MSB 63
LSB 0
4 bit tag
PBR
Byte offset
flit number
0
1
2
61
62
63
MSB 63
LSB 0
....
4 bit tag

図 1: PBR の構成

BPI 命令 8bit の直値データと指定された PBR の 8bit フィールドとの間で演算を行なう命令。ヘッダのフィールドを操作するのに便利である。

Instruction	Instruction Code	Meaning	Call
SUBPI	00000 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR - Im$	Sub PBR Imm.
ADDPI	00001 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR + Im$	Add PBR Imm.
SLLPI	00010 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR << Im$	Shift Left PBR Imm.
SRLPI	00011 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR >> Im$	Shift Right PBR Imm.
ANDPI	00100 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR \wedge Im$	And PBR Immediate
ORPI	00101 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR \vee Im$	OR PBR Immediate
XORPI	00110 | PBR8bit | #Im.8bit	$PBR \leftarrow PBR \oplus Im$	Exclusive OR PBR Immediate
LPI	00111 | PBR8bit | #Im.8bit	$PBR \leftarrow Im$	Load PBR Immediate
CPI	10110 | PBR8bit | #Im.8bit	$PBR - Im$	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 8bit flit 内で行なわれる。

WPG 命令: PBR の指定された 16bit と GPR 間の演算命令 (MVBPG は 8bit データ) である。d の値によって GPR、PBR のどちらを destination にとることもできる。つまり、$d=0 \leftarrow dest=GPR$, $d=1 \leftarrow dest=PBR$ である。

Instruction	Instruction Code	Meaning	Call
SUBPG	11111 | PBR8bit GPR | d000	$dest \leftarrow PBR - GPR$	Sub PBR-GPR
ADDPG	11111 | PBR8bit GPR | d001	$dest \leftarrow PBR - GPR$	Add PBR-GPR
MVBPG	11111 | PBR8bit GPR | d010	$dest \leftarrow PBR - GPR$	Move byte PBR-GPR
MVPG	11111 | PBR8bit GPR | d011	$dest \leftarrow PBR - GPR$	Move PBR-GPR
ANDPG	11111 | PBR8bit GPR | d100	$dest \leftarrow PBR - GPR$	AND PBR-GPR
ORPG	11111 | PBR8bit GPR | d101	$dest \leftarrow PBR - GPR$	OR PBR-GPR
XORPG	11111 | PBR8bit GPR | d110	$dest \leftarrow PBR - GPR$	Exculsive OR PBR-GPR

MPP 命令: PBR 間の移動命令で 8bit, 16bit, 68bit 単位の移動が可能である。

Instruction	Instruction Code	Meaning	Call
MVBPP	10111 | PBR1 8bit | PBR2 8bit	$PBR1 \leftarrow_{8} PBR2$	Move Byte PBR-PBR
MVPP	11100 | PBR1 8bit | PBR2 8bit	$PBR1 \leftarrow_{16} PBR2$	Move PBR-PBR
MVLPP	11101 | PBR1 PBR2 ---- | 0000	$PBR1 \leftarrow_{68} PBR2$	Move long PBR-PBR

68bit の移動時 (MBLPP) は offset が使えないので、PBR1, PBR2 は 4bit(つまりポインタとなる GPR) で表される。

注意 2: ポインタとなる **GPR** については完全にフォワーディングが行なわれるが、**PBR** の値はフォワーディングされない。したがって、**PBR** に書き込んだ値を使うためには **1** 命令分の **NOP** または関連のない命令を挿入しなければならない。

いいわけ **1:** PBR は 2 アドレッシングである。これはフィールド長のみを考えたためではなく、かなりの容量がある PBR を 3-port memory にしたり、1 クロック内に読み / 書きができるようにするのが実装

上困難であったためである。

いいわけ2: PBRは68bit長だが、これが長いのが威力を発揮するのは、RDT、クラスタメモリ、C-busとの転送でのみであり、実際は8bit/16bitのフィールド毎に使えるレジスタとなっている。つまり68bit長の足し算などはもちろんできない。したがって、この構成はPBR上で扱うデータの長さが16bitを越えたとたんにひどい目にあう。しかし、C-busヘッダのフィールド中で最長の24bitアドレスは、Samll Page Address/Small Page Offset/Lineに分かれるわけで、それぞれは16bitを越えない。また、RDTのヘッダは最長のCompressed Addressも16bitは越えない。したがって、大丈夫だと見切ってこの構成にした。現在の構成では32bit Adderを用いると、50MHzでMaxで動作するにはパイプラインを増段する必要が生じる。またOffsetの利用により、16bitデータとして扱える範囲は8bit単位でずらすことができるので、扱うデータが16bit境界を越えても長さ自体が16bitを越えなければ無駄はさほど大きくない。（と思う）。

いいわけ3: PBRはフォワーディングされない。これをやるとマルチプレクサの増大によりクリティカルパスが伸びてしまい、50MHzでの動作が不可能になる。また、68bitのパスをブンブン引き回すのは気が進まない。

いいわけ4: PBRは余ったビットが変な番地についているが、全体をLSB側にずらすと000が無効になってしまうため、やむを得なかった。

いいわけ5: PBRのOffsetは4bitしかなく中途半端な気がするが、これは、68bit長の中のフィールドを指定するのに結構役に立つ。これ以上増やすと指定が難しくなる他、アドレス計算の加算器の遅延が無視できない。

1.3 RDT、クラスタメモリ、C-busとの転送命令

先に述べたように、PBRの役割は、1. クラスタメモリのアクセス、2. C-busとのパケットのやりとり、3. RDTとのパケットのやりとりの3つである。1,2はどのレジスタでも可能だが、3は送信はTo RDT, 受信はFrom RDTしかできない。このように、PBRはC-bus/クラスタメモリ側（つまりはMMC側）のアクセスと、RDT側のアクセスの根本思想が異なっている。すなわち、

- RDT側: 図2に示すようにパケットバッファ単位でサイクリックバッファになっている。RDTインタフェースにより受信が行なわれたパケットが格納されたバッファのひとつは、GRDT命令によりform RDT-PBRとしてMBP Coreで扱えるようになる。一方、To RDT-PBR上に組み立てられたパケットはPRDT命令によって、RDTインタフェース接続され、転送が開始する。これは、内部バッファ間の転送を極力避けるための戦略で、送受信バッファをそのままレジスタとしてアクセスすることが可能となる。

- C-Bus/クラスタメモリ側: RDT-MBP Core側が1対1転送で比較的単純であるのに対し、MMC側はC-Bus-クラスタメモリ-MBP Coreの三者通信になってしまい、（佐藤さんの設計が理解できなかったこともあって）上記の戦略を取ることが困難である。そこで、ここではやむなくTGM, GPB, PBP命令により、MMC内のバッファとPBR間の転送を行なうことにしている。このため、転送中はPBRを利用する命令は動作することができなくなり、PBRを用いる命令を実行するとパイプラインがストールする。

