お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

User's Manual

32

RENESAS

改訂一覧は表紙をクリックして直接ご覧になれます。
改訂一覧は改訂箇所をまとめたものであり、詳細については、
必ず本文の内容をご確認ください。

SH-1/SH-2/SH-DSP

ソフトウェアマニュアル

ルネサス32ビットRISCマイクロコンピュータ
SuperH™ RISC engineファミリ

ルネサス エレクトロニクス
www.renesas.com

Rev.7.00　2005.01

安全設計に関するお願い

1. 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

1. 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
2. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
3. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ (http://www.renesas.com)などを通じて公開される情報に常にご注意ください。
4. 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
5. 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任は負いません。
6. 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
7. 本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
8. 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。

製品に関する一般的注意事項

1. NC 端子の処理

   【注意】NC端子には、何も接続しないようにしてください。

   NC(Non-Connection)端子は、内部回路に接続しない場合の他、テスト用端子やノイズ軽減などの目的で使用します。このため、NC端子には、何も接続しないようにしてください。

2. 未使用入力端子の処理

   【注意】未使用の入力端子は、ハイまたはローレベルに固定してください。

   CMOS製品の入力端子は、一般にハイインピーダンス入力となっています。未使用端子を開放状態で動作させると、周辺ノイズの誘導により中間レベルが発生し、内部で貫通電流が流れて誤動作を起こす恐れがあります。未使用の入力端子は、入力をプルアップかプルダウンによって、ハイまたはローレベルに固定してください。

3. 初期化前の処置

   【注意】電源投入時は，製品の状態は不定です。

   すべての電源に電圧が印加され、リセット端子にローレベルが入力されるまでの間、内部回路は不確定であり、レジスタの設定や各端子の出力状態は不定となります。この不定状態によってシステムが誤動作を起こさないようにシステム設計を行ってください。リセット機能を持つ製品は、電源投入後は、まずリセット動作を実行してください。

4. 未定義・リザーブアドレスのアクセス禁止

   【注意】未定義・リザーブアドレスのアクセスを禁止します。

   未定義・リザーブアドレスは、将来の機能拡張用の他、テスト用レジスタなどが割り付けられています。これらのレジスタをアクセスしたときの動作および継続する動作については、保証できませんので、アクセスしないようにしてください。

本版で改訂された箇所

修正項目	頁	修正内容（詳細はマニュアル参照）
全体	–	社名変更に伴い、マニュアル本文中の「日立製作所」、「株式会社日立製作所」、「日立半導体」、「日立 XX」という表記をすべて「株式会社ルネサステクノロジ」に変更。 カテゴリー呼称の「シリーズ」を「グループ」に変更。

はじめに

SuperH RISC engine ファミリの CPU は RISC（Reduced Instruction Set Computer）タイプの CPU です。基本命令は 1 命令 1 ステートで動作し、高性能な演算処理を実現しています。また、乗算器を内蔵し、汎用 DSP（Digital Signal Processor：デジタル信号プロセッサ）と同等の乗算・積和演算が可能です。

ルネサス SuperH RISC engine（以下 SuperH と略します）ファミリには、SH-1、SH-2、SH-3、SH-DSP、SH3-DSP、SH-4 の CPU コアがあります。

SH-1、SH-2、SH-3、SH-4 の各 CPU はバイナリレベルで上位互換の命令体系を持っています。

SH-DSP は、SH-2 CPU をベースにした、汎用 DSP なみの信号処理性能を重視した 32 ビット RISC マイクロコントローラです。SH-DSP は、SH-2 CPU をベースにした、汎用 DSP 並みの信号処理性能を重視した 32 ビット RISC マイクロコントローラです。SH-DSP は、SuperH RISC engine にある乗算と積和演算の DSP 機能を強化したもので、DSP タイプのデータパス機能を実現しています。SH-DSP は、SH-1、SH-2 マイコンとバイナリレベルの上位互換性を持っています（**図 1**　参照）。

このプログラミングマニュアルは、SH-1、SH-2、SH-DSP の基本的なアーキテクチャと命令の詳細について記載しています。アーキテクチャや命令の動作を知るためにお使いください。SuperH RISC engine の特長であるパイプラインの動作についても述べてあります。

SH-3、SH3-DSP、SH-4 に関しては、別冊のプログラミングマニュアルを参照してください。

ハードウェアについては各製品別のハードウェアマニュアルをご覧ください。

開発環境システムについては、当社営業所までお問い合わせください。

SuperH の概略

SuperH RISC engine ファミリの命令形態を**図 1** に示します。

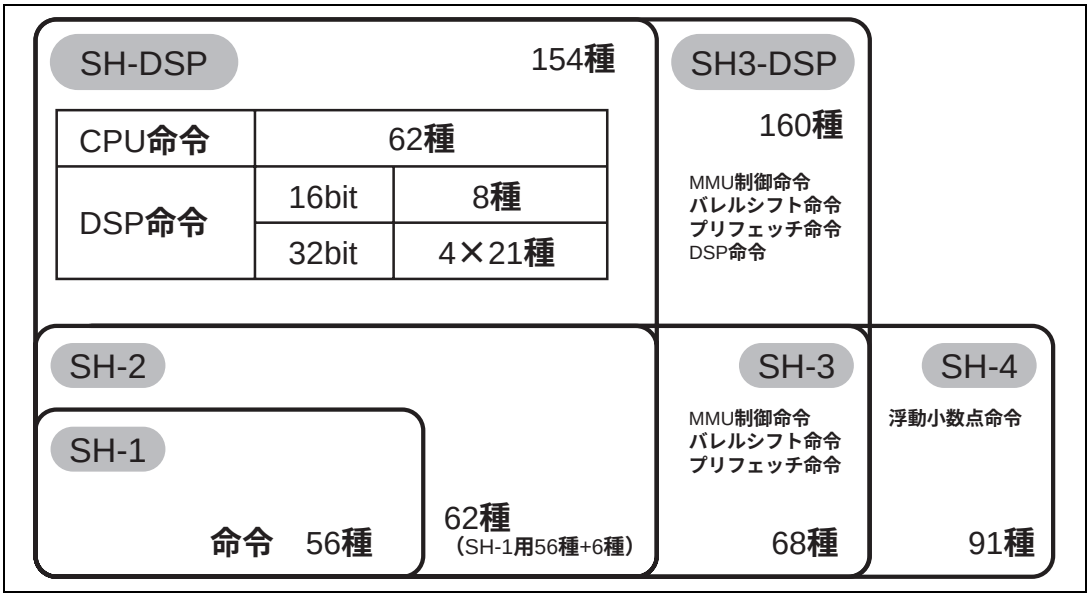


図 1. SuperH RISC engine **ファミリの命令形態**

マニュアルの構成

このマニュアルの構成を**表 1** に示します。

表 1　マニュアルの構成

区分	章名	内容
概要	1.　**特長**	SuperH **の** CPU **と** DSP **の特長**
アーキテクチャ	2.　**レジスタ構成**	**汎用レジスタ、コントロールレジスタ、システムレジスタの種類と構成、**DSP **レジスタの種類と構成**
	3.　**データ形式**	**レジスタとメモリ上のデータ形式、**DSP **のデータ形式**
命令の概要	4.　**命令の特長**	CPU **命令と** DSP **命令の特長** **・アドレッシングモード** **・命令形式** **・演算機能とデータ転送**
	5.　**命令セット**	**分類順の命令概要、各命令の動作説明**
命令の詳細	6.　**各命令の説明**	**書式、動作概略、命令コード、説明、注意、動作内容、使用例**
	7.　**パイプライン動作**	**パイプラインの流れ、各命令の動作とパイプラインの流れ**
付録	SuperH **シリーズの比較**	SH-1、SH-2、SH-DSP **の比較**

目次

1. 特長

1.1 SH-1、SH-2 の特長

SH-1 CPU および SH-2 CPU は、RISC タイプの命令セットを持っており、基本命令は 1 命令 1 ステート（1 システムクロックサイクル）で動作するので、命令実行速度が飛躍的に向上しています。また内部 32 ビット構成を採用しておりデータ処理能力を強化しています。

SH-1 CPU および SH-2 CPU の特長を**表 1.1** に示します。

表 1.1 SH-1 CPU および SH-2 CPU の特長

項目	特長
アーキテクチャ	• ルネサスオリジナルアーキテクチャ • 内部 32 ビット構成
汎用レジスタマシン	• 汎用レジスタ　32 ビット×16 本 • コントロールレジスタ 32 ビット×3 本 • システムレジスタ　32 ビット×4 本
命令セット	• RISC タイプの命令セット – 命令長は 16 ビット固定長、これによるコード効率の向上 – ロードストアアーキテクチャ（基本演算はレジスタ間で実行） – 遅延分岐方式の採用で、分岐時のパイプラインの乱れを軽減 – C 言語指向の命令セット
命令実行時間	• 基本命令は 1 命令／1 ステート（1 命令／1 システムクロックサイクル）
アドレス空間	• アーキテクチャ上は 4GB
乗算器内蔵（SH-1 CPU）	• 乗算器内蔵により、16×16→32 の乗算を 1～3*ステートで実行、 16×16＋42→42 の積和演算を 3/(2)*ステートで実行
乗算器内蔵（SH-2 CPU）	• 乗算器内蔵により、16×16→32 の乗算を 1～3 ステートで実行、 16×16＋64→64 の積和演算を 3/(2)*ステートで実行、 32×32→64 の乗算を 2～4*ステートで実行、 32×32＋64→64 の積和演算を 3/(2～4)*ステートで実行
パイプライン	• 5 段パイプライン方式
処理状態	• プログラム実行状態 • 例外処理状態 • バス権解放状態 • リセット状態 • 低消費電力状態
低消費電力状態	• スリープモード • スタンバイモード

【注】 * 通常実行ステートを示します。（ ）内の値は前後の命令との競合関係による実行ステートです。

1.2 SH-DSP の特長

SH-DSP は、SH-2 CPU をベースにし、汎用 DSP なみの信号処理性能を実現した 32 ビットマイクロコントローラです。 SuperH には、すでに乗算と積和演算の DSP タイプの命令があります。SH-DSP は、 SuperH にある DSP 機能を強化したもので、DSP タイプの完全なデータパス機能を実現しています。SH-DSP は、SH-1、SH-2 CPU とオブジェクトコードレベルの上位互換性を持っています。

SH-1、SH-2 は、16 ビット長の命令のみを持っています。SH-DSP は基本的には同じ 16 ビット長の命令を持ち、DSP タイプの命令を並行処理するために、32 ビット長の DSP タイプの命令が追加されています。 SuperH は標準のノイマン型アーキテクチャですが、SH-DSP は拡張ハーバード型アーキテクチャの DSP データパスを持っています。

SH-DSP に追加された特長を**表** 1.2 に示します。

表 1.2 SH-DSP **の追加された特長**

項目	特長
DSP **ユニット**	• **1サイクル乗算器** • 16 **ビット×**16 **ビット→**32 **ビット（符号付き固定小数点）** • **算術演算器（**ALU：Arithmetic Logic Unit**）** • **バレルシフタ** • DSP **レジスタ** • MSB **検知**
DSP **レジスタ**	• 40 **ビットデータレジスタ×**2 **本** • 32 **ビットデータレジスタ×**6 **本** • DSP **ステータスレジスタ（**DSR**）** • **モジュロレジスタ（**MOD、32 **ビット）をコントロールレジスタに追加** • **リピートカウンタ（**RC**）、モジュロアドレッシング指定（**DMX、DMY**）およびリピートフラグ（**RF1、RF0**）をステータスレジスタ（**SR**）に追加** • **繰り返し開始レジスタ（**RS、32 **ビット）、繰り返し終了レジスタ（**RE、32 **ビット）をコントロールレジスタに追加**
DSP **データパス**	• **拡張ハーバード型アーキテクチャ** • 2 **つのデータバスおよび** 1 **つの命令バスを同時にアクセス**
並列処理	• **最大** 4 **つの並列処理** • ALU **演算、乗算、および** 2 **つのロードまたはストア**
アドレス演算器	• 2 **つのアドレス演算器** • 2 **つのメモリをアクセスするためのアドレス演算**
DSP **データ** **アドレッシングモード**	• **インクリメント、デクリメントおよびインデクス** • **それぞれモジュロアドレッシング付きまたはなし**
繰り返し制御	• **ゼロオーバヘッド繰り返し（ループ）制御**
命令セット	• 16 **ビット長または** 32 **ビット長** – 16 **ビット長（ロードまたはストアだけの場合）** – 32 **ビット長（**ALU **演算、乗算を含む場合）** • DSP **レジスタをアクセスする** SH **マイコン命令を追加**
パイプライン	• 5 **段パイプライン方式** • **最後の第** 5 **ステージは** WB **ステージと** DSP **ステージ兼用**

2. レジスタ構成

SH-1、SH-2 には汎用レジスタ（32 ビット×16 本）、コントロールレジスタ（32 ビット×3 本）、システムレジスタ（32 ビット×4 本）があります。

SH-DSP は SH-1、SH-2 とオブジェクトコードレベルで上位互換性があります。そのため SH-DSP には SH-1、SH-2 と同じレジスタがあり、そのほかにいくつかのレジスタが追加されています。追加されたのは、コントロールレジスタの繰り返し開始レジスタ（RS）、繰り返し終了レジスタ（RE）、モジュロレジスタ（MOD）の 3 本と、システムレジスタの DSP ステータスレジスタ（DSR）、DSP データレジスタの内の A0、A1、X0、X1、Y0、Y1、M0、M1 の 8 本です。

SH-DSP の汎用レジスタは、CPU タイプの命令では、SH-1、SH-2 と同じように使われます。これに対して DSP タイプの命令では、メモリをアクセスするためのアドレスレジスタ、インデックスレジスタとして使われます。

2.1 汎用レジスタ

SH-1、SH-2、SH-DSP の汎用レジスタ（Rn）は、32 ビットの長さで、R0 から R15 までの 16 本あります。汎用レジスタは、データ処理、アドレス計算に使われます。R0 は、インデックスレジスタとしても使用します。いくつかの命令では使用できるレジスタが R0 に固定されています。 R15 は、ハードウェアスタックポインタ（SP）として使われます。例外処理でのステータスレジスタ（SR）とプログラムカウンタ（PC）の退避、回復は R15 を用いてスタックを参照し行います。

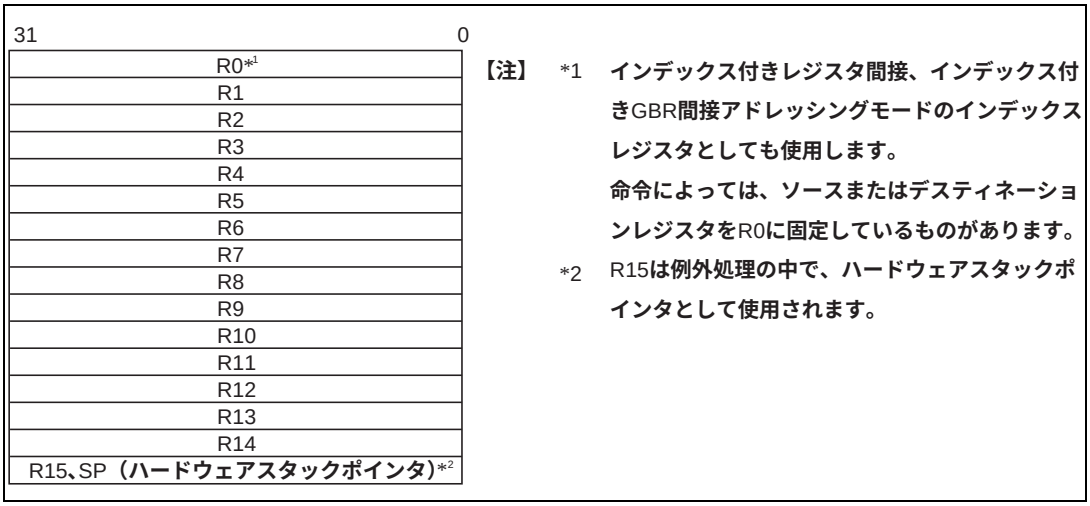


図 2.1 汎用レジスタの構成（SH-1/SH-2）

SH-DSP では DSP タイプの命令で、16 の汎用レジスタの内の 8 つのレジスタを X、Y データメモリおよびメインバスを使うデータメモリ（シングルデータ）のアドレッシングに使用します。

X メモリをアクセスするためには、X アドレスレジスタ［Ax］として R4、R5 を使い、X インデックスレジスタ［Ix］として R8 を使います。Y メモリをアクセスするためには、Y アドレスレジスタ［Ay］として R6、R7 を使い、Y インデックスレジスタ［Iy］として R9 を使います。メインバスを使ってシングルデータをアクセスするためには、シングルデータアドレスレジスタ［As］として R2、R3、R4、R5 を使い、シングルデータインデックスレジスタ［Is］として R8 を使います。

DSP タイプの命令は X と Y データメモリを同時にアクセスできます。X と Y データメモリのアドレスを指定するために、2 組のアドレスポインタがあります。

SH-DSP の汎用レジスタの構成を**図** 2.2 に示します。

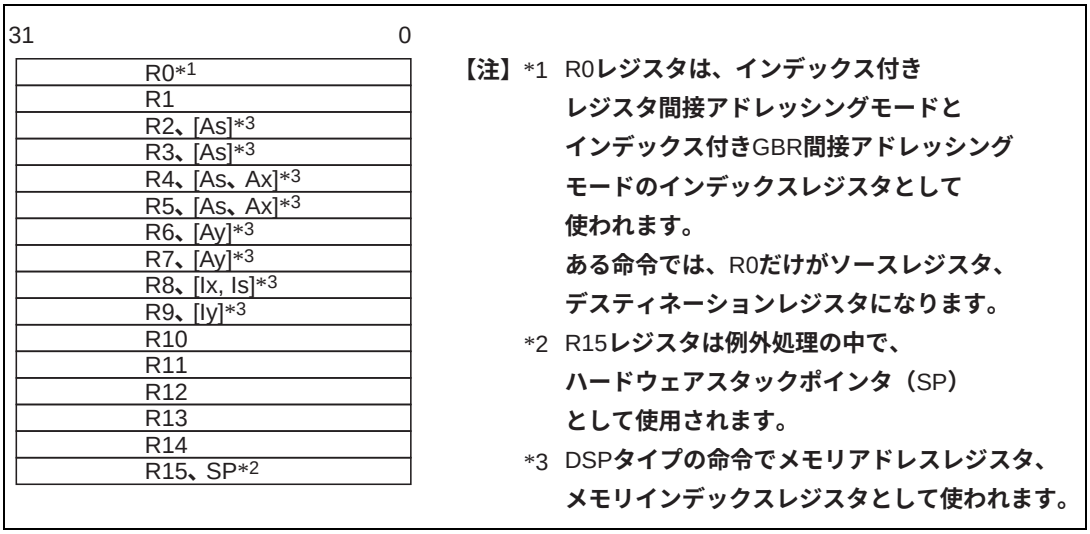


図 2.2　**汎用レジスタの構成**（SH-DSP）

アセンブラでは R2、R3、．．．、R9 の記号名（シンボル）を使います。もし DSP タイプの命令のためのレジスタの役割を明示した名前にしたいときは、レジスタの別名（エイリアス、alias）を使います。アセンブラで次のように書きます。

```
Ix:   .REG   (R8)
```

名前 Ix が R8 の別名になります。そのほか次のように別名を付けます。

```
Ax0: .REG   (R4)
Ax1: .REG   (R5)
Ix:   .REG   (R8)
Ay0: .REG   (R6)
Ay1: .REG   (R7)
Iy:   .REG   (R9)
As0: .REG   (R4);これはシングルデータ転送のために別名が必要なときの定義です。
As1: .REG   (R5);これはシングルデータ転送のために別名が必要なときの定義です。
As2: .REG   (R2);これはシングルデータ転送のために別名が必要なときの定義です。
As3: .REG   (R3);これはシングルデータ転送のために別名が必要なときの定義です。
Is:   .REG   (R8);これはシングルデータ転送のために別名が必要なときの定義です。
```

2.2 コントロールレジスタ

コントロールレジスタは 32 ビットの長さで、ステータスレジスタ(SR: Status register) 、グローバルベースレジスタ(GBR: Global base register)、ベクタベースレジスタ(VBR: Vector base register)の 3 本があります。

SR レジスタは各種命令の処理の状態を表します。

GBR レジスタは GBR 間接アドレッシングモードのベースアドレスとして使用し、内蔵周辺モジュールのレジスタのデータ転送などに使用します。GBR 間接アドレッシングモードは内蔵周辺モジュールのレジスタ領域のデータ転送および論理演算に使われます。

VBR レジスタは割り込みを含む例外処理ベクタ領域のベースアドレスとして使用します。

図 2.3 に SH-1,SH-2 のコントロールレジスタを示します。

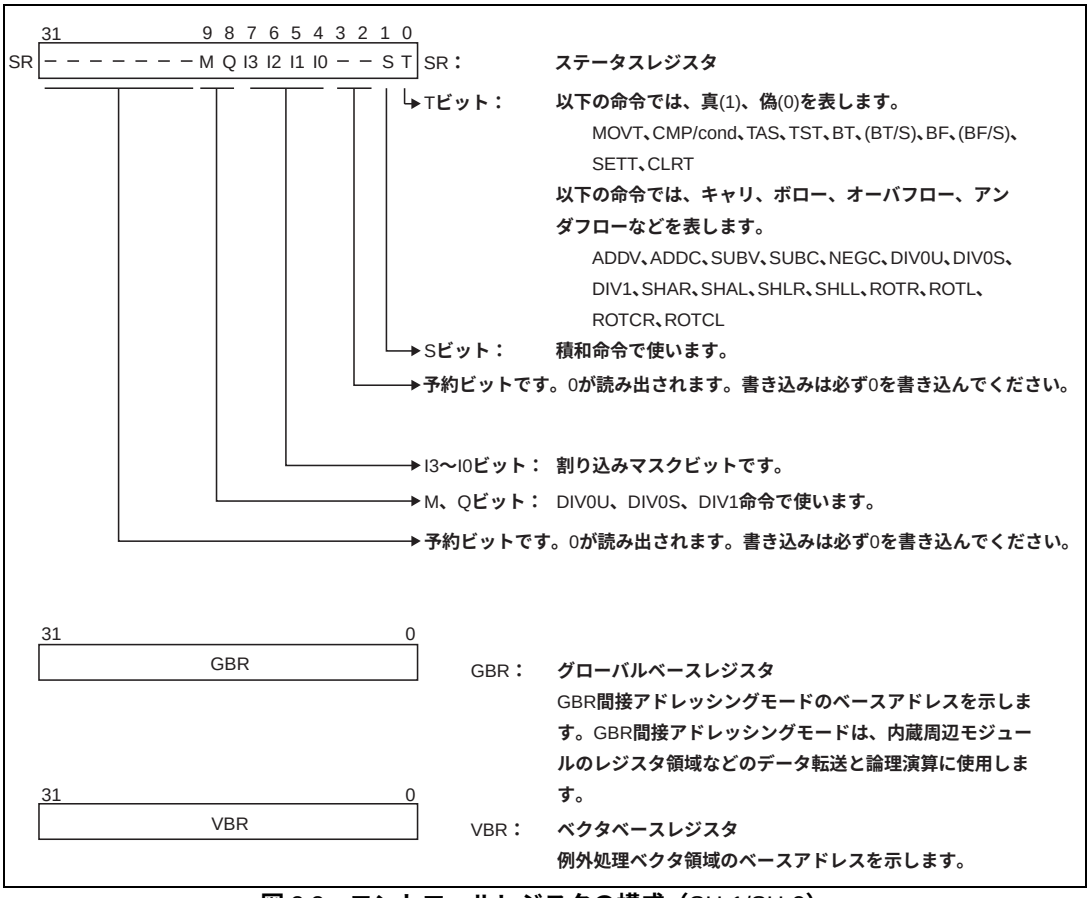


図 2.3 **コントロールレジスタの構成**（SH-1/SH-2）

SH-DSP では、繰り返し開始レジスタ（RS: Repeat start register）、繰り返し終了レジスタ（RE: Repeat end register）、モジュロレジスタ（MOD: Modulo register）の 3 本が追加されています。そのため、SR レジスタにも新たに 4 つの制御ビット（DMX ビット、DMY ビット、RF1 ビット、RF0 ビット）と 12 ビットの RC カウンタが追加されています。

RS レジスタと RE レジスタはプログラムの繰り返し（ループ）を制御するために使います。SR レジスタの繰り返しカウンタ（RC: Repeat counter）に繰り返し回数を指定し、RS レジスタに繰り返し開始アドレスを指定し、RE レジスタに繰り返し終了アドレスを指定します。ただし、RS レジスタと RE レジスタに格納されるアドレスの値は、繰り返しの物理的な開始アドレス、終了アドレスとは値が必ずしも同じとは限りません。

MOD レジスタは繰り返しデータのバッファリングのためのモジュロアドレッシングに使います。DMX ビットまたは DMY ビットでモジュロアドレッシングの指定をし、MOD レジスタの上位 16 ビットにモジュロ終了アドレス（ME）を指定し、下位 16 ビットにモジュロ開始アドレス（MS）を指定します。なお、DMX と DMY ビットは同時にモジュロアドレッシングを指定することはできません。モジュロアドレッシングは X、Y データ転送命令（MOVX、MOVY）のとき可能です。シングルデータ転送命令（MOVS）ではできません。

図 2.4 に SH-DSP で追加されたコントロールレジスタとステータスレジスタのビット構成を示します。**表** 2.1 に SR レジスタのビットを示します。

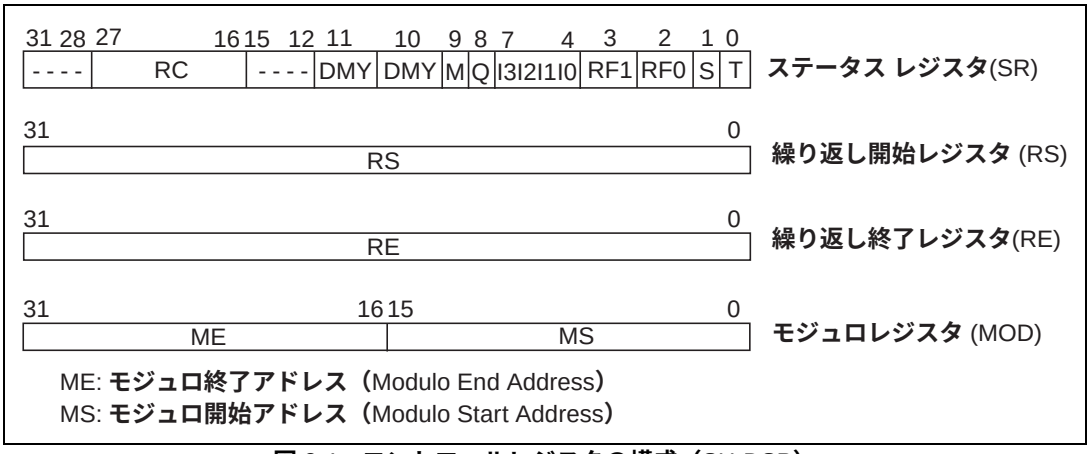


図 2.4　**コントロールレジスタの構成（**SH-DSP**）**

表 2.1 SR レジスタのビット

ビット	名称（略称）	機能
27～16	リピートカウンタ（RC）*	繰り返し（ループ）制御の繰り返し回数を指定します（2～4095）。
11	Y ポインタ用モジュロアドレッシング指定（DMY）*	1：Y メモリアドレスポインタ、Ay（R6、R7）に対し、モジュロアドレシングモードが有効になります。
10	X ポインタ用モジュロアドレッシング指定（DMX）*	1：メモリアドレスポインタ、Ax（R4、R5）に対し、モジュロアドレシングモードが有効になります。
9	M ビット	DIV0S/U、DIV1 命令で使用します。
8	Q ビット	
7～4	割り込み要求マスク（I3～I0）	割り込み要求を受け付けるレベルを表します（0～15）。
3～2	リピートフラグ（RF1、0）*	ゼロオーバヘッド繰り返し（ループ）制御に使用します。 00：1 ステップリピート 01：2 ステップリピート 11：3 ステップリピート 10：4 ステップ以上のリピート
1	飽和演算ビット（S）	MAC 命令および DSP 命令で使用します。 1：飽和演算を指定します（オーバフローを防止します）。
0	T ビット	MOVT、CMP/cond、TAS、TST、BT、BF、SETT、CLRT、および DT 命令のとき 0：偽を表します。 1：真を表します。
		ADDV/C、SUBV/C、DIV0U/S、DIV1、NEGC、SHAR/L、SHLR/L、ROTR/L、および ROTCR/L 命令のとき 1：キャリ、ボロー、オーバフローまたはアンダフローの発生を表します。
31～28 15～12	予約ビット	0：常に 0 が読み出されます。 書き込みは必ず 0 を書き込んでください。

【注】 * SH-DSP のみ

RS、RE、MOD レジスタをアクセスするため専用のロード／ストア命令があります。たとえば RS レジスタをアクセスするときは次のようになります。

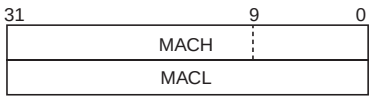
```
LDC    Rm,RS;    Rm   → RS
LDC.L  @Rm+,RS; (Rm)  → RS, Rm+4 →  Rm
STC    RS,Rn;    RS   → Rn
STC.L  RS,@-Rn;  Rn-4 → Rn, RS   → (Rn)
```

ゼロオーバヘッド繰り返し制御のために RS、RE レジスタにアドレスを設定する命令は次のとおりです。

```
LDRS   @(disp,PC); disp×2 + PC → RS
LDRE   @(disp,PC); disp×2 + PC → RE
```

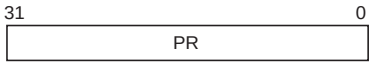
2.3 システムレジスタ

システムレジスタは 32 ビットの長さで、積和レジスタ(MACH: Multiply and accumulate register high、MACL: Multiply and accumulate register low)、プロシージャレジスタ(PR: Procedure register)、プログラムカウンタ(PC: Program counter)の計 4 本があります。MACH, MACL レジスタは乗算または積和演算の結果を格納します*。PR レジスタはサブルーチンプロシージャからの戻り先アドレスを格納します。PC カウンタは実行中のプログラムのアドレスを示し、処理の流れを制御します。PC カウンタは現在実行中の命令の 4 バイト先を示しています。



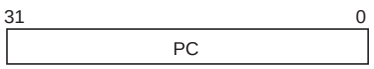
MACH*、MACL：**積和レジスタ（MAC）**

乗算、積和演算の結果の格納レジスタです。SH-2 CPU、SH-DSP CPU**ではMACHは32ビット有効です。**SH-1 CPU**ではMACHは下位10ビットが有効であり、読み出すときは32ビット長まで符号拡張されます。**



PR：**プロシージャレジスタ**

サブルーチンプロシージャからの戻り先アドレスの格納レジスタです。



PC：**プログラムカウンタ**

PC**は現在実行中の命令の４バイト（２命令）先を示しています。**

図 2.5 **システムレジスタの構成**

【注】 * SH-DSP **では、**SH-1**、**SH-2 **でサポートされていた命令実行時にのみ使用されます。**
SH-DSP **で新たに追加された乗算命令（**PMULS**）では使用しません。**

SH-DSP ではさらに 6 本のレジスタがシステムレジスタとして扱われます。**「**2.4 DSP **レジスタ」**で述べる DSP ユニット用のレジスタ（DSP レジスタ）の内、DSP ステータスレジスタ（DSR）、および 8 本のデータレジスタのうちの 5 本（A0、X0、X1、Y0、Y1）です。このうち A0 レジスタは 40 ビットレジスタですが、A0 レジスタからデータを出力する場合はガードビット部分（A0G）は無視され、A0 レジスタにデータを入力する場合はデータの MSB がガードビット部分（A0G）にコピーされます。

2.4 DSP **レジスタ**

SH-DSP の DSP ユニットには DSP レジスタとして 8 つのデータレジスタと 1 つのコントロールレジスタがあります。

DSP データレジスタは 2 本の 40 ビット長の A0、A1 レジスタと、6 本の 32 ビット長の M0、M1、X0、X1、Y0、Y1 レジスタがあります。A0、A1 レジスタには、それぞれ 8 ビットのガードビット、A0G、A1G があります。

DSP データレジスタは、DSP 命令のオペランドとして DSP データのデータ転送、データ処理に使われます。DSP データレジスタをアクセスする命令には、DSP データ処理、X、Y データ転送処理、の 3 つのタイプがあります。

コントロールレジスタは 32 ビット長の DSP ステータスレジスタ（DSR: DSP Status Register）で、演算結果を表します。DSR レジスタには演算結果を表すビット、符号付き大ビット（GT: signed Greater Than）、ゼロビット（Z: Zero value）、負値ビット（N: Negative value）、オーバフロービット（V: overflow）、DSP 状態ビット（DC: DSP Condition）と、DC ビットの設定を制御する状態選択ビット（CS: Condition Select）があります。

DC ビットは状態フラグの一つを表し、SuperH CPU コアの T ビットとよく似ています。条件付き DSP タイプ命令の場合、DSP データ処理は、DC ビットに従って実行が制御されます。この制御は DSP ユニットでの実行にだけ関係し、DSP レジスタだけが更新されます。アドレス計算や、ロード／ストア命令などの SuperH マイコンの CPU コアの実行命令には関係しません。コントロールビット CS（ビット 2 から 0）は DC ビットを設定する状態を指定します。

DSP タイプ命令には、無条件 DSP タイプ命令と条件付き DSP タイプ命令があります。無条件 DSP タイプのデータ処理は、PMULS、MOVX、MOVY、MOVS 命令を除いて、状態ビットと DC ビットを更新します。条件付き DSP タイプ命令は DC ビットの状態によって実行されますが、実行された場合も実行されない場合も DSR レジスタは更新されません。

ただし、A0、X0、X1、Y0、Y1 の 5 本のレジスタは、システムレジスタとしても使用されます。

DSP レジスタを**図** 2.6 に示します。DSR レジスタのビットの機能を**表** 2.2 に示します。

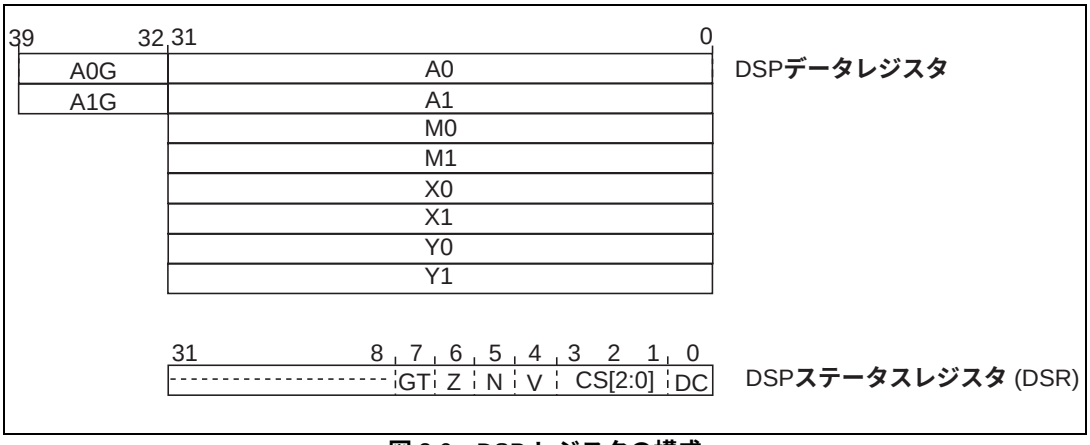


図 2.6 DSP **レジスタの構成**

表 2.2 DSR レジスタのビット

ビット	名称（略称）	機能
31～8	予約ビット	0：常に 0 が読み出されます 書き込みは必ず 0 を書き込んでください。
7	符号付き大ビット（GT）	演算結果が正（ゼロを除く）、またはオペランド 1 がオペランド 2 より大きいことを示します。 1：演算結果が正、またはオペランド 1 がオペランド 2 より大きい
6	ゼロビット（Z）	演算結果がゼロ（0）、またはオペランド 1 がオペランド 2 と等しいことを示します。 1：演算結果がゼロ（0）、または等しい
5	負値ビット（N）	演算結果が負、またはオペランド 1 がオペランド 2 より小さいことを示します。 1：演算結果が負、またはオペランド 1 がオペランド 2 より小さい
4	オーバフロービット（V）	演算結果がオーバフローしたことを示します。 1：演算結果がオーバフロー
3～1	状態選択ビット（CS）	DC ビットに設定する演算結果状態を選択するためのモードを指定します。 110、111 は指定しないでください。 000：キャリ／ボローモード 001：負値モード 010：ゼロ値モード 011：オーバフローモード 100：符号付き大モード 101：符号付き以上モード
0	DSP 状態ビット（DC）	CS ビットで指定されたモードで演算結果の状態を設定します。 0：指定されたモードの状態が成立しない（不成立） 1：指定されたモードの状態が成立

A0、X0、X1、Y0、Y1、DSR レジスタは CPU コア命令ではシステムレジスタとして取り扱われます。

2.5 ガードビットとオーバフローの扱いに関する注意事項

DSP ユニットでのデータ演算は基本的には 32 ビット演算ですが、演算時には、常時 8 ビットのガードビット部分も含めて 40 ビット長で実行されます。ガードビット部分が 32 ビット部分の MSB の値と一致しない場合、演算結果はオーバフローとして扱われます。この場合、N ビットは、オーバフローの有無にかかわらず、演算結果の正しい状態を示します。これはデスティネーションオペランドが 32 ビット長のレジスタであっても同じです。常に 8 ビット分のガードビットが仮定され、各状態フラグがアップデートされます。

ガードビットを使っても正しく結果を表示できないような桁あふれが生じた場合は、N フラグは正しい状態を示すことはできません。詳細は**「**4.8　ALU **固定小数点演算（**3**）**DC **ビット」**を参照してください。

2.6 レジスタの初期値

リセット後のレジスタの値を表 2.3 に示します。

表 2.3 レジスタの初期値

区分	レジスタ	初期値
汎用レジスタ	R0～R14	不定
	R15 (SP)	ベクタアドレステーブル中の SP の値
コントロールレジスタ	SR	• I3～I0 は 1111(H'F)、予約ビットは 0、その他は不定 • RC、DMY、DMX、RF1、RF0 は 0（SH-DSP 追加ビット） SH-1、SH-2、SH-DSP とも H'0000 00F0
	RS RE	不定
	GBR	不定
	VBR	H'0000 0000
	MOD	不定
システムレジスタ	MACH、MACL、PR	不定
	PC	ベクタアドレステーブル中の PC の値
DSP レジスタ	A0、A0G、A1、A1G、M0、M1、X0、X1、Y0、Y1	不定
	DSR	H'0000 0000

3. データ形式

3.1 レジスタのデータ形式

レジスタオペランドのデータサイズは常にロングワード（32 ビット）です。メモリ上のデータをレジスタへロードするとき、メモリオペランドのデータサイズがバイト（8 ビット）、またはワード（16 ビット）の場合は、ロングワードに符号拡張し、レジスタに格納します。

図 3.1 **レジスタのデータ形式**

3.2 メモリ上でのデータ形式

バイト、ワード、ロングワードのデータ形式があります。

バイトデータは任意番地に、ワードデータは 2n 番地から、ロングワードデータは 4n 番地から配置してください。その境界以外からアクセスすると、アドレスエラーが発生します。このとき、アクセスした結果は保証しません。特に、ハードウェアスタックポインタ（SP、R15）が指し示すスタックエリアには、プログラムカウンタ（PC）とステータスレジスタ（SR）をロングワードで格納しますので、ハードウェアスタックポインタの値が必ず 4n になるように設定してください。アドレスエラーについては、各製品別のハードウェアマニュアルを参照してください。

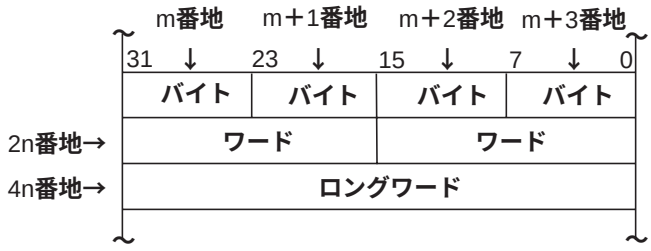


図 3.2 **メモリ上のデータ形式（ビックエンディアン）**

リトルエンディアンの機能を内蔵した製品の場合、以下のようにバイトデータが配置されます。各製品がリトルエンディアンをサポートしているかどうかは、各製品別のハードウェアマニュアルを参照してください。

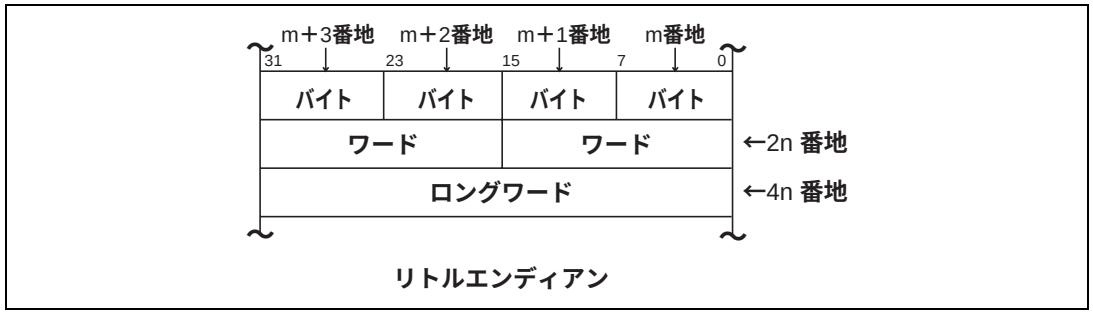


図 3.3 **メモリ上のデータ形式（リトルエンディアン）**

3.3 イミディエイトデータのデータ形式

バイトのイミディエイトデータは命令コードの中に配置します。

MOV、ADD、CMP/EQ 命令ではイミディエイトデータを符号拡張後、レジスタとロングワードで演算します。一方、TST、AND、OR、XOR 命令ではイミディエイトデータをゼロ拡張後、ロングワードで演算します。したがって、AND 命令でイミディエイトデータを用いると、デスティネーションレジスタの上位 24 ビットは常にクリアされます。

ワードとロングワードのイミディエイトデータは命令コードの中に配置せず、メモリ上のテーブルに配置してください。メモリ上のテーブルは、ディスプレースメント付き PC 相対アドレッシングモードを使ったイミディエイトデータのデータ転送命令(MOV)で、参照してください。

具体例については、**「**4. **命令の特長】**の**「**4.1（8）**イミディエイトデータ】**を参照してください。

3.4 DSP タイプデータ形式

SH-DSP には DSP 命令に対応して 3 つの異なるデータ形式があります。固定小数点データ形式、整数データ形式、論理データ形式です。

DSP タイプの固定小数点データ形式はビット 31 とビット 30 の間に 2 進小数点があります。ガードビット付き、ガードビットなし、乗算入力の 3 種類があり、それぞれ有効ビット長と表せる値の範囲が異なります。

DSP タイプの整数データ形式はビット 16 とビット 15 の間に 2 進小数点があります。ガードビット付き、ガードビットなし、シフト量の 3 種類があり、それぞれ有効ビット長と表せる値の範囲が異なります。

算術シフト（PSHA）のシフト量は 7 ビットの領域で－64～＋63 までを表せますが、実際に有効なのは－32～＋32 までの値です。同様に論理シフトのシフト量は 6 ビットの領域ですが、実際に有効なのは－16～＋16 までの値です。

DSP タイプの論理データ形式は小数点がありません。

データ形式とデータの有効な長さは命令と DSP レジスタによって決まります。

3 つの DSP タイプのデータ形式とその 2 進少数点の位置、および参考として CPU タイプのデータ形式を**図** 3.4 に示します。

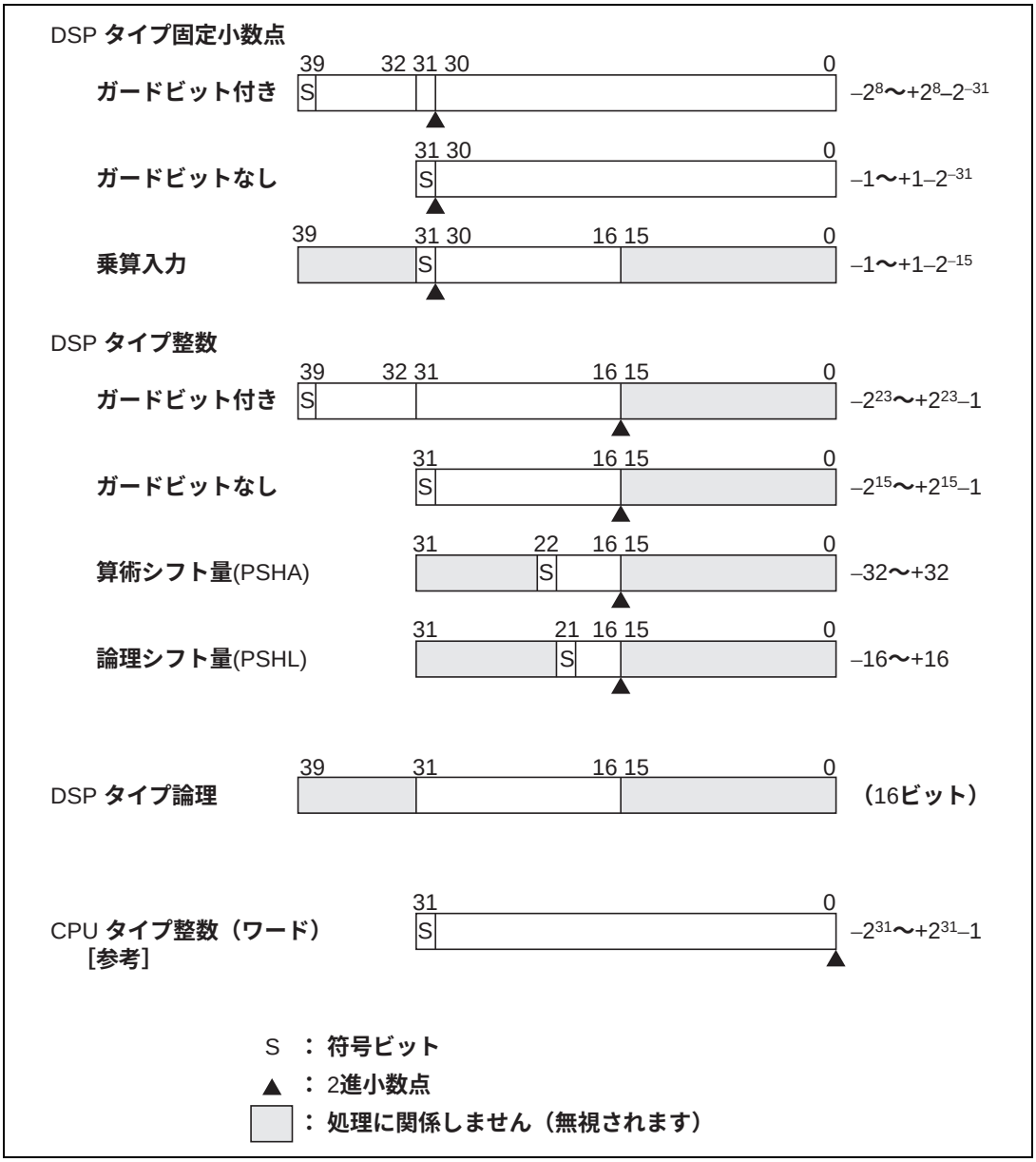


図 3.4 DSP **タイプデータ形式**

3.5 DSP タイプ命令とデータ形式

DSP データ形式とデータの有効な長さは DSP タイプ命令と DSP レジスタによって決まります。DSP データレジスタをアクセスする命令には、DSP データ処理、X、Y データ転送処理、シングルデータ転送処理の 3 つのタイプがあります。

(1) DSP データ処理

DSP 固定小数点データ処理で、A0、A1 レジスタをソースレジスタとして使うときは、ガードビット（ビット 39～32）は有効です。A0、A1 以外のレジスタ（M0、M1、X0、X1、Y0、Y1 レジスタ）をソースレジスタとして使うときは、そのレジスタデータの符号拡張されたものがビット 39～32 のデータとなります。A0、A1 レジスタをデスティネーションレジスタとして使うときは、ガードビット（ビット 39～32）は有効です。A0、A1 以外のレジスタをデスティネーションレジスタとして使うときは、結果のデータのビット 39～32 は無視されます。

DSP 整数データ処理の場合は DSP 固定小数点データ処理と同じです。ただし、ソースレジスタの下位ワード（下位 16 ビット、ビット 15～0）は無視されます。デスティネーションレジスタの下位ワードは 0 にクリアされます。

DSP 論理データ処理のソースレジスタは上位ワード（上位 16 ビット、ビット 31～16）が有効です。下位ワードと A0、A1 レジスタのガードビットは無視されます。デスティネーションレジスタは上位ワードが有効です。下位ワードと A0、A1 レジスタのガードビットは 0 にクリアされます。

(2) X、Y データ転送

MOVX.W、MOVY.W 命令は、X、Y メモリを 16 ビットの X、Y データバスを介してアクセスします。レジスタにロードされるデータ、レジスタからストアされるデータは、常に上位ワード（上位 16 ビット、ビット 31～16）です。レジスタの下位ワードは 0 でクリアされます。

(3) シングルデータ転送

MOVS.W、MOVS.L 命令は、命令データバス（メインバス）を介して、どのメモリでもアクセスできます。すべての DSP レジスタはメインバスとつながっており、データ転送のときソースレジスタ、デスティネーションレジスタいずれにもなります。データ転送にはワードとロングワードの 2 つのモードがあります。ワードモードでは、A0G、A1G レジスタを除いた DSP レジスタの上位ワードにロードまたは上位ワードからストアされます。ロングワードモードでは、A0G、A1G レジスタを除いた DSP レジスタの 32 ビットにロードまたは 32 ビットからストアされます。

シングルデータ転送では A0G、A1G レジスタを独立したレジスタとして取り扱うことができます。A0G、A1G レジスタにロード、ストアするデータ長は 8 ビットです。A0G、A1G レジスタがソースレジスタの場合は、データは 8 ビットだけがレジスタからストアされ、上位ビットは符号拡張されます。A0G、A1G レジスタがデスティネーションレジスタの場合は、データは最下位 8 ビットがレジスタにロードされ、A0、A1 レジスタは 0 でクリアされずに、それまでの値を保持します。

DSP タイプ命令でのレジスタ上のデータ形式を**表** 3.1、**表** 3.2 に示します。命令によってはアクセスできないレジスタがあります。たとえば、PMULS 命令は、A1 レジスタをソースレジスタに指定できますが、A0 レジスタはできません。詳細は、命令の説明を参照してください。

データ転送時の DSP レジスタとバスとの関係を**図** 3.5 に示します。

表 3.1 DSP タイプ命令のソースレジスタのデータ形式

レジスタ	命令		ガードビット 39 32	レジスタビット 31 16	15 0
A0, A1	DSP 演算	固定小数点、PDMSB、PSHA		40bit データ	
		整数		24bit データ	
		論理、PSHL、PMULS		16bit データ	
	データ転送	MOVX/Y.W、MOVS.W		16bit データ	
		MOVS.L		32bit データ	
A0G、A1G	データ転送	MOVS.W	データ		
		MOVS.L	データ		
X0、X1 Y0、Y1 M0、M1	DSP 演算	固定小数点、PDMSB、PSHA	符号*	32bit データ	
		整数	符号*	16bit データ	
		論理、PSHL、PMULS		16bit データ	
	データ転送	MOVS.W		16bit データ	
		MOVS.L		32bit データ	

【注】 * 符号が拡張され ALU のガードビットに格納されます

表 3.2 DSP タイプ命令のデスティネーションレジスタのデータ形式

レジスタ	命令		ガードビット 39 32	レジスタビット 31 16	15 0
A0、A1	DSP 演算	固定小数点、 PSHA、PMULS	（符号拡張）	40bit 結果	
		整数、PDMSB	（符号拡張）	24bit 結果	0 クリア
		論理、PSHL	0 クリア	16bit 結果	0 クリア
	データ転送	MOVS.W	符号拡張	16bit データ	0 クリア
		MOVS.L	符号拡張	32bit データ	
A0G、A1G	データ転送	MOVS.W	データ	更新せず	
		MOVS.L	データ	更新せず	
X0、X1 Y0、Y1 M0、M1	DSP 演算	固定小数点、PSHA、PMULS		32bit 結果	
		整数、論理、PDMSB、PSHL		16bit 結果	0 クリア
	データ転送	MOVX.W、MOVY.W、MOVS.W		16bit データ	0 クリア
		MOVS.L		32bit データ	

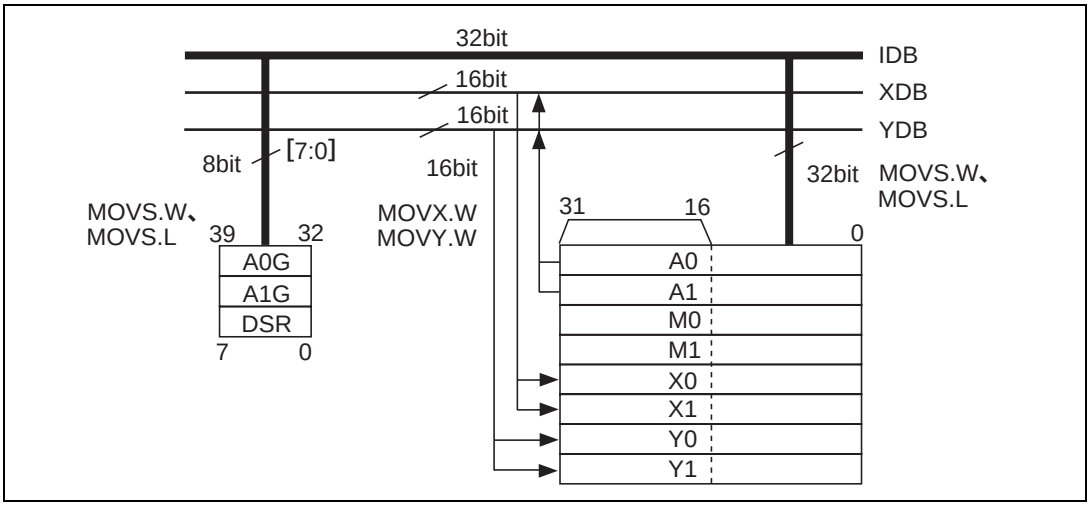


図 3.5　**データ転送時の** DSP **レジスタとバスとの関係**

4. 命令の特長

4.1 CPU タイプ命令

命令は RISC タイプです。特長は次のとおりです。

（1） 16ビット固定長命令
命令長はすべて16ビット固定長です。これによりプログラムのコード効率が向上します。

（2） 1命令／1ステート
パイプライン方式を採用し、基本命令は、1命令を1ステートで実行できます。

（3） データサイズ
演算の基本的なデータサイズはロングワードです。メモリのアクセスサイズは、バイト／ワード／ロングワードを選択できます。メモリのバイトとワードのデータは符号拡張後、ロングワードで演算されます。イミディエイトデータは算術演算では符号拡張後、論理演算ではゼロ拡張後、ロングワードで演算されます。

表 4.1 ワードデータの符号拡張

SH-1/SH-2/SH-DSP の CPU	説明	他の CPU の例
MOV.W @(disp,PC),R1 ADD R1,R0 ・・・・・・・・ .DATA.W H'1234	32 ビットに符号拡張され、R1 は H'00001234 になります。次に ADD 命令で演算されます。	ADD.W #H'1234,R0

【注】 @(disp,PC)でイミディエイトデータを参照します。

（4） ロードストアアーキテクチャ
基本演算はレジスタ間で実行します。メモリとの演算は、レジスタにデータをロードし実行します（ロードストアアーキテクチャ）。ただし、ANDなどのビットを操作する命令は直接メモリに対して実行します。

（5） 遅延分岐
無条件分岐命令などは、遅延分岐命令です。遅延分岐命令の場合、遅延分岐命令の直後の命令を実行してから、分岐します。これにより、分岐時のパイプラインの乱れを軽減しています。
遅延分岐においては、分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

表 4.2 遅延分岐命令

SH-1/SH-2/SH-DSP の CPU	説明	他の CPU の例
BRA TRGET ADD R1,R0	TRGET に分岐する前に ADD を実行します。	ADD.W R1,R0 BRA TRGET

（6） 乗算／積和演算

（a） SH-1 CPUの乗算／積和演算

16×16→32の乗算を1～3ステートで、16×16＋42→42の積和演算を2～3ステートで実行します。

（b） SH-2/SH- DSP CPUの乗算／積和演算

16×16→32の乗算を1～2ステート、16×16＋64→64の積和演算を2～3ステートで実行します。32×32→64の乗算や、32×32＋64→64の積和演算を2～4ステートで実行します。

（7） Tビット

比較結果はステータスレジスタ（SR）のTビットに反映し、その真、偽によって条件分岐します。必要最小限の命令によってのみTビットを変化させ、処理速度を向上させています。

表 4.3 T ビット

SH-1/SH-2/SH-DSP の CPU	説明	他の CPU の例
CMP/GE R1,R0	R0≧R1 のとき T ビットがセットされます。	CMP.W R1,R0
BT TRGET0	R0≧R1 のとき TRGET0 へ	BGE TRGET0
BF TRGET1	R0＜R1 のとき TRGET1 へ分岐します。	BLT TRGET1
ADD #－1,R0	ADD では T ビットが変化しません。	SUB.W #1,R0
CMP/EQ #0,R0	R0＝0 のとき T ビットがセットされます。	BEQ TRGET
BT TRGET	R0＝0 のとき分岐します。	

（8） イミディエイトデータ

バイトのイミディエイトデータは命令コードの中に配置します。ワードとロングワードのイミディエイトデータは命令コードの中に配置せず、メモリ上のテーブルに配置します。メモリ上のテーブルはディスプレースメント付きPC相対アドレッシングモードを使ったイミディエイトデータのデータ転送命令（MOV）で参照します。

表 4.4 イミディエイトデータによる参照

区分	SH-1/SH-2/SH-DSP の CPU	他の CPU の例
8 ビットイミディエイト	MOV #H'12,R0	MOV.B #H'12,R0
16 ビットイミディエイト	MOV.W @(disp,PC),R0 ・・・・・・・・ .DATA.W H'1234	MOV.W #H'1234,R0
32 ビットイミディエイト	MOV.L @(disp,PC),R0 ・・・・・・・・ .DATA.L H'12345678	MOV.L #H'12345678,R0

【注】 @(disp,PC)でイミディエイトデータを参照します。

（9） 絶対アドレス

絶対アドレスでデータを参照するときは、あらかじめ絶対アドレスの値を、メモリ上のテーブルに配置しておきます。命令実行時にイミディエイトデータをロードする方法で、この値をレジスタに転送し、レジスタ間接アドレッシングモードでデータを参照します。

表 4.5 絶対アドレスによる参照

区分	SH-1/SH-2/SH-DSP の CPU	他の CPU の例
絶対アドレス	MOV.L @(disp,PC),R1 MOV.B @R1,R0 ・・・・・・・・ .DATA.L H'12345678	MOV.B @H'12345678,R0

（10）16ビット／32ビットディスプレースメント

16ビットまたは32ビットディスプレースメントでデータを参照するときは、あらかじめディスプレースメントの値をメモリ上のテーブルに配置しておきます。命令実行時にイミディエイトデータをロードする方法で、この値をレジスタに転送し、インデックス付きレジスタ間接アドレッシングモードでデータを参照します。

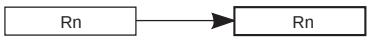
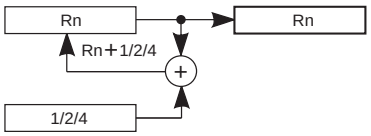
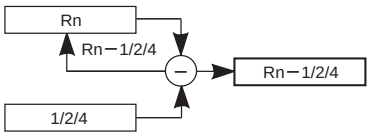
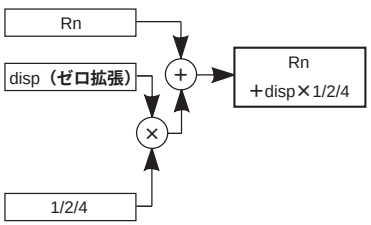
表 4.6 ディスプレースメントによる参照

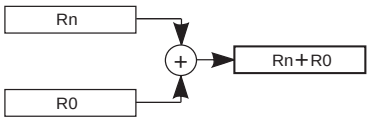
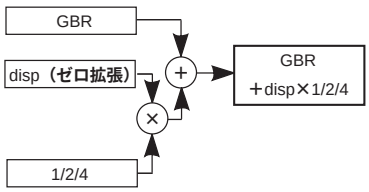
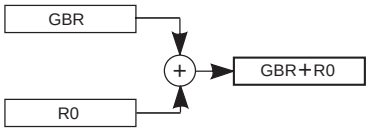
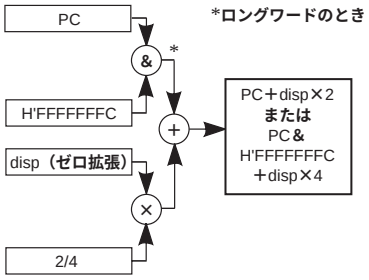
区分	SH-1/SH-2/SH-DSP の CPU	他の CPU の例
16 ビットディスプレースメント	MOV.W @(disp,PC),R0 MOV.W @(R0,R1),R2 ・・・・・・・・ .DATA.W H'1234	MOV.W @(H'1234,R1),R2

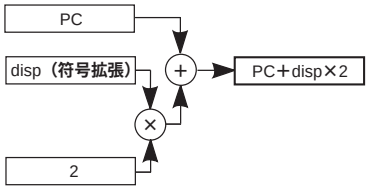
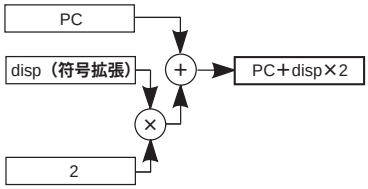
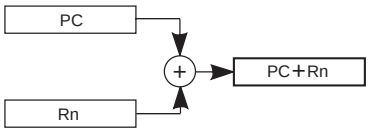
4.2 CPU タイプ命令のアドレッシングモード

CPU タイプ命令の実行で使用されるアドレッシングモードと実効アドレスの計算方法は次のとおりです。

表 4.7 アドレッシングモードと実効アドレス

アドレッシングモード	命令フォーマット	実効アドレスの計算方法	計算式
レジスタ直接	Rn	実効アドレスはレジスタ Rn です。（オペランドはレジスタ Rn の内容です。）	—
レジスタ間接	@Rn	実効アドレスはレジスタ Rn の内容です。 	Rn
ポストインクリメントレジスタ間接	@Rn+	実効アドレスはレジスタ Rn の内容です。命令実行後 Rn に定数を加算します。定数はオペランドサイズがバイトのとき 1、ワードのとき 2、ロングワードのとき 4 です。 	Rn 命令実行後 バイト：Rn＋1→Rn ワード：Rn＋2→Rn ロングワード：Rn＋4→Rn
プリデクリメントレジスタ間接	@-Rn	実効アドレスは、あらかじめ定数を減算したレジスタ Rn の内容です。定数はバイトのとき 1、ワードのとき 2、ロングワードのとき 4 です。 	バイト：Rn－1→Rn ワード：Rn－2→Rn ロングワード：Rn－4→Rn （計算後の Rn で命令実行）
ディスプレースメント付きレジスタ間接	@(disp:4,Rn)	実効アドレスはレジスタ Rn に 4 ビットディスプレースメント disp を加算した内容です。disp はゼロ拡張後、オペランドサイズによってバイトで 1 倍、ワードで 2 倍、ロングワードで 4 倍します。 	バイト：Rn＋disp ワード：Rn＋disp×2 ロングワード：Rn＋disp×4

アドレッシングモード	命令フォーマット	実効アドレスの計算方法	計算式
インデックス付きレジスタ間接	@(R0,Rn)	実効アドレスはレジスタ Rn に R0 を加算した内容です。 	Rn＋R0
ディスプレースメント付き GBR 間接	@(disp:8,GBR)	実効アドレスはレジスタ GBR に 8 ビットディスプレースメント disp を加算した内容です。disp はゼロ拡張後、オペランドサイズによってバイトで 1 倍、ワードで 2 倍、ロングワードで 4 倍します。 	バイト：GBR＋disp ワード：GBR＋disp×2 ロングワード： GBR＋disp×4
インデックス付き GBR 間接	@(R0,GBR)	実効アドレスはレジスタ GBR に R0 を加算した内容です。 	GBR＋R0
ディスプレースメント付き PC 相対	@(disp:8,PC)	実効アドレスはレジスタ PC に 8 ビットディスプレースメント disp を加算した内容です。disp はゼロ拡張後、オペランドサイズによってワードで 2 倍、ロングワードで 4 倍します。 さらにロングワードのときは PC の下位 2 ビットをマスクします。 	ワード：PC＋disp×2 ロングワード： PC&H'FFFFFFFC＋disp×4

アドレッシングモード	命令フォーマット	実効アドレスの計算方法	計算式
PC 相対	disp:8	実効アドレスはレジスタ PC に 8 ビットディスプレースメント disp を符号拡張後 2 倍し、加算した内容です。 	PC＋disp×2
	disp:12	実効アドレスはレジスタ PC に 12 ビットディスプレースメント disp を符号拡張後 2 倍し、加算した内容です。 	PC＋disp×2
	Rn*	実効アドレスはレジスタ PC に Rn を加算した内容です。 	PC＋Rn
イミディエイト	#imm:8	TST、AND、OR、XOR 命令の 8 ビットイミディエイト imm はゼロ拡張します。	—
	#imm:8	MOV、ADD、CMP/EQ 命令の 8 ビットイミディエイト imm は符号拡張します。	—
	#imm:8	TRAPA 命令の 8 ビットイミディエイト imm はゼロ拡張後、4 倍します。	—

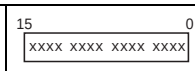
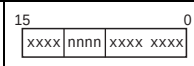
【注】 * SH-2/SH-DSP で実行されます。SH-1 ではこのアドレッシングモードはありません。

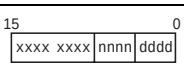
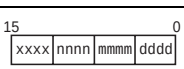
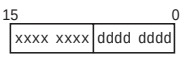
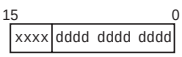
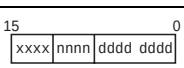
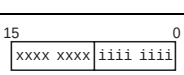
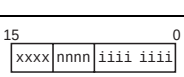
4.3 CPU タイプ命令の命令形式

命令形式とソースオペランドとデスティネーションオペランドの意味を示します。命令コードによりオペランドの意味が異なります。記号は次のとおりです。

xxxx ：命令コード
mmmm ：ソースレジスタ
nnnn ：デスティネーションレジスタ
iiii ：イミディエイトデータ
dddd ：ディスプレースメント

表 4.8 命令形式

命令形式		ソースオペランド	デスティネーション オペランド	命令の例
0 形式		—	—	NOP
n 形式		—	nnnn：レジスタ直接	MOVT Rn
		コントロールレジスタ またはシステムレジスタ	nnnn：レジスタ直接	STS MACH,Rn
		コントロールレジスタ またはシステムレジスタ	nnnn：プリデクリメント レジスタ間接	STC.L SR,@-Rn
m 形式		mmmm：レジスタ直接	コントロールレジスタ またはシステムレジスタ	LDC Rm,SR
		mmmm：ポストインクリメ ントレジスタ間接	コントロールレジスタ またはシステムレジスタ	LDC.L @Rm+,SR
		mmmm：レジスタ間接	—	JMP @Rm
		mmmm：Rm を用いた PC 相対*1	—	BRAF Rm
nm 形式		mmmm：レジスタ直接	nnnn：レジスタ直接	ADD Rm,Rn
		mmmm：レジスタ直接	nnnn：レジスタ間接	MOV.L Rm,@Rn
		mmmm： ポストインクリメントレ ジスタ間接（積和演算） nnnn：*2 ポストインクリメントレ ジスタ間接（積和演算）	MACH,MACL	MAC.W @Rm+,@Rn+
		mmmm：ポストインクリメ ントレジスタ間接	nnnn：レジスタ直接	MOV.L @Rm+,Rn
		mmmm：レジスタ直接	nnnn：プリデクリメント レジスタ間接	MOV.L Rm,@-Rn
		mmmm：レジスタ直接	nnnn：インデックス付き レジスタ間接	MOV.L Rm,@(R0,Rn)

命令形式		ソースオペランド	デスティネーション オペランド	命令の例
md 形式		mmmmdddd： ディスプレースメント付きレジスタ間接	R0（レジスタ直接）	MOV.B @(disp,Rm),R0
nd4 形式		R0（レジスタ直接）	nnnndddd： ディスプレースメント付きレジスタ間接	MOV.B R0,@(disp,Rn)
nmd 形式		mmmm：レジスタ直接	nnnndddd： ディスプレースメント付きレジスタ間接	MOV.L Rm,@(disp,Rn)
		mmmmdddd： ディスプレースメント付きレジスタ間接	nnnn：レジスタ直接	MOV.L @(disp,Rm),Rn
d 形式		dddddddd： ディスプレースメント付き GBR 間接	R0（レジスタ直接）	MOV.L @(disp,GBR),R0
		R0（レジスタ直接）	dddddddd： ディスプレースメント付き GBR 間接	MOV.L R0,@(disp,GBR)
		dddddddd： ディスプレースメント付き PC 相対	R0（レジスタ直接）	MOVA @(disp,PC),R0
		dddddddd：PC 相対	—	BF label
d12 形式		ddddddddddddd：PC 相対	—	BRA label (label=disp+PC)
nd8 形式		dddddddd： ディスプレースメント付き PC 相対	nnnn：レジスタ直接	MOV.L @(disp,PC),Rn
i 形式		iiiiiiii： イミディエイト	インデックス付き GBR 間接	AND.B #imm,@(R0,GBR)
		iiiiiiii： イミディエイト	R0（レジスタ直接）	AND #imm,R0
		iiiiiiii： イミディエイト	—	TRAPA #imm
ni 形式		iiiiiiii： イミディエイト	nnnn：レジスタ直接	ADD #imm,Rn

【注】 *1 SH-2/SH-DSP で使用可能です。SH-1 では使用できません。

*2 積和命令では nnnn は、ソースレジスタです。

4.4 DSP タイプ命令の方式

DSP タイプ命令の演算とデータ転送には次のものがあります。

（1） ALU固定小数点演算：
固定小数点データ40ビット（ガードビット付き）または32ビット（ガードビットなし）の固定小数点算術演算です。加減算、比較命令などがあります。

（2） ALU整数演算：
整数データ24ビット（ガードビット付き）または16ビット（ガードビットなし）の整数算術演算です。インクリメント、デクリメント命令があります。

（3） ALU論理演算：
論理データ16ビットの論理演算です。論理積、論理和、排他的論理和があります。

（4） 固定小数点乗算：
固定小数点データ上位16ビットの固定小数点乗算（算術演算）です。DCビットなどの状態ビットは更新されません。

（5） シフト演算：
算術シフト演算と論理シフト演算があります。算術シフト演算は固定小数点データ40ビット（ガードビット付き）または32ビット（ガードビットなし）の算術シフトです。論理シフト演算は論理データ16ビットの論理演算です。算術シフト演算のシフト量は－32～＋32（負は右シフト、正は左シフト）、論理シフト演算のシフト量は－16～＋16です。

（6） MSB検出命令：
データを正規化するためのシフト量を求める演算です。固定小数点データ40ビット（ガードビット付き）または32ビット（ガードビットなし）のMSBビット位置を整数データ24ビット（ガードビット付き）または16ビット（ガードビットなし）で求めます。

（7） 丸め演算：
固定小数点データ40ビット（ガードビット付き）を24ビットに、または32ビット（ガードビットなし）を16ビットに丸めます。

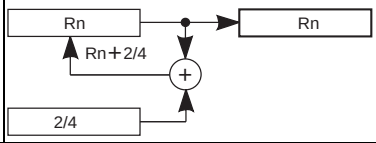
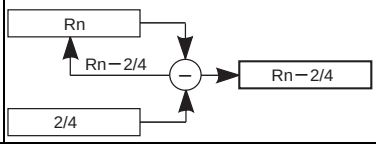
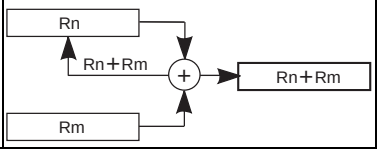
（8） データ転送：
X、Yメモリから16ビットデータをロード、ストアするX、Yデータ転送と、すべてのメモリから16ビット、または32ビットデータをロード、ストアするシングルデータ転送とがあります。X、Yデータ転送は2つの処理を同時に並行して処理することができます。DCビットなどの状態ビットは更新されません。

演算命令には、無条件演算命令と、DC ビットを判定して実行する条件付き命令があります。DC ビットなどの状態ビットは、条件付き命令では更新されません。DC ビットなどの状態ビットは、算術演算、論理演算、算術シフト、論理シフトで、それぞれ設定が異なります。MSB 検出命令、丸め演算での DC ビットなどの状態ビットの設定は、算術演算として設定されます。

算術演算にはオーバフロー防止機能（飽和演算）があります。SR レジスタの S ビットで飽和演算を指定すると、演算結果がオーバフローしたとき最大値（正）または最小値（負）が格納されます。S ビット機能は、ALU、シフト、乗算のすべての算術演算に有効です。

4.5 DSP タイプ命令のアドレッシングモード

表 4.9 アドレッシングモードと実効アドレス

アドレッシングモード	命令フォーマット	実効アドレスの計算方法	計算式	命令
レジスタ間接	@Rn	実効アドレスはレジスタ Rn の内容です。 	Rn	MOVS MOVX MOVY
ポストインクリメントレジスタ間接	@Rn+	実効アドレスはレジスタ Rn の内容です。命令実行後 Rn に定数を加算します。定数はオペランドサイズがワードのとき 2、ロングワードのとき 4 です。 	Rn 命令実行後 ワード： Rn＋2→Rn ロングワード： Rn＋4→Rn	MOVS MOVX （ワードのみ） MOVY （ワードのみ）
プリデクリメントレジスタ間接	@-Rn	実効アドレスは、あらかじめ定数を減算したレジスタ Rn の内容です。定数はワードのとき 2、ロングワードのとき 4 です。 	ワード： Rn－2→Rn ロングワード： Rn－4→Rn （計算後の Rn で命令実行）	MOVS
インデックス付きレジスタ間接	@Rn+Rm	実効アドレスはレジスタ Rn に Rm を加算した内容です。 	Rn+Rm	MOVS MOVX MOVY

4.6 DSP データアドレッシング

DSP タイプ命令では 2 つの異なったメモリアクセスをします。1 つは X、Y データ転送命令（MOVX.W、MOVY.W）で、もう 1 つはシングルデータ転送命令（MOVS.W、MOVS.L）です。これらの 2 種類の命令のデータアドレッシングは異なります。データ転送命令の概要を**表** 4.10 に示します。

表 4.10　**データ転送命令の概要**

	X、Y **データ転送処理** (MOVX.W、MOVY.W)	**シングルデータ転送処理** (MOVS.W、MOVS.L)
アドレスレジスタ	Ax：R4、R5、Ay：R6、R7	As：R2、R3、R4、R5
インデックスレジスタ	Ix：R8、Iy：R9	Is：R8
アドレッシング	Nop/Inc(+2)/**インデクス加算：ポストインクリメント**	Nop/Inc(+2,+4)/**インデクス加算：ポストインクリメント**
	―	Dec(-2,-4)：**プリデクリメント**
モジュロアドレッシング	**可能**	**不可**
データバス	XDB、YDB	**メインバス**
データ長	16bit（**ワード**）	16bit/32bit（**ワード／ロングワード**）
バス競合	**なし**	**あり**
メモリ	X、Y **データメモリ**	**すべてのメモリ空間**
ソースレジスタ	Da：A0、A1	Ds：A0/A1、M0/M1、X0/X1、Y0/Y1、A0G、A1G
デスティネーションレジスタ	Dx：X0/X1、Dy：Y0/Y1	Ds：A0/A1、M0/M1、X0/X1、Y0/Y1、A0G、A1G

4.6.1 X、Y データアドレッシング

DSP タイプ命令では MOVX.W、MOVY.W 命令を使って、X、Y データメモリを同時にアクセスすることができます。DSP タイプ命令には同時に X、Y データメモリをアクセスするために 2 つのアドレスポインタがあります。DSP タイプ命令にはポインタアドレッシングだけが可能で、イミディエイトアドレッシングはありません。アドレスレジスタは 2 つに分けられ、R4、R5 レジスタが X メモリのアドレスレジスタ（Ax）となり、R6、R7 レジスタが Y メモリのアドレスレジスタ（Ay）となります。X、Y データ転送命令には次の 3 つのアドレッシングがあります。

（1） 更新なしアドレスレジスタ：
　　Ax、Ayレジスタがアドレスポインタです。更新されません。

（2） 加算インデクスレジスタ：
　　Ax、Ayレジスタがアドレスポインタです。データ転送後それぞれIx、Iyレジスタの値が加算されます（ポストインクリメント）。

（3） インクリメントアドレスレジスタ：
　　Ax、Ayレジスタがアドレスポインタです。データ転送後それぞれ＋2が加算されます（ポストインクリメント）。

それぞれのアドレスポインタにはインデクスレジスタがあります。R8 レジスタは X メモリアドレスレジスタ（Ax）のインデクスレジスタ（Ix）となり、R9 レジスタは Y メモリアドレスレジスタ（Ay）のインデクスレジスタ（Iy）となります。

X、Y データ転送命令はワードで処理します。X、Y データメモリを 16 ビットでアクセスします。そのためインクリメント処理は、アドレスレジスタに 2 を加えます。デクリメントさせるためには、

−2 をインデクスレジスタに設定し加算インデクスレジスタアドレッシングを指定します。X、Y データアドレッシング時は、アドレスポインタのビット 1〜15 のみ有効となりますので、アドレスポインタ、インデクスレジスタのビット 0 に必ず 0 を書き込んでください。

X、Y データ転送のアドレッシングを**図** 4.1 に示します。X、Y バスを使用して X メモリ、Y メモリへアクセスする場合、Ax（R4 または R5）、Ay（R6 または R7）の上位ワードは無視されます。また、@Ay+、@Ay+Iy の結果は、Ay の下位ワードに格納され、上位ワードは元の値が保持されます。

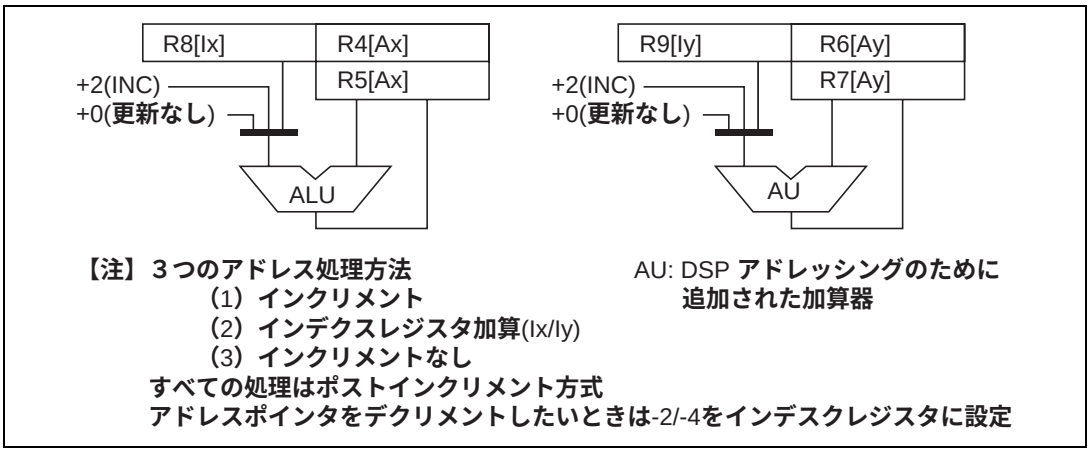


図 4.1　X、Y **データ転送のアドレッシング**

4.6.2　シングルデータアドレッシング

DSP タイプ命令にはシングルデータ転送命令（MOVS.W、MOVS.L）があり、DSP レジスタにデータをロードし、DSP レジスタからデータをストアします。この命令で R2〜R5 レジスタはシングルデータ転送のアドレスレジスタ（As）として使われます。

シングルデータ転送命令には次の 4 つのデータアドレッシングがあります。

（1） インクリメントなしアドレスレジスタ：
Asレジスタがアドレスポインタです。インクリメントされません。

（2） 加算インデクスレジスタ：
Asレジスタがアドレスポインタです。データ転送後Isレジスタの値が加算されます（ポストインクリメント）。

（3） インクリメントアドレスレジスタ：
Asレジスタがアドレスポインタです。データ転送後＋2または＋4が加算されます（ポストインクリメント）。

（4） デクリメントアドレスレジスタ：
Asレジスタがアドレスポインタです。データ転送前に−2、−4が加算（＋2または＋4が減算）されます（プリデクリメント）。

アドレスポインタ（As）は R8 レジスタをインデクスレジスタ（Is）として使います。シングルデータ転送のアドレッシングを**図** 4.2 に示します。

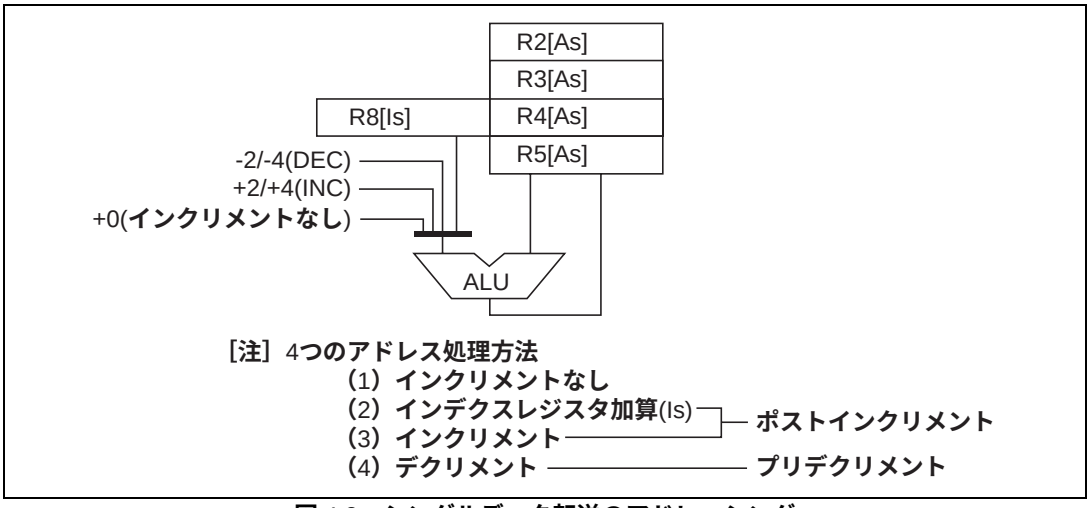


図 4.2 **シングルデータ転送のアドレッシング**

4.6.3 モジュロアドレッシング

SH-DSP には、他の DSP と同様にモジュロアドレッシングモードがあります。このモードでもアドレスレジスタは同じようにインクリメントされます。アドレスポインタの値がすでに設定されたモジュロ終了アドレスになると、アドレスポインタはモジュロ開始アドレスになります。

モジュロアドレッシングは X、Y データ転送命令（MOVX.W、MOVY.W）にだけ有効です。SR レジスタの DMX ビットをセットすると X アドレスレジスタが、DMY ビットをセットすると Y アドレスレジスタがそれぞれモジュロアドレッシングモードになります。モジュロアドレッシングはどちらかの X、Y アドレスレジスタに対してだけ有効です。両方を同時にモジュロアドレッシングモードにすることはできません。したがって、DMX と DMY を同時にセットしないで下さい。万一同時にセットされた場合には、DMY 側のみ有効となります。

モジュロアドレス領域の開始と終了アドレスを指定するための MOD レジスタがあり、MOD レジスタは MS（Modulo Start： モジュロ開始）と、ME（Modulo End： モジュロ終了）を格納します。MOD レジスタ（MS、ME）の使用例を次に示します。

```
            MOV.L ModAddr,Rn;          Rn=ModEnd, ModStart
            LDC Rn,MOD;                ME=ModEnd, MS=ModStart
  ModAddr:  .DATA.W                    mEnd;Lower 16bit of ModEnd
            .DATA.W                    mStart;Lower 16bit of ModStart

  ModStart: .DATA
              :
  ModEnd:   .DATA
```

MS、ME には開始、終了アドレスを指定して、そのあとで DMX 又は DMY ビットを 1 にセットします。アドレスレジスタの内容が ME と比較されます。もし ME と一致したら、開始アドレス MS をアドレスレジスタに格納します。アドレスレジスタの下位 16 ビットが ME と比較されます。最大のモジュロサイズは 64K バイトです。これは X、Y データメモリをアクセスするには十分です。モジュロアドレッシングのブロック図を**図** 4.3 に示します。

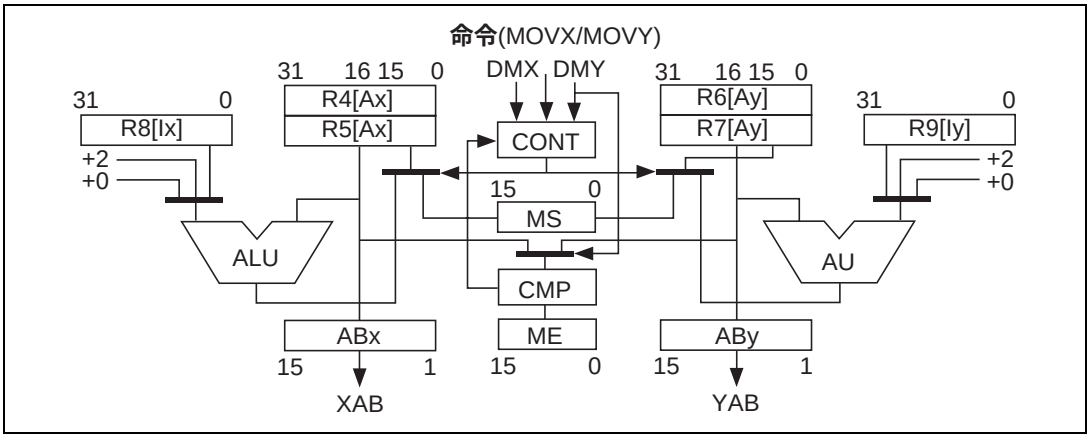


図 4.3 **モジュロアドレッシング**

モジュロアドレッシングの例を次に示します。

```
MS = H'C008; ME=H'C00C; R4=H'C008;
DMX=1; DMY=0; (アドレスレジスタ Ax(R4,R5)に対するモジュロアドレッシングの設定です)
```

以上の設定により R4 レジスタは次のように変化します。

```
          R4: H'C008
Inc.      R4: H'C00A
Inc.      R4: H'C00C
Inc.      R4: H'C008  (モジュロ終了アドレスになったので、モジュロ開始アドレスになります)
```

モジュロ開始、終了アドレスの上位 16 ビットは同じになるようデータを配置します。これはモジュロ開始アドレスがアドレスレジスタの下位 16 ビットだけを置き換えるからです。

【注】 DSP **データアドレッシングに加算インデクスを使う場合は、アドレスポインタは** ME **と一致せずにその値を超えてしまうことがあります。この場合は、アドレスポインタはモジュロ開始アドレスには戻りません。**

4.6.4 DSP アドレッシング動作

モジュロアドレッシングを含めて、パイプラインの実行ステージ（EX）での DSP アドレッシングの動作を次に示します。

```
if ( Opration is MOVX.W MOVY.W ) {
   ABx=Ax; ABy=Ay;
   /* memory access cycle uses ABx and ABy. The addresses to be used have not
been updated */

   /* Ax is one of R4,5 */
   if {DMX==0 || (DMX==1 && DMY == 1 )} Ax=Ax+(+2 or R8[Ix] or +0);
   /* Inc,Index,Not-Update */
   else if (! not-update) Ax=modulo( Ax, (+2 or R8[Ix]) );
```

```
   /* Ay is one of R6,7 */
   if ( DMY==0 ) Ay=Ay+(+2 or R9[Iy] or +0); /* Inc,Index,Not-Update */
   else if (! not-update) Ay=modulo( Ay, (+2 or R9[Iy]) );
}
else if ( Operation is MOVS.W or MOVS.L ) {
  if ( Addressing is Nop, Inc, Add-index-reg ) {
   MAB=As;
   /* memory access cycle uses MAB. The address to be used has not been updated
*/

   /* As is one of R2~5 */
   As=As+(+2 or +4 or R8[Is] or +0); /* Inc,Index,Not-Update */
  else { /* Decrement, Pre-update */
   /* As is one of R2~5 */
   As=As+(-2 or -4);
   MAB=As;
   /* memory access cycle uses MAB. The address to be used has been updated
*/
}

/* The value to be added to the address register depends on addressing operations.
For example, (+2 or R8[Ix] or +0) means that
      +2   : if operation is increment
   R8[Ix]  : if operation is add-index-reg
      +0   : if operation is not-update
*/

function modulo ( AddrReg, Index ) {
   if ( AdrReg[15:0]==ME ) AdrReg[15:0]==MS;
   else AdrReg=AdrReg+Index;
   return AddrReg;
}
```

4.7 DSP タイプ命令の命令形式

SH-DSP にはデジタル信号処理のための新しい命令が追加されています。新しい命令は次の 2 つに分けられます。

（1） メモリとDSPレジスタのダブル、シングルデータ転送命令（16ビット長）

（2） DSPエンジンで処理される並行処理命令（32ビット長）

それぞれの命令形式を**図** 4.4 に示します。

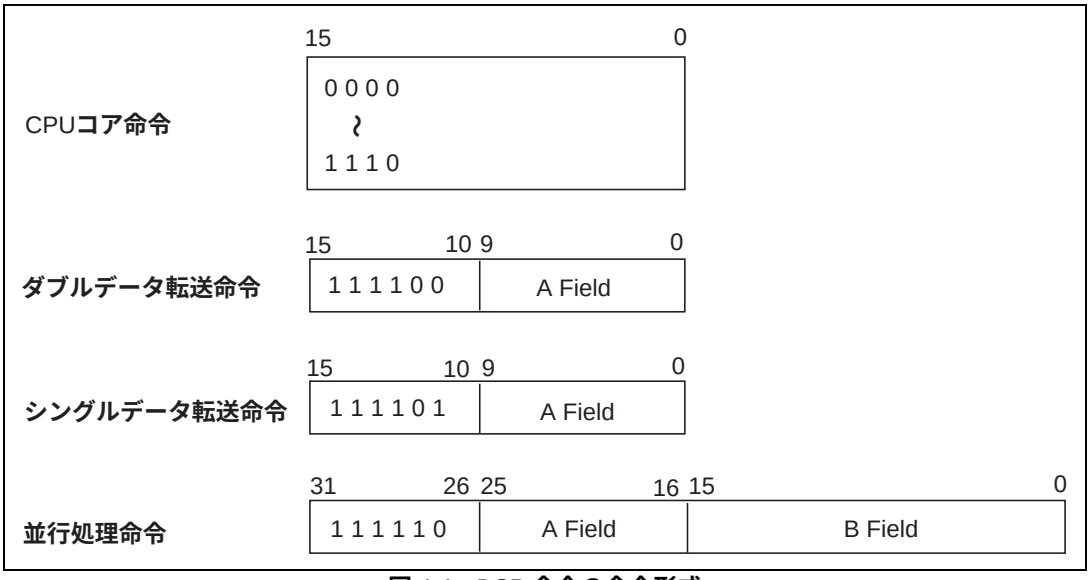


図 4.4 DSP **命令の命令形式**

4.7.1 ダブル、シングルデータ転送命令

ダブルデータ転送命令の命令形式を**表** 4.11 に、シングルデータ転送命令の命令形式を**表** 4.12 に示します。

表 4.11　ダブルデータ転送の命令形式

分類	ニーモニック	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Xメモリ	NOPX	1	1	1	1	0	0	0		0		0		0	0		
データ	MOVX.W @Ax,Dx							Ax		Dx		0		0	1		
転送	MOVX.W @Ax+,Dx													1	0		
	MOVX.W @Ax+Ix,Dx													1	1		
	MOVX.W Da,@Ax									Da		1		0	1		
	MOVX.W Da,@Ax+													1	0		
	MOVX.W Da,@Ax+Ix													1	1		
Yメモリ	NOPY	1	1	1	1	0	0		0		0		0			0	0
データ	MOVY.W @Ay,Dy								Ay		Dy		0			0	1
転送	MOVY.W @Ay+,Dy															1	0
	MOVY.W @Ay+Iy,Dy															1	1
	MOVY.W Da,@Ay										Da		1			0	1
	MOVY.W Da,@Ay+															1	0
	MOVY.W Da,@Ay+Iy															1	1

Ax：0＝R4、1＝R5　Ay：0＝R6、1＝R7　Dx：0＝X0、1＝X1　Dy：0＝Y0、1＝Y1

Da：0=A0、1=A1

表 4.12　シングルデータ転送命令の命令形式

分類	ニーモニック	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
シングル	MOVS.W @-As,Ds	1	1	1	1	0	1	As		Ds		0:(*1)		0	0	0	0
データ	MOVS.W @As,Ds							0:R4				1:(*1)		0	1		
転送	MOVS.W @As+,Ds							1:R5				2:(*1)		1	0		
	MOVS.W @As+Is,Ds							2:R2				3:(*1)		1	1		
	MOVS.W Ds,@-As							3:R3				4:(*1)		0	0	0	1
	MOVS.W Ds,@As											5:A1		0	1		
	MOVS.W Ds,@As+											6:(*1)		1	0		
	MOVS.W Ds,@As+Is											7:A0		1	1		
	MOVS.L @-As,Ds											8:X0		0	0	1	0
	MOVS.L @As,Ds											9:X1		0	1		
	MOVS.L @As+,Ds											A:Y0		1	0		
	MOVS.L @As+Is,Ds											B:Y1		1	1		
	MOVS.L Ds,@-As											C:M0		0	0	1	1
	MOVS.L Ds,@As											D:A1G		0	1		
	MOVS.L Ds,@As+											E:M1		1	0		
	MOVS.L Ds,@As+Is											F:A0G		1	1		

【注】 *1　**システム予約コード**

4.7.2 並行処理命令

並行処理命令は DSP ユニットを使ったデジタル信号処理を効率よく実行するための命令です。32 ビット長で、同時に並行して 4 つの処理、ALU 演算、乗算、2 つのデータ転送ができます。

並行処理命令は A フィールドと B フィールドに分かれています。A フィールドはデータ転送命令を定義し、B フィールドは ALU 演算命令、乗算命令を定義します。これらの命令は独立に定義することができ、処理は独立に、しかも同時に並行して実行されます。A フィールドの並行データ転送命令を**表** 4.13 に、B フィールドの ALU 演算命令、乗算命令を**表** 4.14 に示します。A フィールドの命令は、**表** 4.11 のダブルデータ転送と同じです。

表 4.13 A **フィールドの並行データ転送命令**

分類	ニーモニック	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X **メモリデータ転送**	NOPX	1	1	1	1	1	0	0		0		0		0	0			B**フィールド**															
	MOVX.W @Ax, Dx							Ax		Dx		0		0	1																		
	MOVX.W @Ax+, Dx													1	0																		
	MOVX.W @Ax+Ix, Dx													1	1																		
	MOVX.W Da, @Ax									Da		1		0	1																		
	MOVX.W Da, @Ax+													1	0																		
	MOVX.W Da, @Ax+Ix													1	1																		
Y **メモリデータ転送**	NOPY								0		0		0			0	0	B**フィールド**															
	MOVY.W @Ay, Dy								Ay		Dy		0			0	1																
	MOVY.W @Ay+, Dy															1	0																
	MOVY.W @Ay+Iy, Dy															1	1																
	MOVY.W Da, @Ay										Da		1			0	1																
	MOVY.W Da, @Ay+															1	0																
	MOVY.W Da, @Ay+Iy															1	1																

Ax: 0＝R4、1＝R5　Ay: 0＝R6、1＝R7　Dx: 0＝X0、1＝X1　Dy: 0＝Y0、1＝Y1　Da: 0＝A0、1＝A1

表 4.14　B フィールドの ALU 演算命令、乗算命令

分類	ニーモニック	31 30 29 28 27 26	25 24 23 22 21 20 19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4	3 2 1 0
imm. シフト	PSHL #imm, Dz PSHA #imm, Dz	1 1 1 1 1 0	Aフィールド	0 0 0 0 0 0 0 1	0 -16<=imm<=+16 0 -32<=imm<=+32	Dz
	予約			0 0 0 0 0 1	1	
6オペランド パラレル 命令	PMULS Se, Sf, Dg			0 1 0 0	Se Sf Sx Sy Dg Du	
	予約			0 1 0 1	0:X0 0:Y0 0:X0 0:Y0 0:M0 0:X0 1:X1 1:Y1 1:X1 1:Y1 1:M1 1:Y0	
	PSUB Sx, Sy, Du PMULS Se, Sf, Dg			0 1 1 0	2:Y0 2:X0 2:A0 2:M0 2:A0 2:A0 3:A1 3:A1 3:A1 3:M1 3:A1 3:A1	
	PADD Sx, Sy, Du PMULS Se, Sf, Dg			0 1 1 1		
3オペランド 命令	予約			1 0 0 0	0 0 0 0	Dz
	PSUBC Sx, Sy, Dz			0 1		0:(*1)
	PADDC Sx, Sy, Dz			1 0		1:(*1)
	PCMP Sx, Sy			1 1		2:(*1)
	予約			0 0	0 1	3:(*1)
				0 1		4:(*1)
				1 0		5:A1
				1 1		6:(*1)
	PABS Sx, Dz			0 0	1 0	7:A0
	PRND Sx, Dz			0 1		8:X0
	PABS Sy, Dz			1 0		9:X1
	PRND Sy, Dz			1 1		A:Y0
	予約			0 0	1 1	B:Y1
				0 1		C:M0
				1 0		D:(*1)
				1 1		E:M1
条件付き 3オペランド 命令	[if cc] PSHL Sx, Sy, Dz			0 0	0 0	F:(*1)
	[if cc] PSHA Sx, Sy, Dz			0 1		
	[if cc] PSUB Sx, Sy, Dz			1 0		
	[if cc] PADD Sx, Sy, Dz			1 1		
	予約			0 0	0 1	
	[if cc] PAND Sx, Sy, Dz			0 1		
	[if cc] PXOR Sx, Sy, Dz			1 0		
	[if cc] POR Sx, Sy, Dz			1 1		
	[if cc] PDEC Sx, Dz			0 0	1 0	
	[if cc] PINC Sx, Dz			0 1		
	[if cc] PDEC Sy, Dz			1 0		
	[if cc] PINC Sy, Dz			1 1		
	[if cc] PCLR Dz			0 0	1 1	
	[if cc] PDMSB Sx, Dz			0 1		
	予約			1 0		
	[if cc] PDMSB Sy, Dz			1 1		
	[if cc] PNEG Sx, Dz			1 1 0 0	1 0	
	[if cc] PCOPY Sx, Dz			0 1		
	[if cc] PNEG Sy, Dz			1 0		
	[if cc] PCOPY Sy, Dz			1 1		
	予約				0 0	
	[if cc] PSTS MACH, Dz			0 0	1 1	
	[if cc] PSTS MACL, Dz			0 1		
	[if cc] PLDS Dz, MACH			1 0		
	[if cc] PLDS Dz, MACL			1 1		
	(*2) 予約				0 0	
	予約	1 1 1 1 1 1			0 *	

if cc: 01: 無条件　10: DCT　11: DCF

【注】 *1　システム予約コード
*2　[if cc]：DCT（DCビット真）、DCF（DCビット偽）またはなし（無条件命令）.

4.8 ALU 固定小数点演算

(1) 演算機能

ALU 固定小数点算術演算は、基本の精度 32 ビットにガードビット 8 ビットを加えた 40 ビットで演算されます。ソースオペランドがガードビットなしのレジスタのときは、その符号ビットがガードビットに拡張されて転記されます。デスティネーションオペランドがガードビットのないレジスタのときは、演算結果の下位 32 ビットがデスティネーションレジスタに格納されます。

ALU 固定小数点算術演算はレジスタ間で実行されます。ソースおよびデスティネーションレジスタはそれぞれ独立に DSP レジスタから選べます。選ばれたレジスタにガードビットがあるときは、ガードビットを含めてこれらの演算が実行されます。これらの演算の実行はパイプラインの流れの最後の DSP ステージで実行されます。

ALU 算術演算が実行されるときはいつでも、DSR レジスタの DC、N、Z、V、GT ビットが演算結果によって更新されます。しかし条件付き命令の場合は、指定された状態になっても状態ビットは更新されません。無条件命令の場合は、演算結果に従って更新されます。

DC ビットに反映させる状態は、CS［2：0］ビットによって選択されます。ただし、PADDC 命令と PSUBC 命令の DC ビットは、CS ビットの設定に関係なく更新されます。PADDC 命令ではキャリフラグとして更新され、PSUBC 命令ではボローフラグとして更新されます。

ALU 固定小数点算術演算の流れを**図** 4.5 に示します。

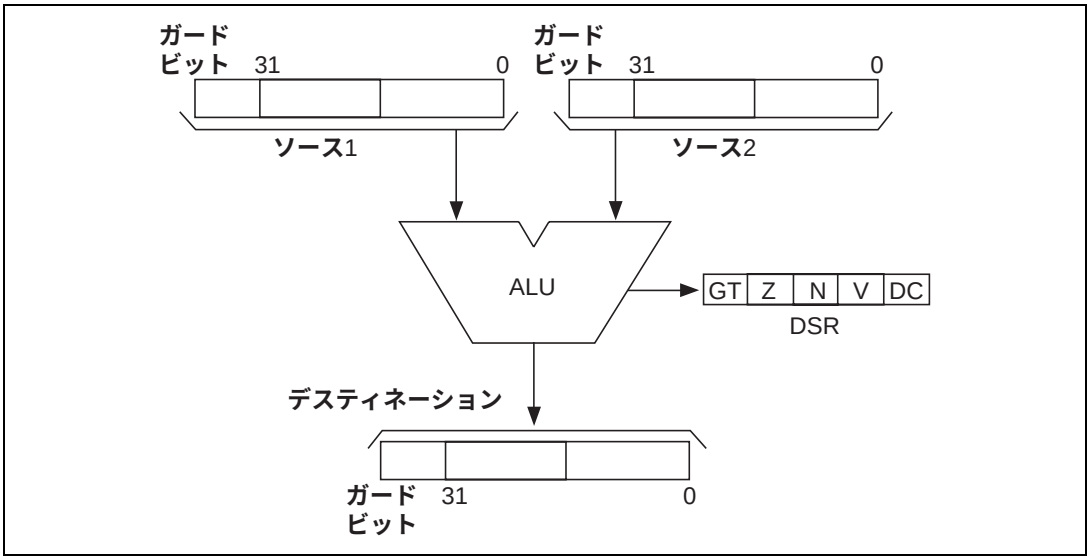


図 4.5 ALU **固定小数点算術演算の流れ**

メモリ読み出しのデスティネーションオペランドと ALU 演算のソースオペランドを同じにし、ALU 演算と同じ行にデータ転送命令のプログラムを書いた場合は、メモリアクセスステージ（MA）でメモリからロードされるデータは、ALU 演算命令のソースオペランドとしては使われません。この場合、先に実行された命令の結果が ALU 演算のソースオペランドとして使われ、そのあとで、データロード命令のデスティネーションオペランドとして更新されます。この流れを**図** 4.6 に示します。

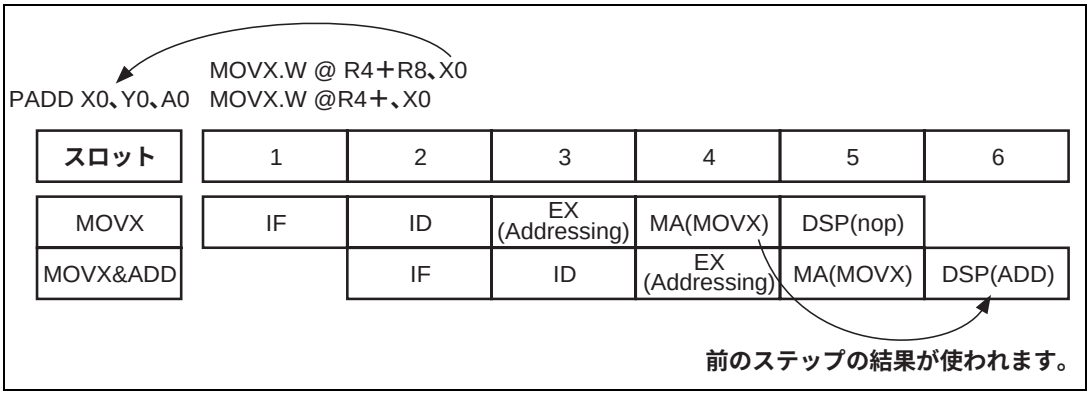


図 4.6　**処理の流れの例**

(2)　命令とオペランド

ALU 固定小数点算術演算の種類を**表** 4.15 に示します。それぞれのオペランドとレジスタとの対応を**表** 4.16 に示します。

表 4.15　ALU **固定小数点算術演算の種類**

ニーモニック	機能	ソース 1	ソース 2	デスティネーション
PADD	**加算**	Sx	Sy	Dz（Du）
PSUB	**減算**	Sx	Sy	Dz（Du）
PADDC	**キャリ付き加算**	Sx	Sy	Dz
PSUBC	**ボロー付き減算**	Sx	Sy	Dz
PCMP	**比較**	Sx	Sy	—
PCOPY	**データ複写**	Sx	—	Dz
		—	Sy	Dz
PABS	**絶対値**	Sx	—	Dz
		—	Sy	Dz
PNEG	**符号反転**	Sx	—	Dz
		—	Sy	Dz
PCLR	**ゼロクリア**	—	—	Dz

表 4.16　ALU **固定小数点算術演算のオペランドとレジスタとの対応**

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Du	Yes		Yes				Yes	Yes

【注】 Yes：**オペランドに使えるレジスタです**
Du： **乗算と組み合わされる場合のオペランドです**

(3)　DC **ビット**

DC ビットは、DSR レジスタの CS2～CS0 ビット（Condition Selection、状態選択）の指定に従って、次のようになります。

(a)　**キャリ／ボローモード：**　CS2～CS0＝000

DC ビットは演算の結果、MSB（Most Significant Bit）ビットからキャリまたはボローが発生したことを表します。ガードビットは関係ありません。DSR レジスタの初期状態は、このモードになっています。キャリとボローの発生例を**図** 4.7 に示します。

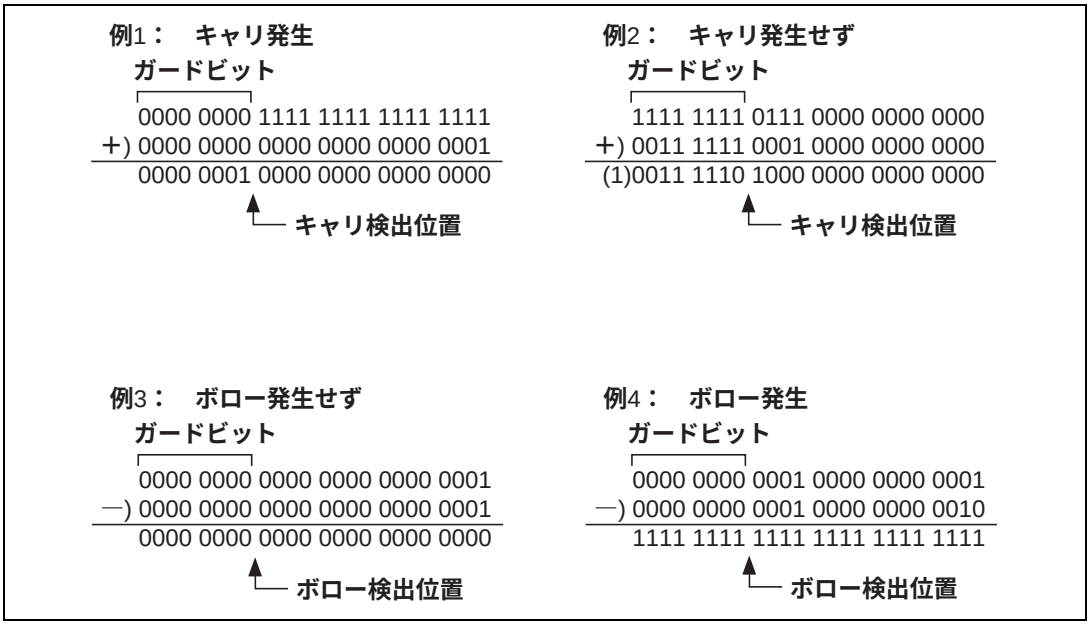


図 4.7　**キャリとボローの発生例**

(b)　**負値モード：**　CS2～CS0＝001

DC ビットは演算結果の MSB ビットの値と同じです。結果が負の値のとき DC ビットは 1 になります。ゼロまたは正の値のとき DC ビットは 0 になります。ALU 算術演算は常に 40 ビットで演算します。そのため正か負かの符号ビットは、デスティネーションオペランドの MSB ビットではなく、演算結果のガードビットを含めた MSB ビットで判定されます。正負の判定例を**図** 4.8 に示します。このモードの DC ビットは、状態ビットの N ビットの値と同じです。

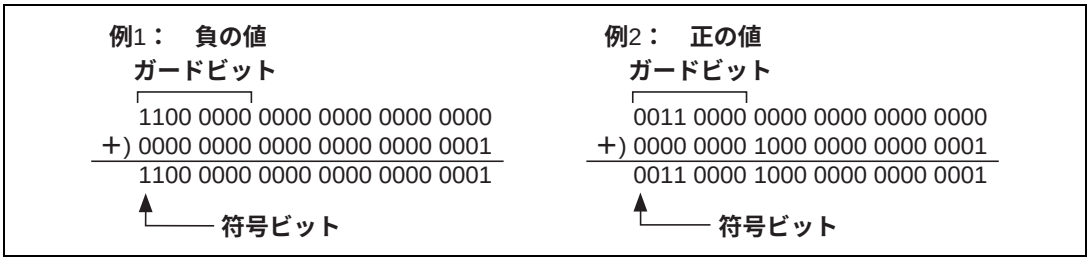


図 4.8 **正負の判定例**

(c) **ゼロ値モード：** CS2～CS0＝010

DC ビットは演算結果がゼロかどうかを表します。結果がゼロのとき DC ビットは 1 になり、結果がゼロでないとき DC ビットは 0 になります。このモードの DC ビットは、状態ビットの Z ビットの値と同じです。

(d) **オーバフローモード：** CS2～CS0＝011

DC ビットは演算の結果、オーバフローが発生したかどうかを表します。演算の結果がガードビットを除いて、デスティネーションレジスタの範囲を超えた場合、DC ビットが 1 にセットされます。DC ビットは、ガードビットがあっても、ガードビットがないと考えてオーバフローを判定します。そのため大きな数がガードビットを使う場合は DC ビットが常に 1 にセットされます。このモードの DC ビットは、状態ビットの V ビットの値と同じです。オーバフローの判定例を**図** 4.9 に示します。

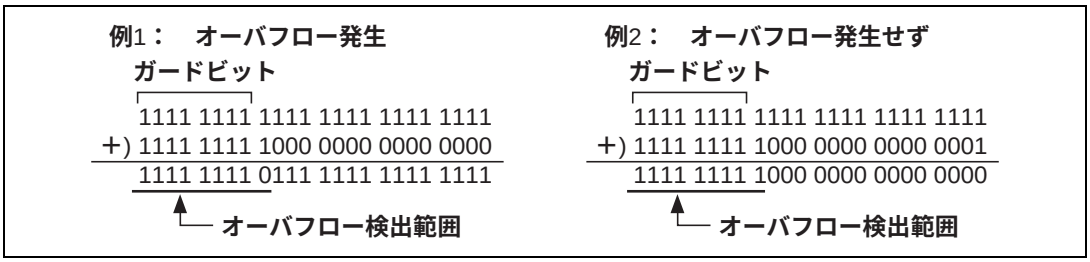


図 4.9 **オーバフローの判定例**

(e) **符号付き大モード：** CS2～CS0＝100

DC ビットは比較命令 PCMP の判定結果、ソース 1 データ（符号付き）がソース 2 データ（符号付き）より大きいかどうかを表します。ソース 1 データがソース 2 データより大きいときは比較の結果が正の値になるため、このモードは負値モードと似ています。しかしソース 1 データがソース 2 データより大きいときでも、デスティネーションオペランドの範囲を超えたときは、比較の結果の符号が負の値になります。DC ビットはこの場合更新されます。このモードの DC ビットは、状態ビットの GT ビットの値と同じです。このモードでの DC ビットを式で定義すると次のようになります。ただし、VR は結果がガードビット領域も含めてデスティネーションオペランドの表示範囲を越えた場合に真となる値です。

DC ビット＝～｛（N ビット ∧ VR）｜ Z ビット｝

DC ビットは、このモードで PCMP 命令を実行させると、CPU タイプの命令の CMP/GT 命令の結果を表す T ビットと同じ値になります。このモードでは、PCMP 命令以外でも上記定義に従って DC ビットは更新されます。

(f) **符号付き以上モード：** CS2～CS0＝101

DC ビットは、比較命令 PCMP の実行結果、ソース 1 データ（符号付き）がソース 2 データ（符号付き）より大きいかまたは等しいか、あるいはそうでないかを表します。そのため PCMP 命令は、このモードで DC ビットを判定するまえに、実行されます。このモードは、等しいかどうかを除いて符号付き大モードと似ています。このモードでの DC ビットを式で定義すると次のようになります。ただし、VR は結果がガードビット領域も含めてデスティネーションオペランドの表示範囲を超えた場合に真となる値です。

DC ビット＝～（N ビット ∧ VR）

DC ビットは、このモードで PCMP 命令を実行させると、SH コア命令の CMP/GE 命令の結果を表す T ビットと同じ値になります。このモードでは、PCMP 命令以外でも上記定義に従って DC ビットは更新されます。

(4) 状態ビット

状態ビットは次のように設定されます。

N ビット（Negative bit、負値ビット）は CS ビットで負値モードを指定したときの DC ビットの値と同じです。演算結果が負の値のとき N ビットは 1 になります。ゼロまたは正の値のとき N ビットは 0 になります。

Z ビット（Zero bit、ゼロビット）は CS ビットでゼロ値モードを指定したときの DC ビットの値と同じです。結果がゼロのとき Z ビットは 1 になり、結果がゼロでないとき Z ビットは 0 になります。

V ビット（Overflow bit、オーバフロービット）は CS ビットでオーバフローモードを指定したときの DC ビットの値と同じです。演算の結果、カードビットを除くデスティネーションレジスタの範囲を超えた場合、V ビットが 1 にセットされます。それ以外は 0 にクリアされます。

GT ビット（Greater Than bit、符号付き大ビット）は CS ビットで符号付き大モードを指定したときの DC ビットの値と同じです。比較の結果、ソース 1 データがソース 2 データより大きいとき、GT ビットが 1 にセットされます。それ以外は 0 にクリアされます。

(5) オーバフロー防止機能（飽和演算）

SR レジスタの S ビットを 1 にセットすると、DSP ユニットで実行されるすべての ALU 算術演算で、オーバフロー防止機能が実行されます。演算結果がオーバフローしたとき最大値（正）または最小値（負）が格納されます。

4.9 ALU 整数演算

ALU 整数演算は基本的には、上位ワード（上位 16 ビット、ビット 31～16）とガードビット 8 ビットとの 24 ビットの演算です。ALU 整数算術演算では、ソースオペランドの下位ワード（下位 16 ビット、ビット 15～0）は無視され、デスティネーションオペランドの下位ワードは 0 クリアされます。ソースオペランドにガードビットがない場合は符号ビットがガードビットとして拡張されて格納されます。デスティネーションオペランドにガードビットがない場合は演算結果のガードビットを除いた上位ワードがデスティネーションレジスタの上位ワードに格納されます。

整数演算は基本的に ALU 固定小数点算術演算と同じです。整数演算の演算命令はインクリメントとデクリメント命令の 2 種類しかなく、第 2 オペランドは実質的には＋1 か－1 です。16 ビットの整数データ（ワードデータ）が DSP レジスタにロードされ、上位ワードに格納されます。そしては DSP レジスタの上位ワードを使って演算されます。ガードビットがある場合は、ガードビットも有効です。これらの動作は、パイプラインの流れの DSP ステージと名付けられた最終ステージで行われます。

ALU 整数算術演算が実行されるときは、DSR レジスタの DC、N、Z、V、GT ビットは、基本的には演算の結果で更新されます。これは ALU 固定小数点演算と同じです。

条件付き命令の場合は、指定された条件が成立し、命令が実行されたときでも、条件ビットとフラグは更新されません。無条件命令の場合は、演算結果に従って常に更新されます。

ALU 整数演算の流れを**図** 4.10 に示します。

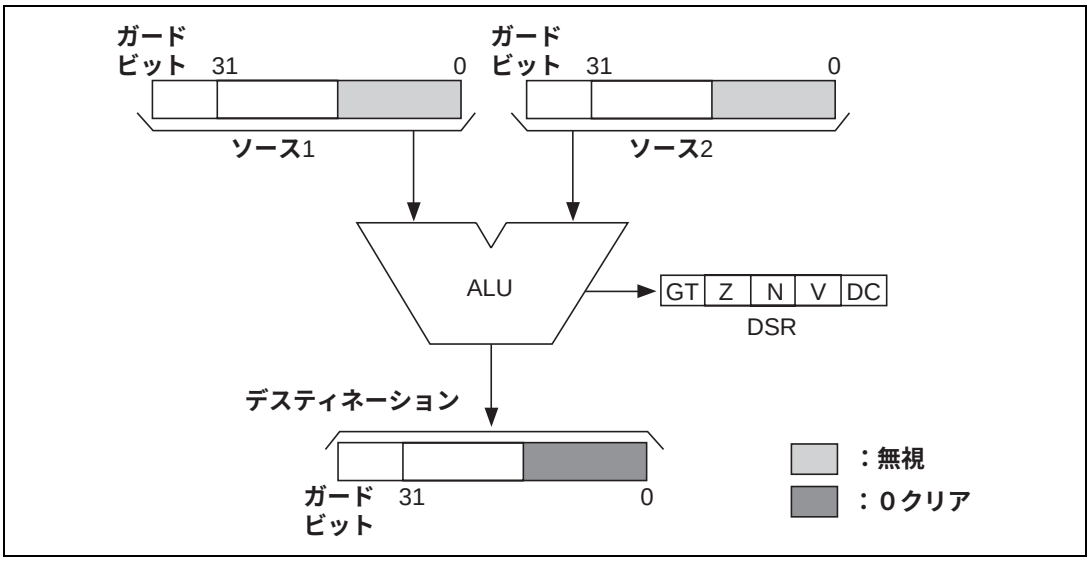


図 4.10 ALU **整数演算の流れ**

ALU 整数演算の種類を**表** 4.17 に示します。それぞれのオペランドのレジスタとの対応を**表** 4.18 に示します。

表 4.17　ALU **整数演算の種類**

ニーモニック	**機能**	**ソース** 1	**ソース** 2	**デスティネーション**
PINC	1 **インクリメント**	Sx	(+1)	Dz
		(+1)	Sy	Dz
PDEC	1 **デクリメント**	Sx	(−1)	Dz
		(−1)	Sy	Dz

表 4.18　ALU **整数演算のオペランドとレジスタとの対応**

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

【注】 Yes**：オペランドに使えるレジスタです**

オーバフロー防止機能（飽和演算）を実行するためには、SR レジスタの S ビットを 1 にセットします。DSP タイプ命令で実行される ALU 整数算術演算に対して、オーバフロー防止機能を指定できます。演算結果がオーバフローしたとき最大値（正）または最小値（負）が格納されます。

4.10 ALU 論理演算

(1) 演算機能

ALU 論理演算もレジスタ間で実行されます。それぞれのソース、デスティネーションオペランドは、独立に、DSP レジスタの一つを選べます。このタイプの演算はそれぞれのオペランドの上位ワードだけを使います。ソースオペランドの下位ワードとガードビットは無視され、デスティネーションオペランドの下位ワードとガードビットは 0 クリアされます。これらの動作は、パイプラインの流れの DSP ステージと名付けられた最終ステージで行われます。

ALU 論理演算が実行されると、DSR レジスタの DC ビット、N、Z、V、GT フラグは、基本的には演算の結果で更新されます。条件付き命令の場合は、指定された条件が成立し、命令が実行されたときでも、条件ビットとフラグは更新されません。無条件命令の場合は、演算結果に従って常に更新されます。DC ビットは CS ビットの指定に従って更新されます。ALU 論理演算の流れを**図** 4.11 に示します。

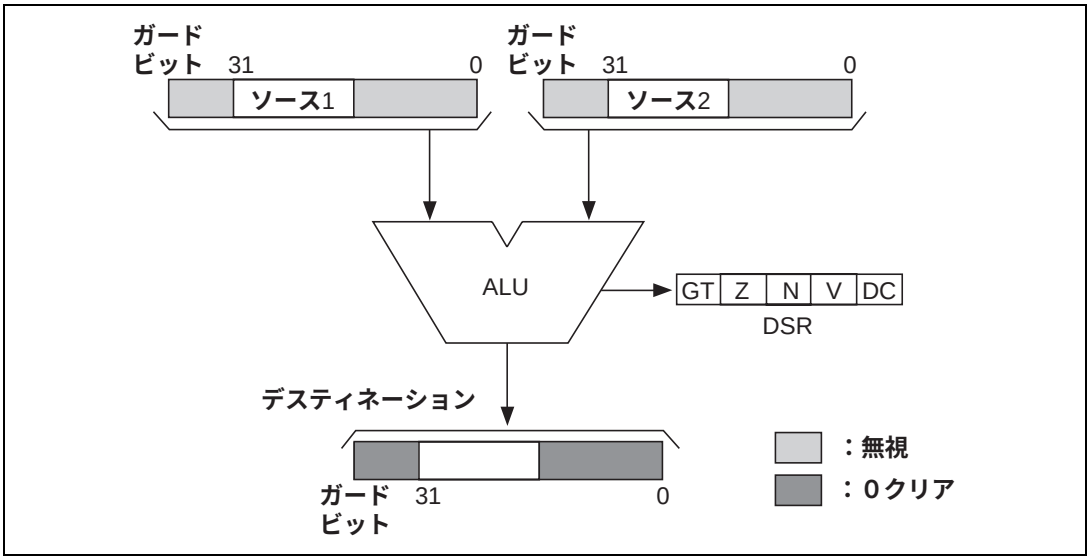


図 4.11 ALU **論理演算の流れ**

(2) 命令とオペランド

ALU 論理演算の種類を**表** 4.19 に示します。それぞれのオペランドのレジスタとの対応は、ALU 固定小数点演算と同じで、**表** 4.20 に示します。

表 4.19　ALU 論理演算の種類

ニーモニック	機能	ソース 1	ソース 2	デスティネーション
PAND	論理積	Sx	Sy	Dz
POR	論理和	Sx	Sy	Dz
PXOR	排他的論理和	Sx	Sy	Dz

表 4.20　ALU 論理演算のオペランドとレジスタとの対応

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

【注】 Yes：オペランドに使えるレジスタです

(3)　DC ビット

論理演算の DC ビットは次のように設定されます。

(a)　**キャリ／ボローモード：**　CS2～CS0＝000

DC ビットは常に 0 にクリアされます。

(b)　**負値モード：**　CS2～CS0＝001

DC ビットは演算結果のビット 31 の値になります。このモードの DC ビットは N ビットの値と同じです。

(c)　**ゼロ値モード：**　CS2～CS0＝010

DC ビットは演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、Z ビットの値と同じです。

(d)　**オーバフローモード：**　CS2～CS0＝011

DC ビットは常に 0 にクリアされます。このモードの DC ビットは、V ビットの値と同じです。

(e)　**符号付き大モード：**　CS2～CS0＝100

DC ビットは常に 0 にクリアされます。このモードの DC ビットは、GT ビットの値と同じです。

(f)　**符号付き以上モード：**　CS2～CS0＝101

DC ビットは常に 0 にクリアされます。

(4)　状態ビット

状態ビットは次のように設定されます。

- N ビットは演算結果のビット 31 の値になります。
- Z ビットは演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。
- V ビットは常に 0 にクリアされます。
- GT ビットは常に 0 にクリアされます。

4.11 固定小数点乗算

DSP タイプ命令の乗算は、符号付き単精度乗算です。この演算は 1 サイクルで処理が完了します。もし倍精度の乗算が必要な場合は、CPU タイプ命令の倍精度演算を使います。

乗算の結果は基本的に 32 ビットの演算結果が得られます。デスティネーションオペランドにガードビットを持つレジスタを指定した場合は、符号拡張されます。

DSP タイプ命令の乗算は整数の計算ではなく、固定小数点算術演算です。そのため、それぞれ、定数と被乗数の上位ワードが、MAC 演算器に入力されます。CPU タイプ命令の乗算では、両方のオペランドの下位ワードが MAC 演算器に入力されます。そのため演算結果は DSP タイプの命令と CPU タイプの命令では異なります。CPU タイプ命令の乗算の結果はデスティネーションの LSB にあわせられますが、DSP タイプ命令の固定小数点乗算の演算結果は MSB にあわせられます。そのため固定小数点乗算の演算結果の LSB は常に 0 になります。

固定小数点乗算の流れを**図** 4.12 に示します。

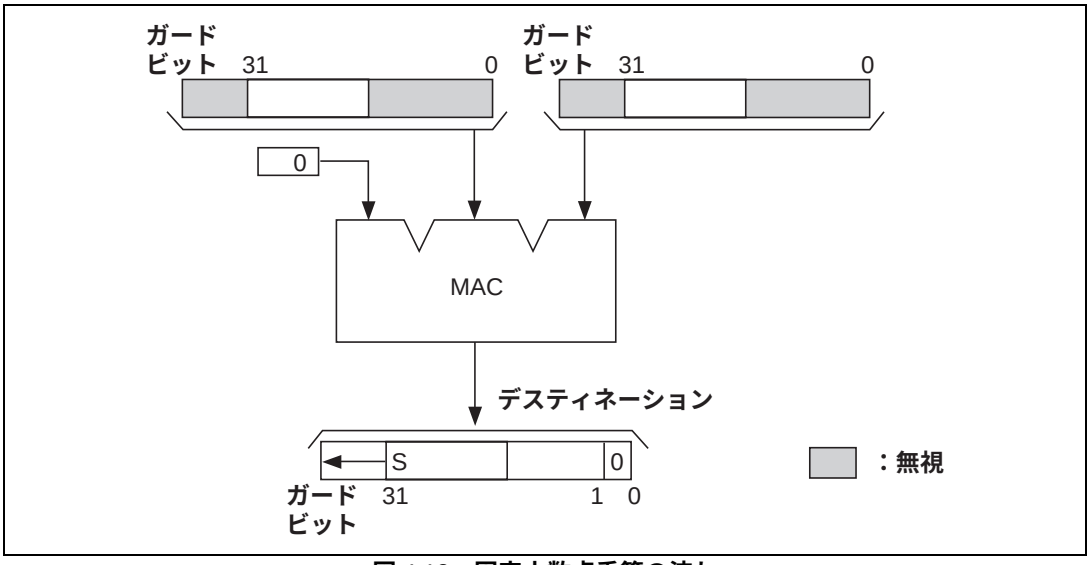


図 4.12 **固定小数点乗算の流れ**

固定小数点乗算の種類を**表** 4.21 に示します。それぞれのオペランドのレジスタとの対応を**表** 4.22 に示します。

表 4.21 **固定小数点乗算の種類**

ニーモニック	機能	ソース 1	ソース 2	デスティネーション
PMULS	符号付き乗算	Se	Sf	Dg

表 4.22 **固定小数点乗算のオペランドとレジスタとの対応**

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Se	Yes	Yes	Yes					Yes
Sf	Yes		Yes	Yes				Yes
Dg					Yes	Yes	Yes	Yes

【注】 Yes：**オペランドに使えるレジスタです**

DSP タイプ命令の固定小数点乗算は 16 ビット×16 ビットの単精度演算を 1 サイクルで完了します。これ以外の乗算は従来の CPU タイプ命令の乗算と同じです。

乗算命令は DSR レジスタのどの状態ビット、DC、N、Z、V、GT ビットも更新しません。

オーバフロー防止機能（飽和演算）は DSP タイプ命令の乗算でも有効です。SR レジスタの S ビットを 1 にセットして指定します。演算結果の値がオーバフローまたはアンダフローしたときそれぞれ最大値または最小値になります。DSP タイプ命令の固定小数点乗算では、H'8000×H'8000（（－1.0）×（－1.0））の場合だけ、オーバフローが発生します。S ビットが 0 のとき、この演算結果は H'8000 0000 となり、これは－1.0 を意味し、正しい値＋1.0 になりません。S ビットが 1 のとき、オーバフロー防止機能が働いて、この結果は H'007FFF FFFF となります。

4.12 シフト演算

シフト演算のシフト量は、レジスタで指定するかまたは直接イミディエイト値で指定します。その他のソースオペランドとデスティネーションオペランドはレジスタで指定します。シフト演算には算術シフトと論理シフトの 2 種類があります。演算の種類を**表** 4.23 に示します。イミディエイトオペランドを除いたそれぞれのオペランドのレジスタとの対応は ALU 固定小数点演算と同じです。対応を**表** 4.24 に示します。

表 4.23 **シフト演算の種類**

ニーモニック	**機能**	**ソース** 1	**ソース** 2	**デスティネーション**
PSHA Sx、Sy、Dz	**算術シフト**	Sx	Sy	Dz
PSHL Sx、Sy、Dz	**論理シフト**	Sx	Sy	Dz
PSHA #imm、Dz	**イミディエイトデータ付き算術シフト**	Dz	imm1	Dz
PSHL #imm、Dz	**イミディエイトデータ付き論理シフト**	Dz	imm2	Dz

－32<=imm1<=＋32、－16<=imm2<=＋16

表 4.24 **シフト演算のオペランドとレジスタとの対応**

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

【注】 Yes：**オペランドに使えるレジスタです**

4.12.1 算術シフト演算

(1) 演算機能

ALU 算術シフト演算は、32 ビット精度とガードビット 8 ビットとの 40 ビットの演算です。基本的にはレジスタ間で実行されます。ソースオペランドにガードビットがない場合は符号ビットがガードビットとして転記されます。デスティネーションオペランドにガードビットがない場合は演算結果の下位 32 ビットがデスティネーションレジスタに格納されます。

この算術シフトではソース 1 オペランドとデスティネーションオペランドはすべてのビットが有効です。シフト量を指定するソース 2 オペランドは整数データです。ソース 2 オペランドはレジスタまたはイミディエイトオペランドで指定します。有効なシフト量は−32 から+32 までです。ここで、負の値は右のシフトを意味し、正の値は左のシフトを意味します。ソース 2 オペランドとして−64 から+63 までを指定することはできますが、有効なシフト量は−32 から+32 までですので、無効な数値を指定した場合の結果は保証されません。シフト量をイミディエイト値で指定した場合は、ソース 1 オペランドはデスティネーションオペランドと同じでなければなりません。この演算動作は、固定小数点演算と同じように、パイプラインの流れの最後の DSP ステージで実行されます。

算術シフト演算が実行されるときは、DSR レジスタの DC、N、Z、V、GT ビットは、基本的には演算の結果に従って更新されます。これは ALU 固定小数点演算と同じです。条件付き命令の場合は、指定された条件が成立し、命令が実行されたときでも、状態ビットは更新されません。無条件命令の場合は、演算結果に従って常に更新されます。

算術シフト演算の流れを**図** 4.13 に示します。

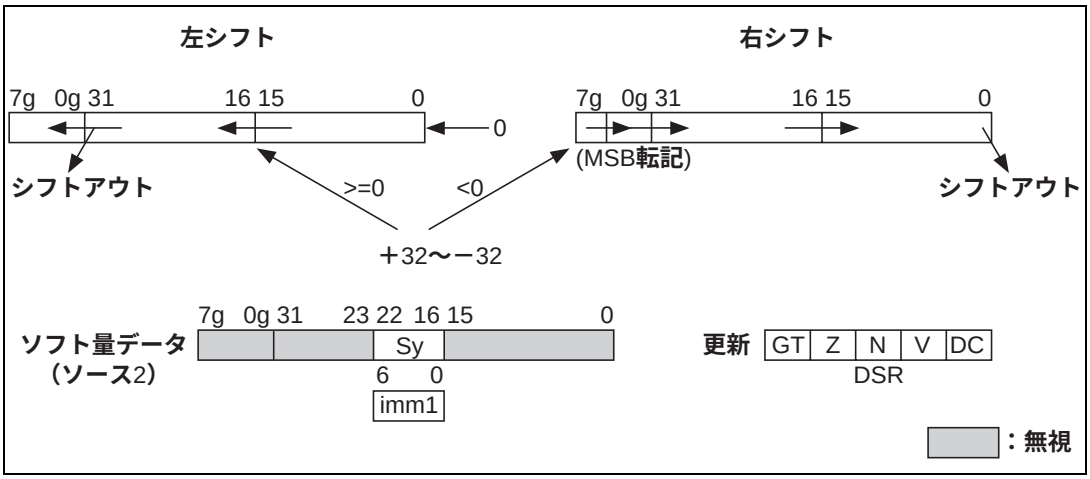


図 4.13 **算術シフト演算の流れ**

(2) DC ビット

DC ビットは CS ビットで指定されたモードに従って、次のように更新されます。

(a) **キャリ／ボローモード：** CS2〜CS0＝000

DC ビットは演算の結果、最後にシフトして押し出されたビットの値になります。

(b) **負値モード：** CS2〜CS0＝001

ビットは結果が負の値のとき 1 にセットされ、ゼロまたは正の値のとき 0 にクリアされます。このモードの DC ビットは N ビットの値と同じです。

(c) **ゼロ値モード：** CS2～CS0＝010

DC ビットは演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、Z ビットの値と同じです。

(d) **オーバフローモード：** CS2～CS0＝011

オーバフローが発生したとき 1 にセットされます。このモードの DC ビットは、V ビットの値と同じです。

(e) **符号付き大モード：** CS2～CS0＝100

DC ビットは常に 0 にクリアされます。このモードの DC ビットは、GT ビットの値と同じです。

(f) **符号付き以上モード：** CS2～CS0＝101

DC ビットは常に 0 にクリアされます。

(3) **状態ビット**

状態ビットは次のように更新されます。

N ビットは ALU 固定小数点算術演算の結果と同じです。演算結果が負の値のとき 1 にセットされ、ゼロまたは正の値のとき 0 にクリアされます。

Z ビットは ALU 固定小数点算術演算の結果と同じです。演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。

V ビットは ALU 固定小数点算術演算の結果と同じです。オーバフローが発生したとき 1 にセットされます。

GT ビットは常に 0 にクリアされます。

(4) **オーバフロー防止機能（飽和演算）**

SR レジスタの S ビットを 1 にセットすると、DSP ユニットで実行される算術シフト演算で、オーバフロー防止機能が実行されます。演算結果がオーバフローしたとき最大値（正）または最小値（負）が格納されます。

4.12.2 論理シフト演算

(1) **演算機能**

論理シフト演算は、ソース 1 オペランドとデスティネーションオペランドの上位ワードを使います。オペランドのガードビットと下位ワードは、ALU 論理演算と同じに、無視されます。シフト量を指定するソース 2 オペランドは整数データです。ソース 2 オペランドはレジスタまたはイミディエイトオペランドで指定します。有効なシフト量は－16 から＋16 までです。ここで、負の値は右のシフトを意味し、正の値は左のシフトを意味します。ソース 2 オペランドとして－32 から＋31 までを指定することはできますが、有効なシフト量は－16 から＋16 までですので、無効な数値を指定した場合の結果は保証されません。シフト量をイミディエイト値で指定した場合は、ソース 1 オペランドはデスティネーションオペランドと同じでなければなりません。この演算動作は、パイプラインの流れの最後の DSP ステージで実行されます。

論理シフト演算が実行されるときは、DSR レジスタの DC、N、Z、V、GT ビットは、基本的には演算の結果に従って更新されます。これは ALU 論理演算と同じです。条件付き命令の場合は、指定された条件が成立し、命令が実行されたときでも、状態ビットは更新されません。無条件命令の場合は、演算結果に従って常に更新されます。

論理シフト演算の流れを**図** 4.14 に示します。

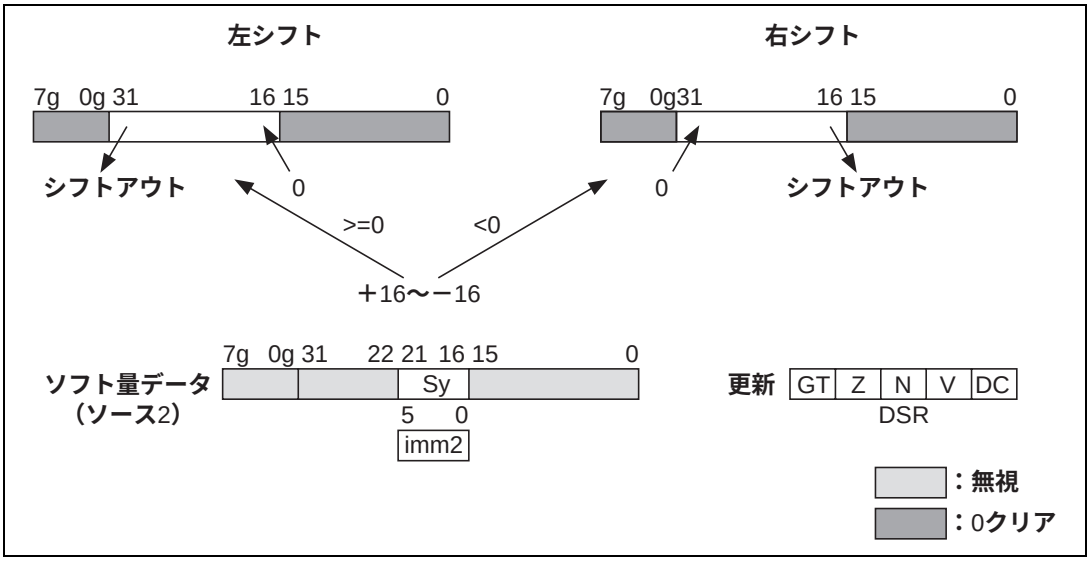


図 4.14 論理シフト演算の流れ

(2) DC **ビット**

CS ビットで指定されたモードに従って、次のように更新されます。

(a) **キャリ／ボローモード：** CS2〜CS0＝000

DC ビットは演算の結果、最後にシフトして押し出されたビットの値になります。

(b) **負値モード：** CS2〜CS0＝001

DC ビットは演算結果のビット 31 の値が格納されます。このモードの DC ビットは N ビットの値と同じです。

(c) **ゼロ値モード：** CS2〜CS0＝010

DC ビットは演算結果がすべてゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、Z ビットの値と同じです。

(d) **オーバフローモード：** CS2〜CS0＝011

DC ビットは常に 0 にクリアされます。このモードの DC ビットは、V ビットの値と同じです。

(e) **符号付き大モード：** CS2〜CS0＝100

DC ビットは常に 0 にクリアされます。このモードの DC ビットは、GT ビットの値と同じです。

(f) **符号付き以上モード：** CS2〜CS0＝101

DC ビットは常に 0 にクリアされます。

(3) **状態ビット**

状態ビットは次のように更新されます。

- N ビットは ALU 論理演算の結果と同じです。演算結果のビット 31 の値が格納されます。
- Z ビットは ALU 論理演算の結果と同じです。演算結果がすべてゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。
- V ビットは常に 0 にクリアされます。
- GT ビットは常に 0 にクリアされます。

4.13 MSB 検出命令

(1) 演算機能

MSB 検出命令（PDMSB: Most Significant Bit detection）は、データを正規化するためのシフト量を求めるものです。

演算結果は、ALU 整数演算と同じに、基本的には上位 16 精度と 8 ビットのガードビットとの 24 ビットが有効です。デスティネーションオペランドがガードビットのないレジスタの場合は、デスティネーションレジスタ上位 16 ビットに格納されます。

MSB 検出命令はソースオペランドのすべてのビットを対象にしていますが、演算結果は整数データとして求められます。これは、正規化のためのシフト量は、算術シフト演算の整数データが必要なためです。この演算動作は、固定小数点演算と同じように、パイプラインの流れの最後の DSP ステージで実行されます。

PDMSB 命令が実行されるときは、DSR レジスタの DC、N、Z、V、GT ビットは、基本的には演算の結果に従って更新されます。条件付き命令の場合は、指定された条件が成立し、命令が実行されたときでも、状態ビットは更新されません。無条件命令の場合は、演算結果に従って常に更新されます。

MSB 検出命令の流れを**図** 4.15 に示します。ソースデータとデスティネーションデータとの関係を**表** 4.25 に示します。

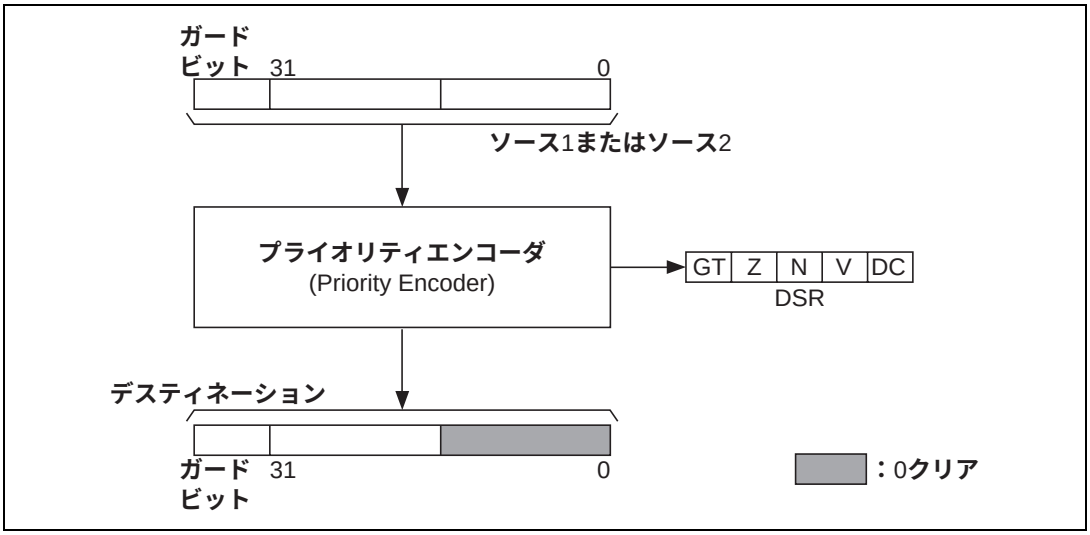


図 4.15 MSB **検出の流れ**

表 4.25 **ソースデータとデスティネーション結果との関係**

ソースデータ																結　果								
ガードビット					上位ワード					下位ワード						ガード	上位ワード							
7g	6g	– – –	1g	0g	31	30	29	28	– – –	3	2	1	0			7g～0g	31～22	21	20	19	18	17	16	10進数
0	0	– – –	0	0	0	0	0	0	– – –	0	0	0	0			all 0	all 0	0	1	1	1	1	1	+31
0	0	– – –	0	0	0	0	0	0	– – –	0	0	0	1			all 0	all 0	0	1	1	1	1	0	+30
0	0	– – –	0	0	0	0	0	0	– – –	0	0	1	*			all 0	all 0	0	1	1	1	0	1	+29
0	0	– – –	0	0	0	0	0	0	– – –	0	1	*	*			all 0	all 0	0	1	1	1	0	0	+28
0	0	– – –	0	0	0	0	0	1	– – –	*	*	*	*			all 0	all 0	0	0	0	0	1	0	+2
0	0	– – –	0	0	0	0	1	*	– – –	*	*	*	*			all 0	all 0	0	0	0	0	0	1	+1
0	0	– – –	0	0	0	1	*	*	– – –	*	*	*	*			all 0	all 0	0	0	0	0	0	0	0
0	0	– – –	0	0	1	*	*	*	– – –	*	*	*	*			all 1	all 1	1	1	1	1	1	1	-1
0	0	– – –	0	1	*	*	*	*	– – –	*	*	*	*			all 1	all 1	1	1	1	1	1	0	-2
0	1	– – –	*	*	*	*	*	*	– – –	*	*	*	*			all 1	all 1	1	1	1	0	0	0	-8
1	0	– – –	*	*	*	*	*	*	– – –	*	*	*	*			all 1	all 1	1	1	1	0	0	0	-8
1	1	– – –	1	0	*	*	*	*	– – –	*	*	*	*			all 1	all 1	1	1	1	1	1	0	-2
1	1	– – –	1	1	0	*	*	*	– – –	*	*	*	*			all 1	all 1	1	1	1	1	1	1	-1
1	1	– – –	1	1	1	0	*	*	– – –	*	*	*	*			all 0	all 0	0	0	0	0	0	0	0
1	1	– – –	1	1	1	1	0	*	– – –	*	*	*	*			all 0	all 0	0	0	0	0	0	1	+1
1	1	– – –	1	1	1	1	1	0	– – –	*	*	*	*			all 0	all 0	0	0	0	0	1	0	+2
1	1	– – –	1	1	1	1	1	1	– – –	1	0	*	*			all 0	all 0	0	1	1	1	0	0	+28
1	1	– – –	1	1	1	1	1	1	– – –	1	1	0	*			all 0	all 0	0	1	1	1	0	1	+29
1	1	– – –	1	1	1	1	1	1	– – –	1	1	1	0			all 0	all 0	0	1	1	1	1	0	+30
1	1	– – –	1	1	1	1	1	1	– – –	1	1	1	1			all 0	all 0	0	1	1	1	1	1	+31

【注】 * 'Don't care'**ビット、影響なし**

(2) **命令とオペランド**

MSB 検出命令の種類を**表** 4.26 に示します。それぞれのオペランドのレジスタとの対応は、ALU 固定小数点演算と同じです。その対応を**表** 4.27 に示します。

表 4.26 MSB **検出の種類**

ニーモニック	機能	ソース 1	ソース 2	デスティネーション
PDMSB	MSB **検出**	Sx	—	Dz
		—	Sy	Dz

表 4.27 MSB **検出のオペランドとレジスタとの対応**

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

【注】 Yes**：オペランドに使えるレジスタです**

(3) DC ビット

DC ビットは CS ビットで指定されたモードに従って、次のように更新されます。

(a) **キャリ／ボローモード：** CS2～CS0＝000

DC ビットは常に 0 にクリアされます。

(b) **負値モード：** CS2～CS0＝001

DC ビットは演算結果が負の値のとき 1 にセットされ、ゼロまたは正の値のとき 0 にクリアされます。このモードの DC ビットは N ビットの値と同じです。

(c) **ゼロ値モード：** CS2～CS0＝010

DC ビットは演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、Z ビットの値と同じです。

(d) **オーバフローモード：** CS2～CS0＝011

常に 0 にクリアされます。このモードの DC ビットは、V ビットの値と同じです。

(e) **符号付き大モード：** CS2～CS0＝100

演算結果が正の値のとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、GT ビットの値と同じです。

(f) **符号付き以上モード：** CS2～CS0＝101

DC ビットは演算結果が正またはゼロの値のとき 1 にセットされます。それ以外は 0 にクリアされます。

(4) **状態ビット**

状態ビットは次のように更新されます。

- N ビットは ALU 整数演算の結果と同じです。演算結果が負の値のとき 1 にセットされ、ゼロまたは正の値のとき 0 にクリアされます。
- Z ビットは ALU 整数演算の結果と同じです。演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。
- V ビットは常に 0 にクリアされます。
- GT ビットは ALU 整数演算の結果と同じです。演算結果が正の値のとき 1 にセットされます。それ以外は 0 にクリアされます。

4.14 丸め処理

(1) 演算機能

SH-DSP には 32 ビットの数値を 16 ビットに丸める機能があります。ガードビットがある場合は、40 ビットの数値を 24 ビットに丸めます。丸めの命令が実行されると、H'0000 8000 がソースオペランドに加えられ、そのあとで下位ワードは 0 でクリアされます。

丸め処理はソースオペランドとデスティネーションオペランドともにすべてのビットデータを使います。この演算動作は、固定小数点演算と同じように、パイプラインの流れの最後の DSP ステージで実行されます。

丸め処理命令は無条件命令です。そのため、DSR レジスタの DC、N、Z、V、GT ビットは、常に演算の結果に従って更新されます。

丸め処理の流れを**図** 4.16 に示します。丸め処理の定義を**図** 4.17 に示します。

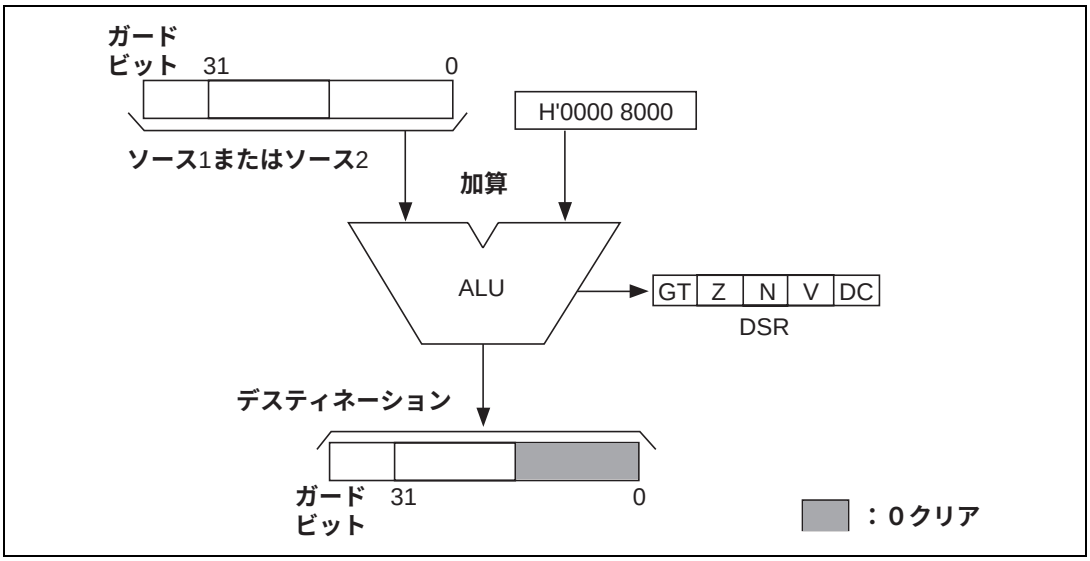


図 4.16 **丸め処理の流れ**

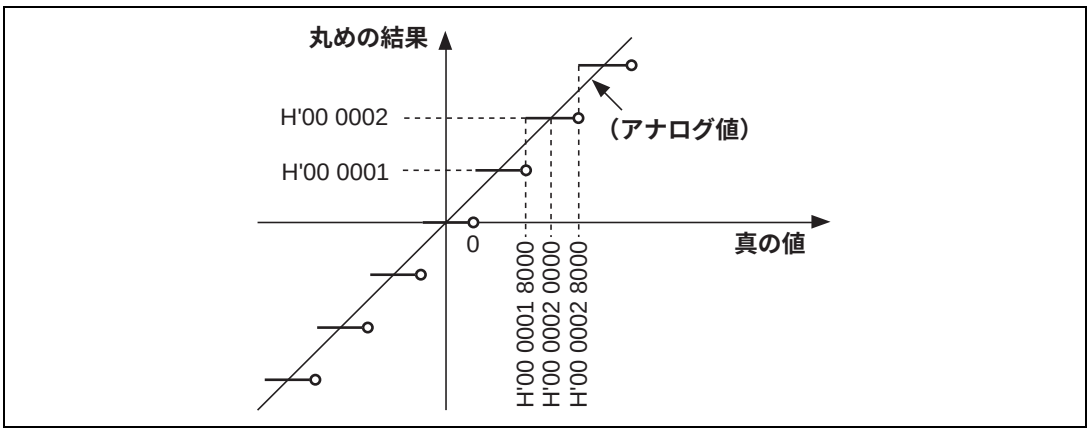


図 4.17 **丸め処理の定義**

(2) 命令とオペランド

命令の種類を**表** 4.28 に示します。オペランドとレジスタの対応は ALU 固定小数点演算と同じです。対応を**表** 4.29 に示します。

表 4.28 **固定小数点乗算の種類**

ニーモニック	機能	ソース 1	ソース 2	デスティネーション
PRND	丸め処理	Sx	—	Dz
		—	Sy	Dz

表 4.29 **固定小数点乗算のオペランドとレジスタとの対応**

オペランド	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

【注】 Yes：**オペランドに使えるレジスタです**

(3) DC ビット

DC ビットは CS ビットで指定されたモードに従って、次のように更新されます。状態ビットの更新は ALU 固定小数点算術演算と同じです。

(a) **キャリ／ボローモード：** CS2～CS0＝000

DC ビットは演算の結果、MSB ビットからキャリまたはボローが発生したとき 1 にセットされます。それ以外は 0 にクリアされます。

(b) **負値モード：** CS2～CS0＝001

DC ビットは演算結果が負の値のとき 1 にセットされ、ゼロまたは正の値のとき 0 にクリアされます。このモードの DC ビットは N ビットの値と同じです。

(c) **ゼロ値モード：** CS2～CS0＝010

DC ビットは演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、Z ビットの値と同じです。

(d) **オーバフローモード：** CS2～CS0＝011

DC ビットはオーバフローが発生すると 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、V ビットの値と同じです。

(e) **符号付き大モード：** CS2～CS0＝100

演算結果が正の値のとき 1 にセットされます。それ以外は 0 にクリアされます。このモードの DC ビットは、GT ビットの値と同じです。

(f) **符号付き以上モード：** CS2～CS0＝101

DC ビットは演算結果が正またはゼロの値のとき 1 にセットされます。それ以外は 0 にクリアされます。

(4) 状態ビット

状態ビットは次のように更新されます。状態ビットの更新は ALU 固定小数点算術演算と同じです。

- N ビットは ALU 固定小数点算術演算の結果と同じです。演算結果が負の値のとき 1 にセットされ、ゼロまたは正の値のとき 0 にクリアされます。
- Z ビットは ALU 固定小数点算術演算の結果と同じです。演算結果がゼロのとき 1 にセットされます。それ以外は 0 にクリアされます。
- V ビット ALU 固定小数点算術演算の結果と同じです。オーバフローが発生したとき 1 にセットされます。それ以外は 0 にクリアされます。
- GT ビットは ALU 固定小数点算術演算 ALU 整数演算の結果と同じです。演算結果が正の値のとき 1 にセットされます。それ以外は 0 にクリアされます。

(5) オーバフロー防止機能（飽和演算）

SR レジスタの S ビットを 1 にセットすると、DSP ユニットで実行されるすべての丸め処理に対して、オーバフロー防止機能を実行します。演算結果の値がオーバフローしたときそれぞれ正の最大値または負の最小値になります。

4.15 状態選択ビット（CS）と DSP 状態ビット（DC）

DSP タイプ命令には無条件命令と条件付き命令があります。無条件命令は DSP 状態ビット（DC）に関係なく実行され、条件付き命令は DC ビットを判定して実行するかしないかが決まります。無条件命令では DSR レジスタの DC ビットおよび状態ビット（N、Z、V、GT）は ALU 演算またはシフト演算の結果によって更新されます。条件付き命令は実行するしないにかかわらず、DC ビットおよび状態ビット（N、Z、V、GT）を更新しません。DC ビットは状態選択ビット（CS）の指定に従って更新されます。更新は、算術演算、論理演算、算術シフト、論理シフトによってそれぞれ異なります。CS ビットと DC ビットの関係を**表** 4.30 に示します。

表 4.30 **状態選択ビット（CS）と** DSP **状態ビット（DC）**

CS **ビット**				
2	1	0	**状態モード**	**説明**
0	0	0	**キャリ/ボローモード**	ALU **算術演算の結果キャリまたはボローが生じたとき、**DC **ビットは** 1 **にセットされます。それ以外は** 0 **にクリアされます。** **論理演算では** DC **ビットは常に** 0 **にクリアされます。** **シフト演算（**PSHA**、**PSHL **命令）のとき、最後にシフトアウトされた（外に出た）ビットが** DC **ビットに転記されます。**
0	0	1	**負値モード**	ALU **算術演算または算術シフト（**PSHA**）演算のとき、ガードビットを含めて、結果の** MSB **ビットが** DC **ビットに転記されます。** ALU **論理演算または論理シフト（**PSHL**）演算のとき、ガードビットを除いて、結果の** MSB **ビットが** DC **ビットに転記されます。**
0	1	0	**ゼロ値モード**	ALU **演算またはシフト演算の結果がすべてゼロ（**0**）のとき、**DC **ビットは** 1 **にセットされます。それ以外は** 0 **にクリアされます。**
0	1	1	**オーバフローモード**	ALU **算術演算または算術シフト（**PSHA**）演算のとき、ガードビットを除いて、演算結果がデスティネーションレジスタの値の範囲を超えたとき、**DC **ビットは** 1 **にセットされます。それ以外は** 0 **にクリアされます。** ALU **論理演算または論理シフト（**PSHL**）演算のとき、**DC **ビットは常に** 0 **にクリアされます。**
1	0	0	**符号付き大モード**	**このモードは符号付き以上モードと似ていますが、演算結果がゼロ（**0**）のとき** DC **ビットは** 0 **にクリアされます。ガードビット部分を含めても演算結果が表現可能範囲を越えた時、真となる状態を** VR **とすると以下のように計算されます。** DC **ビット** = ~ {(N **ビット** ^ VR)	Z **ビット** }; **算術演算の場合** DC **ビット** = 0 ; **論理演算の場合**
1	0	1	**符号付き以上モード**	ALU **算術演算または算術シフト（**PSHA**）演算のとき、かつ結果がオーバフローしないとき、負値モードの** DC **ビットを反転した値になります。ガードビット部分を含めても結果が表現可能範囲を超えたとき、負値モードの** DC **ビットと同じ値になります。** ALU **論理演算または論理シフト（**PSHL**）演算のとき、**DC **ビットは常に** 0 **にクリアされます。ガードビット部分を含めても演算結果が表現範囲を超えたとき、真となる状態を** VR **とすると、以下のように計算されます。** DC **ビット** = ~ (N **ビット** ^ VR) **；算術演算の場合** DC **ビット** = 0 **；論理演算の場合**
1	1	0	**予約コード**	
1	1	1		

4.16 オーバフロー防止機能（飽和演算）

オーバフロー防止機能（飽和演算）は SR レジスタの S ビットで指定します。この機能は DSP ユニットで実行されるすべての算術演算、および CPU タイプ命令で実行される積和演算に有効です。演算結果がガードビットを除いて表現できる 2 の補数の範囲を超えたときオーバフローが発生します。

DSP タイプ命令の固定小数点算術演算のオーバフローの定義を**表** 4.31 に、整数算術演算のオーバフローの定義を**表** 4.32 に示します。

表 4.31 **固定小数点算術演算のオーバフローの定義**

符号	オーバフロー状態	最大値／最小値	16 進表示
正	結果＞$1-2^{-31}$	$1-2^{-31}$	007FFFFFFF
負	結果＜−1	−1	FF80000000

表 4.32 **整数算術演算のオーバフローの定義**

符号	オーバフロー状態	最大値／最小値	16 進表示
正	結果＞$2^{15}-1$	$2^{15}-1$	007FFF****
負	結果＜-2^{15}	-2^{15}	FF8000****

【注】＊： ‘Don’t care’、**影響なし**

オーバフロー防止機能を指定した場合はオーバフローは発生しません。このとき、オーバフロービット（V）はセットされません。CS ビットでオーバフローモードを指定した時の DC ビットもセットされません。

CPU タイプ命令の積和命令（MAC）は 64 ビットレジスタ（MACH、MACL）で演算しているので、オーバフローの値および最大値、最小値は DSP タイプ命令の場合とは異なります。

4.17 データ転送

SH-DSP は DSP ユニットで DSP レジスタと内蔵メモリとの間で最大 2 つのデータを同時に並行して転送することができます。SH-DSP には次の 3 つのデータ転送があります。

（1） X、Yメモリデータ転送： Xバス、Yバスを使ってX、Yメモリとデータ転送

（a） ダブルデータ転送： データ転送だけ、どちらか一方にのみの転送も可

（b） パラレルデータ転送： ALU演算や乗算と並行処理をしながらのデータ転送

（2） シングルデータ転送： メインバスを使って内蔵メモリとデータ転送

データ転送命令は DSR レジスタの状態ビットを更新しません。

それぞれの機能を**表** 4.33 に示します。

表 4.33 **データ転送の機能**

種類	使用バス	転送データ長	ALU 演算との並行処理	データ転送の並行処理	命令長
X、Y メモリデータ転送	X バス Y バス	16 ビット	なし（ダブル）	なし（X バスか Y バス）	16 ビット
				あり（X バスと Y バス）	16 ビット
			あり（パラレル）	なし（X バスか Y バス）	32 ビット
				あり（X バスと Y バス）	32 ビット
シングルデータ転送	メインバス	32 ビット 16 ビット	なし	なし	16 ビット

4.17.1 X、Y メモリデータ転送

X、Y メモリデータ転送は 2 つのデータ転送を同時に並行して実行することができ、データ転送と DSP データ演算を同時に並行して実行することができます。DSP データ演算と転送を同時に並行して実行させるには 32 ビットの命令コードが必要です。これをパラレルデータ転送とよびます。X、Y メモリデータ転送だけを実行する場合は 16 ビットの命令コードです。これをダブルデータ転送とよびます。

データ転送は、X メモリデータ転送と、Y メモリデータ転送があります。X メモリデータは X0、X1 レジスタのどちらかにロードされ、Y メモリデータは Y0、Y1 レジスタのどちらかにロードされます。X0、X1、Y0、Y1 レジスタがデスティネーションレジスタになります。デスティネーションレジスタの上位ワードにデータが転送され、下位ワードは自動的に 0 にクリアされます。A0、A1 レジスタの一方をソースレジスタとして、X、Y メモリにデータをストアすることができます。これらのデータ転送はすべてワードデータ（16 ビット）です。ソースレジスタの上位ワードからデータが転送されます。

同時に並行して実行する演算命令に条件付き命令を指定しても、データ転送命令は影響を受けません。

X、Y メモリデータ転送は X、Y メモリのみをアクセスし、他のメモリエリアはアクセスできません。

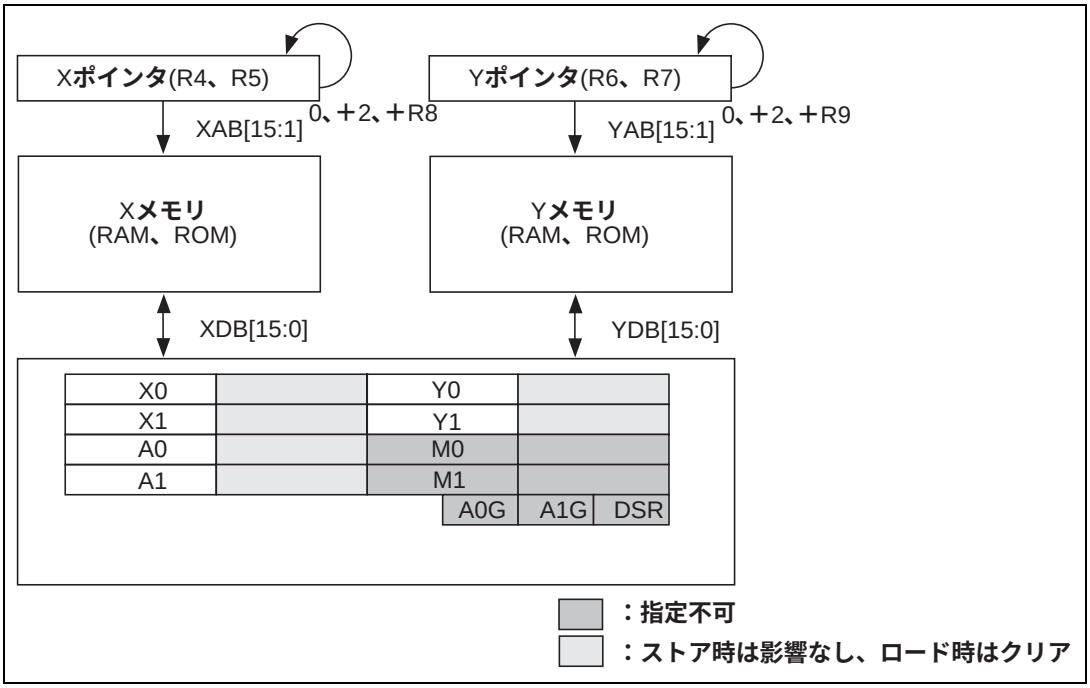


図 4.18　X、Y **メモリデータ転送の流れ**

4.17.2　シングルデータ転送

シングルデータ転送は 1 つのデータ転送だけを実行します。16 ビットの命令コードです。シングルデータ転送は ALU 演算と同時に並行処理はできません。X メモリをアクセスする X ポインタと追加された 2 つのポインタが有効となり、Y ポインタは無効です。CPU タイプ命令と同様に、シングルデータ転送は外部エリアを含むすべてのメモリエリアをアクセスできます。DSR レジスタを除く*DSP レジスタがソースオペランド、デスティネーションオペランドに指定できます。ガードビットレジスタ、A0G、A1G は独立したレジスタとしてオペランドに指定できます。シングルデータ転送では X バス、Y バスの代わりにメインバスを使うので、メインバス上でデータ転送と命令フェッチの競合が発生します。

シングルデータ転送はワードデータとロングワードデータを取り扱います。ワードデータ転送ではレジスタの上位ワードが有効です。レジスタにデータがロードされるときは上位ワードにロードされ、下位ワードは自動的に 0 でクリアされ、ガードビットがあればガードビットには符号ビットが拡張されて格納されます。レジスタからストアされるときは上位ワードがストアされます。

ロングワードが転送されるときは 32 ビットが有効になります。ロードされるときガードビットがあれば、ガードビットには符号ビットが拡張されて格納されます。

ガードビットレジスタがストアされるときは、上位 24 ビットに符号が拡張されて、メインバスに読み出されます。ガードビットレジスタ、A0G、A1G レジスタが MOVS.W 命令のデスティネーションレジスタとしてワードデータがロードされるときは、下位バイトがレジスタに書き込まれます。

【注】 *　DSR **レジスタはシステムレジスタとして定義されているので、**LDS**、**STS **命令でデータの転送が可能です。**

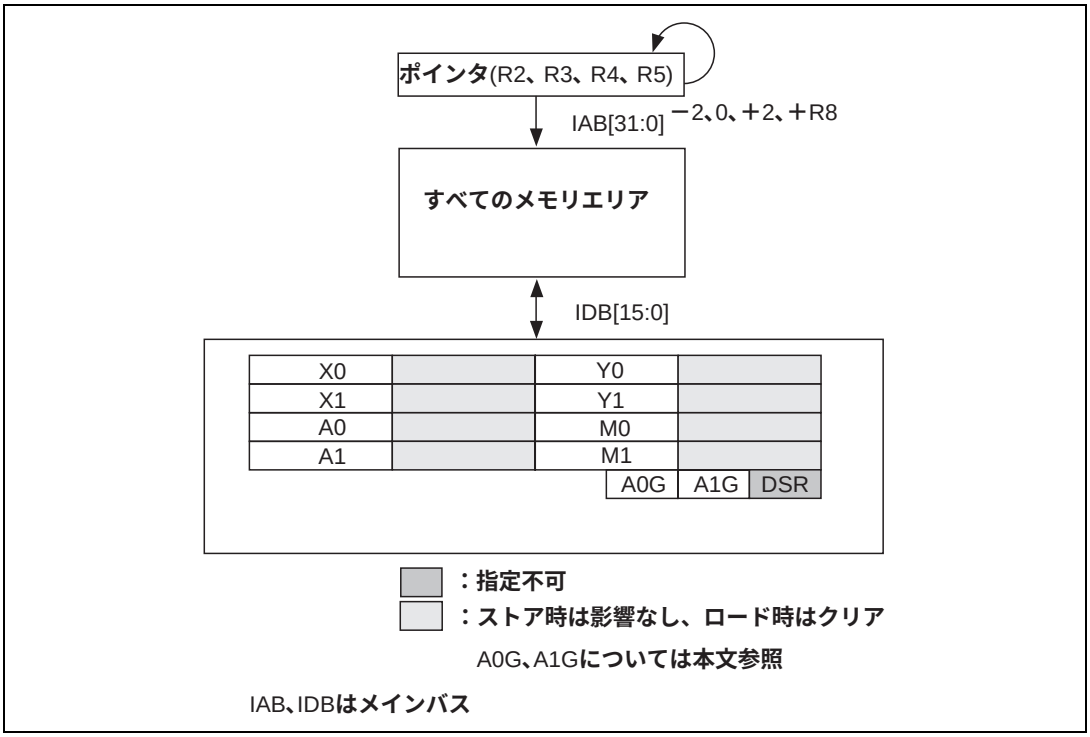


図 4.19　**シングルデータ転送の流れ（ワード）**

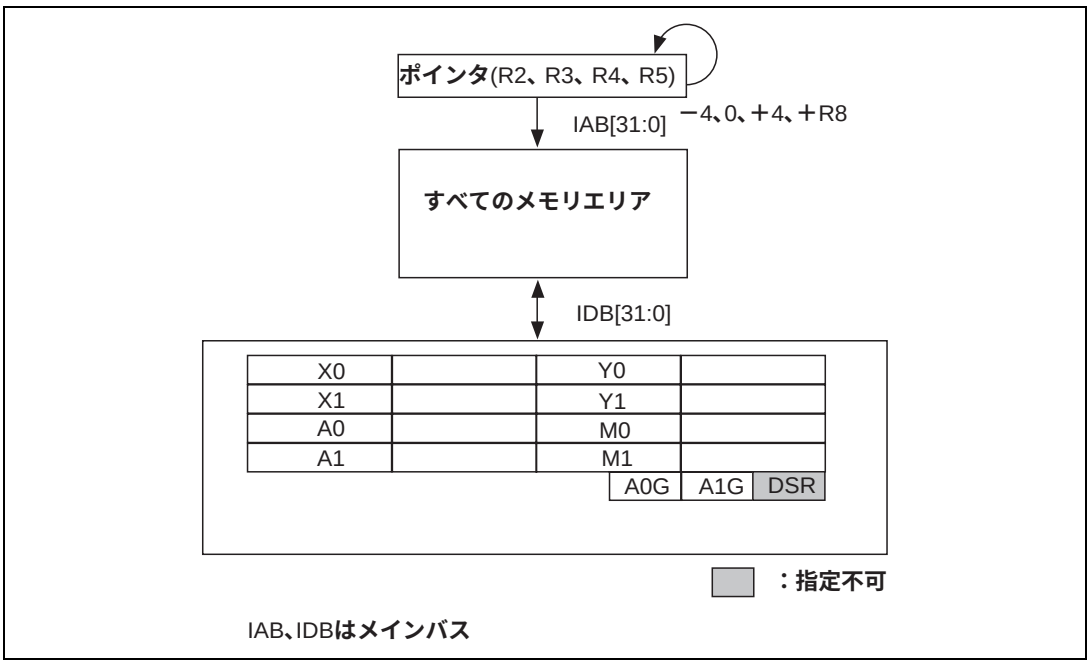


図 4.20　**シングルデータ転送の流れ（ロングワード）**

データ転送はパイプラインの MA ステージで実行され、DSP 演算は DSP ステージで実行されます。演算機能をストアする命令がデータ演算命令のすぐ次の命令行にある場合は、データ演算命令が終わらないうちに次のデータストア命令が始まるため、1 つのストールサイクルが挿入されます。このオーバヘッドサイクルは、1 つの命令をデータ演算命令とデータ転送命令の間に追加することによって避けることができます。この例を**図** 4.21 に示します。

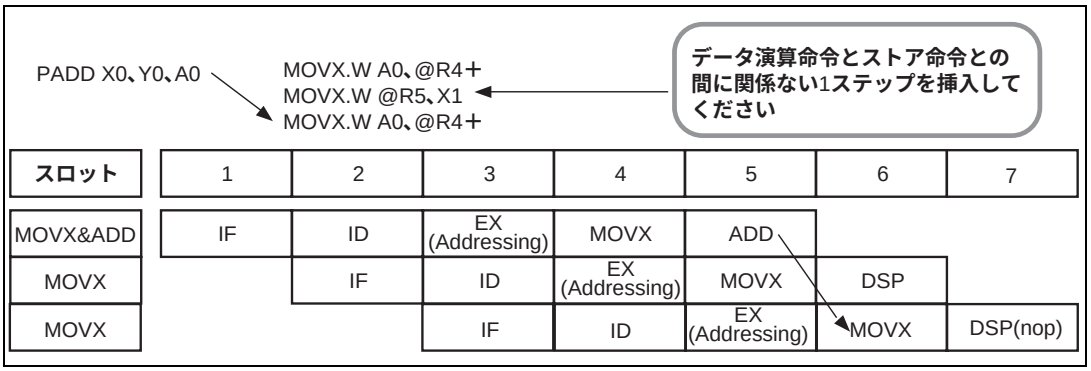


図 4.21　**演算とデータストアの命令実行例**

4.18 オペランド競合

2 つ以上の並行処理命令でデスティネーションオペランドに同じレジスタを指定するとデータの競合が発生します。データの競合は次の 3 つの場合が考えられます。

（1） ALU演算と乗算で同じデスティネーションオペランドを指定した場合（Du、Dg）
（2） XメモリロードとALU演算で同じデスティネーションオペランドを指定した場合（Dx、Du、Dz）
（3） YメモリロードとALU演算で同じデスティネーションオペランドを指定した場合（Dy、Du、Dz）

もし競合した場合の結果は保証されません。競合の発生するオペランドとレジスタの対応を**表** 4.34 に示します。

これらの競合を検出できるアセンブラがありますので、機能を選択してアセンブラをお使いください。

表 4.34 **競合の発生するオペランドとレジスタとの対応**

		DSP**レジスタ**							
		X0	X1	Y0	Y1	M0	M1	A0	A1
X**メモリロード**	Ax								
	Ix								
	Dx	(*)	(*)						
Y**メモリロード**	Ay								
	Iy								
	Dy			(*)	(*)				
6**オペランド** ALU**演算**	Sx	*	*					*	*
	Sy			*	*	*	*		
	Du	(*)		(*)				(*)	(*)
3**オペランド乗算**	Se	*	*	*					*
	Sf	*		*	*				*
	Dg					*	*	(*)	(*)
3**オペランド** ALU**演算**	Sx	*	*					*	*
	Sy			*	*	*	*		
	Dz	(*)	(*)	(*)	(*)	*	*	*	*

↑ (Dx、Du、Dz **の競合**) ↑ (Dy、Du、Dz **の競合**) ↑ (Du、Dg **の競合**) ↑

【注】 * **オペランドに対する設定可能レジスタ**

(*) **オペランド競合**

4.19 DSP 繰り返し（ループ）制御

SH-DSP は、効率よく繰り返し（ループ）制御を行うための特別な機構です。SETRC 命令で、繰り返しカウンタ（RC、12 ビット）に繰り返し回数を格納し、RC が 1 になるまで繰り返しプログラム（ループ）を反復する実行モードを設定します。繰り返し動作が終了すると、RC の内容は 0 になります。

繰り返し開始アドレスレジスタ（RS）は、繰り返しループの開始アドレスを格納しています。繰り返し終了レジスタ（RE）は、繰り返し終了アドレスを格納しています。（例外があります。「4.19.1 **注意事項**」を参照してください。）繰り返しカウンタ（RC）は、繰り返し回数を格納しています。この繰り返し制御を実行する手順は次のようになります。

＃1 繰り返し開始アドレスを RS レジスタに設定します。
＃2 繰り返し終了アドレスを RE レジスタに設定します。
＃3 繰り返し回数を RC カウンタに設定します。
＃4 繰り返しプログラム（ループ）を開始します。
＃1 と＃2 を実行するために次の命令を使います。

```
LDRS @（disp,PC）;および
LDRE @（disp,PC）;
```

＃3 と＃4 を実行するために SETEC 命令を使います。SETEC 命令のオペランドはイミィディエイト値または汎用レジスタで繰り返し回数を指定します。

```
SETRC #imm; #imm→RC,enable repeat control
SETRC Rm;   Rm→RC,enable repeat control
```

#imm は 8 ビットで RC カウンタは 12 ビットです。そのため RC カウンタに 256 以上の数値を指定したいときは、Rm レジスタを使って設定します。プログラム例を次に示します。

```
            LDRS RptStart;
            LDRE RptEnd;
            SETRC #imm;       RC=#imm
            instr0;
; instr1～n executes repeatedly
RptStart:   instr1;
            instr2;
              :
              :
            instr n-2;
            instr n-1;
RptEnd:     instr n;
            instr n+1;
```

この繰り返し命令には次のようないくつかの制限があります。

（1） SETRC命令と繰り返しプログラム（ループ）の最初の命令の間には少なくとも1命令が必要です。
（2） LDRS、LDRE命令を実行したあとで、SETRC命令を実行してください。

(3) 繰り返しプログラム（ループ）が4命令以上の場合は、繰り返し開始アドレス（先述の例ではinstr1のアドレス）がロングワード境界にない時には、繰り返しのたびに1サイクルのストール（実行待ちサイクル）が発生します。

(4) 繰り返しプログラム（ループ）が3命令以下の場合は、分岐命令（BRA、BSR、BT、BF、BT/S、BF/S、BSRF、RTS、BRAF、RTE、JSR、JMP）、繰り返し制御命令（SETRC、LDRS、LDRE）、SR、RS、およびREのロード命令、TRAPAは使えません。もし記述すると、エラー例外処理が起動され、**表**4.35に示すアドレス値がR15がポイントするスタックエリアに押し出されます。

表 4.35 **押し出される** PC **値（**1**）**

条件	位置	押し出されるアドレス
RC>=2	**任意**	RptStart
RC=1	**任意**	**不正な命令のプログラムアドレス**

(5) 繰り返しプログラム（ループ）が4命令以上の場合、分岐命令（BRA、BSR、BT、BF、BT/S、BF/S、BSRF、RTS、BRAF、RTE、JSR、JMP）、繰り返し制御命令（SETRC、LDRS、LDRE）、SR、RS、およびREのロード命令、TRAPAは繰り返しプログラム（ループ）内の最後の3命令には使えません。もし記述すると、エラー例外処理が起動され、**表**4.36に示すアドレス値がR15がポイントするスタックエリアに押し出されます。繰り返し制御命令（SETRC、LDRS、LDRE）、SR、RS、およびREのロード命令の場合には、繰り返しモジュールの他の位置には記述できません。記述すると正しく動作しません。

表 4.36 **押し出される** PC **値（**2**）**

条件	位置	押し出されるアドレス
RC>=2	instr n-2	**不正な命令のプログラムアドレス**
	instr n-1	RptStart-4
	instr n	RptStart-2
RC=1	**任意**	**不正な命令のプログラムアドレス**

(6) 繰り返しプログラム（ループ）が3命令以下の場合には、PC相対命令（MOVA （disp、PC）、R0など）は繰り返しプログラム（ループ）の最初の命令（先述の例ではinstr1）だけに使えます。

(7) 繰り返しプログラム（ループ）が4命令以上の場合には、PC相対命令（MOVA （disp、PC）、R0など）は繰り返しプログラム（ループ）の最後の2命令には使えません。

(8) SH-DSPに繰り返し有効フラグはありませんが、RCカウンタが0のとき繰り返しは無効になります。RCカウンタが0でなく、PCカウンタがREレジスタの内容と一致したとき、繰り返しが開始されます。RCカウンタを0に設定すると、繰り返しプログラム（ループ）は無効ですが繰り返しモジュールを1回だけ実行し、RCが1の場合と同様に繰り返しプログラム（ループ）の開始命令には戻りません。RCカウンタを1に設定すると繰り返しモジュールを1回だけ実行し、繰り返しプログラム（ループ）の開始命令には戻りませんが、RCカウンタはゼロになります。

(9) 繰り返しプログラム（ループ）が4命令以上の場合には、分岐命令の分岐先アドレスとして、繰り返し終了アドレスから2つ前までのアドレス（先述の例ではinstr n-2のアドレス）までは指定できません。
もしこれを実行すると繰り返し制御は正しく動作しません。

(10) 繰り返し実行中は、割込みは制限されます。詳細は、**図**4.22を参照してください。この図のそれぞれのケースのフローがEXの各ステージを示しています。割込みまたはバスエラー

例外の最初のEXステージは、通常、命令のEXステージが終了した直後に開始します。これらを図では"A"で示しています。ただし、次のinstr0のEXステージでは、バスエラー例外だけを"B"で指定して続けられます。instr1のEXステージでは、"C"によって割り込みもバス例外も続けることができません。instr2のEXステージだけを続けることができます。

A: **割り込みおよびバスエラー例外をすべて受け付ける**
B: **バスエラー例外のみを受け付ける**
C: **割り込みおよびバスエラー例外は一切受け付けない**

RC>=1の場合

(1) **1ステップ繰り返し**

```
              instr0 <- A
Start(End):   instr1 <- B
              instr2 <- C
                     <- A
```

(2) **2ステップ繰り返し**

```
              instr0 <- A
              instr1 <- B
Start:        instr2 <- C
End:          instr3 <- C
                     <- A
```

(3) **3ステップ繰り返し**

```
                instr0 <- A
                instr1 <- B
Start:          instr2 <- C
                instr3 <- C
End:            instr4 <- C
                       <- A
```

(4) **4以上のステップ繰り返し**

```
                <- A
       instr0   <- A or C(when returning from instr n)
Start: instr1   <- A
          :        :
          :     <- A
       instr n-3<- B
       instr n-2<- C
       instr n-1<- C
End:   instr n  <- C
       instr n+1<- A
```

RC=0の場合：
割り込みおよびバスエラー例外をすべて受け付ける

図 4.22 繰り返しモジュールでの割り込み受け付けの制限

4.19.1 注意事項

(1) プログラミングの実際

繰り返し開始レジスタ（RS）と繰り返し終了レジスタ（RE）は、繰り返し開始アドレスと繰り返し終了アドレスをそれぞれ格納しています。これらのレジスタに格納されているアドレスは繰り返しプログラム（ループ）内の命令の数によって変わります。この規則を次に示します。

Repeat_Start: 繰り返し開始命令のアドレス
Repeat_Start0: 繰り返し終了命令の 1 つ上の命令のアドレス
Repeat_Start3: 繰り返し終了命令の 3 つ上の命令のアドレス

表 4.36 RS **および** RE **設定規則**

	繰り返しプログラム（ループ）内の命令の数			
	1	2	3	>=4
RS	Repeat_start0 +8	Repeat_start0 +6	Repeat_start0 +4	Repeat_Start
RE	Repeat_start0 +4	Repeat_start0 +4	Repeat_start0 +4	Repeat_End3 + 4

このテーブルに基づいて、さまざまなケースを想定した実際の繰り返しプログラム（ループ）のプログラミング例を次に示します。

ケース 1: 1 繰り返し命令の場合

```
            LDRS   RptStart0+8;              (RptStart)
            LDRE   RptStart0+4;              (RptStart)
            SETRC  RptCount;
               - - - -
RptStart0:  instr0;
RptStart:   instr1;                          繰り返し命令
            instr2;
```

ケース 2: 2 繰り返し命令の場合

```
            LDRS   RptStart0+6;              (RptStart)
            LDRE   RptStart0+4;              (RptEnd)
            SETRC  RptCount;
               - - - -
RptStart0:  instr0;
RptStart:   instr1;                          繰り返し命令 1
RptEnd:     instr2;                          繰り返し命令 2
            instr3;
```

ケース 3: 3 繰り返し命令の場合

```
            LDRS   RptStart0+4;              (RptStart)
            LDRE   RptStart0+4;              (RptEnd)
```

```
            SETRC                               RptCount;
               - - - -
RptStart0:  instr0;
RptStart:   instr1;                             繰り返し命令 1
            instr2;                             繰り返し命令 2
RptEnd:     instr3;                             繰り返し命令 3
            instr4;
```

ケース 4: 4 繰り返し命令以上の場合

```
            LDRS   RptStart;
            LDRE   RptEnd3+4;                   (RptEnd)
            SETRC  RptCount;
               - - - -
RptStart0:  instr0;
RptStart:   instr1;                             繰り返し命令 1
            instr2;                             繰り返し命令 2
            instr3;                             繰り返し命令 3
-----------------------------------------------
RptEnd3:    instr N-3;                          繰り返し命令 N-3
            instr N-2;                          繰り返し命令 N-2
            instr N-1;                          繰り返し命令 N-1
RptEnd:     instr N;                            繰り返し命令 N
            instr N+1;
```

上記の例はこの繰り返しプログラム（ループ）シーケンスをプログラミングするためのテンプレートとして用いることができます。拡張命令"REPEAT"で、これらの複雑なラベリングとオフセットの問題を簡素化できます。詳細を注 2 に記述します。

(2) 拡張命令 REPEAT

拡張命令 REPEAT で、**表** 4.36 および注 1 に記述するラベリングとオフセットの微妙な取り扱いを簡単にできます。次に使用するラベルを示します。

RptStart: 繰り返しプログラム（ループ）の先頭命令のアドレス
RptEnd: 繰り返しプログラム（ループ）の最終命令のアドレス
PptCount: 繰り返し回数イミィディエイト番号

この命令は次のように使用します。
Repeat count は、イミディエイト値#imm またはレジスタ間接値 Rn として指定できます。

ケース 1: 1 繰り返し命令の場合

```
REPEAT RptStart, RptStart, RptCount
```

```
             - - - -
            instr0;
RptStart:   instr1;           繰り返し命令 1
            instr2;
```

ケース 2: 2 繰り返し命令の場合

```
REPEAT RptStart, RptEnd, RptCount
             - - - -
            instr0;
RptStart:   instr1;           繰り返し命令 1
RptEnd:     instr2;           繰り返し命令 2
```

ケース 3: 3 繰り返し命令の場合

```
REPEAT RptStart, RptEnd, RptCount
             - - - -
            instr0;
RptStart:   instr1;           繰り返し命令 1
            instr2;           繰り返し命令 2
RptEnd:     instr3;           繰り返し命令 3
```

ケース 4: 4 繰り返し命令以上の場合

```
REPEAT RptStart, RptEnd, RptCount
               - - - -
            instr0;
RptStart:   instr1;           繰り返し命令 1
            instr2;           繰り返し命令 2
            instr3;           繰り返し命令 3
-----------------------------------------------
            instr N-3;        繰り返し命令 N-3
            instr N-2;        繰り返し命令 N-2
            instr N-1;        繰り返し命令 N-1
RptEnd:     instr N;          繰り返し命令 N
            instr N+1;
```

それぞれのケースでの拡張の結果は、注 1 のケース番号に対応します。

4.20 条件付き命令とデータ転送

データ演算命令には無条件命令と、条件付き命令があります。両者とも並行して実行するデータ転送命令を指定することができますが、条件が不成立の場合でもデータ転送命令には影響せず、常に実行されます。

条件付き命令とデータ転送の例を**図** 4.23 に示します。

```
DCT PADD X0、Y0、A0  MOVX.W @R4+、X0  MOVY.W A0、@R6+R9 ;

［条件が真のとき］
実行前：X0=H'33333333、Y0=H'55555555、A0=H'123456789A、
        R4=H'00008000、R6=H'00008232、R1=H'00000004
        (R4)=H'1111、(R6)=H'2222
実行後：X0=H'11110000、Y0=H'55555555、A0=H'0088888888、
        R4=H'00008002、R6=H'00008236、R1=H'00000004
        (R4)=H'1111、(R6)=H'1234

［条件が偽のとき］
実行前：X0=H'33333333、Y0=H'55555555、A0=H'123456789A、
        R4=H'00008000、R6=H'00008232、R1=H'00000004
        (R4)=H'1111、(R6)=H'2222
実行後：X0=H'11110000、Y0=H'55555555、A0=H'123456789A、
        R4=H'00008002、R6=H'00008236、R1=H'00000004
        (R4)=H'1111、(R6)=H'1234
```

図 4.23　**条件付き命令とデータ転送の例**

5. 命令セット

SH-DSP の命令は３つに分けることができます。主に CPU コアで実行される CPU 命令、主に DSP ユニットで実行される DSP データ転送命令、DSP 演算命令があります。CPU 命令には DSP の機能をサポートするための命令がいくつかあります。命令セットの説明をそれぞれ３つに分けて説明します。

5.1 CPU 命令の命令セット

CPU 命令を分類別に**表** 5.1 に示します。

表 5.1 CPU **命令の分類**

分類	命令の種類	オペコード	機能	適用命令 SH-1	適用命令 SH-2	適用命令 SH-DSP	命令数
データ転送命令	5	MOV	**データ転送** **イミディエイトデータの転送** **周辺モジュールデータの転送** **構造体データの転送**	○	○	○	39
		MOVA	**実効アドレスの転送**	○	○	○	
		MOVT	T **ビットの転送**	○	○	○	
		SWAP	**上位と下位の交換**	○	○	○	
		XTRCT	**連結レジスタの中央切り出し**	○	○	○	
算術演算命令	21	ADD	2 **進加算**	○	○	○	33
		ADDC	**キャリ付き** 2 **進加算**	○	○	○	
		ADDV	**オーバフロー付き** 2 **進加算**	○	○	○	
		CMP/cond	**比較**	○	○	○	
		DIV1	**除算**	○	○	○	
		DIV0S	**符号付き除算の初期化**	○	○	○	
		DIV0U	**符号なし除算の初期化**	○	○	○	
		DMULS	**符号付き倍精度乗算**	−	○	○	
		DMULU	**符号なし倍精度乗算**	−	○	○	
		DT	**デクリメントとテスト**	−	○	○	
		EXTS	**符号拡張**	○	○	○	
		EXTU	**ゼロ拡張**	○	○	○	
		MAC	**積和演算**	○	○	○	
			倍精度積和演算	−	○	○	
		MUL	**倍精度乗算**	−	○	○	
		MULS	**符号付き乗算**	○	○	○	
		MULU	**符号なし乗算**	○	○	○	

分類	命令の種類	オペコード	機能	適用命令 SH-1	SH-2	SH-DSP	命令数
算術演算命令	21	NEG	符号反転	○	○	○	33
		NEGC	ボロー付き符号反転	○	○	○	
		SUB	2 進減算	○	○	○	
		SUBC	ボロー付き 2 進減算	○	○	○	
		SUBV	アンダフロー付き 2 進減算	○	○	○	
論理演算命令	6	AND	論理積演算	○	○	○	14
		NOT	ビット反転	○	○	○	
		OR	論理和演算	○	○	○	
		TAS	メモリテストとビットセット	○	○	○	
		TST	論理積演算の T ビットセット	○	○	○	
		XOR	排他的論理和演算	○	○	○	
シフト命令	10	ROTCL	T ビット付き 1 ビット左回転	○	○	○	
		ROTCR	T ビット付き 1 ビット右回転	○	○	○	
		ROTL	1 ビット左回転	○	○	○	
		ROTR	1 ビット右回転	○	○	○	
		SHAL	算術的 1 ビット左シフト	○	○	○	
		SHAR	算術的 1 ビット右シフト	○	○	○	
		SHLL	論理的 1 ビット左シフト	○	○	○	
		SHLLn	論理的 n ビット左シフト	○	○	○	
		SHLR	論理的 1 ビット右シフト	○	○	○	
		SHLRn	論理的 n ビット右シフト	○	○	○	
分岐命令	9	BF	条件分岐(T＝0 で分岐)	○	○	○	11
			条件付き遅延分岐(T＝0 で分岐)	−	○	○	
		BT	条件分岐(T＝1 で分岐)	○	○	○	
			条件付き遅延分岐(T＝1 で分岐)	−	○	○	
		BRA	無条件分岐	○	○	○	
		BRAF	無条件分岐	−	○	○	
		BSR	サブルーチンプロシージャへの分岐	○	○	○	
		BSRF	サブルーチンプロシージャへの分岐	−	○	○	
		JMP	無条件分岐	○	○	○	
		JSR	サブルーチンプロシージャへの分岐	○	○	○	
		RTS	サブルーチンプロシージャからの復帰	○	○	○	
システム制御命令	14	CLRMAC	MAC レジスタのクリア	○	○	○	71
		CLRT	T ビットのクリア	○	○	○	
		LDC	コントロールレジスタへのロード	○	○	○	
		LDRE	繰り返し終了レジスタへのロード	−	−	○	
		LDRS	繰り返し開始レジスタへのロード	−	−	○	
		LDS	システムレジスタへのロード	○	○	○	
		NOP	無操作	○	○	○	