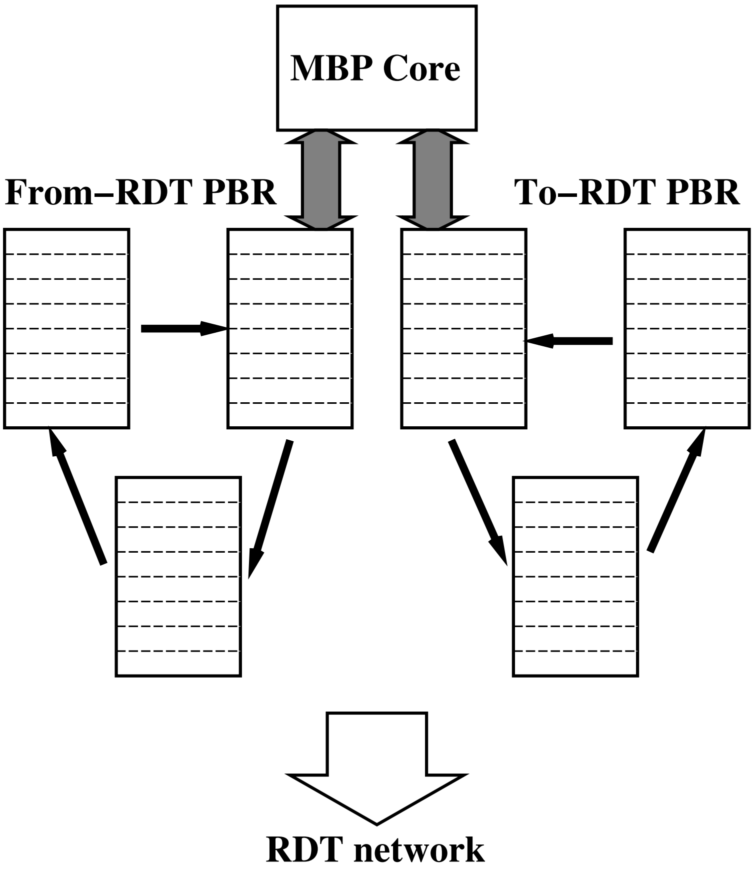

図 2: サイクリックバッファの構成

いいわけ 6 ： この方式は一見恐ろしく汚いようだが、RDT での受信用バッファを直接 MMC に接続して送信したり、その逆を可能とするのは、配線の問題と、マルチプレクサの塊になること、排他制御のタイミングが複雑になることから非現実的である。とすると、MMC 側のバッファを RDT 同様パケットバッファ単位のサイクリックバッファとして、PBR として MBP-Core 内で直接さわれるようにしたところで、Core 内で From MMC-PBR から To RDT-PBR の転送（あるいはその逆）が必要になってしまう。現在の方式では、必要に応じて C-bus からのパケットのデータ部やクラスタメモリのデータ部を To RDT-PBR に直接転送することもできるできるし、逆に From RDT-PBR のパケット本体を直接クラスタメモリや C-bus 用バッファに転送できる。したがって、この方法は必ずしもロスが多いとはいえないと思う。また、From RDT-PBR, To RDT-PBR を汎用 PBR と独立させたのは、次に述べるように RDT との間の転送が極めて特殊なデータの並び変えを必要とするためで、すべてのバッファに対してこれを実行することが困難であったためである。

1.4 RDT とのやりとり

RDT 上でのパケットは、17bit × 2=34bit 構成であり、図 3に示されるように、かなり複雑な構成である。ルータ 2 個をビットスライスに用いたため、ヘッダ部 3flit は 2 つのルータで共通でなければならない。そこで RDT インタフェース部は、図 4に示すように、送受信時に自動的にパケットヘッダの複製または除去、を行なってくれる。このため、From/To RDT PBR 上ではこの点に関して考慮する必要はない。逆にこのことは From/To RDT PBR 上でのデータの配置の自由度をなくしている。パケットは必ず 000 番地から始める必要があり、ACK パケット等の短いパケットを作った場合に、余りの Flit は無駄になる。なお Normal Packet の最終 flit はデータ部の縦パリティであるが、これは自動的にインタフェースが付加およびチェックしてくれるので、ユーザは関知する必要はない。
RDT に対するパケットの送出および受信は次の命令で行なう。

Instruction	Instruction Code	Call
PRDT	11101 | 001000000000 | 0010	Push RDT
GRDT	11101 | 000100000000 | 0010	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 buffer を To RDT-PBR として MBP Core に接続する。使用可能な buffer 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り込みがかかった時点で使用可能 (MBPcore 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 MBPcore に接続することが出来る．j

どうしよう 1: 現在、パケットが RDT から到着して From RDT-PBR として利用可能になった場合は後述のように割り込みがかかる（あるいはテーブルジャンプの対象になる）。しかし、RDT i/f が溢れて、To PRDT が使用不能になった時には割り込みをかける機能は持たせておらず、内部レジスタ（後述）から読んでチェックする方式をとっているが、割り込み機能の中にいれた方がいいかもしれない。また、現在 RDT とのやりとりのためのパケットバッファは送受信共に 3 であるが、明らかに不足するため、面積詳細計算後、増強する予定である。