分類	命令の種類	オペコード	機能	適用命令 SH-1	SH-2	SH-DSP	命令数
システム制御命令	14	RTE	例外処理からの復帰	○	○	○	71
		SETRC	繰り返し回数および繰り返し制御フラグの設定	−	−	○	
		SETT	T ビットのセット	○	○	○	
		SLEEP	低消費電力状態への遷移	○	○	○	
		STC	コントロールレジスタからのストア	○	○	○	
		STS	システムレジスタからのストア	○	○	○	
		TRAPA	トラップ例外処理	○	○	○	
	計 65						計 182

CPU 命令の命令コード、動作、実行ステートを、以下の形式で分類別に説明します。

命令	動作	命令コード	実行ステート	T ビット
ニーモニックで表示しています。 記号の説明 OP.Sz SRC、DEST OP： オペコード Sz： サイズ SRC： ソース DEST：デスティネーション Rm： ソースレジスタ Rn： デスティネーションレジスタ imm： イミディエイトデータ disp： ディスプレースメント*2	動作の概略を表示しています。 記号の説明 →、←： 転送方向 (xx)： メモリオペランド M/Q/T：SR 内のフラグビット &： ビットごとの論理積 |： ビットごとの論理和 ^： ビットごとの排他的論理和 ~： ビットごとの論理否定 <<n： 左 n ビットシフト >>n： 右 n ビットシフト	MSB←→LSB の順で表示しています。 記号の説明 mmmm： ソースレジスタ nnnn： デスティネーションレジスタ 0000： R0 0001： R1 …… 1111： R15 iiii： イミディエイトデータ dddd： ディスプレースメント	ノーウェイトのときの値です。*1	命令実行後の、T ビットの値を表示しています。 記号の説明 −： 変化しない

【注】 *1 命令の実行ステートについて
表に示した実行ステートは最少値です。実際は、
(1) 命令フェッチとデータアクセスの競合が起こる場合
(2) ロード命令（メモリ→レジスタ）のデスティネーションレジスタと、その直後の命令が使うレジスタが同一な場合
などの条件により、命令実行ステート数は増加します。

*2 命令のオペランドサイズなどに応じてスケーリング (×1、×2、×4) されます。
詳細は「6. 各命令の説明」を参照して下さい。

5.1.1 データ転送命令

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
MOV	#imm,Rn	imm→符号拡張→Rn	1110nnnniiiiiiii	1	—	○	○	○
MOV.W	@(disp,PC),Rn	(disp×2＋PC)→符号拡張→Rn	1001nnnndddddddd	1	—	○	○	○
MOV.L	@(disp,PC),Rn	(disp×4＋PC)→Rn	1101nnnndddddddd	1	—	○	○	○
MOV	Rm,Rn	Rm→Rn	0110nnnnmmmm0011	1	—	○	○	○
MOV.B	Rm,@Rn	Rm→(Rn)	0010nnnnmmmm0000	1	—	○	○	○
MOV.W	Rm,@Rn	Rm→(Rn)	0010nnnnmmmm0001	1	—	○	○	○
MOV.L	Rm,@Rn	Rm→(Rn)	0010nnnnmmmm0010	1	—	○	○	○
MOV.B	@Rm,Rn	(Rm)→符号拡張→Rn	0110nnnnmmmm0000	1	—	○	○	○
MOV.W	@Rm,Rn	(Rm)→符号拡張→Rn	0110nnnnmmmm0001	1	—	○	○	○
MOV.L	@Rm,Rn	(Rm)→Rn	0110nnnnmmmm0010	1	—	○	○	○
MOV.B	Rm,@-Rn	Rn－1→Rn, Rm→(Rn)	0010nnnnmmmm0100	1	—	○	○	○
MOV.W	Rm,@-Rn	Rn－2→Rn, Rm→(Rn)	0010nnnnmmmm0101	1	—	○	○	○
MOV.L	Rm,@-Rn	Rn－4→Rn, Rm→(Rn)	0010nnnnmmmm0110	1	—	○	○	○
MOV.B	@Rm+,Rn	(Rm)→符号拡張→Rn, Rm＋1→Rm	0110nnnnmmmm0100	1	—	○	○	○
MOV.W	@Rm+,Rn	(Rm)→符号拡張→Rn, Rm＋2→Rm	0110nnnnmmmm0101	1	—	○	○	○
MOV.L	@Rm+,Rn	(Rm)→Rn, Rm＋4→Rm	0110nnnnmmmm0110	1	—	○	○	○
MOV.B	R0,@(disp,Rn)	R0→(disp＋Rn)	10000000nnnndddd	1	—	○	○	○
MOV.W	R0,@(disp,Rn)	R0→(disp×2＋Rn)	10000001nnnndddd	1	—	○	○	○
MOV.L	Rm,@(disp,Rn)	Rm→(disp×4＋Rn)	0001nnnnmmmmdddd	1	—	○	○	○
MOV.B	@(disp,Rm),R0	(disp＋Rm)→符号拡張→R0	10000100mmmmdddd	1	—	○	○	○
MOV.W	@(disp,Rm),R0	(disp×2＋Rm)→符号拡張→R0	10000101mmmmdddd	1	—	○	○	○
MOV.L	@(disp,Rm),Rn	(disp×4＋Rm)→Rn	0101nnnnmmmmdddd	1	—	○	○	○
MOV.B	Rm,@(R0,Rn)	Rm→(R0＋Rn)	0000nnnnmmmm0100	1	—	○	○	○
MOV.W	Rm,@(R0,Rn)	Rm→(R0＋Rn)	0000nnnnmmmm0101	1	—	○	○	○
MOV.L	Rm,@(R0,Rn)	Rm→(R0＋Rn)	0000nnnnmmmm0110	1	—	○	○	○
MOV.B	@(R0,Rm),Rn	(R0＋Rm)→符号拡張→Rn	0000nnnnmmmm1100	1	—	○	○	○
MOV.W	@(R0,Rm),Rn	(R0＋Rm)→符号拡張→Rn	0000nnnnmmmm1101	1	—	○	○	○
MOV.L	@(R0,Rm),Rn	(R0＋Rm)→Rn	0000nnnnmmmm1110	1	—	○	○	○
MOV.B	R0,@(disp,GBR)	R0→(disp＋GBR)	11000000dddddddd	1	—	○	○	○
MOV.W	R0,@(disp,GBR)	R0→(disp×2＋GBR)	11000001dddddddd	1	—	○	○	○
MOV.L	R0,@(disp,GBR)	R0→(disp×4＋GBR)	11000010dddddddd	1	—	○	○	○
MOV.B	@(disp,GBR),R0	(disp＋GBR)→符号拡張→R0	11000100dddddddd	1	—	○	○	○
MOV.W	@(disp,GBR),R0	(disp×2＋GBR)→符号拡張→R0	11000101dddddddd	1	—	○	○	○
MOV.L	@(disp,GBR),R0	(disp×4＋GBR)→R0	11000110dddddddd	1	—	○	○	○
MOVA	@(disp,PC),R0	disp×4＋PC→R0	11000111dddddddd	1	—	○	○	○
MOVT	Rn	T→Rn	0000nnnn00101001	1	—	○	○	○
SWAP.B	Rm,Rn	Rm→下位 2 バイトの上下バイト交換→Rn	0110nnnnmmmm1000	1	—	○	○	○
SWAP.W	Rm,Rn	Rm→上下ワード交換→Rn	0110nnnnmmmm1001	1	—	○	○	○
XTRCT	Rm,Rn	Rm と Rn の中央 32 ビット→Rn	0010nnnnmmmm1101	1	—	○	○	○

5.1.2 算術演算命令

命令	動作	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
ADD Rm,Rn	Rn＋Rm→Rn	0011nnnnmmmm1100	1	―	○	○	○
ADD #imm,Rn	Rn＋imm→Rn	0111nnnniiiiiiii	1	―	○	○	○
ADDC Rm,Rn	Rn＋Rm＋T→Rn, キャリ→T	0011nnnnmmmm1110	1	キャリ	○	○	○
ADDV Rm,Rn	Rn＋Rm→Rn, オーバフロー→T	0011nnnnmmmm1111	1	オーバフロー	○	○	○
CMP/EQ #imm,R0	R0＝imm のとき 1→T, ≠のとき 0→T	10001000iiiiiiii	1	比較結果	○	○	○
CMP/EQ Rm,Rn	Rn＝Rm のとき 1→T, ≠のとき 0→T	0011nnnnmmmm0000	1	比較結果	○	○	○
CMP/HS Rm,Rn	無符号で Rn≧Rm のとき 1→T, <のとき 0→T	0011nnnnmmmm0010	1	比較結果	○	○	○
CMP/GE Rm,Rn	有符号で Rn≧Rm のとき 1→T, <のとき 0→T	0011nnnnmmmm0011	1	比較結果	○	○	○
CMP/HI Rm,Rn	無符号で Rn＞Rm のとき 1→T, ≦のとき 0→T	0011nnnnmmmm0110	1	比較結果	○	○	○
CMP/GT Rm,Rn	有符号で Rn＞Rm のとき 1→T, ≦のとき 0→T	0011nnnnmmmm0111	1	比較結果	○	○	○
CMP/PL Rn	Rn＞0 のとき 1→T, ≦0 のとき 0→T	0100nnnn00010101	1	比較結果	○	○	○
CMP/PZ Rn	Rn≧0 のとき 1→T, ＜0 のとき 0→T	0100nnnn00010001	1	比較結果	○	○	○
CMP/STR Rm,Rn	いずれかのバイトが等しいとき 1→T, そうでないとき 0→T	0010nnnnmmmm1100	1	比較結果	○	○	○
DIV1 Rm,Rn	1 ステップ除算 (Rn÷Rm)	0011nnnnmmmm0100	1	計算結果	○	○	○
DIV0S Rm,Rn	Rn の MSB→Q, Rm の MSB→M, M^Q→T	0010nnnnmmmm0111	1	計算結果	○	○	○
DIV0U	0→M/Q/T	0000000000011001	1	0	○	○	○
DMULS.L Rm,Rn	符号付きで Rn×Rm→MACH,MACL 32×32→64 ビット	0011nnnnmmmm1101	2～4*1	―	―	○	○
DMULU.L Rm,Rn	符号なしで Rn×Rm→MACH,MACL 32×32→64 ビット	0011nnnnmmmm0101	2～4*1	―	―	○	○
DT Rn	Rn－1→Rn, Rn が 0 のとき 1→T Rn が 0 以外のとき 0→T	0100nnnn00010000	1	比較結果	―	○	○
EXTS.B Rm,Rn	Rm をバイトから符号拡張→Rn	0110nnnnmmmm1110	1	―	○	○	○
EXTS.W Rm,Rn	Rm をワードから符号拡張→Rn	0110nnnnmmmm1111	1	―	○	○	○
EXTU.B Rm,Rn	Rm をバイトからゼロ拡張→Rn	0110nnnnmmmm1100	1	―	○	○	○
EXTU.W Rm,Rn	Rm をワードからゼロ拡張→Rn	0110nnnnmmmm1101	1	―	○	○	○
MAC.L @Rm+,@Rn+	符号付きで (Rn)×(Rm)＋MAC→MAC 32×32＋64→64 ビット	0000nnnnmmmm1111	3/(2～4)*1	―	―	○	○
MAC.W @Rm+,@Rn+	符号付きで (Rn)×(Rm)＋MAC→MAC (SH-2) 16×16＋64→64 ビット (SH-1) 16×16＋42→42 ビット	0100nnnnmmmm1111	3/(2)*1	―	○	○	○
MUL.L Rm,Rn	Rn×Rm→MACL 32×32→32 ビット	0000nnnnmmmm0111	2～4*1	―	―	○	○
MULS.W Rm,Rn	符号付きで Rn×Rm→MAC 16×16→32 ビット	0010nnnnmmmm1111	1～3*1	―	○	○	○

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
MULU.W	Rm,Rn	符号なしで Rn×Rm→MAC 16×16→32 ビット	0010nnnnmmmm1110	1〜3*1	―	○	○	○
NEG	Rm,Rn	0－Rm→Rn	0110nnnnmmmm1011	1	―	○	○	○
NEGC	Rm,Rn	0－Rm－T→Rn, ボロー→T	0110nnnnmmmm1010	1	ボロー	○	○	○
SUB	Rm,Rn	Rn－Rm→Rn	0011nnnnmmmm1000	1	―	○	○	○
SUBC	Rm,Rn	Rn－Rm－T→Rn, ボロー→T	0011nnnnmmmm1010	1	ボロー	○	○	○
SUBV	Rm,Rn	Rn－Rm→Rn, アンダフロー→T	0011nnnnmmmm1011	1	アンダフロー	○	○	○

【注】 *1 通常実行ステートを示します。（ ）内の値は前後の命令との競合関係による実行ステートです。

5.1.3 論理演算命令

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	―	○	○	○
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	1	―	○	○	○
AND.B	#imm,@(R0,GBR)	(R0＋GBR) & imm → (R0＋GBR)	11001101iiiiiiii	3	―	○	○	○
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	1	―	○	○	○
OR	Rm,Rn	Rn｜Rm → Rn	0010nnnnmmmm1011	1	―	○	○	○
OR	#imm,R0	R0｜imm → R0	11001011iiiiiiii	1	―	○	○	○
OR.B	#imm,@(R0,GBR)	(R0＋GBR)｜imm → (R0＋GBR)	11001111iiiiiiii	3	―	○	○	○
TAS.B	@Rn	(Rn)が 0 のとき 1→T, 0 でないとき 0→T. また, (Rn)の値にかかわらず, 1→MSBof(Rn)	0100nnnn00011011	4	テスト結果	○	○	○
TST	Rm,Rn	Rn & Rm, 結果が 0 のとき 1→T, 0 でないとき 0→T	0010nnnnmmmm1000	1	テスト結果	○	○	○
TST	#imm,R0	R0 & imm, 結果が 0 のとき 1→T, 0 でないとき 0→T	11001000iiiiiiii	1	テスト結果	○	○	○
TST.B	#imm,@(R0,GBR)	(R0＋GBR)&imm,結果が 0 のとき 1→T, 0 でないとき 0→T	11001100iiiiiiii	3	テスト結果	○	○	○
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	―	○	○	○
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	―	○	○	○
XOR.B	#imm,@(R0,GBR)	(R0＋GBR) ^ imm → (R0＋GBR)	11001110iiiiiiii	3	―	○	○	○

5.1.4 シフト命令

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
ROTL	Rn	T←Rn←MSB	0100nnnn00000100	1	MSB	○	○	○
ROTR	Rn	LSB→Rn→T	0100nnnn00000101	1	LSB	○	○	○
ROTCL	Rn	T←Rn←T	0100nnnn00100100	1	MSB	○	○	○
ROTCR	Rn	T→Rn→T	0100nnnn00100101	1	LSB	○	○	○
SHAL	Rn	T←Rn←0	0100nnnn00100000	1	MSB	○	○	○
SHAR	Rn	MSB→Rn→T	0100nnnn00100001	1	LSB	○	○	○
SHLL	Rn	T←Rn←0	0100nnnn00000000	1	MSB	○	○	○
SHLR	Rn	0→Rn→T	0100nnnn00000001	1	LSB	○	○	○
SHLL2	Rn	Rn<<2 → Rn	0100nnnn00001000	1	—	○	○	○
SHLR2	Rn	Rn>>2 → Rn	0100nnnn00001001	1	—	○	○	○
SHLL8	Rn	Rn<<8 → Rn	0100nnnn00011000	1	—	○	○	○
SHLR8	Rn	Rn>>8 → Rn	0100nnnn00011001	1	—	○	○	○
SHLL16	Rn	Rn<<16 → Rn	0100nnnn00101000	1	—	○	○	○
SHLR16	Rn	Rn>>16 → Rn	0100nnnn00101001	1	—	○	○	○

5.1.5 分岐命令

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
BF	label	T=0 のとき disp×2+PC→PC, T=1 のとき nop	10001011dddddddd	3/1*2	—	○	○	○
BF/S	label	遅延分岐、T=0 のとき disp×2+PC→PC, T=1 のとき nop	10001111dddddddd	2/1*2	—	—	○	○
BT	label	T=1 のとき disp×2+PC→PC, T=0 のとき nop	10001001dddddddd	3/1*2	—	○	○	○
BT/S	label	遅延分岐、T=1 のとき disp×2+PC→PC, T=0 のとき nop	10001101dddddddd	2/1*2	—	—	○	○
BRA	label	遅延分岐、disp×2+PC→PC	1010dddddddddddd	2	—	○	○	○
BRAF	Rm	遅延分岐、Rm+PC→PC	0000mmmm00100011	2	—	—	○	○
BSR	label	遅延分岐、PC→PR, disp×2+PC→PC	1011dddddddddddd	2	—	○	○	○
BSRF	Rm	遅延分岐、PC→PR, Rm+PC→PC	0000mmmm00000011	2	—	—	○	○
JMP	@Rm	遅延分岐、Rm→PC	0100mmmm00101011	2	—	○	○	○
JSR	@Rm	遅延分岐、PC→PR, Rm→PC	0100mmmm00001011	2	—	○	○	○
RTS		遅延分岐、PR→PC	0000000000001011	2	—	○	○	○

【注】 *2 分岐しないときは 1 ステートになります。

5.1.6 システム制御命令

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
CLRMAC		0→MACH、MACL	0000000000101000	1	—	○	○	○
CLRT		0→T	0000000000001000	1	0	○	○	○
LDC	Rm,SR	Rm→SR	0100mmmm00001110	1	LSB	○	○	○
LDC	Rm,GBR	Rm→GBR	0100mmmm00011110	1	—	○	○	○
LDC	Rm,VBR	Rm→VBR	0100mmmm00101110	1	—	○	○	○
LDC	Rm,MOD	Rm→MOD	0100mmmm01011110	1	—	—	—	○
LDC	Rm,RE	Rm→RE	0100mmmm01111110	1	—	—	—	○
LDC	Rm,RS	Rm→RS	0100mmmm01101110	1	—	—	—	○
LDC.L	@Rm+,SR	(Rm)→SR、Rm＋4→Rm	0100mmmm00000111	3	LSB	○	○	○
LDC.L	@Rm+,GBR	(Rm)→GBR、Rm＋4→Rm	0100mmmm00010111	3	—	○	○	○
LDC.L	@Rm+,VBR	(Rm)→VBR、Rm＋4→Rm	0100mmmm00100111	3	—	○	○	○
LDC.L	@Rm+,MOD	(Rm)→MOD、Rm＋4→Rm	0100mmmm01010111	3	—	—	—	○
LDC.L	@Rm+,RE	(Rm)→RE、Rm＋4→Rm	0100mmmm01110111	3	—	—	—	○
LDC.L	@Rm+,RS	(Rm)→RS、Rm＋4→Rm	0100mmmm01100111	3	—	—	—	○
LDRE	@(disp,PC)	disp•2＋PC→RE	10001110dddddddd	1	—	—	—	○
LDRS	@(disp,PC)	disp•2＋PC→RS	10001100dddddddd	1	—	—	—	○
LDS	Rm,MACH	Rm→MACH	0100mmmm00001010	1	—	○	○	○
LDS	Rm,MACL	Rm→MACL	0100mmmm00011010	1	—	○	○	○
LDS	Rm,PR	Rm→PR	0100mmmm00101010	1	—	○	○	○
LDS	Rm,DSR	Rm→DSR	0100mmmm01101010	1	—	—	—	○
LDS	Rm,A0	Rm→A0	0100mmmm01111010	1	—	—	—	○
LDS	Rm,X0	Rm→X0	0100mmmm10001010	1	—	—	—	○
LDS	Rm,X1	Rm→X1	0100mmmm10011010	1	—	—	—	○
LDS	Rm,Y0	Rm→Y0	0100mmmm10101010	1	—	—	—	○
LDS	Rm,Y1	Rm→Y1	0100mmmm10111010	1	—	—	—	○
LDS.L	@Rm+,MACH	(Rm)→MACH、Rm＋4→Rm	0100mmmm00000110	1	—	○	○	○
LDS.L	@Rm+,MACL	(Rm)→MACL、Rm＋4→Rm	0100mmmm00010110	1	—	○	○	○
LDS.L	@Rm+,PR	(Rm)→PR、Rm＋4→Rm	0100mmmm00100110	1	—	○	○	○
LDS.L	@Rm+,DSR	(Rm)→DSR、Rm＋4→Rm	0100mmmm01100110	1	—	—	—	○
LDS.L	@Rm+,A0	(Rm)→A0、Rm＋4→Rm	0100mmmm01110110	1	—	—	—	○
LDS.L	@Rm+,X0	(Rm)→X0、Rm＋4→Rm	0100mmmm10000110	1	—	—	—	○
LDS.L	@Rm+,X1	(Rm)→X1、Rm＋4→Rm	0100mmmm10010110	1	—	—	—	○
LDS.L	@Rm+,Y0	(Rm)→Y0、Rm＋4→Rm	0100mmmm10100110	1	—	—	—	○
LDS.L	@Rm+,Y1	(Rm)→Y1、Rm＋4→Rm	0100mmmm10110110	1	—	—	—	○
NOP		無操作	0000000000001001	1	—	○	○	○
RTE		遅延分岐、スタック領域→PC/SR	0000000000101011	4	LSB	○	○	○
SETRC	Rm	RE－RS の演算結果（リピート状態）→RF1、RF0 Rm[11:0]→RC (SR[27:16])	0100mmmm00010100	1	—	—	—	○

命令		動作	命令コード	実行ステート	T ビット	適用命令		
						SH-1	SH-2	SH-DSP
SETRC	#imm	RE−RS の演算結果（リピート状態）→RF1、RF0 imm→RC(SR[23:16])、zeros→SR[27:24]	10000010iiiiiiii	1	1	—	—	○
SETT		1→T	0000000000011000	1	1	○	○	○
SLEEP		スリープ	0000000000011011	3*[3]	—	○	○	○
STC	SR,Rn	SR→Rn	0000nnnn00000010	1	—	○	○	○
STC	GBR,Rn	GBR→Rn	0000nnnn00010010	1	—	○	○	○
STC	VBR,Rn	VBR→Rn	0000nnnn00100010	1	—	○	○	○
STC	MOD,Rn	MOD→Rn	0000nnnn01010010	1	—	—	—	○
STC	RE,Rn	RE→Rn	0000nnnn01110010	1	—	—	—	○
STC	RS,Rn	RS→Rn	0000nnnn01100010	1	—	—	—	○
STC.L	SR,@-Rn	Rn−4→Rn、SR→(Rn)	0100nnnn00000011	2	—	○	○	○
STC.L	GBR,@-Rn	Rn−4→Rn、GBR→(Rn)	0100nnnn00010011	2	—	○	○	○
STC.L	VBR,@-Rn	Rn−4→Rn、VBR→(Rn)	0100nnnn00100011	2	—	○	○	○
STC.L	MOD,@-Rn	Rn−4→Rn、MOD→(Rn)	0100nnnn01010011	2	—	—	—	○
STC.L	RE,@-Rn	Rn−4→Rn、RE→(Rn)	0100nnnn01110011	2	—	—	—	○
STC.L	RS,@-Rn	Rn−4→Rn、RS→(Rn)	0100nnnn01100011	2	—	—	—	○
STS	MACH,Rn	MACH→Rn	0000nnnn00001010	1	—	○	○	○
STS	MACL,Rn	MACL→Rn	0000nnnn00011010	1	—	○	○	○
STS	PR,Rn	PR→Rn	0000nnnn00101010	1	—	○	○	○
STS	DSR,Rn	DSR→Rn	0000nnnn01101010	1	—	—	—	○
STS	A0,Rn	A0→Rn	0000nnnn01111010	1	—	—	—	○
STS	X0,Rn	X0→Rn	0000nnnn10001010	1	—	—	—	○
STS	X1,Rn	X1→Rn	0000nnnn10011010	1	—	—	—	○
STS	Y0,Rn	Y0→Rn	0000nnnn10101010	1	—	—	—	○
STS	Y1,Rn	Y1→Rn	0000nnnn10111010	1	—	—	—	○
STS.L	MACH,@-Rn	Rn−4→Rn、MACH→(Rn)	0100nnnn00000010	1	—	○	○	○
STS.L	MACL,@-Rn	Rn−4→Rn、MACL→(Rn)	0100nnnn00010010	1	—	○	○	○
STS.L	PR,@-Rn	Rn−4→Rn、PR→(Rn)	0100nnnn00100010	1	—	○	○	○
STS.L	DSR,@-Rn	Rn−4→Rn、DSR→(Rn)	0100nnnn01100010	1	—	—	—	○
STS.L	A0,@-Rn	Rn−4→Rn、A0→(Rn)	0100nnnn01110010	1	—	—	—	○
STS.L	X0,@-Rn	Rn−4→Rn、X0→(Rn)	0100nnnn10000010	1	—	—	—	○
STS.L	X1,@-Rn	Rn−4→Rn、X1→(Rn)	0100nnnn10010010	1	—	—	—	○
STS.L	Y0,@-Rn	Rn−4→Rn、Y0→(Rn)	0100nnnn10100010	1	—	—	—	○
STS.L	Y1,@-Rn	Rn−4→Rn、Y1→(Rn)	0100nnnn10110010	1	—	—	—	○
TRAPA	#imm	PC/SR→スタック領域、(imm•4+VBR)→PC	11000011iiiiiiii	8	—	○	○	○

【注】 *3　スリープ状態に遷移するまでのステート数です。

[注意事項]

●命令の実行ステートについて

表に示した実行ステートは最少値です。実際は、

(1) 命令フェッチとデータアクセスの競合が起こる場合

(2) ロード命令（メモリ→レジスタ）のデスティネーションレジスタと、その直後の命令が使うレジスタが同一な場合

(3) 分岐命令の分岐先アドレスが4n+2番地

などの条件により、命令実行ステート数は増加します。

5.1.7 DSP 機能をサポートする CPU 命令

DSP 機能をサポートするために CPU コア命令にいくつかのシステム制御命令が追加されました。繰り返し制御、モジュロアドレッシングをサポートする RS、RE、MOD レジスタが追加され、RC カウンタが SR レジスタに追加され、これらをアクセスするため、LDC、STC 命令が追加されました。DSP レジスタの DSR、A0、X0、X1、Y0 および Y1 レジスタをアクセスするために、LDS、STS 命令が追加されました。

SR レジスタの繰り返しカウンタ（RC、ビット 27～16）に値を設定する SETRC 命令が追加されました。SETRC 命令のオペランドがイミディエイトのときは、8 ビットのイミディエイトデータが SR レジスタのビット 23～16 に格納され、ビット 27～24 は 0 にクリアされます　オペランドがレジスタのときは、レジスタのビット 11～0 の 12 ビットが SR レジスタのビット 27～16 に格納されます。

繰り返し開始アドレス、繰り返し終了アドレスを RS、RE レジスタに設定する命令は、LDC 命令のほかに、LDRS、LDRE 命令を追加しました。

追加された命令を**表** 5.2 に示します。

表 5.2　**追加された** CPU **命令**

命令		命令コード	動作	実行ステート	T ビット
LDC	Rm,MOD	0100mmmm01011110	Rm→MOD	1	—
LDC	Rm,RE	0100mmmm01111110	Rm→RE	1	—
LDC	Rm,RS	0100mmmm01101110	Rm→RS	1	—
LDC.L	@Rm+,MOD	0100mmmm01010111	(Rm)→MOD、Rm＋4→Rm	3	—
LDC.L	@Rm+,RE	0100mmmm01110111	(Rm)→RE、Rm＋4→Rm	3	—
LDC.L	@Rm+,RS	0100mmmm01100111	(Rm)→RS、Rm＋4→Rm	3	—
STC	MOD,Rn	0000nnnn01010010	MOD→Rn	1	—
STC	RE,Rn	0000nnnn01110010	RE→Rn	1	—
STC	RS,Rn	0000nnnn01100010	RS→Rn	1	—
STC.L	MOD,@-Rn	0100nnnn01010011	Rn－4→Rn、MOD→(Rn)	2	—
STC.L	RE,@-Rn	0100nnnn01110011	Rn－4→Rn、RE→(Rn)	2	—
STC.L	RS,@-Rn	0100nnnn01100011	Rn－4→Rn、RS→(Rn)	2	—
LDS	Rm,DSR	0100mmmm01101010	Rm→DSR	1	—
LDS.L	@Rm+,DSR	0100mmmm01100110	(Rm)→DSR、Rm＋4→Rm	1	—
LDS	Rm,A0	0100mmmm01111010	Rm→A0	1	—
LDS.L	@Rm+,A0	0100mmmm01110110	(Rm)→A0、Rm＋4→Rm	1	—
LDS	Rm,X0	0100mmmm10001010	Rm→X0	1	—
LDS.L	@Rm+,X0	0100mmmm10000110	(Rm)→X0、Rm＋4→Rm	1	—
LDS	Rm,X1	0100mmmm10011010	Rm→X1	1	—
LDS.L	@Rm+,X1	0100mmmm10010110	(Rm)→X1、Rm＋4→Rm	1	—
LDS	Rm,Y0	0100mmmm10101010	Rm→Y0	1	—
LDS.L	@Rm+,Y0	0100mmmm10100110	(Rm)→Y0、Rm＋4→Rm	1	—
LDS	Rm,Y1	0100mmmm10111010	Rm→Y1	1	—
LDS.L	@Rm+,Y1	0100mmmm10110110	(Rm)→Y1、Rm＋4→Rm	1	—
STS	DSR,Rn	0000nnnn01101010	DSR→Rn	1	—

命令	命令コード	動作	実行ステート	T ビット
STS.L DSR,@-Rn	0100nnnn01100010	Rn－4→Rn、DSR→(Rn)	1	―
STS A0,Rn	0000nnnn01111010	A0→Rn	1	―
STS.L A0,@-Rn	0100nnnn01110010	Rn－4→Rn、A0→(Rn)	1	―
STS X0,Rn	0000nnnn10001010	X0→Rn	1	―
STS.L X0,@-Rn	0100nnnn10000010	Rn－4→Rn、X0→(Rn)	1	―
STS X1,Rn	0000nnnn10011010	X1→Rn	1	―
STS.L X1,@-Rn	0100nnnn10010010	Rn－4→Rn、X1→(Rn)	1	―
STS Y0,Rn	0000nnnn10101010	Y0→Rn	1	―
STS.L Y0,@-Rn	0100nnnn10100010	Rn－4→Rn、Y0→(Rn)	1	―
STS Y1,Rn	0000nnnn10111010	Y1→Rn	1	―
STS.L Y1,@-Rn	0100nnnn10110010	Rn－4→Rn、Y1→(Rn)	1	―
SETRC Rm	0100mmmm00010100	Rm[11:0]→RC (SR[27:16])、 繰り返しフラグ→RF1、RF0	1	―
SETRC #imm	10000010iiiiiiii	imm→RC(SR[23:16]),zeros→SR[27:24]、 繰り返しフラグ→RF1、RF0	1	―
LDRS @(disp,PC)	10001100dddddddd	disp × 2+PC→RS	1	―
LDRE @(disp,PC)	10001110dddddddd	disp × 2+PC→RE	1	―

5.2 DSP データ転送命令の命令セット

DSP データ転送命令を分類別に**表** 5.3 に示します。

表 5.3 DSP **データ転送命令の分類**

分類	命令の種類	オペコード	機能	命令数
ダブルデータ転送命令	4	NOPX	X **メモリ無操作**	14
		MOVX	X **メモリデータ転送**	
		NOPY	Y **メモリ無操作**	
		MOVY	Y **メモリデータ転送**	
シングルデータ転送命令	1	MOVS	**シングルデータ転送**	16
	計 5			**計** 30

データ転送命令は 2 つのグループに分けられます。ダブルデータ転送とシングルデータ転送です。ダブルデータ転送は DSP 演算命令と組み合わせて、DSP 並行処理命令することができます。並行処理命令は 32 ビット長で、A フィールドにダブルデータ転送命令が組み込まれます。並行処理命令でないダブルデータ転送とシングルデータ転送命令は 16 ビット長です。

ダブルデータ転送では X メモリと Y メモリを同時に並行してアクセスできます。それぞれ X、Y メモリデータアクセスから 1 つずつ命令を指定します。Ax ポインタは X メモリをアクセスするために使い、Ay ポインタは Y メモリをアクセスするために使います。ダブルデータ転送は X、Y メモリだけをアクセスできます。

シングルデータ転送はどこのエリアからでもアクセスできます。シングルデータ転送では Ax ポインタとその他の 2 つのポインタを As ポインタとして使います。

5.2.1 ダブルデータ転送命令（X メモリデータ）

命令	動作	命令コード	実行ステート	DC ビット
NOPX	No Operation	1111000*0*0*00**	1	—
MOVX.W @Ax,Dx	(Ax)→MSW of Dx、0→LSW of Dx	111100A*D*0*01**	1	—
MOVX.W @Ax+,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax＋2→Ax	111100A*D*0*10**	1	—
MOVX.W @Ax+Ix,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax＋Ix→Ax	111100A*D*0*11**	1	—
MOVX.W Da,@Ax	MSW of Da→(Ax)	111100A*D*1*01**	1	—
MOVX.W Da,@Ax+	MSW of Da→(Ax)、Ax＋2→Ax	111100A*D*1*10**	1	—
MOVX.W Da,@Ax+Ix	MSW of Da→(Ax)、Ax＋Ix→Ax	111100A*D*1*11**	1	—

5.2.2 ダブルデータ転送命令（Y メモリデータ）

命令	動作	命令コード	実行ステート	DC ビット
NOPY	No Operation	111100*0*0*0**00	1	—
MOVY.W @Ay,Dy	(Ay)→MSW of Dy、0→LSW of Dy	111100*A*D*0**01	1	—
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay＋2→Ay	111100*A*D*0**10	1	—
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay＋Iy→Ay	111100*A*D*0**11	1	—
MOVY.W Da,@Ay	MSW of Da→(Ay)	111100*A*D*1**01	1	—
MOVY.W Da,@Ay+	MSW of Da→(Ay)、Ay＋2→Ay	111100*A*D*1**10	1	—
MOVY.W Da,@Ay+Iy	MSW of Da→(Ay)、Ay＋Iy→Ay	111100*A*D*1**11	1	—

5.2.3 シングルデータ転送命令

命令	動作	命令コード	実行ステート	DC ビット
MOVS.W @-As,Ds	As－2→As、(As)→MSW of Ds、0→LSW of Ds	111101AADDDD0000	1	—
MOVS.W @As,Ds	(As)→MSW of Ds、0→LSW of Ds	111101AADDDD0100	1	—
MOVS.W @As+,Ds	(As)→MSW of Ds、0→LSW of Ds、As＋2→As	111101AADDDD1000	1	—
MOVS.W @As+Is,Ds	(As)→MSW of Ds、0→LSW of Ds、As＋Ix→As	111101AADDDD1100	1	—
MOVS.W Ds,@-As	As－2→As、MSW of Ds→(As)*	111101AADDDD0001	1	—
MOVS.W Ds,@As	MSW of Ds→(As)*	111101AADDDD0101	1	—
MOVS.W Ds,@As+	MSW of Ds→(As)、As＋2→As*	111101AADDDD1001	1	—
MOVS.W Ds,@As+Is	MSW of Ds→(As)、As＋Ix→As*	111101AADDDD1101	1	—
MOVS.L @-As,Ds	As－4→As、(As)→Ds	111101AADDDD0010	1	—
MOVS.L @As,Ds	(As)→Ds	111101AADDDD0110	1	—
MOVS.L @As+,Ds	(As)→Ds、As＋4→As	111101AADDDD1010	1	—
MOVS.L @As+Is,Ds	(As)→Ds、As＋Ix→As	111101AADDDD1110	1	—
MOVS.L Ds,@-As	As－4→As、Ds→(As)	111101AADDDD0011	1	—
MOVS.L Ds,@As	Ds→(As)	111101AADDDD0111	1	—
MOVS.L Ds,@As+	Ds→(As)、As＋4→As	111101AADDDD1011	1	—
MOVS.L Ds,@As+Is	Ds→(As)、As＋Ix→As	111101AADDDD1111	1	—

【注】 * **ソースオペランド** Ds **にガードビットレジスタ** A0G、A1G（8 **ビットレジスタ**）**を指定した場合は、データは符号拡張して使用されます。**

DSP データ転送のオペランドとレジスタとの対応を**表** 5.4 に示します。CPU コアのレジスタはメモリアドレスを示すポインタアドレスとして使われます。

表 5.4　DSP **データ転送のオペランドとレジスタとの対応**

オペランド	SuperH（CPU **コア**）**レジスタ**									
	R0	R1	R2(As2)	R3(As3)	R4(Ax0, As0)	R5(Ax1, As1)	R6(Ay0)	R7(Ay1)	R8(Ix,Is)	R9(Iy)
Ax					Yes	Yes				
Ix（Is）									Yes	
Dx										
Ay							Yes	Yes		
Iy										Yes
Dy										
Da										
As			Yes	Yes	Yes	Yes				
Ds										

オペランド	DSP **レジスタ**									
	X0	X1	Y0	Y1	M0	M1	A0	A1	A0G	A1G
Ax										
Ix（Is）										
Dx	Yes	Yes								
Ay										
Iy										
Dy			Yes	Yes						
Da							Yes	Yes		
As										
Ds	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

【注】 Yes：**設定可能なレジスタ**

5.3 DSP 演算命令の命令セット

DSP 演算命令は DSP ユニットで処理されるデジタル信号処理の命令です。これらの命令は 32 ビット長の命令コードで、複数の命令を並行して実行します。命令コードは A フィールド、B フィールドの 2 つに分かれており、A フィールドにはパラレルデータ転送命令を指定し、B フィールドにはシングルまたはダブルデータ演算命令を指定します。命令は独立して指定することができ、実行も独立に並行して実行されます。A フィールドに指定するパラレルデータ転送命令はダブルデータ転送命令と全く同じです。

B フィールドのデータ演算命令は 3 つに分かれています。ダブルデータ演算命令、条件付きシングルデータ演算命令、無条件シングルデータ演算命令の 3 つです。DSP 演算命令の命令形式を**表** 5.5 に示します。それぞれのオペランドは独立に DSP レジスタから選べます。DSP 演算命令のオペランドとレジスタの対応を**表** 5.6 に示します。

表 5.5 DSP **演算命令の命令形式**

分類		命令形式	命令
ダブルデータ演算命令（6 オペランド）		ALUop. Sx, Sy, Du MLTop. Se, Sf, Dg	PADD PMULS, PSUB PMULS
条件付きシングルデータ演算命令	**3 オペランド**	ALUop. Sx, Sy, Dz DCT ALUop. Sx, Sy, Dz DCF ALUop. Sx, Sy, Dz	PADD, PAND, POR, PSHA, PSHL, PSUB, PXOR
	2 オペランド	ALUop. Sx, Dz DCT ALUop. Sx, Dz DCF ALUop. Sx, Dz ALUop. Sy, Dz DCT ALUop. Sy, Dz DCF ALUop. Sy, Dz	PCOPY, PDEC, PDMSB, PINC, PLDS, PSTS, PNEG
	1 オペランド	ALUop. Dz DCT ALUop. Dz DCF ALUop. Dz	PCLR
無条件シングルデータ演算命令	**3 オペランド**	ALUop. Sx, Sy, Du MLTop. Se, Sf, Dg	PADDC, PSUBC, PMULS
	2 オペランド	ALUop. Sx, Dz ALUop. Sy, Dz ALUop. Sx, Sy	PCMP, PABS, PRND
	1 オペランド	ALUop. Dz	PSHA #imm, PSHL #imm

表 5.6 DSP 命令のオペランドとレジスタの対応

レジスタ	ALU、BPU 命令				乗算命令		
	Sx	Sy	Dz	Du	Se	Sf	Dg
A0	Yes		Yes	Yes			Yes
A1	Yes		Yes	Yes	Yes	Yes	Yes
M0		Yes	Yes				Yes
M1		Yes	Yes				Yes
X0	Yes		Yes	Yes	Yes	Yes	
X1	Yes		Yes		Yes		
Y0		Yes	Yes	Yes	Yes	Yes	
Y1		Yes	Yes			Yes	

並行命令を書くときは最初に B フィールドの命令を書いて、次に A フィールドの命令を書きます。並行処理プログラム例を**図** 5.1 に示します。

```
      PADD A0, M0, A0  PMULS X0, Y0, M0   MOVX.W @R4+, X0      MOVY.W @R6+, Y0 [;]
DCF   PINC X1, A1                         MOVX.W A0, @R5+R8   MOVY.W @R7+, Y0 [;]
      PCMP X1, M0                         MOVX.W @R4          [NOPY] [;]
```

図 5.1 並行処理プログラム例

ここで［ ］は省略可能を意味します。無操作命令 NOPX、NOPY は省略できます。';' は命令行の区切りですが、省略できます。もし区切り ';' を使うときはその後ろをコメント欄として使うことができます。

DSR レジスタの各状態コード（DC、N、Z、V、GT）は無条件の ALU 演算命令、シフト演算命令で常に更新されます。条件付き命令は条件が成立した場合でも状態コードを更新しません。乗算命令も状態コードを更新しません。DC ビットの定義は、DSR レジスタの CS ビットの指定によって決まります。

DSP 演算命令を分類別に**表** 5.7 に示します。

表 5.7　DSP **演算命令の分類**

分類		命令の種類	オペコード	機能	命令数
ALU 算術演算命令	ALU 固定小数点演算命令	11	PABS	絶対値演算	28
			PADD	加算	
			PADD　PMULS	加算と符号付き乗算	
			PADDC	キャリ付き加算	
			PCLR	クリア	
			PCMP	比較	
			PCOPY	転記	
			PNEG	符号反転	
			PSUB	減算	
			PSUB　PMULS	減算と符号付き乗算	
			PSUBC	ボロー付き減算	
	ALU 整数演算命令	2	PDEC	デクリメント	12
			PINC	インクリメント	
	MSB 検出命令	1	PDMSB	MSB 検出	6
	丸め演算命令	1	PRND	丸め演算	2
ALU 論理演算命令		3	PAND	論理積演算	9
			POR	論理和演算	
			PXOR	排他的論理和演算	
固定小数点乗算命令		1	PMULS	符号付き乗算	1
シフト	算術シフト演算命令	1	PSHA	算術シフト	4
	論理シフト演算命令	1	PSHL	論理シフト	4
システム制御命令		2	PLDS	システムレジスタのロード	12
			PSTS	システムレジスタからのストア	
		計　25			計　78

5.3.1 ALU 算術演算命令

(1) ALU 固定小数点演算命令

命令	動作	命令コード	実行ステート	DC ビット
PABS Sx,Dz	もし Sx≧0 ならば Sx→Dz もし Sx＜0 ならば 0－ Sx→Dz	111110********** 10001000xx00zzzz	1	更新
PABS Sy,Dz	もし Sy≧0 ならば Sy→Dz もし Sy＜0 ならば 0－Sy→Dz	111110********** 1010100000yyzzzz	1	更新
PADD Sx,Sy,Dz	Sx＋Sy→Dz	111110********** 10110001xxyyzzzz	1	更新
DCT PADD Sx,Sy,Dz	もし DC＝1 ならば Sx＋Sy→Dz もし 0 ならば nop	111110********** 10110010xxyyzzzz	1	—
DCF PADD Sx,Sy,Dz	もし DC＝0 ならば Sx＋Sy→Dz もし 1 ならば nop	111110********** 10110011xxyyzzzz	1	—
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx＋Sy→Du Se の上位ワード×Sf の上位ワード→Dg	111110********** 0111eeffxxyygguu	1	更新
PADDC Sx,Sy,Dz	Sx＋Sy＋DC→Dz	111110********** 10110000xxyyzzzz	1	更新
PCLR Dz	H'00000000→Dz	111110********** 100011010000zzzz	1	更新
DCT PCLR Dz	もし DC＝1 ならば H'00000000→Dz もし 0 ならば nop.	111110********** 100011100000zzzz	1	—
DCF PCLR Dz	もし DC＝0 ならば H'00000000→Dz もし 1 ならば nop.	111110********** 100011110000zzzz	1	—
PCMP Sx,Sy	Sx－Sy	111110********** 10000100xxyy0000	1	更新
PCOPY Sx,Dz	Sx→Dz	111110********** 11011001xx00zzzz	1	更新
PCOPY Sy,Dz	Sy→Dz	111110********** 1111100100yyzzzz	1	更新
DCT PCOPY Sx,Dz	もし DC＝1 ならば Sx→Dz もし 0 ならば nop.	111110********** 11011010xx00zzzz	1	—
DCT PCOPY Sy,Dz	もし DC＝1 ならば Sy→Dz もし 0 ならば nop.	111110********** 1111101000yyzzzz	1	—
DCF PCOPY Sx,Dz	もし DC＝0 ならば Sx→Dz もし 1 ならば nop	111110********** 11011011xx00zzzz	1	—
DCF PCOPY Sy,Dz	もし DC＝0 ならば Sy→Dz もし 1 ならば nop	111110********** 1111101100yyzzzz	1	—
PNEG Sx,Dz	0－Sx→Dz	111110********** 11001001xx00zzzz	1	更新
PNEG Sy,Dz	0－Sy→Dz	111110********** 1110100100yyzzzz	1	更新
DCT PNEG Sx,Dz	もし DC＝1 ならば 0－Sx→Dz もし 0 ならば nop.	111110********** 11001010xx00zzzz	1	—

命令	動作	命令コード	実行ステート	DCビット
DCT PNEG Sy,Dz	**もし** DC＝1 **ならば** 0－Sy→Dz **もし** 0 **ならば、** nop.	111110********** 1110101000yyzzzz	1	―
DCF PNEG Sx,Dz	**もし** DC＝0 **ならば** 0－Sx→Dz **もし** 1 **ならば** nop.	111110********** 11001011xx00zzzz	1	―
DCF PNEG Sy,Dz	**もし** DC＝0 **ならば** 0－Sy→Dz **もし** 1 **ならば** nop.	111110********** 1110101100yyzzzz	1	―
PSUB Sx,Sy,Dz	Sx－Sy→Dz	111110********** 10100001xxyyzzzz	1	**更新**
DCT PSUB Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx－Sy→Dz **もし** 0 **ならば** nop	111110********** 10100010xxyyzzzz	1	―
DCF PSUB Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx－Sy→Dz **もし** 1 **ならば** nop	111110********** 10100011xxyyzzzz	1	―
PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx－Sy→Du Se **の上位ワード**×Sf **の上位ワード**→Dg	111110********** 0110eeffxxyygguu	1	**更新**
PSUBC Sx,Sy,Dz	Sx－Sy－DC→Dz	111110********** 10100000xxyyzzzz	1	**更新**

(2) ALU **整数演算命令**

命令	動作	命令コード	実行ステート	DCビット
PDEC Sx,Dz	Sx **の上位ワード**－1→Dz **の上位ワード** Dz **の下位ワードをクリア**	111110********** 10001001xx00zzzz	1	**更新**
PDEC Sy,Dz	Sy **の上位ワード**－1→Dz **の上位ワード** Dz **の下位ワードをクリア**	111110********** 1010100100yyzzzz	1	**更新**
DCT PDEC Sx,Dz	**もし** DC＝1 **ならば** Sx **の上位ワード**－1→Dz **の上位ワード、** Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10001010xx00zzzz	1	―
DCT PDEC Sy,Dz	**もし** DC＝1 **ならば** Sy **の上位ワード**－1→Dz **の上位ワード、** Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 1010101000yyzzzz	1	―
DCF PDEC Sx,Dz	**もし** DC＝0 **ならば** Sx **の上位ワード**－1→Dz **の上位ワード、** Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10001011xx00zzzz	1	―
DCF PDEC Sy,Dz	**もし** DC＝0 **ならば** Sy **の上位ワード**－1→Dz **の上位ワード、** Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 1010101100yyzzzz	1	―
PINC Sx,Dz	Sx **の上位ワード**＋1→Dz **の上位ワード** Dz **の下位ワードクリア**	111110********** 10011001xx00zzzz	1	**更新**

命令	動作	命令コード	実行ステート	DCビット
PINC Sy,Dz	Sy の上位ワード＋1→Dz の上位ワード Dz の下位ワードクリア	111110********** 1011100100yyzzzz	1	更新
DCT PINC Sx,Dz	もし DC＝1 ならば Sx の上位ワード＋1→Dz の上位ワード、 Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10011010xx00zzzz	1	—
DCT PINC Sy,Dz	もし DC＝1 ならば Sy の上位ワード＋1→Dz の上位ワード、 Dz の下位ワードクリア もし 0 ならば nop.	111110********** 1011101000yyzzzz	1	—
DCF PINC Sx,Dz	もし DC＝0 ならば Sx の上位ワード＋1→Dz の上位ワード、 Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10011011xx00zzzz	1	—
DCF PINC Sy,Dz	もし DC＝0 ならば Sy の上位ワード＋1→Dz の上位ワード、 Dz の下位ワードクリア もし 1 ならば nop.	111110********** 1011101100yyzzzz	1	—

(3) MSB 検出命令

命令	動作	命令コード	実行ステート	DCビット
PDMSB Sx,Dz	Sx データの MSB 位置→Dz の上位ワード、 Dz の下位ワードクリア	111110********** 10011101xx00zzzz	1	更新
PDMSB Sy,Dz	Sy データの MSB 位置→Dz の上位ワード、 Dz の下位ワードクリア	111110********** 1011110100yyzzzz	1	更新
DCT PDMSB Sx,Dz	もし DC＝1 ならば Sx データの MSB 位置→Dz の上位ワード、 Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10011110xx00zzzz	1	—
DCT PDMSB Sy,Dz	もし DC＝1 ならば Sy データの MSB 位置→Dz の上位ワード、 Dz の下位ワードクリア もし 0 ならば nop.	111110********** 1011111000yyzzzz	1	—
DCF PDMSB Sx,Dz	もし DC＝0 ならば Sx データの MSB 位置→Dz の上位ワード、 Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10011111xx00zzzz	1	—
DCF PDMSB Sy,Dz	もし DC＝0 ならば Sy データの MSB 位置→Dz の上位ワード、 Dz の下位ワードクリア もし 1 ならば nop.	111110********** 1011111100yyzzzz	1	—

(4) 丸め演算命令

命令	動作	命令コード	実行ステート	DCビット
PRND Sx,Dz	Sx＋H'00008000→Dz Dz の下位ワードクリア	111110********** 10011000xx00zzzz	1	更新
PRND Sy,Dz	Sy＋H'00008000→Dz Dz の下位ワードクリア	111110********** 1011100000yyzzzz	1	更新

5.3.2 ALU 論理演算命令

命令	動作	命令コード	実行ステート	DCビット
PAND Sx,Sy,Dz	Sx & Sy→ Dz Dz の下位ワードクリア	111110********** 10010101xxyyzzzz	1	更新
DCT PAND Sx,Sy,Dz	もし DC＝1 ならば Sx & Sy→Dz Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10010110xxyyzzzz	1	—
DCF PAND Sx,Sy,Dz	もし DC＝0 ならば Sx & Sy→Dz Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10010111xxyyzzzz	1	—
POR Sx,Sy,Dz	Sx | Sy→Dz Dz の下位ワードクリア	111110********** 10110101xxyyzzzz	1	更新
DCT POR Sx,Sy,Dz	もし DC＝1 ならば Sx | Sy→Dz Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10110110xxyyzzzz	1	—
DCF POR Sx,Sy,Dz	もし DC＝0 ならば Sx | Sy→Dz Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10110111xxyyzzzz	1	—
PXOR Sx,Sy,Dz	Sx ^ Sy→Dz Dz の下位ワードクリア	111110********** 10100101xxyyzzzz	1	更新
DCT PXOR Sx,Sy,Dz	もし DC＝1 ならば Sx ^ Sy→Dz Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10100110xxyyzzzz	1	—
DCF PXOR Sx,Sy,Dz	もし DC＝0 ならば Sx ^ Sy→Dz Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10100111xxyyzzzz	1	—

5.3.3 固定小数点乗算命令

命令	動作	命令コード	実行ステート	DCビット
PMULS Se,Sf,Dg	Se の上位ワード×Sf の上位ワード→Dg	111110********** 0100eeff0000gg00	1	—

5.3.4 シフト演算命令

(1) 算術シフト演算命令

命令	動作	命令コード	実行ステート	DCビット
PSHA Sx,Sy,Dz	**もし** Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** Sy＜0 **ならば** Sx＞＞Sy→Dz	111110********** 10010001xxyyzzzz	1	**更新**
DCT PSHA Sx,Sy,Dz	**もし** DC＝1 & Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** DC＝1 & Sy＜0 **ならば** Sx＞＞Sy→Dz **もし** DC＝0 **ならば** nop	111110********** 10010010xxyyzzzz	1	—
DCF PSHA Sx,Sy,Dz	**もし** DC＝0 & Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** DC＝0 & Sy＜0 **ならば** Sx＞＞Sy→Dz **もし** DC＝1 **ならば** nop	111110********** 10010011xxyyzzzz	1	—
PSHA #imm,Dz	**もし** imm≧0 **ならば** Dz＜＜imm→Dz **もし** imm＜0 **ならば** Dz＞＞imm→Dz	111110********** 00010iiiiiiizzzz	1	**更新**

(2) 論理シフト演算命令

命令	動作	命令コード	実行ステート	DCビット
PSHL Sx,Sy,Dz	**もし** Sy≧0 **ならば** Sx＜＜Sy→Dz, Dz **の下位ワードクリア** **もし** Sy＜0 **ならば** Sx＞＞Sy→Dz, Dz **の下位ワードクリア**	111110********** 10000001xxyyzzzz	1	**更新**
DCT PSHL Sx,Sy,Dz	**もし** DC＝1 & Sy≧0 **ならば** Sx＜＜Sy→Dz, Dz **の下位ワードクリア** **もし** DC＝1 & Sy＜0 **ならば** Sx＞＞Sy→Dz, Dz **の下位ワードクリア** **もし** DC＝0 **ならば** nop	111110********** 10000010xxyyzzzz	1	—
DCF PSHL Sx,Sy,Dz	**もし** DC＝0 & Sy≧0 **ならば** Sx＜＜Sy→Dz, Dz **の下位ワードクリア** **もし** DC＝0 & Sy＜0 **ならば** Sx＞＞Sy→Dz, Dz **の下位ワードクリア** **もし** DC＝1 **ならば** nop	111110********** 10000011xxyyzzzz	1	—
PSHL #imm,Dz	**もし** imm≧0 **ならば** Dz＜＜imm→Dz, Dz **の下位ワードクリア** **もし** imm＜0 **ならば** Dz＞＞imm→Dz, Dz **の下位ワードクリア**	111110********** 00000iiiiiiizzzz	1	**更新**

5.3.5 システム制御命令

命令	動作	命令コード	実行ステート	DCビット
PLDS Dz,MACH	Dz→MACH	111110********** 1110110100000zzzz	1	—
PLDS Dz,MACL	Dz→MACL	111110********** 1111110100000zzzz	1	—
DCT PLDS Dz,MACH	**もし** DC＝1 **ならば** Dz→MACH **もし** 0 **ならば** nop.	111110********** 1110111000000zzzz	1	—
DCT PLDS Dz,MACL	**もし** DC＝1 **ならば** Dz→MACL **もし** 0 **ならば** nop.	111110********** 1111111000000zzzz	1	—
DCF PLDS Dz,MACH	**もし** DC＝0 **ならば** Dz→MACH **もし** 1 **ならば** nop.	111110********** 1110111100000zzzz	1	—
DCF PLDS Dz,MACL	**もし** DC＝0 **ならば** Dz→MACL **もし** 1 **ならば** nop.	111110********** 1111111100000zzzz	1	—
PSTS MACH,Dz	MACH→Dz	111110********** 1100110100000zzzz	1	—
PSTS MACL,Dz	MACL→Dz	111110********** 1101110100000zzzz	1	—
DCT PSTS MACH,Dz	**もし** DC＝1 **ならば** MACH→Dz **もし** 0 **ならば** nop.	111110********** 1100111000000zzzz	1	—
DCT PSTS MACL,Dz	**もし** DC＝1 **ならば** MACL→Dz **もし** 0 **ならば** nop.	111110********** 1101111000000zzzz	1	—
DCF PSTS MACH,Dz	**もし** DC＝0 **ならば** MACH→Dz **もし** 1 **ならば** nop.	111110********** 1100111100000zzzz	1	—
DCF PSTS MACL,Dz	**もし** DC＝0 **ならば** MACL→Dz **もし** 1 **ならば** nop.	111110********** 1101111100000zzzz	1	—

5.3.6 NOPX と NOPY の命令コード

DSP 演算命令と同時に並行処理されるデータ転送命令がないときは、データ転送命令に NOPX、NOPY 命令を書くかあるいは命令を省略することもできます。NOPX、NOPY 命令を書いても省略しても命令コードは同じです。NOPX と NOPY の命令コードの例を**表** 5.8 に示します。

表 5.8 NOPX **と** NOPY **の命令コードの例**

命令			コード
PADD X0, Y0, A0	MOVX. W @R4+, X0	MOVY.W @R6+R9, Y0	1111100000001011 1011000100000111
PADD X0, Y0, A0	NOPX	MOVY.W @R6+R9, Y0	1111100000000011 1011000100000111
PADD X0, Y0, A0	NOPX	NOPY	1111100000000000 1011000100000111
PADD X0, Y0, A0	NOPX		1111100000000000 1011000100000111
PADD X0, Y0, A0			1111100000000000 1011000100000111
	MOVX. W @R4+, X0	MOVY.W @R6+R9, Y0	1111000000001011
	MOVX. W @R4+, X0	NOPY	1111000000001000
	MOVS. W @R4+, X0		1111010010001000
	NOPX	MOVY.W @R6+R9, Y0	1111000000000011
		MOVY.W @R6+R9, Y0	1111000000000011
	NOPX	NOPY	1111000000000000
NOP			0000000000001001

6. 命令の説明

CPU 命令、DSP データ転送命令、DSP 演算命令に分けて、それぞれをアルファベット順に説明します。

6.1 CPU 命令の説明

以下の形式でアルファベット順に説明します。

命令の名称　　命令の機能(英文)　　命令の分類
命令の機能　　遅延分岐命令、または割り込み禁止命令の表示

書式	動作概略	命令コード	実行ステート	T ビット	適用命令
アセンブラの入力書式で表示しています。imm、disp は数値、式またはシンボルになります。	動作の概略を表示しています。	MSB←→LSB の順で表示しています。	ノーウェイトのときの値です。	命令実行後の、T ビットの値を表示しています。	命令が SH-1、SH-2、SH-DSP のどの CPU に適用しているかを示します。

(1) **説明**

動作の説明を行います。

(2) **注意**

命令を使用する上で特に注意が必要なことを説明します。

(3) **動作内容**

C で動作内容を表示しています。ここでは以下の資源の使用を仮定しています。

```
unsigned char  Read_Byte(unsigned long Addr);
unsigned short Read_Word(unsigned long Addr);
unsigned long  Read_Long(unsigned long Addr);
```

アドレス Addr のそれぞれのサイズの内容を返します。2n 番地以外からのワード、4n 番地以外からのロングワードの読み込みはアドレスエラーとして検出します。

```
unsigned char Write_Byte(unsigned long Addr, unsigned long Data);
unsigned short Write_Word(unsigned long Addr, unsigned long Data);
unsigned long Write_Long(unsigned long Addr, unsigned long Data);
```

アドレス Addr にデータ Data をそれぞれのサイズで書き込みます。2n 番地以外へのワード、4n 番地以外へのロングワードの書き込みはアドレスエラーとして検出します。

```
Delay_Slot(unsigned long Addr);
```
アドレス(Addr-4)のスロット命令に実行を移します。これは例えば“ Delay_Slot(4); ”のとき、4 番地ではなく 0 番地の命令に実行が移ることを意味します。また、この関数から以下の命令に実行が移されようとすると、その直前に以下の命令をスロット不当命令として検出します。遅延スロット命令が以下の命令だと、スロット不当命令となります。

BF、BT、BRA、BSR、JMP、JSR、RTS、RTE、TRAPA、BF/S、BT/S、BRAF、BSRF

```
unsigned long IS_32bit,Inst(unsigned long Addr);
```
アドレス（Addr_4）の命令が 32bit 長だと 2 を返し、16bit 長だと 0 を返します。

```
unsigned long R [16];
unsigned long SR,GBR,VBR;
unsigned long MACH,MACL,PR;
unsigned long PC;
```
各レジスタの本体

```
struct SR0 {
  unsigned long     dummy0:4;
  unsigned long        RC0:12;
  unsigned long     dummy1:4;
  unsigned long       DMY0:1;
  unsigned long       DMX0:1;
  unsigned long         M0:1;
  unsigned long         Q0:1;
  unsigned long         I0:4;
  unsigned long       RF10:1;
  unsigned long       RF00:1;
  unsigned long         S0:1;
  unsigned long         T0:1;
};
```
SR の構造の定義

```
#define M ((*(struct SR0 *)(&SR)).M0)
#define Q ((*(struct SR0 *)(&SR)).Q0)
#define S ((*(struct SR0 *)(&SR)).S0)
#define T ((*(struct SR0 *)(&SR)).T0)
#define RF1 ((*(struct SR0 *)(&SR)).RF10)
#define RF0 ((*(struct SR0 *)(&SR)).RF00)
```
SR 内ビットの定義

```
Error( char *er );
```
エラー表示関数

これ以外に、PC は現在実行中の命令の 4 バイト先を示しているものと仮定しています。これは、たとえば“ PC＝4; ”は 4 番地ではなく 0 番地の命令に実行が移ることを意味します。

(4) **使用例**

アセンブラニーモニックで例を示し、命令の実行前後の状態を表示しています。

イタリック字体(例: *.align*) はアセンブラ制御命令であることを示します。アセンブラ制御命令の意味は次のようになります。詳しくは、**「クロスアセンブラユーザーズマニュアル」**を参照してください。

.org	ロケーションカウンタ設定
.data.w	ワード整数データ確保
.data.l	ロングワード整数データ確保
.sdata	文字列データ確保
.align 2	2 バイト境界調整
.align 4	4 バイト境界調整
.arepeat 16	16 回繰り返し展開
.arepeat 32	32 回繰り返し展開
.aendr	回数指定繰り返し展開終了

【注】 SH **シリーズクロスアセンブラ** Ver 1.0 **では、条件付きアセンブラ機能をサポートしておりません。**

【注】 *1 **下記のディスプレースメント** (disp) **を伴うアドレッシングモードにおいて、本マニュアルのアセンブラ記述は、オペランドサイズに応じたスケーリング**(×1、×2、×4) **を行う前の値を書いています。これは、**LSI **の動作を明確にするためで、実際のアセンブラの記述は、各アセンブラの表記ルールをご参照下さい。**

@ (disp : 4, Rn) ; **ディスプレースメント付きレジスタ間接**

@ (disp : 8, GBR) ; **ディスプレースメント付き** GBR **間接**

@ (disp : 8, PC) ; **ディスプレースメント付き** PC **相対**

disp : 8, disp : 12; PC **相対**

*2 **命令コード** 16 **ビットのうち、命令として割り当てられていないコードは一般不当命令として扱われ、不当命令例外処理を発生します。**

例 H'FFFF **[一般不当命令]**

*3 BRA, BT/S **などの遅延分岐命令の次命令が一般不当命令または分岐命令であると** (**これをスロット不当命令といいます**)**、不当命令例外処理を発生します。**

例 1 ・・・・

```
BRA   LABEL
.data. w   H'FFFF    ←  スロット不当命令
・・・・    [H'FFFF は本来一般不当命令]
```

例 2 RTE

BT/S LABEL ← **スロット不当命令**

*4 **遅延分岐の分岐動作そのものは、スロット命令実行後に発生します。しかし、レジスタの更新などの分岐動作を除く命令の実行は、遅延分岐命令、遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されているレジスタの内容を変更しても、分岐先アドレスは変更前のレジスタ内容のままです。**

*5 3 **命令以下の繰り返しプログラム（ループ）内または、**4 **命令以上の繰り返しプログラム（ループ）内の最後の** 3 **命令に、一般不当命令、分岐命令または、**SR**、**RS**、**RE **レジスタを更新する命令（**SETRC**、**LDRS **など）が存在すると、不当命令例外処理を発生します。詳しくは、「**4.19 DSP **繰り返し（ループ）制御」を参照してください。**

6.1.1 ADD ADD binary 算術演算命令

2 進加算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ADD Rm,Rn	Rn＋Rm→Rn	0011nnnnmmmm1100	1	―	○	○	○
ADD #imm,Rn	Rn＋imm→Rn	0111nnnniiiiiiii	1	―	○	○	○

(1) 説明

汎用レジスタ Rn の内容と Rm とを加算し、結果を Rn に格納します。
汎用レジスタ Rn と 8 ビットのイミディエイトデータとの加算も可能です。
8 ビットのイミディエイトデータは 32 ビットに符号拡張しますので減算との兼用が可能です。

(2) 動作内容

```
ADD(long m, long n)      /* ADD Rm,Rn */
{
    R[n]+=R[m];
    PC+=2;
}

ADDI(long i, long n)     /* ADD #imm,Rn */
{
    if ((i&0x80)==0) R[n]+=(0x000000FF & (long)i);
    else R[n]+=(0xFFFFFF00 | (long)i);
    PC+=2;
}
```

(3) 使用例

```
ADD  R0,R1                      ;実行前 R0=H'7FFFFFFF,R1=H'00000001
                                ;実行後 R1=H'80000000
ADD  #H'01,R2                   ;実行前 R2=H'00000000
                                ;実行後 R2=H'00000001
ADD  #H'FE,R3                   ;実行前 R3=H'00000001
                                ;実行後 R3=H'FFFFFFFF
```

6.1.2 ADDC ADD with Carry 算術演算命令

キャリ付き 2 進加算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ADDC Rm,Rn	Rn＋Rm＋T→Rn、キャリ→T	0011nnnnmmmm1110	1	キャリ	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm と T ビットを加算し、結果を Rn に格納します。演算の結果によってキャリを T ビットに反映します。32 ビットを超える加算を行うとき使用します。

(2) **動作内容**

```
ADDC(long m, long n)    /* ADDC Rm,Rn */
{
unsigned long tmp0,tmp1;

tmp1=R[n]+R[m];
     tmp0=R[n];
     R[n]=tmp1+T;
     if (tmp0>tmp1) T=1;
     else T=0;
     if (tmp1>R[n]) T=1;
     PC+=2;
}
```

(3) **使用例**

```
CLRT                ; R0:R1(64ﾋﾞｯﾄ)+R2:R3(64ﾋﾞｯﾄ)=R0:R1(64ﾋﾞｯﾄ)
ADDC  R3,R1         ;実行前 T=0,R1=H'00000001,R3=H'FFFFFFFF
                    ;実行後 T=1,R1=H'00000000
ADDC  R2,R0         ;実行前 T=1,R0=H'00000000,R2=H'00000000
                    ;実行後 T=0,R0=H'00000001
```

6.1.3 ADDV ADD with (Vflag) overflow check 算術演算命令

オーバフロー付き 2 進加算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ADDV Rm,Rn	Rn＋Rm→Rn、 オーバフロー→T	0011nnnnmmmm1111	1	オーバフロー	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm とを加算し、結果を Rn に格納します。オーバフローが発生すると、T ビットをセットします。

(2) **動作内容**

```
ADDV(long m, long n)    /* ADDV Rm,Rn */
{
     long dest,src,ans;

     if ((long)R[n]>=0) dest=0;
     else dest=1;
     if ((long)R[m]>=0) src=0;
     else src=1;
     src+=dest;
     R[n]+=R[m];
     if ((long)R[n]>=0) ans=0;
     else ans=1;
     ans+=dest;
     if (src==0 || src==2) {
          if (ans==1) T=1;
          else T=0;
     }
     else T=0;
     PC+=2;
}
```

(3) **使用例**

```
ADDV  R0,R1         ;実行前 R0=H'00000001,R1=H'7FFFFFFE,      T=0
                    ;実行後 R1=H'7FFFFFFF,                    T=0
ADDV  R0,R1         ;実行前 R0=H'00000002,R1=H'7FFFFFFE,      T=0
                    ;実行後 R1=H'80000000,                    T=1
```

6.1.4 AND AND logical 論理演算命令

論理積演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
AND Rm,Rn	Rn & Rm→Rn	0010nnnnmmmm1001	1	―	○	○	○
AND #imm,R0	R0 & imm→R0	11001001iiiiiiii	1	―	○	○	○
AND.B #imm, @(R0,GBR)	(R0＋GBR) & imm → (R0＋GBR)	11001101iiiiiiii	3	―	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm の論理積をとり、結果を Rn に格納します。

汎用レジスタ R0 とゼロ拡張した 8 ビットのイミディエイトデータとの論理積、もしくはインデックス付き GBR 間接アドレッシングモードで 8 ビットのメモリと 8 ビットのイミディエイトデータとの論理積が可能です。

(2) **注意**

AND #imm,R0 では演算の結果、R0 の上位 24 ビットは常にクリアされます。

(3) **動作内容**

```
AND(long m, long n)    /* AND Rm,Rn */
{
     R[n]&=R[m];
     PC+=2;
}

ANDI(long i)    /* AND #imm,R0 */
{
     R[0]&=(0x000000FF & (long)i);
     PC+=2;
}

ANDM(long i)    /* AND.B #imm,@(R0,GBR) */
{
     long temp;

     temp=(long)Read_Byte(GBR+R[0]);
     temp&=(0x000000FF & (long)i);
     Write_Byte(GBR+R[0],temp);
     PC+=2;
}
```

(4) **使用例**

```
AND   R0,R1                 ;実行前 R0=H'AAAAAAAA,R1=H'55555555
                            ;実行後 R1=H'00000000
AND   #H'0F,R0              ;実行前 R0=H'FFFFFFFF
                            ;実行後 R0=H'0000000F
AND.B #H'80,@(R0,GBR)       ;実行前 @(R0,GBR)=H'A5
                            ;実行後 @(R0,GBR)=H'80
```

6.1.5 BF Branch if False 分岐命令

条件分岐

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
BF label	T＝0 **のとき** disp×2＋PC→PC、T＝1 **のとき** nop	10001011dddddddd	3/1	―	○	○	○

(1) **説明**

T ビットを参照する条件付き分岐命令です。T＝0 のとき、分岐先アドレスに分岐します。T＝1 のときは、次の命令を実行します。

分岐先は PC にディスプレースメントを加えたアドレスです。ただし、この際アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。8 ビットディスプレースメントは符号拡張後 2 倍しますので、分岐先との相対距離は－256 バイトから＋254 バイトの範囲になります。分岐先に届かないときは BRA 命令などとの組み合わせで対応する必要があります。

(2) **注意**

分岐するときは 3 ステート、分岐しないときは 1 ステートになります。

(3) **動作内容**

```
BF(long d)    /* BF disp */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==0) PC=PC+(disp<<1);
    else PC+=2;
}
```

(4) **使用例**

```
      CLRT              ;常に T=0
      BT   TRGET_T      ;T=0 のため分岐しません。
      BF   TRGET_F      ;T=0 のため TRGET_F へ分岐します。
      NOP               ;
      NOP               ;←BF 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
      ---
TRGET_F:                ;←BF 命令の分岐先
```

6.1.6 BF/S Branch if False with delay Slot 分岐命令
条件付き遅延分岐 遅延分岐命令

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
BF/S label	T＝0 のとき disp×2＋PC→PC、 T＝1 のとき nop	10001111dddddddd	2/1	―	―	○	○

(1) 説明

T ビットを参照する条件付き遅延分岐命令です。 T＝0 のとき、次の命令を実行した後で分岐します。 T＝1 のときは、次の命令を実行します。

分岐先は PC にディスプレースメントを加えたアドレスです。ただし、この際アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。8 ビットディスプレースメントは符号拡張後 2 倍しますので、分岐先との相対距離は－256 バイトから＋254 バイトの範囲になります。分岐先に届かないときは BRA 命令などとの組み合わせで対応する必要があります。

(2) 注意

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

分岐するときは 2 ステート、分岐しないときは 1 ステートになります。

(3) 動作内容

```
BFS(long d)   /* BFS disp */
{
     long disp;
     unsigned long temp;

     temp=PC;
     if ((d&0x80)==0) disp=(0x000000FF & (long)d);
     else disp=(0xFFFFFF00 | (long)d);
     if (T==0) {
          PC=PC+(disp<<1);
          Delay_Slot(temp+2);
     }
     else PC+=2;
}
```

(4) **使用例**

```
      CLRT              ;常に T=0
      BT/S TRGET_T      ;T=0 のため分岐しません。
      NOP               ;
      BF/S TRGET_F      ;T=0 のため TRGET_F に分岐します。
      ADD R0,R1         ;分岐に先立ち実行します。
      NOP               ;←BF/S 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
      ---
TRGET_F                 ;←BF/S 命令の分岐先
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.7 BRA BRAnch 分岐命令

無条件分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
BRA label	disp×2+PC→PC	1010dddddddddddd	2	―	○	○	○

(1) **説明**

無条件の遅延分岐命令です。分岐先は PC にディスプレースメントを加えたアドレスです。ただし、この際アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。12 ビットディスプレースメントは符号拡張後 2 倍しますので、分岐先との相対距離は−4096 バイトから+4094 バイトの範囲になります。分岐先に届かないときは、分岐先アドレスを MOV 命令でレジスタに転送した上で、JMP 命令への変更が必要です。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
BRA(long d)/* BRA disp */
{
     unsigned long temp;

     long disp;
     if ((d&0x800)==0) disp=(0x00000FFF & (long) d);
     else disp=(0xFFFFF000 | (long) d);
     temp=PC;
     PC=PC+(disp<<1);
     Delay_Slot(temp+2);
}
```

(4) **使用例**

```
     BRA  TRGET          ;TRGET へ分岐します。
     ADD  R0,R1          ;分岐に先立ち実行します。
     NOP                 ;←BRA 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
     ---
TRGET:                   ;←BRA 命令の分岐先
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.8 BRAF BRAnch Far 分岐命令

無条件分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
BRAF Rm	Rm＋PC→PC	0000mmmm00100011	2	―	―	○	○

(1) **説明**

無条件の遅延分岐命令です。分岐先は PC に汎用レジスタ Rm の内容の 32 ビットを加えたアドレスです。この際、アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
BRAF(long m)   /* BRAF Rm */
{
     unsigned long temp;

     temp=PC;
     PC+=R[m];
     Delay_Slot(temp+2);
}
```

(4) **使用例**

```
     MOV.L #(TRGET-BSRF_PC),R0 ;ﾃﾞｨｽﾌﾟﾚｰｽﾒﾝﾄを設定します。
     BRAF  R0                  ;TRGET へ分岐します。
     ADD   R0,R1               ;分岐に先立ち実行します。
BRAF_PC:                       ;←BRAF 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
     NOP
     ---
TRGET:                         ;←BRAF 命令の分岐先
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.9 BSR Branch to SubRoutine 分岐命令

サブルーチンプロシージャへの分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
BSR label	PC→PR、disp×2＋PC→PC	1011dddddddddddd	2	―	○	○	○

(1) **説明**

指定されたアドレスのサブルーチンプロシージャへ遅延分岐します。PC の内容を PR に退避し、PC にディスプレースメントを加えたアドレスへ分岐します。この際、アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。12 ビットディスプレースメントは符号拡張後 2 倍しますので、分岐先との相対距離は－4096 バイトから＋4094 バイトの範囲になります。分岐先に届かないときは、分岐先アドレスを MOV 命令でレジスタに転送した上で、JSR 命令への変更が必要です。RTS と組み合わせて、サブルーチンプロシージャコールに使用します。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
BSR(long d)     /* BSR disp */
{
     long disp;

     if ((d&0x800)==0) disp=(0x00000FFF & (long) d);
     else disp=(0xFFFFF000 | (long) d);
     PR=PC+Is_32bit_Inst(PR+2);
     PC=PC+(disp<<1);
     Delay_Slot(PR+2);
}
```

(4) **使用例**

```
     BSR  TRGET      ;TRGET へ分岐します。
     MOV  R3,R4      ;分岐に先立ち実行します。
     ADD  R0,R1      ;←BSR で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置であり
     .....            ﾌﾟﾛｼｰｼﾞｬからの戻り先（PR の内容）です。
     .....
TRGET:               ;←ﾌﾟﾛｼｰｼﾞｬの入り口
     MOV  R2,R3      ;
     RTS             ;上記 ADD 命令に戻ります。
     MOV  #1,R0      ;分岐に先立ち実行します。
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、

命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.10 BSRF Branch to SubRoutine Far 分岐命令

サブルーチンプロシージャへの分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
BSRF Rm	PC→PR、Rm＋PC→PC	0000mmmm00000011	2	―	―	○	○

(1) 説明

指定されたアドレスのサブルーチンプロシージャへ遅延分岐します。PC の内容を PR に退避します。分岐先は PC に汎用レジスタ Rm の内容の 32 ビットデータを加えたアドレスです。この際、アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。RTS と組み合わせて、サブルーチンプロシージャコールに使用します。

(2) 注意

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) 動作内容

```
     BSRF(long m)    /* BSRF Rm */
{
    PR=PC;
    PC+=R[m];
    Delay_Slot(PR+2);
}
```

(4) 使用例

```
   MOV.L #(TRGET-BSRF_PC),R0      ;ﾃﾞｨｽﾌﾟﾚｰｽﾒﾝﾄを設定します。
   BSRF  R0                       ;TRGET へ分岐します。
   MOV   R3,R4                    ;分岐に先立ち実行します。
BSRF_PC:                          ;←BSRF 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
   ADD   R0,R1                    ;
    .....
    .....
TRGET:                            ;←ﾌﾟﾛｼｰｼﾞｬの入り口
   MOV  R2,R3                     ;
   RTS                            ;上記 ADD 命令に戻ります。
   MOV  #1,R0                     ;分岐に先立ち実行します。
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.11 BT Branch if True 分岐命令

条件分岐

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
BT label	T＝1 のとき disp×2＋PC→PC、T＝0 のとき nop	10001001dddddddd	3/1	―	○	○	○

(1) **説明**

T ビットを参照する条件付き分岐命令です。T＝1 のとき、分岐します。T＝0 のときは、次の命令を実行します。

分岐先は PC にディスプレースメントを加えたアドレスです。この際、アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。8 ビットディスプレースメントは符号拡張後 2 倍しますので、分岐先との相対距離は－256 バイトから＋254 バイトの範囲になります。分岐先に届かないときは BRA 命令などとの組み合わせで対応する必要があります。

(2) **注意**

分岐するときは 3 ステート、分岐しないときは 1 ステートになります。

(3) **動作内容**

```
BT(long d)    /* BT disp */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==1) PC=PC+(disp<<1);
    else PC+=2;
}
```

(4) **使用例**

```
      SETT              ;常に T=1
      BF   TRGET_F      ;T=1 のため分岐しません。
      BT   TRGET_T      ;T=1 のため TRGET_T へ分岐します。
      NOP               ;
      NOP               ;←BT 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
      ---
TRGET_T:                ;←BT 命令の分岐先
```

6.1.12 BT/S Branch if True with delay Slot 分岐命令
条件付き遅延分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
BT/S label	T＝1 のとき disp×2＋PC→PC、T＝0 のとき nop	10001101dddddddd	2/1	―	―	○	○

(1) **説明**

T ビットを参照する条件付き遅延分岐命令です。T＝1 のとき、次の命令を実行した後で分岐します。T＝0 のとき、次の命令を実行します。

分岐先は PC にディスプレースメントを加えたアドレスです。この際、アドレス計算に使用する PC は、本命令の 4 バイト後のアドレスです。8 ビットディスプレースメントは符号拡張後 2 倍しますので、分岐先との相対距離は－256 バイトから＋254 バイトの範囲になります。分岐先に届かないときは BRA 命令などとの組み合わせで対応する必要があります。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

分岐するときは 2 ステート、分岐しないときは 1 ステートになります。

(3) **動作内容**

```
     BTS(long d)    /* BTS disp */
  {
     long disp;
     unsigned long temp;

     temp=PC;
     if ((d&0x80)==0) disp=(0x000000FF & (long)d);
     else disp=(0xFFFFFF00 | (long)d);
     if (T==1) {
        PC=PC+(disp<<1);
        Delay_Slot(temp+2);
     }
     else PC+=2;
  }
```

(4) **使用例**

```
      SETT              ;常に T=1
      BF/S TRGET_F      ;T=1 のため分岐しません。
      NOP               ;
      BT/S TRGET_T      ;T=1 のため TRGET_T に分岐します。
      ADD R0,R1         ;分岐に先立ち実行します。
      NOP               ;←BT/S 命令で分岐先ｱﾄﾞﾚｽ計算に用いる PC の位置
      ---
  TRGET_T:              ;←BT/S 命令の分岐先
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.13 CLRMAC CLeaR MAC register システム制御命令

MAC レジスタのクリア

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
CLRMAC	0→MACH、MACL	0000000000101000	1	―	○	○	○

(1) 説明

MACH、MACL レジスタをクリアします。

(2) 動作内容

```
CLRMAC( )    /* CLRMAC */
{
     MACH=0;
     MACL=0;
     PC+=2;
}
```

(3) 使用例

```
     CLRMAC                ;MAC ﾚｼﾞｽﾀをｸﾘｱして初期化します。
     MAC.W  @R0+,@R1+      ;積和演算
     MAC.W  @R0+,@R1+      ;
```

6.1.14 CLRT CLeaR Tbit システム制御命令

T ビットのクリア

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
CLRT	0→T	0000000000001000	1	0	○	○	○

(1) **説明**

T ビットをクリアします。

(2) **動作内容**

```
CLRT( )    /* CLRT */
{
    T=0;
    PC+=2;
}
```

(3) **使用例**

```
    CLRT                        ;実行前 T=1
                                ;実行後 T=0
```

6.1.15 CMP/cond CoMPare conditionally 算術演算命令

比較

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
CMP/EQ Rm,Rn	Rn＝Rm のとき 1→T	0011nnnnmmmm0000	1	比較結果	○	○	○
CMP/GE Rm,Rn	有符号で Rn≧Rm のとき 1→T	0011nnnnmmmm0011	1	比較結果	○	○	○
CMP/GT Rm,Rn	有符号で Rn＞Rm のとき 1→T	0011nnnnmmmm0111	1	比較結果	○	○	○
CMP/HI Rm,Rn	無符号で Rn＞Rm のとき 1→T	0011nnnnmmmm0110	1	比較結果	○	○	○
CMP/HS Rm,Rn	無符号で Rn≧Rm のとき 1→T	0011nnnnmmmm0010	1	比較結果	○	○	○
CMP/PL Rn	Rn＞0 のとき 1→T	0100nnnn00010101	1	比較結果	○	○	○
CMP/PZ Rn	Rn≧0 のとき 1→T	0100nnnn00010001	1	比較結果	○	○	○
CMP/STR Rm,Rn	いずれかのﾊﾞｲﾄが等しいとき 1→T	0010nnnnmmmm1100	1	比較結果	○	○	○
CMP/EQ #imm,R0	R0＝imm のとき 1→T	10001000iiiiiiii	1	比較結果	○	○	○

(1) 説明

汎用レジスタ Rn と Rm とを比較し、その結果、指定された条件(cond)が成立していると T ビットをセットします。条件が不成立のときは T ビットをクリアします。Rn の内容は変化しません。8 条件が指定できます。PZ と PL の 2 条件については Rn と 0 との比較になります。

EQ の条件については符号拡張した 8 ビットのイミディエイトデータと R0 との比較も可能です。R0 の内容は変化しません。

ニーモニック	説明
CMP/EQ Rm,Rn	Rn＝Rm のとき T＝1
CMP/GE Rm,Rn	有符号値として Rn≧Rm のとき T＝1
CMP/GT Rm,Rn	有符号値として Rn＞Rm のとき T＝1
CMP/HI Rm,Rn	無符号値として Rn＞Rm のとき T＝1
CMP/HS Rm,Rn	無符号値として Rn≧Rm のとき T＝1
CMP/PL Rn	Rn＞0 のとき T＝1
CMP/PZ Rn	Rn≧0 のとき T＝1
CMP/STR Rm,Rn	いずれかのバイトが等しいとき T＝1
CMP/EQ #imm,R0	R0＝imm のとき T＝1

(2) **動作内容**

```
CMPEQ(long m, long n)   /* CMP_EQ Rm,Rn */
{
    if (R[n]==R[m]) T=1;
    else T=0;
    PC+=2;
}
CMPGE(long m, long n)   /* CMP_GE Rm,Rn */
{
    if ((long)R[n]>=(long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPGT(long m, long n)   /* CMP_GT Rm,Rn */
{
    if ((long)R[n]>(long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPHI(long m, long n)   /* CMP_HI Rm,Rn */
{
    if ((unsigned long)R[n]>(unsigned long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPHS(long m, long n)   /* CMP_HS Rm,Rn */
{
    if ((unsigned long)R[n]>=(unsigned long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPPL(long n)   /* CMP_PL Rn */
{
    if ((long)R[n]>0) T=1;
    else T=0;
    PC+=2;
}

CMPPZ(long n)   /* CMP_PZ Rn */
{
    if ((long)R[n]>=0) T=1;
```

```
      else T=0;
      PC+=2;
  }

  CMPSTR(long m, long n)    /* CMP_STR Rm,Rn */
  {
      unsigned long temp;
      long HH,HL,LH,LL;

      temp=R[n]^R[m];
      HH=(temp>>12)&0x000000FF;
      HL=(temp>>8)&0x000000FF;
      LH=(temp>>4)&0x000000FF;
      LL=temp&0x000000FF;
      HH=HH&&HL&&LH&&LL;
      if (HH==0) T=1;
      else T=0;
      PC+=2;
  }

  CMPIM(long i)    /* CMP_EQ #imm,R0 */
  {
      long imm;

      if ((i&0x80)==0) imm=(0x000000FF & (long i));
      else imm=(0xFFFFFF00 | (long i));
      if (R[0]==imm) T=1;
      else T=0;
      PC+=2;
  }
```

(3) **使用例**

```
  CMP/GE   R0,R1                    ;R0=H'7FFFFFFF,R1=H'80000000
  BT TRGET_T                        ;T=0 なので分岐しません。
  CMP/HS   R0,R1                    ;R0=H'7FFFFFFF,R1=H'80000000
  BT TRGET_T                        ;T=1 なので分岐します。
  CMP/STR  R2,R3                    ;R2="ABCD",R3="XYCZ"
  BT TRGET_T                        ;T=1 なので分岐します。
```

6.1.16 DIV0S DIVide(step0) as Signed 算術演算命令

符号付き除算の初期化

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
DIV0S Rm,Rn	Rn の MSB→Q、 Rm の MSB→M、M^Q→T	0010nnnnmmmm0111	1	計算結果	○	○	○

(1) **説明**

符号付き除算の初期設定をします。本命令に続けて 1 桁分の除算をする DIV1 命令などを組み合わせて、繰り返し除算を行い商を求めます。詳しくは DIV1 の説明を参照してください。

(2) **動作内容**

```
DIV0S(long m, long n)    /* DIV0S Rm,Rn */
{
    if ((R[n] & 0x80000000)==0) Q=0;
    else Q=1;
    if ((R[m] & 0x80000000)==0) M=0;
    else M=1;
    T=!(M==Q);
    PC+=2;
}
```

(3) **使用例**

DIV1 の使用例を参照してください。

6.1.17 DIV0U DIVide (step0) as Unsigned 算術演算命令

符号なし除算の初期化

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
DIV0U	0→M/Q/T	0000000000011001	1	0	○	○	○

(1) **説明**

符号なし除算の初期設定をします。本命令に続けて 1 桁分の除算をする DIV1 命令などを組み合わせて、繰り返し除算を行い商を求めます。詳しくは DIV1 の説明を参照してください。

(2) **動作内容**

```
DIV0U( )  /* DIV0U */
{
    M=Q=T=0;
    PC+=2;
}
```

(3) **使用例**

DIV1 の使用例を参照してください。

6.1.18 DIV1 DIVide 1 step 算術演算命令

除算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
DIV1 Rm,Rn	1 ステップ除算 (Rn÷Rm)	0011nnnnmmmm0100	1	計算結果	○	○	○

(1) **説明**

汎用レジスタ Rn の 32 ビットの内容(被除数)を Rm の内容(除数)で 1 桁分の除算(1 ステップ除算)を実行する命令です。本命令単独でまたは他の命令と組み合わせて繰り返し実行し商を求めます。この繰り返し中は、指定したレジスタと M、Q、T ビットを書き換えないでください。

1 ステップ除算とは、被除数を左に 1 ビットシフトし、それから除数を減算し、結果の正負によって商のビットを Q ビットに反映するという処理を実行します。

割り算で余りを求めるには、DIV1 命令を用いて商を求めたあと、

（被除数）－（除数）×（商）＝（余り）

として求めてください。

ゼロ除算とオーバフローの検出は用意していません。除算の前にゼロ除算とオーバフロー除算をチェックしてください。剰余の演算は用意していません。除数と求められた商との積を求めて、被除数から減算して剰余を求めてください。

最初に、DIV0S または DIV0U で初期設定します。DIV1 を除数のビット数分繰り返します。商が 17 ビット以上必要なとき、ROTCL を DIV1 の前に置きます。詳しい 除算のシーケンスは下記の使用例を参考にしてください。

(2) **動作内容**

```
DIV1(long m, long n)   /* DIV1 Rm,Rn */
{
    unsigned long tmp0;
    unsigned char  old_q, tmp1;

    old_q=Q;
    Q=(unsigned char)((0x80000000 & R[n])!=0);
    R[n]<<=1;
    R[n]|=(unsigned long)T;
    switch(old_q){
    case 0:switch(M){
       case 0:tmp0=R[n];
          R[n]-=R[m];
          tmp1=(R[n]>tmp0);
          switch(Q){
          case 0:Q=tmp1;
               break;
          case 1:Q=(unsigned char)(tmp1==0);
               break;
```

```
            }
            break;
        case 1:tmp0=R[n];
            R[n]+=R[m];
            tmp1=(R[n]<tmp0);
            switch(Q){
            case 0:Q=(unsigned char)(tmp1==0);
                 break;
            case 1:Q=tmp1;
                 break;
            }
            break;
        }
        break;
    case 1:switch(M){
        case 0:tmp0=R[n];
             R[n]+=R[m];
             tmp1=(R[n]<tmp0);
             switch(Q){
             case 0:Q=tmp1;
                   break;
             case 1:Q=(unsigned char)(tmp1==0);
                   break;
              }
              break;
        case 1:tmp0=R[n];
              R[n]-=R[m];
              tmp1=(R[n]>tmp0);
              switch(Q){
              case 0:Q=(unsigned char)(tmp1==0);
                   break;
              case 1:Q=tmp1;
                   break;
              }
              break;
        }
        break;
    }
    T=(Q==M);
    PC+=2;
}
```

(3) **使用例** 1

```
                                ;R1(32ﾋﾞｯﾄ)÷R0(16ﾋﾞｯﾄ)=R1(16ﾋﾞｯﾄ):符号なし
  SHLL16      R0                ;除数を上位 16ﾋﾞｯﾄ、下位 16ﾋﾞｯﾄを 0 に設定
  TST         R0,R0             ;ゼロ除算チェック
  BT          ZERO_DIV          ;
  CMP/HS      R0,R1             ;オーバフローチェック
  BT          OVER_DIV          ;
  DIV0U                         ;フラグの初期化
  .arepeat    16                ;
  DIV1        R0,R1             ;16 回繰り返し
  .aendr                        ;
  ROTCL       R1                ;
  EXTU.W      R1,R1             ;R1=商
```

(4) **使用例** 2

```
                                ;R1:R2(64ﾋﾞｯﾄ)÷R0(32ﾋﾞｯﾄ)=R2(32ﾋﾞｯﾄ):符号なし
  TST         R0,R0             ;ゼロ除算チェック
  BT          ZERO_DIV          ;
  CMP/HS      R0,R1             ;オーバフローチェック
  BT          OVER_DIV          ;
  DIV0U                         ;フラグの初期化
  .arepeat    32                ;
  ROTCL       R2                ;32 回繰り返し
  DIV1        R0,R1             ;
  .aendr                        ;
  ROTCL       R2                ;R2=商
```

(5) **使用例** 3

```
                                ;R1(16ﾋﾞｯﾄ)÷R0(16ﾋﾞｯﾄ)=R1(16ﾋﾞｯﾄ):符号付き
  SHLL16      R0                ;除数を上位 16 ビット、下位 16 ビットを 0 に設定
  EXTS.W      R1,R1             ;被除数は符号拡張して 32 ビット
  XOR         R2,R2             ;R2=0
  MOV         R1,R3             ;
  ROTCL       R3                ;
  SUBC        R2,R1             ;被除数が負のとき、-1 する。
  DIV0S       R0,R1             ;ﾌﾗｸﾞ の初期化
  .arepeat    16                ;
  DIV1        R0,R1             ;16 回繰り返し
  .aendr                        ;
  EXTS.W      R1,R1             ;
  ROTCL       R1                ;R1=商(1 の補数表現)
  ADDC        R2,R1             ;商の MSB が 1 のとき、+1 して 2 の補数表現に変換
  EXTS.W      R1,R1             ;R1=商(2 の補数表現)
```

(6) **使用例** 4

```
                                ;R2(32ﾋﾞｯﾄ)÷R0(32ﾋﾞｯﾄ)=R2(32ﾋﾞｯﾄ):符号付き
  MOV           R2,R3           ;
  ROTCL         R3              ;
  SUBC          R1,R1           ;被除数は符号拡張して 64ﾋﾞｯﾄ(R1:R2)
  XOR           R3,R3           ;R3=0
  SUBC          R3,R2           ;被除数が負のとき、-1 して 1 の補数表現に変換
  DIV0S         R0,R1           ;ﾌﾗｸﾞ の初期化
  .arepeat      32              ;
  ROTCL         R2              ;32 回繰り返し
  DIV1          R0,R1           ;
  .aendr                        ;
  ROTCL         R2              ;R2=商(1 の補数表現)
  ADDC          R3,R2           ;商の MSB が 1 のとき、+1 して 2 の補数表現に変換
                                ;R2=商(2 の補数表現)
```

6.1.19 DMULS.L Double-length MULtiply as Signed 算術演算命令

符号付き倍精度乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
DMULS.L Rm,Rn	符号付きで Rn×Rm→MACH、MACL	0011nnnnmmmm1101	2〜4	―	―	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm を 32 ビットで乗算し、結果の 64 ビットを MACH レジスタと MACL レジスタに格納します。演算は符号付き算術演算で行います。

(2) **動作内容**

```
DMULS(long m, long n) /* DMULS.L Rm,Rn */
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,temp1,temp2,temp3;
    long tempm,tempn,fnLmL;

    tempn=(long)R[n];
    tempm=(long)R[m];
    if (tempn<0) tempn=0-tempn;
    if (tempm<0) tempm=0-tempm;
    if ((long)(R[n]^R[m])<0) fnLmL=-1;
    else fnLmL=0;

    temp1=(unsigned long)tempn;
    temp2=(unsigned long)tempm;

    RnL=temp1&0x0000FFFF;
    RnH=(temp1>>16)&0x0000FFFF;
    RmL=temp2&0x0000FFFF;
    RmH=(temp2>>16)&0x0000FFFF;

    temp0=RmL*RnL;
    temp1=RmH*RnL;
    temp2=RmL*RnH;
    temp3=RmH*RnH;

    Res2=0
    Res1=temp1+temp2;
    if (Res1<temp1) Res2+=0x00010000;
```

```
    temp1=(Res1<<16)&0xFFFF0000;
    Res0=temp0+temp1;
    if (Res0<temp0) Res2++;

    Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

    if (fnLmL<0) {
        Res2=~Res2;
        if (Res0==0)
            Res2++;
        else
            Res0=(~Res0)+1;
    }

    MACH=Res2;
    MACL=Res0;
    PC+=2;
}
```

(3) **使用例**

```
DMULS.L  R0,R1      ;実行前 R0=H'FFFFFFFE,R1=H'00005555
                    ;実行後 MACH=H'FFFFFFFF,MACL=H'FFFF5556
STS      MACH,R0    ;演算結果(上位)を得る
STS      MACL,R0    ;演算結果(下位)を得る
```

6.1.20 DMULU.L Double-length MULtiply as Unsigned 算術演算命令

符号なし倍精度乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
DMULU.L Rm,Rn	符号なしで Rn×Rm→MACH、MACL	0011nnnnmmmm0101	2〜4	—	—	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm を 32 ビットで乗算し、結果の 64 ビットを MACH レジスタと MACL レジスタに格納します。演算は符号なし算術演算で行います。

(2) **動作内容**

```
DMULU(long m, long n)    /* DMULU.L Rm,Rn */
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,temp1,temp2,temp3;

    RnL=R[n]&0x0000FFFF;
    RnH=(R[n]>>16)&0x0000FFFF;

    RmL=R[m]&0x0000FFFF;
    RmH=(R[m]>>16)&0x0000FFFF;

    temp0=RmL*RnL;
    temp1=RmH*RnL;
    temp2=RmL*RnH;
    temp3=RmH*RnH;

    Res2=0
    Res1=temp1+temp2;
    if (Res1<temp1) Res2+=0x00010000;

    temp1=(Res1<<16)&0xFFFF0000;
    Res0=temp0+temp1;
    if (Res0<temp0) Res2++;

    Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

    MACH=Res2;
    MACL=Res0;
    PC+=2;
}
```

(3) **使用例**

```
DMULU.L  R0,R1       ;実行前 R0=H'FFFFFFFE,R1=H'00005555
                     ;実行後 MACH=H'00005554,MACL=H'FFFF5556
STS      MACH,R0     ;演算結果(上位)を得る
STS      MACL,R0     ;演算結果(下位)を得る
```

6.1.21 DT Decrement and Test 算術演算命令

デクリメントとテスト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
DT Rn	Rn-1→Rn、Rn が 0 のとき 1→T Rn が 0 以外のとき 0→T	0100nnnn00010000	1	比較結果	―	○	○

(1) **説明**

汎用レジスタ Rn の内容を 1 デクリメントして、結果を 0（ゼロ）と比較します。結果が 0 のとき T ビットを 1 にセットします。結果が 0 以外のとき、T ビットを 0 にセットします。

(2) **動作内容**

```
DT(long n)          /* DT Rn */
{
    R[n]--;
    if (R[n]==0) T=1;
    else T=0;
    PC+=2;
}
```

(3) **使用例**

```
      MOV #4,R5      ;ループ回数を設定します。
LOOP:
      ADD R0,R1      ;
      DT RS          ;R5 の値をデクリメントし、0 になったかどうか判定します。
      BF LOOP        ;T=0 なら LOOP へ分岐します(この例では 4 回ループします)。
```

6.1.22 EXTS EXTend as Signed 算術演算命令

符号拡張

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
EXTS.B Rm,Rn	Rm をﾊﾞｲﾄから符号拡張→Rn	0110nnnnmmmm1110	1	—	○	○	○
EXTS.W Rm,Rn	Rm をﾜｰﾄﾞから符号拡張→Rn	0110nnnnmmmm1111	1	—	○	○	○

(1) 説明

汎用レジスタ Rm の内容を符号拡張して、結果を Rn に格納します。
バイト指定のとき、Rn のビット 8 からビット 31 に Rm のビット 7 の内容を転送します。
ワード指定のとき、Rn のビット 16 からビット 31 に Rm のビット 15 の内容を転送します。

(2) 動作内容

```
EXTSB(long m, long n)      /* EXTS.B Rm,Rn */
{
    R[n]=R[m];
    if ((R[m]&0x00000080)==0) R[n]&=0x000000FF;
    else R[n]|=0xFFFFFF00;
    PC+=2;
}

EXTSW(long m, long n)      /* EXTS.W Rm,Rn */
{
    R[n]=R[m];
    if ((R[m]&0x00008000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}
```

(3) 使用例

```
EXTS.B   R0,R1    ;実行前 R0=H'00000080
                  ;実行後 R1=H'FFFFFF80
EXTS.W   R0,R1    ;実行前 R0=H'00008000
                  ;実行後 R1=H'FFFF8000
```

6.1.23 EXTU EXTend as Unsigned 算術演算命令

ゼロ拡張

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
EXTU.B Rm,Rn	Rm をﾊﾞｲﾄからｾﾞﾛ拡張→Rn	0110nnnnmmmm1100	1	—	○	○	○
EXTU.W Rm,Rn	Rm をﾜｰﾄﾞからｾﾞﾛ拡張→Rn	0110nnnnmmmm1101	1	—	○	○	○

(1) **説明**

汎用レジスタ Rm の内容をゼロ拡張して、結果を Rn に格納します。

バイト指定のとき、Rn のビット 8 からビット 31 に 0 を転送します。ワード指定のとき、Rn のビット 16 からビット 31 に 0 を転送します。

(2) **動作内容**

```
EXTUB(long m, long n)      /* EXTU.B Rm,Rn */
{
    R[n]=R[m];
    R[n]&=0x000000FF;
    PC+=2;
}

EXTUW(long m, long n)      /* EXTU.W Rm,Rn */
{
    R[n]=R[m];
    R[n]&=0x0000FFFF;
    PC+=2;
    }
```

(3) **使用例**

```
EXTU.B   R0,R1    ;実行前 R0=H'FFFFFF80
                  ;実行後 R1=H'00000080
EXTU.W   R0,R1    ;実行前 R0=H'FFFF8000
                  ;実行後 R1=H'00008000
```

6.1.24 JMP JuMP 分岐命令

無条件分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
JMP @Rm	Rm→PC	0100mmmm00101011	2	―	○	○	○

(1) **説明**

レジスタ間接で指定したアドレスへ無条件に遅延分岐します。分岐先は汎用レジスタ Rm の内容の 32 ビットデータで表されるアドレスです。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
JMP(long m)      /* JMP @Rm */
{
     unsigned long temp;

     temp=PC;
     PC=R[m]+4;
     Delay_Slot(temp+2);
}
```

(4) **使用例**

```
                MOV.L    JMP_TABLE,R0     ;R0=TRGET のｱﾄﾞﾚｽ
                JMP      @R0              ;TRGET へ分岐します。
                MOV      R0,R1            ;分岐に先立ち実行します。
                .align   4
JMP_TABLE:      .data.l  TRGET            ;ｼﾞｬﾝﾌﾟﾃｰﾌﾞﾙ
          ..........
TRGET:          ADD      #1,R1            ;←分岐先
```

【注】 遅延分岐においては、分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.25 JSR Jump to SubRoutine 分岐命令

サブルーチンプロシージャへの分岐 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
JSR @Rm	PC→PR、 Rm→PC	0100mmmm00001011	2	―	○	○	○

(1) **説明**

レジスタ間接で指定したアドレスのサブルーチンプロシージャへ遅延分岐します。PC の内容を PR に退避し、汎用レジスタ Rm の内容の 32 ビットデータで表されるアドレスへ分岐します。退避される PC は、本命令の 4 バイト後のアドレスです。RTS と組み合わせて、サブルーチンプロシージャコールに使用します。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
JSR(long m)/* JSR @Rm */
{
     PR=PC;
     PC=R[m]+4;
     Delay_Slot(PR+2);
}
```

(4) **使用例**

```
              MOV.L     JSR_TABLE,R0    ;R0=TRGET のｱﾄﾞﾚｽ
              JSR       @R0             ;TRGET へ分岐します。
              XOR       R1,R1           ;分岐に先立ち実行します。
              ADD       R0,R1           ;←ﾌﾟﾛｼｰｼﾞｬからの戻り先
              .......                   (PR の内容)です。
              .align    4
JSR_TABLE:    .data.l   TRGET           ;ｼﾞｬﾝﾌﾟﾃｰﾌﾞﾙ
TRGET:        NOP                       ;←ﾌﾟﾛｼｰｼﾞｬの入り口
              MOV       R2,R3           ;
              RTS                       ;上記 ADD 命令に戻ります。
              MOV       #70,R1          ;RTS に先立ち実行します。
```

【注】 遅延分岐においては、分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.26 LDC LoaD to Control register システム制御命令

コントロールレジスタへのロード 割り込み禁止命令

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
LDC Rm,SR	Rm→SR	0100mmmm00001110	1	LSB	○	○	○
LDC Rm,GBR	Rm→GBR	0100mmmm00011110	1	—	○	○	○
LDC Rm,VBR	Rm→VBR	0100mmmm00101110	1	—	○	○	○
LDC Rm,MOD	Rm→MOD	0100mmmm01011110	1	—	—	—	○
LDC Rm,RE	Rm→RE	0100mmmm01111110	1	—	—	—	○
LDC Rm,RS	Rm→RS	0100mmmm01101110	1	—	—	—	○
LDC.L @Rm+,SR	(Rm)→SR、Rm＋4→Rm	0100mmmm00000111	3	LSB	○	○	○
LDC.L @Rm+,GBR	(Rm)→GBR、Rm＋4→Rm	0100mmmm00010111	3	—	○	○	○
LDC.L @Rm+,VBR	(Rm)→VBR、Rm＋4→Rm	0100mmmm00100111	3	—	○	○	○
LDC.L @Rm+,MOD	(Rm)→MOD、Rm＋4→Rm	0100mmmm01010111	3	—	—	—	○
LDC.L @Rm+,RE	(Rm)→RE、Rm＋4→Rm	0100mmmm01110111	3	—	—	—	○
LDC.L @Rm+,RS	(Rm)→RS、Rm＋4→Rm	0100mmmm01100111	3	—	—	—	○

(1) **説明**

ソースオペランドをコントロールレジスタ SR、GBR、VBR、MOD、RE、RS に格納します。

(2) **注意**

本命令と直後の命令との間には、割り込みを受け付けません。(アドレスエラーは受け付けます。)

(3) **動作内容**

```
LDCSR(long m)       /* LDC Rm,SR */
{
    SR=R[m]&0x000003F3;
```

SH-1 CPU と SH-2 CPU の場合

```
    SR=R[m]&0x0FFF0FFF;
```

SH-DSP の場合

```
    PC+=2;
}

LDCGBR(long m)      /* LDC Rm,GBR */
{
    GBR=R[m];
```

```
    PC+=2;
}

LDCVBR(long m)      /* LDC Rm,VBR */
{
    VBR=R[m];
    PC+=2;
}
LDCMOD(long m)      /* LDC Rm,MOD */
{
    MOD=R[m];
    PC+=2;
}

LDCRE(long m)       /* LDC Rm,RE */
{
    RE=R[m];
    PC+=2;
}

LDCRS(long m)       /* LDC Rm,RS */
{
    RS=R[m];
    PC+=2;
}

LDCMSR(long m)      /* LDC.L @Rm+,SR */
{
    SR=Read_Long(R[m])&0x000003F3;
    SR=Read_Long(R[m])&0x0FFF0FFF;
    R[m]+=4;
    PC+=2;
```

SH-1 CPU と SH-2 CPU の場合

SH-DSP の場合

```
}

LDCMGBR(long m)    /* LDC.L @Rm+,GBR */
{
    GBR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMVBR(long m)    /* LDC.L @Rm+,VBR */
{
    VBR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
LDCMMOD(long m)    /*LDC.L @Rm+,MOD */
{
    MOD=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMRE(long m)     /*LDC.L @Rm+,RE */
 {
    RE=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMRS(long m)     /*LDC.L @Rm+,RS */
{
    RS=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
```

(4) **使用例**

```
LDC     R0,SR          ;実行前   R0=H'FFFFFFFF,SR=H'00000000
                       ;実行後   SR=H'0FFF0FFF*
LDC.L   @R15+,GBR      ;実行前   R15=H'10000000
                       ;実行後   R15=H'10000004,GBR=@H'10000000
```

【注】 * SH-DSP **の場合の実行結果になります。**

6.1.27 LDRE LoaD effective address to RE register システム制御命令

繰り返し終了レジスタへのロード

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
LDRE @(disp,PC)	disp×2 ＋PC→RE	10001110dddddddd	1	―	―	―	○

(1) **説明**

ソースオペランドの実効アドレス値を繰り返し終了レジスタ RE に格納します。実効アドレスは PC にディスプレースメントを加えたアドレスです。PC は、本命令の 4 バイト後のアドレスです。8 ビットディスプレースメントは符号拡張後 2 倍しますので、−256 バイトから＋254 バイトの範囲になります。

(2) **注意**

RE レジスタに指定する実効アドレス値は実際の繰り返し終了アドレスとは異なります。詳しくは、**表** 4.36 を参照してください。本命令を遅延分岐命令の直後に配置すると、PC は分岐先の “ 先頭アドレス＋2 ” になります。

(3) **動作内容**

```
LDRE(long d)    /* LDRE @(disp,PC) */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    RE=PC+(disp<<1);
    PC+=2;
}
```

(4) **使用例**

```
      LDRS STA    ; set repeat start address to RS.
      LDRE END    ; set repeat end address to RE.
      SETRC #32   ; repeat 32 times from inst.A to inst.C.
      inst.0      ;
STA:     inst.A   ;
         inst.B   ;
        ..........
END:     inst.C   ;
     inst.E       ;
       ..........
```

6.1.28 LDRS LoaD effective address to RS register システム制御命令

繰り返し開始レジスタへのロード

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
LDRS @(disp,PC)	disp×2 ＋PC→RS	10001100dddddddd	1	―	―	―	○

(1) **説明**

ソースオペランドの実効アドレス値を繰り返し開始レジスタ RS に格納します。実効アドレスは PC にディスプレースメントを加えたアドレスです。PC は、本命令の 4 バイト後のアドレスです。8 ビットディスプレースメントは符号拡張後 2 倍しますので、−256 バイトから＋254 バイトの範囲になります。

(2) **注意**

繰り返し（ループ）プログラムが 3 命令以下のときは、RS レジスタに指定する実効アドレス値は実際の繰り返し開始アドレスとは異なります。詳しくは、**表** 4.36 を参照してください。本命令を遅延分岐命令の直後に配置すると、PC は分岐先の “ 先頭アドレス＋2 ” になります。

(3) **動作内容**

```
LDRS(long d)    /* LDRS @(disp,PC) */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    RS=PC+(disp<<1);
    PC+=2;
}
```

(4) **使用例**

```
      LDRS STA                    ; set repeat start address to RS.
      LDRE END                    ; set repeat end address to RE.
      SETRC #32                   ; repeat 32 times from inst.A to inst.C.
      inst.0                      ;
STA:     inst.A                   ;
         inst.B                   ;
        ..........
END:     inst.C                   ;
     inst.D                      ;
       ..........
```

6.1.29 LDS LoaD to System register システム制御命令

システムレジスタへのロード 割り込み禁止命令

書式		動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
LDS	Rm,MACH	Rm→MACH	0100mmmm00001010	1	—	○	○	○
LDS	Rm,MACL	Rm→MACL	0100mmmm00011010	1	—	○	○	○
LDS	Rm,PR	Rm→PR	0100mmmm00101010	1	—	○	○	○
LDS	Rm,DSR	Rm→DSR	0100mmmm01101010	1	—	—	—	○
LDS	Rm,A0	Rm→A0	0100mmmm01111010	1	—	—	—	○
LDS	Rm,X0	Rm→X0	0100mmmm10001010	1	—	—	—	○
LDS	Rm,X1	Rm→X1	0100mmmm10011010	1	—	—	—	○
LDS	Rm,Y0	Rm→Y0	0100mmmm10101010	1	—	—	—	○
LDS	Rm,Y1	Rm→Y1	0100mmmm10111010	1	—	—	—	○
LDS.L	@Rm+,MACH	(Rm)→MACH、Rm＋4→Rm	0100mmmm00000110	1	—	○	○	○
LDS.L	@Rm+,MACL	(Rm)→MACL、Rm＋4→Rm	0100mmmm00010110	1	—	○	○	○
LDS.L	@Rm+,PR	(Rm)→PR、Rm＋4→Rm	0100mmmm00100110	1	—	○	○	○
LDS.L	@Rm+,DSR	(Rm)→DSR、Rm＋4→Rm	0100mmmm01100110	1	—	—	—	○
LDS.L	@Rm+,A0	(Rm)→A0、Rm＋4→Rm	0100mmmm01110110	1	—	—	—	○
LDS.L	@Rm+,X0	(Rm)→X0、Rm＋4→Rm	0100mmmm10000110	1	—	—	—	○
LDS.L	@Rm+,X1	(Rm)→X1、Rm＋4→Rm	0100mmmm10010110	1	—	—	—	○
LDS.L	@Rm+,Y0	(Rm)→Y0、Rm＋4→Rm	0100mmmm10100110	1	—	—	—	○
LDS.L	@Rm+,Y1	(Rm)→Y1、Rm＋4→Rm	0100mmmm10110110	1	—	—	—	○

(1) **説明**

ソースオペランドをシステムレジスタ MACH、MACL、PR に、または DSP レジスタ DSR、A0、X0、X1、Y0、Y1 に格納します。A0 をデスティネーションに指定した場合は、データの MSB が A0G にコピーされます。

(2) **注意**

本命令と直後の命令との間には、割り込みを受け付けません。(アドレスエラーは受け付けます。)
SH-1 CPU では MACH は下位 10 ビットが格納されます。
SH-2 CPU、SH-DSP では MACH は 32 ビットが格納されます。

(3) 動作内容

```
LDSMACH(long m)    /* LDS Rm,MACH */
{
    MACH=R[m];
    if((MACH&0x00000200)==0)MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    PC+=2;
}

LDSMACL(long m)    /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}

LDSPR(long m)      /* LDS Rm,PR */
{
    PR=R[m];
    PC+=2;
}

LDSDSR(long m)     /* LDS Rm,DSR */
{
    DSR=R[m]&0x0000000F;
    PC+=2;
}

LDSA0(long m)      /* LDS Rm,A0 */
{
    A0=R[m];
    if((A0&0x80000000)==0)A0G=0x00;
    else A0G=0xFF;
    PC+=2;
}

LDSX0(long m)      /* LDS Rm,X0 */
{
    X0=R[m];
    PC+=2;
}

LDSX1(long m)      /* LDS Rm,X1 */
{
    X1=R[m];
```

SH-1 CPU の場合（この 2 行は、SH-2 CPU と SH-DSP では不要です。）

```
    PC+=2;
}

LDSY0(long m)      /* LDS Rm,Y0 */
{
    Y0=R[m];
    PC+=2;
}

LDSY1(long m)      /* LDS Rm,Y1 */
{
    Y1=R[m];
    PC+=2;
}

LDSMMACH(long m)   /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
    if((MACH&0x00000200)==0)MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    R[m]+=4;
    PC+=2;
}

LDSMMACL(long m)   /* LDS.L @Rm+,MACL */
{
    MACL=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMPR(long m)     /* LDS.L @Rm+,PR */
{
    PR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMDSR(long m)    /* LDS.L @Rm+,DSR */
{
    DSR=Read_Long(R[m])&0x0000000F;
    R[m]+=4;
    PC+=2;
}
```

SH-1 CPU の場合（この 2 行は、SH-2 CPU と SH-DSP では不要です。）

```
LDSMA0(long m)     /* LDS.L @Rm+,A0 */
{
    A0=Read_Long(R[m]);
    if((A0&0x80000000)==0)A0G=0x00;
    else A0G=0xFF;
    R[m]+=4;
    PC+=2;
}

LDSMX0(long m)     /* LDS.L @Rm+,X0 */
{
    X0=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
LDSMX1(long m)     /* LDS.L @Rm+,X1 */
{
    X1=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMY0(long m)     /* LDS.L @Rm+,Y0 */
{
    Y0=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMY1(long m)     /* LDS.L @Rm+,Y1 */
{
    Y1=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
```

(4) **使用例**

```
LDS       R0,PR       ;実行前   R0=H'12345678,PR=H'00000000
                      ;実行後   PR=H'12345678
LDS.L     @R15+,MACL  ;実行前   R15=H'10000000
                      ;実行後   R15=H'10000004,MACL=@H'10000000
```

6.1.30 MAC.L Multiply and ACcumulate Long 算術演算命令

倍精度積和演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MAC.L @Rm+,@Rn+	符号付きで (Rn)×(Rm)＋MAC→MAC	0000nnnnmmmm1111	3/(2〜4)	―	―	○	○

(1) **説明**

汎用レジスタ Rm と Rn の内容をアドレスとする 32 ビットオペランドを符号付きで乗算し、結果の 64 ビットと MAC レジスタの内容とを加算し、結果を MAC レジスタに格納します。各オペランドを読み出すごとにそれぞれ、Rm を＋4、Rn を＋4 します。

S ビットが 0 のときは、連結した MACH、MACL レジスタに結果の 64 ビットを格納します。

S ビットが 1 のときは、MAC レジスタとの加算は LSB から 48 番目のビットで飽和演算になります。飽和演算では、MAC レジスタの下位 48 ビットのみが有効となり結果の範囲を H'FFFF800000000000（最小値）から H'00007FFFFFFFFFFF（最大値）までに制限します。

(2) **動作内容**

```
MACL(long m, long n)  /* MAC.L @Rm+,@Rn+ */
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,temp1,temp2,temp3;
    long tempm,tempn,fnLmL;

    tempn=(long)Read_Long(R[n]);
    R[n]+=4;
    tempm=(long)Read_Long(R[m]);
    R[m]+=4;

    if ((long)(tempn^tempm)<0) fnLmL=-1;
    else fnLmL=0;
    if (tempn<0) tempn=0-tempn;
    if (tempm<0) tempm=0-tempm;
    temp1=(unsigned long)tempn;
    temp2=(unsigned long)tempm;

    RnL=temp1&0x0000FFFF;
    RnH=(temp1>>16)&0x0000FFFF;
    RmL=temp2&0x0000FFFF;
    RmH=(temp2>>16)&0x0000FFFF;

    temp0=RmL*RnL;
    temp1=RmH*RnL;
```

```
    temp2=RmL*RnH;
    temp3=RmH*RnH;

    Res2=0;

Res1=temp1+temp2;
if (Res1<temp1) Res2+=0x00010000;

temp1=(Res1<<16)&0xFFFF0000;
Res0=temp0+temp1;
if (Res0<temp0) Res2++;

Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

if(fnLm<0){
    Res2=~Res2;
    if (Res0==0) Res2++;
    else Res0=(~Res0)+1;
}
if(S==1){
    Res0=MACL+Res0;
    if (MACL>Res0) Res2++;
    Res2+=(MACH&0x0000FFFF);

    if(((long)Res2<0)&&(Res2<0xFFFF8000)){
        Res2=0x00008000;
        Res0=0x00000000;
    }
    if(((long)Res2>0)&&(Res2>0x00007FFF)){
        Res2=0x00007FFF;
        Res0=0xFFFFFFFF;
    };

    MACH=Res2;
    MACL=Res0;
}
else {
    Res0=MACL+Res0;
    if (MACL>Res0) Res2++;
    Res2+=MACH;

    MACH=Res2;
    MACL=Res0;
}
PC+=2;
}
```

(3) **使用例**

```
        MOVA      TBLM,R0       ;ﾃｰﾌﾞﾙのｱﾄﾞﾚｽを得る
        MOV       R0,R1         ;
        MOVA      TBLN,R0       ;ﾃｰﾌﾞﾙのｱﾄﾞﾚｽを得る
        CLRMAC                  ;MAC ﾚｼﾞｽﾀの初期化
        MAC.L     @R0+,@R1+     ;
        MAC.L     @R0+,@R1+     ;
        STS       MACL,R0       ;結果を R0 に得る
    ............
        .align    2             ;
TBLM    .data.l   H'1234ABCD    ;
        .data.l   H'5678EF01    ;
TBLN    .data.l   H'0123ABCD    ;
        .data.l   H'4567DEF0    ;
```

6.1.31 MAC.W Multiply and ACcumulate Word 算術演算命令

積和演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MAC.W @Rm+,@Rn+	**符号付きで** (Rn)×(Rm)＋MAC→MAC	0100nnnnmmmm1111	3/(2)	―	―	○	○
MAC @Rm+,@Rn+					○	○	○

(1) **説明**

汎用レジスタ Rm と Rn の内容をアドレスとする 16 ビットオペランドを符号付きで乗算し、結果の 32 ビットと MAC レジスタの内容とを加算し、結果を MAC レジスタに格納します。各オペランドを読み出すごとにそれぞれ、Rm を＋2、Rn を＋2 します。

S ビットが 0 のとき、16×16＋64→64 ビットの積和演算となり、連結した MACH、MACL レジスタに結果の 64 ビットを格納します。

S ビットが 1 のとき、16×16＋32→32 ビットの積和演算となり、MAC レジスタとの加算は飽和演算になります。飽和演算では、MACL レジスタのみが有効となり結果の範囲を H'80000000（最小値）から H'7FFFFFFF（最大値）までに制限します。オーバフローが発生すると、MACH レジスタの LSB を 1 にセットします。結果が負の方向にオーバフローしたときは H'80000000（最小値）を、正の方向にオーバフローしたときは H'7FFFFFFF（最大値）を、MACL レジスタに格納します。

(2) **注意**

S ビットが 0 のとき SH-2 CPU と SH-DSP では 16×16＋64→64 ビットの積和演算を、SH-1 CPU では 16×16＋42→42 ビットの積和演算を行います。

(3) **動作内容**

```
MACW(long m, long n)  /* MAC.W @Rm+,@Rn+ */
{
    long tempm,tempn,dest,src,ans;
    unsigned long templ;
    tempn=(long)Read_Word(R[n]);
    R[n]+=2;
    tempm=(long)Read_Word(R[m]);
    R[m]+=2;
    templ=MACL;
    tempm=((long)(short)tempn*(long)(short)tempm);
    if ((long)MACL>=0) dest=0;
    else dest=1;
    if ((long)tempm>=0) {
        src=0;
        tempn=0;

    }
    else {
```

```
        src=1;
        tempn=0xFFFFFFFF;
    }
    src+=dest;
    MACL+=tempm;
    if ((long)MACL>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (S==1) {
        if (ans==1) {
        if(src==0)||(src==2)
        MACH|=0x00000001;

            if (src==0) MACL=0x7FFFFFFF;
            if (src==2) MACL=0x80000000;
        }
    }
else {
        MACH+=tempn;
        if (templ>MACL) MACH+=1;
        if((MACH&0x00000200)==0)
        MACH&=0x000003FF;
        else MACH|=0xFFFFFC00;
    }
    PC+=2;
}
```

SH-1 CPU の場合（この 2 行は、SH-2 CPU と SH-DSP では不要です。）

SH-1 CPU の場合（この 2 行は、SH-2 CPU と SH-DSP では不要です。）

(4) **使用例**

```
        MOVA    TBLM,R0     ;ﾃｰﾌﾞﾙのｱﾄﾞﾚｽを得る
        MOV     R0,R1       ;
        MOVA    TBLN,R0     ;ﾃｰﾌﾞﾙのｱﾄﾞﾚｽを得る
        CLRMAC              ;MAC ﾚｼﾞｽﾀの初期化
        MAC.W   @R0+,@R1+   ;
        MAC.W   @R0+,@R1+   ;
        STS     MACL,R0     ;結果を R0 に得る
    ............
        .align  2           ;
TBLM    .data.w H'1234      ;
        .data.w H'5678      ;
TBLN    .data.w H'0123      ;
        .data.w H'4567      ;
```

6.1.32 MOV MOVe data データ転送命令

データ転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
MOV Rm,Rn	Rm→Rn	0110nnnnmmmm0011	1	—	○	○	○
MOV.B Rm,@Rn	Rm→(Rn)	0010nnnnmmmm0000	1	—	○	○	○
MOV.W Rm,@Rn	Rm→(Rn)	0010nnnnmmmm0001	1	—	○	○	○
MOV.L Rm,@Rn	Rm→(Rn)	0010nnnnmmmm0010	1	—	○	○	○
MOV.B @Rm,Rn	(Rm)→**符号拡張**→Rn	0110nnnnmmmm0000	1	—	○	○	○
MOV.W @Rm,Rn	(Rm)→**符号拡張**→Rn	0110nnnnmmmm0001	1	—	○	○	○
MOV.L @Rm,Rn	(Rm)→Rn	0110nnnnmmmm0010	1	—	○	○	○
MOV.B Rm,@-Rn	Rn－1→Rn、Rm→(Rn)	0010nnnnmmmm0100	1	—	○	○	○
MOV.W Rm,@-Rn	Rn－2→Rn、Rm→(Rn)	0010nnnnmmmm0101	1	—	○	○	○
MOV.L Rm,@-Rn	Rn－4→Rn、Rm→(Rn)	0010nnnnmmmm0110	1	—	○	○	○
MOV.B @Rm+,Rn	(Rm)→**符号拡張**→Rn、Rm＋1→Rm	0110nnnnmmmm0100	1	—	○	○	○
MOV.W @Rm+,Rn	(Rm)→**符号拡張**→Rn、Rm＋2→Rm	0110nnnnmmmm0101	1	—	○	○	○
MOV.L @Rm+,Rn	(Rm)→Rn、Rm＋4→Rm	0110nnnnmmmm0110	1	—	○	○	○
MOV.B Rm,@(R0,Rn)	Rm→(R0＋Rn)	0000nnnnmmmm0100	1	—	○	○	○
MOV.W Rm,@(R0,Rn)	Rm→(R0＋Rn)	0000nnnnmmmm0101	1	—	○	○	○
MOV.L Rm,@(R0,Rn)	Rm→(R0＋Rn)	0000nnnnmmmm0110	1	—	○	○	○
MOV.B @(R0,Rm),Rn	(R0＋Rm) →**符号拡張**→Rn	0000nnnnmmmm1100	1	—	○	○	○
MOV.W @(R0,Rm),Rn	(R0＋Rm)→**符号拡張**→Rn	0000nnnnmmmm1101	1	—	○	○	○
MOV.L @(R0,Rm),Rn	(R0＋Rm)→Rn	0000nnnnmmmm1110	1	—	○	○	○

(1) **説明**

ソースオペランドをデスティネーションへ転送します。オペランドがメモリのときは転送するデータサイズをバイト／ワード／ロングワードの範囲で指定できます。ソースオペランドがメモリのときは、ロードされたデータをロングワードに符号拡張後レジスタに格納します。

(2) **動作内容**

```
MOV(long m, long n)       /* MOV Rm,Rn */
{
     R[n]=R[m];
     PC+=2;
}

MOVBS(long m, long n)     /* MOV.B Rm,@Rn */
{
    Write_Byte(R[n],R[m]);
    PC+=2;
```

```
}

MOVWS(long m, long n)     /* MOV.W Rm,@Rn */
{
   Write_Word(R[n],R[m]);
   PC+=2;
}

MOVLS(long m, long n)     /* MOV.L Rm,@Rn */
{
   Write_Long(R[n],R[m]);
   PC+=2;
}

MOVBL(long m, long n)     /* MOV.B @Rm,Rn */
{
   R[n]=(long)Read_Byte(R[m]);
   if ((R[n]&0x80)==0) R[n]&=0x000000FF;
   else R[n]|=0xFFFFFF00;
   PC+=2;
}

MOVWL(long m, long n)     /* MOV.W @Rm,Rn */
{
   R[n]=(long)Read_Word(R[m]);
   if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
   else R[n]|=0xFFFF0000;
   PC+=2;
}

MOVLL(long m, long n)     /* MOV.L @Rm,Rn */
{
   R[n]=Read_Long(R[m]);
   PC+=2;
}

MOVBM(long m, long n)     /* MOV.B Rm,@-Rn */
{
   Write_Byte(R[n]-1,R[m]);
   R[n]-=1;
   PC+=2;
}

MOVWM(long m, long n)     /* MOV.W Rm,@-Rn */
{
   Write_Word(R[n]-2,R[m]);
```

```
    R[n]-=2;
    PC+=2;
}

MOVLM(long m, long n)     /* MOV.L Rm,@-Rn */
{
    Write_Long(R[n]-4,R[m]);
    R[n]-=4;
    PC+=2;
}

MOVBP(long m, long n)     /* MOV.B @Rm+,Rn */
{
    R[n]=(long)Read_Byte(R[m]);
    if ((R[n]&0x80)==0) R[n]&=0x000000FF;
    else R[n]|=0xFFFFFF00;
    if (n!=m) R[m]+=1;
    PC+=2;
}

MOVWP(long m, long n)     /* MOV.W @Rm+,Rn */
{
    R[n]=(long)Read_Word(R[m]);
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    if (n!=m) R[m]+=2;
    PC+=2;
}

MOVLP(long m, long n)     /* MOV.L @Rm+,Rn */
{
    R[n]=Read_Long(R[m]);
    if (n!=m) R[m]+=4;
    PC+=2;
}

MOVBS0(long m, long n)    /* MOV.B Rm,@(R0,Rn) */
{
    Write_Byte(R[n]+R[0],R[m]);
    PC+=2;
}

MOVWS0(long m, long n)    /* MOV.W Rm,@(R0,Rn) */
{
    Write_Word(R[n]+R[0],R[m]);
    PC+=2;
```

```
}

MOVLS0(long m, long n)    /* MOV.L Rm,@(R0,Rn) */
{
    Write_Long(R[n]+R[0],R[m]);
    PC+=2;
}

MOVBL0(long m, long n)    /* MOV.B @(R0,Rm),Rn */
{
    R[n]=(long)Read_Byte(R[m]+R[0]);
    if ((R[n]&0x80)==0) R[n]&=0x000000FF;
    else R[n]|=0xFFFFFF00;
    PC+=2;
}

MOVWL0(long m, long n)    /* MOV.W @(R0,Rm),Rn */
{
    R[n]=(long)Read_Word(R[m]+R[0]);
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}

MOVLL0(long m, long n)    /* MOV.L @(R0,Rm),Rn */
{
    R[n]=Read_Long(R[m]+R[0]);
    PC+=2;
}
```

(3) **使用例**

```
MOV     R0,R1           ;実行前   R0=H'FFFFFFFF,R1=H'00000000
                        ;実行後   R1=H'FFFFFFFF
MOV.W   R0,@R1          ;実行前   R0=H'FFFF7F80
                        ;実行後   @R1=H'7F80
MOV.B   @R0,R1          ;実行前   @R0=H'80,R1=H'00000000
                        ;実行後   R1=H'FFFFFF80
MOV.W   R0,@-R1         ;実行前   R0=H'AAAAAAAA,R1=H'FFFF7F80
                        ;実行後   R1=H'FFFF7F7E,@R1=H'AAAA
MOV.L   @R0+,R1         ;実行前   R0=H'12345670
                        ;実行後   R0=H'12345674,R1=@H'12345670
MOV.B   R1,@(R0,R2)     ;実行前   R2=H'00000004,R0=H'10000000
                        ;実行後   R1=@H'10000004
MOV.W   @(R0,R2),R1     ;実行前   R2=H'00000004,R0=H'10000000
                        ;実行後   R1=@H'10000004
```

6.1.33 MOV MOVe immediate data データ転送命令

イミディエイトデータの転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
MOV #imm,Rn	imm→符号拡張→Rn	1110nnnniiiiiiii	1	—	○	○	○
MOV.W @(disp,PC),Rn	(disp×2+PC)→符号拡張→Rn	1001nnnndddddddd	1	—	○	○	○
MOV.L @(disp,PC),Rn	(disp×4+PC)→Rn	1101nnnndddddddd	1	—	○	○	○

(1) **説明**

ロングワードに符号拡張したイミディエイトデータを汎用レジスタ Rn に格納します。データがワードまたはロングワードのときは、PC にディスプレースメントを加えたアドレスに格納されたテーブル内のデータを参照します。

データがワードのとき、8 ビットディスプレースメントはゼロ拡張後 2 倍しますので、テーブルとの相対距離は PC+510 バイトまでの範囲になります。PC は本命令の 2 命令後の先頭アドレスです。

データがロングワードのとき、8 ビットディスプレースメントはゼロ拡張後 4 倍しますので、オペランドとの相対距離は PC+1020 バイトまでの範囲になります。PC は本命令の 2 命令後の先頭アドレスですが、下位 2 ビットを B'00 に補正した値をアドレス計算に使用します。

(2) **注意**

テーブルはモジュール後端あるいは無条件分岐命令の 1 命令後への配置が最適です。510 バイト／1020 バイト以内に無条件分岐命令がないなどの理由で最適配置が不可能なときは、BRA 命令でテーブルを飛び超す対策が必要です。本命令を遅延分岐命令の直後に配置すると、PC は分岐先の"先頭アドレス+2"になります。

(3) **動作内容**

```
MOVI(long i, long n)  /* MOV #imm,Rn */
{
    if ((i&0x80)==0) R[n]=(0x000000FF & (long)i);
    else R[n]=(0xFFFFFF00 | (long)i);
    PC+=2;
}

MOVWI(long d, long n)     /* MOV.W @(disp,PC),Rn */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[n]=(long)Read_Word(PC+(disp<<1));
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
```

```
}

MOVLI(long d, long n)     /* MOV.L @(disp,PC),Rn */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[n]=Read_Long((PC&0xFFFFFFFC)+(disp<<2));
    PC+=2;
}
```

(4) **使用例**

```
ｱﾄﾞﾚｽ
1000         MOV       #H'80,R1     ;R1=H'FFFFFF80
1002         MOV.W     IMM,R2       ;R2=H'FFFF9ABC  IMM は@(H'08,PC)の
                                      意味
1004         ADD       #-1,R0       ;
1006         TST       R0,R0        ;←MOV.W 命令でｱﾄﾞﾚｽ計算に用いる PC の位置
1008         MOVT      R13          ;
100A         BRA       NEXT         ;遅延分岐命令
100C         MOV.L     @(1,PC),R3   ;R3=H'12345678
100E  IMM    .data.w   H'9ABC       ;
1010         .data.w   H'1234       ;
1012  NEXT   JMP       @R3          ;BRA の分岐先
1014         CMP/EQ    #0,R0        ;←MOV.L 命令でｱﾄﾞﾚｽ計算に用いる PC の位置
             .align    4            ;
1018         .data.l   H'12345678   ;
```

6.1.34 MOV MOVe peripheral data データ転送命令

周辺モジュールデータの転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
MOV.B @(disp,GBR),R0	(disp＋GBR)→符号拡張→R0	11000100dddddddd	1	―	○	○	○
MOV.W @(disp,GBR), R0	(disp×2＋GBR)→符号拡張→R0	11000101dddddddd	1	―	○	○	○
MOV.L @(disp,GBR),R0	(disp×4＋GBR)→R0	11000110dddddddd	1	―	○	○	○
MOV.B R0,@(disp,GBR)	R0→(disp＋GBR)	11000000dddddddd	1	―	○	○	○
MOV.W R0,@(disp,GBR)	R0→(disp×2＋GBR)	11000001dddddddd	1	―	○	○	○
MOV.L R0,@(disp,GBR)	R0→(disp×4＋GBR)	11000010dddddddd	1	―	○	○	○

(1) **説明**

ソースオペランドをデスティネーションへ転送します。内蔵周辺モジュール領域内のデータアクセスに最適です。データサイズをバイト、ワード、またはロングワードの範囲で指定できますが、レジスタが R0 固定になります。GBR には、内蔵周辺モジュールのベースアドレスを設定します。

内蔵周辺モジュールのデータがバイトサイズのとき 8 ビットディスプレースメントはゼロ拡張するだけですので、＋255 バイトまでの範囲が指定できます。ワードサイズのとき 8 ビットディスプレースメントはゼロ拡張後 2 倍しますので、＋510 バイトまでの範囲が指定できます。ロングワードサイズのとき 8 ビットディスプレースメントはゼロ拡張後 4 倍しますので、＋1020 バイトまでの範囲が指定できます。メモリオペランドに届かないときは GBR を汎用レジスタに転送したあと、前述の@（R0,Rn）モードを使う必要があります。

ソースオペランドがメモリのときは、ロードされたデータをロングワードに符号拡張後レジスタへ格納します。

(2) **注意**

ロードするときデスティネーションレジスタが R0 固定です。したがって、直後の命令で R0 を参照しようとしてもロード命令の実行完了まで待たされます。これは命令の順序を変えることによって最適化が可能です。

```
MOV.B   @(12,GBR),R0          MOV.B   @(12,GBR),R0
AND     #80,R0          →     ADD     #20,R1
ADD     #20,R1                AND     #80,R0
```

(3) **動作内容**

```
MOVBLG(long d)     /* MOV.B @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Byte(GBR+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}

MOVWLG(long d)     /* MOV.W @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Word(GBR+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLLG(long d)     /* MOV.L @(disp,GBR),R0 */
{
    long disp;
```

```
    disp=(0x000000FF & (long)d);
    R[0]=Read_Long(GBR+(disp<<2));
    PC+=2;
}
MOVBSG(long d)     /* MOV.B R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Byte(GBR+disp,R[0]);
    PC+=2;
}

MOVWSG(long d)     /* MOV.W R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Word(GBR+(disp<<1),R[0]);
    PC+=2;
}

MOVLSG(long d)     /* MOV.L R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Long(GBR+(disp<<2),R[0]);
    PC+=2;
}
```

(4) **使用例**

```
MOV.L   @(2,GBR),R0    ;実行前   @(GBR+8)=H'12345670
                       ;実行後   R0=H'12345670
MOV.B   R0,@(1,GBR)    ;実行前   R0=H'FFFF7F80
                       ;実行後   @(GBR+1)=H'80
```

6.1.35 MOV MOVe structure data データ転送命令

構造体データの転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
MOV.B R0,@(disp,Rn)	R0→(disp＋Rn)	10000000nnnndddd	1	―	○	○	○
MOV.W R0,@(disp,Rn)	R0→(disp×2＋Rn)	10000001nnnndddd	1	―	○	○	○
MOV.L Rm,@(disp,Rn)	Rm→(disp×4＋Rn)	0001nnnnmmmmdddd	1	―	○	○	○
MOV.B @(disp,Rm),R0	(disp＋Rm)→**符号拡張**→R0	10000100mmmmdddd	1	―	○	○	○
MOV.W @(disp,Rm),R0	(disp×2＋Rm)→**符号拡張**→R0	10000101mmmmdddd	1	―	○	○	○
MOV.L @(disp,Rm),Rn	(disp×4＋Rm)→Rn	0101nnnnmmmmdddd	1	―	○	○	○

(1) **説明**

ソースオペランドをデスティネーションへ転送します。構造体、スタック内のデータアクセスに最適です。データサイズをバイト、ワード、またはロングワードの範囲で指定できますが、バイトまたはワードのときはレジスタが R0 固定になります。

データがバイトサイズのとき 4 ビットディスプレースメントはゼロ拡張するだけですので、＋15 バイトまでの範囲が指定できます。ワードサイズのとき 4 ビットディスプレースメントはゼロ拡張後 2 倍しますので、＋30 バイトまでの範囲が指定できます。ロングワードサイズのとき 4 ビットディスプレースメントはゼロ拡張後 4 倍しますので、＋60 バイトまでの範囲が指定できます。メモリオペランドに届かないときは前述の@（R0,Rn）モードを使う必要があります。

ソースオペランドがメモリのときは、ロードされたデータをロングワードに符号拡張後レジスタへ格納します。

(2) **注意**

バイト／ワードデータをロードするときデスティネーションレジスタが R0 固定です。したがって、直後の命令で R0 を参照しようとしてもロード命令の実行完了まで待たされます。これは命令の順序を変えることによって最適化が可能です。

```
MOV.B   @(2,R1),R0          MOV.B   @(2,R1),R0
AND     #80,R0      ---->   ADD     #20,R1
ADD     #20,R1      ---->   AND     #80,R0
```

(3) 動作内容

```
MOVBS4(long d, long n)  /* MOV.B R0,@(disp,Rn) */
{
    long disp;

    disp=(0x0000000F & (long)d);
    Write_Byte(R[n]+disp,R[0]);
    PC+=2;
}
MOVWS4(long d, long n)             /* MOV.W R0,@(disp,Rn) */
{
    long disp;

    disp=(0x0000000F & (long)d);
    Write_Word(R[n]+(disp<<1),R[0]);
    PC+=2;
}

MOVLS4(long m, long d, long n)     /* MOV.L Rm,@(disp,Rn) */
{
    long disp;

    disp=(0x0000000F & (long)d);
    Write_Long(R[n]+(disp<<2),R[m]);
    PC+=2;
}

MOVBL4(long m, long d)  /* MOV.B @(disp,Rm),R0 */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[0]=Read_Byte(R[m]+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}

MOVWL4(long m, long d)  /* MOV.W @(disp,Rm),R0 */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[0]=Read_Word(R[m]+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
```

```
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLL4(long m, long d, long n)     /* MOV.L @(disp,Rm),Rn */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[n]=Read_Long(R[m]+(disp<<2));
    PC+=2;
}
```

(4) **使用例**

```
MOV.L   @(2,R0),R1      ;実行前   @(R0+8)=H'12345670
                        ;実行後   R1=H'12345670
MOV.L   R0,@(H'F,R1)    ;実行前   R0=H'FFFF7F80
                        ;実行後   @(R1+60)=H'FFFF7F80
```

6.1.36 MOVA MOVe effective Address データ転送命令

実効アドレスの転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MOVA @(disp,PC),R0	disp×4＋PC→R0	11000111dddddddd	1	―	○	○	○

(1) **説明**

汎用レジスタ R0 にソースオペランドの実効アドレスを格納します。8 ビットディスプレースメントはゼロ拡張後 4 倍しますので、オペランドとの相対距離は PC＋1020 バイトまでの範囲になります。PC は本命令の 4 バイト後のアドレスですが、下位 2 ビットを B'00 に補正した値をアドレス計算に使用します。

(2) **注意**

本命令が遅延分岐命令の直後に配置されているとき、PC は分岐先の"先頭アドレス＋2"になります。

(3) **動作内容**

```
MOVA(long d)       /* MOVA @(disp,PC),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(PC&0xFFFFFFFC)+(disp<<2);
    PC+=2;
}
```

(4) **使用例**

```
ｱﾄﾞﾚｽ       .org    H'1006
1006        MOVA    STR,R0      ;STR のｱﾄﾞﾚｽ→R0
1008        MOV.B   @R0,R1      ;R1="X" ←PC 下位 2 ﾋﾞｯﾄ補正後の位置
100A        ADD     R4,R5       ;←MOVA 命令のｱﾄﾞﾚｽ計算時、PC の本来の位置
            .align  4
100C  STR:.sdata    "XYZP12"
    ..........
2002        BRA     TRGET       ;遅延分岐命令
2004        MOVA    @(0,PC),R0  ;TRGET のｱﾄﾞﾚｽ+2→R0
2006        NOP     ;
```

6.1.37 MOVT MOVe Tbit データ転送命令

T ビットの転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MOVT Rn	T→Rn	0000nnnn00101001	1	―	○	○	○

(1) **説明**

T ビットを汎用レジスタ Rn に格納します。T＝1 のとき Rn＝1、T＝0 のとき Rn＝0 になります。

(2) **動作内容**

```
MOVT(long n)        /* MOVT Rn */
{
    R[n]=(0x00000001 & SR);
    PC+=2;
}
```

(3) **使用例**

```
XOR       R2,R2    ;R2=0
CMP/PZ    R2       ;T=1
MOVT      R0       ;R0=1
CLRT               ;T=0
MOVT      R1       ;R1=0
```

6.1.38 MUL.L MULtiply Long 算術演算命令

倍精度乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MUL.L Rm,Rn	Rn×Rm→MACL	0000nnnnmmmm0111	2〜4	—	—	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm を 32 ビットで乗算し、結果の下位側 32 ビットを MACL レジスタに格納します。MACH の内容は変化しません。

(2) **動作内容**

```
MUL.L (long m, long n)  /* MUL.L Rm,Rn */
{
    MACL=R[n]*R[m];
    PC+=2;
}
```

(3) **使用例**

```
MUL.L   R0,R1      ;実行前   R0=H'FFFFFFFE,R1=H'00005555
                   ;実行後   MACL=H'FFFF5556
STS     MACL,R0    ;演算結果を得る
```

6.1.39 MULS.W MULtiply as Signed Word 算術演算命令

符号付き乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MULS.W Rm,Rn MULS Rm,Rn	符号付きで Rn×Rm→MACL	0010nnnnmmmm1111	1〜3	―	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm を 16 ビットで乗算し、結果の 32 ビットを MACL レジスタに格納します。演算は符号付き算術演算で行います。MACH の内容は変化しません。

(2) **動作内容**

```
MULS(long m, long n)      /* MULS Rm,Rn */
{
    MACL=((long)(short)R[n]*(long)(short)R[m]);
    PC+=2;
}
```

(3) **使用例**

```
MULS    R0,R1     ;実行前    R0=H'FFFFFFFE,R1=H'00005555
                  ;実行後    MACL=H'FFFF5556
STS     MACL,R0   ;演算結果を得る
```

6.1.40 MULU.W MULtiply as Unsigned Word 算術演算命令

符号なし乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MULU.W Rm,Rn MULU Rm,Rn	**符号なしで** Rn×Rm→MACL	0010nnnnmmmm1110	1〜3	―	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm を 16 ビットで乗算し、結果の 32 ビットを MACL レジスタに格納します。演算は符号なし算術演算で行います。MACH の内容は変化しません。

(2) **動作内容**

```
MULU(long m, long n)    /* MULU Rm,Rn */
{
    MACL=((unsigned long)(unsigned short)R[n]*
         (unsigned long)(unsigned short)R[m];
    PC+=2;
}
```

(3) **使用例**

```
MULU    R0,R1     ;実行前    R0=H'00000002,R1=H'FFFFAAAA
                  ;実行後    MACL=H'00015554
STS     MACL,R0   ;演算結果を得る
```

6.1.41 NEG NEGate 算術演算命令

符号反転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
NEG Rm,Rn	0−Rm→Rn	0110nnnnmmmm1011	1	—	○	○	○

(1) **説明**

汎用レジスタ Rm の内容の 2 の補数をとり、結果を Rn に格納します。すなわち 0 から Rm を減算し、結果を Rn に格納します。

(2) **動作内容**

```
NEG(long m, long n)      /* NEG Rm,Rn */
{
     R[n]=0-R[m];
     PC+=2;
}
```

(3) **使用例**

```
NEG   R0,R1       ;実行前    R0=H'00000001
                  ;実行後    R1=H'FFFFFFFF
```

6.1.42 NEGC NEGate with Carry 算術演算命令

ボロー付き符号反転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
NEGC Rm,Rn	0ーRmーT→Rn、ボロー→T	0110nnnnmmmm1010	1	ボロー	○	○	○

(1) **説明**

0 から汎用レジスタ Rm の内容と T ビットを減算し、結果を Rn に格納します。演算の結果によってボローを T ビットに反映します。32 ビットを超える値の符号反転を行うとき使用します。

(2) **動作内容**

```
NEGC(long m, long n)  /* NEGC Rm,Rn */
{
    unsigned long temp;

    temp=0-R[m];
    R[n]=temp-T;
    if (0<temp) T=1;
    else T=0;
    if (temp<R[n]) T=1;
    PC+=2;
}
```

(3) **使用例**

```
CLRT              ;R0:R1(64ﾋﾞｯﾄ)の符号反転
NEGC  R1,R1       ;実行前   R1=H'00000001,T=0
                  ;実行後   R1=H'FFFFFFFF,T=1
NEGC  R0,R0       ;実行前   R0=H'00000000,T=1
                  ;実行後   R0=H'FFFFFFFF,T=1
```

6.1.43 NOP No OPeration システム制御命令

無操作

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
NOP	無操作	0000000000001001	1	—	○	○	○

(1) **説明**

PC のインクリメントのみを行い、次の命令に実行を移します。

(2) **動作内容**

```
NOP( )    /* NOP */
{
    PC+=2;
}
```

(3) **使用例**

```
NOP                                    ;1 サイクルの時間が過ぎます。
```

6.1.44 NOT NOT-logical complement 論理演算命令

ビット反転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
NOT Rm,Rn	~Rm→Rn	0110nnnnmmmm0111	1	―	○	○	○

(1) **説明**

汎用レジスタ Rm の内容の 1 の補数をとり、結果を Rn に格納します。すなわち Rm のビットを反転して Rn に格納します。

(2) **動作内容**

```
NOT(long m, long n)/* NOT Rm,Rn */
{
     R[n]=~R[m];
     PC+=2;
}
```

(3) **使用例**

```
NOT   R0,R1       ;実行前    R0=H'AAAAAAAA
                  ;実行後    R1=H'55555555
```

6.1.45 OR OR logical 論理演算命令

論理和演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
OR Rm,Rn	Rn | Rm→Rn	0010nnnnmmmm1011	1	—	○	○	○
OR #imm,R0	R0 | imm→R0	11001011iiiiiiii	1	—	○	○	○
OR.B #imm,@(R0,GBR)	(R0＋GBR) | imm→(R0＋GBR)	11001111iiiiiiii	3	—	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm の論理和をとり、結果を Rn に格納します。

汎用レジスタ R0 とゼロ拡張した 8 ビットのイミディエイトデータとの論理和、もしくはインデックス付き GBR 間接アドレッシングモードで 8 ビットのメモリと 8 ビットのイミディエイトデータとの論理和が可能です。

(2) **動作内容**

```
OR(long m, long n) /* OR Rm,Rn */
{
    R[n]|=R[m];
    PC+=2;
}

ORI(long i) /* OR #imm,R0 */
{
    R[0]|=(0x000000FF & (long)i);
    PC+=2;
}

ORM(long i)/* OR.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp|=(0x000000FF & (long)i);
    Write_Byte(GBR+R[0],temp);
    PC+=2;
}
```

(3) **使用例**

```
OR      R0,R1                ;実行前   R0=H'AAAA5555,R1=H'55550000
                             ;実行後   R1=H'FFFF5555
OR      #H'F0,R0             ;実行前   R0=H'00000008
                             ;実行後   R0=H'000000F8
OR.B    #H'50,@(R0,GBR)      ;実行前   @(R0,GBR)=H'A5
                             ;実行後   @(R0,GBR)=H'F5
```

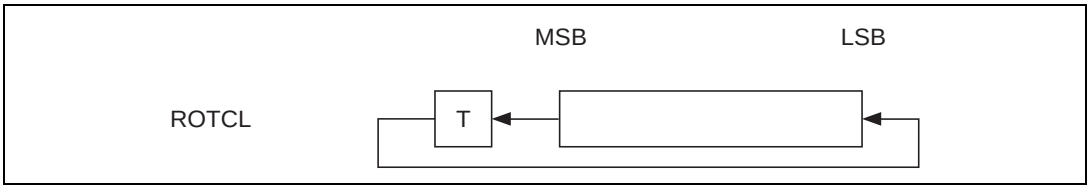
6.1.46 ROTCL ROTate with Carry Left シフト命令

T ビット付き 1 ビット左回転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ROTCL Rn	T←Rn←T	0100nnnn00100100	1	MSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を左方向に T ビットを含めて 1 ビットローテート（回転）し、結果を Rn に格納します。ローテートしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
ROTCL(long n)      /* ROTCL Rn */
{
    long temp;

    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]<<=1;
    if (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
}
```

(3) **使用例**

```
ROTCL   R0          ;実行前   R0=H'80000000,T=0
                    ;実行後   R0=H'00000000,T=1
```

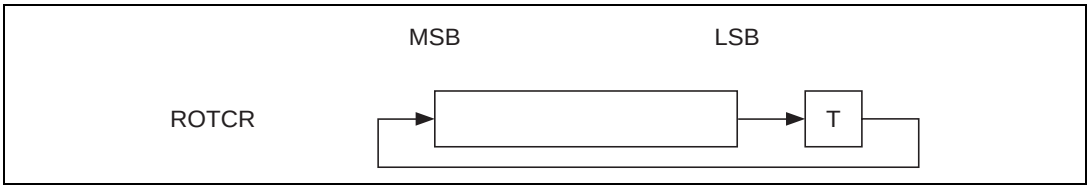
6.1.47 ROTCR ROTate with Carry Right シフト命令

T ビット付き 1 ビット右回転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ROTCR Rn	T→Rn→T	0100nnnn00100101	1	LSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を右方向に T ビットを含めて 1 ビットローテート（回転）し、結果を Rn に格納します。ローテートしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
ROTCR(long n)        /* ROTCR Rn */
{
    long temp;

    if ((R[n]&0x00000001)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (T==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
}
```

(3) **使用例**

```
ROTCR  R0          ;実行前   R0=H'00000001,T=1
                   ;実行後   R0=H'80000000,T=1
```

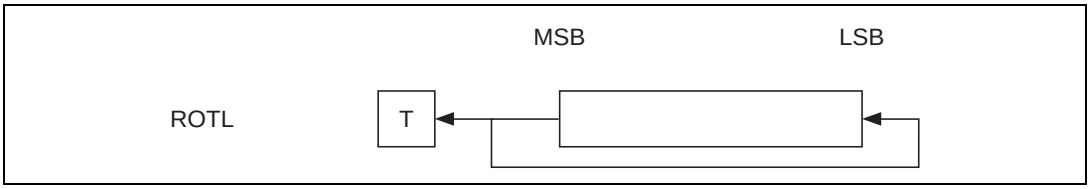
6.1.48 ROTL ROTate Left シフト命令

1 ビット左回転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ROTL Rn	T←Rn←MSB	0100nnnn00000100	1	MSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を左方向に 1 ビットローテート（回転）し、結果を Rn に格納します。ローテートしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
ROTL(long n)      /* ROTL Rn */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    f (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    PC+=2;
}
```

(3) **使用例**

```
ROTL    R0          ;実行前    R0=H'80000000,T=0
                    ;実行後    R0=H'00000001,T=1
```

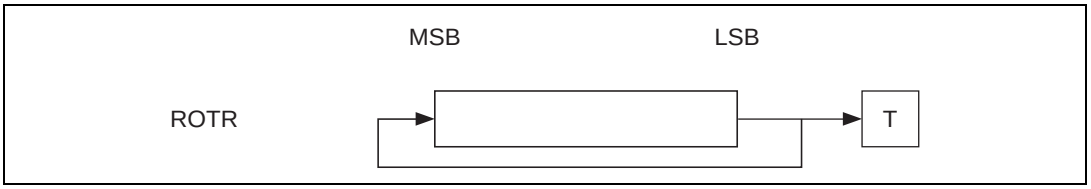
6.1.49 ROTR ROTate Right シフト命令

1 ビット右回転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
ROTR Rn	LSB→Rn→T	0100nnnn00000101	1	LSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を右方向に 1 ビットローテート（回転）し、結果を Rn に格納します。ローテートしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
ROTR(long n)        /* ROTR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    if (T==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

(3) **使用例**

```
ROTR    R0          ;実行前    R0=H'00000001,T=0
                    ;実行後    R0=H'80000000,T=1
```

6.1.50 RTE ReTurn from Exception システム制御命令

例外処理からの復帰 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
RTE	**スタック領域→** PC/SR	0000000000101011	4	LSB	○	○	○

(1) **説明**

割り込みルーチンから復帰します。すなわち、PC と SR をスタックから復帰し、復帰した PC の示すアドレスから処理を続行します。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
RTE(  )   /* RTE */
{
    unsigned long temp;

    temp=PC;
    PC=Read_Long(R[15])+4;
    R[15]+=4;
    SR=Read_Long(R[15])&0x000003F3;
    SR=Read_Long(R[15])&0x0FFF0FFF;
    SR=Read_Long(R[15])&0x0FFF0FFF;
    R[15]+=4;
    Delay_Slot(temp+2);
}
```

SH-1 CPU と SH-2 CPU の場合

SH-DSP の場合

(4) **使用例**

```
RTE                         ;元のルーチンへ復帰します。
ADD #8,R14                  ;分岐に先立ち実行します。
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.51 RTS ReTurn from SubRoutine 分岐命令

サブルーチンプロシージャからの復帰 **遅延分岐命令**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
RTS	PR→PC	0000000000001011	2	―	○	○	○

(1) **説明**

サブルーチンプロシージャから復帰します。すなわち、PC を PR から復帰し、復帰した PC の示すアドレスから処理を続行します。本命令によって、BSR、BSRF および JSR 命令でコールされたサブルーチンプロシージャからコール元へ戻ることができます。

(2) **注意**

遅延分岐命令ですので、本命令の直後の命令を先に実行してから、分岐します。

本命令と直後の命令との間には、アドレスエラーと割り込みを受け付けません。直後の命令が分岐命令のときは、それをスロット不当命令として認識します。

(3) **動作内容**

```
RTS( )    /* RTS */
{
    unsigned long temp;

    temp=PC;
    PC=PR+4;
    Delay_Slot(temp+2);
}
```

(4) **使用例**

```
          MOV.L     TABLE,R3    ;R3=TRGET のｱﾄﾞﾚｽ
          JSR       @R3         ;TRGET へ分岐します。
          NOP                   ;JSR に先立ち実行します。
          ADD       R0,R1       ;←ﾌﾟﾛｼｰｼﾞｬからの戻り先(PR の内容)
          ........
TABLE:    .data.l   TRGET       ;ｼﾞｬﾝﾌﾟﾃｰﾌﾞﾙ
          ........
TRGET:    MOV       R1,R0       ;←ﾌﾟﾛｼｰｼﾞｬの入り口
          RTS                   ;PR の内容→PC
          MOV       #12,R0      ;分岐に先立ち実行します。
```

【注】 遅延分岐においては分岐という動作そのものは、スロット命令の実行後に発生しますが、命令の実行（レジスタの更新など）は、あくまでも遅延分岐命令→遅延スロット命令の順に行われます。たとえば、遅延スロットで分岐先アドレスが格納されたレジスタを変更しても、変更前のレジスタ内容が分岐先アドレスとなります。

6.1.52 SETRC SET repeat count to RC システム制御命令

RC カウンタの設定および繰り返し制御フラグの設定

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
SETRC Rm	Rm［11:0］→ RC（SR［27:16］）、繰返し	0100mmmm00010100	1	—	—	—	○
SETRC #imm	制御フラグ→RF1、RF0 imm→RC(SR［23:16］)zeros→ SR［27:24］、繰り返し制御フラグ→RF1、RF0	10000010iiiiiii0	1	—	—	—	○

(1) **説明**

繰り返し回数を SR レジスタの RC カウンタに設定します。オペランドがレジスタの場合は、下位 12 ビットで繰り返し回数を指定します。イミディエイトデータの場合は、8 ビットで繰り返し回数を指定します。このとき RC の上位 4 ビットは、ゼロ詰めされます。

また、繰り返し制御フラグを SR レジスタの RF1、RF0 ビットにセットします。

SETRC 命令の使用についてはいくつかの制限があります。詳しくは、**「**4.19 DSP **繰り返し（ループ）制御」**を参照してください。

(2) **動作内容**

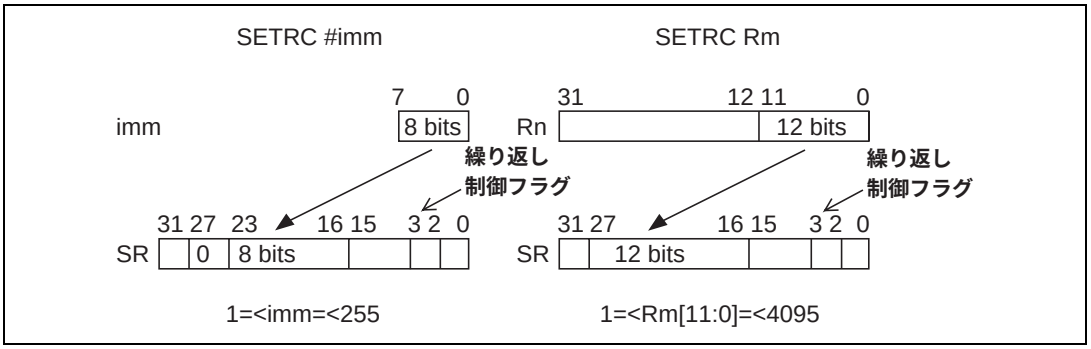
```
SETRC(long m)    /* SETRC Rm */
{
    long temp;

    temp=(R[m] & 0x00000FFF)<<16;
    SR&=0x00000FFF;
    SR|=temp;
    RF1=Repeat_Control_Flag1;
    RF0=Repeat_Control_Flag0;
    PC+=2;
}

SETRCI(long i)    /* SETRC #imm */
{
    long temp;

    temp=((long)i & 0x000000FF)<<16;
    SR&=0x00000FFF;
    SR|=temp;
    RF1=Repeat_Control_Flag1;
    RF0=Repeat_Control_Flag0;
```

```
    PC+=2;
}
```



(3) **使用例**

```
      LDRS STA        ; set repeat start address to RS.
      LDRE END        ; set repeat end address to RE.
      SETRC #32       ; repeat 32 times from inst.A to inst.C.
      inst.0          ;
STA:      inst.A      ;
          inst.B      ;
        ..........
END:      inst.C      ;
      inst.D          ;
        ..........
```

6.1.53 SETT SET Tbit システム制御命令

T ビットのセット

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SETT	1→T	0000000000011000	1	1	○	○	○

(1) **説明**

T ビットをセットします。

(2) **動作内容**

```
SETT( )  /* SETT */
{
    T=1;
    PC+=2;
}
```

(3) **使用例**

```
SETT                ;実行前   T=0
                    ;実行後   T=1
```

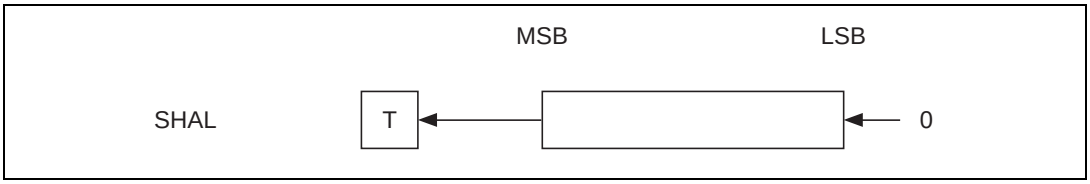
6.1.54 SHAL SHift Arithmetic Left シフト命令

算術的 1 ビット左シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SHAL Rn	T←Rn←0	0100nnnn00100000	1	MSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を左方向に算術的に 1 ビットシフトし、結果を Rn に格納します。シフトしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
SHAL(long n)      /* SHAL Rn (Same as SHLL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

(3) **使用例**

```
SHAL   R0           ;実行前   R0=H'80000001,T=0
                    ;実行後   R0=H'00000002,T=1
```

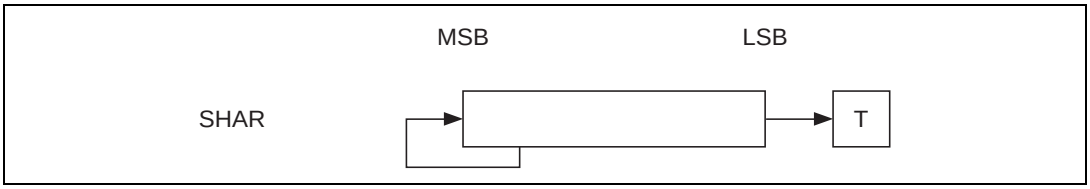
6.1.55 SHAR SHift Arithmetic Right シフト命令

算術的 1 ビット右シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SHAR Rn	MSB→Rn→T	0100nnnn00100001	1	LSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を右方向に算術的に 1 ビットシフトし、結果を Rn に格納します。シフトしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
SHAR(long n)        /* SHAR Rn */
{
    long temp;

    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (temp==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

(3) **使用例**

```
SHAR   R0           ;実行前   R0=H'80000001,T=0
                    ;実行後   R0=H'C0000000,T=1
```

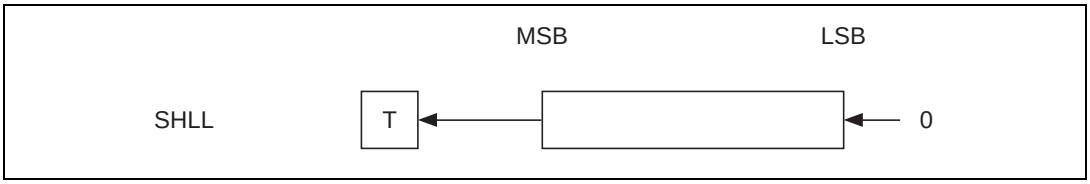
6.1.56 SHLL SHift Logical Left シフト命令

論理的 1 ビット左シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SHLL Rn	T←Rn←0	0100nnnn00000000	1	MSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を左方向に論理的に 1 ビットシフトし、結果を Rn に格納します。シフトしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
SHLL(long n)        /* SHLL Rn  (Same as SHAL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

(3) **使用例**

```
SHLL   R0           ;実行前   R0=H'80000001,T=0
                    ;実行後   R0=H'00000002,T=1
```

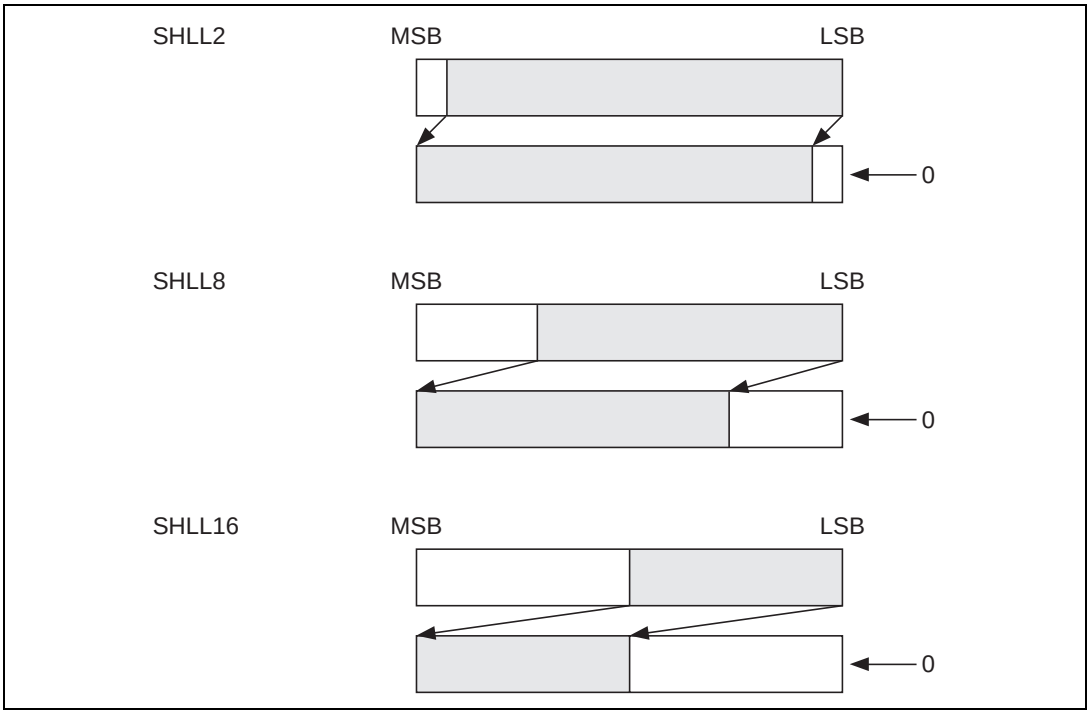
6.1.57 SHLLn n bits SHift Logical Left シフト命令

論理的 n ビット左シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SHLL2 Rn	Rn＜＜2→Rn	0100nnnn00001000	1	—	○	○	○
SHLL8 Rn	Rn＜＜8→Rn	0100nnnn00011000	1	—	○	○	○
SHLL16 Rn	Rn＜＜16→Rn	0100nnnn00101000	1	—	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を左方向に論理的に 2/8/16 ビットシフトし、結果を Rn に格納します。シフトしてオペランドの外に出てしまったビットは捨てます。



(2) **動作内容**

```
SHLL2(long n)     /* SHLL2 Rn */
{
    R[n]<<=2;
    PC+=2;
}

SHLL8(long n)     /* SHLL8 Rn */
{
    R[n]<<=8;
    PC+=2;
}

SHLL16(long n)    /* SHLL16 Rn */
{
    R[n]<<=16;
    PC+=2;
}
```

(3) **使用例**

```
SHLL2  R0          ;実行前   R0=H'12345678
                   ;実行後   R0=H'48D159E0
SHLL8  R0          ;実行前   R0=H'12345678
                   ;実行後   R0=H'34567800
SHLL16 R0          ;実行前   R0=H'12345678
                   ;実行後   R0=H'56780000
```

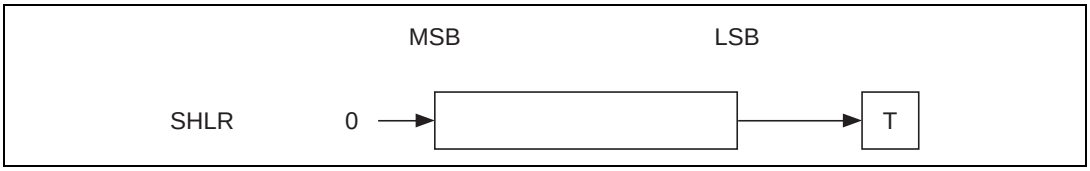
6.1.58 SHLR SHift Logical Right シフト命令

論理的 1 ビット右シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SHLR Rn	0→Rn→T	0100nnnn00000001	1	LSB	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を右方向に論理的に 1 ビットシフトし、結果を Rn に格納します。シフトしてオペランドの外に出てしまったビットは、T ビットへ転送します。



(2) **動作内容**

```
SHLR(long n)        /* SHLR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

(3) **使用例**

```
SHLR   R0           ;実行前   R0=H'80000001,T=0
                    ;実行後   R0=H'40000000,T=1
```

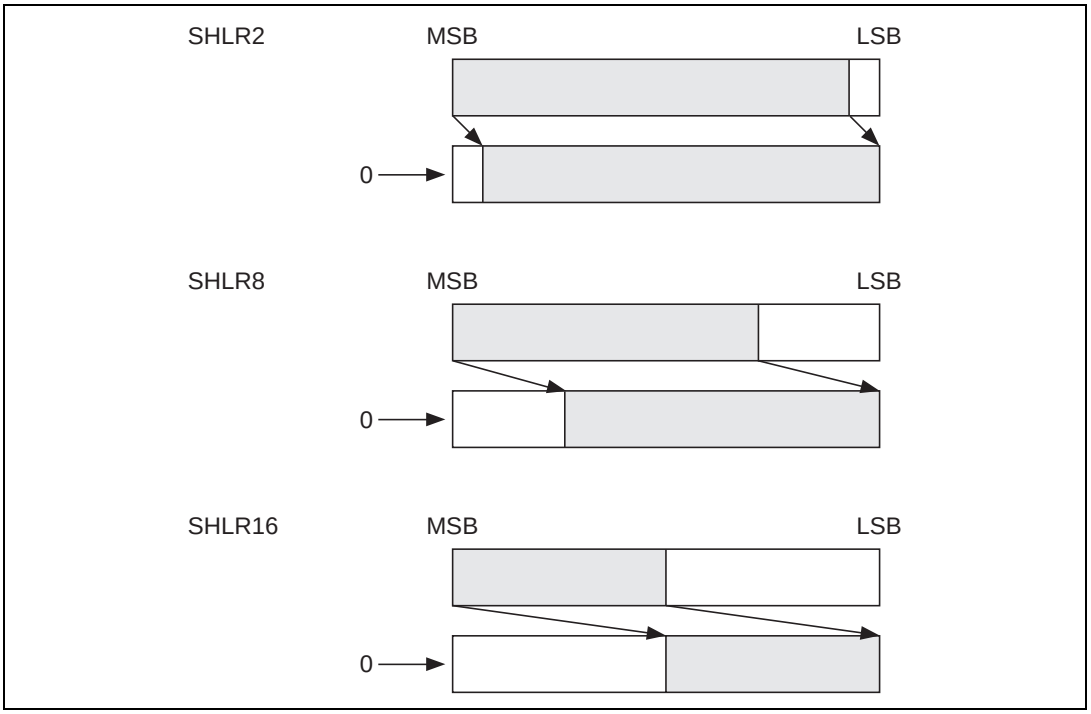
6.1.59 SHLRn n bits SHift Logical Right シフト命令

論理的 n ビット右シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SHLR2 Rn	Rn＞＞2→Rn	0100nnnn00001001	1	—	○	○	○
SHLR8 Rn	Rn＞＞8→Rn	0100nnnn00011001	1	—	○	○	○
SHLR16 Rn	Rn＞＞16→Rn	0100nnnn00101001	1	—	○	○	○

(1) **説明**

汎用レジスタ Rn の内容を右方向に論理的に 2/8/16 ビットシフトし、結果を Rn に格納します。シフトしてオペランドの外に出てしまったビットは捨てます。



(2) **動作内容**

```
SHLR2(long n)      /* SHLR2 Rn */
{
    R[n]>>=2;
    R[n]&=0x3FFFFFFF;
    PC+=2;
}

SHLR8(long n)      /* SHLR8 Rn */
{
    R[n]>>=8;
    R[n]&=0x00FFFFFF;
    PC+=2;
}

SHLR16(long n)     /* SHLR16 Rn */
{
    R[n]>>=16;
    R[n]&=0x0000FFFF;
    PC+=2;
}
```

(3) **使用例**

```
SHLR2  R0          ;実行前   R0=H'12345678
                   ;実行後   R0=H'048D159E
SHLR8  R0          ;実行前   R0=H'12345678
                   ;実行後   R0=H'00123456
SHLR16 R0          ;実行前   R0=H'12345678
                   ;実行後   R0=H'00001234
```

6.1.60 SLEEP SLEEP システム制御命令

低消費電力状態への遷移

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SLEEP	スリープ	0000000000011011	3	—	○	○	○

(1) **説明**

CPU を低消費電力状態にします。

低消費電力状態では、CPU の内部状態を保持し、直後の命令の実行を停止し、割り込み要求の発生を待ちます。要求が発生すると、低消費電力状態から抜けます。

(2) **注意**

実行ステートの 3 は、スリープモードに遷移するまでのステート数です。

(3) **動作内容**

```
SLEEP( )   /* SLEEP */
{
    PC-=2;
    wait_for_exception;
}
```

(4) **使用例**

```
SLEEP                ;低消費電力状態への遷移
```

6.1.61 STC STore Control register システム制御命令

コントロールレジスタからのストア 割り込み禁止命令

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
STC SR,Rn	SR→Rn	0000nnnn00000010	1	—	○	○	○
STC GBR,Rn	GBR→Rn	0000nnnn00010010	1	—	○	○	○
STC VBR,Rn	VBR→Rn	0000nnnn00100010	1	—	○	○	○
STC MOD,Rn	MOD→Rn	0000nnnn01010010	1	—	—	—	○
STC RE,Rn	RE→Rn	0000nnnn01110010	1	—	—	—	○
STC RS,Rn	RS→Rn	0000nnnn01100010	1	—	—	—	○
STC.L SR,@-Rn	Rn−4→Rn、SR→(Rn)	0100nnnn00000011	2	—	○	○	○
STC.L GBR,@-Rn	Rn−4→Rn、GBR→(Rn)	0100nnnn00010011	2	—	○	○	○
STC.L VBR,@-Rn	Rn−4→Rn、VBR→(Rn)	0100nnnn00100011	2	—	○	○	○
STC.L MOD,@-Rn	Rn−4→Rn、MOD→(Rn)	0100nnnn01010011	2	—	—	—	○
STC.L RE,@-Rn	Rn−4→Rn、RE→(Rn)	0100nnnn01110011	2	—	—	—	○
STC.L RS,@-Rn	Rn−4→Rn、RS→(Rn)	0100nnnn01100011	2	—	—	—	○

(1) **説明**

コントロールレジスタ SR、GBR、VBR、MOD、RE、SE のデータをディスティネーションに格納します。

(2) **注意**

本命令と直後の命令との間には、割り込みを受け付けません。(アドレスエラーは受け付けます。)

(3) **動作内容**

```
STCSR(long n)       /*STC SR,Rn */
{
    R[n]=SR;
    PC+=2;
}

STCGBR(long n)      /* STC GBR,Rn */
{
    R[n]=GBR;
    PC+=2;
}

STCVBR(long n)      /* STC VBR,Rn */
{
    R[n]=VBR;
    PC+=2;
}
```

```
STCMOD(long n)     /* STC MOD,Rn */
{
    R[n]=MOD
    PC+=2;
}

STCRE(long n)      /* STC RE, Rn */
{
    R[n]=RE;
    PC+=2;
}

STCRS(long n)      /* STC RS,Rn */
{
    R[n]=RS;
    PC+=2;
}
STCMSR(long n)     /* STC.L SR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SR);
    PC+=2;
}

STCMGBR(long n)    /* STC.L GBR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],GBR);
    PC+=2;
}

STCMVBR(long n)    /* STC.L VBR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],VBR);
    PC+=2;
}

STCMMOD(long n)    /* STC.L MOD,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],MOD);
    PC+=2;
}
```

```
STCMRE(long n)     /* STC.L RE,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],RE);
    PC+=2;
}

STCMRS(long n)     /* STC.L RS, @-Rn */
{
    R[n]-=4;
    Write_Long(R[n],RS);
    PC+=2;
}
```

(4) **使用例**

```
STC     SR,R0     ;実行前         R0=H'FFFFFFFF,SR=H'00000000
                  ;実行後         R0=H'00000000
STC.L   GBR,@-R15 ;実行前         R15=H'10000004
                  ;実行後         R15=H'10000000,@R15=GBR
```

6.1.62 STS STore System register システム制御命令

システムレジスタからのストア 割り込み禁止命令

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
STS MACH,Rn	MACH→Rn	0000nnnn00001010	1	—	○	○	○
STS MACL,Rn	MACL→Rn	0000nnnn00011010	1	—	○	○	○
STS PR,Rn	PR→Rn	0000nnnn00101010	1	—	○	○	○
STS DSR,Rn	DSR→Rn	0000nnnn01101010	1	—	—	—	○
STS A0,Rn	A0→Rn	0000nnnn01111010	1	—	—	—	○
STS X0,Rn	X0→Rn	0000nnnn10001010	1	—	—	—	○
STS X1,Rn	X1→Rn	0000nnnn10011010	1	—	—	—	○
STS Y0,Rn	Y0→Rn	0000nnnn10101010	1	—	—	—	○
STS Y1,Rn	Y1→Rn	0000nnnn10111010	1	—	—	—	○
STS.L MACH,@-Rn	Rn−4→Rn、MACH→(Rn)	0100nnnn00000010	1	—	○	○	○
STS.L MACL,@-Rn	Rn−4→Rn、MACL→(Rn)	0100nnnn00010010	1	—	○	○	○
STS.L PR,@-Rn	Rn−4→Rn、PR→(Rn)	0100nnnn00100010	1	—	○	○	○
STS.L DSR,@-Rn	Rn−4→Rn、DSR→(Rn)	0100nnnn01100010	1	—	—	—	○
STS.L A0,@-Rn	Rn−4→Rn、A0→(Rn)	0100nnnn01110010	1	—	—	—	○
STS.L X0,@-Rn	Rn−4→Rn、X0→(Rn)	0100nnnn10000010	1	—	—	—	○
STS.L X1,@-Rn	Rn−4→Rn、X1→(Rn)	0100nnnn10010010	1	—	—	—	○
STS.L Y0,@-Rn	Rn−4→Rn、Y0→(Rn)	0100nnnn10100010	1	—	—	—	○
STS.L Y1,@-Rn	Rn−4→Rn、Y1→(Rn)	0100nnnn10110010	1	—	—	—	○

(1) **説明**

システムレジスタ MACH、MACL、PR、または DSP レジスタ DSR、A0、X0、X1、Y0、Y1 をデスティネーションに格納します。

(2) **注意**

本命令と直後の命令との間には、割り込みを受け付けません（アドレスエラーは受け付けます）。

SH-1 CPU で MACH からストアするときは、デスティネーションの上位 22 ビット（ビット 31～ビット 10）には、ビット 9 の内容を転送して格納します。SH-2 CPU と SH-DSP では、MACH の 32 ビットをそのまま格納します。

(3) **動作内容**

```
STSMACH(long n)    /* STS MACH,Rn */
{
    R[n]=MACH;
    if((R[n]&0x00000200)==0)R[n]&=0x000003FF;
    else R[n]|=0xFFFFFC00;
    PC+=2;
}

STSMACL(long n)    /* STS MACL,Rn */
{
    R[n]=MACL;
    PC+=2;
}

STSPR(long n)      /* STS PR,Rn */
{
    R[n]=PR;
    PC+=2;
}

STSDSR(long n)     /* STS DSR,Rn */
{
    R[n]=DSR;
    PC+=2;
}

STSA0(long n)      /* STS A0,Rn */
{
    R[n]=A0;
    PC+=2;
}

STSX0(long n)      /* STS X0,Rn */
{
    R[n]=X0;
    PC+=2;
}
STSX1(long n)      /* STS X1,Rn */
{
    R[n]=X1;
    PC+=2;
}
```

SH-1 CPU の場合（この 2 行は、SH-2 CPU と SH-DSP では不要です。）

```
STSY0(long n)      /* STS Y0,Rn */
{
    R[n]=Y0;
    PC+=2;
}

STSY1(long n)      /* STS Y1,Rn */
{
    R[n]=Y1;
    PC+=2;
}

STSMMACH(long n)   /* STS.L MACH,@-Rn */
{
    R[n]-=4;
    if((MACH&0x00000200)==0)
    Write_Long(R[n],MACH0x000003FF);
    else Write_Long(R[n], MACH|0xFFFFFC00);
```

SH-1 CPU の場合

```
    Write_Long(R[n],MACH);
```

SH-2 CPU と SH-DSP の場合

```
    PC+=2;
}

STSMMACL(long n)   /* STS.L MACL,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],MACL);
    PC+=2;
}

STSMPR(long n)     /* STS.L PR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],PR);
    PC+=2;
}
STSMDSR(long n)    /* STS.L DSR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],DSR);
    PC+=2;
}

STSMA0(long n)     /* STS.L A0,@-Rn */
{
```

```
    R[n]-=4;
    Write_Long(R[n],A0);
    PC+=2;
}

STSMX0(long n)     /* STS.L X0,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],X0);
    PC+=2;
}

STSMX1(long n)     /* STS.L X1,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],X1);
    PC+=2;
}

STSMY0(long n)     /* STS.L Y0,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],Y0);
    PC+=2;
}

STSMY1(long n)     /* STS.L Y1,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],Y1);
    PC+=2;
}
```

(4) **使用例**

```
STS     MACH,R0     ;実行前     R0=H'FFFFFFFF,MACH=H'00000000
                    ;実行後     R0=H'00000000
STS.L   PR,@-R15    ;実行前    R15=H'10000004
                    ;実行後    R15=H'10000000,@R15=PR
```

6.1.63 SUB SUBtract binary 算術演算命令

2 進減算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SUB Rm,Rn	Rn－Rm→Rn	0011nnnnmmmm1000	1	―	○	○	○

(1) 説明

汎用レジスタ Rn の内容から Rm を減算し、結果を Rn に格納します。イミディエイトデータとの減算は ADD #imm,Rn を使います。

(2) 動作内容

```
SUB(long m, long n)     /* SUB Rm,Rn */
{
    R[n]-=R[m];
    PC+=2;
}
```

(3) 使用例

```
SUB   R0,R1          ;実行前   R0=H'00000001,R1=H'80000000
                     ;実行後   R1=H'7FFFFFFF
```

6.1.64 SUBC SUBtract with Carry 算術演算命令

ボロー付き 2 進減算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SUBC Rm,Rn	Rn－Rm－T→Rn、ﾎﾞﾛｰ→T	0011nnnnmmmm1010	1	ボロー	○	○	○

(1) **説明**

汎用レジスタ Rn の内容から Rm と T ビットを減算し、結果を Rn に格納します。演算の結果によってボローを T ビットに反映します。32 ビットを超える減算を行うとき使用します。

(2) **動作内容**

```
SUBC(long m, long n)    /* SUBC Rm,Rn */
{
    unsigned long tmp0,tmp1;

    tmp1=R[n]-R[m];
    tmp0=R[n];
    R[n]=tmp1-T;
    if (tmp0<tmp1) T=1;
    else T=0;
    if (tmp1<R[n]) T=1;
    PC+=2;
}
```

(3) **使用例**

```
CLRT                ; R0:R1(64ﾋﾞｯﾄ)-R2:R3(64ﾋﾞｯﾄ)=R0:R1(64ﾋﾞｯﾄ)
SUBC  R3,R1         ;実行前   T=0,R1=H'00000000,R3=H'00000001
                    ;実行後   T=1,R1=H'FFFFFFFF
SUBC  R2,R0         ;実行前   T=1,R0=H'00000000,R2=H'00000000
                    ;実行後   T=1,R0=H'FFFFFFFF
```

6.1.65 SUBV SUBtract with (Vflag) underflow check 算術演算命令

アンダフロー付き 2 進減算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
SUBV Rm,Rn	Rn－Rm→Rn、アンダフロー→T	0011nnnnmmmm1011	1	アンダフロー	○	○	○

(1) **説明**

汎用レジスタ Rn の内容から Rm を減算し、結果を Rn に格納します。アンダフローが発生すると、T ビットをセットします。

(2) **動作内容**

```
SUBV(long m, long n)    /* SUBV Rm,Rn */
        {
            long dest,src,ans;

            if ((long)R[n]>=0) dest=0;
            else dest=1;
            if ((long)R[m]>=0) src=0;
            else src=1;
            src+=dest;
            R[n]-=R[m];
            if ((long)R[n]>=0) ans=0;
            else ans=1;
            ans+=dest;
            if (src==1) {
                if (ans==1) T=1;
                else T=0;
            }
            else T=0;
            PC+=2;
        }
```

(3) **使用例**

```
SUBV   R0,R1        ;実行前   R0=H'00000002,R1=H'80000001
                    ;実行後   R1=H'7FFFFFFF,T=1
SUBV   R2,R3        ;実行前   R2=H'FFFFFFFE,R3=H'7FFFFFFE
                    ;実行後   R3=H'80000000,T=1
```

6.1.66 SWAP SWAP register halves データ転送命令

上位と下位の交換

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
SWAP.B Rm,Rn	Rm→下位 2 ﾊﾞｲﾄの上下ﾊﾞｲﾄ交換→Rn	0110nnnnmmmm1000	1	―	○	○	○
SWAP.W Rm,Rn	Rm→上下ﾜｰﾄﾞ交換→Rn	0110nnnnmmmm1001	1	―	○	○	○

(1) **説明**

汎用レジスタ Rm の内容の上位と下位を交換して、結果を Rn に格納します。

バイト指定のとき、Rm のビット 0 からビット 7 の 8 ビットと、ビット 8 からビット 15 の 8 ビットを交換します。Rn の上位 16 ビットには Rm の上位 16 ビットをそのまま転送します。

ワード指定のとき、Rm のビット 0 からビット 15 の 16 ビットと、ビット 16 からビット 31 の 16 ビットを交換します。

(2) **動作内容**

```
SWAPB(long m, long n)    /* SWAP.B Rm,Rn */
{
    unsigned long temp0,temp1;

    temp0=R[m]&0xffff0000;
    temp1=(R[m]&0x000000ff)<<8;
    R[n]=(R[m]>>8)&0x000000ff;
    R[n]=R[n]|temp1|temp0;
    PC+=2;
}

SWAPW(long m, long n)    /* SWAP.W Rm,Rn */
{
    unsigned long temp;

    temp=(R[m]>>16)&0x0000FFFF;
    R[n]=R[m]<<16;
    R[n]|=temp;
    PC+=2;
}
```

(3) **使用例**

```
SWAP.B    R0,R1      ;実行前   R0=H'12345678
                     ;実行後   R1=H'12347856
SWAP.W    R0,R1      ;実行前   R0=H'12345678
                     ;実行後   R1=H'56781234
```

6.1.67 TAS Test And Set 論理演算命令

メモリテストとビットセット

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
TAS.B @Rn	(Rn)が 0 のとき 1→T、1→MSBof(Rn)	0100nnnn00011011	4	ﾃｽﾄ結果	○	○	○

(1) **説明**

汎用レジスタ Rn の内容をアドレスとし、そのアドレスの示すバイトデータを読み込み、そのデータがゼロのとき T＝1、ゼロでないとき T＝0 とします。その後、ビット 7 を 1 にセットして同じアドレスへ書き込みます。この間、バス権は解放しません。

(2) **動作内容**

```
TAS(long n)              /* TAS.B @Rn */
{
    long temp;

    temp=(long)Read_Byte(R[n]);     /* Bus Lock enable */
    if (temp==0) T=1;
    else T=0;
    temp|=0x00000080;
    Write_Byte(R[n],temp);          /* Bus Lock disable */
    PC+=2;
}
```

(3) **使用例**

```
    _LOOP    TAS.B    @R7     ;R7=1000
             BF       _LOOP   ;1000 番地がｾﾞﾛになるまでﾙｰﾌﾟします。
```

6.1.68 TRAPA TRAP Always システム制御命令

トラップ例外処理

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
TRAPA #imm	PC/SR→スタック領域、 (imm×4+VBR)→PC	11000011iiiiiiii	8	—	○	○	○

(1) **説明**

トラップ例外処理を開始します。すなわち PC と SR をスタック領域に退避し、指定ベクタの内容で表されるアドレスへ分岐します。ベクタは 8 ビットイミディエートデータをゼロ拡張後 4 倍したメモリアドレスそのものです。PC は次命令の先頭アドレスです。

RTE と組み合わせて、システムコールに使用します。

(2) **動作内容**

```
TRAPA(long i)      /* TRAPA #imm */
{
    long imm;

    imm=(0x000000FF & i);
    R[15]-=4;
    Write_Long(R[15],SR);
    R[15]-=4;
    Write_Long(R[15],PC-2);
    PC=Read_Long(VBR+(imm<<2))+4;
}
```

(3) **使用例**

```
アドレス
VBR+H'80    .data.l   10000000  ;
            .......
            TRAPA     #H'20     ;VBR+H'80 番地の内容のアドレスに分岐します。
            TST       #0,R0     ;←トラップルーチンからの戻り先
            .......             （スタックした PC の内容）です。
            .......
10000000    XOR       R0,R0     ;←トラップルーチンの入り口
10000002    RTE                 ;上記 TST に戻ります。
10000004    NOP                 ;RTE に先立ち実行します。
```

6.1.69 TST TeST logical 論理演算命令

論理積演算の T ビットセット

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
TST Rm,Rn	Rn&Rm、結果が 0 のとき 1→T	0010nnnnmmmm1000	1	テスト結果	○	○	○
TST #imm,R0	R0&imm、結果が 0 のとき 1→T	11001000iiiiiiii	1	テスト結果	○	○	○
TST.B #imm,@(R0,GBR)	(R0＋GBR)&imm、結果が 0 のとき 1→T	11001100iiiiiiii	3	テスト結果	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm の論理積をとり、結果がゼロのとき T ビットをセットします。結果がゼロでないとき T ビットをクリアします。Rn の内容は変更しません。

汎用レジスタ R0 とゼロ拡張した 8 ビットのイミディエイトデータとの論理積、もしくはインデックス付き GBR 間接アドレッシングモードで 8 ビットのメモリと 8 ビットのイミディエイトデータとの論理積が可能です。R0、もしくはメモリの内容は変更しません。

(2) **動作内容**

```
TST(long m, long n)     /* TST Rm,Rn */
{
    if ((R[n]&R[m])==0) T=1;
    else T=0;
    PC+=2;
}

TSTI(long i)            /* TST #imm,R0 */
{
    long temp;

    temp=R[0]&(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}

TSTM(long i)            /* TST.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp&=(0x000000FF & (long)i);
    if (temp==0) T=1;
```

```
    else T=0;
    PC+=2;
}
```

(3) **使用例**

```
TST     R0,R0               ;実行前   R0=H'00000000
                            ;実行後   T=1
TST     #H'80,R0            ;実行前   R0=H'FFFFFF7F
                            ;実行後   T=1
TST.B   #H'A5,@(R0,GBR)     ;実行前   @(R0,GBR)=H'A5
                            ;実行後   T=0
```

6.1.70 XOR eXclusive OR logical 論理演算命令

排他的論理和演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
XOR Rm,Rn	Rn ^ Rm→Rn	0010nnnnmmmm1010	1	―	○	○	○
XOR #imm,R0	R0 ^ imm→R0	11001010iiiiiiii	1	―	○	○	○
XOR.B #imm,@(R0,GBR)	(R0＋GBR)^imm→(R0＋GBR)	11001110iiiiiiii	3	―	○	○	○

(1) **説明**

汎用レジスタ Rn の内容と Rm の排他的論理和をとり、結果を Rn に格納します。

汎用レジスタ R0 とゼロ拡張した 8 ビットのイミディエイトデータとの排他的論理和、もしくはインデックス付き GBR 間接アドレッシングモードで 8 ビットのメモリと 8 ビットのイミディエイトデータとの排他的論理和が可能です。

(2) **動作内容**

```
XOR(long m, long n)        /* XOR Rm,Rn */
{
    R[n]^=R[m];
    PC+=2;
}

XORI(long i)               /* XOR #imm,R0 */
{
    R[0]^=(0x000000FF & (long)i);
    PC+=2;
}

XORM(long i)               /* XOR.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp^=(0x000000FF & (long)i);
    Write_Byte(GBR+R[0],temp);
    PC+=2;
}
```

(3) **使用例**