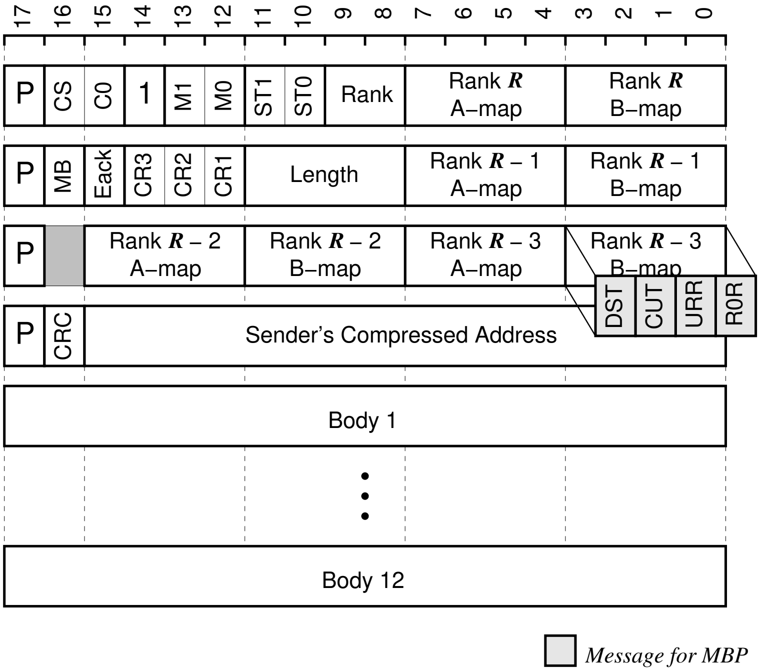

図 3: RDT マルチキャストパケットのフォーマット

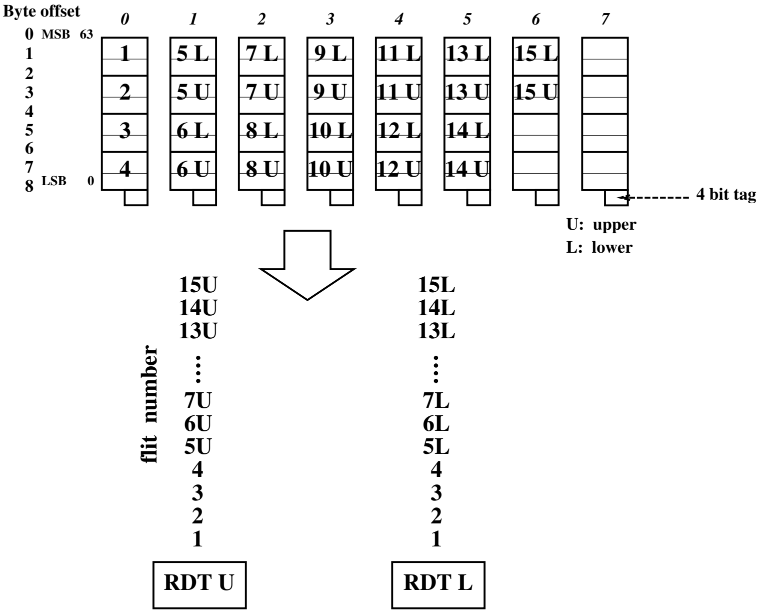

図 4: RDT i/f におけるパケットの複製

1.5 MMC とのやりとり

MMC とのやりとりは、以下の命令で行なわれる。これらの命令では Offset 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 4bit で指定される。

Instruction	Instruction Code	Call
TGM	11000 | ---- | PBR1 4bit | ---- PBR2 4bit	Transfer Grobal Memory
PBP	11001 | ---- | PBR1 4bit | ---- PBR2 4bit	Put Bus Pakcet
GBPS	11010 | 0--- | PBR1 4bit | ---- PBR2 4bit	Get Bus Packet from SBUS
GBPM	11010 | 1--- | PBR1 4bit | ---- PBR2 4bit	Get Bus Packet from MTC
MMCP	10011 | 1--- | PBR1 4bit | ---- PBR2 4bit	MMC Put
MMCG	10011 | 0--- | PBR1 4bit | ---- PBR2 4bit	MMC Get
MEMBAR	11011 | 000000000000 | 1010	Memory Barrier

メモリバリアについて:　以前は，TGM，PBP，GBPM，GBPS の後にその動作が完了する前に PBR にアクセスする際には，Core は必ずストールさせていたが，MEMBAR の導入により，それらの動作を完全に uncompletion にし，その動作に使用していない PBR へのアクセスは自由に行なうことが出来るようになった．ここで言う，PBR 間の区別は (MMC-PBR, From-RDT PBR, To-RDT PBR) の 3 種類に分類している．
たとえば，MMC-PBR にアドレスフリット，データフリットを指定し TGM を行なった場合は，
MEMBAR をはさむことなく From-RDT PBR や To-RDT PBR はアクセスすることが出来，
MMC-PBR へのアクセスを行なう際には MEMBAR をはさまなくてはならない (確実に TGM が終了していることがわかっている際にはその限りではない)．また，MMC-PBR にアドレスフリット，
From-PBR にデータフリットを指定し，TGM を行なった際には，MEMBAR をはさむことなく To-RDT PBR をアクセスすることができる．

補足注意:　MEMBAR 命令は遅延分岐で実行できない。

クラスタメモリのアクセスについて:　クラスタメモリのアクセスは、TGM 命令を用いる。この命令はまず PBR1 にアドレスフリットが入る。アドレスフリットは Cbus 仕様書の 29 ページおよび MMC 仕様書の 6 ページを参照されたい。
まずは、Write について説明する。クラスタメモリへの Write には、write_req(CBus 仕様書の 50 ページ) を用いる。Write のデータサイズは block(4 double word) 単位である。この時、WWE フィールドには **10** を、ccbl フィールドには **1** を設定しなければならない (一応、SIZE フィー ルドを **100** としておくことが望ましい)。例えば、アドレス 0x00C0 にデータを Write するならば、アドレスフリットを 0A 88 00 00 00 00 00 C0 とすればよい。
次は、Read について説明する。クラスタメモリからの Read には、read_req(CBus 仕様書の 42 ページ) を用いる。Read のデータサイズは 1 double word か、block のどちらかを選択できる。つまり、1 double word を Read するならば、WWE フィールドを **01** に設定する必要がある (一応、SIZE フィー ルドを **011** にしておくことが望ましい)。block の場合は上記の Write の場合を参照してほしい。Read の場合には、Write とは違って ccbl フィールドに **1** を設定する必要はない。例をあげると、アドレス 0x00C0 からデータを Read するならば、アドレスフリットを 00 08 00 00 00 00 00 C0 とすればよい。
この際に同時に DTAG がアクセスされて、それぞれのデータフリットの WWE にセットされる。

TAG アクセスについて: まず、Write について説明する。DTAG への Write には、control_write_req(CBus 仕様書の 67 ページ) を用いる。クラスタメモリと同じように、Write のデータサイズは block(4 double word) 単位である。この時、WWE フィールドには **10** を、ccbl フィールドには **1** を設定しなければならない (一応、SIZE フィー ルドを **100** としておくことが望ましい)。
TAG のアクセスで注意すべき点は、DSP アドレス (メモリコントローラ仕様書の 6 ページ参照) を用いるということである。このアドレスはアドレスフリット **[33:2]** に対応している。そして、DSP アドレスの D/T フィールドに **1** を設定しておく必要がある。また、書き込むデータはデータフリットの下位 bit に入れておかなければならない。例えば、アドレス 0x00C0 の DTAG にデータ 010 を Write するならば、アドレスフリットを 0A E8 00 00 08 00 00 C0 とし、データフリットを 00 00 00 00 00 00 00 02 とすればよい。
次は、Read について説明する。TAG からの Read には、control_read_req(CBus 仕様書の 66 ページ) を用いる。Read のデータサイズは 1 double word か、block のどちらかを選択できる。つまり、1 double word を Read するならば、WWE フィールドを **01** に設定する必要がある (一応、SIZE フィー ルドを **011** にしておくことが望ましい)。block の場合は上記の Write の場合を参照してほしい。Read の場合には、Write とは違って ccbl フィールドに **1** を設定する必要はない。
Read の TAG のアクセスで注意すべき点は、DSP アドレス (メモリコントローラ仕様書の 6 ページ参照) を用いるということである。このアドレスはアドレスフリット **[33:2]** に対応している。そして、その DSP アドレスの D/T フィールドに **1** を設定しておく必要がある。例えば、アドレス 0x00C0 の DTAG からデータを Read するならば、アドレスフリットを 00 E8 00 00 08 00 00 C0 とすればよい。