```
XOR     R0,R1               ;実行前   R0=H'AAAAAAAA,R1=H'55555555
                            ;実行後   R1=H'FFFFFFFF
XOR     #H'F0,R0            ;実行前   R0=H'FFFFFFFF
                            ;実行後   R0=H'FFFFFF0F
XOR.B   #H'A5,@(R0,GBR)     ;実行前   @(R0,GBR)=H'A5
                            ;実行後   @(R0,GBR)=H'00
```

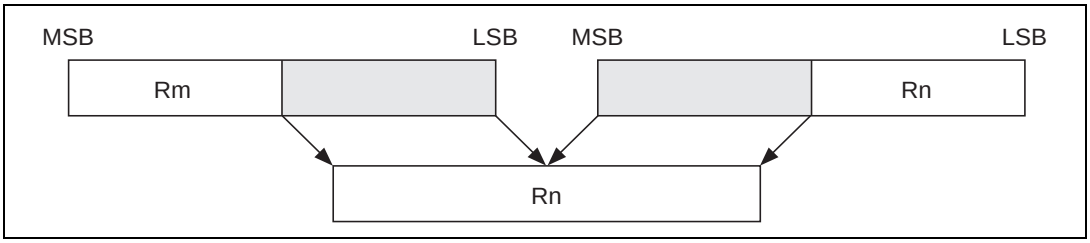
6.1.71 XTRCT eXTRaCT データ転送命令

連結レジスタの中央切り出し

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
XTRCT Rm,Rn	Rm：Rn の中央 32 ﾋﾞｯﾄ →Rn	0010nnnnmmmm1101	1	―	○	○	○

(1) **説明**

汎用レジスタ Rm と Rn とを連結した 64 ビットの内容から中央の 32 ビットを切り出し、結果を Rn に格納します。



(2) **動作内容**

```
XTRCT(long m, long n)     /* XTRCT Rm,Rn */
{
    unsigned long temp;

    temp=(R[m]<<16)&0xFFFF0000;
    R[n]=(R[n]>>16)&0x0000FFFF;
    R[n]|=temp;
    PC+=2;
}
```

(3) **使用例**

```
XTRCT   R0,R1       ;実行前  R0=H'01234567,R1=H'89ABCDEF
                    ;実行後  R1=H'456789AB
```

6.2 DSP データ転送命令の説明

X、Y データ転送（MOVX.W、MOVY.W）、シングルデータ転送（MOVS.W、MOVS.L）の動作を説明します。

(1) X、Y データ転送（MOVX.W、MOVY.W）

この命令は XDB バス、YDB バスを使って X、Y メモリをアクセスします。X、Y メモリ以外のエリアはアクセスできません。メモリアクセスはワード単位のアクセスです。独立したバスを使用するため、命令フェッチ（メインバスを使用）とはアクセス競合が発生しません。

同時並行して実行されるデータ演算命令が条件付命令であっても、X、Y データ転送命令は条件に関係なく実行されます。

X、Y データ転送のロード、ストアの動作を**図** 6.1 に示します。

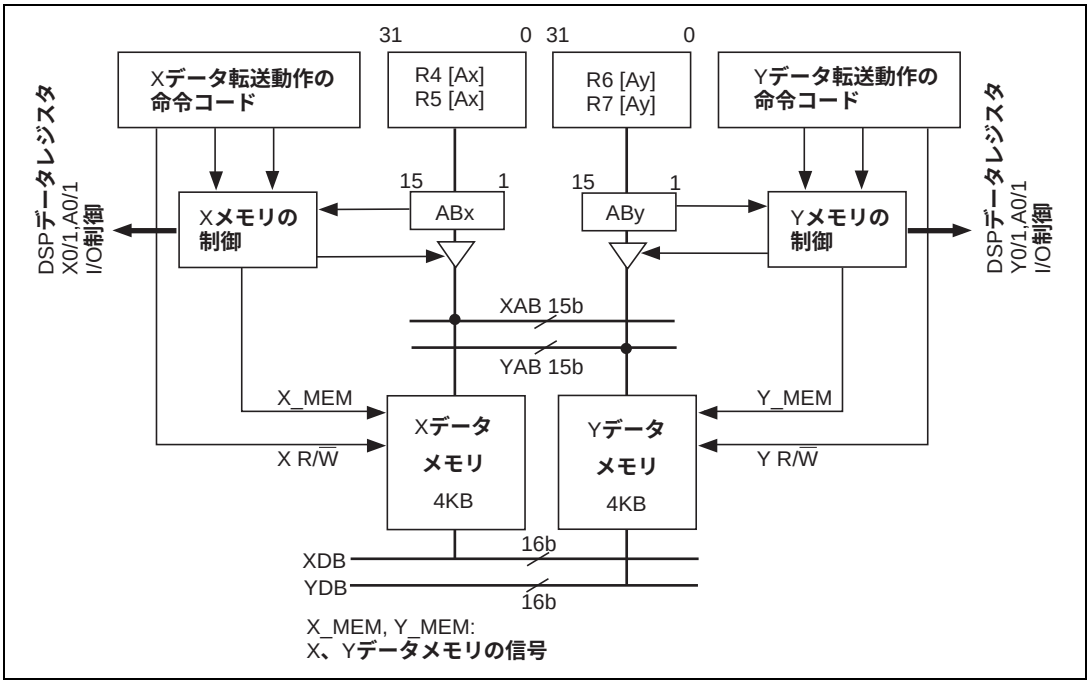


図 6.1 X、Y **データ転送のロード、ストアの動作**

X メモリデータ転送の動作を次に示します。Y メモリデータ転送も同様です。

```
if ( !Nop ) {
      X_MEM=1; XAB=ABx;X R/W＝1
      if ( load opeation ) {
            Dx[31:16]=XDB;
            Dx[15:0] =0x0000;     /* Dx is X0 or X1 */
      }
      else { XDB=Dx[31:16];X R/W＝0; }
                                  /* Dx is A0 or A1 */
}
else { X_MEM=0; XAB=Unknown; }
```

(2) **シングルデータ転送**（MOVS.W、MOVS.L）

シングルデータ転送は DSP レジスタにロード、ストアする命令で、システムレジスタのロード、ストア命令と似ています。メモリと DSP レジスタとの間のデータ転送をメインバスを使って転送します。CPU コアの命令と同じようにデータアクセスと命令アクセスの競合が発生します。

シングルデータ転送はワード長、ロングワード長のデータが転送できます。

シングルデータ転送のロード、ストアの動作を**図** 6.2 に示します。

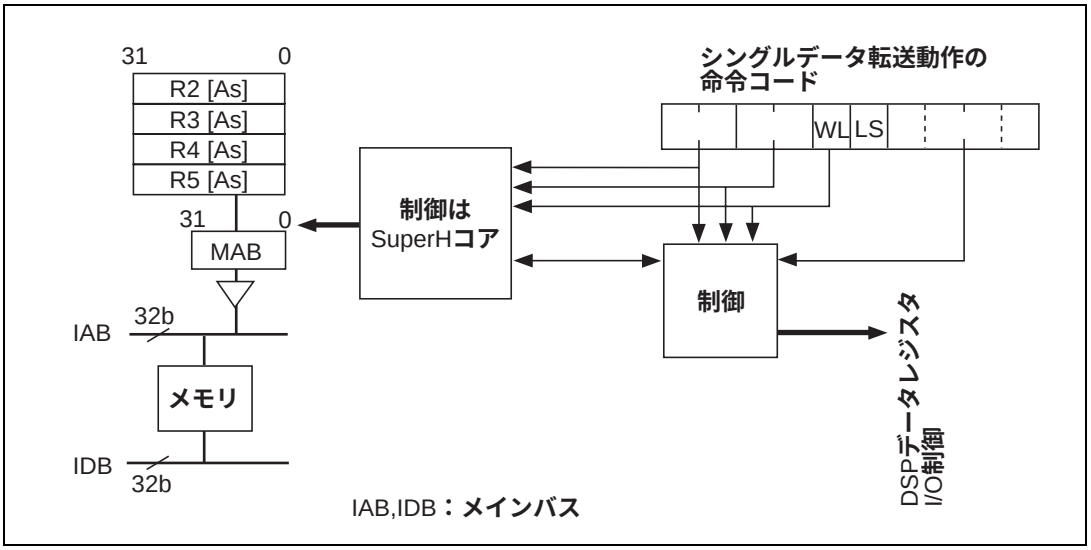


図 6.2 **シングルデータ転送のロード、ストアの動作**

DSP データ転送命令を、アルファベット順に示します。

命令の名称　　命令の機能(英文)　　命令の分類

命令の機能　　遅延分岐命令、または割り込み禁止命令の表示

書式	動作概略	命令コード	実行ステート	T ビット	適用命令
アセンブラの入力書式を表示します。	動作の概略を表示します。	MSB⇔LSB の順に表示します。	DSP 命令はすべて 1 です。	命令実行後の、DC ビットの状態を表示します。	命令が SH-1、SH-2、SH-DSP のどの CPU に適用しているかを示します。

・書式

[if cc] OP.Sz SRC1,SRC2,DEST

[if cc]　：条件、（無条件、DCT または DCF）
OP　：オペレーションニーモニック
Sz　：サイズ
SRC1　：ソース 1 オペランド
SRC2　：ソース 2 オペランド
DEST　：デスティネーションオペランド

・動作概略

→,←　：転送方向
(xx)　：メモリオペランド
DC　：DSR 内のフラグビット
&　：ビットごとの論理積
|　：ビットごとの論理和
^　：ビットごとの排他的論理和
~　：ビットごとの論理否定
<<n　：左 n ビットシフト
>>n　：右 n ビットシフト
MSW　：上位ワード（ビット 31～16）
LSW　：下位ワード（ビット 15～0）
[n1:n2]　：ビット n1～n2

・命令コード

ソースレジスタ、デスティネーションレジスタを表します。

X データ転送命令

A(Ax): 0=R4, 1=R5
D(デスティネーション、Dx): 0=X0, 1=X1
D(ソース、Da): 0=A0, 1=A1

Y データ転送命令

A(Ay): 0=R6, 1=R7
D(デスティネーション、Dy): 0=Y0, 1=Y1
D(ソース、Da): 0=A0, 1=A1

シングルデータ転送命令

AA(As): 0=R4, 1=R5, 2=R2, 3=R3
DDDD(Ds): 5=A1, 7=A0, 8=X0, 9=X1, A=Y0, B=Y1, C=M0, D=A1G,

E=M1,
F=A0G

DSP 演算命令

iiiiiii(imm): PSHA の場合は–32～+32、PSHL の場合は–16～+16
ee(Se): 0=X0, 1=X1, 2=Y0, 3=A1
ff(Sf): 0=Y0, 1=Y1, 2=X0, 3=A1
xx(Sx): 0=X0, 1=X1, 2=A0, 3=A1
yy(Sy): 0=Y0, 1=Y1, 2=M0, 3=M1
gg(Dg): 0=M0, 1=M1, 2=A0, 3=A1
uu(Du): 0=X0, 1=Y0, 2=A0, 3=A1
zzzz(Dz): 5=A1, 7=A0, 8=X0, 9=X1, A=Y0, B=Y1, C=M0, E=M1

・DC ビット

更新　：演算結果と状態選択ビット（CS）の指定に従って更新されます。
—　：更新されません。

(1) 説明

命令の動作を説明します。

(2) 注意

命令を使う上で特に注意が必要なことを説明します。

(3) 動作内容

C 言語で動作内容を表示します。

(4) 使用例

アセンブラニーモニックで例を示し、命令の実行前後の状態を表示します。

6.2.1 MOVS MOVe Single data between memory and dsp register DSP データ転送命令

シングルデータ転送

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
MOVS.W @-As,Ds	As－2→As、(As)→MSW of Ds、0→LSW of Ds	111101AADDDD0000	1	－	－	－	○
MOVS.W @As,Ds	(As)→MSW of Ds、0→LSW of Ds	111101AADDDD0100	1	－	－	－	○
MOVS.W @As+,Ds	(As)→MSW of Ds、0→LSW of Ds、As＋2→As	111101AADDDD1000	1	－	－	－	○
MOVS.W @As+Is,Ds	(As)→MSW of Ds、0→LSW of Ds、As＋Is→As	111101AADDDD1100	1	－	－	－	○
MOVS.W Ds,@-As	As－2→As、MSW of Ds→(As)	111101AADDDD0001	1	－	－	－	○
MOVS.W Ds,@As	MSW of Ds→(As)	111101AADDDD0101	1	－	－	－	○
MOVS.W Ds,@As+	MSW of Ds→(As)、As＋2→As	111101AADDDD1001	1	－	－	－	○
MOVS.W Ds,@As+Is	MSW of Ds→(As)、As＋Is→As	111101AADDDD1101	1	－	－	－	○
MOVS.L @-As,Ds	As－4→As、(As)→Ds	111101AADDDD0010	1	－	－	－	○
MOVS.L @As,Ds	(As)→Ds	111101AADDDD0110	1	－	－	－	○
MOVS.L @As+,Ds	(As)→Ds、As＋4→As	111101AADDDD1010	1	－	－	－	○
MOVS.L @As+Is,Ds	(As)→Ds、As＋Is→As	111101AADDDD1110	1	－	－	－	○
MOVS.L Ds,@-As	As－4→As、Ds→(As)	111101AADDDD0011	1	－	－	－	○
MOVS.L Ds,@As	Ds→(As)	111101AADDDD0111	1	－	－	－	○
MOVS.L Ds,@As+	Ds→(As)、As＋4→As	111101AADDDD1011	1	－	－	－	○
MOVS.L Ds,@As+Is	Ds→(As)、As＋Is→As	111101AADDDD1111	1	－	－	－	○

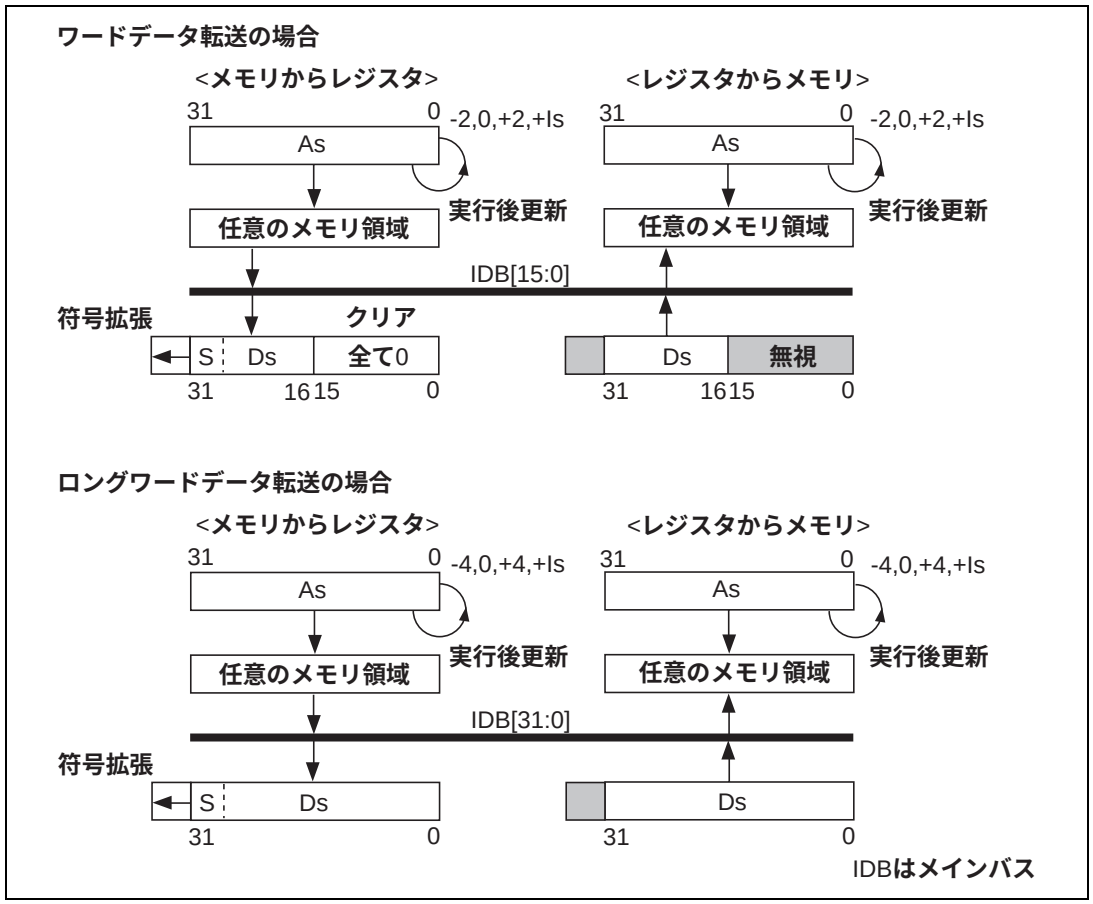
(1) **説明**

ソースオペランドのデータをデスティネーションオペランドへ転送します。メモリからレジスタへ、レジスタからメモリへ転送します。ワード長、ロングワード長を指定できます。ワード転送の場合、ソースオペランドがメモリでデスティネーションオペランドがレジスタのときは、ワードデータはレジスタの上位ワードにロードされ、下位ワードは 0 でクリアされます。ソースオペランドがレジスタでデスティネーションオペランドがメモリのときは、レジスタの上位ワードがワードデータとしてストアされます。ロングワード転送の場合はロングワードデータが転送されます。レジスタがデスティネーションオペランドでガードビットがある場合は、符号が拡張されてガードビットに格納されます。

(2) **注意**

ガードビットレジスタ、A0G、A1G の 1 つがストア処理のソースオペランドのときは、データは最下位 8 ビット（ビット 7～0）に出力され、上位 24 ビット（ビット 31～8）は符号拡張されます。

(3) **動作内容**



(4) **使用例**

```
MOVS.W   @R4＋,A0     ;実行前：R4=H'00000400,(R4)=H'8765,
                            A0=H'123456789A
                       実行後：R4=H'00000402,A0=H'FF87650000
MOVS.L   A1,@-R3      ;実行前：R3=H'00000800,A1=H'123456789A
                       実行後：R3=H'000007FC,
                            (H'000007FC)=H'3456789A
```

6.2.2 MOVX MOVe between X memory and dsp register DSP データ転送命令

X メモリデータ転送

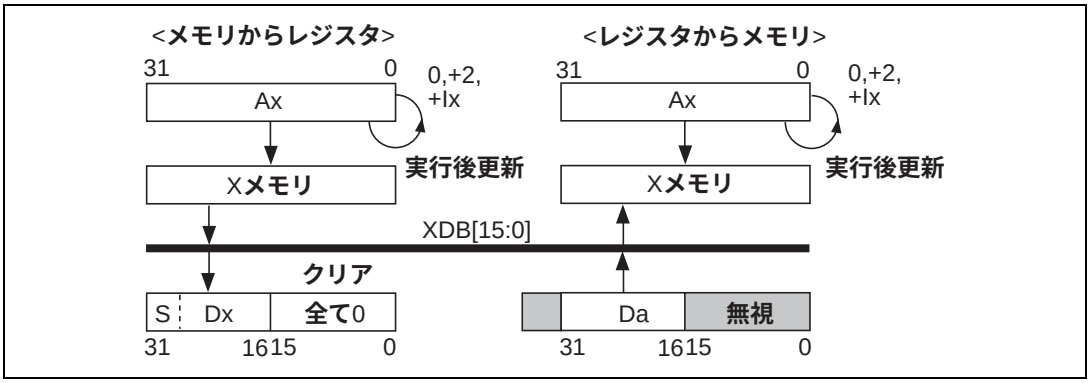
書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MOVX.W @Ax,Dx	(Ax)→MSW of Dx、0→LSW of Dx	111100A*D*0*01**	1	−	−	−	○
MOVX.W @Ax+,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax＋2→Ax	111100A*D*0*10**	1	−	−	−	○
MOVX.W @Ax+Ix,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax＋Ix→Ax	111100A*D*0*11**	1	−	−	−	○
MOVX.W Da,@Ax	MSW of Da→(Ax)	111100A*D*1*01**	1	−	−	−	○
MOVX.W Da,@Ax+	MSW of Da→(Ax)、Ax＋2→Ax	111100A*D*1*10**	1	−	−	−	○
MOVX.W Da,@Ax+Ix	MSW of Da→(Ax)、Ax＋Ix→Ax	111100A*D*1*11**	1	−	−	−	○

命令コードの「*」部分は、MOVY 命令指定領域です。

(1) **説明**

ソースオペランドのデータをデスティネーションオペランドへ転送します。メモリからレジスタへ、レジスタからメモリへ転送します。X メモリとワード長だけが指定できます。ソースオペランドがメモリでデスティネーションオペランドがレジスタのときは、ワードデータはレジスタの上位ワードにロードされ、下位ワードは 0 でクリアされます。ソースオペランドがレジスタでデスティネーションオペランドがメモリのときは、レジスタの上位ワードがワードデータとしてストアされます。

(2) **動作内容**



(3) **使用例**

```
MOVX.W @R4＋,X0          ；実行前：R4=H'08010000,(R4)=H'5555, X0=H'12345678
                           実行後：R4=H'08010002,X0=H'55550000
```

6.2.3 MOVY MOVe between Y memory and dsp register DSP データ転送命令

Y メモリデータ転送

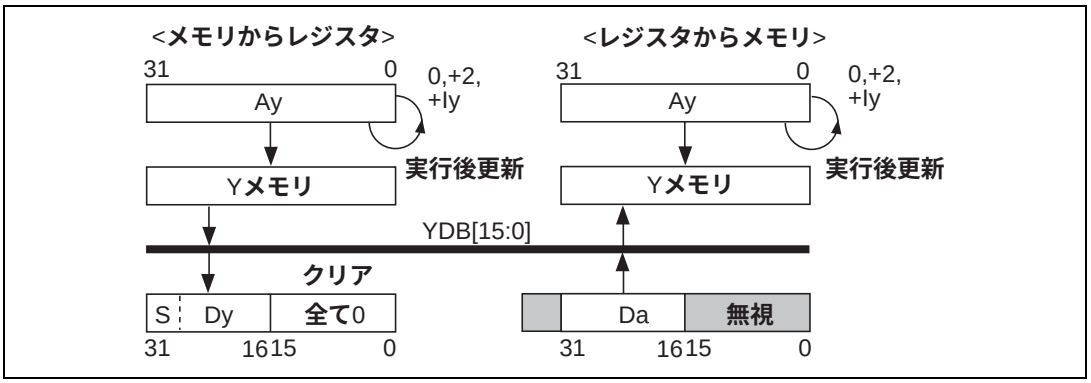
書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
MOVY.W @Ay,Dy	(Ay)→MSW of Dy、0→LSW of Dy	111100*A*D*0**01	1	−	−	−	○
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay＋2→Ay	111100*A*D*0**10	1	−	−	−	○
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay＋Iy→Ay	111100*A*D*0**11	1	−	−	−	○
MOVY.W Da,@Ay	MSW of Da→(Ay)	111100*A*D*1**01	1	−	−	−	○
MOVY.W Da,@Ay+	MSW of Da→(Ay)、Ay＋2→Ay	111100*A*D*1**10	1	−	−	−	○
MOVY.W Da,@Ay+Iy	MSW of Da→(Ay)、Ay＋Iy→Ay	111100*A*D*1**11	1	−	−	−	○

命令コードの「*」部分は、MOVX 命令の指定領域です。

(1) 説明

ソースオペランドのデータをデスティネーションオペランドへ転送します。メモリからレジスタへ、レジスタからメモリへ転送します。Y メモリとワード長だけが指定できます。ソースオペランドがメモリでデスティネーションオペランドがレジスタのときは、ワードデータはレジスタの上位ワードにロードされ、下位ワードは 0 でクリアされます。ソースオペランドがレジスタでデスティネーションオペランドがメモリのときは、レジスタの上位ワードがワードデータとしてストアされます。

(2) 動作内容



(3) **使用例**

```
MOVY.W A0,@R6＋R9        ;実行前：R6=H'08020000, R9=H'00000006,
                                  A0=H'123456789A
                          実行後：R6=H'08020006, (H'08020000)=H'3456
```

6.2.4 NOPX No access OPeration for X memory DSP データ転送命令

X メモリ無操作

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
NOPX	No Operation	1111000*0*0*00**	1	ー	ー	ー	○

命令コードの「*」部分は、MOVY 命令の指定領域です。

(1) 説明

X メモリのアクセスについて無操作です。
このニーモニックは省略可能です。

6.2.5 NOPY No access OPeration for Y memory DSP データ転送命令

Y メモリ無操作

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
NOPY	No Operation	111100*0*0*0**00	1	ー	ー	ー	○

命令コードの「*」部分は、MOVX 命令の指定領域です。

(1) 説明

Y メモリのアクセスについて無操作です。
このニーモニックは省略可能です。

6.3 DSP 演算命令の説明

DSP 命令は CPU 命令と同じ形式で説明します。しかし、C を用いた動作内容の記述のところでは以下の DSP の資源の使用を仮定しています。

(1) DSP レジスタ

DSP レジスタの名称は下記の DSP_Register_Set という名前のユニオンを基に定義します。このユニオンは 11 個のロングワードで構成され、各々のロングワードは 11 個の DSP レジスタ（A0、A1、M0、M1、X0、X1、Y0、Y1、AG0、AG1、DSR）に対応します。

```
/* Union DSP_Register_Set の定義  */

union {
        unsigned long int uli[11];
        unsigned short int usi[22];
        struct {
              struct {
                unsigned short int usi[2];
              } ee[11];
        } dd;
        struct {
              struct {
                union {
                        unsigned long int uli;
                        unsigned short int usi[2];
                        struct {
                                unsigned msb:  1;
                                unsigned :     23;
                                unsigned g_msb:1;
                                unsigned :     7;
                        } bb;
                        struct {
                                unsigned :     24;
                                unsigned lsb8: 8;
                        } cc;
                } mm;
              } a0, a1, m0, m1, x0, x1, y0, y1, a0g, a1g;
              union {
                unsigned long int uli;
                struct {
                        unsigned Reserved:  24;
                        unsigned   gz: 1;      /* Signed greater than */
                        unsigned    z: 1;      /* Zero value */
                        unsigned    n: 1;      /* Negative value */
                        unsigned    v: 1;      /* Overflow */
                        unsigned   cs: 3;      /* Condition Selection */
                        unsigned   dc: 1;      /* dsp condition bit */
                } a;
              } dsr;
        } name;
        struct {
              unsigned short int a[2][2];
              unsigned short int m[2][2];
              unsigned short int x[2][2];
```

```
            unsigned short int y[2][2];
            unsigned short int ag[2][2];
            unsigned short int dsr[2];
        } word;
} DSP_Register_Set;
```

上記のユニオン DSP_Register_Set を用いて以下のように DSP レジスタの名称を定義します。

```
/* DSP Register 名の定義 */

#define MACL                DSP_Register_Set.name.a0.mm.uli
#define A0                  DSP_Register_Set.name.a0.mm.uli
#define A0_HW               DSP_Register_Set.name.a0.mm.usi[0]
#define A0_LW               DSP_Register_Set.name.a0.mm.usi[1]
#define A0_MSB              DSP_Register_Set.name.a0.mm.bb.msb

#define MACH                DSP_Register_Set.name.a1.mm.uli
#define A1                  DSP_Register_Set.name.a1.mm.uli
#define A1_HW               DSP_Register_Set.name.a1.mm.usi[0]
#define A1_LW               DSP_Register_Set.name.a1.mm.usi[1]
#define A1_MSB              DSP_Register_Set.name.a1.mm.bb.msb

#define M0                  DSP_Register_Set.name.m0.mm.uli
#define M0_HW               DSP_Register_Set.name.m0.mm.usi[0]
#define M0_LW               DSP_Register_Set.name.m0.mm.usi[1]
#define M0_MSB              DSP_Register_Set.name.m0.mm.bb.msb

#define M1                  DSP_Register_Set.name.m1.mm.uli
#define M1_HW               DSP_Register_Set.name.m1.mm.usi[0]
#define M1_LW               DSP_Register_Set.name.m1.mm.usi[1]
#define M1_MSB              DSP_Register_Set.name.m1.mm.bb.msb

#define X0                  DSP_Register_Set.name.x0.mm.uli
#define X0_HW               DSP_Register_Set.name.x0.mm.usi[0]
#define X0_LW               DSP_Register_Set.name.x0.mm.usi[1]
#define X0_MSB              DSP_Register_Set.name.x0.mm.bb.msb

#define X1                  DSP_Register_Set.name.x1.mm.uli
#define X1_HW               DSP_Register_Set.name.x1.mm.usi[0]
#define X1_LW               DSP_Register_Set.name.x1.mm.usi[1]
#define X1_MSB              DSP_Register_Set.name.x1.mm.bb.msb

#define Y0                  DSP_Register_Set.name.y0.mm.uli
#define Y0_HW               DSP_Register_Set.name.y0.mm.usi[0]
#define Y0_LW               DSP_Register_Set.name.y0.mm.usi[1]
#define Y0_MSB              DSP_Register_Set.name.y0.mm.bb.msb

#define Y1                  DSP_Register_Set.name.y1.mm.uli
#define Y1_HW               DSP_Register_Set.name.y1.mm.usi[0]
#define Y1_LW               DSP_Register_Set.name.y1.mm.usi[1]
#define Y1_MSB              DSP_Register_Set.name.y1.mm.bb.msb

#define A0G                 DSP_Register_Set.name.a0g.mm.uli
#define A0G_HW              DSP_Register_Set.name.a0g.mm.usi[0]
#define A0G_LW              DSP_Register_Set.name.a0g.mm.usi[1]
```

```
#define A0G_LSB8            DSP_Register_Set.name.a0g.mm.cc.lsb8
#define A0G_MSB             DSP_Register_Set.name.a0g.mm.bb.g_msb

#define A1G                 DSP_Register_Set.name.a1g.mm.uli
#define A1G_HW              DSP_Register_Set.name.a1g.mm.usi[0]
#define A1G_LW              DSP_Register_Set.name.a1g.mm.usi[1]
#define A1G_LSB8            DSP_Register_Set.name.a1g.mm.cc.lsb8
#define A1G_MSB             DSP_Register_Set.name.a1g.mm.bb.g_msb

#define DSR DSP_Register_Set.name.dsr.uli
```

さらに DSR レジスタの各ビットも同様にユニオン DSP_Register_Set を使って以下のように定義します。

```
#define DSPGTBIT            DSP_Register_Set.name.dsr.a.gt
#define DSPZBIT             DSP_Register_Set.name.dsr.a.z
#define DSPNBIT             DSP_Register_Set.name.dsr.a.n
#define DSPVBIT             DSP_Register_Set.name.dsr.a.v
#define DSPCSBITS           DSP_Register_Set.name.dsr.a.cs
#define DSPDCBIT            DSP_Register_Set.name.dsr.a.dc
```

(2) ALU の入出力と演算結果を表す変数

ALU の入出力は下記の DSP_ALU_Set という名前のユニオンを基に定義します。このユニオンは 6 個のロングワードで構成されます。このうち 3 つのロングワードは 2 つの入力と 1 つの出力（src1、src2、dst）に対応します。残りの 3 つのロングワードはこれら 2 つの入力と 1 つの出力のガードビット用（src1g、src2g、dstg）です。

```
/* Union DSP_ALU_Set の定義 */

union {
        unsigned long int      uli[6];
        unsigned short int     usi[12];
        struct {
              struct {
                unsigned msb:  1;
                unsigned:      31;
              } src1, src2, dst;
              struct {
                union {
                        unsigned long int      uli;
                        struct {
                                unsigned:      24;
                                unsigned bit7: 1;
                                unsigned:      7;
                        } a;
                        struct {
                                unsigned:      24;
                                unsigned lsb8: 8;
                        } b;
                } u;
              } src1g, src2g, dstg;
        } n;

} DSP_ALU_Set;
```

上記のユニオン DSP_ALU_Set を用いて以下のように ALU 入出力の名称を定義します。

```
/* DSP 演算命令における ALU の入出力の定義 */

#define DSP_ALU_SRC1        DSP_ALU_Set.uli[0]
#define DSP_ALU_SRC2        DSP_ALU_Set.uli[1]
#define DSP_ALU_DST         DSP_ALU_Set.uli[2]

#define DSP_ALU_SRC1G       DSP_ALU_Set.uli[3]
#define DSP_ALU_SRC2G       DSP_ALU_Set.uli[4]
#define DSP_ALU_DSTG        DSP_ALU_Set.uli[5]

#define DSP_ALU_SRC1_HW     DSP_ALU_Set.usi[0]
#define DSP_ALU_SRC2_HW     DSP_ALU_Set.usi[2]
#define DSP_ALU_DST_HW      DSP_ALU_Set.usi[4]

#define DSP_ALU_SRC1_MSB    DSP_ALU_Set.n.src1.msb
#define DSP_ALU_SRC2_MSB    DSP_ALU_Set.n.src2.msb
#define DSP_ALU_DST_MSB     DSP_ALU_Set.n.dst.msb

#define DSP_ALU_SRC1G_BIT7  DSP_ALU_Set.n.src1g.u.a.bit7
#define DSP_ALU_SRC2G_BIT7  DSP_ALU_Set.n.src2g.u.a.bit7
#define DSP_ALU_DSTG_BIT7   DSP_ALU_Set.n.dstg.u.a.bit7

#define DSP_ALU_SRC1G_LSB8  DSP_ALU_Set.n.src1g.u.b.lsb8
#define DSP_ALU_SRC2G_LSB8  DSP_ALU_Set.n.src2g.u.b.lsb8
#define DSP_ALU_DSTG_LSB8   DSP_ALU_Set.n.dstg.u.b.lsb8
```

さらに演算結果を表す変数も上記の定義を使って以下のように定義します。これらの変数は各命令の動作内容の説明の中で DSR レジスタ内の DC ビットを計算するのに使われます。

```
/* DSP 演算結果を表す変数の定義 */

#define PLUS_OP_G_OV ((~DSP_ALU_SRC1G_BIT7 && ~DSP_ALU_SRC2G_BIT7 && DSP_ALU_DSTG_BIT7) ||
(DSP_ALU_SRC1G_BIT7 && DSP_ALU_SRC2G_BIT7 && ~DSP_ALU_DSTG_BIT7))

#define MINUS_OP_G_OV ((~DSP_ALU_SRC1G_BIT7 && DSP_ALU_SRC2G_BIT7 && DSP_ALU_DSTG_BIT7) ||
(DSP_ALU_SRC1G_BIT7 && ~DSP_ALU_SRC2G_BIT7 && ~DSP_ALU_DSTG_BIT7))

#define POS_NOT_OV ((DSP_ALU_DSTG_LSB8==0x00) && (DSP_ALU_DST_MSB==0x0))
#define NEG_NOT_OV ((DSP_ALU_DSTG_LSB8==0xff) && (DSP_ALU_DST_MSB==0x1))
```

(3) 乗算器の入出力

乗算器の入出力は下記の DSP_MUL_Set という名前のユニオンを基に定義します。このユニオンは 4 個のロングワードで構成されます。2 つの入力にはロングワードが 1 つずつ割り当てられていますが、両者とも上位 16 ビット（usi［0］、usi［2］）のみが使われます。出力にはガードビット用を含めた 2 つのロングワード（dst、dstg）が対応します。

```
* Union DSP_MUL_Set の定義  */

union {
        unsigned long int      uli[4];
```

```
        struct {
              unsigned short int  usi[4];
              struct {
                      unsigned msb:   1;
                      unsigned:       31;
              } dst;
              struct {
                      unsigned:       24;
                      unsigned lsb8:  8;
              } dstg;
        } aa;
} DSP_MUL_Set;
```

上記のユニオン DSP_MUL_Set を用いて以下のように乗算器入出力の名称を定義します。

```
/* DSP 演算命令における乗算器入出力の定義 */

#define DSP_M_SRC1          DSP_MUL_Set.aa.usi[0]
#define DSP_M_SRC2          DSP_MUL_Set.aa.usi[2]
#define DSP_M_DST           DSP_MUL_Set.uli[2]
#define DSP_M_DST_MSB       DSP_MUL_Set.aa.dst.msb
#define DSP_M_DSTG          DSP_MUL_Set.uli[3]
#define DSP_M_DSTG_LSB8     DSP_MUL_Set.aa.dstg.lsb8
```

(4) その他の命令動作内容説明で使われる変数など

DCT、DCF という条件が指定できる DSP 演算命令の動作内容説明のときには以下の変数を使用しています。

```
#define DSP_UNCONDITIONAL_UPDATE (!EX_DCT && !EX_DCF)
#define DSP_CONDITION_MATCH ((EX_DCT && DSPDCBIT) || (EX_DCF && !DSPDCBIT))
#define DSP_CONDITION_NOT_MATCH ((EX_DCT && !DSPDCBIT)||(EX_DCF && DSPDCBIT))
```

上記の定義において EX_DCT と EX_DCF はそれぞれ命令に DCT、DCF という条件が指定されている場合に真となる変数です。また、DSPDCBIT は**「（1）DSP レジスタの定義」**を参照のこと。

DSP の算術演算は SR レジスタの飽和ビットが 1 のとき、飽和処理を行います。この飽和ビットを動作内容説明の時には SBIT と呼びます。

さらに、動作内容説明の記述を簡略にするため、共通に用いられる以下の関数を定義します。

```
/* DSP 演算命令の説明で共通に使用される関数 */

unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

overflow_protection()
{
   if(SBIT && overflow_bit) { /* Overflow Protection Enable & overflow */
        if(DSP_ALU_DSTG_BIT7==0) { /* positive value */
              if((DSP_ALU_DSTG_LSB8!=0x0) || (DSP_ALU_DST_MSB!=0)) {
                DSP_ALU_DSTG= 0x0;
                DSP_ALU_DST = 0x7fffffff;
              }
        }
        else {                  /* negative value */
              if((DSP_ALU_DSTG_LSB8!=0xff) || (DSP_ALU_DST_MSB!=1)) {
```

```
                DSP_ALU_DSTG= 0xff;
                DSP_ALU_DST = 0x80000000;
              }
        }
        overflow_bit = 0; /* No more overflow when protected */
    }
}
```

下記の 6 個の関数は DSR レジスタの更新に使われます。DSR レジスタ内の DC ビットは、DSP 演算命令の演算結果と状態選択ビット（CS）の指示に従って更新されます。DSR レジスタ内の他のビットは DSP 演算命令の演算結果のみにに従って更新されます。

```
/* 無条件に DC ビット(DSPDCBIT)をボローフラグでアップデートする関数 */
dc_always_borrow()
{
        /* DC update policy: don't care the status of DSPCSBITS */
        DSPDCBIT        = borrow_bit;
        DSPGTBIT        = ~((negative_bit ^ overflow_bit) | zero_bit);
        DSPZBIT         = zero_bit;
        DSPNBIT         = negative_bit;
        DSPVBIT         = overflow_bit;
}

/* 無条件に DC ビット(DSPDCBIT)をキャリーフラグでアップデートする関数 */
dc_always_carry()
{
        /* DC update policy: don't care the status of DSPCSBITS */
        DSPDCBIT        = carry_bit;
        DSPGTBIT        = ~((negative_bit ^ overflow_bit) | zero_bit);
        DSPZBIT         = zero_bit;
        DSPNBIT         = negative_bit;
        DSPVBIT         = overflow_bit;
}

/* 引き算のとき、DC ビット(DSPDCBIT)をアップデートする関数 */
minus_dc_bit()
{
        switch (DSPCSBITS) {
              case 0x0:     /* Borrow Mode */
                      DSPDCBIT = borrow_bit;
                      break;
              case 0x1:     /* Negative Value Mode */
                      DSPDCBIT = negative_bit;
                      break;
              case 0x2:     /* Zero Value Mode */
                      DSPDCBIT = zero_bit;
                      break;
              case 0x3:     /* Overflow Mode */
                      DSPDCBIT = overflow_bit;
                      break;
              case 0x4:     /* Signed Greater Than Mode */
                      DSPDCBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
                      break;
              case 0x5:     /* Signed Greater Than or Equal Mode */
```

```
                        DSPDCBIT = ~(negative_bit ^ overflow_bit);
                        break;
                case 0x6:     /* Reserved */
                case 0x7:     /* Reserved */
                  break;
        }
        DSPGTBIT        = ~((negative_bit ^ overflow_bit) | zero_bit);
        DSPZBIT         = zero_bit;
        DSPNBIT         = negative_bit;
        DSPVBIT         = overflow_bit;
}

/* 加算のとき、DC ビット(DSPDCBIT)をアップデートする関数 */
plus_dc_bit()
{
        switch (DSPCSBITS) {
                case 0x0:     /* Carry Mode */
                        DSPDCBIT = carry_bit;
                        break;
                case 0x1:     /* Negative Value Mode */
                        DSPDCBIT = negative_bit;
                        break;
                case 0x2:     /* Zero Value Mode */
                        DSPDCBIT = zero_bit;
                        break;
                case 0x3:     /* Overflow Mode */
                        DSPDCBIT = overflow_bit;
                        break;
                case 0x4:     /* Signed Greater Than Mode */
                        DSPDCBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
                        break;
                case 0x5:     /* Signed Greater Than or Equal Mode */
                        DSPDCBIT = ~(negative_bit ^ overflow_bit);
                        break;
                case 0x6:     /* Reserved */
                case 0x7:     /* Reserved */
                  break;
        }
        DSPGTBIT        = ~((negative_bit ^ overflow_bit) | zero_bit);
        DSPZBIT         = zero_bit;
        DSPNBIT         = negative_bit;
        DSPVBIT         = overflow_bit;
}

/* 論理演算のとき、DC ビット(DSPDCBIT)をアップデートする関数 */
logical_dc_bit()
{
        switch (DSPCSBITS) {
                case 0x0:     /* Carry Mode */
                        DSPDCBIT = 0;
                        break;
                case 0x1:     /* Negative Value Mode */
                        DSPDCBIT = negative_bit;
                        break;
                case 0x2:     /* Zero Value Mode */
```

```
                DSPDCBIT = zero_bit;
                break;
        case 0x3:     /* Overflow Mode */
                DSPDCBIT = 0;
                break;
        case 0x4:     /* Signed Greater Than Mode */
                DSPDCBIT = 0;
                break;
        case 0x5:     /* Signed Greater Than or Equal Mode */
                DSPDCBIT = 0;
                break;
        case 0x6:     /* Reserved */
        case 0x7:     /* Reserved */
          break;
    }
    DSPGTBIT       = 0;
    DSPZBIT        = zero_bit;
    DSPNBIT        = negative_bit;
    DSPVBIT        = 0;
}

shift_dc_bit()
{
    switch (DSPCSBITS) {
        case 0x0:     /* Carry Mode */
                DSPDCBIT = carry_bit;
                break;
        case 0x1:     /* Negative Value Mode */
                DSPDCBIT = negative_bit;
                break;
        case 0x2:     /* Zero Value Mode */
                DSPDCBIT = zero_bit;
                break;
        case 0x3:     /* Overflow Mode */
                DSPDCBIT = overflow_bit;
                break;
        case 0x4:     /* Signed Greater Than Mode */
                DSPDCBIT = 0;
                break;
        case 0x5:     /* Signed Greater Than or Equal Mode */
                DSPDCBIT = 0;
                break;
        case 0x6:     /* Reserved */
        case 0x7:     /* Reserved */
          break;
    }
    DSPGTBIT       = 0;
    DSPZBIT        = zero_bit;
    DSPNBIT        = negative_bit;
    DSPVBIT        = overflow_bit;
}
```

DSP 演算命令の詳細命令を、アルファベット順に示します。

6.3.1 PABS ABSolute DSP 算術演算命令

絶対値演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PABS Sx,Dz	**もし** Sx≧0 **ならば** Sx→Dz **もし** Sx<0 **ならば** 0−Sx→Dz	111110********** 10001000xx00zzzz	1	**更新**	−	−	○
PABS Sy,Dz	**もし** Sy≧0 **ならば** Sy→Dz **もし** Sy<0 **ならば** 0−Sy→Dz	111110********** 1010100000yyzzzz	1	**更新**	−	−	○

(1) 説明

絶対値を求めます。もし Sx、Sy オペランドの内容が正の値のときは Sx、Sy オペランドの内容を Dz オペランドへ格納します。もし負の値のときは符号を反転して Dz オペランドへ格納します。

DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。

(2) 動作内容

```
/*      Case1 : PABS Sx,Dz    */
/*      Case2 : PABS Sy,Dz    */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit, borrow_bit;

/* ALU Sources assignment */

        DSP_ALU_SRC1 = 0;
        DSP_ALU_SRC1G= 0;

        if (Case1) {           /* PABS Sx,Dz */
                switch (xx) {          /* Sx Operand selection bit (xx) */

                   case 0x0:   DSP_ALU_SRC2  = X0;
                               if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
                               else                   DSP_ALU_SRC2G = 0x0;
                               break;
                   case 0x1:   DSP_ALU_SRC2  = X1;
                               if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
                               else                   DSP_ALU_SRC2G = 0x0;
                               break;
                   case 0x2:   DSP_ALU_SRC2  = A0;
                               DSP_ALU_SRC2G = A0G;
                               break;
                   case 0x3:   DSP_ALU_SRC2  = A1;
                               DSP_ALU_SRC2G = A1G;
                               break;
                }
        }
        else  {         /* PABS Sy,Dz */
```

```
        switch (yy) {
           case 0x0:   DSP_ALU_SRC2  = Y0;
                       break;
           case 0x1:   DSP_ALU_SRC2  = Y1;
                       break;
           case 0x2:   DSP_ALU_SRC2  = M0;
                       break;
           case 0x3:   DSP_ALU_SRC2  = M1;
                       break;
        }
        if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
        else                   DSP_ALU_SRC2G = 0x0;
}

/* ALU Operation */
if(DSP_ALU_SRC2G_BIT7==0) {    /* positive value */
        DSP_ALU_DST = 0x0 + DSP_ALU_SRC2;
        carry_bit = 0;
        DSP_ALU_DSTG_LSB8= 0x0 + DSP_ALU_SRC2G_LSB8 + carry_bit;
}
else {                         /* negative value */
        DSP_ALU_DST = 0x0 - DSP_ALU_SRC2;
        borrow_bit = 1;
        DSP_ALU_DSTG_LSB8= 0x0 - DSP_ALU_SRC2G_LSB8 - borrow_bit;
}

overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

overflow_protection();

/* ALU Destination assignment */
switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
        case 0x5:      A1 = DSP_ALU_DST;
                       A1G = DSP_ALU_DSTG & 0x000000FF;
                       if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
        case 0x7:      A0 = DSP_ALU_DST;
                       A0G = DSP_ALU_DSTG & 0x000000FF;
                       if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
        case 0x8:      X0 = DSP_ALU_DST;
        break;
        case 0x9:      X1 = DSP_ALU_DST;
        break;
        case 0xa:      Y0 = DSP_ALU_DST;
        break;
        case 0xb:      Y1 = DSP_ALU_DST;
        break;
        case 0xc:      M0 = DSP_ALU_DST;
        break;
        case 0xe:      M1 = DSP_ALU_DST;
        break;
        default:       printf("\nERROR:Illegal DSPInstruction"); break;
}
```

```
        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        if(DSP_ALU_SRC2G_BIT7==0) {
                plus_dc_bit();
        }
        else    {
                overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
                minus_dc_bit();
        }
}
```

(3) **使用例**

```
  PABS X0,M0  NOPX  NOPY  ;実行前：X0=H'33333333, M0=H'12345678
                           実行後：X0=H'33333333, M0=H'33333333
  PABS X1,X1  NOPX  NOPY  ;実行前：X1=H'DDDDDDDD
                           実行後：X1=H'22222223
                           DC ビットは CS［2:0］の状態に従って更新。
```

6.3.2 [if cc] PADD ADDition with Condition DSP 算術演算命令

条件付き加算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PADD Sx,Sy,Dz	Sx＋Sy→Dz	111110**********	1	更新	−	−	○
		10110001xxyyzzzz					
DCT PADD Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx＋Sy→Dz	111110**********	1	−	−	−	○
	もし 0 **ならば** nop.	10110010xxyyzzzz					
DCF PADD Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx＋Sy→Dz	111110**********					
	もし 1 **ならば** nop.	10110011xxyyzzzz	1	−	−	−	○

(1) 説明

Sx、Sy オペランドの内容を加算し、その結果を Dz オペランドへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合でも、DC、N、Z、V、GT ビットは更新されません。

(2) 動作内容

```
/*      PADD Sx,Sy,Dz    */
{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
switch (xx) {          /* Sx Operand selection bit (xx) */
        case 0x0:      DSP_ALU_SRC1  = X0;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x1:      DSP_ALU_SRC1  = X1;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x2:      DSP_ALU_SRC1  = A0;
                       DSP_ALU_SRC1G = A0G;
                       break;
        case 0x3:      DSP_ALU_SRC1  = A1;
                       DSP_ALU_SRC1G = A1G;
                       break;
}
switch (yy) {          /* Sy Operand selection bit (yy) */
        case 0x0:      DSP_ALU_SRC2  = Y0;
                       break;
        case 0x1:      DSP_ALU_SRC2  = Y1;
```

```
                        break;
        case 0x2:       DSP_ALU_SRC2  = M0;
                        break;
        case 0x3:       DSP_ALU_SRC2  = M1;
                        break;
}

if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
else                   DSP_ALU_SRC2G = 0x0;

/* ALU Operation */
DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2;

carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU_DST_MSB) |
               (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);

DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit

overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

overflow_protection();

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1 = DSP_ALU_DST;
                               A1G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:      A0 = DSP_ALU_DST;
                               A0G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:      X0 = DSP_ALU_DST;
                break;
                case 0x9:      X1 = DSP_ALU_DST;
                break;
                case 0xa:      Y0 = DSP_ALU_DST;
                break;
                case 0xb:      Y1 = DSP_ALU_DST;
                break;
                case 0xc:      M0 = DSP_ALU_DST;
                break;
                case 0xe:      M1 = DSP_ALU_DST;
                break;
                default:       printf("\nERROR:Illegal DSPInstruction"); break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        plus_dc_bit();
```

```
}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = DSP_ALU_DST;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = DSP_ALU_DST;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = DSP_ALU_DST;
                break;
                case 0x9:       X1 = DSP_ALU_DST;
                break;
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
}
}
```

(3) **使用例**

```
  PADD X0,Y0,A0  NOPX  NOPY   ;実行前：X0=H'22222222, Y0=H'33333333,
                                        A0=H'123456789A
                              実行後：X0=H'22222222, Y0=H'33333333,
                                        A0=H'0055555555
```

無条件実行の場合、DC ビットは演算直前の CS［2:0］ビットの状態に従って更新。

6.3.3 PADD ADDition & MULtiply PMULS Signed by Signed DSP 算術演算命令

加算と符号付き乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx＋Sy→Du Se の上位ワード×Sf の上位ワード→Dg	111110********** 0111eeffxxyygguu	1	更新	ー	ー	○

(1) 説明

Sx、Sy オペランドの内容を加算し、結果を Du オペランドへ格納します。Se、Sf オペランドの上位ワードの内容を符号付きとして乗算し、結果を Dg オペランドに格納します。この 2 つの処理は同時に並行して実行されます。

DSR レジスタの DC ビットは ALU 演算の結果と CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも ALU 演算の結果に従って更新されます。

(2) 注意

PMULS は固定小数点乗算ですので、ソースデータが同じでも MULS とは演算結果が異なります。

(3) 動作内容

```
/*      PADD Sx,Sy,Du   PMULS Se,Sf,Dg  */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* Multiplier Sources assignment */
        switch (ee) {           /* Se Operand selection bit (ee) */
                case 0x0:       DSP_M_SRC1 = X0_HW;
                                break;
                case 0x1:       DSP_M_SRC1 = X1_HW;
                                break;
                case 0x2:       DSP_M_SRC1 = Y0_HW;
                                break;
                case 0x3:       DSP_M_SRC1 = A1_HW;
                                break;
        }
        switch (ff) {           /* Sf Operand selection bit (ff) */
                case 0x0:       DSP_M_SRC2 = Y0_HW;
                                break;
                case 0x1:       DSP_M_SRC2 = Y1_HW;
                                break;
                case 0x2:       DSP_M_SRC2 = X0_HW;
                                break;
                case 0x3:       DSP_M_SRC2 = A1_HW;
                                break;
        }

/* ALU Sources assignment */
```

```
        switch (xx) {          /* Sx Operand selection bit (xx) */
                case 0x0:      DSP_ALU_SRC1  = X0;
                               if (DSP_ALU_SRC1_MSB)
                                       DSP_ALU_SRC1G_LSB8 = 0xff;
                               else    DSP_ALU_SRC1G_LSB8 = 0x0;
                               break;
                case 0x1:      DSP_ALU_SRC1  = X1;
                               if (DSP_ALU_SRC1_MSB)
                                       DSP_ALU_SRC1G_LSB8 = 0xff;
                               else    DSP_ALU_SRC1G_LSB8 = 0x0;
                               break;
                case 0x2:      DSP_ALU_SRC1  = A0;
                               DSP_ALU_SRC1G = A0G;
                               break;
                case 0x3:      DSP_ALU_SRC1  = A1;
                               DSP_ALU_SRC1G = A1G;
                               break;
        }
        switch (yy) {          /* Sy Operand selection bit (yy) */
                case 0x0:      DSP_ALU_SRC2  = Y0;
                               break;
                case 0x1:      DSP_ALU_SRC2  = Y1;
                               break;
                case 0x2:      DSP_ALU_SRC2  = M0;
                               break;
                case 0x3:      DSP_ALU_SRC2  = M1;
                               break;
        }
        if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G_LSB8 = 0xff;
        else                   DSP_ALU_SRC2G_LSB8 = 0x0;

/* Multiplier Operation */

        /* PMULS Se, Sf, Dg */
        if ((SBIT==1) && (DSP_M_SRC1==0x8000) && (DSP_M_SRC2==0x8000)) {
              DSP_M_DST=0x7fffffff;  /* overflow protection */
        }
        else {
              DSP_M_DST=((long)(short)DSP_M_SRC1*(long)(short)DSP_M_SRC2)<<1;
        }
        if (DSP_M_DST_MSB) DSP_M_DSTG_LSB8 = 0xff;
        else    DSP_M_DSTG_LSB8 = 0x0;

        switch (gg) {  /* Dg Operand selection bit (gg) */
                case 0x0:      M0 = DSP_M_DST;
                               break;
                case 0x1:      M1 = DSP_M_DST;
                               break;
                case 0x2:      A0 = DSP_M_DST;
                               if(DSP_M_DSTG_LSB8==0x0) A0G=0x0;
                               else A0G=0xffffffff;
                               break;
                case 0x3:      A1 = DSP_M_DST;
                               if(DSP_M_DSTG_LSB8==0x0) A1G=0x0;
                               else A1G=0xffffffff;
```

```
                        break;
        }

/* ALU operation */

        DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2;
        carry_bit=((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU_DST_MSB) |
                (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
        DSP_ALU_DSTG_LSB8=DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;

        overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

        overflow_protection();

        switch (uu) {  /* Du Operand selection bit (uu) */
                case 0x0:
                        X0  = DSP_ALU_DST;
                        negative_bit = DSP_ALU_DST_MSB;
                        zero_bit = (DSP_ALU_DST==0);
                        break;
                case 0x1:
                        Y0  = DSP_ALU_DST;
                        negative_bit = DSP_ALU_DST_MSB;
                        zero_bit = (DSP_ALU_DST==0);
                        break;
                case 0x2:
                        A0  = DSP_ALU_DST;
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                        negative_bit = DSP_ALU_DSTG_BIT7;
                        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
                        break;
                case 0x3:
                        A1  = DSP_ALU_DST;
                        A1G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                        negative_bit = DSP_ALU_DSTG_BIT7;
                        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
                        break;
        }

        /* DSR register update */
        plus_dc_bit();

}
```

(4) **使用例**

```
PADD A0,M0,A0 PMULS X0,Y0,M0  NOPX  NOPY;
                                実行前：X0=H'00020000, Y0=H'00030000,
                                        M0=H'22222222, A0=H'0055555555
                                実行後：X0=H'00020000, Y0=H'00030000,
                                        M0=H'0000000C, A0=H'0077777777
```

DC ビットは CS［2:0］の状態に従って PADD 動作の結果に基いて更新。

6.3.4 PADDC ADDition with Carry DSP 算術演算命令

キャリ付き加算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PADDC Sx,Sy,Dz	Sx＋Sy＋DC→Dz	111110********** 10110000xxyyzzzz	1	キャリ	ー	ー	○

(1) 説明

Sx、Sy オペランドの内容と DC ビットを加算し、Dz オペランドへ格納します。

DSR レジスタの DC ビットはキャリフラグとして更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。

(2) 注意

PADDC 命令実行後の DC ビットは、CS ビットに関係なく、キャリフラグとして更新されます。

(3) 動作内容

```
/*      PADDC Sx,Sy,Dz    */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
        switch (xx) {           /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                                else                   DSP_ALU_SRC1G = 0x0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                                if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                                else                   DSP_ALU_SRC1G = 0x0;
                                break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                                DSP_ALU_SRC1G = A0G;
                                break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                                DSP_ALU_SRC1G = A1G;
                                break;
        }
        switch (yy) {           /* Sy Operand selection bit (yy) */
                case 0x0:       DSP_ALU_SRC2  = Y0;
                                break;
                case 0x1:       DSP_ALU_SRC2  = Y1;
                                break;
                case 0x2:       DSP_ALU_SRC2  = M0;
                                break;
                case 0x3:       DSP_ALU_SRC2  = M1;
                                break;
        }
        if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
```

```
        else                    DSP_ALU_SRC2G = 0x0;

/* ALU Operation */
DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2 + DSPDCBIT;

carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU_DST_MSB)
               |(DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);

DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;

overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
overflow_protection();

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = DSP_ALU_DST;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = DSP_ALU_DST;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = DSP_ALU_DST;
                break;
                case 0x9:       X1 = DSP_ALU_DST;
                break;
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        dc_always_carry();
}
```

(4) **使用例**

CS[2:0]=***:CS ビットの状態に関係なく、常に Carry or Borrow Mode として動作します。

```
PADDC X0,Y0,M0  NOPX  NOPY ;          実行前: X0=H'B3333333, Y0=H'55555555
                                              M0=H'12345678, DC=0
                                      実行後: X0=H'B3333333, Y0=H'55555555
                                              M0=H'08888888, DC=1
PADDC X0,Y0,M0  NOPX  NOPY ;          実行前: X0=H'33333333, Y0=H'55555555
                                              M0=H'12345678, DC=1
                                      実行後: X0=H'33333333, Y0=H'55555555
                                              M0=H'88888889, DC=0
```

DC ビットは、CS[2:0]ビットの状態に従って更新。

6.3.5 [if cc] PAND logical AND DSP 論理演算命令

条件付き論理積演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PAND Sx,Sy,Dz	Sx & Sy→Dz、Dz の下位ワードクリア	111110********** 10010101xxyyzzzz	1	更新	—	—	○
DCT PAND Sx,Sy,Dz	もし DC＝1 ならば Sx&Sy→Dz、	111110**********	1	—	—	—	○
	Dz の下位ワードクリア もし 0 ならば nop.	10010110xxyyzzzz					
DCF PAND Sx,Sy,Dz	もし DC＝0 ならば Sx&Sy→Dz、	111110**********	1	—	—	—	○
	Dz の下位ワードクリア もし 1 ならば nop.	10010111xxyyzzzz					

(1) 説明

Sx オペランドの上位ワードの内容と Sy オペランドの上位ワードの内容との論理積を演算し、その結果を Dz オペランドの上位ワードへ格納し、Dz オペランドの下位ワードを 0 でクリアします。Dz がガードビットを持つレジスタの場合は、ガードビットも 0 でクリアします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合でも、DC、N、Z、V、GT ビットは更新されません。

(2) 注意

デスティネーションレジスタの下位ワードの内容とガードビットの内容は DC ビットの更新には無視されます。

(3) 動作内容

```
/*      PAND Sx,Sy,Dz    */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
        switch (xx) {           /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                                break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                                break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                                break;
        }
        switch (yy) {           /* Sy Operand selection bit (yy) */
```

```
            case 0x0:       DSP_ALU_SRC2  = Y0;
                            break;
            case 0x1:       DSP_ALU_SRC2  = Y1;
                            break;
            case 0x2:       DSP_ALU_SRC2  = M0;
                            break;
            case 0x3:       DSP_ALU_SRC2  = M1;
                            break;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW & DSP_ALU_SRC2_HW;

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

    /* ALU Destination assignment */
    switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
            case 0x5:       A1_HW = DSP_ALU_DST_HW;
                            A1_LW = 0x0;            /* clear LSW */
                            A1G = 0x0;              /* clear Guard bits */
            break;
            case 0x7:       A0_HW = DSP_ALU_DST_HW;
                            A0_LW = 0x0;            /* clear LSW */
                            A0G = 0x0;              /* clear Guard bits */
            break;
            case 0x8:       X0_HW = DSP_ALU_DST_HW;
                            X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:       X1_HW = DSP_ALU_DST;
                            X1_LW = 0x0;            /* clear LSW */
            break;
            case 0xa:       Y0_HW = DSP_ALU_DST;
                            Y0_LW = 0x0;            /* clear LSW */
            break;
            case 0xb:       Y1_HW = DSP_ALU_DST;
                            Y1_LW = 0x0;            /* clear LSW */
            break;
            case 0xc:       M0_HW = DSP_ALU_DST;
                            M0_LW = 0x0;            /* clear LSW */
            break;
            case 0xe:       M1_HW = DSP_ALU_DST;
                            M1_LW = 0x0;            /* clear LSW */
            break;
            default:        printf("\nERROR:Illegal DSPInstruction"); break;
    }

            carry_bit       = 0x0;
            negative_bit    = DSP_ALU_DST_MSB;
            zero_bit        = (DSP_ALU_DST_HW==0);
            overflow_bit    = 0x0;

            /* DSR register update */
            logical_dc_bit();
}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
    /* ALU Destination assignment */
```

```
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1_HW = DSP_ALU_DST_HW;
                                A1_LW = 0x0;            /* clear LSW */
                                A1G = 0x0;              /* clear Guard bits */
                break;
                case 0x7:       A0_HW = DSP_ALU_DST_HW;
                                A0_LW = 0x0;            /* clear LSW */
                                A0G = 0x0;              /* clear Guard bits */
                break;
                case 0x8:       X0_HW = DSP_ALU_DST_HW;
                                X0_LW = 0x0;            /* clear LSW */
                break;
                case 0x9:       X1_HW = DSP_ALU_DST;
                                X1_LW = 0x0;            /* clear LSW */
                break;
                case 0xa:       Y0_HW = DSP_ALU_DST;
                                Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:       Y1_HW = DSP_ALU_DST;
                                Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:       M0_HW = DSP_ALU_DST;
                                M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:       M1_HW = DSP_ALU_DST;
                                M1_LW = 0x0;            /* clear LSW */
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }

                }
}
```

(4) **使用例**

```
  PAND X0,Y0,A0  NOPX  NOPY ;          実行前: X0=H'33333333, Y0=H'55555555
                                               A0=H'123456789A
                                       実行後: X0=H'33333333, Y0=H'55555555
                                               A0=H'0011110000
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.6 [if cc] PCLR CLeaR DSP 算術演算命令

条件付きクリア

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PCLR Dz	H'00000000→Dz	111110********** 100011010000zzzz	1	更新	−	−	○
DCT PCLR Dz	**もし** DC＝1 **ならば** H'00000000→Dz **もし** 0 **ならば** nop.	111110********** 100011100000zzzz	1	−	−	−	○
DCF PCLR Dz	**もし** DC＝0 **ならば** H'00000000→Dz **もし** 1 **ならば** nop.	111110********** 100011110000zzzz	1	−	−	−	○

(1) **説明**

Dz オペランドの内容を 0 でクリアします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの Z ビットは 1 にセットされます。N、V、GT ビットは 0 にクリアされます。条件が指定されている場合は、条件が真で命令が実行された場合でも、DC、N、Z、V、GT ビットは更新されません。

(2) **動作内容**

```
/*      PCLR Dz     */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

   if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

       /* ALU Destination assignment */
       switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
              case 0x5:       A1 = 0x0;
                              A1G = 0x0;
              break;
              case 0x7:       A0 = 0x0;
                              A0G = 0x0;
              break;
              case 0x8:       X0 = 0x0;
              break;
              case 0x9:       X1 = 0x0;
              break;
              case 0xa:       Y0 = 0x0;
              break;
              case 0xb:       Y1 = 0x0;
              break;
              case 0xc:       M0 = 0x0;
              break;
```

```
                case 0xe:       M1 = 0x0;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }

                carry_bit       = 0;
                negative_bit    = 0;
                zero_bit        = 1;
                overflow_bit    = 0;

                /* DSR register update */
                plus_dc_bit();
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = 0x0;
                                A1G = 0x0;
                break;
                case 0x7:       A0 = 0x0;
                                A0G = 0x0;
                break;
                case 0x8:       X0 = 0x0;
                break;
                case 0x9:       X1 = 0x0;
                break;
                case 0xa:       Y0 = 0x0;
                break;
                case 0xb:       Y1 = 0x0;
                break;
                case 0xc:       M0 = 0x0;
                break;
                case 0xe:       M1 = 0x0;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(3) **使用例**

```
  PCLR A0  NOPX  NOPY ;
```

実行前:A0=H'FF87654321
実行後:A0=H'0000000000
無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.7 PCMP CoMPare two data DSP 算術演算命令

比較

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PCMP Sx,Sy	Sx－Sy	111110********** 10000100xxyy0000	1	更新	－	－	○

(1) 説明

Sx オペランドの内容から Sy オペランドの内容を減算します。

DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。

(2) 動作内容

```
/*      PCMP Sx,Sy    */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
        switch (xx) {          /* Sx Operand selection bit (xx) */
                case 0x0:      DSP_ALU_SRC1  = X0;
                               if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                               else                   DSP_ALU_SRC1G = 0x0;
                               break;
                case 0x1:      DSP_ALU_SRC1  = X1;
                               if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                               else                   DSP_ALU_SRC1G = 0x0;
                               break;
                case 0x2:      DSP_ALU_SRC1  = A0;
                               DSP_ALU_SRC1G = A0G;
                               break;
                case 0x3:      DSP_ALU_SRC1  = A1;
                               DSP_ALU_SRC1G = A1G;
                               break;
        }
        switch (yy) {          /* Sy Operand selection bit (yy) */
                case 0x0:      DSP_ALU_SRC2  = Y0;
                               break;
                case 0x1:      DSP_ALU_SRC2  = Y1;
                               break;
                case 0x2:      DSP_ALU_SRC2  = M0;
                               break;
                case 0x3:      DSP_ALU_SRC2  = M1;
                               break;
        }
        if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
        else                   DSP_ALU_SRC2G = 0x0;

        DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
```

```
        carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB) |
        (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
        borrow_bit = !carry_bit;
        DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8
                          - borrow_bit;

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
        overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

        overflow_protection();

        /* DSR register update */
        minus_dc_bit();

}
```

(3) **使用例**

```
  PCMP X0,Y0  NOPX  NOPY          ;実行前:X0=H'22222222, Y0=H'33333333
                                   実行後:X0=H'22222222, Y0=H'33333333
                                          N=1,Z=0,V=0,GT=0
                                   DC ビットは CS［2:0］の状態に従って更新。
```

6.3.8 [if cc] PCOPY　COPY with Condition　DSP 算術演算命令

条件付き転記

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PCOPY Sx,Dz	Sx→Dz	111110********** 11011001xx00zzzz	1	更新	−	−	○
PCOPY Sy,Dz	Sy→Dz	111110********** 1111100100yyzzzz	1	更新	−	−	○
DCT PCOPY Sx,Dz	**もし** DC＝1 **ならば** Sx→Dz **もし** 0 **ならば** nop.	111110********** 11011010xx00zzzz	1	−	−	−	○
DCT PCOPY Sy,Dz	**もし** DC＝1 **ならば** Sy→Dz **もし** 0 **ならば** nop.	111110********** 1111101000yyzzzz	1	−	−	−	○
DCF PCOPY Sx,Dz	**もし** DC＝0 **ならば** Sx→Dz **もし** 1 **ならば** nop.	111110********** 11011011xx00zzzz	1	−	−	−	○
DCF PCOPY Sy,Dz	**もし** DC＝0 **ならば** Sy→Dz **もし** 1 **ならば** nop	111110********** 1111101100yyzzzz	1	−	−	−	○

(1) **説明**

Sx、Sy オペランドの内容を Dz オペランドへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合でも、DC、N、Z、V、GT ビットは更新されません。

(2) **動作内容**

```
/*      Case1 : PCOPY Sx,Dz     */
/*      Case2 : PCOPY Sy,Dz     */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */

        if (Case1) {            /* PCOPY Sx,Dz */
                switch (xx) {           /* Sx Operand selection bit (xx) */
                    case 0x0:   DSP_ALU_SRC1  = X0;
                                if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                                else                   DSP_ALU_SRC1G = 0x0;
                                break;
                    case 0x1:   DSP_ALU_SRC1  = X1;
                                if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                                else                   DSP_ALU_SRC1G = 0x0;
                                break;
                    case 0x2:   DSP_ALU_SRC1  = A0;
                                DSP_ALU_SRC1G = A0G;
                                break;
                    case 0x3:   DSP_ALU_SRC1  = A1;
```

```
                        DSP_ALU_SRC1G = A1G;
                        break;
                }
                DSP_ALU_SRC2 = 0;
                DSP_ALU_SRC2G= 0;
        }
        else  {         /* PCOPY Sy,Dz */
                DSP_ALU_SRC1 = 0;
                DSP_ALU_SRC1G= 0;

                switch (yy) {
                   case 0x0:   DSP_ALU_SRC2  = Y0;
                               break;
                   case 0x1:   DSP_ALU_SRC2  = Y1;
                               break;
                   case 0x2:   DSP_ALU_SRC2  = M0;
                               break;
                   case 0x3:   DSP_ALU_SRC2  = M1;
                               break;
                }
                if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
                else                   DSP_ALU_SRC2G = 0x0;
        }

        DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2;
        carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU_DST_MSB) |
                        (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);

        DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit

        overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
        overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = DSP_ALU_DST;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = DSP_ALU_DST;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = DSP_ALU_DST;
                break;
                case 0x9:       X1 = DSP_ALU_DST;
                break;
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
```

```
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        plus_dc_bit();

    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = DSP_ALU_DST;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = DSP_ALU_DST;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = DSP_ALU_DST;
                break;
                case 0x9:       X1 = DSP_ALU_DST;
                break;
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(3) **使用例**

```
  PCOPY X0,A0  NOPX  NOPY                ;実行前:X0=H'55555555, A0=H'FFFFFFFFFF
                                          実行後:X0=H'55555555, A0=H'0055555555
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.9 [if cc] PDEC DECrement by 1 DSP 算術演算命令

条件付きデクリメント

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PDEC Sx,Dz	Sx の上位ワード－1→Dz の上位ワード、Dz の下位ワードをクリア	111110********** 10001001xx00zzzz	1	更新	―	―	○
PDEC Sy,Dz	Sy の上位ワード－1→Dz の上位ワード、Dz の下位ワードをクリア	111110********** 1010100100yyzzzz	1	更新	―	―	○
DCT PDEC Sx,Dz	もし DC＝1 ならば Sx の上位ワード－1→Dz の上位ワード、Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10001010xx00zzzz	1	―	―	―	○
DCT PDEC Sy,Dz	もし DC＝1 ならば Sy の上位ワード－1→Dz の上位ワード、Dz の下位ワードクリア もし 0 ならば nop.	111110********** 1010101000yyzzzz	1	―	―	―	○
DCF PDEC Sx,Dz	もし DC＝0 ならば Sx の上位ワード－1→Dz の上位ワード、Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10001011xx00zzzz	1	―	―	―	○
DCF PDEC Sy,Dz	もし DC＝0 ならば Sy の上位ワード－1→Dz の上位ワード、Dz の下位ワードクリア もし 1 ならば nop.	111110********** 1010101100yyzzzz	1	―	―	―	○

(1) **説明**

Sx、Sy オペランドの上位ワードの内容から 1 を減算し、その結果を Dz オペランドの上位ワードへ格納し、Dz オペランドの下位ワードを 0 でクリアします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合でも、DC、N、Z、V、GT ビットは更新されません。

(2) **注意**

デスティネーションレジスタの下位ワードの内容は DC ビットの更新には無視されます。

(3) 動作内容

```
/*      Case1 : PDEC Sx,Dz    */
/*      Case2 : PDEC Sy,Dz    */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */

        DSP_ALU_SRC2 = 0x1;
        DSP_ALU_SRC2G= 0x0;

        if (Case1) {           /* MSW of Sx -1 -> Dz */
           switch (xx) {       /* Sx Operand selection bit (xx) */
                case 0x0:      DSP_ALU_SRC1  = X0;
                               if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                               else                   DSP_ALU_SRC1G = 0x0;
                               break;
                case 0x1:      DSP_ALU_SRC1  = X1;
                               if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                               else                   DSP_ALU_SRC1G = 0x0;
                               break;
                case 0x2:      DSP_ALU_SRC1  = A0;
                               DSP_ALU_SRC1G = A0G;
                               break;
                case 0x3:      DSP_ALU_SRC1  = A1;
                               DSP_ALU_SRC1G = A1G;
                               break;
           }
        }
        else {                 /* MSW of Sy -1 -> Dz */
           switch (yy) {       /* Sy Operand selection bit (yy) */
                case 0x0:      DSP_ALU_SRC1  = Y0;
                               break;
                case 0x1:      DSP_ALU_SRC1  = Y1;
                               break;
                case 0x2:      DSP_ALU_SRC1  = M0;
                               break;
                case 0x3:      DSP_ALU_SRC1  = M1;
                               break;
           }
           if (DSP_ALU_SRC1_MSB)      DSP_ALU_SRC1G = 0xff;
           else                       DSP_ALU_SRC1G = 0x0;
        }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW - 1;
    carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB) |
                (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

    overflow_protection();
```

```
if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

            /* ALU Destination assignment */
    switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
            case 0x5:      A1_HW = DSP_ALU_DST_HW;
                           A1_LW = 0x0;            /* clear LSW */
                           A1G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
            case 0x7:      A0_HW = DSP_ALU_DST_HW;
                           A0_LW = 0x0;            /* clear LSW */
                           A0G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
            case 0x8:      X0_HW = DSP_ALU_DST_HW;
                           X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:      X1_HW = DSP_ALU_DST_HW;
                           X1_LW = 0x0;            /* clear LSW */
            break;
            case 0xa:      Y0_HW = DSP_ALU_DST_HW;
                           Y0_LW = 0x0;            /* clear LSW */
            break;
            case 0xb:      Y1_HW = DSP_ALU_DST_HW;
                           Y1_LW = 0x0;            /* clear LSW */
            break;
            case 0xc:      M0_HW = DSP_ALU_DST_HW;
                           M0_LW = 0x0;            /* clear LSW */
            break;
            case 0xe:      M1_HW = DSP_ALU_DST_HW;
                           M1_LW = 0x0;            /* clear LSW */
            break;
            default:       printf("\nERROR:Illegal DSPInstruction"); break;
    }
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    minus_dc_bit.c"
}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

            /* ALU Destination assignment */
    switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
            case 0x5:      A1_HW = DSP_ALU_DST_HW;
                           A1_LW = 0x0;            /* clear LSW */
                           A1G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
            case 0x7:      A0_HW = DSP_ALU_DST_HW;
                           A0_LW = 0x0;            /* clear LSW */
                           A0G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
            case 0x8:      X0_HW = DSP_ALU_DST_HW;
```

```
                        X0_LW = 0x0;            /* clear LSW */
                break;
                case 0x9:       X1_HW = DSP_ALU_DST_HW;
                        X1_LW = 0x0;            /* clear LSW */
                break;
                case 0xa:       Y0_HW = DSP_ALU_DST_HW;
                        Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:       Y1_HW = DSP_ALU_DST_HW;
                        Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:       M0_HW = DSP_ALU_DST_HW;
                        M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:       M1_HW = DSP_ALU_DST_HW;
                        M1_LW = 0x0;            /* clear LSW */
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(4) **使用例**

```
  PDEC X0,M0  NOPX  NOPY  ;
  PDEC X1,X1  NOPX  NOPY  ;
```

実行前: X0=H'0052330F, M0=H'12345678
実行後: X0=H'0052330F, M0=H'00510000
実行前: X1=H'FC342855
実行後: X1=H'FC330000
無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.10 [if cc] PDMSB Detect MSB with Condition DSP 算術演算命令

条件付き MSB **検出**

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PDMSB Sx,Dz	Sx データの MSB 位置→Dz の上位ワード、Dz の下位ワードクリア	111110********** 10011101xx00zzzz	1	更新	−	−	○
PDMSB Sy,Dz	Sy データの MSB 位置→Dz の上位ワード、Dz の下位ワードクリア	111110********** 1011110100yyzzzz	1	更新	−	−	○
DCT PDMSB Sx,Dz	もし DC＝1 ならば Sx データの MSB 位置→Dz の上位ワード、Dz の下位ワードクリア もし 0 ならば nop	111110********** 10011110xx00zzzz	1	−	−	−	○
DCT PDMSB Sy,Dz	もし DC＝1 ならば Sy データの MSB 位置→Dz の上位ワード、Dz の下位ワードクリア もし 0 ならば nop	111110********** 1011111000yyzzzz	1	−	−	−	○
DCF PDMSB Sx,Dz	もし DC＝0 ならば Sx データの MSB 位置→Dz の上位ワード、Dz の下位ワードクリア もし 1 ならば nop	111110********** 10011111xx00zzzz	1	−	−	−	○
DCF PDMSB Sy,Dz	もし DC＝0 ならば Sy データの MSB 位置→Dz の上位ワード、Dz の下位ワードクリア もし 1 ならば nop.	111110********** 1011111100yyzzzz	1	−	−	−	○

(1) 説明

Sx、Sy オペランドのビットの並びが最初に変わる位置を探し、そのビット位置を Dz オペランドへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 動作内容

```
/*      Case1 : PDMSB Sx,Dz     */
/*      Case2 : PDMSB Sy,Dz     */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
```

```
    DSP_ALU_SRC2 = 0x0;
    DSP_ALU_SRC2G= 0x0;

    if (Case1) {           /*  msb(Sx) -> Dz */
       switch (xx) {       /* Sx Operand selection bit (xx) */
           case 0x0:       DSP_ALU_SRC1  = X0;
                           if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                           else                   DSP_ALU_SRC1G = 0x0;
                           break;
           case 0x1:       DSP_ALU_SRC1  = X1;
                           if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                           else                   DSP_ALU_SRC1G = 0x0;
                           break;
           case 0x2:       DSP_ALU_SRC1  = A0;
                           DSP_ALU_SRC1G = A0G;
                           break;
           case 0x3:       DSP_ALU_SRC1  = A1;
                           DSP_ALU_SRC1G = A1G;
                           break;
       }
    }
    else {                 /* msb(Sy) -> Dz */
       switch (yy) {       /* Sy Operand selection bit (yy) */
           case 0x0:       DSP_ALU_SRC1  = Y0;
                           break;
           case 0x1:       DSP_ALU_SRC1  = Y1;
                           break;
           case 0x2:       DSP_ALU_SRC1  = M0;
                           break;
           case 0x3:       DSP_ALU_SRC1  = M1;
                           break;
       }
       if (DSP_ALU_SRC1_MSB)      DSP_ALU_SRC1G = 0xff;
       else                       DSP_ALU_SRC1G = 0x0;
    }
    {
       short int i;
       unsigned char msb, src1g;
       unsigned long src1=DSP_ALU_SRC1;
       msb= DSP_ALU_SRC1G_BIT7;
       src1g=(DSP_ALU_SRC1G_LSB8 << 1);
       for(i=38;((msb==(src1g>>7))&&(i>=32));i--) { src1g <<= 1; }
       if(i==31)  {
           for(i;((msb==(src1>>31))&&(i>=0));i--) { src1 <<= 1; }
       }
       DSP_ALU_DST = 0x0;
       DSP_ALU_DST_HW = (short int) (30-i);
       if (DSP_ALU_DST_MSB)       DSP_ALU_DSTG_LSB8 = 0xff;
       else                       DSP_ALU_DSTG_LSB8 = 0x0;
    }

    carry_bit = 0;

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
```

```
            overflow_bit= 0;

            /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
            case 0x5:      A1_HW = DSP_ALU_DST_HW;
                           A1_LW = 0x0;            /* clear LSW */
                           A1G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
            case 0x7:      A0_HW = DSP_ALU_DST_HW;
                           A0_LW = 0x0;            /* clear LSW */
                           A0G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
            case 0x8:      X0_HW = DSP_ALU_DST_HW;
                           X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:      X1_HW = DSP_ALU_DST_HW;
                           X1_LW = 0x0;            /* clear LSW */
            break;
            case 0xa:      Y0_HW = DSP_ALU_DST_HW;
                           Y0_LW = 0x0;            /* clear LSW */
            break;
            case 0xb:      Y1_HW = DSP_ALU_DST_HW;
                           Y1_LW = 0x0;            /* clear LSW */
            break;
            case 0xc:      M0_HW = DSP_ALU_DST_HW;
                           M0_LW = 0x0;            /* clear LSW */
            break;
            case 0xe:      M1_HW = DSP_ALU_DST_HW;
                           M1_LW = 0x0;            /* clear LSW */
            break;
            default:       printf("\nERROR:Illegal DSPInstruction"); break;
        }
        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        plus_dc_bit();

    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

            /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
            case 0x5:      A1_HW = DSP_ALU_DST_HW;
                           A1_LW = 0x0;            /* clear LSW */
                           A1G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
            case 0x7:      A0_HW = DSP_ALU_DST_HW;
                           A0_LW = 0x0;            /* clear LSW */
                           A0G = DSP_ALU_DSTG & 0x000000FF;
                           if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
```

```
                case 0x8:       X0_HW = DSP_ALU_DST_HW;
                                X0_LW = 0x0;            /* clear LSW */
                break;
                case 0x9:       X1_HW = DSP_ALU_DST_HW;
                                X1_LW = 0x0;            /* clear LSW */
                break;
                case 0xa:       Y0_HW = DSP_ALU_DST_HW;
                                Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:       Y1_HW = DSP_ALU_DST_HW;
                                Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:       M0_HW = DSP_ALU_DST_HW;
                                M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:       M1_HW = DSP_ALU_DST_HW;
                                M1_LW = 0x0;            /* clear LSW */
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(3) **使用例**

```
  PDMSB X0,M0  NOPX  NOPY  ;            実行前: X0=H'0052330F, M0=H'12345678
                                        実行後: X0=H'0052330F, M0=H'00080000
  PDMSB X1,X1  NOPX  NOPY  ;            実行前: X1=H'FC342855
                                        実行後: X1=H'00050000
                          無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。
```

6.3.11 [if cc] PINC INCrement by 1 with Condition DSP 算術演算命令

条件付きインクリメント

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PINC Sx,Dz	Sx の上位ワード＋1→Dz の上位ワード、Dz の下位ワードクリア	111110********** 10011001xx00zzzz	1	更新	−	−	○
PINC Sy,Dz	Sy の上位ワード＋1→Dz の上位ワード、Dz の下位ワードクリア	111110********** 1011100100yyzzzz	1	更新	−	−	○
DCT PINC Sx,Dz	もし DC＝1 ならば Sx の上位ワード＋1→Dz の上位ワード、Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10011010xx00zzzz	1	−	−	−	○
DCT PINC Sy,Dz	もし DC＝1 ならば Sy の上位ワード＋1→Dz の上位ワード、Dz の下位ワードクリア もし 0 ならば nop.	111110********** 1011101000yyzzzz	1	−	−	−	○
DCF PINC Sx,Dz	もし DC＝0 ならば Sx の上位ワード＋1→Dz の上位ワード、Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10011011xx00zzzz	1	−	−	−	○
DCF PINC Sy,Dz	もし DC＝0 ならば Sy の上位ワード＋1→ Dz の上位ワード、Dz の下位ワードクリア もし 1 ならば nop.	111110********** 1011101100yyzzzz	1	−	−	−	○

(1) 説明

Sx、Sy オペランドの上位ワードの内容に 1 を加算し、その結果を Dz オペランドの上位ワードへ格納し、Dz オペランドの下位ワードを 0 でクリアします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 注意

デスティネーションレジスタの下位ワードの内容は DC ビットの更新には無視されます。

(3) 動作内容

```
/*      Case1 : PINC Sx,Dz    */
/*      Case2 : PINC Sy,Dz    */
```

```
{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */

        DSP_ALU_SRC2 = 0x1;
        DSP_ALU_SRC2G= 0x0;

        if (Case1) {            /* MSW of Sx +1 -> Dz */
           switch (xx) {        /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                                else                   DSP_ALU_SRC1G = 0x0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                                if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                                else                   DSP_ALU_SRC1G = 0x0;
                                break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                                DSP_ALU_SRC1G = A0G;
                                break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                                DSP_ALU_SRC1G = A1G;
                                break;
           }
        }
        else {                  /* MSW of Sy +1 -> Dz */
           switch (yy) {        /* Sy Operand selection bit (yy) */
                case 0x0:       DSP_ALU_SRC1  = Y0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = Y1;
                                break;
                case 0x2:       DSP_ALU_SRC1  = M0;
                                break;
                case 0x3:       DSP_ALU_SRC1  = M1;
                                break;
           }
           if (DSP_ALU_SRC1_MSB)       DSP_ALU_SRC1G = 0xff;
           else                        DSP_ALU_SRC1G = 0x0;
        }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW + 1;
    carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU_DST_MSB) |
                (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
    overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

                /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1_HW = DSP_ALU_DST_HW;
                                A1_LW = 0x0;            /* clear LSW */
```

```
                        A1G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
            case 0x7:   A0_HW = DSP_ALU_DST_HW;
                        A0_LW = 0x0;            /* clear LSW */
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
            case 0x8:   X0_HW = DSP_ALU_DST_HW;
                        X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:   X1_HW = DSP_ALU_DST_HW;
                        X1_LW = 0x0;            /* clear LSW */
            break;
            case 0xa:   Y0_HW = DSP_ALU_DST_HW;
                        Y0_LW = 0x0;            /* clear LSW */
            break;
            case 0xb:   Y1_HW = DSP_ALU_DST_HW;
                        Y1_LW = 0x0;            /* clear LSW */
            break;
            case 0xc:   M0_HW = DSP_ALU_DST_HW;
                        M0_LW = 0x0;            /* clear LSW */
            break;
            case 0xe:   M1_HW = DSP_ALU_DST_HW;
                        M1_LW = 0x0;            /* clear LSW */
            break;
            default:    printf("\nERROR:Illegal DSPInstruction"); break;
    }
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    plus_dc_bit();

}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

            /* ALU Destination assignment */
    switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
            case 0x5:   A1_HW = DSP_ALU_DST_HW;
                        A1_LW = 0x0;            /* clear LSW */
                        A1G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
            case 0x7:   A0_HW = DSP_ALU_DST_HW;
                        A0_LW = 0x0;            /* clear LSW */
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
            case 0x8:   X0_HW = DSP_ALU_DST_HW;
                        X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:   X1_HW = DSP_ALU_DST_HW;
                        X1_LW = 0x0;            /* clear LSW */
            break;
```

```
                case 0xa:       Y0_HW = DSP_ALU_DST_HW;
                                Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:       Y1_HW = DSP_ALU_DST_HW;
                                Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:       M0_HW = DSP_ALU_DST_HW;
                                M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:       M1_HW = DSP_ALU_DST_HW;
                                M1_LW = 0x0;            /* clear LSW */
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(4) **使用例**

```
  PINC X0,M0  NOPX  NOPY  ;             実行前: X0=H'0052330F, M0=H'12345678
                                        実行後: X0=H'0052330F, M0=H'00530000
  PINC X1,X1  NOPX  NOPY  ;             実行前: X1=H'FC342855
                                        実行後: X1=H'FC350000
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.12 [if cc] PLDS LoaD System register DSP システム制御命令

条件付きシステムレジスタへのロード

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PLDS Dz,MACH	Dz→MACH	111110**********	1	−	−	−	○
		111011010000zzzz					
PLDS Dz,MACL	Dz→MACL	111110**********	1	−	−	−	○
		111111010000zzzz					
DCT PLDS Dz,MACH	**もし** DC＝1 **ならば** Dz→MACH	111110**********	1	−	−	−	○
	もし 0 **ならば** nop.	111011100000zzzz					○
DCT PLDS Dz,MACL	**もし** DC＝1 **ならば** Dz→MACL	111110**********	1	−	−	−	
	もし 0 **ならば** nop.	111111100000zzzz					○
DCF PLDS Dz,MACH	**もし** DC＝0 **ならば** Dz→MACH	111110**********	1	−	−	−	○
	もし 1 **ならば** nop.	111011110000zzzz					
DCF PLDS Dz,MACL	**もし** DC＝0 **ならば** Dz→MACL	111110**********	1	−	−	−	
	もし 1 **ならば** nop.	111111110000zzzz					

(1) **説明**

Dz オペランドの内容を、MACH、MACL レジスタへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

DSR レジスタの DC、N、Z、V、GT ビットはいずれも更新されません。

(2) **注意**

PLDS と MOVX、MOVY は並列に指定できますが、実行には 2 サイクルかかる場合があります。

(3) **動作内容**

```
/*      Case1 : PLDS Dz,MACH    */
/*      Case2 : PLDS Dz,MACL    */

{

  if(CASE1){    /* Dz -> MACH */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       MACH = A1;
                break;
                case 0x7:       MACH = A0;
                break;
                case 0x8:       MACH = X0;
                break;
                case 0x9:       MACH = X1;
                break;
```

```
                case 0xa:       MACH = Y0;
                break;
                case 0xb:       MACH = Y1;
                break;
                case 0xc:       MACH = M0;
                break;
                case 0xe:       MACH = M1;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
}
        }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       MACH = A1;
                break;
                case 0x7:       MACH = A0;
                break;
                case 0x8:       MACH = X0;
                break;
                case 0x9:       MACH = X1;
                break;
                case 0xa:       MACH = Y0;
                break;
                case 0xb:       MACH = Y1;
                break;
                case 0xc:       MACH = M0;
                break;
                case 0xe:       MACH = M1;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
}
        }
  else{    /* Dz -> MACL */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       MACL = A1;
                break;
                case 0x7:       MACL = A0;
                break;
                case 0x8:       MACL = X0;
                break;
                case 0x9:       MACL = X1;
                break;
                case 0xa:       MACL = Y0;
                break;
                case 0xb:       MACL = Y1;
                break;
                case 0xc:       MACL = M0;
                break;
                case 0xe:       MACL = M1;
                break;
```

```
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       MACL = A1;
                break;
                case 0x7:       MACL = A0;
                break;
                case 0x8:       MACL = X0;
                break;
                case 0x9:       MACL = X1;
                break;
                case 0xa:       MACL = Y0;
                break;
                case 0xb:       MACL = Y1;
                break;
                case 0xc:       MACL = M0;
                break;
                case 0xe:       MACL = M1;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
        }
}
```

(4) **使用例**

```
  PLDS A0,MACH  NOPX  NOPY                          ;実行前: A0=H'123456789A,
                                                            MACH=H'66666666
                                                     実行後: A0=H'123456789A,
                                                            MACH=H'3456789A
```

6.3.13 PMULS MULtiply Signed by Signed DSP 算術演算命令

符号付き乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PMULS Se,Sf,Dg	Se の上位ワード×Sf の上位ワード→Dg	111110********** 0100eeff0000gg00	1	ー	ー	ー	○

(1) **説明**

Se、Sf オペランドの上位ワードの内容を符号付きとして乗算し、結果を Dg オペランドに格納します。

DSR レジスタの DC、N、Z、V、GT ビットはいずれも更新されません。

(2) **注意**

PMULS は固定小数点乗算ですので、ソースデータが同じでも整数乗算の MULS とは演算結果が異なります。

【注】 整数データとして見ると演算結果が 2 倍の値にみえます。

(3) **動作内容**

```
/*      PMULS Se,Sf,Dg    */

{
/* Multiplier Sources assignment */
        switch (ee) {          /* Se Operand selection bit (ee) */
                case 0x0:      DSP_M_SRC1 = X0_HW;
                               break;
                case 0x1:      DSP_M_SRC1 = X1_HW;
                               break;
                case 0x2:      DSP_M_SRC1 = Y0_HW;
                               break;
                case 0x3:      DSP_M_SRC1 = A1_HW;
                               break;
        }
        switch (ff) {          /* Sf Operand selection bit (ff) */
                case 0x0:      DSP_M_SRC2 = Y0_HW;
                               break;
                case 0x1:      DSP_M_SRC2 = Y1_HW;
                               break;
                case 0x2:      DSP_M_SRC2 = X0_HW;
                               break;
                case 0x3:      DSP_M_SRC2 = A1_HW;
                               break;
        }

/* Multiplier Operation */
        if ((SBIT==1) && (DSP_M_SRC1==0x8000) && (DSP_M_SRC2==0x8000)) {
                DSP_M_DST=0x7fffffff;  /* overflow protection */
        }
        else {
```

```
            DSP_M_DST=((long)(short)DSP_M_SRC1*(long)(short)DSP_M_SRC2)<<1;
        }
        if (DSP_M_DST_MSB) DSP_M_DSTG_LSB8 = 0xff;
        else    DSP_M_DSTG_LSB8 = 0x0;

/* Multiplier Destination assignment */
        switch (gg) {          /* Dg Operand selection bit (gg) */
                case 0x0:      M0 = DSP_M_DST;
                               break;
                case 0x1:      M1 = DSP_M_DST;
                               break;
                case 0x2:      A0 = DSP_M_DST;
                               if(DSP_M_DSTG_LSB8==0x0) A0G=0x0;
                               else A0G=0xffffffff;
                               break;
                case 0x3:      A1 = DSP_M_DST;
                               if(DSP_M_DSTG_LSB8==0x0) A1G=0x0;
                               else A1G=0xffffffff;
                               break;
        }

}
```

(4) **使用例**

```
  PMULS X0,Y0,M0  NOPX  NOPY  ; 実行前: X0=H'00010000,Y0=H'00020000,
                                          (2^-15)          (2^-14)
                                      M0=H'33333333
                                実行後: X0=H'00010000,Y0=H'00020000,
                                       M0=H'00000004←整数データとして見ると、
                                          (2^-29)     2 倍の値になります。
  PMULS X1,Y1,A0  NOPX  NOPY  ;          実行前: X1=H'FFFE2222,Y1=H'0001AAAA,
                                                 A0=H'4444444444
                                         実行後: X1=H'FFFE2222,Y1=H'0001AAAA,
                                                 A0=H'FFFFFFFFFC

                                          (    ) : 固定小数点値
```

6.3.14 [if cc] PNEG NEGate DSP 算術演算命令

条件付き符号反転

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PNEG Sx,Dz	0－Sx→Dz	111110********** 11001001xx00zzzz	1	更新	－	－	○
PNEG Sy,Dz	0－Sy→Dz	111110********** 1110100100yyzzzz	1	更新	－	－	○
DCT PNEG Sx,Dz	**もし** DC＝1 **ならば** 0－Sx→Dz	111110**********	1	－	－	－	○
	もし 0 **ならば** nop.	11001010xx00zzzz					
DCT PNEG Sy,Dz	**もし** DC＝1 **ならば** 0－Sy→Dz	111110**********	1	－	－	－	○
	もし 0 **ならば** nop.	1110101000yyzzzz					
DCF PNEG Sx,Dz	**もし** DC＝0 **ならば** 0－Sx→Dz	111110**********	1	－	－	－	○
	もし 1 **ならば** nop.	11001011xx00zzzz					
DCF PNEG Sy,Dz	**もし** DC＝0 **ならば** 0－Sy→Dz	111110**********	1	－	－	－	○
	もし 1 **ならば** nop.	1110101100yyzzzz					

(1) 説明

符号を反転します。0 から Sx、Sy オペランドの内容を減算して Dz オペランドへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 動作内容

```
/*      Case1 : PNEG Sx,Dz    */
/*      Case2 : PNEG Sy,Dz    */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

        DSP_ALU_SRC1 = 0;
        DSP_ALU_SRC1G= 0;

/* ALU Sources assignment */
        if (Case1) {           /* 0 - Sx -> Dz */
                switch (xx) {          /* Sx Operand selection bit (xx) */
                   case 0x0:   DSP_ALU_SRC2  = X0;
                               if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
                               else                   DSP_ALU_SRC2G = 0x0;
                               break;
```

```
                case 0x1:   DSP_ALU_SRC2  = X1;
                            if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
                            else                   DSP_ALU_SRC2G = 0x0;
                            break;
                case 0x2:   DSP_ALU_SRC2  = A0;
                            DSP_ALU_SRC2G = A0G;
                            break;
                case 0x3:   DSP_ALU_SRC2  = A1;
                            DSP_ALU_SRC2G = A1G;
                            break;
            }
        }
        else  {         /* 0 - Sy -> Dz */
                switch (yy) {          /* Sy Operand selection bit (yy) */
                case 0x0:   DSP_ALU_SRC2  = Y0;
                            break;
                case 0x1:   DSP_ALU_SRC2  = Y1;
                            break;
                case 0x2:   DSP_ALU_SRC2  = M0;
                            break;
                case 0x3:   DSP_ALU_SRC2  = M1;
                            break;
                }
                if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
                else                   DSP_ALU_SRC2G = 0x0;
        }

    DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
    carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB) |
                (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

    overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

    overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1 = DSP_ALU_DST;
                               A1G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:      A0 = DSP_ALU_DST;
                               A0G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:      X0 = DSP_ALU_DST;
                break;
                case 0x9:      X1 = DSP_ALU_DST;
                break;
                case 0xa:      Y0 = DSP_ALU_DST;
                break;
```

```
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        minus_dc_bit();
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = DSP_ALU_DST;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = DSP_ALU_DST;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = DSP_ALU_DST;
                break;
                case 0x9:       X1 = DSP_ALU_DST;
                break;
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(3) **使用例**

```
PNEG X0,A0  NOPX  NOPY        ;実行前: X0=H'55555555,A0=H'A987654321
                               実行後: X0=H'55555555,A0=H'FFAAAAAAAB
PNEG Y1,Y1  NOPX  NOPY        ;実行前: Y1=H'99999999
                               実行後: Y1=H'66666667
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.15 [if cc] POR logical OR DSP 論理演算命令

条件付き論理和演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
POR Sx,Sy,Dz	Sx | Sy→Dz、 Dz の下位ワードクリア	111110********** 10110101xxyyzzzz	1	更新	−	−	○
DCT POR Sx,Sy,Dz	もし DC＝1 ならば Sx|Sy →Dz、 Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10110110xxyyzzzz	1	−	−	−	○
DCF POR Sx,Sy,Dz	もし DC＝0 ならば Sx|Sy →Dz、 Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10110111xxyyzzzz	1	−	−	−	○

(1) 説明

Sx オペランドの上位ワードの内容と Sy オペランドの上位ワードの内容との論理和を演算し、その結果を Dz オペランドの上位ワードへ格納し、Dz オペランドの下位ワードを 0 でクリアします。Dz オペランドがガードビットを持つレジスタの場合は、ガードビットも 0 でクリアします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 注意

デスティネーションレジスタの下位ワードの内容とガードビットの内容は DC ビットの更新には無視されます。

(3) 動作内容

```
/*      POR Sx,Sy,Dz    */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
        switch (xx) {           /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                                break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                                break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                                break;
        }
        switch (yy) {           /* Sy Operand selection bit (yy) */
```

```
            case 0x0:       DSP_ALU_SRC2  = Y0;
                            break;
            case 0x1:       DSP_ALU_SRC2  = Y1;
                            break;
            case 0x2:       DSP_ALU_SRC2  = M0;
                            break;
            case 0x3:       DSP_ALU_SRC2  = M1;
                            break;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW | DSP_ALU_SRC2_HW;

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

    /* ALU Destination assignment */
    switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
            case 0x5:       A1_HW = DSP_ALU_DST_HW;
                            A1_LW = 0x0;            /* clear LSW */
                            A1G = 0x0;              /* clear Guard bits */
            break;
            case 0x7:       A0_HW = DSP_ALU_DST_HW;
                            A0_LW = 0x0;            /* clear LSW */
                            A0G = 0x0;              /* clear Guard bits */
            break;
            case 0x8:       X0_HW = DSP_ALU_DST_HW;
                            X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:       X1_HW = DSP_ALU_DST;
                            X1_LW = 0x0;            /* clear LSW */
            break;
            case 0xa:       Y0_HW = DSP_ALU_DST;
                            Y0_LW = 0x0;            /* clear LSW */
            break;
            case 0xb:       Y1_HW = DSP_ALU_DST;
                            Y1_LW = 0x0;            /* clear LSW */
            break;
            case 0xc:       M0_HW = DSP_ALU_DST;
                            M0_LW = 0x0;            /* clear LSW */
            break;
            case 0xe:       M1_HW = DSP_ALU_DST;
                            M1_LW = 0x0;            /* clear LSW */
            break;
            default:        printf("\nERROR:Illegal DSPInstruction"); break;
    }

            carry_bit       = 0x0;
            negative_bit    = DSP_ALU_DST_MSB;
            zero_bit        = (DSP_ALU_DST_HW==0);
            overflow_bit    = 0x0;

            /* DSR register update */
            logical_dc_bit();
}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
    /* ALU Destination assignment */
```

```
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1_HW = DSP_ALU_DST_HW;
                                A1_LW = 0x0;            /* clear LSW */
                                A1G = 0x0;              /* clear Guard bits */
                break;
                case 0x7:       A0_HW = DSP_ALU_DST_HW;
                                A0_LW = 0x0;            /* clear LSW */
                                A0G = 0x0;              /* clear Guard bits */
                break;
                case 0x8:       X0_HW = DSP_ALU_DST_HW;
                                X0_LW = 0x0;            /* clear LSW */
                break;
                case 0x9:       X1_HW = DSP_ALU_DST;
                                X1_LW = 0x0;            /* clear LSW */
                break;
                case 0xa:       Y0_HW = DSP_ALU_DST;
                                Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:       Y1_HW = DSP_ALU_DST;
                                Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:       M0_HW = DSP_ALU_DST;
                                M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:       M1_HW = DSP_ALU_DST;
                                M1_LW = 0x0;            /* clear LSW */
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(4) **使用例**

```
  POR X0,Y0,A0  NOPX  NOPY ;
```

実行前:X0=H'33333333, Y0=H'55555555
A0=H'123456789A
実行後:X0=H'33333333, Y0=H'55555555
A0=H'0077770000

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.16 PRND RouNDing DSP 算術演算命令

丸め演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PRND Sx,Dz	Sx＋H'00008000→Dz	111110**********	1	更新	―	―	○
	Dz の下位ワードクリア	10011000xx00zzzz					
PRND Sy,Dz	Sy＋H'00008000→Dz	111110**********	1	更新	―	―	○
	Dz の下位ワードクリア	1011100000yyzzzz					

(1) **説明**

丸めを行います。Sx、Sy オペランドの内容にイミディエイトデータ H'00008000 を加算し、結果の上位ワードを Dz オペランドへ格納し、Dz オペランドの下位ワードを 0 でクリアします。

DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。

(2) **動作内容**

```
/*      Case1 : PRND Sx,Dz    */
/*      Case2 : PRND Sy,Dz    */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */

        DSP_ALU_SRC2 = 0x00008000;
        DSP_ALU_SRC2G= 0x0;

        if (Case1) {    /* Sx + H'00008000 -> Dz; clr Dz LW */
            switch (xx) {       /* Sx Operand selection bit (xx) */
            case 0x0:   DSP_ALU_SRC1  = X0;
                        if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                        else                   DSP_ALU_SRC1G = 0x0;
                        break;
            case 0x1:   DSP_ALU_SRC1  = X1;
                        if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                        else                   DSP_ALU_SRC1G = 0x0;
                        break;
            case 0x2:   DSP_ALU_SRC1  = A0;
                        DSP_ALU_SRC1G = A0G;
                        break;
            case 0x3:   DSP_ALU_SRC1  = A1;
                        DSP_ALU_SRC1G = A1G;
                        break;
            }
        }
        else {                  /* Sy + H'00008000 -> Dz; clr Dz LW */
            switch (yy) {       /* Sy Operand selection bit (yy) */
            case 0x0:   DSP_ALU_SRC1  = Y0;
```

```
                break;
    case 0x1:   DSP_ALU_SRC1  = Y1;
                break;
    case 0x2:   DSP_ALU_SRC1  = M0;
                break;
    case 0x3:   DSP_ALU_SRC1  = M1;
                break;
    }
    if (DSP_ALU_SRC1_MSB)       DSP_ALU_SRC1G = 0xff;
    else                        DSP_ALU_SRC1G = 0x0;
}

DSP_ALU_DST = (DSP_ALU_SRC1 + DSP_ALU_SRC2) & 0xFFFF0000;
carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU_DST_MSB)
                |(DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);

DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;
overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

overflow_protection();

        /* ALU Destination assignment */
switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
        case 0x5:       A1_HW = DSP_ALU_DST_HW;
                        A1_LW = 0x0;            /* clear LSW */
                        A1G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
        case 0x7:       A0_HW = DSP_ALU_DST_HW;
                        A0_LW = 0x0;            /* clear LSW */
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
        case 0x8:       X0_HW = DSP_ALU_DST_HW;
                        X0_LW = 0x0;            /* clear LSW */
        break;
        case 0x9:       X1_HW = DSP_ALU_DST_HW;
                        X1_LW = 0x0;            /* clear LSW */
        break;
        case 0xa:       Y0_HW = DSP_ALU_DST_HW;
                        Y0_LW = 0x0;            /* clear LSW */
        break;
        case 0xb:       Y1_HW = DSP_ALU_DST_HW;
                        Y1_LW = 0x0;            /* clear LSW */
        break;
        case 0xc:       M0_HW = DSP_ALU_DST_HW;
                        M0_LW = 0x0;            /* clear LSW */
        break;
        case 0xe:       M1_HW = DSP_ALU_DST_HW;
                        M1_LW = 0x0;            /* clear LSW */
        break;
        default:        printf("\nERROR:Illegal DSPInstruction"); break;
}
negative_bit = DSP_ALU_DSTG_BIT7;
zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);
```

```
        /* DSR register update */
        plus_dc_bit();
}
```

(3) **使用例**

```
  PRND X0,M0  NOPX  NOPY  ;
```

実行前: X0=H'0052330F, M0=H'12345678
実行後: X0=H'0052330F, M0=H'00520000

```
  PRND X1,X1  NOPX  NOPY  ;
```

実行前: X1=H'FC34C087
実行後: X1=H'FC350000
DC ビットは CS［2:0］の状態に従って更新。

6.3.17 [if cc] PSHA SHift Arithmetically with Condition DSP 算術シフト命令

条件付き算術シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PSHA Sx,Sy,Dz	**もし** Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** Sy＜0 **ならば** Sx＞＞｜Sy｜→Dz	111110********** 10010001xxyyzzzz	1	**更新**	−	−	○
DCT PSHA Sx,Sy,Dz	**もし** DC＝1 & Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** DC＝1 & Sy＜0 **ならば** Sx＞＞｜Sy｜→Dz **もし** DC＝0 **ならば** nop	111110********** 10010010xxyyzzzz	1	−	−	−	○
DCF PSHA Sx,Sy,Dz	**もし** DC＝0 & Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** DC＝0 & Sy＜0 **ならば** Sx＞＞｜Sy｜→Dz **もし** DC＝1 **ならば** nop	111110********** 10010011xxyyzzzz	1	−	−	−	○
PSHA #imm,Dz	**もし** imm≧0 **ならば** Dz＜＜imm→Dz **もし** imm＜0 **ならば** Dz＞＞｜imm｜→Dz	111110********** 00010iiiiiiizzzz	1	**更新**	−	−	○

(1) 説明

Sx または Dz オペランドの内容を算術的にシフトし、その結果を Dz オペランドへ格納します。シフト量は Sy オペランドまたはイミディエイト値 imm オペランドで指定します。シフト量が正の値のとき左にシフトします。負の値のとき右にシフトします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 動作内容

```
/*      PSHA Sx,Sy,Dz     */
＜レジスタオペランドによる場合＞

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
```

```
switch (xx) {          /* Sx Operand selection bit (xx) */
        case 0x0:      DSP_ALU_SRC1  = X0;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x1:      DSP_ALU_SRC1  = X1;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x2:      DSP_ALU_SRC1  = A0;
                       DSP_ALU_SRC1G = A0G;
                       break;
        case 0x3:      DSP_ALU_SRC1  = A1;
                       DSP_ALU_SRC1G = A1G;
                       break;
}
switch (yy) {          /* Sy Operand selection bit (yy) */
        case 0x0:      DSP_ALU_SRC2  = Y0 & 0x007F0000;
                       break;
        case 0x1:      DSP_ALU_SRC2  = Y1 & 0x007F0000;
                       break;
        case 0x2:      DSP_ALU_SRC2  = M0 & 0x007F0000;
                       break;
        case 0x3:      DSP_ALU_SRC2  = M1 & 0x007F0000;
                       break;
}
if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
else                   DSP_ALU_SRC2G = 0x0;

if((DSP_ALU_SRC2_HW & 0x0040)==0) {   /* Left Shift 0<=cnt<=32 */
   char cnt = (DSP_ALU_SRC2_HW & 0x003F);
   if(cnt > 32) {
      printf("¥nPSHA Sz,Sy,Dz Error! Shift %2X exceed range.¥n",cnt);
      exit();
   }
   DSP_ALU_DST = DSP_ALU_SRC1 << cnt;
   DSP_ALU_DSTG        = ((DSP_ALU_SRC1G << cnt) |
                       (DSP_ALU_SRC1 >> (32-cnt))) & 0x000000FF;
   carry_bit = ((DSP_ALU_DSTG & 0x00000001)==0x1);
}
else  {                               /* Right Shift 0< cnt <=32 */
   char cnt = ((~DSP_ALU_SRC2_HW & 0x003F)+1);
   if(cnt > 32) {
     printf("¥nPSHA Sz,Sy,Dz Error! shift -%2X exceed range.¥n",cnt);
     exit();
   }
   if((cnt>8) && DSP_ALU_SRC1G_BIT7) { /* MSB copy */
      DSP_ALU_DST=((DSP_ALU_SRC1>>8) | (DSP_ALU_SRC1G<<(32-8)));
      DSP_ALU_DST=(long) DSP_ALU_DST >> (cnt-8);
   }
   else {
      DSP_ALU_DST=((DSP_ALU_SRC1>>cnt)|(DSP_ALU_SRC1G<<(32-cnt)));
   }
   DSP_ALU_DSTG_LSB8 = (char) DSP_ALU_SRC1G_LSB8 >> cnt-- ;
   carry_bit = (((DSP_ALU_SRC1 >> cnt) & 0x00000001)==0x1);
```

```
}

overflow_bit = !(POS_NOT_OV || NEG_NOT_OV);
overflow_protection();

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

/* ALU Destination assignment */
   switch (zzzz) {     /* Dz Operand selection bit (zzzz) */
        case 0x5:      A1 = DSP_ALU_DST;
                       A1G = DSP_ALU_DSTG & 0x000000FF;
                       if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
        case 0x7:      A0 = DSP_ALU_DST;
                       A0G = DSP_ALU_DSTG & 0x000000FF;
                       if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
        case 0x8:      X0 = DSP_ALU_DST;
        break;
        case 0x9:      X1 = DSP_ALU_DST;
        break;
        case 0xa:      Y0 = DSP_ALU_DST;
        break;
        case 0xb:      Y1 = DSP_ALU_DST;
        break;
        case 0xc:      M0 = DSP_ALU_DST;
        break;
        case 0xe:      M1 = DSP_ALU_DST;
        break;
        default:       printf("\nERROR:Illegal DSPInstruction"); break;
   }

negative_bit = DSP_ALU_DSTG_BIT7;
zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

/* DSR register update */
shift_dc_bit();

}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

/* ALU Destination assignment */
   switch (zzzz) {     /* Dz Operand selection bit (zzzz) */
        case 0x5:      A1 = DSP_ALU_DST;
                       A1G = DSP_ALU_DSTG & 0x000000FF;
                       if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
        case 0x7:      A0 = DSP_ALU_DST;
                       A0G = DSP_ALU_DSTG & 0x000000FF;
                       if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
        case 0x8:      X0 = DSP_ALU_DST;
        break;
        case 0x9:      X1 = DSP_ALU_DST;
        break;
```

```
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
            }
        }
}

/*      PSHA #Imm,Dz     */
＜イミディエイトオペランドによる場合＞

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;
unsigned short tmp_imm;

/* ALU Sources assignment */
           switch (zzzz) {     /* Dz Operand selection bit (zzzz) */
                case 0x5:       DSP_ALU_SRC1 = A1;
                                DSP_ALU_SRC1G = A1G;
                break;
                case 0x7:       DSP_ALU_SRC1 = A0;
                                DSP_ALU_SRC1G = A1G;
                break;
                case 0x8:       DSP_ALU_SRC1 = X0;
                break;
                case 0x9:       DSP_ALU_SRC1 = X1;
                break;
                case 0xa:       DSP_ALU_SRC1 = Y0;
                break;
                case 0xb:       DSP_ALU_SRC1 = Y1;
                break;
                case 0xc:       DSP_ALU_SRC1 = M0;
                break;
                case 0xe:       DSP_ALU_SRC1 = M1;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
           }
           if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
           else                    DSP_ALU_SRC1G = 0x0;


        tmp_imm = (#Imm) & 0x0000007F); /* Extract 7bit Immidiate Data */

        if((tmp_imm & 0x0040)==0) {    /* Left Shift 0<= cnt <=32 */
           char cnt = (tmp_imm & 0x003F);
           if(cnt > 32) {
           printf("¥nPSHA Dz,#Imm,Dz Error! #Imm=%7X exceed range¥n",tmp_imm);
           exit();
           }
           DSP_ALU_DST = DSP_ALU_SRC1 << cnt;
```

```
    DSP_ALU_DSTG        = ((DSP_ALU_SRC1G << cnt)
                        |(DSP_ALU_SRC1 >> (32-cnt))) & 0x000000FF;
    carry_bit = ((DSP_ALU_DSTG & 0x00000001)==0x1);
}
else {                          /* Right Shift 0< cnt <=32 */
    char cnt = ((~tmp_imm & 0x003F)+1);
    if(cnt > 32) {
    printf("¥nPSHL Dz,#Imm,Dz Error! #Imm=%7X exceed range¥n",tmp_imm);
    exit();
    }
    if((cnt>8) && DSP_ALU_SRC1G_BIT7) { /* MSB copy */
    DSP_ALU_DST=((DSP_ALU_SRC1>>8) | (DSP_ALU_SRC1G<<(32-8)));
    DSP_ALU_DST=(long) DSP_ALU_DST >> (cnt-8);
    }
    else {
           DSP_ALU_DST=((DSP_ALU_SRC1>>cnt)|(DSP_ALU_SRC1G<<(32-cnt)));
    }
    DSP_ALU_DSTG_LSB8 = (char) DSP_ALU_SRC1G_LSB8 >>  cnt--;
    carry_bit = (((DSP_ALU_SRC1 >> cnt) & 0x00000001)==0x1);
}

overflow_bit = !(POS_NOT_OV || NEG_NOT_OV);
overflow_protection();

{ /* unconditional operation */
/* ALU Destination assignment */
    switch (zzzz) {     /* Dz Operand selection bit (zzzz) */
        case 0x5:       A1 = DSP_ALU_DST;
                        A1G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
        case 0x7:       A0 = DSP_ALU_DST;
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
        case 0x8:       X0 = DSP_ALU_DST;
        break;
        case 0x9:       X1 = DSP_ALU_DST;
        break;
        case 0xa:       Y0 = DSP_ALU_DST;
        break;
        case 0xb:       Y1 = DSP_ALU_DST;
        break;
        case 0xc:       M0 = DSP_ALU_DST;
        break;
        case 0xe:       M1 = DSP_ALU_DST;
        break;
        default:        printf("\nERROR:Illegal DSPInstruction"); break;
    }

negative_bit = DSP_ALU_DSTG_BIT7;
zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

/* DSR register update */
shift_dc_bit();
```

```
      }
}
```

(3) **使用例**

```
  PSHA X0,Y0,A0  NOPX  NOPY            ;実行前: X0=H'88888888, Y0=H'00020000,
                                                A0=H'123456789A
                                        実行後: X0=H'88888888, Y0=H'00020000,
                                                A0=H'FE22222222
  PSHA X0,Y0,X0  NOPX  NOPY            ;実行前: X0=H'33333333, Y0=H'FFFF0000,
                                        実行後: X0=H'19999999, Y0=H'FFFE0000,
  PSHA #-5,A1  NOPX  NOPY              ;実行前: A1=H'AAAAAAAAAA
                                        実行後: A1=H'FD55555555
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.18 [if cc] PSHL SHift Logically with condition DSP 論理シフト命令

条件付き論理シフト

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PSHL Sx,Sy,Dz	**もし** Sy≧0 **ならば** Sx＜＜Sy→Dz、Dz **の下位ワードクリア** **もし** Sy＜0 **ならば** Sx＞＞Sy→Dz、Dz **の下位ワードクリア**	111110********** 10000001xxyyzzzz	1	**更新**	―	―	○
DCT PSHL Sx,Sy,Dz	**もし** DC＝1 & Sy≧0 **ならば** Sx＜＜Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝1 & Sy＜0 **ならば** Sx＞＞Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝0 **ならば** nop	111110********** 10000010xxyyzzzz	1	―	―	―	○
DCF PSHL Sx,Sy,Dz	**もし** DC＝0 & Sy≧0 **ならば** Sx＜＜Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝0 & Sy＜0 **ならば** Sx＞＞Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝1 **ならば** nop	111110********** 10000011xxyyzzzz	1	―	―	―	○
PSHL #imm,Dz	**もし** imm≧0 **ならば** Dz＜＜imm→Dz、Dz **の下位ワードクリア** **もし** imm＜0 **ならば** Dz＞＞imm→Dz、Dz **の下位ワードクリア**	111110********** 00000iiiiiiizzzz	1	**更新**	―	―	○

(1) **説明**

Sx または Dz オペランドの上位ワードの内容を論理的にシフトし、その結果を Dz オペランドの上位ワードへ格納し、Dz オペランドの下位ワードを 0 でクリアします。Dz オペランドがガードビットを持つレジスタの場合は、ガードビットも 0 でクリアします。シフト量は Sy オペランドまたはイミディエイト値 imm オペランドで指定します。シフト量が正の値のとき左にシフトします。負の値のとき右にシフトします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新され

ます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 動作内容

```
/*      PSHL Sx,Sy,Dz    */
＜レジスタオペランドによる場合＞

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
        switch (xx) {          /* Sx Operand selection bit (xx) */
                case 0x0:      DSP_ALU_SRC1  = X0;
                               break;
                case 0x1:      DSP_ALU_SRC1  = X1;
                               break;
                case 0x2:      DSP_ALU_SRC1  = A0;
                               break;
                case 0x3:      DSP_ALU_SRC1  = A1;
                               break;
        }
        switch (yy) {          /* Sy Operand selection bit (yy) */
                case 0x0:      DSP_ALU_SRC2  = Y0 & 0x003F0000;
                               break;
                case 0x1:      DSP_ALU_SRC2  = Y1 & 0x003F0000;
                               break;
                case 0x2:      DSP_ALU_SRC2  = M0 & 0x003F0000;
                               break;
                case 0x3:      DSP_ALU_SRC2  = M1 & 0x003F0000;
                               break;
        }
        if((DSP_ALU_SRC2_HW & 0x0020)==0) {   /* Left Shift 0<=cnt<=16 */
           char cnt = (DSP_ALU_SRC2_HW & 0x001F);
           if(cnt > 16) {
           printf("PSHL Sx,Sy,Dz Error! Shift %2X exceed range¥n",cnt);
           exit();
           }
           DSP_ALU_DST_HW = DSP_ALU_SRC1_HW << cnt--;
           carry_bit = (((DSP_ALU_SRC1_HW << cnt) & 0x8000)==0x8000);
        }
        else  {                               /* Right Shift 0<cnt<=16 */
           char cnt = ((~DSP_ALU_SRC2_HW & 0x000F)+1);
           if(cnt > 16) {
           printf("PSHL Sx,Sy,Dz Error! Shift -%2X exceed range¥n",cnt);
           exit();
           }
           DSP_ALU_DST_HW = DSP_ALU_SRC1_HW >> cnt--;
           carry_bit = (((DSP_ALU_SRC1_HW >> cnt) & 0x0001)==0x1);
        }

        if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1_HW = DSP_ALU_DST_HW;
                               A1_LW = 0x0;            /* clear LSW */
```

```
                        A1G = 0x0;              /* clear Guard bits */
        break;
        case 0x7:       A0_HW = DSP_ALU_DST_HW;
                        A0_LW = 0x0;            /* clear LSW */
                        A0G = 0x0;              /* clear Guard bits */
        break;
        case 0x8:       X0_HW = DSP_ALU_DST_HW;
                        X0_LW = 0x0;            /* clear LSW */
        break;
        case 0x9:       X1_HW = DSP_ALU_DST;
                        X1_LW = 0x0;            /* clear LSW */
        break;
        case 0xa:       Y0_HW = DSP_ALU_DST;
                        Y0_LW = 0x0;            /* clear LSW */
        break;
        case 0xb:       Y1_HW = DSP_ALU_DST;
                        Y1_LW = 0x0;            /* clear LSW */
        break;
        case 0xc:       M0_HW = DSP_ALU_DST;
                        M0_LW = 0x0;            /* clear LSW */
        break;
        case 0xe:       M1_HW = DSP_ALU_DST;
                        M1_LW = 0x0;            /* clear LSW */
        break;
        default:        printf("\nERROR:Illegal DSPInstruction"); break;
}

        carry_bit       = 0x0;
        negative_bit    = DSP_ALU_DST_MSB;
        zero_bit        = (DSP_ALU_DST_HW==0);
        overflow_bit    = 0x0;

        /* DSR register update */
        shift_dc_bit();
}

else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

/* ALU Destination assignment */
    switch (zzzz) {     /* Dz Operand selection bit (zzzz) */
        case 0x5:       A1_HW = DSP_ALU_DST_HW;
                        A1_LW = 0x0;            /* clear LSW */
                        A1G = 0x0;              /* clear Guard bits */
        break;
        case 0x7:       A0_HW = DSP_ALU_DST_HW;
                        A0_LW = 0x0;            /* clear LSW */
                        A0G = 0x0;              /* clear Guard bits */
        break;
        case 0x8:       X0_HW = DSP_ALU_DST_HW;
                        X0_LW = 0x0;            /* clear LSW */
        break;
        case 0x9:       X1_HW = DSP_ALU_DST;
                        X1_LW = 0x0;            /* clear LSW */
        break;
        case 0xa:       Y0_HW = DSP_ALU_DST;
```

```
                                Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:       Y1_HW = DSP_ALU_DST;
                                Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:       M0_HW = DSP_ALU_DST;
                                M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:       M1_HW = DSP_ALU_DST;
                                M1_LW = 0x0;            /* clear LSW */
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
            }
        }
}

/*      PSHL #Imm,Dz     */
<イミディエイトオペランドによる場合>

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;
unsigned short tmp_imm;

/* ALU Sources assignment */
        switch (xx) {           /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                                break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                                break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                                break;
        }
        switch (yy) {           /* Sy Operand selection bit (yy) */
                case 0x0:       DSP_ALU_SRC2  = Y0 & 0x003F0000;
                                break;
                case 0x1:       DSP_ALU_SRC2  = Y1 & 0x003F0000;
                                break;
                case 0x2:       DSP_ALU_SRC2  = M0 & 0x003F0000;
                                break;
                case 0x3:       DSP_ALU_SRC2  = M1 & 0x003F0000;
                                break;
        }

        tmp_imm = (#Imm) & 0x0000007F); /* Extract 7bit Immidiate Data */

        if((tmp_imm & 0x0020)==0) {    /* Left Shift 0<= cnt <16 */
            char cnt = (tmp_imm & 0x001F);
            if(cnt > 16) {
            printf("PSHL Dz,#Imm,Dz Error! #Imm=%6X exceed range¥n",tmp_imm);
            exit();
            }
            DSP_ALU_DST_HW = DSP_ALU_SRC1_HW << cnt--;
            carry_bit = (((DSP_ALU_SRC1_HW << cnt) & 0x8000)==0x8000);
```

```
        }
        else  {                         /* Right Shift 0< cnt <=16 */
            char cnt = ((~tmp_imm & 0x001F)+1);
            if(cnt > 16) {
            printf("PSHL Dz,#Imm,Dz Error! #Imm=%6X exceed range¥n",tmp_imm);
            exit();
            }
            DSP_ALU_DST_HW = DSP_ALU_SRC1_HW >> cnt--;
            carry_bit = (((DSP_ALU_SRC1_HW >> cnt) & 0x0001)==0x1);
        }

        { /* unconditional operation */

        /* ALU Destination assignment */
            switch (zzzz) {    /* Dz Operand selection bit (zzzz) */
               case 0x5:       A1_HW = DSP_ALU_DST_HW;
                               A1_LW = 0x0;            /* clear LSW */
                               A1G = 0x0;              /* clear Guard bits */
               break;
               case 0x7:       A0_HW = DSP_ALU_DST_HW;
                               A0_LW = 0x0;            /* clear LSW */
                               A0G = 0x0;              /* clear Guard bits */
               break;
               case 0x8:       X0_HW = DSP_ALU_DST_HW;
                               X0_LW = 0x0;            /* clear LSW */
               break;
               case 0x9:       X1_HW = DSP_ALU_DST;
                               X1_LW = 0x0;            /* clear LSW */
               break;
               case 0xa:       Y0_HW = DSP_ALU_DST;
                               Y0_LW = 0x0;            /* clear LSW */
               break;
               case 0xb:       Y1_HW = DSP_ALU_DST;
                               Y1_LW = 0x0;            /* clear LSW */
               break;
               case 0xc:       M0_HW = DSP_ALU_DST;
                               M0_LW = 0x0;            /* clear LSW */
               break;
               case 0xe:       M1_HW = DSP_ALU_DST;
                               M1_LW = 0x0;            /* clear LSW */
               break;
               default:        printf("\nERROR:Illegal DSPInstruction"); break;
            }

               carry_bit       = 0x0;
               negative_bit    = DSP_ALU_DST_MSB;
               zero_bit        = (DSP_ALU_DST_HW==0);
               overflow_bit    = 0x0;

               /* DSR register update */
               shift_dc_bit();
        }
}
```

(3) **使用例**

```
PSHL X0,Y0,A0  NOPX  NOPY              ;実行前:X0=H'22222222, Y0=H'00030000,
                                               A0=H'123456789A
                                        実行後:X0=H'22222222, Y0=H'00030000,
                                               A0=H'0011100000
PSHL X1,Y1,X1  NOPX  NOPY              ;実行前:X1=H'CCCCCCCC, Y1=H'FFFE0000
                                        実行後:X1=H'33330000, Y1=H'FFFE0000
PSHL #7,A1  NOPX  NOPY                 ;実行前:A1=H'55555555
                                        実行後:A1=H'AA800000
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.19 [if cc] PSTS STore System register

DSP システム制御命令

条件付きシステムレジスタからのストア

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PSTS MACH,Dz	MACH→Dz	111110**********	1	−	−	−	○
		1100l1010000zzzz					
PSTS MACL,Dz	MACL→Dz	111110**********	1	−	−	−	○
		1101l1010000zzzz					
DCT PSTS MACH,Dz	**もし** DC＝1 **ならば** MACH→Dz	111110**********	1	−	−	−	○
	もし 0 **ならば** nop.	1100l1100000zzzz					
DCT PSTS MACL,Dz	**もし** DC＝1 **ならば** MACL→Dz	111110**********	1	−	−	−	○
	もし 0 **ならば** nop.	1101l1100000zzzz					
DCF PSTS MACH,Dz	**もし** DC＝0 **ならば** MACH→Dz	111110**********	1	−	−	−	○
	もし 1 **ならば** nop.	1100l1110000zzzz					
DCF PSTS MACL,Dz	**もし** DC＝0 **ならば** MACL→Dz	111110**********	1	−	−	−	○
	もし 1 **ならば** nop.	1101l1110000zzzz					

(1) **説明**

MACH、MACL レジスタの内容を、Dz オペランドへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

DSR レジスタの DC、N、Z、V、GT ビットはいずれも更新されません。

(2) **注意**

PSTS と MOVX、MOVY は並列に指定できますが、実行には 2 サイクルかかる場合があります。

(3) **動作内容**

```
/*      Case1 : PSTS MACH,Dz    */
/*      Case2 : PSTS MACL,Dz    */

{

  if(CASE1){    /* MACH -> Dz */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1 = MACH;
                               A1G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:      A0 = MACH;
```

```
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = MACH;
                break;
                case 0x9:       X1 = MACH;
                break;
                case 0xa:       Y0 = MACH;
                break;
                case 0xb:       Y1 = MACH;
                break;
                case 0xc:       M0 = MACH;
                break;
                case 0xe:       M1 = MACH;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
}
        }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = MACH;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = MACH;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = MACH;
                break;
                case 0x9:       X1 = MACH;
                break;
                case 0xa:       Y0 = MACH;
                break;
                case 0xb:       Y1 = MACH;
                break;
                case 0xc:       M0 = MACH;
                break;
                case 0xe:       M1 = MACH;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
}
        }
  else{    /* MACL -> Dz */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = MACL;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = MACL;
```

```
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = MACL;
                break;
                case 0x9:       X1 = MACL;
                break;
                case 0xa:       Y0 = MACL;
                break;
                case 0xb:       Y1 = MACL;
                break;
                case 0xc:       M0 = MACL;
                break;
                case 0xe:       M1 = MACL;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
}
        }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = MACL;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = MACL;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = MACL;
                break;
                case 0x9:       X1 = MACL;
                break;
                case 0xa:       Y0 = MACL;
                break;
                case 0xb:       Y1 = MACL;
                break;
                case 0xc:       M0 = MACL;
                break;
                case 0xe:       M1 = MACL;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
  }
}
```

(4) **使用例**

```
  PSTS MACH,A0  NOPX  NOPY  ;実行前:A0=H'123456789A,
                                    MACH=H'88888888
                              実行後:A0=H'FF88888888,
                                    MACH=H'88888888
```

6.3.20 [if cc] PSUB SUBtract with Condition DSP 算術演算命令

条件付き減算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PSUB Sx,Sy,Dz	Sx－Sy→Dz	111110**********	1	更新	－	－	○
		10100001xxyyzzzz					
DCT PSUB Sx,Sy,Dz	**もし** DC＝1 **ならば**	111110**********	1	－	－	－	○
	Sx－Sy→Dz	10100010xxyyzzzz					
	もし 0 **ならば** nop.						
DCF PSUB Sx,Sy,Dz	**もし** DC＝0 **ならば**	111110**********	1	－	－	－	○
	Sx－Sy→Dz	10100011xxyyzzzz					
	もし 1 **ならば** nop.						

(1) 説明

Sx オペランドの内容から Sy オペランドの内容を減算し、その結果を Dz オペランドへ格納します。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 動作内容

```
/*      PSUB Sx,Sy,Dz    */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
switch (xx) {          /* Sx Operand selection bit (xx) */
        case 0x0:      DSP_ALU_SRC1  = X0;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x1:      DSP_ALU_SRC1  = X1;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x2:      DSP_ALU_SRC1  = A0;
                       DSP_ALU_SRC1G = A0G;
                       break;
        case 0x3:      DSP_ALU_SRC1  = A1;
                       DSP_ALU_SRC1G = A1G;
                       break;
}
switch (yy) {          /* Sy Operand selection bit (yy) */
        case 0x0:      DSP_ALU_SRC2  = Y0;
```

```
                        break;
        case 0x1:       DSP_ALU_SRC2  = Y1;
                        break;
        case 0x2:       DSP_ALU_SRC2  = M0;
                        break;
        case 0x3:       DSP_ALU_SRC2  = M1;
                        break;
}
if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
else                   DSP_ALU_SRC2G = 0x0;

DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB) |
        (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
borrow_bit = !carry_bit;
DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1 = DSP_ALU_DST;
                               A1G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:      A0 = DSP_ALU_DST;
                               A0G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:      X0 = DSP_ALU_DST;
                break;
                case 0x9:      X1 = DSP_ALU_DST;
                break;
                case 0xa:      Y0 = DSP_ALU_DST;
                break;
                case 0xb:      Y1 = DSP_ALU_DST;
                break;
                case 0xc:      M0 = DSP_ALU_DST;
                break;
                case 0xe:      M1 = DSP_ALU_DST;
                break;
                default:       printf("\nERROR:Illegal DSPInstruction"); break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        minus_dc_bit();
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
```

```
        /* ALU Destination assignment */
        switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
                case 0x5:       A1 = DSP_ALU_DST;
                                A1G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:       A0 = DSP_ALU_DST;
                                A0G = DSP_ALU_DSTG & 0x000000FF;
                                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:       X0 = DSP_ALU_DST;
                break;
                case 0x9:       X1 = DSP_ALU_DST;
                break;
                case 0xa:       Y0 = DSP_ALU_DST;
                break;
                case 0xb:       Y1 = DSP_ALU_DST;
                break;
                case 0xc:       M0 = DSP_ALU_DST;
                break;
                case 0xe:       M1 = DSP_ALU_DST;
                break;
                default:        printf("\nERROR:Illegal DSPInstruction"); break;
        }
                }
}
```

(3) **使用例**

```
  PSUB X0,Y0,A0  NOPX  NOPY        ;実行前:     X0=H'55555555, Y0=H'33333333,
                                                A0=H'123456789A
                                    実行後:      X0=H'55555555, Y0=H'33333333,
                                                A0=H'0022222222
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

6.3.21 PSUB SUBtraction & MULtiply Signed DSP 算術演算命令
PMULS by Signed

減算と符号付き乗算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PSUB Sx,Sy,Du	Sx－Sy→Du	111110**********	1	更新	－	－	○
PMULS Se,Sf,Dg	Se の上位ワード×Sf の上位ワード→Dg	0110eeffxxyygguu					

(1) **説明**

Sx オペランドの内容から Sy オペランドの内容を減算し、結果を Du オペランドへ格納します。Se、Sf オペランドの上位ワードの内容を符号付きとして乗算し、結果を Dg オペランドに格納します。この 2 つの処理は同時に並行して実行されます。

DSR レジスタの DC ビットは ALU 演算の結果と CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも ALU 演算の結果に従って更新されます。

(2) **動作内容**

```
/*      PSUB Sx,Sy,Du   PMULS Se,Sf,Dg  */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* Multiplier Sources assignment */
        switch (ee) {           /* Se Operand selection bit (ee) */
                case 0x0:       DSP_M_SRC1 = X0_HW;
                                break;
                case 0x1:       DSP_M_SRC1 = X1_HW;
                                break;
                case 0x2:       DSP_M_SRC1 = Y0_HW;
                                break;
                case 0x3:       DSP_M_SRC1 = A1_HW;
                                break;
        }
        switch (ff) {           /* Sf Operand selection bit (ff) */
                case 0x0:       DSP_M_SRC2 = Y0_HW;
                                break;
                case 0x1:       DSP_M_SRC2 = Y1_HW;
                                break;
                case 0x2:       DSP_M_SRC2 = X0_HW;
                                break;
                case 0x3:       DSP_M_SRC2 = A1_HW;
                                break;
        }

/* ALU Sources assignment */
        switch (xx) {           /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                if (DSP_ALU_SRC1_MSB)
```

```
                                DSP_ALU_SRC1G_LSB8 = 0xff;
                        else    DSP_ALU_SRC1G_LSB8 = 0x0;
                        break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                        if (DSP_ALU_SRC1_MSB)
                                DSP_ALU_SRC1G_LSB8 = 0xff;
                        else    DSP_ALU_SRC1G_LSB8 = 0x0;
                        break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                        DSP_ALU_SRC1G = A0G;
                        break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                        DSP_ALU_SRC1G = A1G;
                        break;
        }
        switch (yy) {           /* Sy Operand selection bit (yy) */
                case 0x0:       DSP_ALU_SRC2  = Y0;
                        break;
                case 0x1:       DSP_ALU_SRC2  = Y1;
                        break;
                case 0x2:       DSP_ALU_SRC2  = M0;
                        break;
                case 0x3:       DSP_ALU_SRC2  = M1;
                        break;
        }
        if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G_LSB8 = 0xff;
        else                   DSP_ALU_SRC2G_LSB8 = 0x0;

/* Multiplier Operation */

        /* PMULS Se, Sf, Dg */
        if ((SBIT==1) && (DSP_M_SRC1==0x8000) && (DSP_M_SRC2==0x8000)) {
                DSP_M_DST=0x7fffffff;  /* overflow protection */
        }
        else {
                DSP_M_DST=((long)(short)DSP_M_SRC1*(long)(short)DSP_M_SRC2)<<1;
        }
        if (DSP_M_DST_MSB) DSP_M_DSTG_LSB8 = 0xff;
        else    DSP_M_DSTG_LSB8 = 0x0;

        switch (gg) {  /* Dg Operand selection bit (gg) */
                case 0x0:       M0 = DSP_M_DST;
                        break;
                case 0x1:       M1 = DSP_M_DST;
                        break;
                case 0x2:       A0 = DSP_M_DST;
                        if(DSP_M_DSTG_LSB8==0x0) A0G=0x0;
                        else A0G=0xffffffff;
                        break;
                case 0x3:       A1 = DSP_M_DST;
                        if(DSP_M_DSTG_LSB8==0x0) A1G=0x0;
                        else A1G=0xffffffff;
                        break;
        }
```

```
/* ALU operation */

        DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
        carry_bit=((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB)&& !DSP_ALU_DST_MSB)|
                (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
        borrow_bit = !carry_bit;
        DSP_ALU_DSTG_LSB8=DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

        overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
        overflow_protection();

        switch (uu) {  /* Du Operand selection bit (uu) */
                case 0x0:
                        X0  = DSP_ALU_DST;
                        negative_bit = DSP_ALU_DST_MSB;
                        zero_bit = (DSP_ALU_DST==0);
                        break;
                case 0x1:
                        Y0  = DSP_ALU_DST;
                        negative_bit = DSP_ALU_DST_MSB;
                        zero_bit = (DSP_ALU_DST==0);
                        break;
                case 0x2:
                        A0  = DSP_ALU_DST;
                        A0G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                        negative_bit = DSP_ALU_DSTG_BIT7;
                        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
                        break;
                case 0x3:
                        A1  = DSP_ALU_DST;
                        A1G = DSP_ALU_DSTG & 0x000000FF;
                        if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                        negative_bit = DSP_ALU_DSTG_BIT7;
                        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
                        break;
        }

        /* DSR register update */
        minus_dc_bit();

}
```

(3) **使用例**

```
PSUB A0,M0,A0  PMULS X0,Y0,M0  NOPX  NOPY ; 実行前: X0=H'00020000,
                                                Y0=H'FFFE0000,
                                                M0=H'33333333,
                                                A0=H'0022222222
                                           実行後: X0=H'00020000,
                                                Y0=H'FFFE0000,
                                                M0=H'FFFFFFF8,
                                                A0=H'0055555555
```

6.3.22 PSUBC SUBtract with Carry DSP 算術演算命令

ボロー付き減算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令		
					SH-1	SH-2	SH-DSP
PSUBC Sx,Sy,Dz	Sx−Sy−DC→Dz	111110********** 10100000xxyyzzzz	1	ボロー	−	−	○

(1) **説明**

Sx オペランドの内容から Sy オペランドの内容と DC ビットを減算し、Dz オペランドへ格納します。

DSR レジスタの DC ビットはボローフラグとして更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。

(2) **注意**

PADDC 命令実行後の DC ビットは、CS ビットに関係なく、ボローフラグとして更新されます。

(3) **動作内容**

```
/*      PSUBC Sx,Sy,Dz    */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
switch (xx) {          /* Sx Operand selection bit (xx) */
        case 0x0:      DSP_ALU_SRC1  = X0;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x1:      DSP_ALU_SRC1  = X1;
                       if (DSP_ALU_SRC1_MSB)  DSP_ALU_SRC1G = 0xff;
                       else                   DSP_ALU_SRC1G = 0x0;
                       break;
        case 0x2:      DSP_ALU_SRC1  = A0;
                       DSP_ALU_SRC1G = A0G;
                       break;
        case 0x3:      DSP_ALU_SRC1  = A1;
                       DSP_ALU_SRC1G = A1G;
                       break;
}
switch (yy) {          /* Sy Operand selection bit (yy) */
        case 0x0:      DSP_ALU_SRC2  = Y0;
                       break;
        case 0x1:      DSP_ALU_SRC2  = Y1;
                       break;
        case 0x2:      DSP_ALU_SRC2  = M0;
                       break;
        case 0x3:      DSP_ALU_SRC2  = M1;
                       break;
```

```
}
if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
else                   DSP_ALU_SRC2G = 0x0;

DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2 - DSPDCBIT;
carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB)
           | (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
borrow_bit = !carry_bit;
DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
overflow_protection();

        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1 = DSP_ALU_DST;
                               A1G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
                break;
                case 0x7:      A0 = DSP_ALU_DST;
                               A0G = DSP_ALU_DSTG & 0x000000FF;
                               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
                break;
                case 0x8:      X0 = DSP_ALU_DST;
                break;
                case 0x9:      X1 = DSP_ALU_DST;
                break;
                case 0xa:      Y0 = DSP_ALU_DST;
                break;
                case 0xb:      Y1 = DSP_ALU_DST;
                break;
                case 0xc:      M0 = DSP_ALU_DST;
                break;
                case 0xe:      M1 = DSP_ALU_DST;
                break;
                default:       printf("\nERROR:Illegal DSPInstruction"); break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        dc_always_borrow();
}
```

(4) **使用例**

```
  CS[2:0]=***: Always Carry or Borrow Mode
  PSUBC X0,Y0,M0  NOPX  NOPY ;          実行前:X0=H'33333333, Y0=H'55555555
                                              M0=H'12345678, DC=0
                                        実行後:X0=H'33333333, Y0=H'55555555
                                              M0=H'DDDDDDDE, DC=1
  PSUBC X0,Y0,M0  NOPX  NOPY ;          実行前:X0=H'33333333, Y0=H'55555555
                                              M0=H'12345678, DC=1
                                        実行後:X0=H'33333333, Y0=H'55555555
                                              M0=H'DDDDDDDD, DC=1
```

6.3.23 [if cc] PXOR logical eXclusive OR DSP 論理演算命令

条件付き排他的論理和演算

書式	動作概略	命令コード	実行ステート	T ビット	適用命令 SH-1	SH-2	SH-DSP
PXOR Sx,Sy,Dz	Sx^Sy→Dz、Dz の下位ワードクリア	111110********** 10100101xxyyzzzz	1	更新	―	―	○
DCT PXOR Sx,Sy,Dz	もし DC＝1 ならば Sx^Sy→Dz、Dz の下位ワードクリア もし 0 ならば nop.	111110********** 10100110xxyyzzzz	1	―	―	―	○
DCF PXOR Sx,Sy,Dz	もし DC＝0 ならば Sx^Sy→Dz、Dz の下位ワードクリア もし 1 ならば nop.	111110********** 10100111xxyyzzzz	1	―	―	―	○

(1) 説明

Sx オペランドの上位ワードの内容と Sy オペランドの上位ワードの内容との排他的論理和を演算し、その結果を Dz オペランドの上位ワードへ格納し、Dz オペランドの下位ワードを 0 でクリアします。Dz オペランドがガードビットを持つレジスタの場合は、ガードビットも 0 でクリアします。DCT、DCF の条件が指定されている場合は、条件が真のとき命令が実行されます。条件が偽のとき命令は実行されません。

条件が指定されていない場合は DSR レジスタの DC ビットは CS ビットの指定に従って更新されます。DSR レジスタの N、Z、V、GT ビットも更新されます。条件が指定されている場合は、条件が真で命令が実行された場合も、DC、N、Z、V、GT ビットは更新されません。

(2) 注意

デスティネーションレジスタの下位ワードの内容とガードビットの内容は DC ビットの更新には無視されます。

(3) 動作内容

```
/*      PXOR Sx,Sy,Dz    */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
        switch (xx) {           /* Sx Operand selection bit (xx) */
                case 0x0:       DSP_ALU_SRC1  = X0;
                                break;
                case 0x1:       DSP_ALU_SRC1  = X1;
                                break;
                case 0x2:       DSP_ALU_SRC1  = A0;
                                break;
                case 0x3:       DSP_ALU_SRC1  = A1;
                                break;
        }
        switch (yy) {           /* Sy Operand selection bit (yy) */
```

```
            case 0x0:       DSP_ALU_SRC2  = Y0;
                            break;
            case 0x1:       DSP_ALU_SRC2  = Y1;
                            break;
            case 0x2:       DSP_ALU_SRC2  = M0;
                            break;
            case 0x3:       DSP_ALU_SRC2  = M1;
                            break;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW ^ DSP_ALU_SRC2_HW;

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

    /* ALU Destination assignment */
    switch (zzzz) {         /* Dz Operand selection bit (zzzz) */
            case 0x5:       A1_HW = DSP_ALU_DST_HW;
                            A1_LW = 0x0;            /* clear LSW */
                            A1G = 0x0;              /* clear Guard bits */
            break;
            case 0x7:       A0_HW = DSP_ALU_DST_HW;
                            A0_LW = 0x0;            /* clear LSW */
                            A0G = 0x0;              /* clear Guard bits */
            break;
            case 0x8:       X0_HW = DSP_ALU_DST_HW;
                            X0_LW = 0x0;            /* clear LSW */
            break;
            case 0x9:       X1_HW = DSP_ALU_DST;
                            X1_LW = 0x0;            /* clear LSW */
            break;
            case 0xa:       Y0_HW = DSP_ALU_DST;
                            Y0_LW = 0x0;            /* clear LSW */
            break;
            case 0xb:       Y1_HW = DSP_ALU_DST;
                            Y1_LW = 0x0;            /* clear LSW */
            break;
            case 0xc:       M0_HW = DSP_ALU_DST;
                            M0_LW = 0x0;            /* clear LSW */
            break;
            case 0xe:       M1_HW = DSP_ALU_DST;
                            M1_LW = 0x0;            /* clear LSW */
            break;
            default:        printf("\nERROR:Illegal DSPInstruction");
            break;
    }

            carry_bit       = 0x0;
            negative_bit    = DSP_ALU_DST_MSB;
            zero_bit        = (DSP_ALU_DST_HW==0);
            overflow_bit    = 0x0;

            /* DSR register update */
            logical_dc_bit();

}
```

```
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

        /* ALU Destination assignment */
        switch (zzzz) {        /* Dz Operand selection bit (zzzz) */
                case 0x5:      A1_HW = DSP_ALU_DST_HW;
                               A1_LW = 0x0;            /* clear LSW */
                               A1G = 0x0;              /* clear Guard bits */
                break;
                case 0x7:      A0_HW = DSP_ALU_DST_HW;
                               A0_LW = 0x0;            /* clear LSW */
                               A0G = 0x0;              /* clear Guard bits */
                break;
                case 0x8:      X0_HW = DSP_ALU_DST_HW;
                               X0_LW = 0x0;            /* clear LSW */
                break;
                case 0x9:      X1_HW = DSP_ALU_DST;
                               X1_LW = 0x0;            /* clear LSW */
                break;
                case 0xa:      Y0_HW = DSP_ALU_DST;
                               Y0_LW = 0x0;            /* clear LSW */
                break;
                case 0xb:      Y1_HW = DSP_ALU_DST;
                               Y1_LW = 0x0;            /* clear LSW */
                break;
                case 0xc:      M0_HW = DSP_ALU_DST;
                               M0_LW = 0x0;            /* clear LSW */
                break;
                case 0xe:      M1_HW = DSP_ALU_DST;
                               M1_LW = 0x0;            /* clear LSW */
                break;
                default:       printf("\nERROR:Illegal DSPInstruction"); break;
        }
    }
}
```

(4) **使用例**