いいわけ 8: この方法は、アドレスフリットを作るのに結構時間がかかって悲しいような気がするが、PBR は 64 本あるので、アドレスフリットのテンプレートを何本か作っておいて、アドレス部のみ書き換えるようにすればいい。アドレスフリットの PBR1 には普通の PBR を用いて、PBR2 には、To RDT-PBR, From RDT-PBR も指定できるので、クラスタメモリからの 1 ライン分をそのまま To RDT-PBR に転送して、ボディとしてパケットを構成できる。

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、PBR1 にアドレスフリットを入れ、PBR2 にデータの先頭を置いて送信する。インタフェースは、req か reply かを判別して MMC の所定のバッファに突っ込む。

補足注意: 汎用 PBR にデータフリットを置いておく場合、その先頭フリット番号は flit number $\equiv 0 \pmod 4$ としなければならない。つまり、0-3 フリット、4-7 フリット ... 60-63 フリットなどにデータフリットを置けるということである。これを守らない場合にはその動作は保証されない。

GBPS/GBPM: 本来 PBP と対になるはずだが、MMC からのパケットは、1. C-bus 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。これらは後に述べるように割り込みあるいはテーブルジャンプにより判定可能である。

MMCP/MMCG: MBP-Core は MMC の内部バッファに直接パケットをつっこんだり、無理矢理読みだしたりは原則としてしない。これは、インタフェースを介さずにこんなことを行なうとクラスタメモリからキャッシュが読み出すパケットを破壊するためである。しかし、インタフェースを用いていると、所定のバッファ以外はアクセスできなくなってしまう。そこで裏技として、一応バッファとの間に転送するコマンドを用意した。これが MMCP と MMCG である。PGR1 をコマンド用アドレスフリットとして、

PBR2 を先頭にしてデータのやりとりを行なう。したがってコマンドの内容と MMCP/MMCG の種類が一致しないとバッファの内容が壊れたり、むちゃくちゃな動作をする可能性があり、通常は決して用いてはならない。

注意 3:　上記の命令は、実行に **9-12** クロックほど程度要する。この間、**MBP Core** は次の命令の実行を行なうことができる。つまりこれらの命令は **Out-of-order Completion** を招く。しかし、継続した命令がこれらの命令の終了前に、**PBR** を用いる場合 (あるいは割り込みリセット等のシグナル命令を実行する場合)、その時点でストールがかかり、命令実行終了後にストールが解除される。

注意 4:　上記の命令の実行時は、直前の命令で **PBR** をアクセスしてはならない。

いいわけ 9:　クラスタメモリアクセスや C-bus バッファからの転送は基本的にそんなに時間を要するわけではないが、C-bus アクセスとの衝突により、場合によっては待たされる可能性があって固定時間では終らない。しかし、いちいちこれらの命令を使う毎に割り込みやテーブルジャンプ、フラグのポーリング等で実行終了を検出するのはしんどいものがある。また、これらの命令では PBR は、MMC に接続して用いる（サイクルスチールみたいなのはますますマルチプレクサが増えてしまい遅延時間がしんどい）ことが必要なため、どっちみち命令実行中は PBR はアクセスできない。このため、Out of order Completion+ PBR を使う命令でストールという方法を採用した。賢いプログラム作成者は、これらの命令の使用後に PBR を使わない命令を挿入して性能向上を図ることができる。めんどうならば、いい加減に書いてもストールしてくれる。注意 4 は Write back のステージとの衝突の問題から起き、検出してストールさせることも考えられるが、制御が面倒なのでやめた。

1.6　Table Jump と割り込み

MBP Core は RDT や MMC の状況に応じて Table Jump したり、割り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 と割り込みは Mask bit を除いて同じ機構を利用しており、割り込みは Table Jump が自動的にかかるだけである。Table Jump ／割り込みの要因は下の 13 で、番号が大きいもの程優先順位が高い。

```
Table jump priority/mask
0:      RDT Shootdown
1:      RDT Ack emergency buffer
2:      RDT Send empty
3:      RDT Packet arrival
4:      CBUS Interrupt
5:      MTC Interrupt
6:      Pend counter overflow
7:      Pend counter underflow
8:      Time out (Timer)
9:      I/O 0
10:     I/O 1
11:     I/O 2
12:     I/O 3
```

MBP Core では後述する内部メモリの特定のアドレスに割り当てられている Table Jump Mask, Interrupt Mask, Priority の 3 つのレジスタに値を書きこむことにより、受け付ける要因をコントロールする。Table

Jump Mask および Interrupt Mask は、それぞれ Table Jump 命令時、割り込みにジャンプする要因のマスクである。このマスクはリセット時に自動的に 0 になり、対応する bit を 1 にしないと、Table Jump または割り込みが発生しない。ポーリングベースの Table Jump 命令と割り込みとでは性質が違うので、別々のレジスタを設けている。さて、割り込み要因の優先順位は固定で番号が大きい方が高いが、時にこれは不便となる。そこで、Priority レジスタを設けており、ここに 1 が立ったものは Priority Class となり、0 の要因より優先順位が高くなる。もちろん、Priority Class 同士では番号の大きい方が勝つので、本質的な公平性の保証にはならないが、そもそも割り込み要因は不公平なものなので、これで十分だろう。Mask bit が 1 になっており、要因が発生して、Table Jump/割り込みが受け付けられた場合、以下のジャンプテーブルに従ってテーブルジャンプする。このジャンプテーブルは後述するように内部メモリとしてアクセスすることができる。Lv で、対応する割り込みレベルを示す。同一レベルで複数の割り込み要因が存在した場合、原則として、ジャンプ先の番地が小さい方が優先される。MTC Self buffer interrupt は、address flit の bit47 を 1 にしておいた場合、MBP からのメモリアクセスを処理した後、trap する。たとえば、MBP からバスパケットを止めたまま、atomic な処理を行なう場合に用いる。

```
Reserved Jump Table address(used by TJRQ)
0:  CBus  Request packet calls MBP (Lv. 4)
1:  CBus  Reply packet calls MBP (Lv. 4)
2:  CBus  Pend lock counter calls MBP (Lv. 4)
3:  CBus  Lock counter calls MBP (Lv. 4)
4:  MTC  Request packet flag checks (Lv. 5)
5:  MTC  Reply packet flag checks (Really existing?) (Lv. 5)
6:  MTC  Self buffer interrupt (Lv. 5)
7:  DRAM error (Lv. 5)
8:  Normal packet arrival (Lv. 3)
9:  Ack packet arrival (Lv. 3)
A:  RDT Special packet arrival (Lv. 3)
B:  RDT Parity Error packet aarival (Lv. 3)
C:  RDT Ack emergency buffer (Lv.1)
D:  RDT Shoot down mode end (Lv.0)
E:  RDT Packet send buffer not full (Lv. 2)
18: Pend counter overflow (Lv. 6)
19: Pend counter underflows (Lv. 7)
1B: Time out (Lv. 8)
1C: I/O 0 (Lv. 9)
1D: I/O 1 (Lv. 10)
1E: I/O 2 (Lv. 11)
1F: I/O 3 (Lv. 12)
```

MTC Self buffer interrupt (Lv. 5) について: この割り込み処理ルーチンには、EMMTRAP が 1 になった時や MBP core が

- CNTL = 0
- CNTL = 1 & control_read & BUFX
- CNTL = 1 & control_write & BUFY

以外のアドレスフリットを用いた時も飛ぶ。

どうしよう 2: Bus packet と MTC packet は FIFO 化が行なわれる前に作ったため、バッファ番号によって別の番地に飛ぶことができる。しかし、FIFO 化された以上、この機能はあまり意味がないように思える。ただ、reply 用か req 用かはバッファ番号によって区別できるのでこれだけ分けてあとは、一緒にしようかと思っている。この改造はすぐできる。→ 7/8 改造済み

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なう命令。TJ は、GPR1 によって示した内部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時に使う。

Instruction	Instruction Code	Call
TJRQ	11101 | ---- ---- ---- | 0011	Table Jump with Requests
TJ	11101 | ---- GPR1 ---- | 0100	Table Jump

注意 5: **TJRQ, TJ** もジャンプ命令なので、**1 slot** を持つ遅延分岐である。しかも、**TJ** の引数である **GPR** へはフォワーディングができない。

割り込み: 割り込みが起きるためには、Interrupt Mask レジスタに 1 が立っている要因が発生する他に、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。割り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、割り込み禁止になる。したがって割り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、(Lv.2 RDT send not full に対しては PRDT, Lv. 3 RDT packet arrival に対しては GRDT が要因のリセットの当たる。ただし、Lv.2 は通常状態で割り込みを禁止しておくのが普通だと思う。) また CSM では shoot down mode、CAB は ack 収集緊急バッファをクリアする場合に用いる。これらのモード、バッファに対しては RDT インタフェース仕様書に書く予定である。
さて、その割り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 delay slot で INTE 命令を実行すれば、リターン直後に割り込み可能になる。INTD は割り込みを禁止する命令であるが、プログラマが必要とするかどうかわからない。
MBP-Core は、多重割り込みをサポートする最小限の機能を持っている。LIPA(Load Interrupt Address) は,IPA(InterruPt Address register) 中の割り込みの戻り番地を GPR にロードする。また、LIPS(Load Interrupt Status) は、IPS(InterruPt Status register) 中のフラグ 2bit を GPR の LSB 2bit 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、割り込み許可にすることにより多重割り込みが可能になる。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、これは用意していない。したがって、多重割り込みのプロ

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。

Instruction	Instruction Code	Call
RFI	11101 | ------------ | 0110	Return From Interrupt
INTE	11101 | 000010000000 | 0010	Interrupt Enable
INTD	11101 | 010000000000 | 0010	Interrupt Disable
CSM	11101 | 000001000000 | 0010	Clear Shootdown Lv.0
CAB	11101 | 000000100000 | 0010	Clear Ack em. Buffer Lv.1
CCB	11101 | 000000000001 | 0010	Dequeue CBus Lv.4
COV	11101 | 000000000010 | 0010	Clear Pending overflow Lv.6
CUF	11101 | 000000000100 | 0010	Clear Pending underlow Lv.7
CMT	11101 | 000000001000 | 0010	Clear MTC Lv.5
LIPA	11011 | GPRd --------- | 0011	Load Interrupt Address
LIPS	11011 | GPRd --------- | 0100	Load Interrupt Status
SSR	11011 | --------- GPRs | 0101	Set Status

どうしよう3:　割り込み処理中に割り込みを許すためには、割り込み中にPCが格納されているIARとGPR間、フラグとGPR間の転送命令を設ければよい。これは可能であるが、やる価値があるだろうか？
→ 結局、実装したが、やや不完全か？

1.7　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。この内部メモリは恐ろしいことに、68bit単位にアドレスが降ってあるが、実際にアクセスするデータ幅は16bitが主である。16bitアクセスした場合はLSB16bitが転送の対象となる。PBRの指定はOffset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．

LIMG	11101 | GPRdGPRs offset | 1000	$GPRd \leftarrow_{16} IM[GPRs + offset]$	Load Internal memory to GPR
SIMG	11101 | GPRdGPRs offset | 1001	$IM[GPRd + offset] \leftarrow_{16} GPRs$	Store Internal memory from GPR
LIMP	11101 | PBRdGPRs offset | 1010	$PBRd \leftarrow_{68} IM[GPRs + offset]$	Load Internal memory to PBR
SIMP	11101 | PBRsGPRd offset | 1011	$IM[GBRd + offset] \leftarrow_{68} PBRs$	Store Internal memory to PBR

内部メモリアドレスについては3節に示す。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。しかし以下の変な制約がある。

注意6:　内部メモリアクセスの直後に**TJRQ**命令、**TJ**命令を行なってはならない。

いいわけ10:　注意6については、ストールするのがめんどくさいだけです。多分あんまり上記のケースはないと思います。ごめんなさい。

1.8 その他の命令

MBP Core は専用プロセッサなので、変な命令を持っている。

Instruction	Instruction Code	Call
HSHC	11101 | GPRd PBRs ---- | 1100	Hash Cluster Address
HSHG	11101 | GPRd GPRs ---- | 1101	Hash GPR
HSHN	11101 | GPRd PBRs ---- | 1110	Hash Net Address
GUN	11101 | GPRd GPRs ---- | 1111	Get Unique Number
BITS	11011 | GPRd GPR1 GPR2 | 0001	Bit set
BITR	11011 | GPRd GPR1 GPR2 | 0000	Bit reset
BITC8	11011 | GPRd GPRs ---- | 0010	Bit Count (8bit)
TJRI	11101 | ---- ---- ---- | 0101	Table Jump interrupt
CRDT	11101 | 000000010000 | 0010	Clear RDT interface
LUDR	11011 | GPRd ---- ---- | 1000	Load Upper Data Register
SUDR	11011 | ---- ---- GPRs | 1001	Store Upper DataRegister

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下の機能を持つ。

- HSHC: PBR 上の Cluster Address 分つまり、27–5bit 目を 8bit 単位に折り畳んで Exclusive OR をして 8bit 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 16bit を 8bit 単位に折り畳んで Exclusive OR をして 8bit 幅にして GPRd の下位にセットする。上位は 0 にする。RDT の Compressed Address 用
- HSHN: PBR 上の Net address 分、つまり 41–5bit 目を 8bit 単位に折り畳んで Exclusive OR をして 8bit 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 番目のビットをセットして GPRd に格納する。
- BITR: GPR1 上の GPR2 番目のビットをリセットして GPRd に格納する。
- BITC8: GPRs 上の下位 8 ビット中の 1 の数を数えて GPRd に格納する。

TJRI 命令は、割り込み処理のため、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 5bit 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。

いいわけ 11: 当初、UDR は置かずに PBR とローカルメモリとの転送命令を用意する予定だったが、PBR はパイプラインピッチの関係で EX ステージでアクセスされることになったので、実装が困難になった。実際上、命令を I/O からロードしてメモリ空間に置く時以外は上 5bit は使わないので、これでもいいのではないでしょうか。