```
PXOR X0,Y0,A0  NOPX  NOPY ;  実行前:X0=H'33333333, Y0=H'55555555
                                    A0=H'123456789A
                             実行後:X0=H'33333333, Y0=H'55555555
                                    A0=H'0066660000
```

無条件実行の場合、DC ビットは CS［2:0］の状態に従って更新。

7. パイプライン動作

各命令のパイプライン動作を説明します。これは、CPU の命令実行ステート数（システムクロックサイクル数）の算出をするための情報を提供するものです。

7.1 パイプラインの基本構成

7.1.1 5 段パイプライン

パイプラインは、次の 5 つのステージから構成されます。

●IF： 命令フェッチ……プログラムが格納されているメモリから命令を取り込みます。

●ID： 命令デコード……取り込んだ命令を解読します。

●EX： 命令実行……解読結果に従い、データ演算やアドレス計算を行います。

●MA： メモリアクセス……メモリのデータアクセスを行います。
メモリアクセスを伴う命令で発生しますが一部例外があります。

●WB/DSP（W/D）：
ライトバック（CPU コア）または DSP（DSP ユニット）
ライトバック……メモリアクセスした結果（データ）をレジスタに戻します。
メモリロードを伴う命令で発生しますが一部例外があります。
DSP……DSP ユニットの ALU、MAC を使って演算します。
また、メモリアクセスした結果（データ）をレジスタに戻します。メモリへのライト、無操作（NOP）の場合は発生しません。

これらの各ステージは、命令の実行と共に流れていき、パイプラインを構成します。ある瞬間をとらえれば、5 つの命令が同時に実行されることになります。基本的なパイプラインの流れを**図** 7.1 に示します。1 つのステージが実行される期間をスロットとよび"←→"で表します。

IF、ID、EX の 3 ステージは全ての命令に存在しますが、命令によっては、MA、WB/DSP がない場合もあります。また、パイプラインの流れ方も、命令の種類によって変化します。IF と MA との競合などが発生することもあり、競合が発生するとパイプラインの流れ方が変わります。

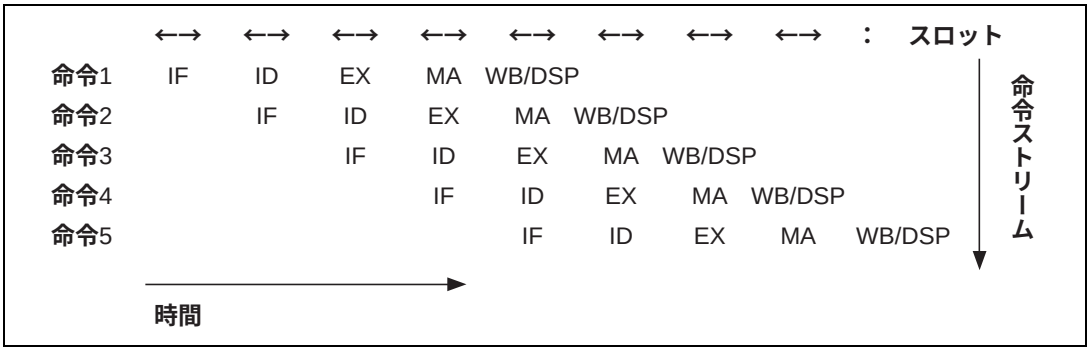


図 7.1 **パイプラインの基本構成**

7.1.2 スロットとパイプラインの流れ

1 つのステージが実行される期間をスロットと呼びます。このスロットについて以下に示すルールがあります。

（1） 命令の各ステージ（IF、ID、EX、MA、WB/DSP）は、必ず1スロットで実行されます。1スロット内で2つ以上のステージを実行することはありません。
ただし、WBはMA直後に実行されますので、ある命令のMAとWBが同一スロット内で実行されることがあります。

（2） 1スロットには別の命令の異なるステージが最大1つずつ設定されます。1スロット内で、別の命令の同じステージが実行されることはありません。
ありえないパイプラインの流れを**図**7.2、**図**7.3に示します。

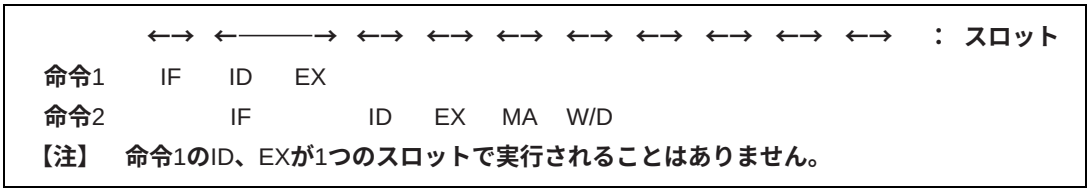


図 7.2 **ありえないパイプラインの流れ（1）**

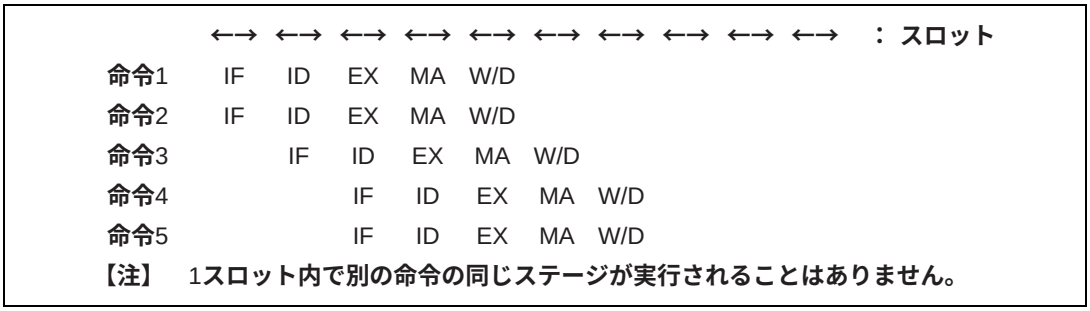


図 7.3 **ありえないパイプラインの流れ（2）**

7.1.3 1 スロットの実行にかかるステート数

1 スロットの実行にかかるステート数（システムクロックサイクル数）S は次の条件で算出します。

S＝（1 スロット内に含まれる各命令のステージのうちの最長ステート数）

すなわち、最も長いステージによって、他の短いステージを持つ命令はストールすることになります

各ステージの実行ステート数は……

- ●IF： 命令フェッチのためのメモリアクセスクロック数
- ●ID： 常に 1 ステート
- ●EX： 常に 1 ステート
- ●MA： データアクセスのためのメモリアクセスクロック数
- ●WB/DSP： 常に 1 ステート

たとえば、命令 1、命令 2 の IF（命令フェッチのためのメモリアクセス）に 2 サイクル、命令 1 の MA（データアクセスのためのメモリアクセス）に 3 サイクル、他は 1 サイクルかかるとした場合のパイプラインの流れを**図** 7.4 に示します。"—"はストールを表します。

スロット	←——→		←——→		←→	←————→			←→	←→
ステート数	(2)		(2)		(1)	(3)			(1)	(1)
命令1	IF	IF	ID	—	EX	MA	MA	MA	W/D	
命令2			IF	IF	ID	EX	—	—	MA	W/D

図 7.4 **複数サイクルかかるスロット**

7.1.4 命令実行ステート数

各命令の実行ステート数は、EX ステージの実行間隔で数えます。命令 1 の EX 段実行開始から、次に続く命令 2 の EX 段開始時点までのステート数が、命令 1 の実行時間となります。命令実行ステート数の数え方の例を**図** 7.5 に示します。

例えば、**図** 7.5 のようなパイプラインの流れの場合、命令 1 と命令 2 の EX ステージの間隔は 5 ステートですから、命令 1 の実行時間は 5 ステートになります。また、命令 2 と命令 3 の EX ステージの間隔は 1 ステートですから、命令 2 の実行時間は 1 ステートになります。もし、プログラムが、命令 3 で終了しているなら、命令 3 の実行時間は、命令 3 の次に仮想的に命令 4 として、MOV Rm,Rn をおいて、命令 3 と命令 4 の EX ステージの間隔から算出します。この例の場合、命令 3 の実行時間は 1 ステートになります。命令 1～命令 3 までの実行時間は合計 5＋1＋1＝7 ステートとなります。

この例では、命令 1 の MA と命令 4 の IF とが競合しています。MA と IF の競合時の動作については、**「**7.2.1 **命令フェッチ（**IF**）とメモリアクセス（**MA**）の競合」**を参照してください。

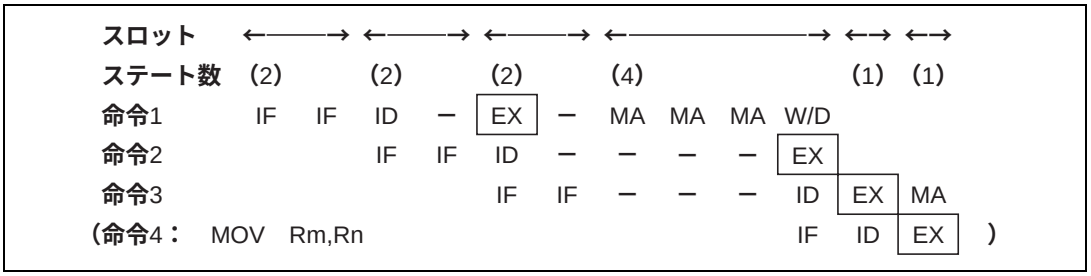


図 7.5 **命令実行ステート数の数え方の例**

7.2 競合の発生

競合は、次の 4 つの場合に発生します。競合が発生すると、そのスロットはスプリットし、1 スロットが 2 ステート以上必要になります。

（1） 命令フェッチ（IF）とメモリアクセス（MA）の競合
（2） 先行命令のデスティネーションレジスタを使うときの競合
（3） 乗算器アクセスの競合
（4） DSP演算又はメモリ・ロード（WB/DSP）とメモリ・ストア（MA）の競合

7.2.1 命令フェッチ（IF）とメモリアクセス（MA）の競合

(1) IF と MA の競合時の基本動作（共通）

IF ステージと MA ステージは共にメモリをアクセスしにいきますので、同時に動作できません。IF ステージと MA ステージが同じスロット内でメモリをアクセスしにいこうとすると、そのスロットはスプリットします。なお、WB がある場合は MA 終了直後に実行されます。IF と MA の競合時の動作を**図** 7.6 に示します。

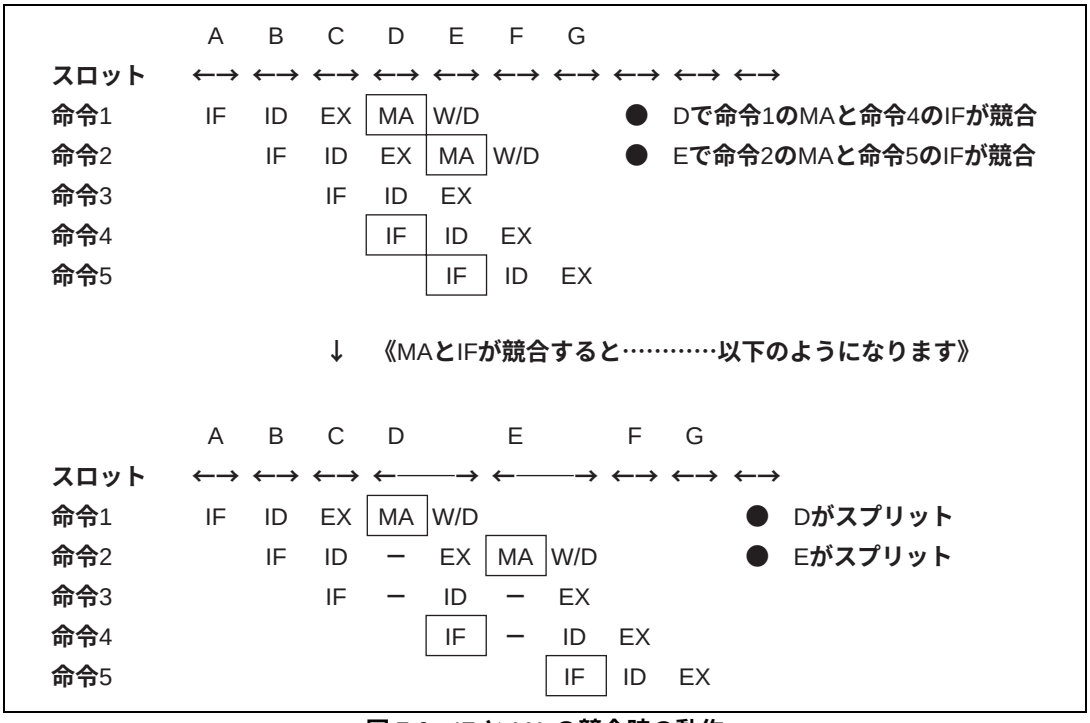


図 7.6 IF **と** MA **の競合時の動作**

MA と IF が競合したスロットはスプリットされ、スロットの前半で、MA が優先して実行され、スロットの後半で他の EX、ID、IF が同時に実行されます。MA に続いて WB がある場合はその MA 終了直後に実行されます。たとえば、**図** 7.6 の場合、スロット D では命令 1 の MA が優先し、命令 2 の EX、命令 3 の ID、命令 4 の IF の 3 つは、スロットの後半に同時に実行されることになります。またスロット E では、命令 2 の MA が優先し、命令 3 の EX、命令 4 の ID、命令 5 の IF は、後から同時に実行されます。

MA と IF が競合したスロットでかかるステート数は、MA でのメモリアクセスサイクル数と IF で

のメモリアクセスサイクル数の和になります。

(2) 内蔵 ROM/RAM または内蔵キャッシュに配置された命令の位置と、IF の関係（SH-1/2 の場合）

命令が SuperH マイコンの内蔵 ROM/RAM または内蔵キャッシュに配置されている場合、SuperH マイコンは内蔵 ROM/RAM または内蔵キャッシュを 32 ビットでアクセスします。SuperH マイコンの命令はすべて 16 ビット固定長なので、基本的に IF ステージの 1 回のアクセスで 2 命令持ってくることができます。

この時、IF で 1 回命令フェッチすると 2 命令持ってこれますので、次命令の IF では、メモリから命令をフェッチするバスサイクルは発生しません。さらにその次の命令の IF は命令を 2 つ分取り込み、またその次の命令の IF ではバスサイクルは発生しないことになります。

すなわち、内蔵 ROM/RAM または内蔵キャッシュに配置されている命令の内、ロングワード境界から配置されている命令（命令アドレスの下位 2 ビットが 00 の位置：A1＝0、A0＝0）の IF が、2 命令フェッチを行います。その次の命令の IF はバスサイクルを発生しません。このバスサイクルを発生しない IF を"if"と小文字で表します。必ず if は 1 ステートです。

一方、分岐により、ワード境界から配置されている命令（命令アドレスの下位 2 ビットが 10 の位置：A1＝1、A0＝0）からフェッチする場合、その IF のバスサイクルは、その命令 1 個しか取り込むことができません。したがって、その次の命令の IF はバスサイクルを発生することになりますが、その IF からは 2 命令取り込みを行います。以上の動作について、**図** 7.7 に示します。

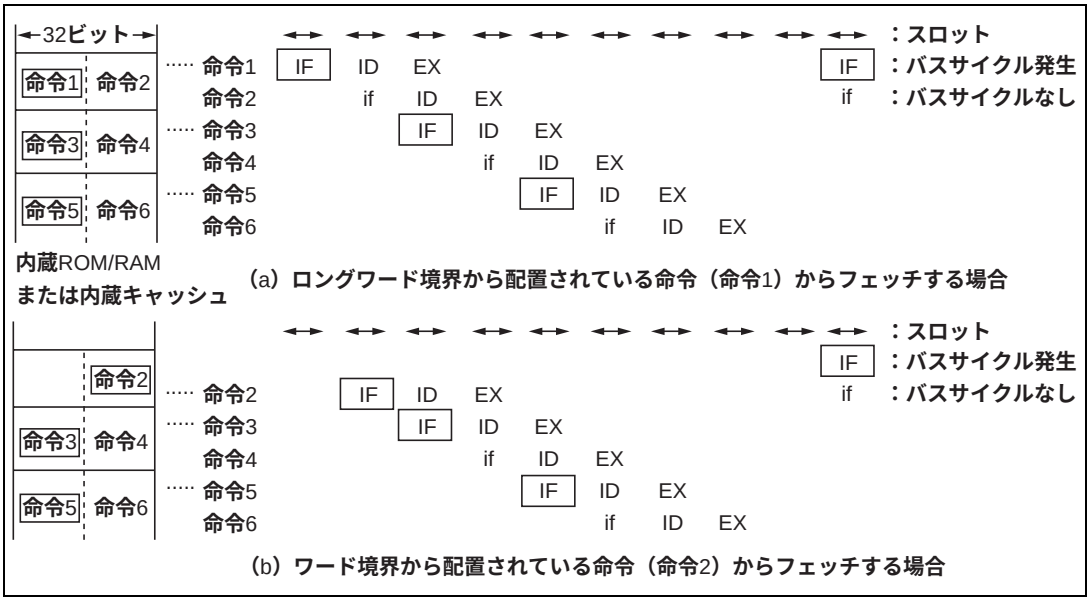


図 7.7 **内蔵** ROM/RAM **または内蔵キャッシュに配置された命令の位置と** IF **の関係**

(3) 内蔵 ROM/RAM または内蔵キャッシュに配置された命令の位置と、IF、MA の競合動作との関係（SH-1/2 の場合）

命令が内蔵 ROM/RAM または内蔵キャッシュに配置されている場合、上記（2）で示したように、バスサイクルが発生しない命令フェッチステージ（"if"と小文字で表します）が存在します。この if と MA が競合したときは、IF と MA の競合と異なり、スロットのスプリットは発生しません。すなわち、if と MA は同時実行可能です。このスロットの実行にかかるステート数は MA によるメモリアクセスサイクル数になります。この様子を**図** 7.8 に示します。

プログラムとして、なるべく MA と IF が競合しないで、MA と if が競合するように書くと、命令実行速度が向上します。すなわち、IF、ID、EX、MA、（WB）という 4（5）段のパイプラインを持つ命令は、内蔵 ROM/RAM または内蔵キャッシュのロングワード境界（命令アドレスの下位 2 ビットが 00 の位置：A1＝0、A0＝0）から配置すると、その命令の MA ステージがその後の if と同一スロットになり、ストールが発生しないようになります。

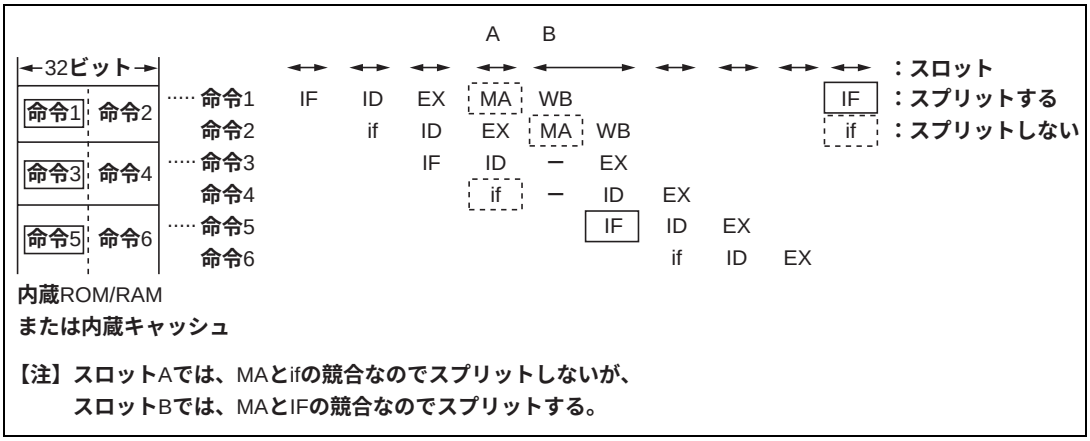


図 7.8 **内蔵** ROM/RAM **または内蔵キャッシュに配置された命令の位置と、**IF**、**MA **の競合動作との関係**

(4) 配置された命令の位置と IF の関係（SH-DSP の場合）

命令フェッチする場合、SH-DSP は 32 ビットでアクセスします。したがって、基本的に IF ステージの 1 回のアクセスで、命令が 16 ビット長の場合は 2 命令、命令が 32 ビット長の場合は 1 命令持ってきます。

まず、ロングワード境界から配置されている命令（命令アドレスの下位 2 ビットが 00 の位置）へ分岐した場合について説明します。この場合は、**図** 7.9（a）に示すように、最初の 3 つの IF ステージ（IF1、IF2、IF3）でバスサイクルを発生し、メモリから 6 ワード分の命令をフェッチします。これよりあとの IF ステージで、メモリから命令をフェッチするバスサイクルを発生するか、あるいは、メモリから命令をフェッチせずにバスサイクルを発生しない（この IF を“ if ”と小文字で表します。必ず if は 1 ステートです。）かは、以下の 3 つの規則に基づいて決定されます。

（a） 2つ前の命令がロングワード境界に配置されている場合、IFステージはバスサイクルを発生する。
（b） 2つ前の命令が32ビット長の場合、IFステージはバスサイクルを発生する。
（c） 2つ前の命令が上記（a）、（b）以外の場合、IFステージはバスサイクルを発生しない。

たとえば、**図** 7.9（a）の if4 は、2 つ前の命令（命令 2）がロングワード境界に配置されておらず、かつ、16 ビット長の命令であるため、バスサイクルを発生しません。また、IF5 は 2 つ前の命令（命令 3）がロングワード境界に配置されているのでバスサイクルを発生し、IF6 は 2 つ前の命令（命令 4）が 32 ビット長であるため、バスサイクルを発生します。

次に、ワード境界から配置されている命令（命令アドレスの下位 2 ビットが 10 の位置）へ分岐した場合について説明します。この場合は、**図** 7.9（b）に示すように、分岐先の命令（命令 2）の ID ステージの前に、2 つの IF ステージ（IF1、IF2）があります。これら 2 つの IF ステージと次の IF ステージ（IF3）でバスサイクルを発生し、メモリから 5 ワード分の命令をフェッチします。これよりあとの IF ステージで、メモリから命令をフェッチするバスサイクルを発生するかしないかは、前述の 3 つの規則に基づいて決定されます。

また、ID ステージの直前の IF ステージでフェッチされる命令は、ID ステージの命令とは一致しない場合があります。たとえば、**図** 7.9（a）の命令 2 の ID ステージの直前にある IF ステージ（IF2）でフェッチされる命令は、命令 3 と命令 4 になります。

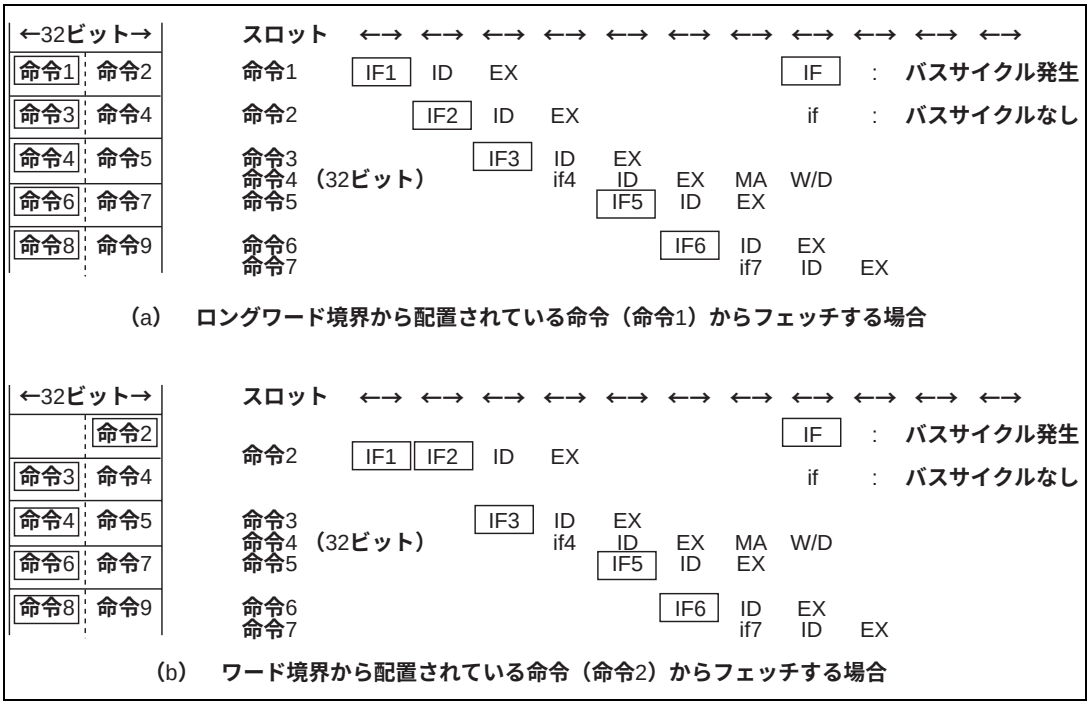


図 7.9　**内蔵** ROM/RAM **または内蔵キャッシュに配置された命令の位置と** IF **の関係**

(5) 配置された命令の位置と、IF、MA の競合動作との関係（SH-DSP の場合）

命令フェッチする場合、上記（2）で示したように、バスサイクルが発生しない命令フェッチステージ（"if"と小文字で表します）が存在します。この if と MA が競合したときは、IF と MA の競合と異なり、スロットのスプリットは発生しません。すなわち、if と MA は同時実行可能です。このスロットの実行にかかるステート数は MA によるメモリアクセスサイクル数になります。この様子を**図** 7.10 に示します。

プログラムとして、なるべく MA と IF が競合しないで、MA と if が競合するように書くと、命令実行速度が向上します。

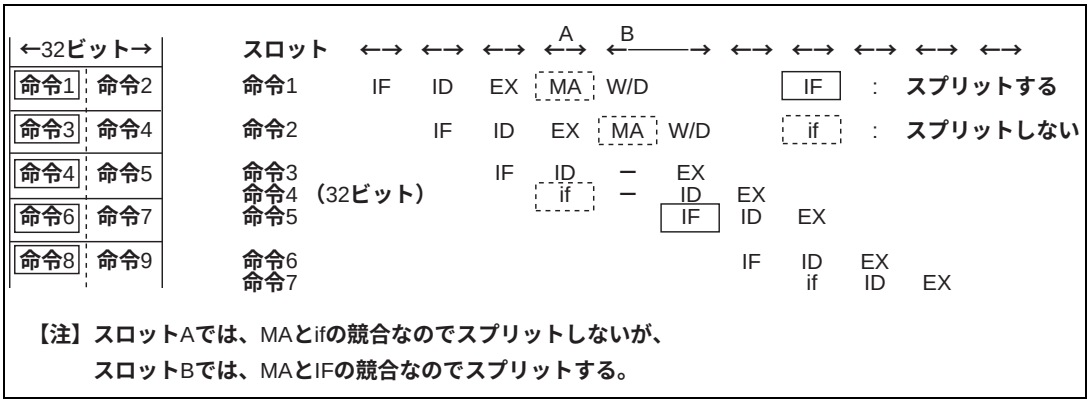


図 7.10 **内蔵** ROM/RAM **または内蔵キャッシュに配置された命令の位置と、**IF**、**MA **の競合動作との関係**

7.2.2 先行命令のデスティネーションレジスタを使うときの競合

(1) ロード命令と次の命令との関係

メモリから CPU レジスタへのロード命令は、デスティネーションレジスタへのデータ戻しがパイプライン最後の WB/DSP ステージで行われます。そのロード命令（ロード命令 1 とします）と、その直後の CPU での演算またはストア命令（命令 2 とします）に着目すると、ロード命令 1 の WB/DSP ステージが終わる前に、命令 2 の EX ステージが始まることになります。

このとき、ロード命令 1 のデスティネーションレジスタを命令 2 が使おうとすると、まだそのレジスタの内容が準備できていないので、命令 1 の MA と命令 2 の EX があるスロットはスプリットされます。ロード命令 1 のデスティネーションレジスタが、命令 2 のソースでなくデスティネーションと同じでもスロットはスプリットされます。

なお、ロード命令 1 のデスティネーションがステータスレジスタ（SR）で、その中のフラグを命令 2 が取り込んで使用する場合（たとえば、ADDC など）もスプリットが発生します。

ただし、次の場合はスプリットしません。

（1） 命令2がロード命令で、そのデスティネーションが、ロード命令1のデスティネーションと同じ場合。

（2） 命令2がMAC @Rm+,@Rn+で、Rmとロード命令1のデスティネーションが同じだった場合。

このスプリットが発生したスロットのステート数は、MA のサイクル数＋IF（または if）のサイクル数になります。その様子を**図** 7.11 および**図** 7.12 に示します。

したがって、ロード命令の直後に、その結果を使う命令を配置するようなプログラムを書くと、

実行速度が低下します。ロード命令の結果を使う命令は、ロード命令の 2 命令以後に置くと速度が低下しません。

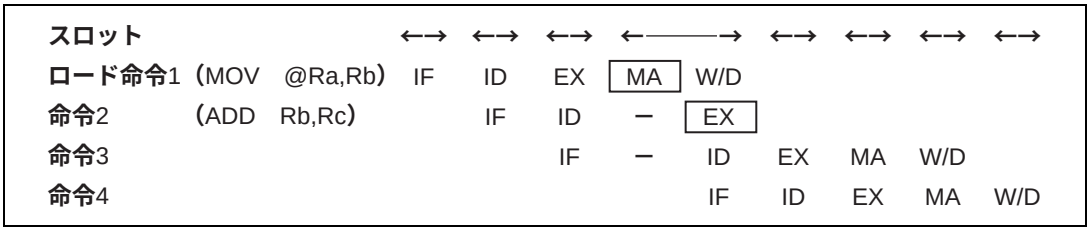


図 7.11 **メモリロード命令によるパイプラインへの影響**

先行する命令でレジスタにデータをロードし、後続のメモリアクセス命令がそのレジスタをアドレスポインタに使うときは、同じように先行する命令の MA ステージのデータロードが終了するまで、メモリアクセスは引き延ばされます。

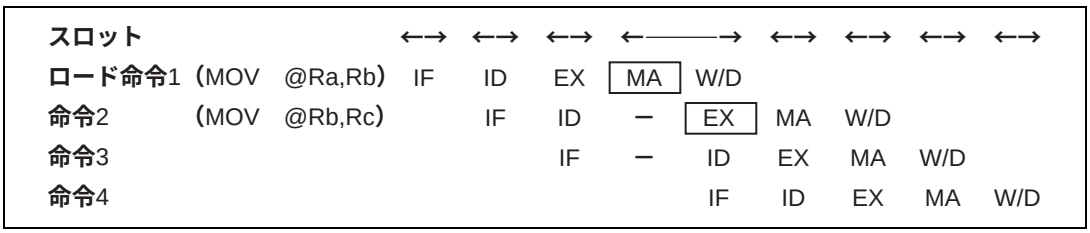


図 7.12 **メモリロード命令によるパイプラインへの影響**

DSP ユニットではすべての演算命令は WB/DSP ステージで実行されるため、転送と演算の競合は発生しません。その様子を**図** 7.13 に示します。ただし、先行する MOV 命令のデスティネーションが、後続の命令のアドレスポインタに使われるときは、**図** 7.12 と同じように競合が発生します。

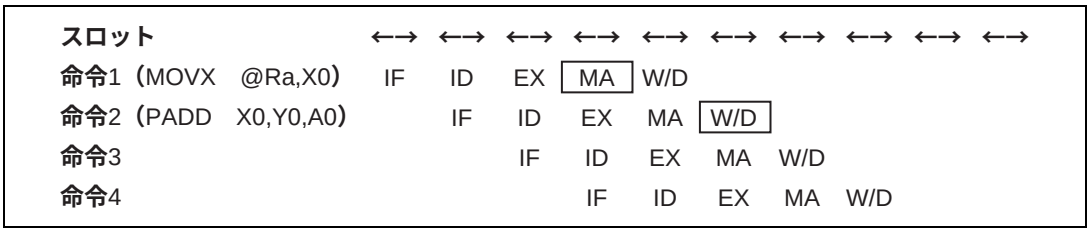


図 7.13 DSP **ユニットでのメモリロード命令によるパイプラインへの影響**

(2) データ演算命令とストア命令の関係

DSP ユニットで DSP 演算を実行し、その結果を次の命令でメモリにストアするとき、メモリロード命令と同じ競合が発生します。このとき、先行する命令の WB/DSP ステージのデータ演算が終了するまで、後続の命令の MA ステージのデータストアは引き延ばされます。

CPU コアでは演算が EX ステージで実行されるのでストールサイクルは発生しません。

DSP ユニットのデータ演算命令とストア命令の関係を**図** 7.14 に示し、CPU コアの関係を**図** 7.15 に示します。

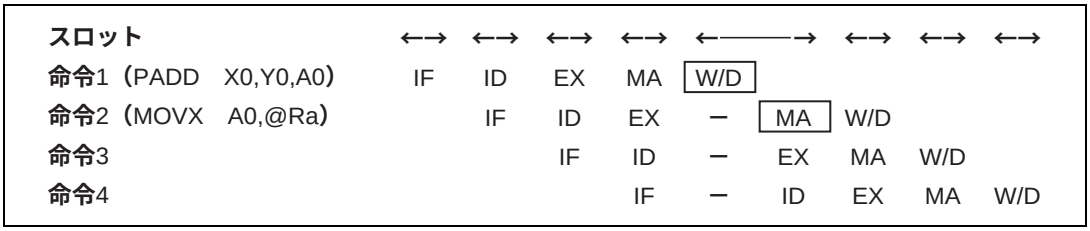


図 7.14 DSP **ユニットでの演算命令とストア命令の関係**

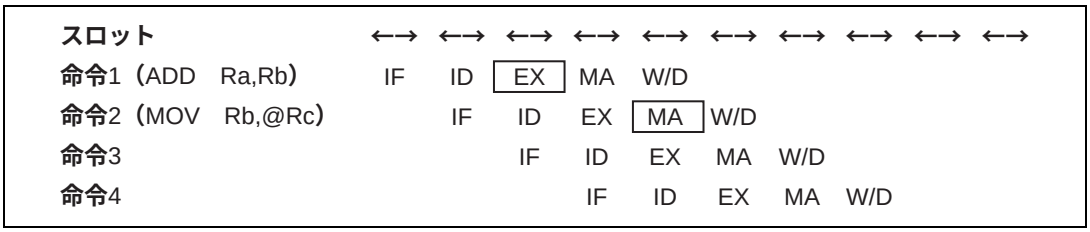


図 7.15 CPU **コアでの演算命令とストア命令の関係**

(3) ロード命令とストア命令の関係

メモリからデスティネーションレジスタにロードし、そのレジスタが後続するストア命令のソースオペランドに指定された場合、先行命令のロードは WB/DSP ステージで実行され、後続命令のストアは MA ステージで実行されます。これらのステージはまったく同じサイクルで実行されます。しかし競合は発生しません。CPU コアも DSP ユニットも同じデータ転送の方法を使っており、この場合内部バスに入力されたデータはデスティネーションレジスタに格納されると同時に、同じデータが再び内部バスに出力されます。

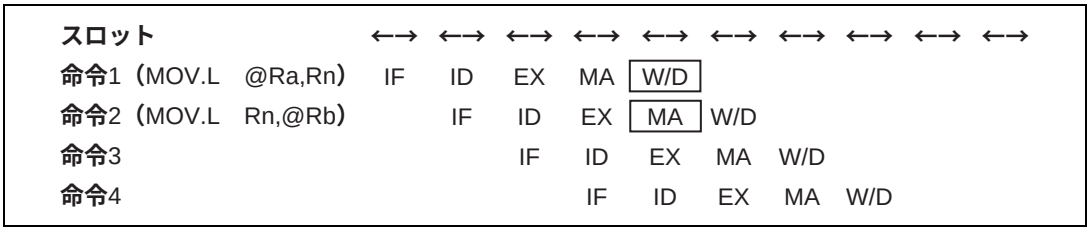


図 7.16 CPU **コアでのロード命令とストア命令の関係**

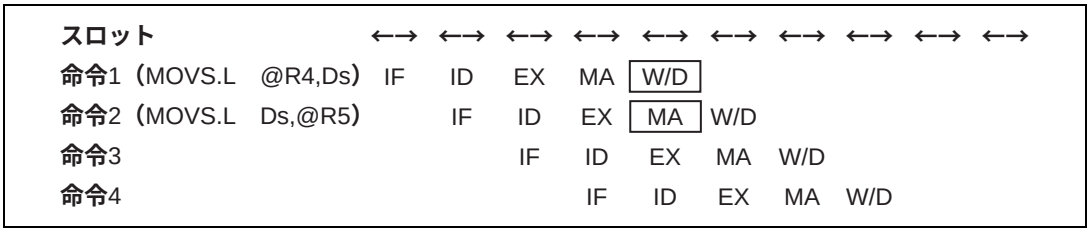


図 7.17 DSP **ユニットでのロード命令とストア命令の関係**

(4) MAC 命令と STS 命令の関係

MAC.W 命令には MA、mm（乗算器アクセス）ステージがそれぞれ 2 つずつあります。MAC.W 命令の次に MACL、MACH レジスタを Rn レジスタにストアする STS 命令が続くときは、MAC.W

命令の mm ステージが終了してから STS 命令の MA ステージが実行されます。MAC.W 命令の次に MACL、MACH レジスタをメモリにストアする STS.L 命令の場合も同じように MAC.W 命令の mm ステージが終了してから STS.L 命令の MA ステージが実行されます。

MAC.L 命令でも同様で、STS 命令、STS.L 命令が続いている場合は、mm ステージが終了（MAC.L 命令の mm ステージは 4 つあります）してから STS、STS.L 命令の MA ステージが実行されます。

スロット												
命令1（MAC.W	@Ra+,@Rb+）	if	ID	EX	MA	MA	mm	mm				
命令2（STS	MACL,Rc）		IF	ー	ID	EX	ー	ー	MA	W/D		
命令3					if	ID	ー	ー	EX	MA	W/D	

図 7.18　MAC.W **命令と** STS **命令の関係**

スロット												
MAC.L	IF	ID	EX	MA	MA	mm	mm	mm	mm			
STS.L MACL,**メモリ**	IF	ー	ID	EX	ー	ー	ー	ー	MA			
次々命令			IF	ID	ー	ー	ー	ー	EX			

図 7.19　MAC.L **命令と** STS.L **命令の関係**

7.2.3　乗算器アクセスによる競合

乗算系命令（積和命令、乗算命令）、積和レジスタ（MACH、MACL）をアクセスする命令は、乗算器アクセスで競合が発生します。

乗算系命令では、乗算器は MA 終了後スロットに関係なく倍精度命令（64 ビット）では 4 ステートの間動作し、単精度命令（32 ビット）のときは 2 ステートの間動作します。乗算系命令（積和命令、乗算命令）の MA（2 つあるときは 2 番目の MA）がその前の乗算系命令の乗算器アクセス（mm）と競合した場合には、その MA のバスサイクルは mm が終了するまで引き延ばされ、その引き伸ばされた MA は 1 つのスロットになります。

また、倍精度命令の次命令の ID は 1 スロット分後ろにストールされます。

乗算系命令、積和レジスタアクセス命令は、MA を持っているので、IF との競合も発生します。

乗算器アクセスの競合の例を**図** 7.20 に示します。ここでは MA と IF の競合は考慮していません。

スロット													
MAC.L	IF	ID	EX	MA	MA	mm	mm	mm	mm				
MAC.L		IF	ー	ID	EX	MA	M	ー	A	mm	mm	mm	mm
次々命令				IF	ー	ID	EX	ー	ー	MA	…		

図 7.20　**乗算器アクセスの競合の例（**MAC.L **命令と** MAC.L **命令の競合）**

7.2.4 DSP レジスタ間転送とメモリ・ロード／ストア動作の競合

DSP ユニットには、従来の SuperH マイコン（SH-1、SH-2）の乗算／積和演算命令の実行に使用されるレジスタ（MACH、MACL）と、DSP 演算命令で使用されるレジスタ（A0、A1、M0、M1、X0、X1、Y0、Y1）との間でデータ転送を行うための命令（PSTS、PLDS）があります。この命令と並列に MOVX.W でメモリロードを行うと、競合が発生します。また、この命令の直後に MOVX.W、MOVS.W または MOVS.L でメモリストアを行うと、競合が発生します。

この命令は定義上、他の DSP 演算命令と同じ命令コード領域に割り付けられているので、ダブルデータ転送命令(MOVX.W、MOVY.W）と並列に記述することができます。しかし動作のためのハードウェアを MOVX.W と共用しているため、PSTS または PLDS と MOVX.W（ロード）を並列記述すると、WB/DSP ステージで競合が生じます。この場合、まず PSTS または PLDS が先に実行され、MOVX.W（ロード）は 1 スロット分後ろにストールされます。

MOVS.W、MOVS.L も MOVX.W と同じハードウェアを使用しますが、DSP 演算命令と並列に記述できないため、先述の競合はありえません。しかしこの命令の直後に MOVX.W、MOVS.W または MOVS.L でストア動作を実行する場合は、PSTS または PLDS の WB/DSP ステージとストア動作の MA ステージとの競合が発生します。

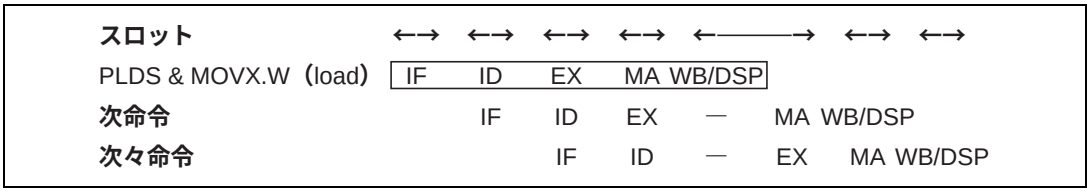


図 3.21 DSP **レジスタ間転送とメモリロード並列動作の競合**

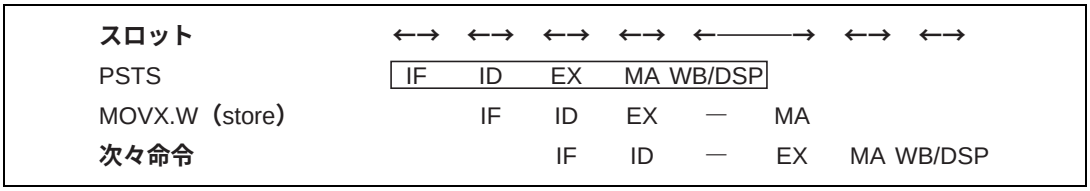


図 3.22 DSP **レジスタ間転送直後のメモリストア動作の競合**

7.3 プログラミングの指針

7.3.1 競合の種類と命令との対応

競合の種類と命令との対応をまとめると次のようになります。

（1） 競合が発生しない命令
（2） メモリアクセス（MA）があり、命令フェッチ（IF）と競合する命令
（3） 直前のDSP演算動作の結果をXバスまたはYバスを使ってメモリストアする命令
（4） メモリアクセス（MA）があり命令フェッチ（IF）と競合し、その上ライトバック（WB/DSP）がありメモリロードの競合を起こすことがある命令
（5） メモリアクセス（MA）があり命令フェッチ（IF）と競合し、その上乗算器をアクセス（mm）し乗算器の競合を起こすことがある命令
（6） メモリアクセス（MA）があり命令フェッチ（IF）と競合し、直前のDSP演算動作の結果をメモリストアする命令
（7） メモリアクセス（MA）があり命令フェッチ（IF）と競合し、乗算器をアクセス（mm）し乗算器の競合を起こすことがあり、その上直前のDSP演算動作の結果を転送することがあり、さらにライトバック（WB/DSP）がありメモリロードの競合を起こすことがある命令
（8） MOV.X、MOVS.WあるいはMOVS.L命令との競合を起こすことがある命令

競合の種類と命令との対応を**表** 7.1 に示します。

表 7.1 **競合の種類と命令との対応**

競合	実行ステート	ステージ段数	命令
なし	1	3	**レジスタ間転送命令** **レジスタ間演算（乗算系命令を除く）** **レジスタ間論理演算命令** **シフト命令** **システム制御** ALU **命令**
	2	3	**無条件分岐**
	3/1	3	**条件分岐**
	2/1	3	**遅延付き条件分岐命令**
	3	3	SLEEP **命令**
	4	5	RTE **命令**
	8	9	TRAP **命令**
	1	5	DSP **演算命令、**MOVX.W**（ロード）、**MOVY.W**（ロード）命令**
●MA **は** IF **と競合します**	1	4	**メモリストア命令** STS.L **命令（**PR**）**
	2	4	STC.L **命令**
	3	6	**メモリ論理演算**
	4	6	TAS **命令**
	1	5	MOVS.W**（ロード）、**MOVS.L**（ロード）命令**
●DSP **演算との競合を起こします**	1	4	MOVX.W**（ストア）、**MOVY.W**（ストア）命令**

競合	実行ステート	ステージ段数	命令
●MA は IF と競合します ●メモリロードの競合を起こします	1	5	メモリロード命令 LDS.L 命令（PR）
	3	5	LDC.L 命令
●MA は IF と競合します ●乗算器との競合を起こします	1	4	レジスタ→MAC 転送命令（MACH/MACL） メモリ→MAC 転送命令（MACH/MACL） MAC→メモリ転送命令（MACH/MACL）
	1(〜3)*	6	乗算命令（PMULS を除く）
	2(〜3)*	7	積和命令
	2(〜4)*	9	倍精度積和命令
	2(〜4)*	9	倍精度乗算命令
●MA は IF と競合します ●DSP 演算との競合を起こします	1	4	MOVS.W（ストア）、MOVS.L（ストア）命令
●MA は IF と競合します ●乗算器との競合を起こします ●DSP 演算との競合を起こします ●メモリロードの競合を起こします	1	5	STS 命令（PR を除く）
●MOVX.W、MOVY.W、MOVS.W あるいは MOVS.L 命令との競合を起こします	1	5	PLDS、PSTS 命令

【注】 * 通常実行ステートを示します。( ) 内の値は、前後の命令との競合関係による実行ステートです。

7.3.2 命令実行速度の向上

プログラムを作る場合は、競合ができるだけ発生しないようにすると、命令実行速度が向上します。競合を発生しないよう次のようにプログラムを作ります。

（1） メモリアクセス（MA）と命令フェッチ（IF）との競合が発生しないよう、MAを持つ命令を、内蔵メモリのロングワード境界（命令アドレスの下位2ビットが00の位置：A1＝0、A0＝0）に配置します。

（2） メモリからのロード命令の直後のCPUコアの演算命令には、ロード命令のデスティネーションレジスタと同じレジスタを使わない命令を配置します。これによりライトバック（WB/DSP）によるメモリロードの競合を発生しないようにします。

（3） 乗算器を使う命令が連続しないように配置します（PMULS命令を除く）。乗算器からの結果の取り出しのためのMACH、MACLレジスタへのアクセスも乗算器を使う命令に連続しないように配置します。これにより乗算器アクセス（mm）による乗算器の競合が発生しないようになります。

（4） DSPユニットでのデータ演算直後に演算結果を格納したレジスタの内容を、メモリあるいはCPUコアのレジスタへデータ転送をしないようにします。何か別の命令を間にはさむことにより、競合が発生しないようになります。

（5） DSPユニットでのPLDS/PSTS命令直後にMOVX.W、MOVY.W、MOVS.WあるいはMOVS.Lによるメモリストアを行わないようにします。また、PLDS/PSTS命令とMOVX.Wによるメモリストア命令をパラレルに記述しないようにします。

7.3.3 ステート数

基本命令は 1 命令を 1 ステートで実行するように設計されています。1 ステート命令の中には、競合の発生しない命令と競合の発生する命令があります。レジスタ間の演算、転送は 1 ステートで実行し競合は発生しません。

競合が発生しなくとも 1 命令が 2 ステート以上になる命令もあります。分岐命令などで分岐先アドレスを変更する命令、メモリ論理演算命令やいくつかのシステム制御命令のようにメモリアクセスを 2 回以上実行する命令、および乗算命令、積和命令のようにメモリアクセスと乗算器アクセスを持つ命令（PMULS を除く）は、2 ステート以上になります。

1 命令が 2 ステート以上になる命令にも、競合の発生しない命令と競合の発生する命令があります。

効率の良いプログラムを作るためには、競合を避けて命令実行速度が向上させると同時に、ステート数の少ない命令を使う配慮が必要です。

7.4 各命令のパイプラインの動作

以下に、各命令のパイプラインの動作を説明します。これに、前述のルールをあわせることで、プログラムのパイプラインの流れ方、および命令実行ステート数を算出することができます。

以下のパイプラインの図において、"命令 A"とあるのが、説明しようとしている命令です。また、命令フェッチステージの IF と if は区別せずに IF と書いてあります。この IF が MA と競合するとスロットがスプリットしますが、そのスプリットの状況については、特別な場合を除き、以下の図では表していません。スロットがスプリットすると判断された場合は、**「7.2.1 命令フェッチ（IF）とメモリアクセス（MA）の競合」**のルールに基づいてパイプラインの動作を考慮してください。

命令のステージ段数と実行ステート数を以下の様式で表示します。

命令のステージ段数と実行ステート数の表様式

分類	区分	命令	実行ステート	ステージ段数	競合
機能別の分類です。	命令を動作の違いで区分しています。	対応する命令をニーモニックで表示します。	競合がない場合の実行ステート数です。	命令のステージの段数です。	発生する競合を表します。

表 7.2 命令のステージ段数と実行ステート数

分類	区分	命令	実行ステート	ステージ段数	競合

データ転送命令	レジスタ－レジスタ間転送命令	MOV #imm,Rn
		MOV Rm,Rn
		MOVA @(disp,PC),R0
		MOVT Rn
		SWAP.B Rm,Rn
		SWAP.W Rm,Rn
		XTRCT Rm,Rn
			1	3	―

	メモリロード命令	MOV.W @(disp,PC),Rn MOV.L @(disp,PC),Rn MOV.B @Rm,Rn MOV.W @Rm,Rn MOV.L @Rm,Rn MOV.B @Rm+,Rn MOV.W @Rm+,Rn MOV.L @Rm+,Rn MOV.B @(disp,Rm),R0 MOV.W @(disp,Rm),R0 MOV.L @(disp,Rm),Rn MOV.B @(R0,Rm),Rn MOV.W @(R0,Rm),Rn MOV.L @(R0,Rm),Rn MOV.B @(disp,GBR),R0 MOV.W @(disp,GBR),R0 MOV.L @(disp,GBR),R0	1	5	●この命令の直後に、この命令のデスティネーションレジスタを使う CPU 演算命令を置くと、競合します。 ●MA は IF と競合します。
	メモリストア命令	MOV.B Rm,@Rn MOV.W Rm,@Rn MOV.L Rm,@Rn MOV.B Rm,@-Rn MOV.W Rm,@-Rn MOV.L Rm,@-Rn MOV.B R0,@(disp,Rn) MOV.W R0,@(disp,Rn) MOV.L Rm,@(disp,Rn) MOV.B Rm,@(R0,Rn) MOV.W Rm,@(R0,Rn) MOV.L Rm,@(R0,Rn) MOV.B R0,@(disp,GBR) MOV.W R0,@(disp,GBR) MOV.L R0,@(disp,GBR)	1	4	●MA は IF と競合します。

分類	区分	命令	実行ステート	ステージ段数	競合

算術演算命令	レジスタ間算術演算命令（乗算系命令を除く）	ADD Rm,Rn
		ADD #imm,Rn
		ADDC Rm,Rn
		ADDV Rm,Rn
		CMP/EQ #imm,R0
		CMP/EQ Rm,Rn
		CMP/HS Rm,Rn
		CMP/GE Rm,Rn
		CMP/HI Rm,Rn
		CMP/GT Rm,Rn
		CMP/PZ Rn
		CMP/PL Rn
		CMP/STR Rm,Rn
		DIV1 Rm,Rn
		DIV0S Rm,Rn
		DIV0U
		DT Rn
		EXTS.B Rm,Rn
		EXTS.W Rm,Rn
		EXTU.B Rm,Rn
		EXTU.W Rm,Rn
		NEG Rm,Rn
		NEGC Rm,Rn
		SUB Rm,Rn
		SUBC Rm,Rn
		SUBV Rm,Rn
			1	3	―

	乗算命令	MULS.W Rm,Rn MULU.W Rm,Rn	1(〜3)*1	6/7*3	●この命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生します。 ●MA は IF と競合します。
	積和命令	MAC.W @Rm+,@Rn+	2(〜3)*1	7/8*3	●この命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生します。 ●MA は IF と競合します。
	倍精度積和命令	MAC.L @Rm+,@Rn+	2(〜4)*1	9	●この命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生します。 ●MA は IF と競合します。
	倍精度乗算命令	DMULS.L Rm,Rn DMULU.L Rm,Rn MUL.L Rm,Rn	2(〜4)*1	9	●この命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生します。 ●MA は IF と競合します。

分類	区分	命令	実行ステート	ステージ段数	競合
論理演算命令	レジスター レジスタ間 論理演算命令	AND Rm,Rn AND #imm,R0 NOT Rm,Rn OR Rm,Rn OR #imm,R0 TST Rm,Rn TST #imm,R0 XOR Rm,Rn XOR #imm,R0	1	3	—
	メモリ 論理演算命令	AND.B #imm,@(R0,GBR) OR.B #imm,@(R0,GBR) TST.B #imm,@(R0,GBR) XOR.B #imm,@(R0,GBR)	3	6	●MA は IF と競合します。
	TAS 命令	TAS.B @Rn	4	6	●MA は IF と競合します。
シフト命令	シフト命令	ROTL Rn ROTR Rn ROTCL Rn ROTCR Rn SHAL Rn SHAR Rn SHLL Rn SHLR Rn SHLL2 Rn SHLR2 Rn SHLR8 Rn SHLL8 Rn SHLL16 Rn SHLR16 Rn	1	3	—
分岐命令	条件分岐命令	BF label BT label	3/1*[2]	3	—
	遅延付き条件分岐命令	BF/S label BT/S label	2/1*[2]	3	—
	無条件分岐命令	BRA label BRAF Rm BSR label BSRF Rm JMP @Rm JSR @Rm RTS	2	3	—

分類	区分	命令	実行ステート	ステージ段数	競合

システム制御命令（続く）	システム制御 ALU 命令	CLRT
		LDC Rm,SR
		LDC Rm,GBR
		LDC Rm,VBR
		LDC Rm,MOD
		LDC Rm,RE
		LDC Rm,RS
		LDRE @(disp,PC)
		LDRS @(disp,PC)
		LDS Rm,PR
		NOP
		SETRC Rm
		SETRC #imm
		SETT
		STC SR,Rn
		STC GBR,Rn
		STC VBR,Rn
		STC MOD,Rn
		STC RE,Rn
		STC RS,Rn
		STS PR,Rn
			1	3	—

	LDS.L 命令（PR）	LDS.L @Rm+,PR	1	5	●この命令の直後に、この命令のデスティネーションレジスタを使う命令を置くと、競合します。 ●MA は IF と競合します。
	STS.L 命令（PR）	STS.L PR,@-Rn	1	4	●MA は IF と競合します。
	LDC.L 命令	LDC.L @Rm+,SR LDC.L @Rm+,GBR LDC.L @Rm+,VBR LDC.L @Rm+,MOD LDC.L @Rm+,RE LDC.L @Rm+,RS	3	5	●この命令の直後に、この命令のデスティネーションレジスタを使う命令を置くと、競合します。 ●MA は IF と競合します。
	STC.L 命令	STC.L SR,@-Rn STC.L GBR,@-Rn STC.L VBR,@-Rn STC.L MOD,@-Rn STC.L RE,@-Rn STC.L RS,@-Rn	2	4	●MA は IF と競合します。

分類	区分	命令	実行ステート	ステージ段数	競合

システム制御命令（続き）	レジスタ→MAC 転送命令	CLRMAC
		LDS Rm,MACH
		LDS Rm,MACL
			1	4	●乗算器との競合を起こします。
					●MA は IF と競合します。

	レジスタ→DSP 転送命令	LDS Rm,DSR
		LDS Rm,A0
		LDS Rm,X0
		LDS Rm,X1
		LDS Rm,Y0
		LDS Rm,Y1
			1	4	—

	メモリ→MAC 転送命令	LDS.L @Rm+,MACH LDS.L @Rm+,MACL	1	4	●乗算器との競合を起こします。 ●MA は IF と競合します。
	メモリ→DSP 転送命令	LDS.L @Rm+,DSR LDS.L @Rm+,A0 LDS.L @Rm+,X0 LDS.L @Rm+,X1 LDS.L @Rm+,Y0 LDS.L @Rm+,Y1	1	4	—
	MAC→レジスタ転送命令	STS MACH,Rn STS MACL,Rn	1	5	●乗算器との競合を起こします。 ●この命令の直後に、この命令のデスティネーションレジスタを使う命令を置くと、競合します。 ●MA は IF と競合します。 ●DSP 演算との競合を起こします。
	DSP→レジスタ転送命令	STS DSR,Rn STS A0,Rn STS X0,Rn STS X1,Rn STS Y0,Rn STS Y1,Rn			
	MAC→メモリ転送命令	STS.L MACH,@-Rn STS.L MACL,@-Rn	1	4	●乗算器との競合を起こします。 ●MA は IF と競合します。
	DSP→メモリ転送命令	STS.L DSR, @-Rn STS.L A0, @-Rn STS.L X0, @-Rn STS.L X1, @-Rn STS.L Y0, @-Rn STS.L Y1, @-Rn	1	4	—
	RTE 命令	RTE	4	5	—
	TRAP 命令	TRAPA #imm	8	9	—
	SLEEP 命令	SLEEP	3	3	—

分類	区分	命令	実行ステート	ステージ段数	競合
DSP データ転送命令	X メモリロード命令	NOPX MOVX.W @Ax,Dx MOVX.W @Ax+,Dx MOVX.W @Ax+Ix,Dx	1	5	—
	X メモリストア命令	MOVX.W Da, @Ax MOVX.W Da, @Ax+ MOVX.W Da, @Ax+Ix	1	4	●DSP 演算との競合を起こします。
	Y メモリロード命令	NOPY MOVY.W @Ay,Dy MOVY.W @Ay+,Dy MOVY.W @Ay+Iy,Dy	1	5	—
	Y メモリストア命令	MOVY.W Da, @Ay MOVY.W Da, @Ay+ MOVY.W Da, @Ay+Iy	1	4	●DSP 演算との競合を起こします。
	シングルロード命令	MOVS.W @-As,Ds MOVS.W @As,Ds MOVS.W @As+, Ds MOVS.W @As+Is, Ds MOVS.L @-As,Ds MOVS.L @As,Ds MOVS.L @As+, Ds MOVS.L @As+Is, Ds	1	5	●MA は IF と競合を起こします。
	シングルストア命令	MOVS.W Ds, @-As MOVS.W Ds, @As MOVS.W Ds, @As+ MOVS.W Ds, @As+Is MOVS.L Ds, @-As MOVS.L Ds, @As MOVS.L Ds, @As+ MOVS.L Ds, @As+Is	1	4	●MA は IF と競合を起こします。 ●DSP 演算との競合を起こします。
DSP 演算命令（続く）	ALU 算術演算命令（続く）	PADD Sx, Sy,Dz(Du) DCT PADD Sx, Sy,Dz DCF PADD Sx, Sy,Dz PSUB Sx, Sy,Dz(Du) DCT PSUB Sx, Sy,Dz DCF PSUB Sx, Sy,Dz PCOPY Sx,Dz DCT PCOPY Sx,Dz DCF PCOPY Sx,Dz PCOPY Sy,Dz DCT PCOPY Sy,Dz DCF PCOPY Sy,Dz	1	5	—

分類	区分	命令	実行ステート	ステージ段数	競合
DSP 演算命令（続く）	ALU 算術演算命令（続き）	PDMSB Sx,Dz	1	5	—
		DCT PDMSB Sx,Dz			
		DCF PDMSB Sx,Dz			
		PDMSB Sy,Dz			
		DCT PDMSB Sy,Dz			
		DCF PDMSB Sy,Dz			
		PINC Sx,Dz			
		DCT PINC Sx,Dz			
		DCF PINC Sx,Dz			
		PINC Sy,Dz			
		DCT PINC Sy,Dz			
		DCF PINC Sy,Dz			
		PNEG Sx,Dz			
		DCT PNEG Sx,Dz			
		DCF PNEG Sx,Dz			
		PNEG Sy,Dz			
		DCT PNEG Sy,Dz			
		DCF PNEG Sy,Dz			
		PDEC Sx,Dz			
		DCT PDEC Sx,Dz			
		DCF PDEC Sx,Dz			
		PDEC Sy,Dz			
		DCT PDEC Sy,Dz			
		DCF PDEC Sy,Dz			
		PCLR Dz			
		DCT PCLR Dz			
		DCF PCLR Dz			
		PADDC Sx,Sy,Dz			
		PSUBC Sx,Sy,Dz			
		PCMP Sx,Sy			
		PABS Sx,Dz			
		PABS Sy,Dz			
		PRND Sx,Dz			
		PRND Sy,Dz			

分類	区分	命令	実行ステート	ステージ段数	競合
DSP 演算命令（続き）	ALU 論理演算命令	POR Sx,Sy,Dz DCT POR Sx,Sy,Dz DCF POR Sx,Sy,Dz PAND Sx,Sy,Dz DCT PAND Sx,Sy,Dz DCF PAND Sx,Sy,Dz PXOR Sx,Sy,Dz DCT PXOR Sx,Sy,Dz DCF PXOR Sx,Sy,Dz	1	5	―
	シフト命令	PSHA Sx,Sy,Dz DCT PSHA Sx,Sy,Dz DCF PSHA Sx,Sy,Dz PSHA #imm,Dz PSHL Sx,Sy,Dz DCT PSHL Sx,Sy,Dz DCF PSHL Sx,Sy,Dz PSHL #imm,Dz	1	5	―
	乗算命令	PMULS Se,Sf,Dg	1	5	―
	レジスタ間転送命令	PSTS MACH,Dz DCT PSTS MACH,Dz DCF PSTS MACH,Dz PSTS MACL,Dz DCT PSTS MACL,Dz DCF PSTS MACL,Dz PLDS Dz,MACH DCT PLDS Dz,MACH DCF PLDS Dz,MACH PLDS Dz,MACL DCT PLDS Dz,MACL DCF PLDS Dz,MACL	1	5	●MOVX.W,MOVY.W,MOVS.W, MOVS.L との競合を起こします。

【注】 *1 通常実行ステートを示します。
（ ）内の値は、後続の命令との競合関係による実行ステート数です。
*2 分岐しないときは 1 ステートになります。
*3 SH-1 CPU のステージ段数です。

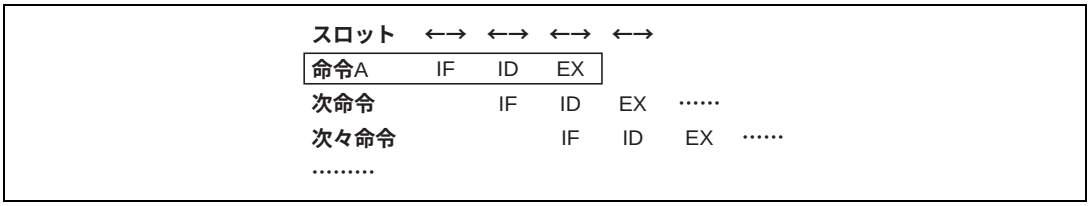
7.4.1 データ転送命令

(1) レジスタ－レジスタ間転送命令（共通）

●命令の種類

MOV	#imm,Rn	SWAP.B	Rm,Rn
MOV	Rm,Rn	SWAP.W	Rm,Rn
MOVA	@(disp,PC),R0	XTRACT	Rm,Rn
MOVT	Rn		

●パイプライン



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。EX ステージで、ALU をとおしてデータ転送を行います。

(2) メモリロード命令（共通）

●命令の種類

MOV.W	@(disp,PC),Rn	MOV.B	@(disp,Rm),R0
MOV.L	@(disp,PC),Rn	MOV.W	@(disp,Rm),R0
MOV.B	@Rm,Rn	MOV.L	@(disp,Rm),Rn
MOV.W	@Rm,Rn	MOV.B	@(R0,Rm),Rn
MOV.L	@Rm,Rn	MOV.W	@(R0,Rm),Rn
MOV.B	@Rm+,Rn	MOV.L	@(R0,Rm),Rn
MOV.W	@Rm+,Rn	MOV.B	@(disp,GBR),R0
MOV.L	@Rm+,Rn	MOV.W	@(disp,GBR),R0
		MOV.L	@(disp,GBR),R0

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB	
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA、WB の 5 段です。この命令の直後に、この命令のデスティネーションレジスタを使う命令を置くと、競合が発生します（「7.2.2 **先行命令のデスティネーションレジスタを使うときの競合**」参照）。

(3) メモリストア命令（共通）

●命令の種類

MOV.B	Rm,@Rn	MOV.B	Rm,@(R0,Rn)
MOV.W	Rm,@Rn	MOV.W	Rm,@(R0,Rn)
MOV.L	Rm,@Rn	MOV.L	Rm,@(R0,Rn)
MOV.B	Rm,@-Rn	MOV.B	R0,@(disp,GBR)
MOV.W	Rm,@-Rn	MOV.W	R0,@(disp,GBR)
MOV.L	Rm,@-Rn	MOV.L	R0,@(disp,GBR)
MOV.B	R0,@(disp,Rn)		
MOV.W	R0,@(disp,Rn)		
MOV.L	Rm,@(disp,Rn)		

●パイプライン

スロット	←→	←→	←→	←→	←→	
命令A	IF	ID	EX	MA		
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA の 4 段で終了します。レジスタへのデータの戻しがないので WB ステージはありません。

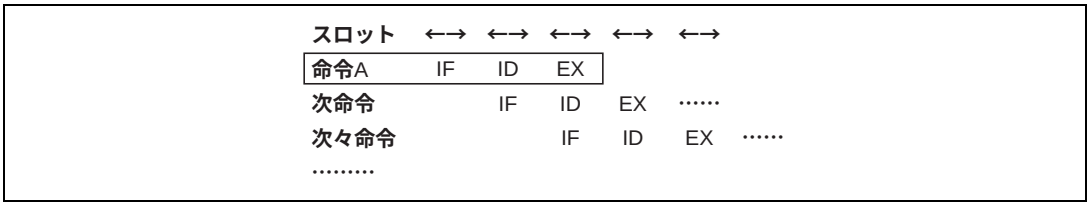
7.4.2 算術演算命令

(1) レジスタ間算術演算命令（乗算系命令を除く）（共通もしくは SH-2、SH-DSP）

●命令の種類

ADD	Rm,Rn	DIV1	Rm,Rn
ADD	#imm,Rn	DIV0S	Rm,Rn
ADDC	Rm,Rn	DIV0U	
ADDV	Rm,Rn	DT	Rn（SH-2、SH-DSP）
CMP/EQ	#imm,R0	EXTS.B	Rm,Rn
CMP/EQ	Rm,Rn	EXTS.W	Rm,Rn
CMP/HS	Rm,Rn	EXTU.B	Rm,Rn
CMP/GE	Rm,Rn	EXTU.W	Rm,Rn
CMP/HI	Rm,Rn	NEG	Rm,Rn
CMP/GT	Rm,Rn	NEGC	Rm,Rn
CMP/PZ	Rn	SUB	Rm,Rn
CMP/PL	Rn	SUBC	Rm,Rn
CMP/STR	Rm,Rn	SUBV	Rm,Rn

●パイプライン



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。EX ステージで、ALU をとおしてデータ演算は完結します。

(2) 積和命令（SH-1）

【命令の種類】

MAC.W @Rm+,@Rn+

【パイプライン】

	↔	↔	↔	↔	↔	↔	↔	↔ ：スロット
MAC.W	IF	ID	EX	MA	MA	mm	mm	mm
次命令		IF	—	ID	EX	MA	WB	
次々命令				IF	ID	EX	MA	WB
……								

【動作説明】

パイプラインは、IF、ID、EX、MA、MA、mm、mm、mm の 8 段で終了します。2 番目の MA は、メモリ読み込みと共に乗算器のアクセスも行います。mm は乗算器が動作している状態を表しています。mm は最後の MA 終了後、スロットに関係なく 3 ステートの間、動作します。また、MAC.W 命令の次命令の ID は 1 スロット分後ろにストールされます。MAC.W 命令の 2 個の MA も、IF と競合する場合は、「7.4 **命令フェッチ（IF）とメモリアクセス（MA）の競合**」で説明したようにスロットがスプリットします。

MAC.W 命令の後ろに乗算器を使わない命令がくる場合は、MAC.W 命令は IF、ID、EX、MA、MA の 5 段パイプライン命令とみなして考えてかまいません。すなわち、この場合は、次命令の ID が 1 スロット分ストールするだけで、あとは、通常のパイプライン動作と同様になります。

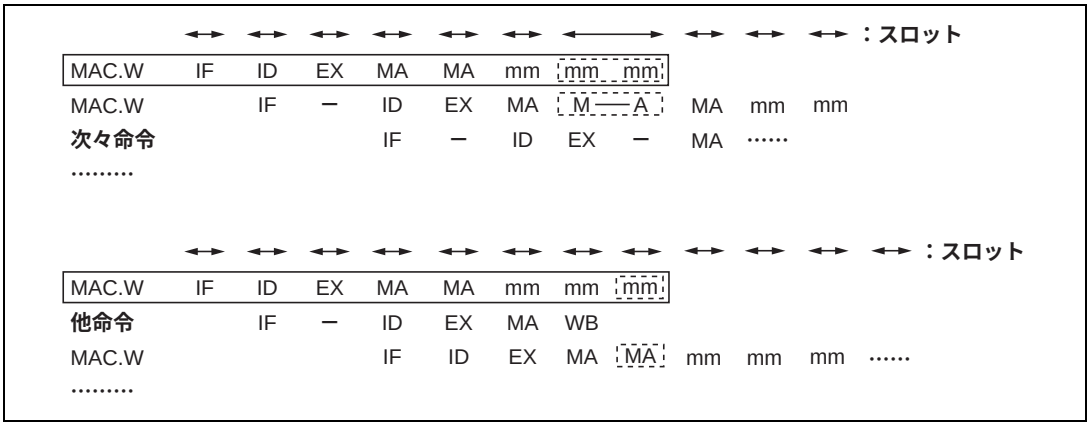
しかし、MAC.W 命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生しますので、通常の動作と異なってきます。この場合は、以下のケースが考えられます。

（a） MAC.W 命令の直後に、MAC.W 命令が連続してくる場合
（b） MAC.W 命令の直後に、MULS.W 命令が連続してくる場合
（c） MAC.W 命令の直後に、STS（レジスタ）命令がくる場合
（d） MAC.W 命令の直後に、STS.L（メモリ）命令がくる場合
（e） MAC.W 命令の直後に、LDS（レジスタ）命令がくる場合
（f） MAC.W 命令の直後に、LDS.L（メモリ）命令がくる場合

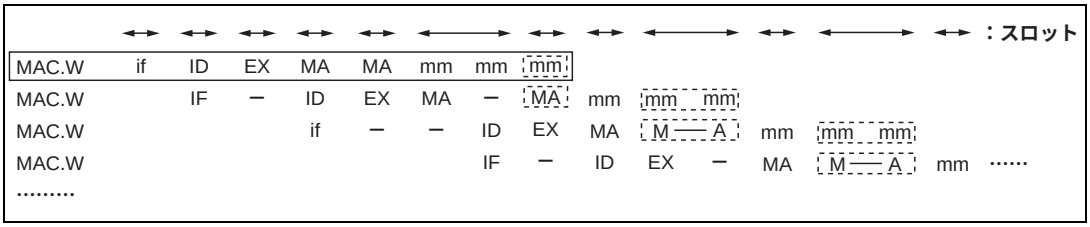
(a) MAC.W命令の直後に、MAC.W命令が連続してくる場合

MAC.W命令の2番目のMAが、その前の乗算系命令によって発生したmmと競合した場合、そのMAのバスサイクルはmmが終了するまで引き延ばされ（下図のM――A）、その引き延ばされたMAは1つのスロットになります。

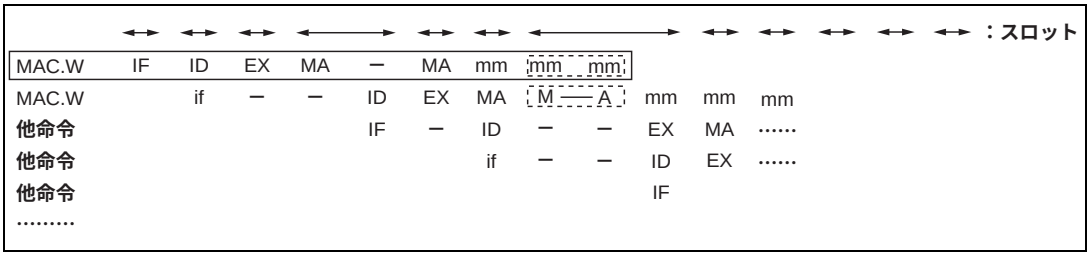
MAC.W命令とMAC.W命令の間に、乗算器と無関係な命令が1つ以上入ると、MAC.W命令どうしの乗算器の競合によるストールはなくなります。



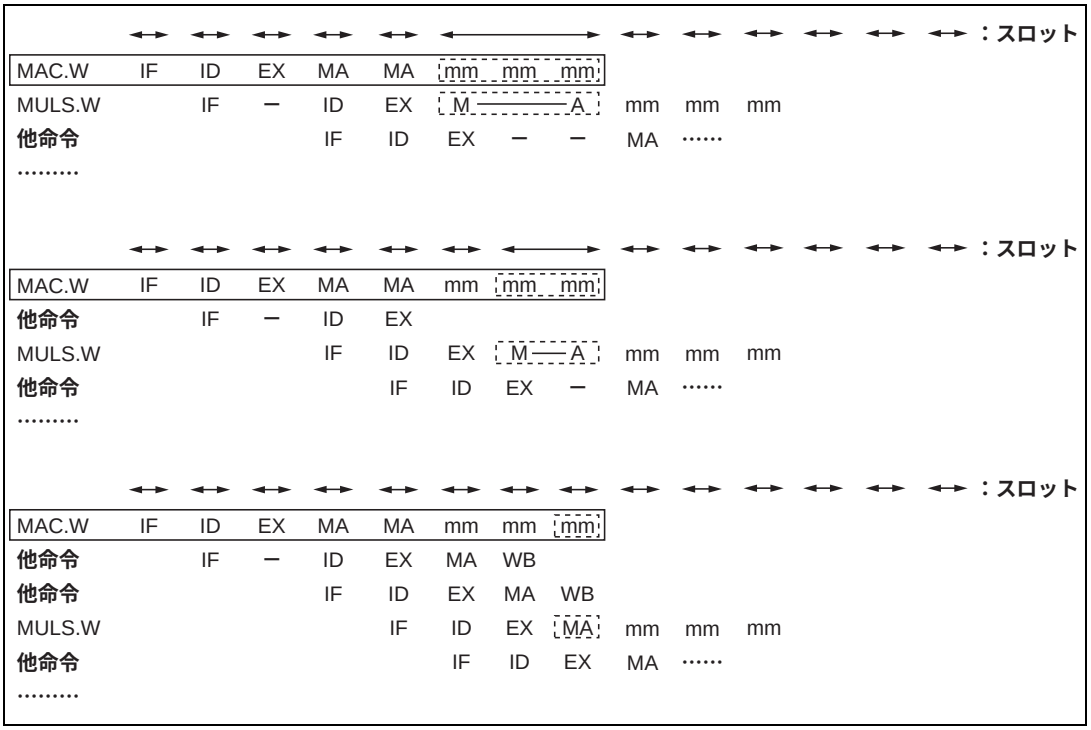
MAC.Wの連続により、MAとIFの競合で命令実行がずれても乗算器の競合がなくなる場合があります。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



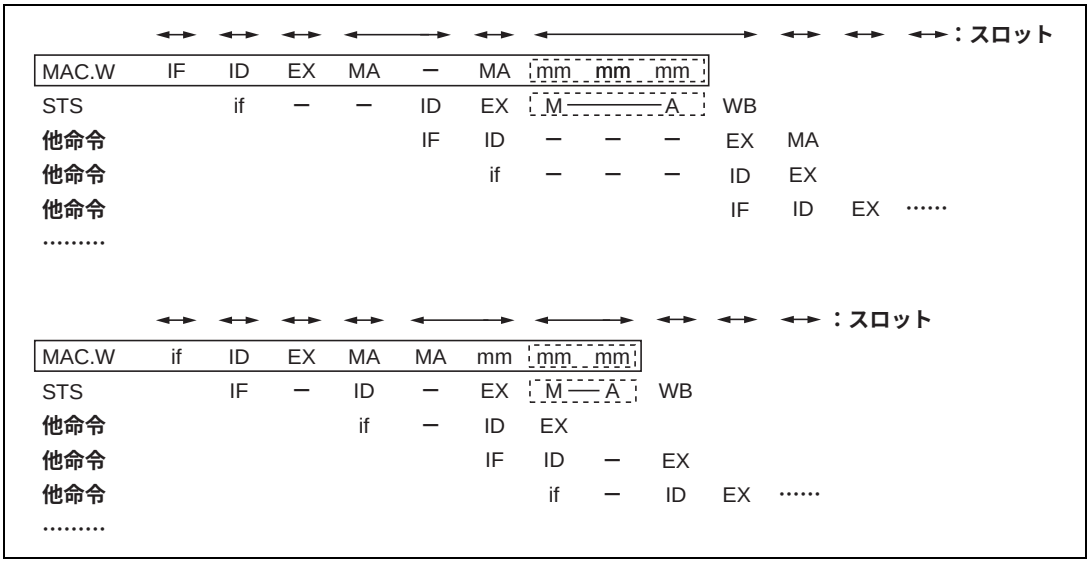
MAC.W命令の2番目のMAがmm終了で引き延ばされている場合、このMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照にしてください。この図については、MAとIFの競合を考慮して書いてあります。



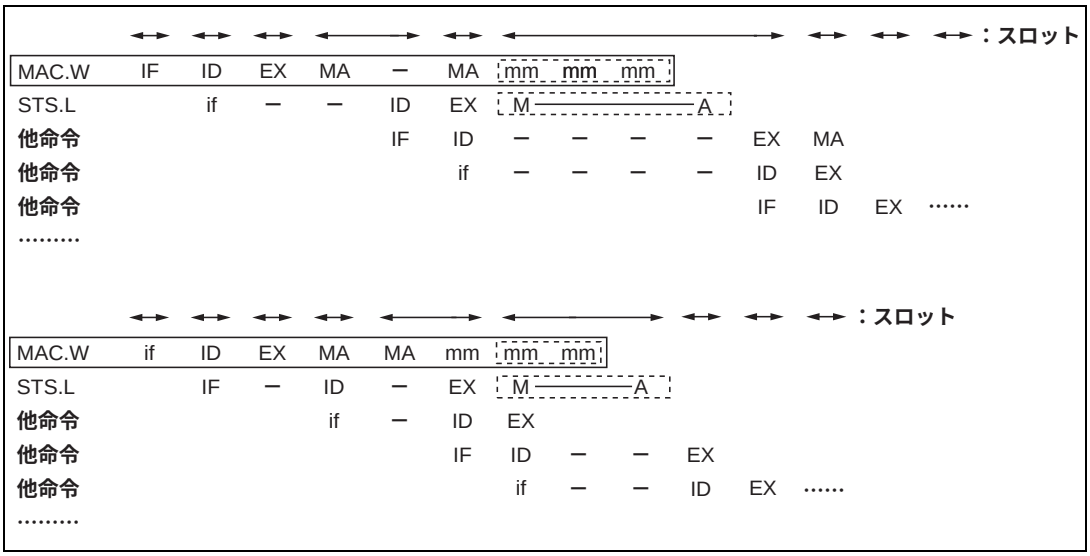
（b） MAC.W命令直後に、MULS.W命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがあります。MAC.W命令の乗算器の動作中（mm）にMULS.WのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。MAC.WとMULS.Wの間に乗算器と無関係な命令が2つ以上入るとMAC.WとMULS.Wの競合によるストールはなくなります。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。



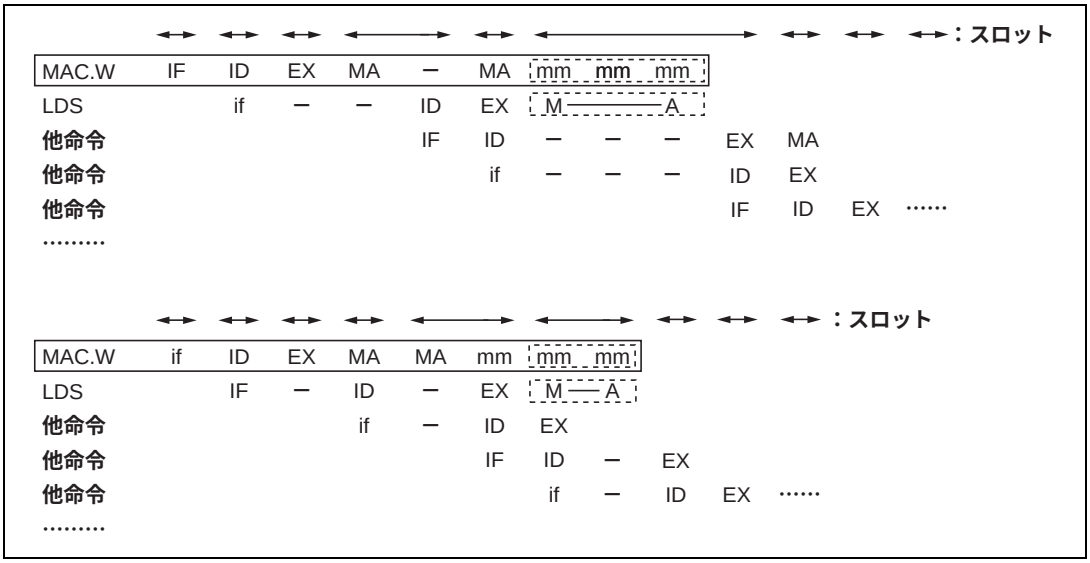
(c) MAC.W命令の直後に、STS（レジスタ）命令がくる場合
STS命令で、MACレジスタの内容を汎用レジスタにストアする場合、後述のとおり、STS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



（d） MAC.W命令の直後に、STS.L（メモリ）命令が来る場合
STS命令で、MACレジスタの内容をメモリにストアする場合、後述のとおり、STS命令には乗算器のアクセスと、メモリライトのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了してから1ステート経過するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。
これらの図については、MAとIFの競合を考慮して書いてあります。

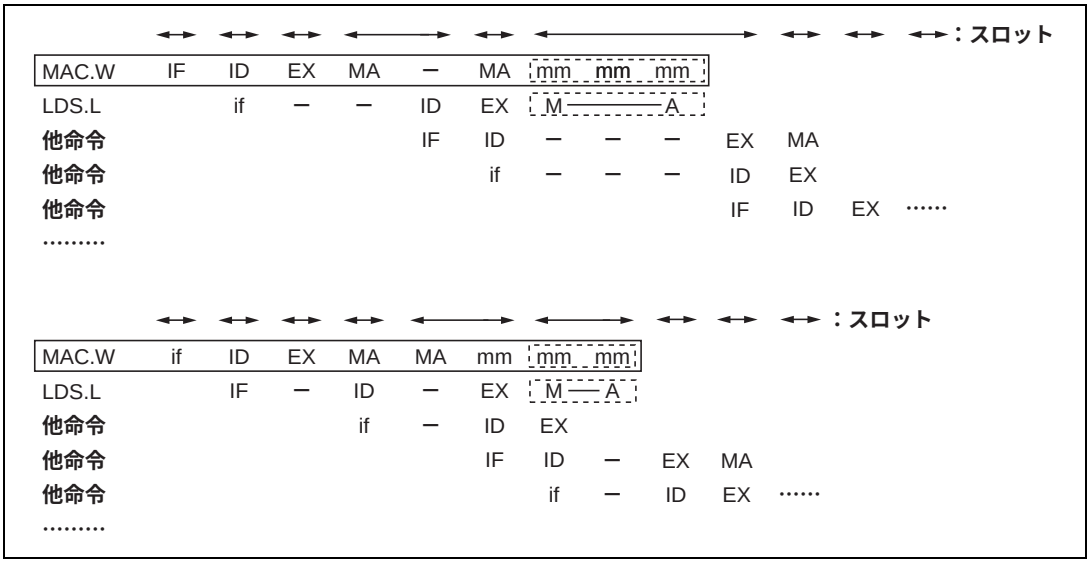


(e) MAC.W命令の直後に、LDS（レジスタ）命令がくる場合
LDS命令で、MACレジスタの内容を汎用レジスタからロードする場合、後述のとおり、LDS命令には乗算器のアクセスのためのMAステージがはいります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(f) MAC.W命令の直後に、LDS.L（メモリ）命令がくる場合

LDS命令で、MACレジスタの内容をメモリからロードする場合、後述のとおり、LDS命令にはメモリのアクセスと乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(3) **積和命令（SH-2、SH-DSP）**

●命令の種類

MAC.W @Rm+,@Rn+

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	MA	mm	mm	
次命令		IF	ー	ID	EX	MA	WB	
次々命令				IF	ID	EX	MA	WB
……								

●動作説明

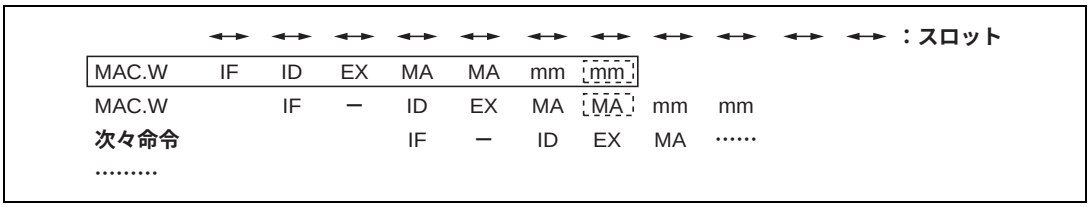
パイプラインは、IF、ID、EX、MA、MA、mm、mm の 7 段で終了します。2 番目の MA は、メモリ読み込みとともに乗算器のアクセスも行います。mm は乗算器が動作している状態を表しています。mm は最後の MA 終了後、スロットに関係なく 2 ステートの間、動作します。また、MAC.W 命令の次命令の ID は 1 スロット分後ろにストールされます。MAC.W 命令の 2 個の MA も、IF と競合する場合は、**「**7.2.1 **命令フェッチ（**IF**）とメモリアクセス（**MA**）の競合」**で説明したようにスロットがスプリットします。

MAC.W 命令の後ろに乗算器を使わない命令がくる場合は、MAC.W 命令は IF、ID、EX、MA、MA の 5 段パイプライン命令と考えてかまいません。すなわち、この場合は、次命令の ID が 1 スロット分ストールするだけで、あとは、通常のパイプライン動作と同様になります。

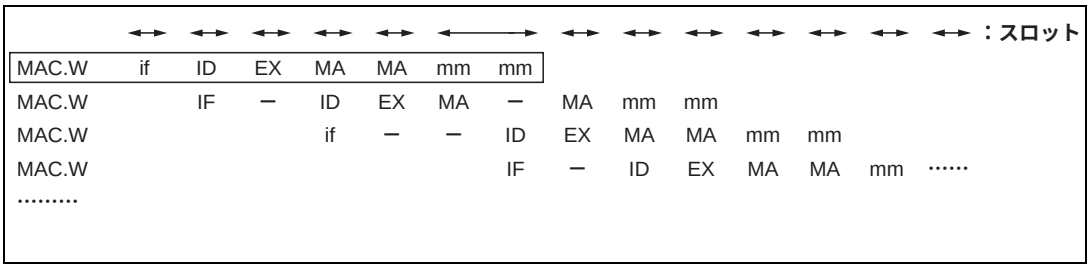
しかし、MAC.W 命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生しますので、通常の動作と異なってきます。この場合は、以下のケースが考えられます。

（a） MAC.W 命令の直後に、MAC.W 命令が連続してくる場合
（b） MAC.W 命令の直後に、MAC.L 命令が連続してくる場合
（c） MAC.W 命令の直後に、MULS.W 命令が連続してくる場合
（d） MAC.W 命令の直後に、DMULS.L 命令が連続してくる場合
（e） MAC.W 命令の直後に、STS（レジスタ）命令がくる場合
（f） MAC.W 命令の直後に、STS.L（メモリ）命令がくる場合
（g） MAC.W 命令の直後に、LDS（レジスタ）命令がくる場合
（h） MAC.W 命令の直後に、LDS.L（メモリ）命令がくる場合

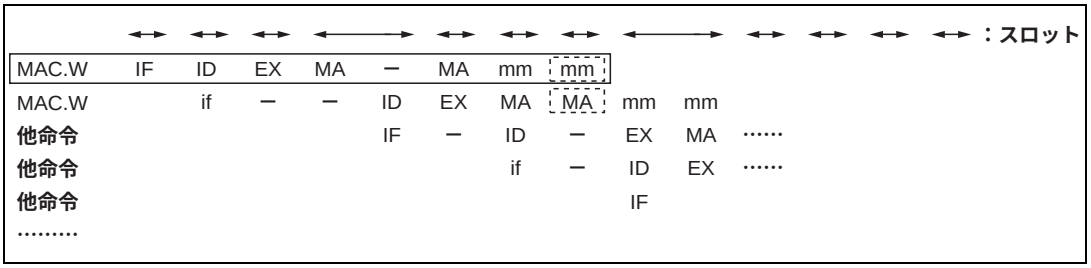
(a) MAC.W命令の直後に、MAC.W命令が連続してくる場合
MAC.W命令の2番目のMAと、その前の乗算系命令によって発生したmmとは競合しません。



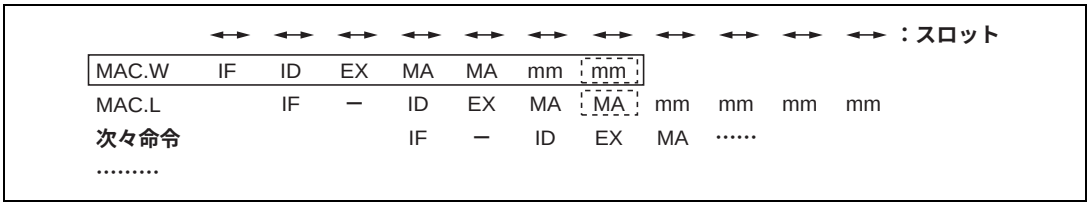
MAC.Wの連続により、MAとIFの競合で命令実行がずれる場合もあります。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



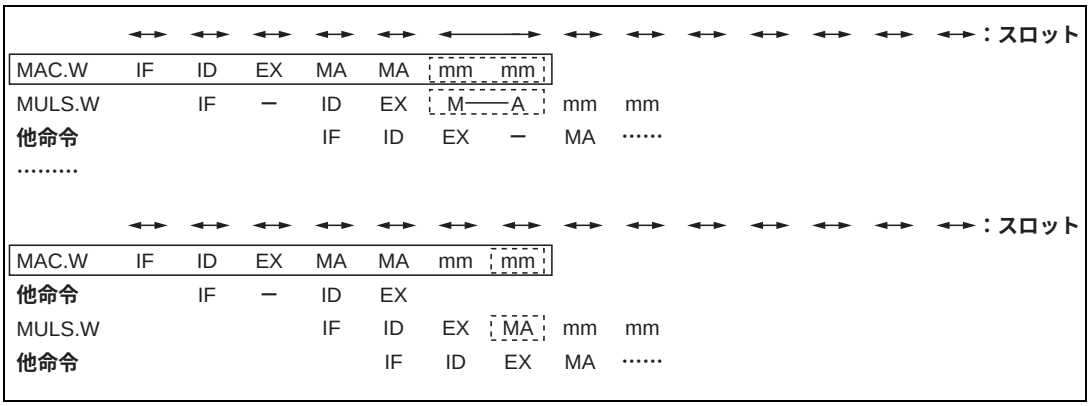
MAC.W命令の2番目のMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



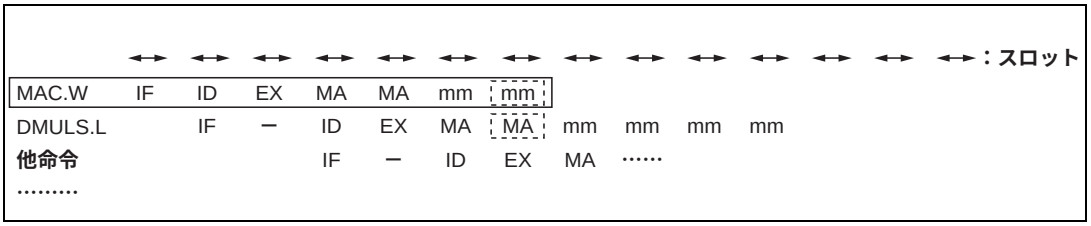
(b) MAC.W命令の直後に、MAC.L命令が連続してくる場合
MAC.W命令の2番目のMAと、その前の乗算系命令によって発生したmmとは競合しません。



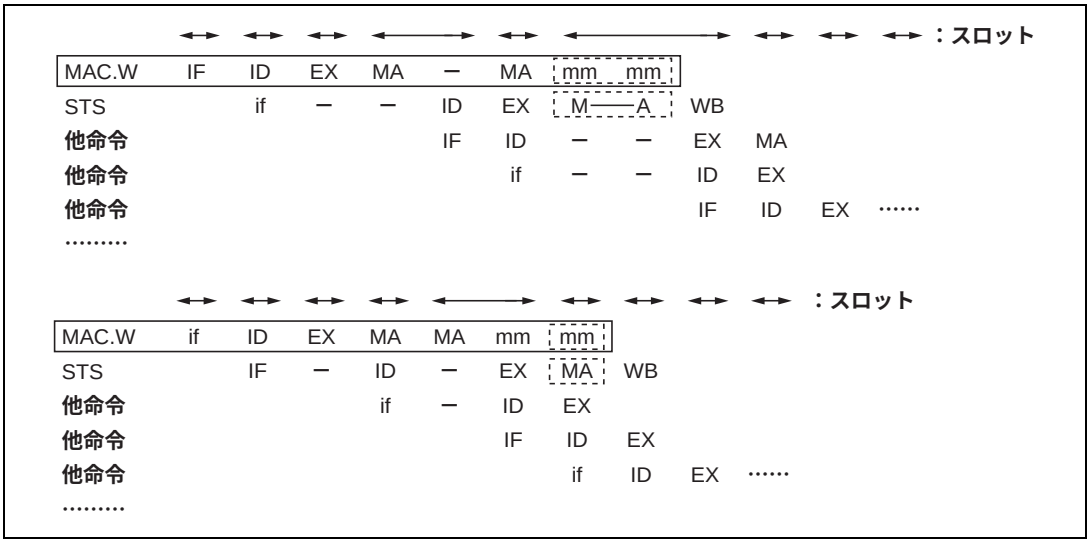
（c） MAC.W命令の直後に、MULS.W命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがあります。MAC.W命令の乗算器の動作中（mm）にMULS.WのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。MAC.WとMULS.Wの間に乗算器と無関係な命令が1つ以上はいるとMAC.WとMULS.Wの競合によるストールはなくなります。MAC.W命令の乗算器の動作中（mm）にMULS.WのMAは競合しません。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。



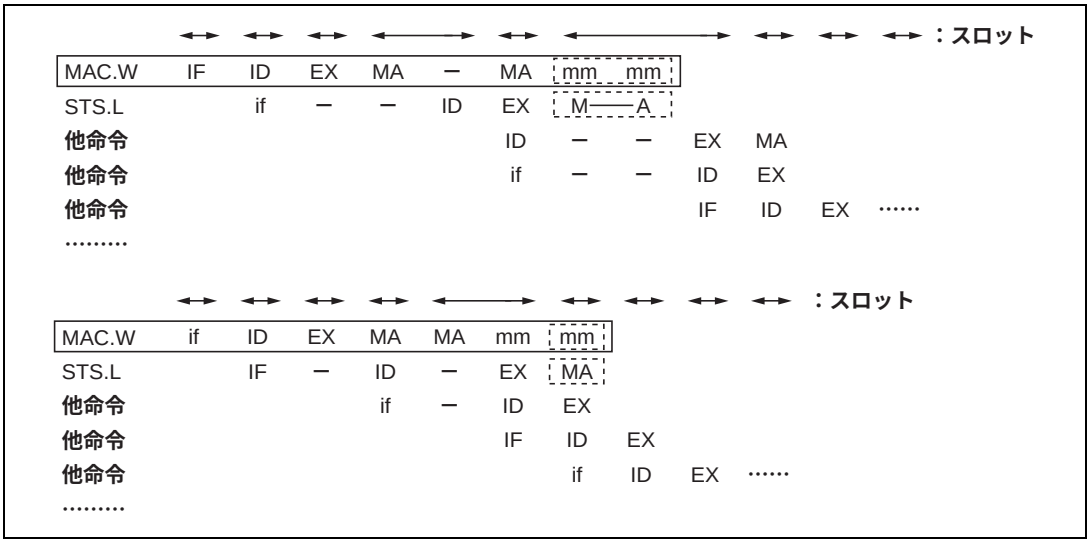
（d） MAC.W命令の直後に、DMULS.L命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがありますが、MAC.W命令の乗算器の動作中（mm）にMULS.WのMAは競合しません。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。



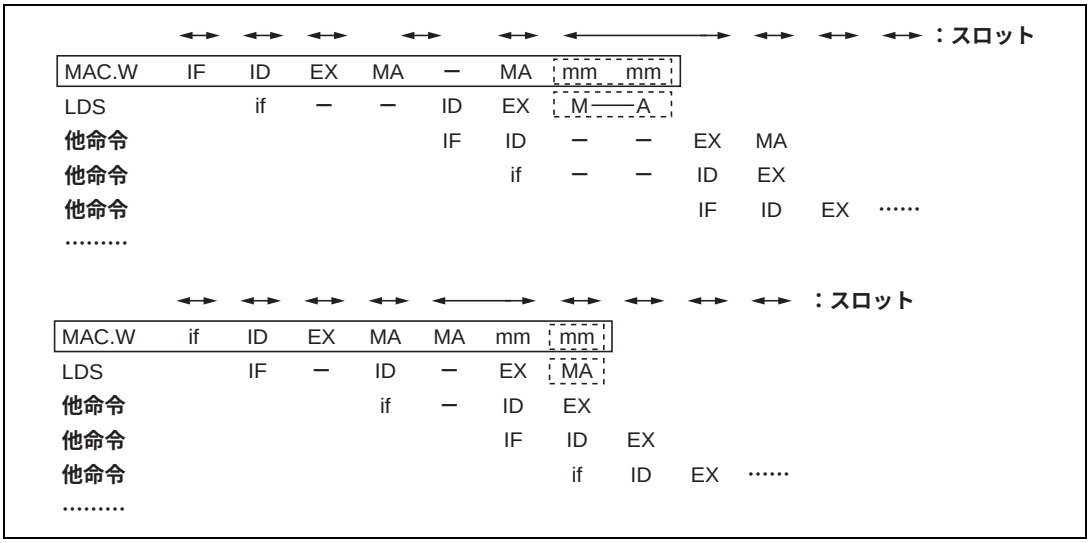
(e) MAC.W命令の直後に、STS（レジスタ）命令がくる場合
STS命令で、MACレジスタの内容を汎用レジスタにストアする場合、後述のとおり、STS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



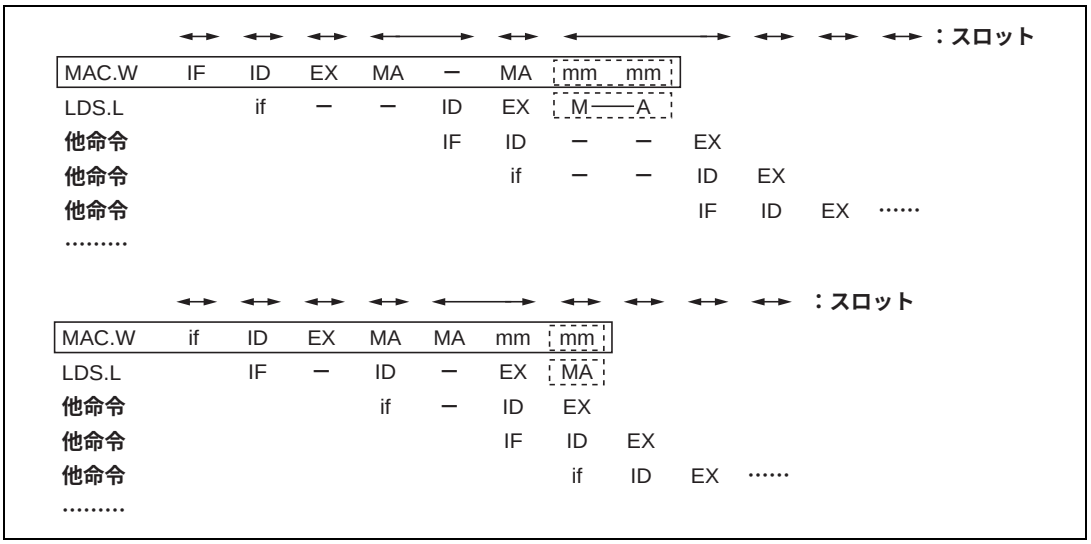
(f) MAC.W命令の直後に、STS.L（メモリ）命令がくる場合
STS命令で、MACレジスタの内容をメモリにストアする場合、後述のとおり、STS命令には乗算器のアクセスと、メモリライトのためのMAステージがはいります。これらの図については、MAとIFの競合を考慮して書いてあります。



（g） MAC.W命令の直後に、LDS（レジスタ）命令がくる場合
LDS命令で、MACレジスタの内容を汎用レジスタからロードする場合、後述のとおり、LDS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM－－－ A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



（h） MAC.W命令の直後に、LDS.L（メモリ）命令がくる場合
LDS命令で、MACレジスタの内容をメモリからロードする場合、後述のとおり、LDS命令にはメモリのアクセスと乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM－－－ A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(4) **倍精度積和命令（SH-2、SH-DSP）**

●命令の種類

MAC.L @Rm+,@Rn+

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	MA	mm	mm	mm	mm
次命令		IF	ー	ID	EX	MA	WB		
次々命令				IF	ID	EX	MA	WB	
……									

●動作説明

パイプラインは、IF、ID、EX、MA、MA、mm、mm、mm、mm の 9 段で終了します。2 番目の MA は、メモリ読み込みと共に乗算器のアクセスも行います。mm は乗算器が動作している状態を表しています。mm は最後の MA 終了後、スロットに関係なく 4 ステートの間、動作します。また、MAC.L 命令の次命令の ID は 1 スロット分後ろにストールされます。MAC.L 命令の 2 個の MA も、IF と競合する場合は、**「7.2.1 命令フェッチ（IF）とメモリアクセス（MA）の競合」**で説明したようにスロットがスプリットします。

MAC.L 命令の後ろに乗算器を使わない命令がくる場合は、MAC.L 命令は IF、ID、EX、MA、MA の 5 段パイプライン命令と考えてかまいません。すなわち、この場合は、次命令の ID が 1 スロット分ストールするだけで、あとは、通常のパイプライン動作と同様になります。

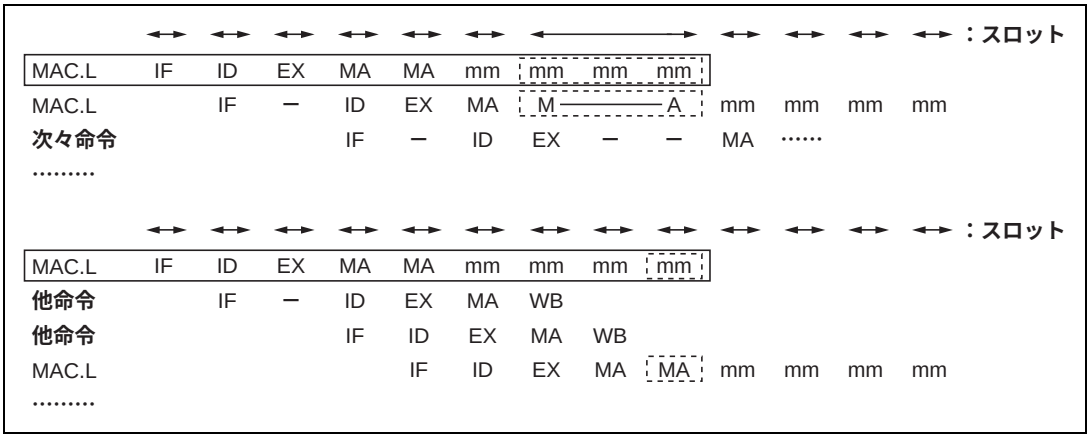
しかし、MAC.L 命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生しますので、通常の動作と異なってきます。この場合は、以下のケースが考えられます。

（a） MAC.L 命令の直後に、MAC.L 命令が連続してくる場合
（b） MAC.L 命令の直後に、MAC.W 命令が連続してくる場合
（c） MAC.L 命令の直後に、DMULS.L 命令が連続してくる場合
（d） MAC.L 命令の直後に、MULS.W 命令が連続してくる場合
（e） MAC.L 命令の直後に、STS（レジスタ）命令がくる場合
（f） MAC.L 命令の直後に、STS.L（メモリ）命令がくる場合
（g） MAC.L 命令の直後に、LDS（レジスタ）命令がくる場合
（h） MAC.L 命令の直後に、LDS.L（メモリ）命令がくる場合

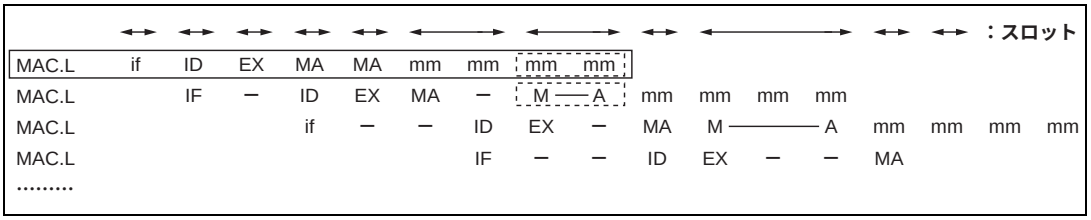
（a） MAC.L命令の直後に、MAC.L命令が連続してくる場合

MAC.L命令の2番目のMAが、その前の乗算系命令によって発生したmmと競合した場合、そのMAのバスサイクルはmmが終了するまで引き延ばされ（下図のM――A）、その引き延ばされたMAは1つのスロットになります。

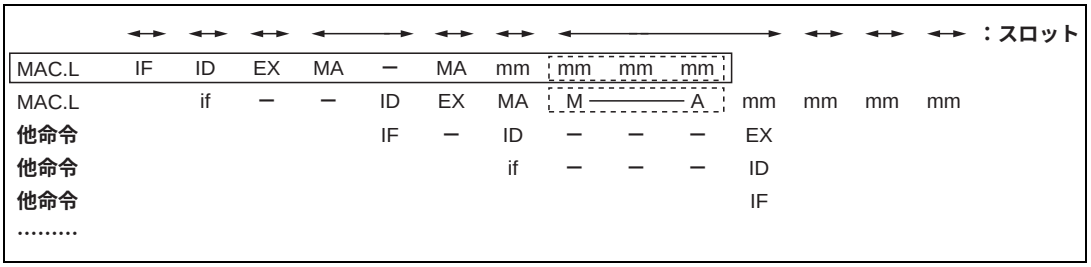
MAC.L命令とMAC.L命令の間に、乗算器と無関係な命令が2つ以上入ると、MAC.L命令どうしの乗算器の競合によるストールはなくなります。



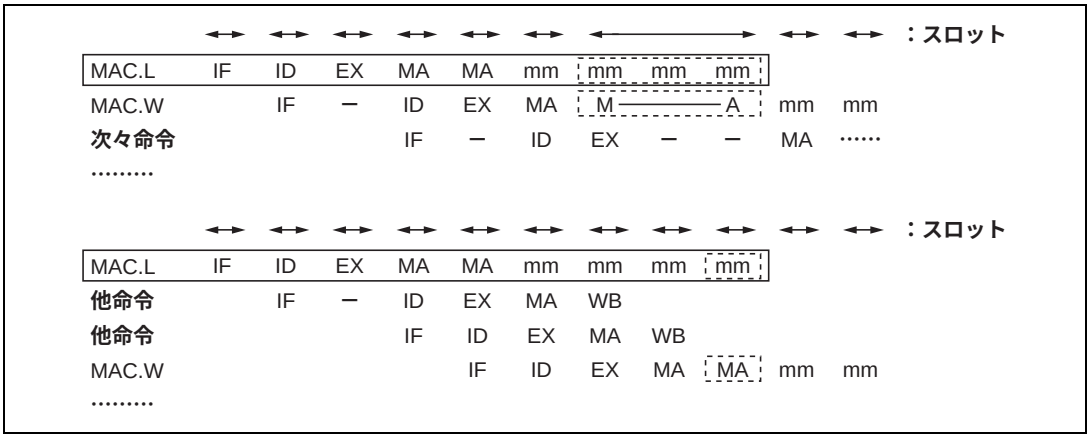
MAC.Lの連続により、MAとIFの競合で命令実行がずれても乗算器の競合が少なくなる場合があります。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



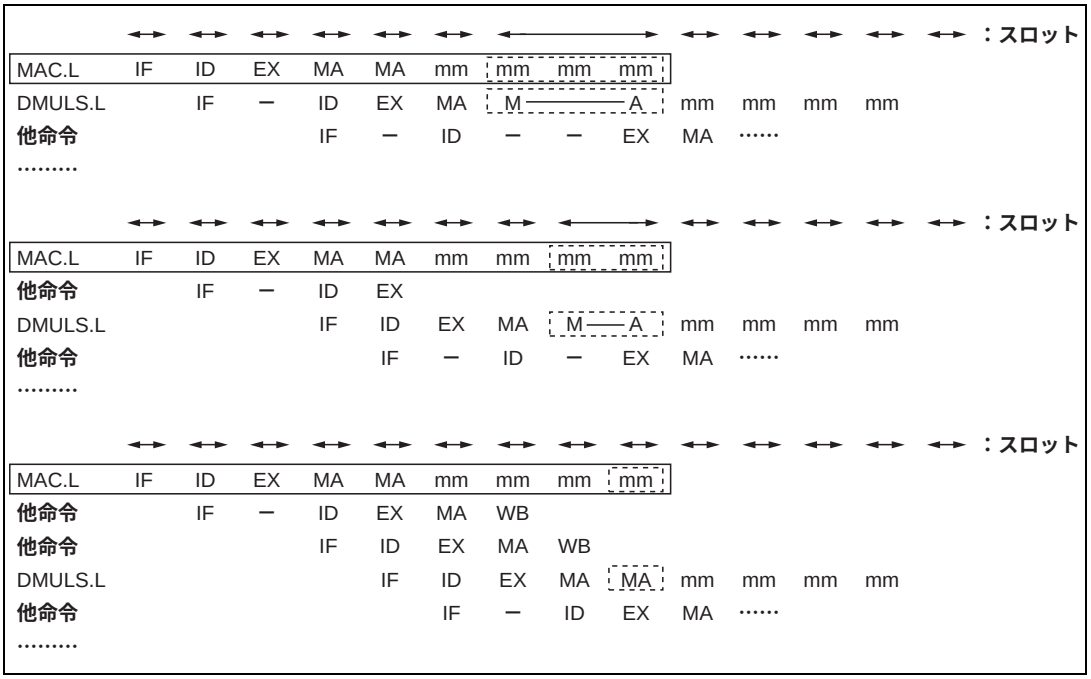
MAC.L命令の2番目のMAがmm終了まで引き延ばされている場合、このMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



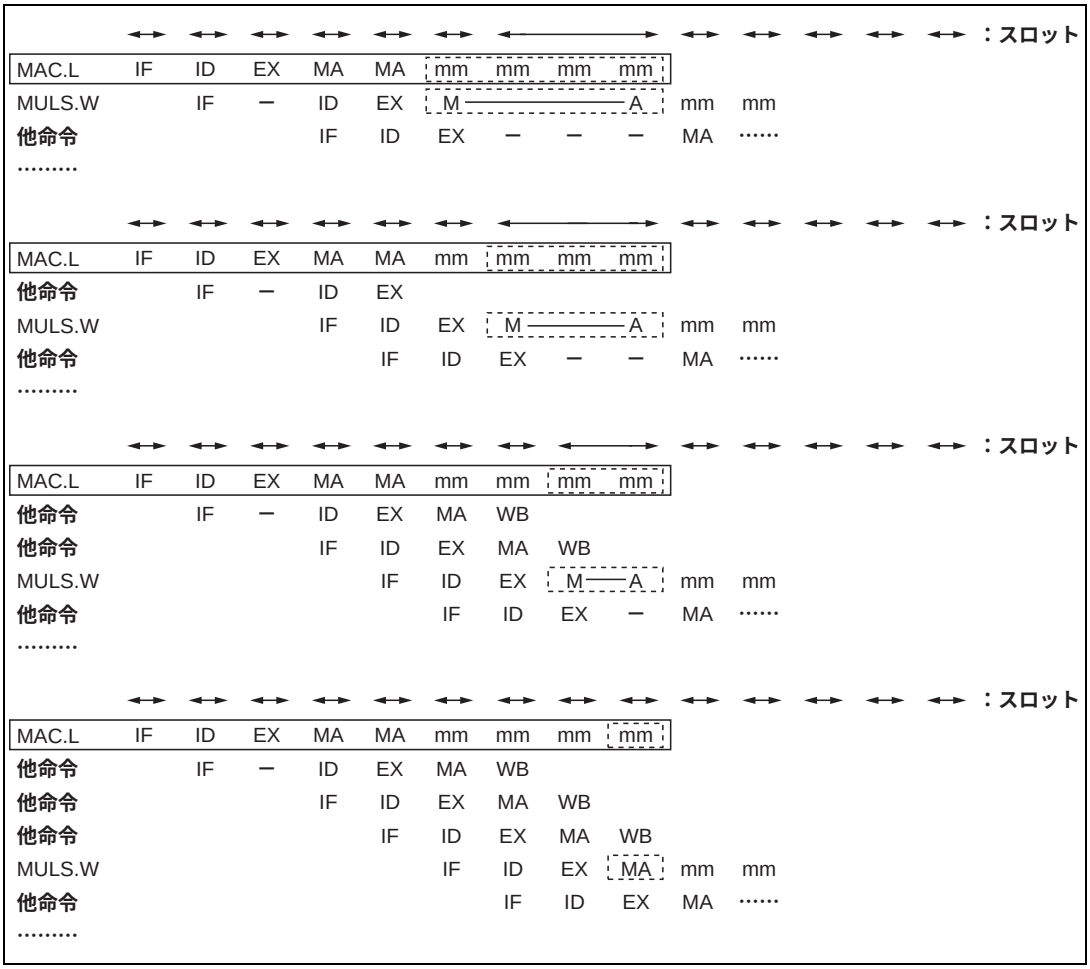
(b) MAC.L命令の直後に、MAC.W命令が連続してくる場合
MAC.W命令の2番目のMAが、その前の乗算系命令によって発生したmmと競合した場合、そのMAのバスサイクルはmmが終了するまで引き延ばされ（下図のM――A）、その引き延ばされたMAは1つのスロットになります。
MAC.L命令とMAC.W命令の間に、乗算器と無関係な命令が2つ以上入ると、MAC.L命令とMAC.W命令の乗算器の競合によるストールはなくなります。



（c） MAC.L命令の直後に、DMULS.L命令が連続してくる場合
DMULS.L命令には、乗算器アクセスのためのMAステージがあります。MAC.L命令の乗算器の動作中（mm）にDMULS.L命令の2番目のMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。MAC.LとDMULS.Lの間に乗算器と無関係な命令が2つ以上入るとMAC.LとDMULS.Lの競合によるストールはなくなります。なお、DMULS.LのMAもIFと競合するとスロットがスプリットします。

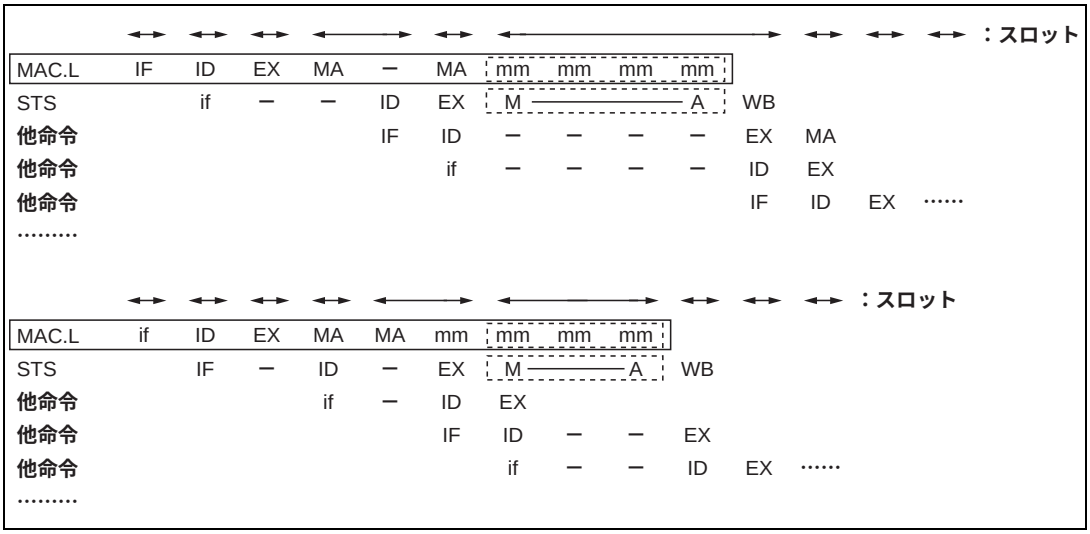


（d） MAC.L命令の直後に、MULS.W命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがあります。MAC.L命令の乗算器の動作中（mm）にMULS.WのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。MAC.LとMULS.Wの間に乗算器と無関係な命令が3つ以上入るとMAC.LとMULS.Wの競合によるストールはなくなります。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。



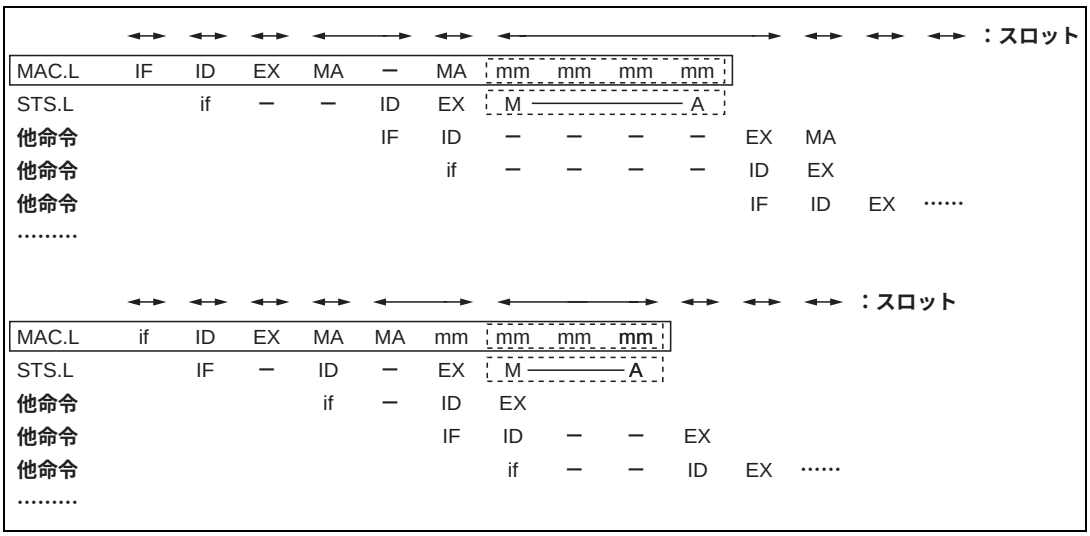
(e) MAC.L命令の直後に、STS（レジスタ）命令がくる場合

STS命令で、MACレジスタの内容を汎用レジスタにストアする場合、後述のとおり、STS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



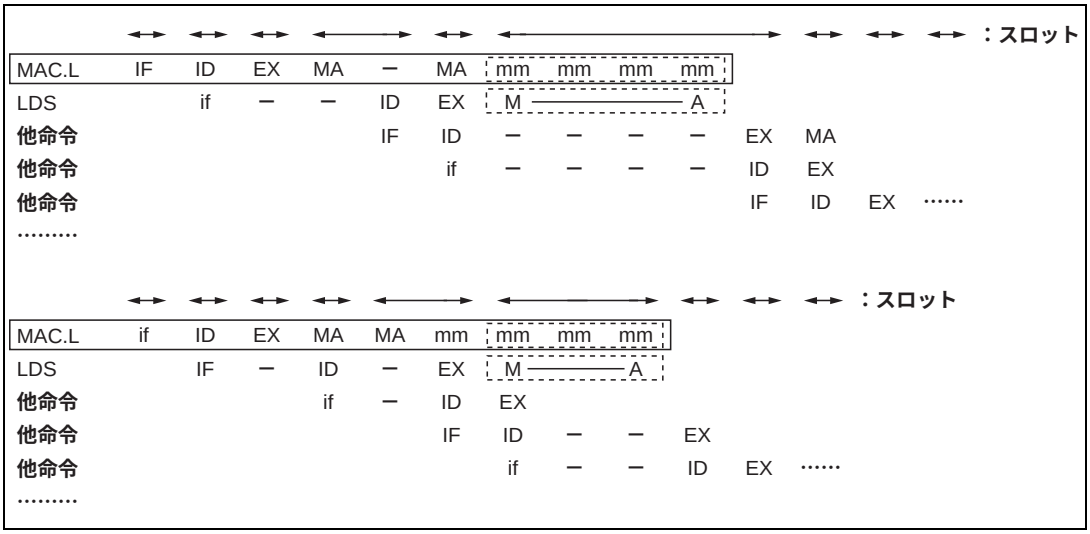
(f) MAC.L命令の直後に、STS.L（メモリ）命令がくる場合

STS命令で、MACレジスタの内容をメモリにストアする場合、後述のとおり、STS命令には乗算器のアクセスと、メモリライトのためのMAステージが入ります。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



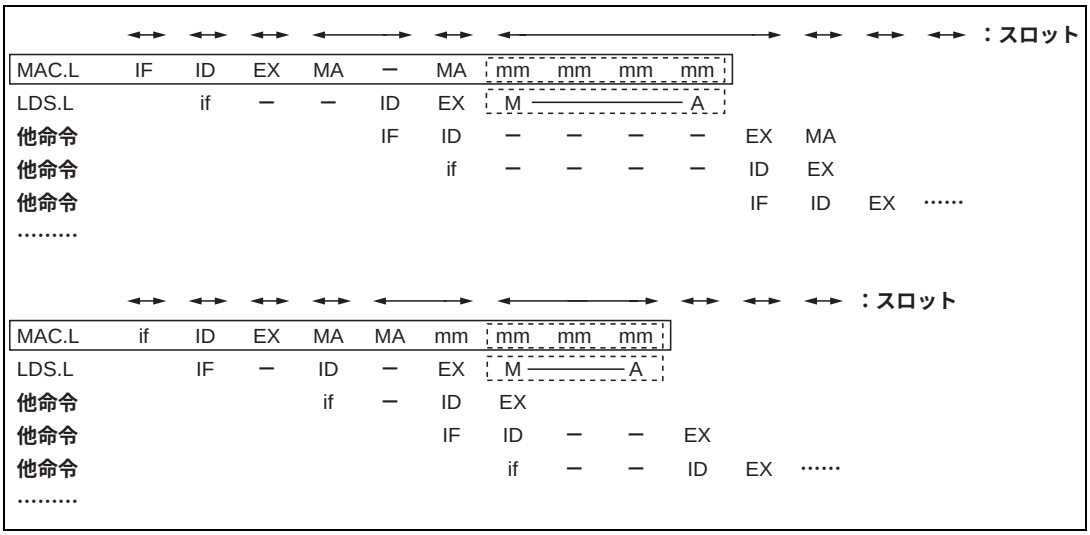
（g） MAC.L命令の直後に、LDS（レジスタ）命令がくる場合

LDS命令で、MACレジスタの内容を汎用レジスタからロードする場合、後述のとおり、LDS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



（h） MAC.L命令の直後に、LDS.L（メモリ）命令がくる場合

LDS命令で、MACレジスタの内容をメモリからロードする場合、後述のとおり、LDS命令にはメモリのアクセスと乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。

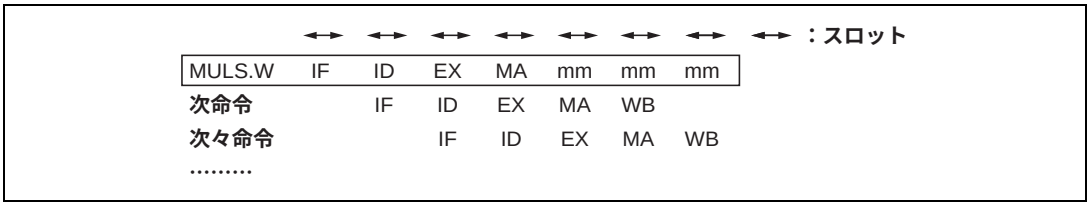


(5) 乗算命令（SH-1）

【命令の種類】

MULS.W Rm,Rn
MULU.W Rm,Rn

【パイプライン】



【動作説明】

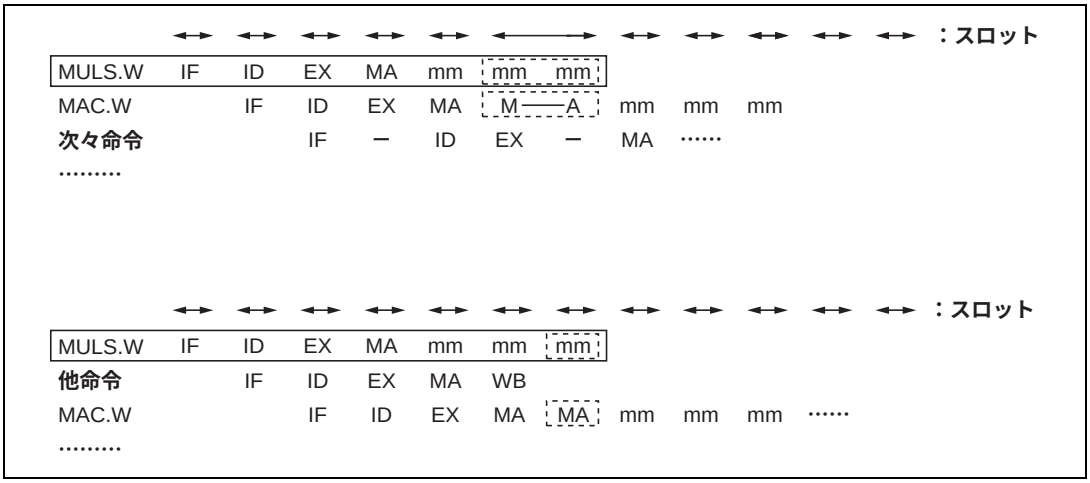
パイプラインは、IF、ID、EX、MA、mm、mm、mm の 7 段で終了します。MA は、乗算器のアクセスを行います。mm は乗算器が動作している状態を表しています。mm は MA 終了後、スロットに関係なく 3 ステートの間、動作します。MULS.W 命令の MA も、IF と競合する場合は**「7.4 命令フェッチ（IF）とメモリアクセス（MA）の競合」**で説明したようにスロットがスプリットします。

MULS.W 命令の後ろに乗算器を使わない命令がくる場合は、MULS.W 命令は IF、ID、EX、MA の 4 段パイプライン命令とみなして考えてかまいません。すなわち、この場合は、通常のパイプライン動作と同様になります。

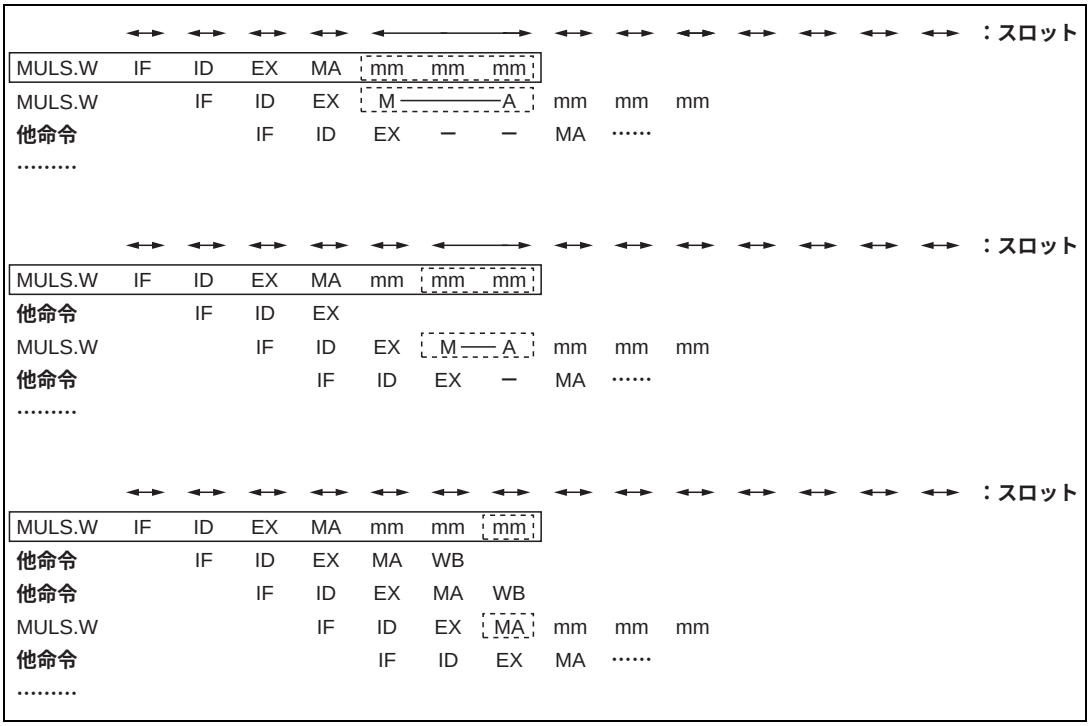
しかし、MULS.W 命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生しますので、通常の動作と異なってきます。この場合は、以下のケースが考えられます。

（a） MULS.W 命令の直後に、MAC.W 命令が連続してくる場合
（b） MULS.W 命令の直後に、MULS.W 命令が連続してくる場合
（c） MULS.W 命令の直後に、STS（レジスタ）命令がくる場合
（d） MULS.W 命令の直後に、STS.L（メモリ）命令がくる場合
（e） MULS.W 命令の直後に、LDS（レジスタ）命令がくる場合
（f） MULS.W 命令の直後に、LDS.L（メモリ）命令がくる場合

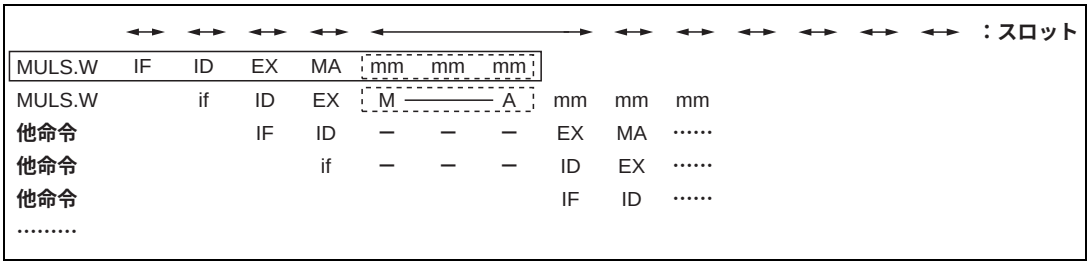
(a) MULS.W命令の直後に、MAC.W命令が連続してくる場合
MAC.W命令の2番目のMAが、その前の乗算系命令によって発生したmmと競合した場合、そのMAのバスサイクルはmmが終了するまで引き延ばされ（下図のM――A）、その引き延ばされたMAは1つのスロットになります。
MULS.W命令とMAC.W命令の間に、乗算器と無関係な命令が1つ以上入ると、MULS.WとMAC.W命令どうしの乗算器の競合によるストールはなくなります。



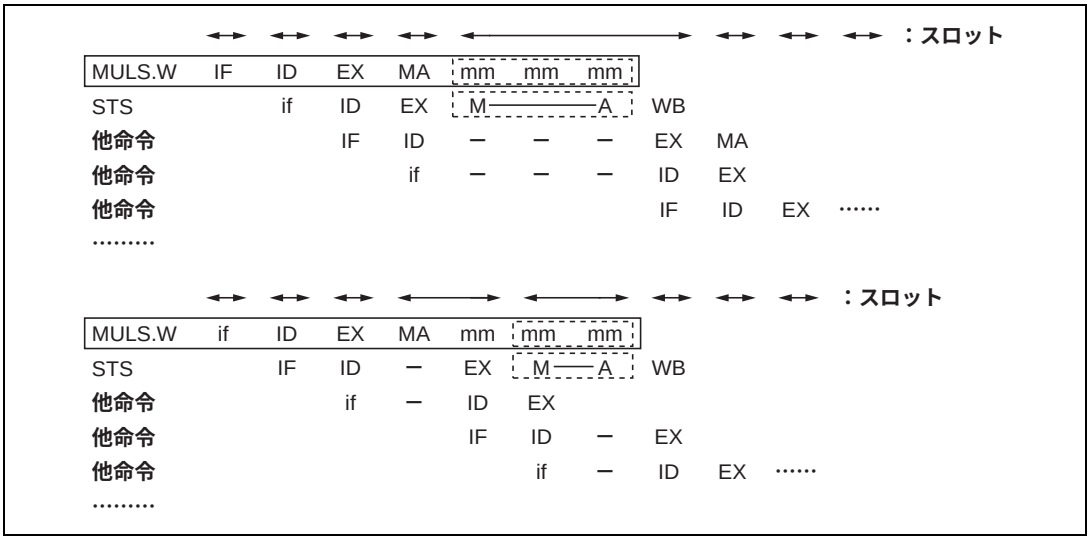
（b） MULS.W命令の直後に、MULS.W命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがあります。MULS.W命令の乗算器の動作中（mm）に他のMULS.WのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。MULS.WとMULS.Wの間に乗算器と無関係な命令が2つ以上入るとMULS.WとMULS.Wの競合によるストールはなくなります。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。



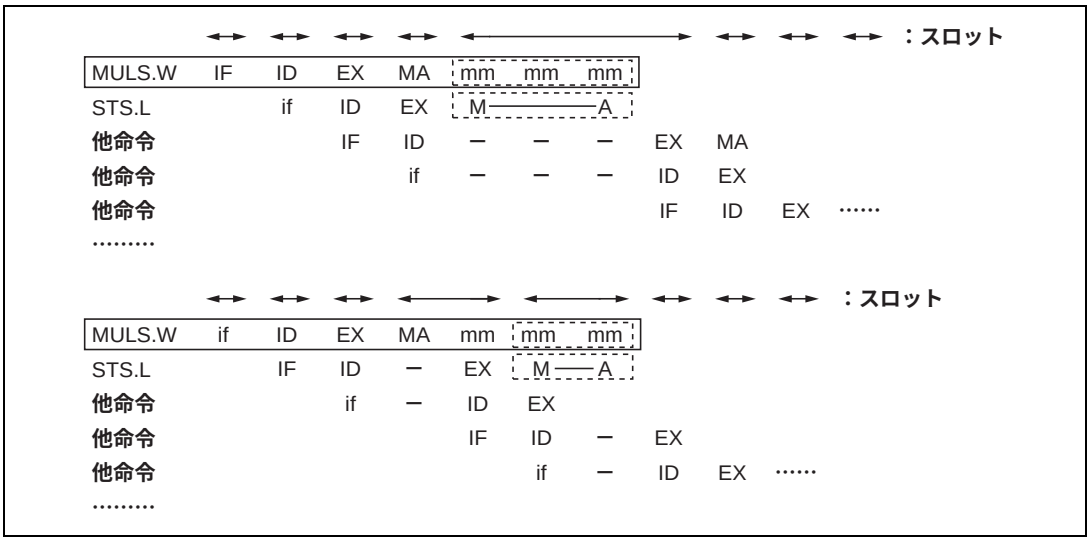
MULS.W命令のMAがmm終了まで引き延ばされている場合、このMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



(c) MULS.W命令の直後に、STS（レジスタ）命令がくる場合
STS命令で、MACレジスタの内容を汎用レジスタにストアする場合、後述のとおり、STS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。

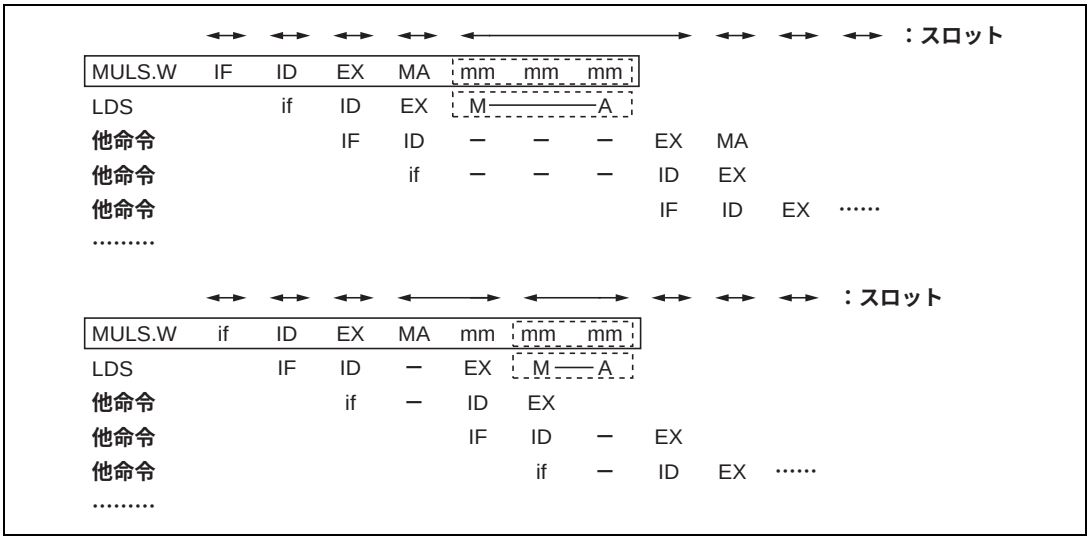


（d） MULS.W命令の直後に、STS.L（メモリ）命令がくる場合
STS命令で、MACレジスタの内容をメモリにストアする場合、後述のとおり、STS命令には乗算器のアクセスと、メモリライトのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了してから1ステート経過するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



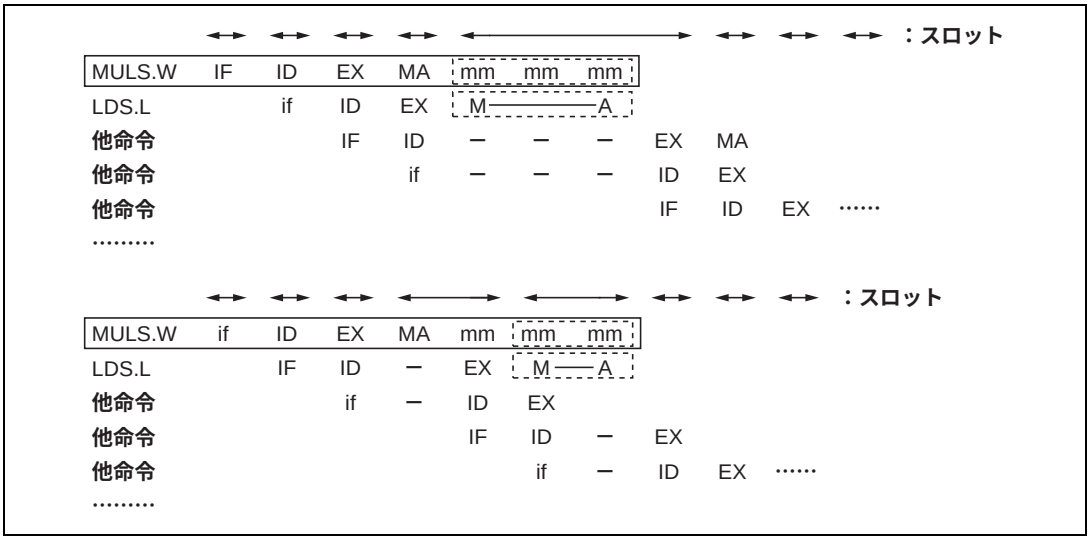
(e) MULS.W命令の直後に、LDS（レジスタ）命令がくる場合

LDS命令で、MACレジスタの内容を汎用レジスタからロードする場合、後述のとおり、LDS命令には乗算器のアクセスのためのMAステージがはいります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(f) MULS.W命令の直後に、LDS.L（メモリ）命令がくる場合

LDS命令で、MACレジスタの内容をメモリからロードする場合、後述のとおり、LDS命令にはメモリアクセスと乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(6) 乗算命令（SH-2、SH-DSP）

●命令の種類