注意 7: LUDR, SUDR はフォワーディングされないので、直後の SLM には NOP を入れる必要がある。

2　タイマとモニタ

モニタとタイマは、基本的には32bitのカウンタである。内部アドレスの以下の番地に割り付けられている。

```
0x6000 : Monitor 0 Upper 16bit (31..16)
0x6001 : Monitor 0 Lower 16bit (15..0)
0x6002 : Monitor 0 command (2..0)
0x6003 : Monitor 0 overflow (0)
0x6004 : Monitor 1 Upper 16bit (31..16)
0x6005 : Monitor 1 Lower 16bit (15..0)
0x6006 : Monitor 1 command (2..0)
0x6007 : Monitor 1 overflow (0)
0x6008 : Monitor 2 Upper 16bit (31..16)
0x6009 : Monitor 2 Lower 16bit (15..0)
0x600A : Monitor 2 command (3..0)
0x600B : Monitor 2 overflow (0)
0x600C : Monitor 3 Upper 16bit (31..16)
0x600D : Monitor 3 Lower 16bit (15..0)
0x600E : Monitor 3 command (3..0)
0x600F : Monitor 3 overflow (0)
0x6010 : Timer Upper 16bit (31..16)
0x6011 : Timer Lower 16bit (15..0)
0x6012 : Timer command (2..0)
```

Commandのみ読み書き可能で, 他は読み専用である。

Monitor:　MBP-lightはモニタ用に4つの32bitカウンタを持つ。モニタ用カウンタは2種類に分かれる。Monitor0,1は、レベルカウンタで、このレベルがアクティブになっている間のクロック数を計測する。これに対してMonitor 2,3はイベントカウンタで、計測する信号がLからHへ変化した時にカウントアップする。
コマンドに以下の値を書き込むことにより下に示す項目を計測することができる。

- Monitor 0
    - 0: クリアする。
    - 1: カウンタの値をホールドする。
    - 2: 全クロック数をカウントする。
    - 3: RDTの受信バッファがFullである。
    - 4: RDTの受信バッファがEmptyである。
    - 5: ACK緊急バッファがFullである。
    - 6: NET Cache receive bufferがFullである。
    - 7: MMCのC-busからの要求がある。
- Monitor 1

  - 0: クリアする。
  - 1: カウンタの値をホールドする。
  - 2: 全クロック数をカウントする。
  - 3: RDT の送信バッファが Full である。
  - 4: RDT の送信バッファが Empty である。
  - 5: ACK Cache Double Full
  - 6: NET Cache sending buffer が Full である。
  - 7: Not Used
- Monitor 2
  - 0: クリアする。
  - 1: カウンタの値をホールドする。
  - 2: カウントアップする。
  - 3: RDT 送信回数
  - 4: RDT ACK 送信回数
  - 5: MBP Core からの送信回数
  - 6: Rank 0 ACK Cache マッチ回数
  - 7: ACK 収集要求回数
  - 8: NET Cache 要求回数
  - 9: Not used
  - A: Not used
  - B: Not used
  - C: Not used
  - D: Not used
  - E: Not used
  - F: Not used
- Monitor 3
  - 0: クリアする。
  - 1: カウンタの値をホールドする。
  - 2: カウントアップする。
  - 3: RDT 受信回数
  - 4: RDT ACK 受信回数
  - 5: Upper Rank ACK Cache マッチ回数
  - 6: ACK 登録失敗回数 (Overflow)
  - 7: ACK Packet 生成回数

  - 8: NET Cacheマッチ回数
  - 9: Not used
  - A: Not used
  - B: Not used
  - C: Not used
  - D: Not used
  - E: Not used
  - F: Not used

モニタは、まず0を書き込み、計測したい項目を示す数をコマンドレジスタに書き込むことにより計測をスタートする。

Timer: タイマはクロックを数える専門のモニタと思って良いが、大きな違いは、TJRQ, 割り込みの対象となる点である。

- 000: クリア
- 001: 2^4 クロックでタイムアウト(テスト用)
- 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

最初0を書き込んでおき、0以外のどれかのコマンドを書き込むとスタートする。タイムアウトするとTJRQまたは割り込み時にテーブルジャンプする。

3 内部Memoryアドレスマップ

MBP CoreとRDT i/fやMMCとのインタフェースはMBP Coreからの内部メモリとして提供している. 図5にInternal Memoryのアドレスマップを示し，以下の章でアドレスに関する詳細な説明を示す.

ACK収集用キャッシュ

ACK収集用キャッシュはrank0用と上位rank用の2つの256エントリのキャッシュが2-wayセットアソシアティブで構成されている。Compressed Addressの下位7bitでオフセットが決まり、上位9bitをタグとして登録する。Compressed Addressの構成を図6に示す.

- Bug bitはRDTルータのバグである，「rank2以上のランクからEackを1にしてマルチキャストをするとAck収集バッファの登録に失敗する(4flit目ではなく，5flit目をCompressed Addressとして登録してしまう)」ということに対応してRDT i/fが動作するかどうかを示すビットで，Bug bitが1の際には4，5flit目の両方にCompressed Addressをつけて送信する必要がある.

0000 –007F	Ack combining buffer for rank–0	
0100 –017F	Ack combining buffer for upper rank	
0200 –0203	RDT emergency buffer	
0300 –037F	Lock bit for Ack combining buffer	
0800 –09FF	Net cache	0
0A00–0BFF		1
0C00–0DFF		2
0E00–0FFF		3
1000 –10FF	Ack packet bitmap cache tag	
1100 –11FF	Ack packet bitmap cache map	
1800	RDT interface command/status	
2000	Jump table mask bit	
2001	Jump table priority bit	
2002	Interrupt mask bit	
2003	Interrupt enable	
3000	Jump table result 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	

図 5: Internal Memory のアドレスマップ

- Extension bit は，256 クラスタ (8bit) に対応した Source node ID を 1024 クラスタ (10bit) に拡張するためのビットである.
- Source node ID は，256 クラスタ (8bit) のクラスタ番号を示す.
- Version number(5bit) は，あるクラスタからのマルチキャストを区別するためのバージョン番号である.

ACK 収集用キャッシュへの登録はハードウェアで自動で行うことができるが，キャッシュがあふれた場合は MBP Core で ACK の収集を行わなければならない. また，RDT i/f に対するコマンドレジスタを用いて，ACK 収集をハードウェアで行うか行わないかの指定を行うことが出来る.
このキャッシュは最上位に ACK か NACK かを示すビットをもち (1:ACK,0:NACK)、下位 4 ビットはカウンタの値を格納するためのビットで，マルチキャスト時にエントリを生成する際にいくつの Ack パケットが返送されるかを予測し (bitmap の 1 の数)，その数をカウンタフィールドに登録しておく. Ack パケットが返送されるたびにそのカウンタのフィールドがデクリメントされ，0 になったときに MBP Core の受信バッファに Ack パケットが格納される. もし，0 になった際にそのクラスタがマルチキャスト開始ノードでなかった場合には，自動的に次の上位ノードに対してルーティングされる.

```
ACK combining buffer
Ack/Nack  1 | Tag  9  | Version  4
```

3.0.1 ACK 緊急バッファ

ACK 緊急バッファはパケットの送受信時の緊急事態の場合に対処するためのバッファである。緊急事態とは、ACK の自動生成を行い RDT に送信する際に RDT への送信バッファがいっぱいであった場合と、到着したパケットを ACK 収集バッファに登録しようとしてエントリがいっぱいであった場合である。この様な場合には、MBP Core でソフトウェアによって対処しなければならず、MBP へのパケット送信用のバッファがこの ACK 緊急バッファである。このバッファは RDT の 1 パケット分の容量しかもたない。また、このバッファを clear するための MBP Core の命令が用意されている。
ACK 緊急バッファのデータ構造は、下のような構成である。

```
1. ACK/NOR | Upper full | Zero full | Send full | 0000...0

if ACK/NOR = 0
2,3,4 <= Ack packet (LSB 16bit)

else
2. 001100000 | Rank R B-map Rank R A-map
3. Compressed Address
4. 0 | DST | CUT | URR | ROR | 0000 | R-1 Bmap R-1 Amap
```

3.0.2 NET キャッシュ

ここでは、NET とは RDT ルータからの shootdown もしくは ACK,NACK 自動生成のことをいう。
ACK，NACK 自動生成を行う際にはあらかじめ MBP Core で NET キャッシュに登録しておく必要があ

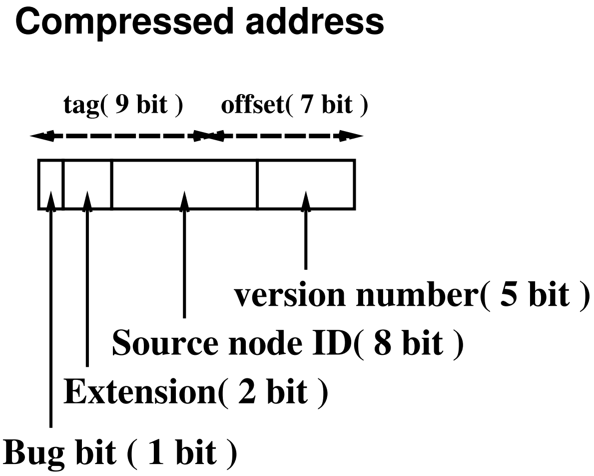

図 6: Compressed Address の構成

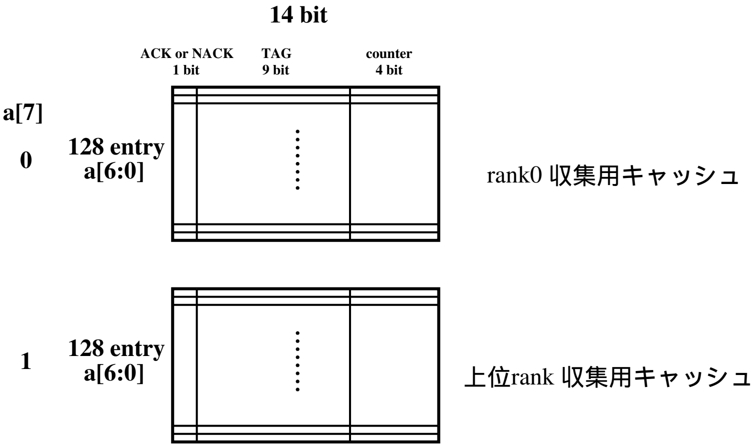

図 7: ACK 収集用キャッシュの構成

る．NACKを自動生成した場合にはCoreにはなにも知らされないが，ACKを生成した場合には到着したマルチキャストパケットにNet infoを付加して受信バッファに格納する．Net infoは，RDTのマルチキャストパケットの3フリット目の4 ~ 7ビット目を用いる．Net infoのフィールドは図8のようになっている．

NETキャッシュには、Cluster Addressの上位9bitをキャッシュアドレスとして用い、下位の14bitをtagとして格納する。また、NETキャッシュの下位3bitは、ネットキャッシュエントリがvalidかどうか(0bit目)、パケットを廃棄するかどうか(1bit目)(0で廃棄する)、Ack/Nack(1でAck)を示すビット(2bit目)，拡張用(3bit目)にそれぞれ用いられる。Cluster Addressは10bitのソースノード番号が付加されて33bitの情報として、RDTパケットの5フリット目に格納される。

また，NETキャッシュによりAck/Nackの自動生成を行う場合には，Ack map cacheのエントリにその宛先までのbitmapを格納しておく必要がある．NETキャッシュのエントリとAck map cacheの両方のエントリがvalidであった場合にAck/Nackが自動生成される．Ack map cacheのエントリ数は256で，256クラスタのシステムであれば追い出しは起こり得ない．

```
Net cache tag
Extension 1 | Ack/Nack 1 | packet discard 1 | Net cache valid 1

ACK packet bit map cache tag
Source address 2 | Valid bit 1

ACK packet bit map cache Map
Root rank 2|Dest. rank 2|M3 3|M2 3|M1 3|M0 3|
```

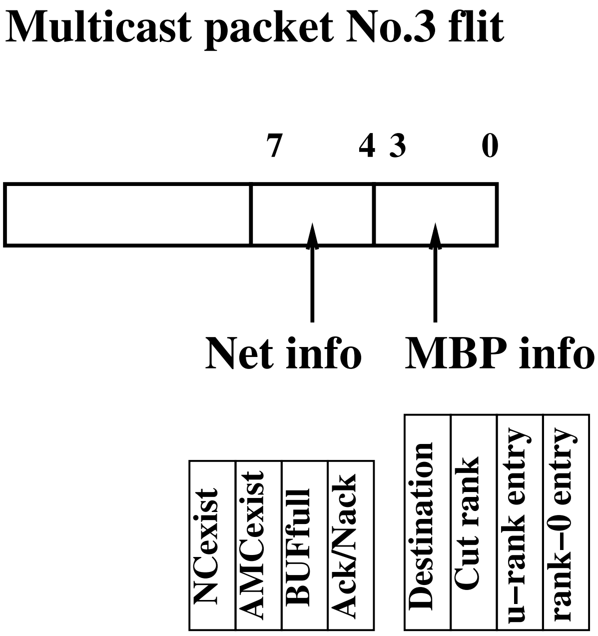

図8: Net info, MBP infoフィールド

一方，shootdownモードになった場合、NETキャッシュの中身を上書きする形で到着したshootdownパケットを上から順に格納していく。到着するパケットの構成を図11に示す．逆にRDTにsetupパケット

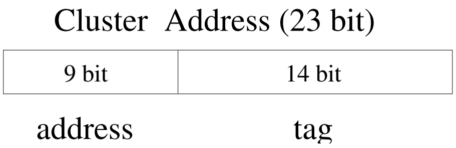

図 9: Cluster Address の構成

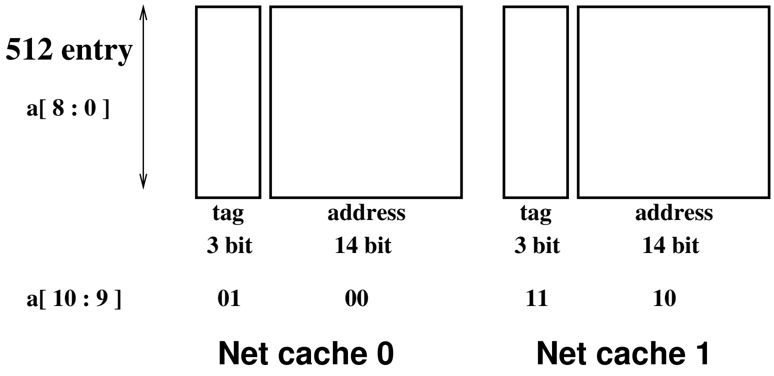

図 10: Net cache の構成

を送信する場合は通常のパケットと同様にして RDT に送信する。shootdown されてきたパケットはそのまま図 11の様に Net cache に取り込まれる. setup を行う際には，Net cache の内容をそのままのパケットの形式で RDT ルータに対して送信する必要がある.

3.0.3 RDT i/f コマンドレジスタ

RDT インタフェースに対してコマンドを指定することができ，またステータスも読み出すことができる.

```
RDT controller command/status
d: bufrempty  (R)
c: bufsfull  (R)
b: shmode  (R)
a: a2mf          (R)
9: spe  (R)
8: perr  (R)
4: parity enable (W)
3,2: crank  (W)
1: autnet  (W)
0: autack  (W)
```

3.0.4 Jump Table

Jump Table は RDT からのパケットのビットを判定して、MBP Core で必要なルーチンにジャンプするためのテーブルである。また，テーブルジャンプの優先順位を指定することができる. ジャンプテーブル

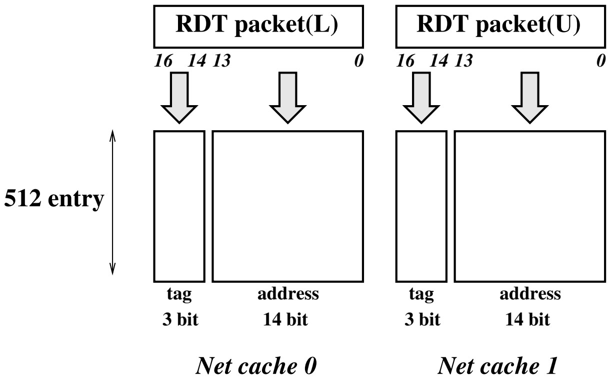

図 11: shootdown 時のパケットと Net cache の関係

の飛び先に関しては、ユーザマニュアルの項を参照のこと。

3.0.5 MMC

MMC のレジスタに関する詳しい内容は佐藤さんの MMC 仕様書を参照のこと。

```
---Address registers of CBUS pakcet buffers
9000 RECV_ADR_REG0 (68bits)
9001 RECV_ADR_REG1 (68bits)
9002 RECV_ADR_REG2 (68bits)
9003 RECV_ADR_REG3 (68bits)
9004 RECV_ADR_REG4 (68bits)
9005 RECV_ADR_REG5 (68bits)
9006 RECV_ADR_REG6 (68bits)
9007 RECV_ADR_REG7 (68bits)
--- Buffer and Controller Status ---
9008 RECV_BUF_STATE (9bits)
9009 SEND_BUF_STATE (6bits)
900a MTC_REQ_STATE (9bits)
--- Interrupt (Read Only)
900b CBC_INT_PKT_SRC + OVF + UDF  / 11 UDF, 10 OVF, 9-0 INT_PKT_SRC
900c CBC_PEND_DOWN (8bit: Write only)
900d CBC_ADR_CMP_MODE (3bit)
900e CBC_IRR_REG (1bit)
900f -
--- Pending
9010 CBC_PEND_L_BMP
9011 CBC_PEND_L_MASK
9012 CBC_PEND_L_ACT
9013 -
9014 CBC_PEND_COUNTER0
```

```
9015 CBC_PEND_COUNTER1
9016 CBC_PEND_COUNTER2
9017 CBC_PEND_COUNTER3
-- MTC
9018 MTC_PROC_NUM
9019 MTC_DECE_REG
901a MTC_INT 8: Unknown Packet, 7-6: DCTL_ERROR2_NUM, 5-4: DCTL_ERROR_NUM,
                3: EMMTRAP, 2: DCTL_ERROR2, 1: DCTL_ERROR, 0:MTC_INTERRUPT
-- Refresh
901b REF_CMP
901c CBC_LOCK_REG
-- Lock
9020 CBC_LOCK_BMP0
9021 CBC_LOCK_MASK0
9022 CBC_LOCK_ACT0
9023 -
9024 CBC_LOCK_BMP1
9025 CBC_LOCK_MASK1
9026 CBC_LOCK_ACT1
9027 -
9028 CBC_LOCK_BMP2
9029 CBC_LOCK_MASK2
902a CBC_LOCK_ACT2
902b -
902c CBC_LOCK_BMP3
902d CBC_LOCK_MASK3
902e CBC_LOCK_ACT3
902f -
```

4 おわりに

最後のいいわけ：　本マニュアルの後半の内部アドレスに関しては、MMC のレジスタのアクセスについてはMMC 仕様書を参照されたい。これは佐藤さんが作ったのに、新しい部分を付け足せば多分理解できると思う。（まとめてくれると嬉しいが）RDT 周辺に関してはRDT インタフェース仕様書がないと理解が難しいと思うが、これは結構しんどそうなので、もうちょっと待っていただきたい。また、この仕様書は、MBP-light 自体のハードウェア構成（例えばパイプライン構成など）は書いていないので、これもそのうち書こうと思う。

MBP-light の設計思想は、「そーいうことにしておきな。角度変えれば、またいい感じ」というものである。MBP-light は、Magic やSCLIC チップに比べ、(1) 設計期間は一年、(2) 利用ゲート数は20 万ゲート程度、という圧倒的な制約があった。それにもかかわらず、応答メッセージの返答、応答メッセージの収集等、さぼると確実に遅くなる部分は既に明らかになっており、その辺の部分はハードウェア化せざるを得なかった。Buffer-Register Architecture にしても、少ないゲート数を効率的に用いて、しかも性能を落さないために生まれた方法で、名前は後で付けた。全体として圧倒的な制約の元で、まずなんとかなる方

法を探した後に、論文を書く時に設計思想をくっつけたものである。つまりMBP-lightは徹底的に無思想なので、楽に性能を上げることができるアイディア、性能を損なわず簡単化するアイディアはなんであれ歓迎である。必要があってどうしてもこうして欲しい、という要求も、納得できるならば、ゲート数とクリティカルパスが許す限り努力したい。しかし、格好悪いから、思想的に合わないから、ということではあまり変えたくない。変えればうんと格好良くなって論文も通る、、、というなら別だが、、、