```
MULS.W    Rm,Rn
MULU.W    Rm,Rn
```

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	mm	mm		
次命令		IF	ID	EX	MA	WB		
次々命令			IF	ID	EX	MA	WB	
……								

●動作説明

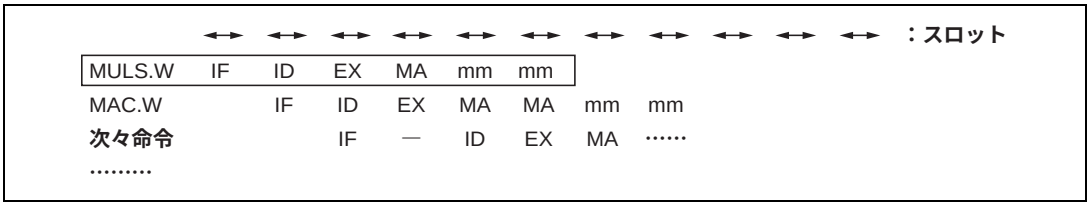
パイプラインは、IF、ID、EX、MA、mm、mm の 6 段で終了します。MA は、乗算器のアクセスを行います。mm は乗算器が動作している状態を表しています。mm は MA 終了後、スロットに関係なく 2 ステートの間、動作します。MULS.W 命令の MA も、IF と競合する場合は**「7.2.1 命令フェッチ（IF）とメモリアクセス（MA）の競合」**で説明したようにスロットがスプリットします。

MULS.W 命令の後ろに乗算器を使わない命令がくる場合は、MULS.W 命令は IF、ID、EX、MA の 4 段パイプライン命令と考えてかまいません。すなわち、この場合は、通常のパイプライン動作と同様になります。

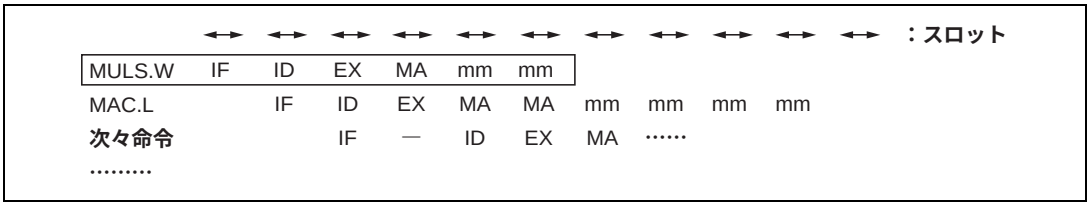
しかし、MULS.W 命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生しますので、通常の動作と異なってきます。この場合は、以下のケースが考えられます。

（a） MULS.W 命令の直後に、MAC.W 命令が連続してくる場合
（b） MULS.W 命令の直後に、MAC.L 命令が連続してくる場合
（c） MULS.W 命令の直後に、MULS.W 命令が連続してくる場合
（d） MULS.W 命令の直後に、DMULS.L 命令が連続してくる場合
（e） MULS.W 命令の直後に、STS（レジスタ）命令がくる場合
（f） MULS.W 命令の直後に、STS.L（メモリ）命令がくる場合
（g） MULS.W 命令の直後に、LDS（レジスタ）命令がくる場合
（h） MULS.W 命令の直後に、LDS.L（メモリ）命令がくる場合

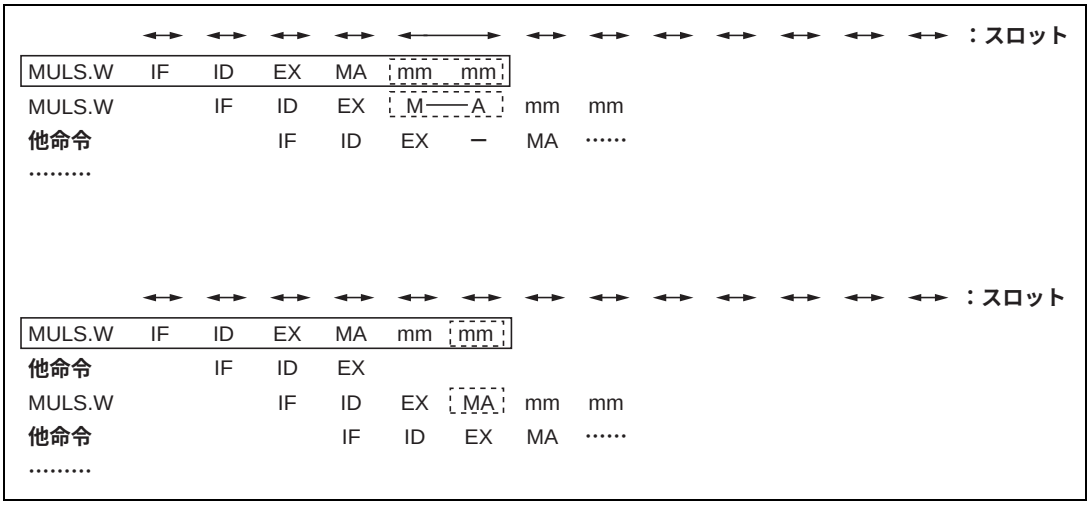
(a) MULS.W命令の直後に、MAC.W命令が連続してくる場合
MAC.W命令の2番目のMAと、その前の乗算系命令によって発生したmmとは競合しません。



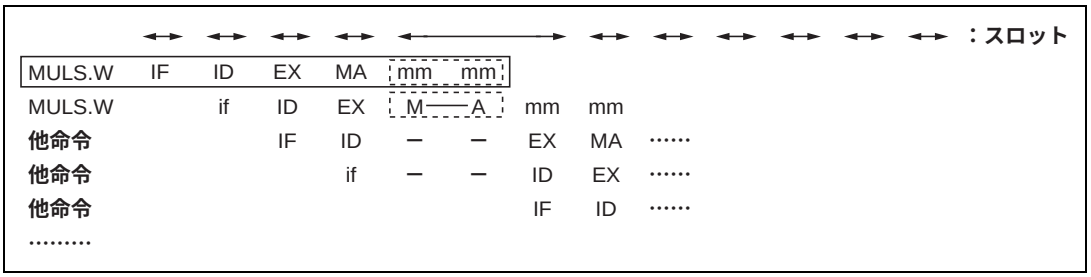
(b) MULS.W命令の直後に、MAC.L命令が連続してくる場合
MAC.W命令の2番目のMAと、その前の乗算系命令によって発生したmmとは競合しません。



(c) MULS.W命令の直後に、MULS.W命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがあります。MULS.W命令の乗算器の動作中（mm）に他のMULS.WのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。MULS.WとMULS.Wの間に乗算器と無関係な命令が1つ以上入るとMULS.WとMULS.Wの競合によるストールはなくなります。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。

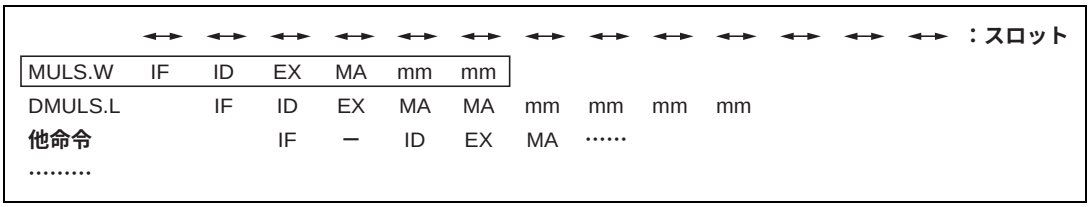


MULS.W命令のMAがmm終了まで引き延ばされている場合、このMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



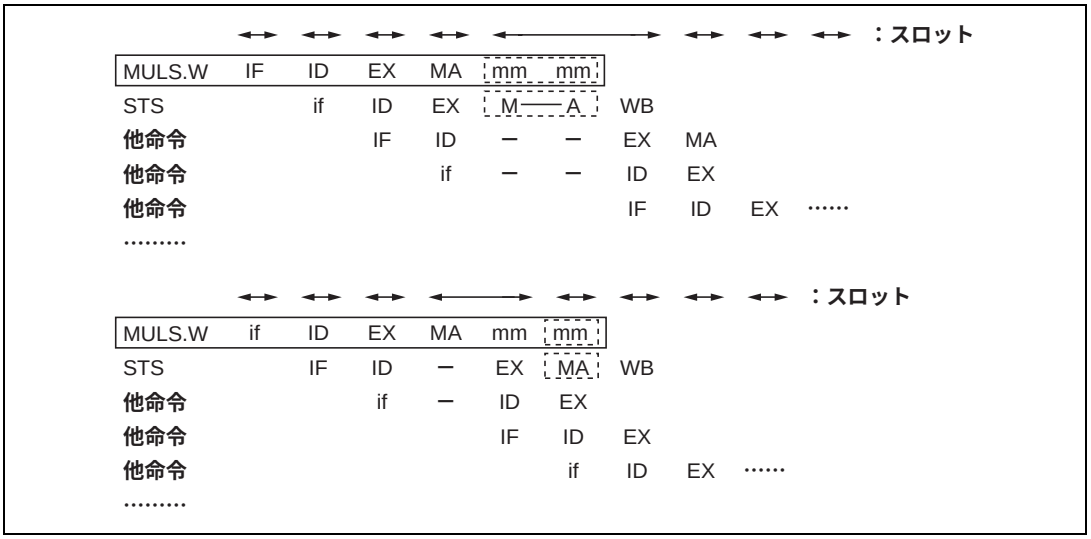
（d） MULS.W命令の直後に、DMULS.L命令が連続してくる場合

DMULS.L命令の2番目のMAは乗算器アクセスを行いますがMULS.W命令によって発生したmmとは競合しません。

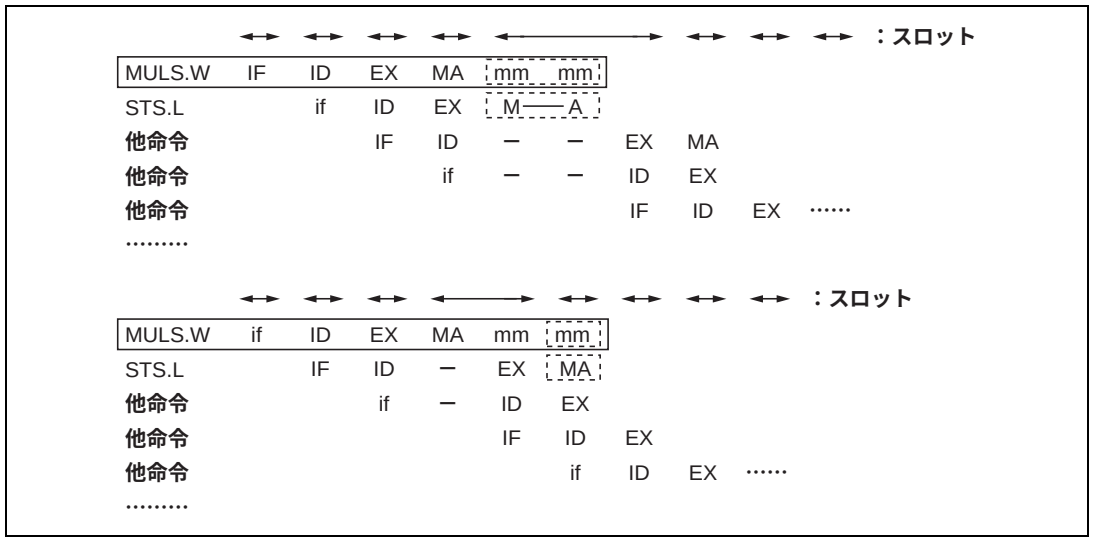


（e） MULS.W命令の直後に、STS（レジスタ）命令がくる場合

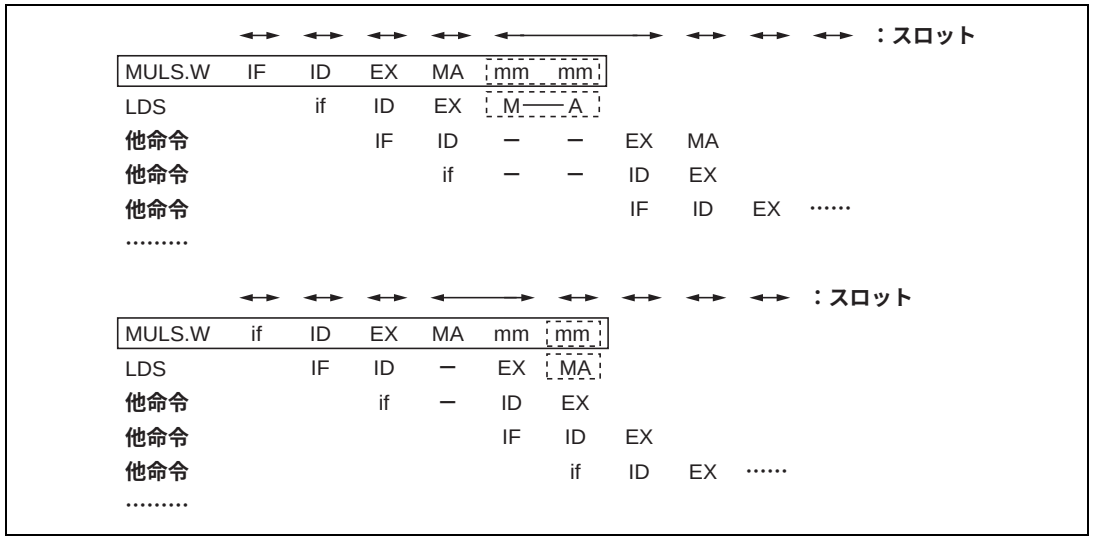
STS命令で、MACレジスタの内容を汎用レジスタにストアする場合、後述のとおり、STS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM——A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



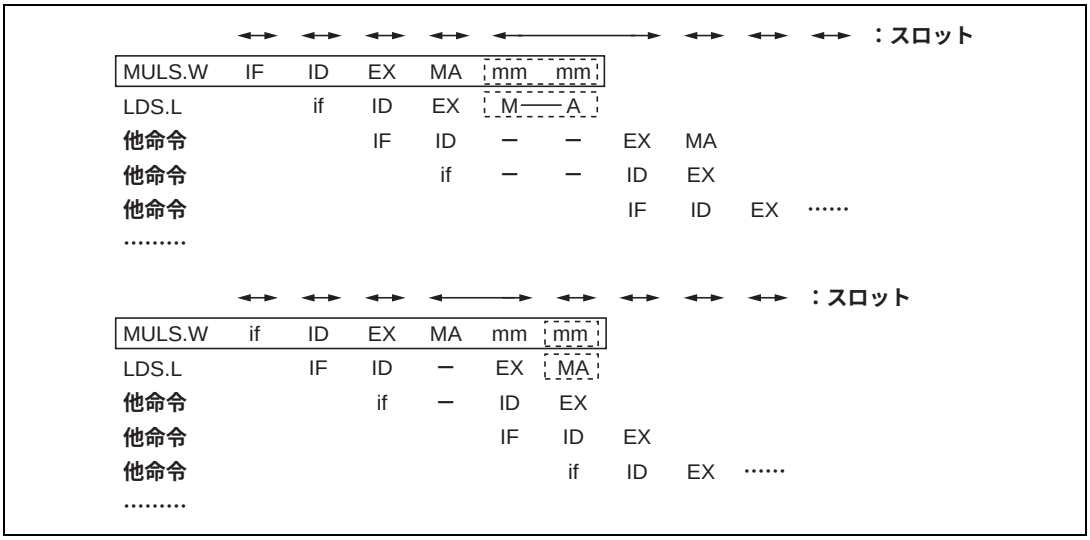
(f) MULS.W命令の直後に、STS.L（メモリ）命令がくる場合
STS命令で、MACレジスタの内容をメモリにストアする場合、後述のとおり、STS命令には乗算器のアクセスと、メモリライトのためのMAステージが入ります。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(g) MULS.W命令直後に、LDS（レジスタ）命令が来る場合
LDS命令で、MACレジスタの内容を汎用レジスタからロードする場合、後述のとおり、LDS命令には乗算器のアクセスするためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM——A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



（h） MULS.W命令の直後に、LDS.L（メモリ）命令が来る場合
LDS命令で、MACレジスタの内容をメモリからロードする場合、後述のとおり、LDS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



(7) 倍精度の乗算命令（SH-2、SH-DSP）

●命令の種類

```
DMULS.L   Rm,Rn
DMULU.L   Rm,Rn
MUL.L     Rm,Rn
```

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	MA	mm	mm	mm	mm
次命令		IF	ー	ID	EX	MA	WB		
次々命令				IF	ID	EX	MA	WB	
……									

●動作説明

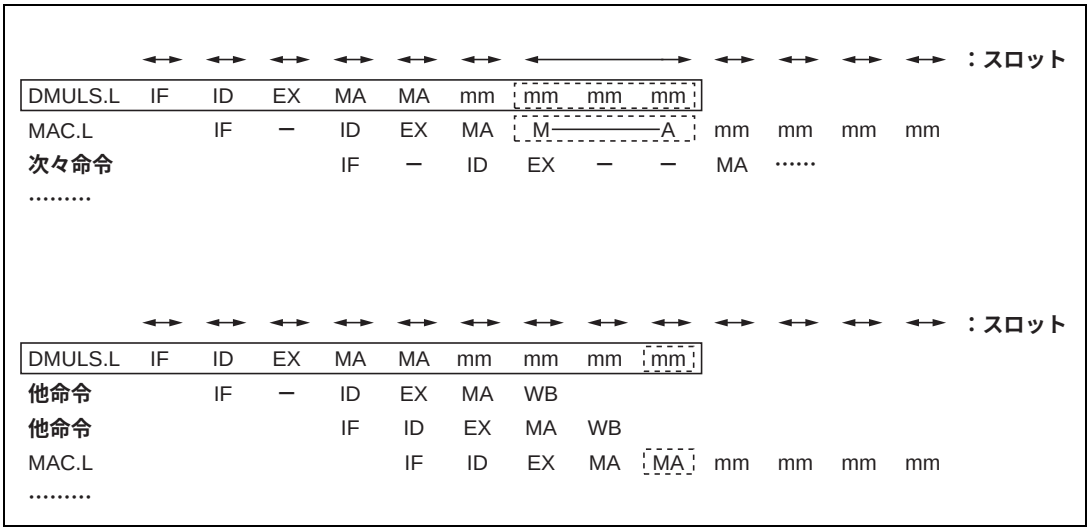
パイプラインは、IF、ID、EX、MA、MA、mm、mm、mm、mm の 9 段で終了します。2 番目の MA は、乗算器のアクセスを行います。mm は乗算器が動作している状態を表しています。mm は MA 終了後、スロットに関係なく 4 ステートの間、動作します。また、DMULS.L 命令の次命令の ID は 1 スロット分後ろにストールされます（積和命令参照）。DMULS.L 命令の 2 個の MA も、IF と競合する場合は**「7.2.1 命令フェッチ（IF）とメモリアクセス（MA）の競合」**で説明したようにスロットがスプリットします。

DMULS.L 命令の後ろに乗算器を使わない命令がくる場合は、DMULS.L 命令は IF、ID、EX、MA、MA の 5 段パイプライン命令と考えてかまいません。すなわち、この場合は、通常のパイプライン動作と同様になります。

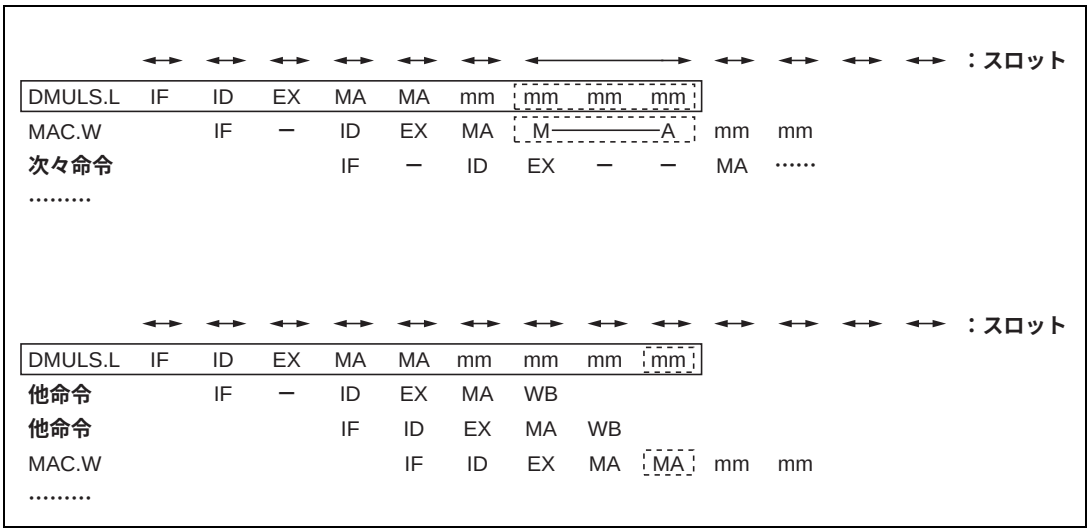
しかし、DMULS.L 命令の後ろに乗算器を使う命令がくる場合、乗算器の競合が発生しますので、通常の動作と異なってきます。この場合は、以下のケースが考えられます。

（a） DMULS.L 命令の直後に、MAC.L 命令が連続してくる場合
（b） DMULS.L 命令の直後に、MAC.W 命令が連続してくる場合
（c） DMULS.L 命令の直後に、DMULS.L 命令が連続してくる場合
（d） DMULS.L 命令の直後に、MULS.W 命令が連続してくる場合
（e） DMULS.L 命令の直後に、STS（レジスタ）命令がくる場合
（f） DMULS.L 命令の直後に、STS.L（メモリ）命令がくる場合
（g） DMULS.L 命令の直後に、LDS（レジスタ）命令がくる場合
（h） DMULS.L 命令の直後に、LDS.L（メモリ）命令がくる場合

（a） DMULS.L命令の直後に、MAC.L命令が連続してくる場合
MAC.L命令の2番目のMAが、その前の乗算系命令によって発生したmmと競合した場合、そのMAのバスサイクルはmmが終了するまで引き延ばされ（下図のM――A）、その引き延ばされたMAは1つのスロットになります。
DMULS.L命令とMAC.L命令の間に、乗算器と無関係な命令が2つ以上入ると、DMULS.LとMAC.L命令どうしの乗算器の競合によるストールはなくなります。

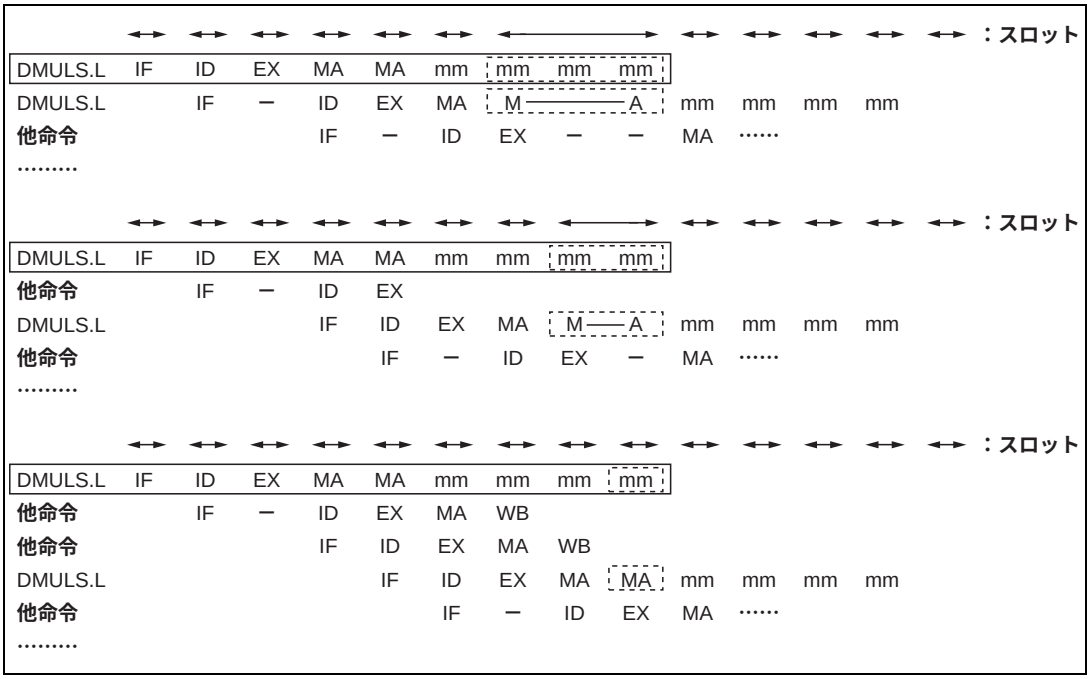


（b） DMULS.L命令の直後に、MAC.W命令が連続してくる場合
MAC.W命令の2番目のMAが、その前の乗算系命令によって発生したmmと競合した場合、そのMAのバスサイクルはmmが終了するまで引き延ばされ（下図のM――A）、その引き延ばされたMAは1つのスロットになります。
DMULS.L命令とMAC.W命令の間に、乗算器と無関係な命令が2つ以上入ると、DMULS.LとMAC.W命令どうしの乗算器の競合によるストールはなくなります。

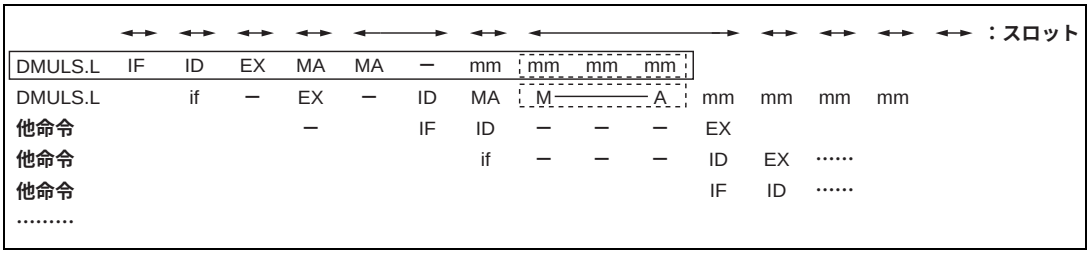


（c） DMULS.L命令の直後に、DMULS.L命令が連続してくる場合

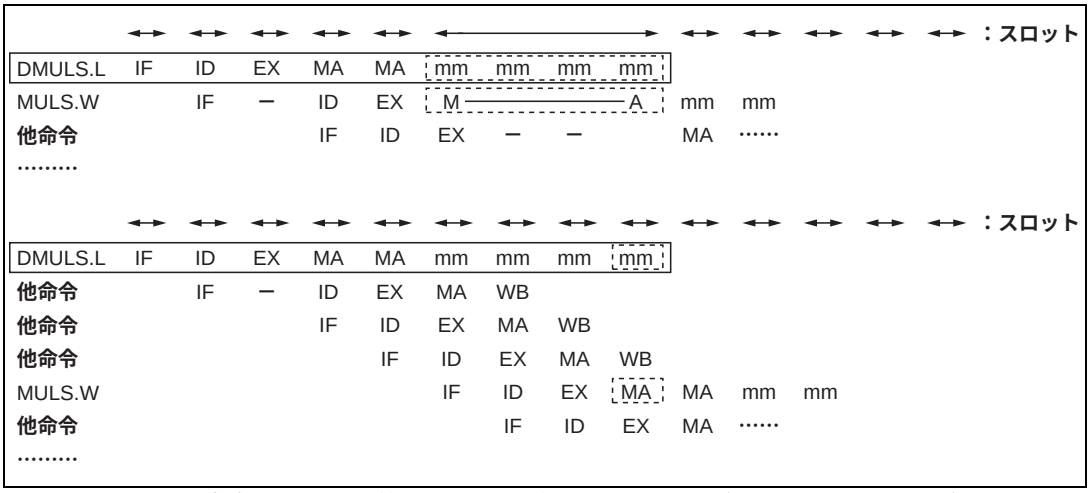
DMULS.L命令には、乗算器アクセスのためのMAステージがあります。DMULS.L命令の乗算器の動作中（mm）に他のDMULS.LのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――Aの）、1つのスロットを形成します。DMULS.LとDMULS.Lの間に乗算器と無関係な命令が2つ以上入るとDMULS.LとDMULS.Lの競合によるストールはなくなります。なお、DMULS.LのMAもIFと競合するとスロットがスプリットします。



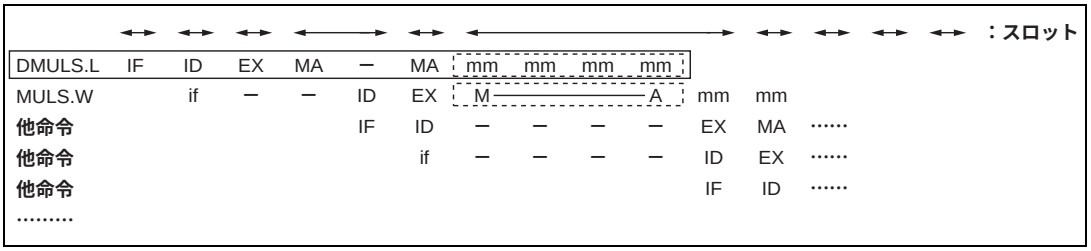
DMULS.L命令のMAがmm終了まで引き延ばされている場合、このMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



（d） DMULS.L命令の直後に、MULS.W命令が連続してくる場合
MULS.W命令には、乗算器アクセスのためのMAステージがあります。DMULS.L命令の乗算器の動作中（mm）に他のMULS.WのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――Aの）、1つのスロットを形成します。DMULS.LとMULS.Wの間に乗算器と無関係な命令が3つ以上入るとDMULS.LとDMULS.Lの競合によるストールはなくなります。なお、MULS.WのMAもIFと競合するとスロットがスプリットします。

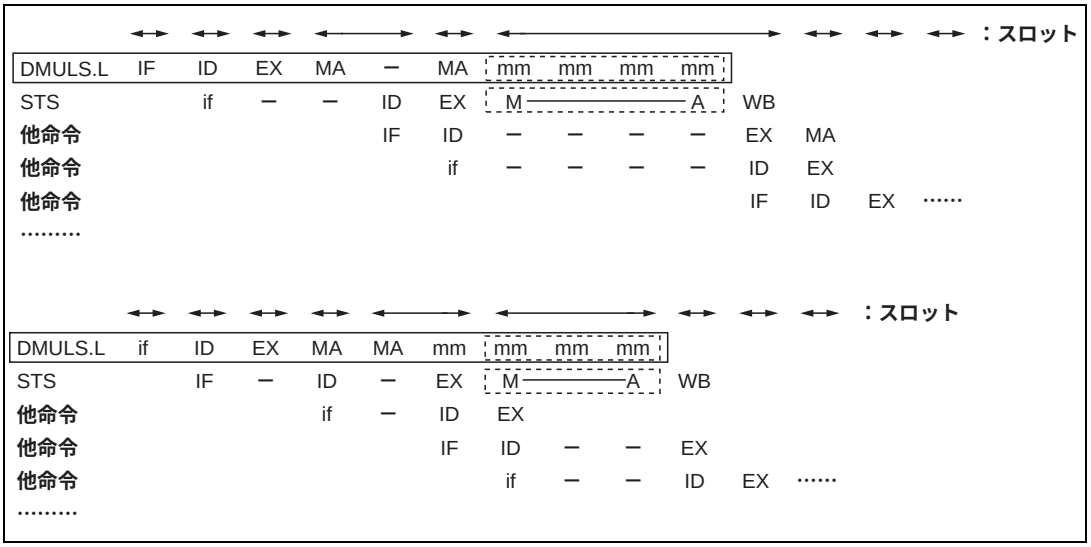


DMULS.L命令のMAがmm終了まで引き延ばされている場合、このMAとIFが競合すると、通常どおりスロットがスプリットします。下図を参照してください。この図については、MAとIFの競合を考慮して書いてあります。



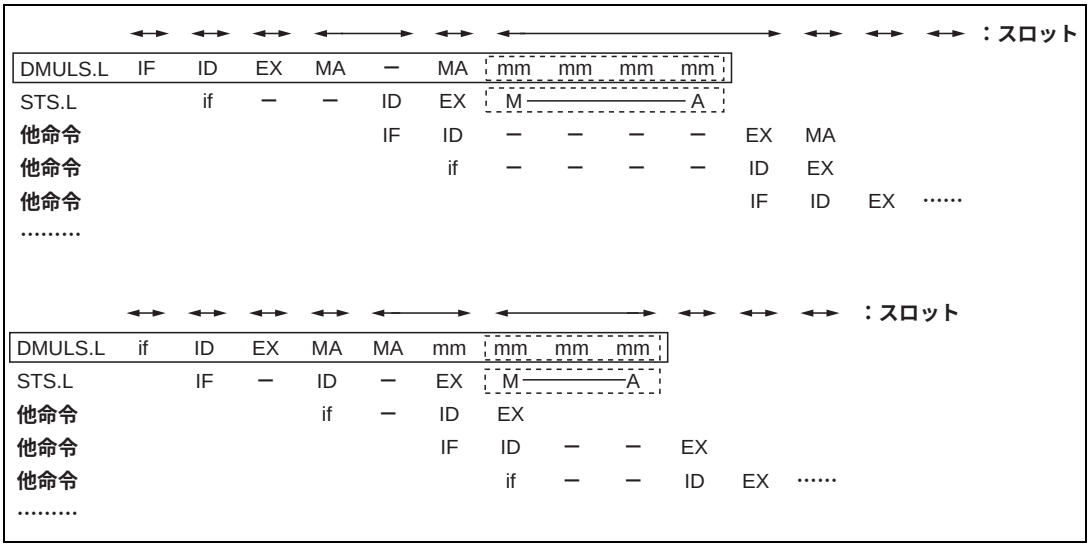
(e) DMULS.L命令の直後に、STS（レジスタ）命令がくる場合

STS命令で、MACレジスタの内容を汎用レジスタにストアする場合、後述のとおり、STS命令には乗算器のアクセスのためのMAステージがはいります。乗算器の動作中（mm）にSTSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



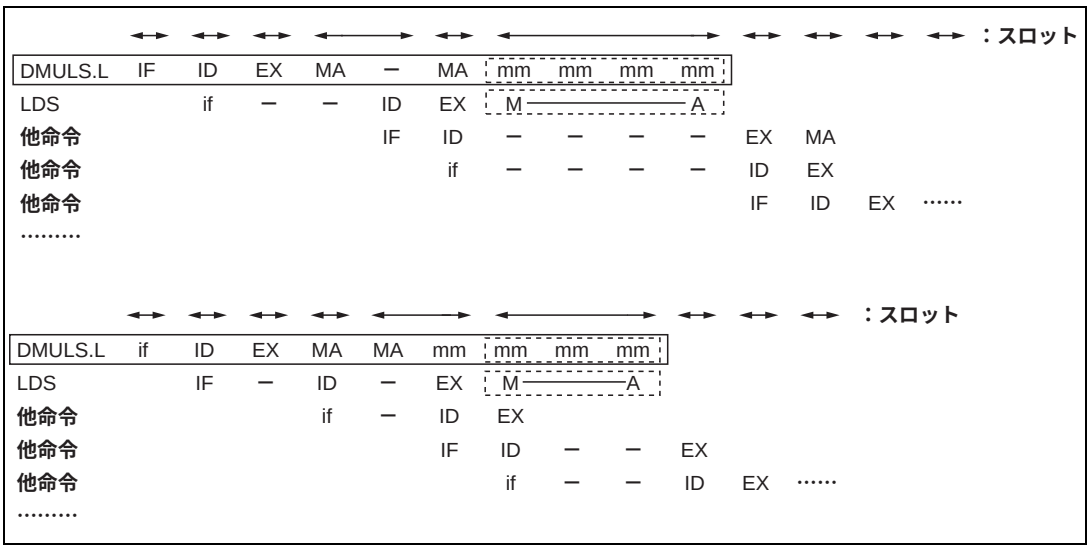
(f) DMULS.L命令の直後に、STS.L（メモリ）命令がくる場合

STS命令で、MACレジスタの内容をメモリにストアする場合、後述のとおり、STS命令には乗算器のアクセスと、メモリライトのためのMAステージが入ります。また、このSTSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



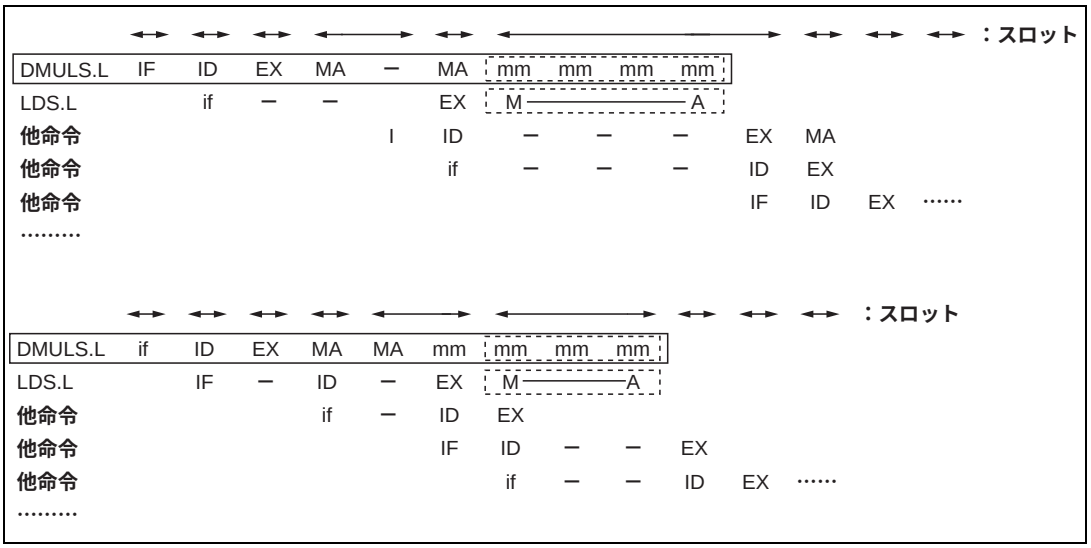
（g） DMULS.L命令の直後に、LDS（レジスタ）命令がくる場合

LDS命令で、MACレジスタの内容を汎用レジスタからロードする場合、後述のとおり、LDS命令には乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



（h） DMULS.L命令の直後に、LDS.L（メモリ）命令がくる場合

LDS命令で、MACレジスタの内容をメモリからロードする場合、後述のとおり、LDS命令にはメモリのアクセスと乗算器のアクセスのためのMAステージが入ります。乗算器の動作中（mm）にLDSのMAが競合すると、そのMAはmmが終了するまで引き延ばされ（下図のM――A）、1つのスロットを形成します。また、このLDSのMAはIFと競合します。その様子を下図に示します。これらの図については、MAとIFの競合を考慮して書いてあります。



7.4.3 論理演算命令

(1) レジスタ－レジスタ間論理演算命令（共通）

●命令の種類

AND	Rm,Rn	TST	Rm,Rn
AND	#imm,R0	TST	#imm,R0
NOT	Rm,Rn	XOR	Rm,Rn
OR	Rm,Rn	XOR	#imm,R0
OR	#imm,R0		

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX			
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

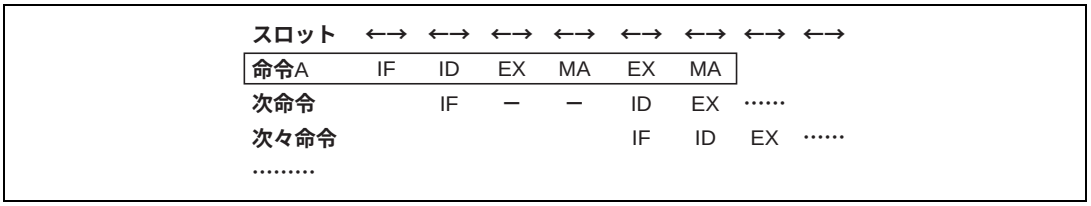
パイプラインは、IF、ID、EX の 3 段で終了します。EX ステージで、ALU をとおしてデータ演算は完結します。

(2) メモリ論理演算命令（共通）

●命令の種類

AND.B	#imm,@(R0,GBR)
OR.B	#imm,@(R0,GBR)
TST.B	#imm,@(R0,GBR)
XOR.B	#imm,@(R0,GBR)

●パイプライン



●動作説明

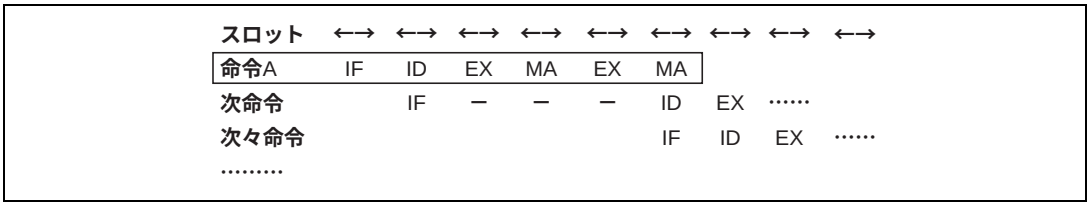
パイプラインは、IF、ID、EX、MA、EX、MA の 6 段で終了します。次命令の ID は 2 スロット分ストールされます。もちろん、これらの命令の MA は IF と競合します。

(3) TAS 命令（共通）

●命令の種類

TAS.B	@Rn

●パイプライン



●動作説明

パイプラインは、IF、ID、EX、MA、EX、MA の 6 段で終了します。次命令の ID は 3 スロット分ストールされます。もちろん、TAS 命令の MA は IF と競合します。

7.4.4 シフト命令

●命令の種類

ROTL	Rn	SHLL2	Rn
ROTR	Rn	SHLR2	Rn
ROTCL	Rn	SHLL8	Rn
ROTCR	Rn	SHLR8	Rn
SHAL	Rn	SHLL16	Rn
SHAR	Rn	SHLR16	Rn
SHLL	Rn		
SHLR	Rn		

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX			
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。EX ステージで、ALU をとおしてデータ演算は完結します。

7.4.5 分岐命令

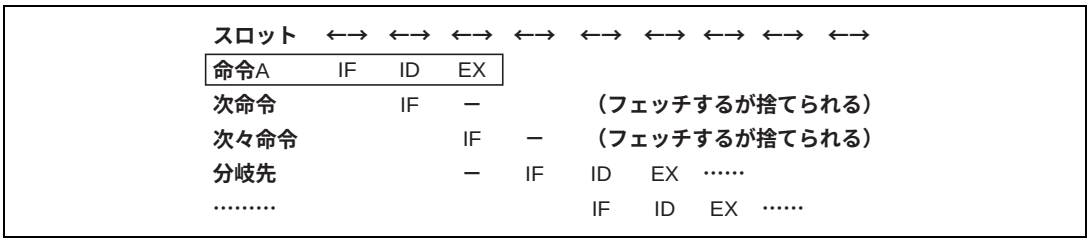
(1) 条件分岐命令（共通）

●命令の種類

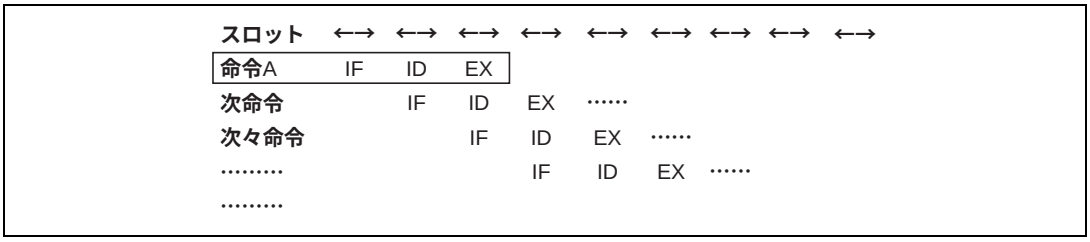
BF	label
BT	label

●パイプライン

（a） 条件が成立したとき



（b） 条件が成立しないとき



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。ID ステージで条件判断を行います。条件分岐命令は遅延分岐ではありません。

（a） 条件が成立したとき

EXステージで、分岐先アドレスを計算します。条件分岐命令（命令A）の次命令と次々命令は、フェッチしますが捨てられます。分岐先命令は、命令AのEXステージがあるスロットの次のスロットからフェッチを開始します。

（b） 条件が成立しないとき

IDステージで条件が成立しないと判断したら、EXステージでは何もせずに進みます。次命令も続けてフェッチし実行していきます。

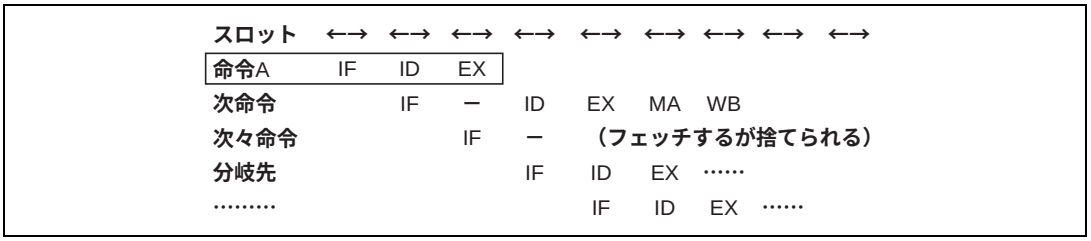
(2) 遅延付き条件分岐命（SH-2、SH-DSP）

●命令の種類

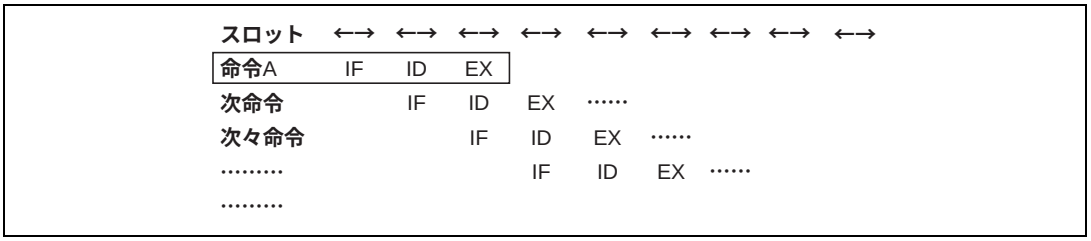
BF/S label
BT/S label

●パイプライン

（a） 条件が成立したとき



（b） 条件が成立しないとき



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。ID ステージで条件判断を行います。

（a） 条件が成立したとき
EXステージで、分岐先アドレスを計算します。条件分岐命令（命令A）の次命令はフェッチされ実行されますが、次々命令はフェッチされても捨てられます。分岐先命令は、命令AのEXステージがあるスロットの次のスロットからフェッチを開始します。

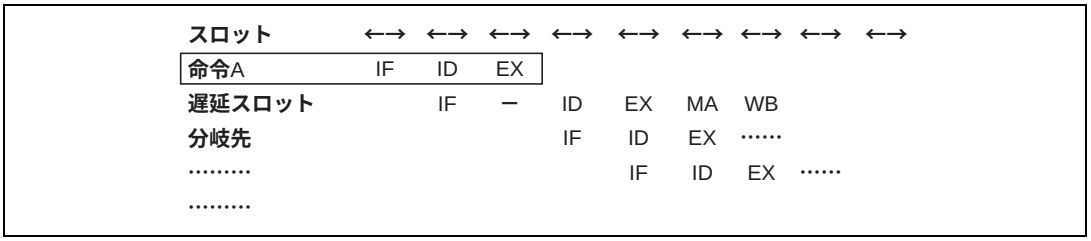
（b） 条件が成立しないとき
IDステージで条件が成立しないと判断したら、EXステージでは何もせずに進みます。次命令も続けてフェッチし実行していきます。

(3) 無条件分岐命令（共通もしくは SH-2、SH-DSP）

●命令の種類

BRA	label	JMP	@Rm
BRAF	Rm（SH-2、SH-DSP）	JSR	@Rm
BSR	label	RTS	
BSRF	Rm（SH-2、SH-DSP ）		

●パイプライン



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。無条件分岐命令は遅延分岐です。

EX ステージで、分岐先アドレスを計算します。無条件分岐命令（命令 A）の次命令すなわち遅延スロット命令は、フェッチしてから条件分岐命令のように捨てられず、そのまま実行します。ただし、この遅延スロット命令は ID ステージが 1 スロット分ストールされます。分岐先命令は、命令 A の EX ステージがあるスロットの次のスロットからフェッチを開始します。

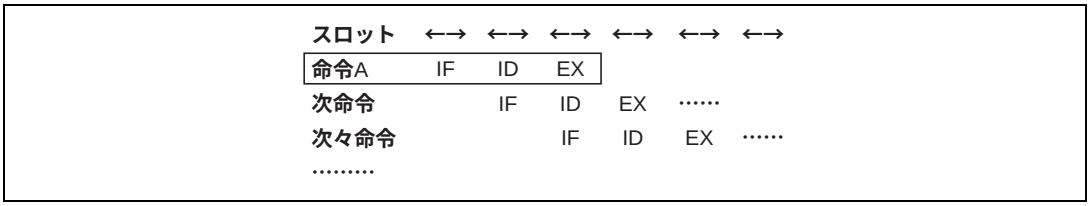
7.4.6 システム制御命令

(1) システム制御 ALU 命令（共通もしくは、SH-DSP）

●命令の種類

CLRT		SETT	
LDC	Rm,SR	STC	SR,Rn
LDC	Rm,GBR	STC	GBR,Rn
LDC	Rm,VBR	STC	VBR,Rn
LDC	Rm,MOD（SH-DSP）	STC	MOD,Rn（SH-DSP）
LDC	Rm,RE（SH-DSP）	STC	RE,Rn（SH-DSP）
LDC	Rm,RS（SH-DSP）	STC	RS,Rn（SH-DSP）
LDRE	@(disp,PC)	STS	PR,Rn
LDRS	@(disp,PC)		
LDS	Rm,PR		
NOP			
SETRC	Rm（SH-DSP）		
SETRC	#imm（SH-DSP）		

●パイプライン



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。EX ステージで、ALU をとおしてデータ演算は完結します。

(2) LDC.L 命令（共通もしくは、SH-DSP）

●命令の種類

LDC.L	@Rm+,SR	LDC.L	@Rm+,MOD（SH-DSP）
LDC.L	@Rm+,GBR	LDC.L	@Rm+,RE（SH-DSP）
LDC.L	@Rm+,VBR	LDC.L	@Rm+,RS（SH-DSP）

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	EX			
次命令		IF	ー	ー	ID	EX	……	
次々命令					IF	ID	EX	……
……								

●動作説明

パイプラインは、IF、ID、EX、MA、EX の 5 段で終了します。次命令の ID は 2 スロット分ストールされます。

(3) STC.L 命令（共通もしくは、SH-DSP）

●命令の種類

STC.L	SR,@-Rn	STC.L	MOD,@-Rn（SH-DSP）
STC.L	GBR,@-Rn	STC.L	RE,@-Rn（SH-DSP）
STC.L	VBR,@-Rn	STC.L	RS,@-Rn（SH-DSP）

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA			
次命令		IF	ー	ID	EX	……	
次々命令				IF	ID	EX	……
……							

●動作説明

パイプラインは、IF、ID、EX、MA の 4 段で終了します。次命令の ID は 1 スロット分ストールされます。

(4) LDS.L 命令（共通）

●命令の種類

LDS.L @Rm+,PR

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB	
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA、WB の 5 段で終了します。通常のロード命令と同様です。

(5) STS.L 命令（共通）

●命令の種類

STS.L PR,@-Rn

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA		
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA の 4 段で終了します。通常のストア命令と同様です。

(6) レジスタ→MAC、DSP 転送命令（共通もしくは、SH-DSP）

●命令の種類

```
CLRMAC
LDS     Rm,MACH
LDS     Rm,MACL
LDS     Rm,DSR（SH-DSP）
LDS     Rm,A0（SH-DSP）
LDS     Rm,X0（SH-DSP）
LDS     Rm,X1（SH-DSP）
LDS     Rm,Y0（SH-DSP）
LDS     Rm,Y1（SH-DSP）
```

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA		
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA の 4 段で終了します。MA は乗算器アクセスのためのステージです。この MA は IF と競合を起こします。すなわち、通常のストア命令と同様です。ただし、乗算器との競合を起こしますので、乗算命令、積和命令、倍精度乗算命令、倍精度積和命令の項を参照してください。

(7) メモリ→MAC、DSP 転送命令（共通もしくは、SH-DSP）

●命令の種類

```
LDS.L    @Rm+,MACH
LDS.L    @Rm+,MACL
LDS.L    @Rm+,DSR（SH-DSP）
LDS.L    @Rm+,A0（SH-DSP）
LDS.L    @Rm+,X0（SH-DSP）
LDS.L    @Rm+,X1（SH-DSP）
LDS.L    @Rm+,Y0（SH-DSP）
LDS.L    @Rm+,Y1（SH-DSP）
```

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA		
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA の 4 段で終了します。MA はメモリアクセスと乗算器アクセスのためのステージです。この MA は IF と競合を起こします。すなわち、通常のロード命令と同様です。ただし、乗算器との競合を起こしますので、乗算命令、積和命令、倍精度乗算命令、倍精度積和命の項を参照してください。

(8) MAC、DSP→レジスタ転送命令（共通もしくは、SH-DSP）

●命令の種類

STS	MACH,Rn
STS	MACL,Rn
STS	DSR,Rn（SH-DSP）
STS	A0,Rn（SH-DSP）
STS	X0,Rn（SH-DSP）
STS	X1,Rn（SH-DSP）
STS	Y0,Rn（SH-DSP）
STS	Y1,Rn（SH-DSP）

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB	
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA、WB の 5 段で終了します。MA は乗算器アクセスのためのステージです。この MA は IF と競合を起こします。すなわち、通常のロード命令と同様です。ただし、乗算器との競合を起こしますので、乗算命令、積和命令、倍精度乗算命令、倍精度積和命の項を参照してください。

(9) MAC、DSP→メモリ転送命令（共通もしくは、SH-DSP）

●命令の種類

STS.L	MACH,@-Rn
STS.L	MACL,@-Rn
STS.L	DSR,@-Rn（SH-DSP）
STS.L	A0,@-Rn（SH-DSP）
STS.L	X0,@-Rn（SH-DSP）
STS.L	X1,@-Rn（SH-DSP）
STS.L	Y0,@-Rn（SH-DSP）
STS.L	Y1,@-Rn（SH-DSP）

●パイプライン

スロット	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA		
次命令		IF	ID	EX	……	
次々命令			IF	ID	EX	……
……						

●動作説明

パイプラインは、IF、ID、EX、MA の 4 段で終了します。MA はメモリアクセスと乗算器アクセスのためのステージです。この MA は IF と競合を起こします。すなわち、通常のストア命令と同様です。ただし、乗算器との競合を起こしますので、乗算命令、積和命令、倍精度乗算命令、倍精度積和命の項を参照してください。

(10) RTE 命令（共通）

●命令の種類

RTE

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→	←→
RTE	IF	ID	EX	MA	MA				
遅延スロット		IF	ー	ー	ー	ID	EX	……	
分岐先						IF	ID	EX	……
……									

●動作説明

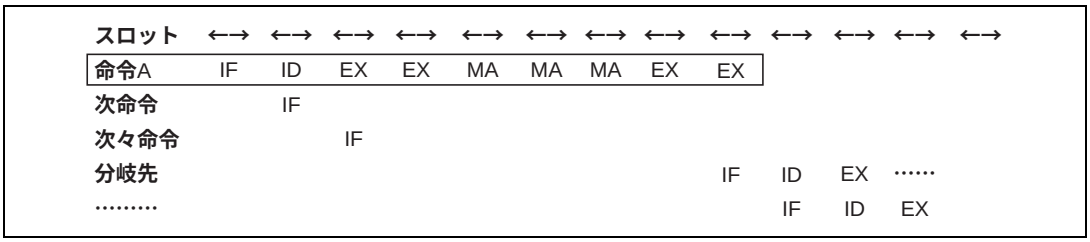
パイプラインは、IF、ID、EX、MA、MA の 5 段で終了します。これらの MA は IF と競合を起こしません。RTE は遅延分岐命令です。遅延スロット命令の ID は 3 スロット分ストールします。分岐先命令の IF は RTE の MA の次のスロットから開始されます。

(11) TRAP 命令（共通）

●命令の種類

TRAPA　#imm

●パイプライン



●動作説明

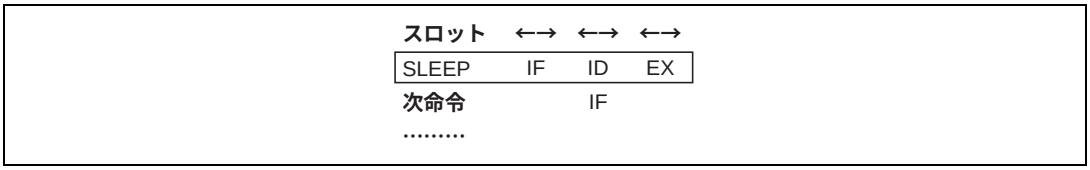
パイプラインは、IF、ID、EX、EX、MA、MA、MA、EX、EX の 9 段で終了します。これらの MA は IF と競合を起こしません。TRAP 命令は遅延分岐命令ではありません。TRAP 命令の次命令と次々命令はフェッチされますが、実行されずに捨てられます。分岐先命令の IF は TRAP 命令の 9 ステージ目の EX のスロットから開始されます。

(12) SLEEP 命令（共通）

●命令の種類

SLEEP

●パイプライン



●動作説明

パイプラインは、IF、ID、EX の 3 段で終了します。次命令の IF までは発行されます。SLEEP 命令を実行後、スリープモードまたはスタンバイモードに入ります。

7.4.7 DSP データ転送命令

(1) X メモリロード命令（SH-DSP）

●命令の種類

```
NOPX
MOVX.W   @Ax,Dx
MOVX.W   @Ax+,Dx
MOVX.W   @Ax+Ix,Dx
```

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB/DSP			
次命令		IF	ID	EX	MA	WB/DSP		
次々命令			IF	ID	EX	MA	WB/DSP	
・・・・・・								

●動作説明

パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。このデータ転送は X バスを経由して実行されるため、別命令の IF とは競合しません。

(2) Y メモリロード命令（SH-DSP）

●命令の種類

```
NOPY
MOVY.W   @Ay,Dy
MOVY.W   @Ay+,Dy
MOVY.W   @Ay+Iy,Dy
```

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB/DSP			
次命令		IF	ID	EX	MA	WB/DSP		
次々命令			IF	ID	EX	MA	WB/DSP	
・・・・・・								

●動作説明

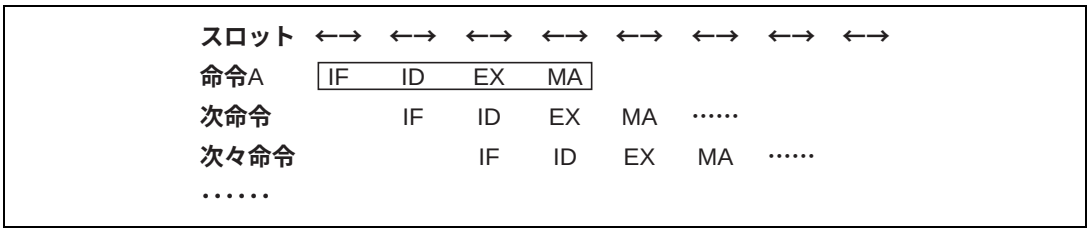
パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。このデータ転送は Y バスを経由して実行されるため、別命令の IF とは競合しません。

(3) X メモリストア命令（SH-DSP）

●命令の種類

```
MOVX.W   Da, @Ax
MOVX.W   Da, @Ax+
MOVX.W   Da, @Ax+Ix
```

●パイプライン



●動作説明

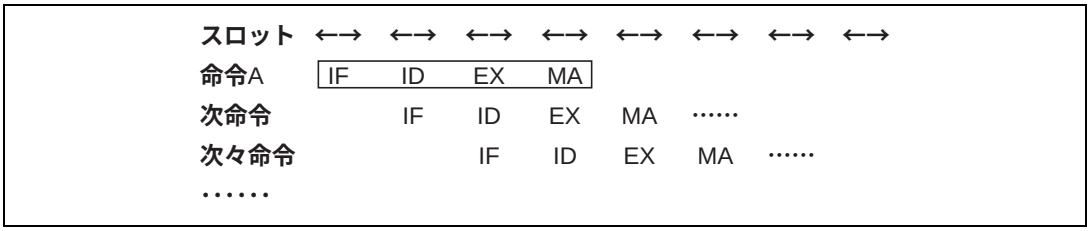
パイプラインは、IF、ID、EX、MA の 4 段です。DSP 演算命令の直後にこの命令で DSP 演算結果をストアしようとすると、競合が発生します（「7.2.2 **先行命令のデスティネーションレジスタを使うときの競合**」の（2）を参照）。

(4) Y メモリストア命令（SH-DSP）

●命令の種類

```
MOVY.W   Da, @Ay
MOVY.W   Da, @Ay+
MOVY.W   Da, @Ay+Iy
```

●パイプライン



●動作説明

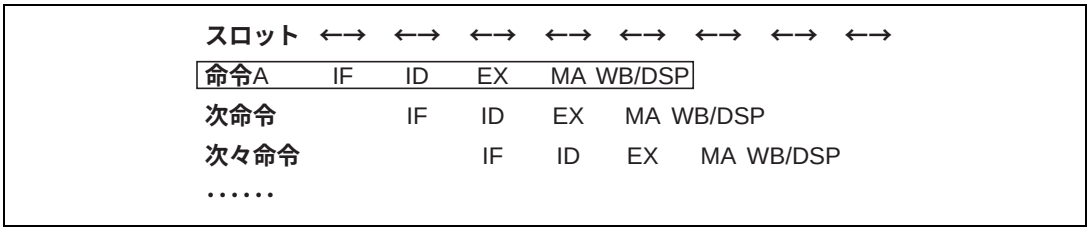
パイプラインは、IF、ID、EX、MA の 4 段です。DSP 演算命令の直後にこの命令で DSP 演算結果をストアしようとすると、競合が発生します（「7.2.2 **先行命令のデスティネーションレジスタを使うときの競合**」の（2）を参照）。

(5) シングルロード命令（SH-DSP）

●命令の種類

MOVS.W @-As,Ds
MOVS.W @As,Ds
MOVS.W @As+, Ds
MOVS.W @As+Is, Ds
MOVS.L @-As,Ds
MOVS.L @As,Ds
MOVS.L @As+, Ds
MOVS.L @As+Is, Ds

●パイプライン



●動作説明

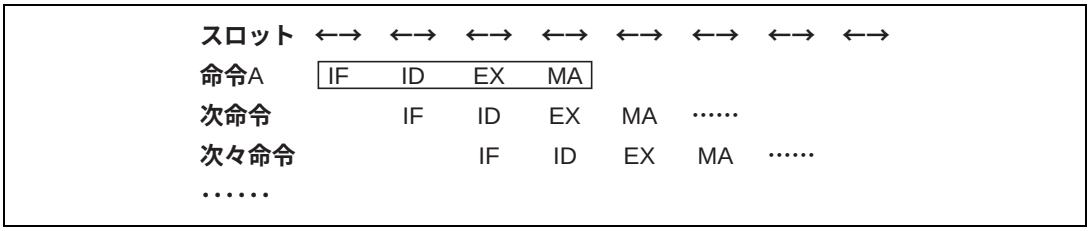
パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。この命令のデスティネーションレジスタを使う命令を置いても、競合は発生しません。

(6) シングルストア命令（SH-DSP）

●命令の種類

```
MOVS.W   Ds, @-As
MOVS.W   Ds, @As
MOVS.W   Ds, @As+
MOVS.W   Ds, @As+Is
MOVS.L   Ds, @-As
MOVS.L   Ds, @As
MOVS.L   Ds, @As+
MOVS.L   Ds, @As+I
```

●パイプライン



●動作説明

パイプラインは、IF、ID、EX、MA の 4 段です。DSP 演算命令の直後にこの命令で DSP 演算結果をストアしようとすると、競合が発生します（**「**7.2.2 **先行命令のデスティネーションレジスタを使うときの競合」**の（2）を参照）。

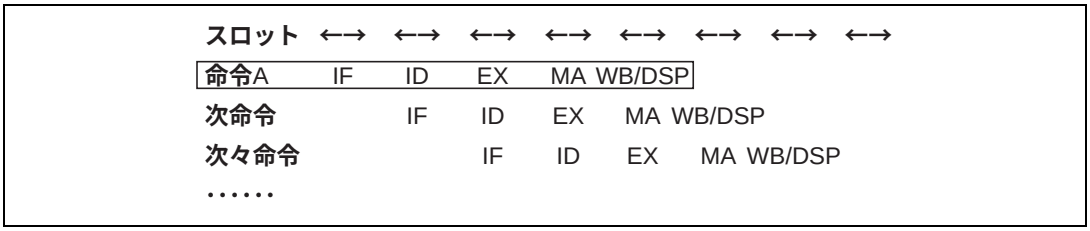
7.4.8 DSP 演算命令

(1) ALU 算術演算命令（SH-DSP）

●命令の種類

	PADD Sx, Sy,Dz(Du)		PNEG Sx,Dz
DCT	PADD Sx, Sy,Dz	DCT	PNEG Sx,Dz
DCF	PADD Sx, Sy,Dz	DCF	PNEG Sx,Dz
	PSUB Sx, Sy,Dz(Du)		PNEG Sy,Dz
DCT	PSUB Sx, Sy,Dz	DCT	PNEG Sy,Dz
DCF	PSUB Sx, Sy,Dz	DCF	PNEG Sy,Dz
	PCOPY Sx,Dz		PDEC Sx,Dz
DCT	PCOPY Sx,Dz	DCT	PDEC Sx,Dz
DCF	PCOPY Sx,Dz	DCF	PDEC Sx,Dz
	PCOPY Sy,Dz		PDEC Sy,Dz
DCT	PCOPY Sy,Dz	DCT	PDEC Sy,Dz
DCF	PCOPY Sy,Dz	DCF	PDEC Sy,Dz
	PDMSB Sx,Dz		PCLR Dz
DCT	PDMSB Sx,Dz	DCT	PCLR Dz
DCF	PDMSB Sx,Dz	DCF	PCLR Dz
	PDMSB Sy,Dz		PADDC Sx,Sy,Dz
DCT	PDMSB Sy,Dz		PSUBC Sx,Sy,Dz
DCF	PDMSB Sy,Dz		PCMP Sx,Sy
	PINC Sx,Dz		PABS Sx,Dz
DCT	PINC Sx,Dz		PABS Sy,Dz
DCF	PINC Sx,Dz		PRND Sx,Dz
	PINC Sy,Dz		PRND Sy,Dz
DCT	PINC Sy,Dz		
DCF	PINC Sy,Dz		

●パイプライン



●動作説明

パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。条件付き演算命令では条件が不成立の場合、WB/DSP ステージが無作動（ノー・オペレーション）となりますが、パイプラインは変わりません。

(2) ALU 論理演算命令（SH-DSP）

●命令の種類

	POR Sx,Sy,Dz
DCT	POR Sx,Sy,Dz
DCF	POR Sx,Sy,Dz
	PAND Sx,Sy,Dz
DCT	PAND Sx,Sy,Dz
DCF	PAND Sx,Sy,Dz
	PXOR Sx,Sy,Dz
DCT	PXOR Sx,Sy,Dz
DCF	PXOR Sx,Sy,Dz

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB/DSP			
次命令		IF	ID	EX	MA	WB/DSP		
次々命令			IF	ID	EX	MA	WB/DSP	
・・・・・・								

●動作説明

パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。条件付き演算命令では条件が不成立の場合、WB/DSP ステージが無作動（ノー・オペレーション）となりますが、パイプラインは変わりません。

(3) ALU 論理演算命令（SH-DSP）

●命令の種類

	PSHA Sx,Sy,Dz
DCT	PSHA Sx,Sy,Dz
DCF	PSHA Sx,Sy,Dz
	PSHA #imm,Dz
	PSHL Sx,Sy,Dz
DCT	PSHL Sx,Sy,Dz
DCF	PSHL Sx,Sy,Dz
	PSHL #imm,Dz

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB/DSP			
次命令		IF	ID	EX	MA	WB/DSP		
次々命令			IF	ID	EX	MA	WB/DSP	
・・・・・・								

●動作説明

パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。条件付き演算命令では条件が不成立の場合、WB/DSP ステージが無作動（ノー・オペレーション）となりますが、パイプラインは変わりません。

(4) 符号付き乗算命令（SH-DSP）

●命令の種類

PMULS Se,Sf,Dg

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB/DSP			
次命令		IF	ID	EX	MA	WB/DSP		
次々命令			IF	ID	EX	MA	WB/DSP	
・・・・・・								

●動作説明

パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。

(5) **レジスタ間転送命令**（SH-DSP）

●命令の種類

	PSTS MACH,Dz
DCT	PSTS MACH,Dz
DCF	PSTS MACH,Dz
	PSTS MACL,Dz
DCT	PSTS MACL,Dz
DCF	PSTS MACL,Dz
	PLDS Dz,MACH
DCT	PLDS Dz,MACH
DCF	PLDS Dz,MACH
	PLDS Dz,MACL
DCT	PLDS Dz,MACL
DCF	PLDS Dz,MACL

●パイプライン

スロット	←→	←→	←→	←→	←→	←→	←→	←→
命令A	IF	ID	EX	MA	WB/DSP			
次命令		IF	ID	EX	MA	WB/DSP		
次々命令			IF	ID	EX	MA	WB/DSP	
・・・・・・								

●動作説明

パイプラインは、IF、ID、EX、MA、WB/DSP の 5 段です。条件付き演算命令では条件が不成立の場合、WB/DSP ステージが無作動（ノー・オペレーション）となりますが、パイプラインは変わりません。この命令と並列に MOVX.W、MOVY.W、MOVS.W 又は MOVS.L でメモリロードを行うと、競合が発生します。また、この命令の直後に MOVX.W、MOVY.W、MOVS.W または MOVS.L でメモリストアを行うと、競合が発生します。（「7.2.4　DSP **レジスタ間転送とメモリ・ロード／ストア動作の競合**」を参照）。

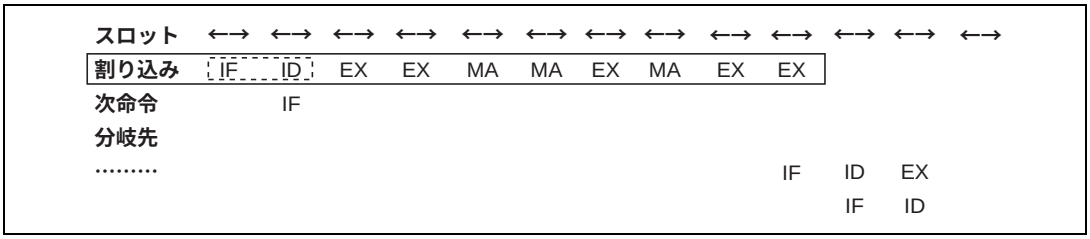
7.4.9 例外処理

(1) 割り込み例外処理（共通）

●命令の種類

割り込み例外処理

●パイプライン



●動作説明

割り込みは命令の ID ステージで受け付けられ、その ID ステージ以降を割り込み例外処理のシーケンスに置き換えます。

パイプラインは、IF、ID、EX、EX、MA、MA、EX、MA、EX、EX の 10 段で終了します。割り込み例外処理は遅延分岐ではありません。割り込み例外処理では、オーバランフェッチ（IF）が起こります。分岐先命令は割り込み例外処理の最後の EX があるスロットから IF を開始します。

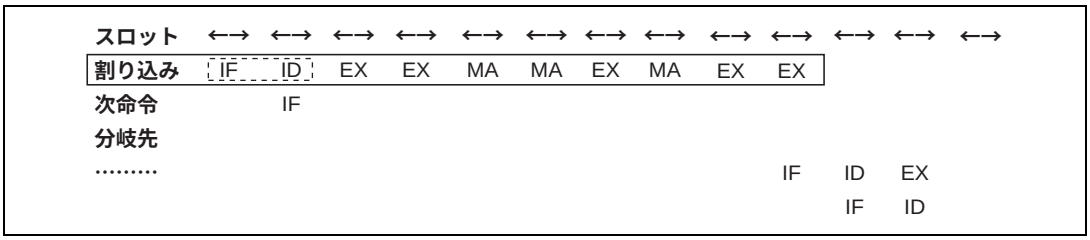
割り込み要因には、NMI、ユーザブレーク、IRQ、内蔵周辺モジュールによる割り込みがあります。

(2) アドレスエラー例外処理（共通）

●命令の種類

アドレスエラー例外処理

●パイプライン



●動作説明

アドレスエラーは命令の ID ステージで受け付けられ、その ID ステージ以降をアドレスエラー例外処理のシーケンスに置き換えます。

パイプラインは、IF、ID、EX、EX、MA、MA、EX、MA、EX、EX の 10 段で終了します。アドレスエラー例外処理は遅延分岐ではありません。アドレスエラー例外処理では、オーバランフェッチ（IF）が起こります。分岐先命令はアドレスエラー例外処理の最後の EX があるスロットから IF を開始します。

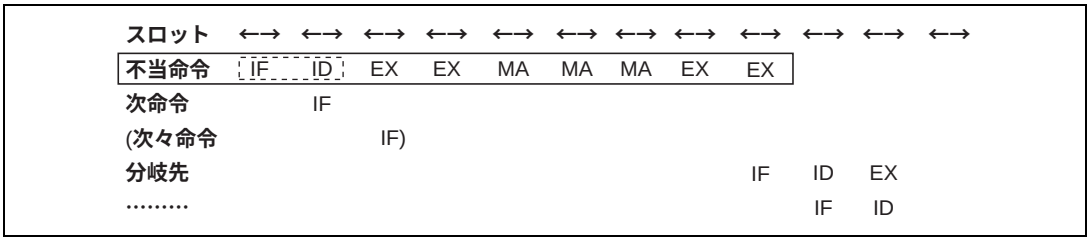
アドレスエラーの発生要因には、命令フェッチによるものとデータの読み出しあるいは書き込みによるものがあります。発生要因の詳細についてはハードウェアマニュアルを参照してください。

(3) 不当命令例外処理（共通）

●命令の種類

不当命令例外処理

●パイプライン



●動作説明

不当命令は命令の ID ステージで受け付けられ、その ID ステージ以降を不当命令例外処理のシーケンスに置き換えます。

パイプラインは、IF、ID、EX、EX、MA、MA、MA、EX、EX の 9 段で終了します。不当命令例外処理は遅延分岐ではありません。不当命令例外処理でも、オーバランフェッチ（IF）は起こります。IF が次命令だけか、あるいは次々命令でも起こるのかは、実行しようとしていた命令によります。分岐先命令は不当命令例外処理の最後の EX があるスロットから IF を開始します。

不当命令例外処理要因には、一般不当命令によるものとスロット不当命令によるものがあります。遅延分岐命令直後のスロット（遅延スロットとよぶ）以外に配置されている未定義コードをデコードすると一般不当命令例外処理となります。遅延スロットに配置されている未定義コードをデコードする、または遅延スロットにプログラムカウンタを書き換える命令を配置し、それをデコードするとスロット不当命令例外処理となります。

付録

A. 命令コード

A.1 CPU 命令の説明

CPU コアで実行される命令を、アルファベット順に示します。

表 A.1 **アルファベット順** CPU **命令**

命令		命令コード	動作	実行ステート	T ビット
ADD	#imm,Rn	0111nnnniiiiiiii	Rn＋imm→Rn	1	―
ADD	Rm,Rn	0011nnnnmmmm1100	Rn＋Rm→Rn	1	―
ADDC	Rm,Rn	0011nnnnmmmm1110	Rn＋Rm＋T→Rn、**キャリ**→T	1	**キャリ**
ADDV	Rm,Rn	0011nnnnmmmm1111	Rn＋Rm→Rn、**オーバフロー**→T	1	**オーバフロー**
AND	#imm,R0	11001001iiiiiiii	R0 & imm→R0	1	―
AND	Rm,Rn	0010nnnnmmmm1001	Rn & Rm→Rn	1	―
AND.B	#imm,@(R0,GBR)	11001101iiiiiiii	(R0＋GBR) & imm→ (R0＋GBR)	3	―
BF	label	10001011dddddddd	T＝0 **のとき** disp×2＋PC→PC、T＝1 **のとき** nop	3/1*[1]	―
BF/S	label	10001111dddddddd	T＝0 **のとき** disp×2＋PC→PC、T＝1 **のとき** nop	2/1*[1]	―
BRA	label	1010dddddddddddd	**遅延分岐**、disp×2＋PC→PC	2	―
BRAF	Rm	0000mmmm00100011	**遅延分岐**、Rm＋PC→PC	2	―
BSR	label	1011dddddddddddd	**遅延分岐**、PC→PR、disp×2＋PC→PC	2	―
BSRF	Rm	0000mmmm00000011	**遅延分岐**、PC→PR、Rm＋PC→PC	2	―
BT	label	10001001dddddddd	T＝1 **のとき** disp×2＋PC→PC、T＝0 **のとき** nop	3/1*[1]	―
BT/S	label	10001101dddddddd	T＝1 **のとき** disp×2＋PC→PC、T＝0 **のとき** nop	2/1*[1]	―
CLRMAC		0000000000101000	0→MACH、MACL	1	―
CLRT		0000000000001000	0→T	1	0
CMP/EQ	#imm,R0	10001000iiiiiiii	R0＝imm **のとき** 1→T	1	**比較結果**
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	Rn＝Rm **のとき** 1→T	1	**比較結果**
CMP/GE	Rm,Rn	0011nnnnmmmm0011	**有符号で** Rn≧Rm **のとき** 1→T	1	**比較結果**
CMP/GT	Rm,Rn	0011nnnnmmmm0111	**有符号で** Rn＞Rm **のとき** 1→T	1	**比較結果**
CMP/HI	Rm,Rn	0011nnnnmmmm0110	**無符号で** Rn＞Rm **のとき** 1→T	1	**比較結果**
CMP/HS	Rm,Rn	0011nnnnmmmm0010	**無符号で** Rn≧Rm **のとき** 1→T	1	**比較結果**
CMP/PL	Rn	0100nnnn00010101	Rn＞0 **のとき** 1→T	1	**比較結果**
CMP/PZ	Rn	0100nnnn00010001	Rn≧0 **のとき** 1→T	1	**比較結果**
CMP/STR	Rm,Rn	0010nnnnmmmm1100	**いずれかのバイトが等しいとき** 1→T	1	**比較結果**
DIV0S	Rm,Rn	0010nnnnmmmm0111	Rn **の** MSB→Q、Rm **の** MSB→M、M^Q→T	1	**計算結果**
DIV0U		0000000000011001	0→M/Q/T	1	0

命令		命令コード	動作	実行ステート	T ビット
DIV1	Rm,Rn	0011nnnnmmmm0100	1 ステップ除算 (Rn÷Rm)	1	計算結果
DMULS.L	Rm,Rn	0011nnnnmmmm1101	符号付きで Rn×Rm→MACH、MACHL	2〜4*2	—
DMULU.L	Rm,Rn	0011nnnnmmmm0101	符号なしで Rn×Rm→ MACH、MACL	2〜4*2	—
DT	Rn	0100nnnn00010000	Rn-1→Rn、Rn が 0 のとき 1→TRn が 0 以外のとき 0→T	1	比較結果
EXTS.B	Rm,Rn	0110nnnnmmmm1110	Rm をバイトから符号拡張→Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	Rm をワードから符号拡張→Rn	1	—
EXTU.B	Rm,Rn	0110nnnnmmmm1100	Rm をバイトからゼロ拡張→Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	Rm をワードからゼロ拡張→Rn	1	—
JMP	@Rm	0100mmmm00101011	遅延分岐、Rm→PC	2	—
JSR	@Rm	0100mmmm00001011	遅延分岐、PC→PR、 Rm→PC	2	—
LDC	Rm,GBR	0100mmmm00011110	Rm→GBR	1	—
LDC	Rm,MOD	0100mmmm01011110	Rm→MOD	1	—
LDC	Rm,RE	0100mmmm01111110	Rm→RE	1	—
LDC	Rm,RS	0100mmmm01101110	Rm→RS	1	—
LDC	Rm,SR	0100mmmm00001110	Rm→SR	1	LSB
LDC	Rm,VBR	0100mmmm00101110	Rm→VBR	1	—
LDC.L	@Rm+,GBR	0100mmmm00010111	(Rm)→GBR、Rm＋4→Rm	3	—
LDC.L	@Rm+,MOD	0100mmmm01010111	(Rm)→MOD、Rm＋4→Rm	3	—
LDC.L	@Rm+,RE	0100mmmm01110111	(Rm)→RE、Rm＋4→Rm	3	—
LDC.L	@Rm+,RS	0100mmmm01100111	(Rm)→RS、Rm＋4→Rm	3	—
LDC.L	@Rm+,SR	0100mmmm00000111	(Rm)→SR、Rm＋4→Rm	3	LSB
LDC.L	@Rm+,VBR	0100mmmm00100111	(Rm)→VBR、Rm＋4→Rm	3	—
LDRE	@(disp,PC)	10001110dddddddd	disp×2 ＋PC→RE	1	—
LDRS	@(disp,PC)	10001100dddddddd	disp×2 ＋PC→RS	1	—
LDS	Rm,A0	0100mmmm01111010	Rm→A0	1	—
LDS	Rm,DSR	0100mmmm01101010	Rm→DSR	1	—
LDS	Rm,MACH	0100mmmm00001010	Rm→MACH	1	—
LDS	Rm,MACL	0100mmmm00011010	Rm→MACL	1	—
LDS	Rm,PR	0100mmmm00101010	Rm→PR	1	—
LDS	Rm,X0	0100mmmm10001010	Rm→X0	1	—
LDS	Rm,X1	0100mmmm10011010	Rm→X1	1	—
LDS	Rm,Y0	0100mmmm10101010	Rm→Y0	1	—
LDS	Rm,Y1	0100mmmm10111010	Rm→Y1	1	—
LDS.L	@Rm+,A0	0100mmmm01110110	(Rm)→A0、Rm＋4→Rm	1	—
LDS.L	@Rm+,DSR	0100mmmm01100110	(Rm)→DSR、Rm＋4→Rm	1	—
LDS.L	@Rm+,MACH	0100mmmm00000110	(Rm)→MACH、Rm＋4→Rm	1	—
LDS.L	@Rm+,MACL	0100mmmm00010110	(Rm)→MACL、Rm＋4→Rm	1	—
LDS.L	@Rm+,PR	0100mmmm00100110	(Rm)→PR、Rm＋4→Rm	1	—
LDS.L	@Rm+,X0	0100mmmm10000110	(Rm)→Rm＋4→Rm	1	—
LDS.L	@Rm+,X1	0100mmmm10010110	(Rm)→Rm＋4→Rm	1	—

命令		命令コード	動作	実行ステート	Tビット
LDS.L	@Rm+,Y0	0100mmmm10100110	(Rm)→Rm＋4→Rm	1	—
LDS.L	@Rm+,Y1	0100mmmm10110110	(Rm)→Rm＋4→Rm	1	—
MAC.L	@Rm+,@Rn+	0000nnnnmmmm1111	**符号付きで** (Rn)×(Rm)＋MAC→MAC	3/(2〜4)*2	—
MAC.W	@Rm+,@Rn+	0100nnnnmmmm1111	**符号付きで** (Rn)×(Rm)＋MAC→MAC	3/(2)*2	—
MOV	#imm,Rn	1110nnnniiiiiiii	imm→**符号拡張**→Rn	1	—
MOV	Rm,Rn	0110nnnnmmmm0011	Rm→Rn	1	—
MOV.B	@(disp,GBR),R0	11000100dddddddd	(disp＋GBR)→**符号拡張**→R0	1	—
MOV.B	@(disp,Rm),R0	10000100mmmmdddd	(disp＋Rm)→**符号拡張**→R0	1	—
MOV.B	@(R0,Rm),Rn	0000nnnnmmmm1100	(R0＋Rm)→**符号拡張**→Rn	1	—
MOV.B	@Rm+,Rn	0110nnnnmmmm0100	(Rm)→**符号拡張**→Rn、Rm＋1→Rm	1	—
MOV.B	@Rm,Rn	0110nnnnmmmm0000	(Rm)→**符号拡張**→Rn	1	—
MOV.B	R0,@(disp,GBR)	11000000dddddddd	R0→(disp＋GBR)	1	—
MOV.B	R0,@(disp,Rn)	10000000nnnndddd	R0→(disp＋Rn)	1	—
MOV.B	Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm→(R0＋Rn)	1	—
MOV.B	Rm,@-Rn	0010nnnnmmmm0100	Rn－1→Rn、Rm→(Rn)	1	—
MOV.B	Rm,@Rn	0010nnnnmmmm0000	Rm→(Rn)	1	—
MOV.L	@(disp,GBR),R0	11000110dddddddd	(disp×4＋GBR)→R0	1	—
MOV.L	@(disp,PC),Rn	1101nnnndddddddd	(disp×4＋PC)→Rn	1	—
MOV.L	@(disp,Rm),Rn	0101nnnnmmmmdddd	(disp×4＋Rm)→Rn	1	—
MOV.L	@(R0,Rm),Rn	0000nnnnmmmm1110	(R0＋Rm)→Rn	1	—
MOV.L	@Rm+,Rn	0110nnnnmmmm0110	(Rm)→Rn、Rm＋4→Rm	1	—
MOV.L	@Rm,Rn	0110nnnnmmmm0010	(Rm)→Rn	1	—
MOV.L	R0,@(disp,GBR)	11000010dddddddd	R0→(disp×4＋GBR)	1	—
MOV.L	Rm,@(disp,Rn)	0001nnnnmmmmdddd	Rm→(disp×4＋Rn)	1	—
MOV.L	Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm→(R0＋Rn)	1	—
MOV.L	Rm,@-Rn	0010nnnnmmmm0110	Rn-4→Rn、Rm→(Rn)	1	—
MOV.L	Rm,@Rn	0010nnnnmmmm0010	Rm→(Rn)	1	—
MOV.W	@(disp,GBR),R0	11000101dddddddd	(disp×2＋GBR)→**符号拡張**→R0	1	—
MOV.W	@(disp,PC),Rn	1001nnnndddddddd	(disp×2＋PC)→**符号拡張**→Rn	1	—
MOV.W	@(disp,Rm),R0	10000101mmmmdddd	(disp×2＋Rm)→**符号拡張**→R0	1	—
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0＋Rm)→**符号拡張**→Rn	1	—
MOV.W	@Rm+,Rn	0110nnnnmmmm0101	(Rm)→**符号拡張**→Rn、Rm＋2→Rm	1	—
MOV.W	@Rm,Rn	0110nnnnmmmm0001	(Rm)→**符号拡張**→Rn	1	—
MOV.W	R0,@(disp,GBR)	11000001dddddddd	R0→(disp×2＋GBR)	1	—
MOV.W	R0,@(disp,Rn)	10000001nnnndddd	R0→(disp×2＋Rn)	1	—
MOV.W	Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm→(R0＋Rn)	1	—
MOV.W	Rm,@-Rn	0010nnnnmmmm0101	Rn－2→Rn、Rm→(Rn)	1	—
MOV.W	Rm,@Rn	0010nnnnmmmm0001	Rm→(Rn)	1	—
MOVA	@(disp,PC),R0	11000111dddddddd	disp×4＋PC→R0	1	—
MOVT	Rn	0000nnnn00101001	T→Rn	1	—
MUL.L	Rm,Rn	0000nnnnmmmm0111	Rn×Rm→MACL	2〜4*2	—

命令		命令コード	動作	実行ステート	Tビット
MULS.W	Rm,Rn	0010nnnnmmmm1111	**符号付きで** Rn×Rm→MAC	1〜3*2	—
MULU.W	Rm,Rn	0010nnnnmmmm1110	**符号なしで** Rn×Rm→MAC	1〜3*2	—
NEG	Rm,Rn	0110nnnnmmmm1011	0−Rm→Rn	1	—
NEGC	Rm,Rn	0110nnnnmmmm1010	0−Rm−T→Rn、**ボロー**→T	1	**ボロー**
NOP		0000000000001001	**無操作**	1	—
NOT	Rm,Rn	0110nnnnmmmm0111	~Rm → Rn	1	—
OR	#imm,R0	11001011iiiiiiii	R0 | imm→R0	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn | Rm→Rn	1	—
OR.B	#imm,@(R0,GBR)	11001111iiiiiiii	(R0＋GBR) | imm→ (R0＋GBR)	3	—
ROTCL	Rn	0100nnnn00100100	T←Rn←T	1	MSB
ROTCR	Rn	0100nnnn00100101	T→Rn→T	1	LSB
ROTL	Rn	0100nnnn00000100	T←Rn←MSB	1	MSB
ROTR	Rn	0100nnnn00000101	LSB→Rn→T	1	LSB
RTE		0000000000101011	**遅延分岐、スタック領域**→PC/SR	4	LSB
RTS		0000000000001011	**遅延分岐**、PR→PC	2	—
SETRC	#imm	10000010iiiiiiii	imm→RC (SR[23:16])、 0→SR[27:24]	1	—
SETRC	Rm	0100mmmm00010100	Rm[11:0]→RC (SR[27:16])	1	—
SETT		0000000000011000	1→T	1	1
SHAL	Rn	0100nnnn00100000	T←Rn←0	1	MSB
SHAR	Rn	0100nnnn00100001	MSB→Rn→T	1	LSB
SHLL	Rn	0100nnnn00000000	T←Rn←0	1	MSB
SHLL2	Rn	0100nnnn00001000	Rn＜＜2→Rn	1	—
SHLL8	Rn	0100nnnn00011000	Rn＜＜8→Rn	1	—
SHLL16	Rn	0100nnnn00101000	Rn＜＜16→Rn	1	—
SHLR	Rn	0100nnnn00000001	0→Rn→T	1	LSB
SHLR2	Rn	0100nnnn00001001	Rn＞＞2→Rn	1	—
SHLR8	Rn	0100nnnn00011001	Rn＞＞8→Rn	1	—
SHLR16	Rn	0100nnnn00101001	Rn＞＞16→Rn	1	—
SLEEP		0000000000011011	**スリープ**	3	—
STC	GBR,Rn	0000nnnn00010010	GBR→Rn	1	—
STC	MOD,Rn	0000nnnn01010010	MOD→Rn	1	—
STC	RE,Rn	0000nnnn01110010	RE→Rn	1	—
STC	RS,Rn	0000nnnn01100010	RS→Rn	1	—
STC	SR,Rn	0000nnnn00000010	SR→Rn	1	—
STC	VBR,Rn	0000nnnn00100010	VBR→Rn	1	—
STC.L	GBR,@-Rn	0100nnnn00010011	Rn−4→Rn、GBR→(Rn)	2	—
STC.L	MOD,@-Rn	0100nnnn01010011	Rn−4→Rn、MOD→(Rn)	2	—
STC.L	RE,@-Rn	0100nnnn01110011	Rn−4→Rn、RE→(Rn)	2	—
STC.L	RS,@-Rn	0100nnnn01100011	Rn−4→Rn、RS→(Rn)	2	—
STC.L	SR,@-Rn	0100nnnn00000011	Rn−4→Rn、SR→(Rn)	2	—
STC.L	VBR,@-Rn	0100nnnn00100011	Rn−4→Rn、VBR→(Rn)	2	—

命令		命令コード	動作	実行ステート	Tビット
STS	A0,Rn	0000nnnn01111010	A0→Rn	1	—
STS	DSR,Rn	0000nnnn01101010	DSR→Rn	1	—
STS	MACH,Rn	0000nnnn00001010	MACH→Rn	1	—
STS	MACL,Rn	0000nnnn00011010	MACL→Rn	1	—
STS	PR,Rn	0000nnnn00101010	PR→Rn	1	—
STS	X0,Rn	0000nnnn10001010	X0→Rn	1	—
STS	X1,Rn	0000nnnn10011010	X1→Rn	1	—
STS	Y0,Rn	0000nnnn10101010	Y0→Rn	1	—
STS	Y1,Rn	0000nnnn10111010	Y1→Rn	1	—
STS.L	A0,@-Rn	0100nnnn01110010	Rn－4→Rn、A0→(Rn)	1	—
STS.L	DSR,@-Rn	0100nnnn01100010	Rn－4→Rn、DSR→(Rn)	1	—
STS.L	MACH,@-Rn	0100nnnn00000010	Rn－4→Rn、MACH→(Rn)	1	—
STS.L	MACL,@-Rn	0100nnnn00010010	Rn－4→Rn、MACL→(Rn)	1	—
STS.L	PR,@-Rn	0100nnnn00100010	Rn－4→Rn、PR→(Rn)	1	—
STS.L	X0,@-Rn	0100nnnn10000010	Rn－4→Rn、X0→(Rn)	1	—
STS.L	X1,@-Rn	0100nnnn10010010	Rn－4→Rn、X1→(Rn)	1	—
STS.L	Y0,@-Rn	0100nnnn10100010	Rn－4→Rn、Y0→(Rn)	1	—
STS.L	Y1,@-Rn	0100nnnn10110010	Rn－4→Rn、Y1→(Rn)	1	—
SUB	Rm,Rn	0011nnnnmmmm1000	Rn－Rm→Rn	1	—
SUBC	Rm,Rn	0011nnnnmmmm1010	Rn－Rm－T→Rn、ボロー→T	1	ボロー
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn－Rm→Rn、アンダフロー→T	1	アンダフロー
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm→下位 2 バイトの上下バイト交換→Rn	1	—
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm→上下ワード交換→Rn	1	—
TAS.B	@Rn	0100nnnn00011011	(Rn)が 0 のとき 1→T、1→MSBof(Rn)	4	テスト結果
TRAPA	#imm	11000011iiiiiiii	PC/SR→スタック領域、(imm×4＋VBR)→PC	8	—
TST	#imm,R0	11001000iiiiiiii	R0 & imm、結果が 0 のとき 1→T	1	テスト結果
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm、結果が 0 のとき 1→T	1	テスト結果
TST.B	#imm,@(R0,GBR)	11001100iiiiiiii	(R0＋GBR)&imm、結果が 0 のとき 1→T	3	テスト結果
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XOR.B	#imm,@(R0,GBR)	11001110iiiiiiii	(R0＋GBR) ^ imm → (R0＋GBR)	3	—
XTRCT	Rm,Rn	0010nnnnmmmm1101	Rm：Rn の中央 32 ビット→Rn	1	—

【注】 *1　分岐しないときは 1 ステートになります。
*2　通常実行ステートを示します。前後の命令との競合関係により実行ステート数は変わります。

A.2　追加された CPU 命令

SH-2 に対して追加された SH-DSP の CPU 命令（3 種、40 命令）を**表** A.2 に示します。
SH-1 に対して追加された SH-2 の CPU 命令（6 種、9 命令）を**表** A.3 に示します。

表 A.2　SH-2 **に追加された** SH-DSP **の** CPU **命令**

命令		命令コード	動作	実行ステート	T ビット
LDC	Rm,MOD	0100mmmm01011110	Rm→MOD	1	—
LDC	Rm,RE	0100mmmm01111110	Rm→RE	1	—
LDC	Rm,RS	0100mmmm01101110	Rm→RS	1	—
LDC.L	@Rm+,MOD	0100mmmm01010111	(Rm)→MOD、Rm＋4→Rm	3	—
LDC.L	@Rm+,RE	0100mmmm01110111	(Rm)→RE、Rm＋4→Rm	3	—
LDC.L	@Rm+,RS	0100mmmm01100111	(Rm)→RS、Rm＋4→Rm	3	—
LDRE	@(disp,PC)	10001110dddddddd	disp×2＋PC→RE	1	—
LDRS	@(disp,PC)	10001100dddddddd	disp×2＋PC→RS	1	—
LDS	Rm,DSR	0100mmmm01101010	Rm→DSR	1	—
LDS	Rm,A0	0100mmmm01111010	Rm→A0	1	—
LDS	Rm,X0	0100mmmm10001010	Rm→X0	1	—
LDS	Rm,X1	0100mmmm10011010	Rm→X1	1	—
LDS	Rm,Y0	0100mmmm10101010	Rm→Y0	1	—
LDS	Rm,Y1	0100mmmm10111010	Rm→Y1	1	—
LDS.L	@Rm+,DSR	0100mmmm01100110	(Rm)→DSR、Rm＋4→Rm	1	—
LDS.L	@Rm+,A0	0100mmmm01110110	(Rm)→A0、Rm＋4→Rm	1	—
LDS.L	@Rm+,X0	0100mmmm10000110	(Rm)→X0、Rm＋4→Rm	1	—
LDS.L	@Rm+,X1	0100mmmm10010110	(Rm)→X1、Rm＋4→Rm	1	—
LDS.L	@Rm+,Y0	0100mmmm10100110	(Rm)→Y0、Rm＋4→Rm	1	—
LDS.L	@Rm+,Y1	0100mmmm10110110	(Rm)→Y1、Rm＋4→Rm	1	—
SETRC	Rm	0100mmmm00010100	Rm［11：0］→ RC(SR［27：16］)、**繰り返し制御フラグ**→RF1、RF0	1	—
SETRC	#imm	10000010iiiiiiii	imm→RC(SR［23：16］)、zeros→SR［27：24］、**繰り返し制御フラグ**→RF1、RF0	1	—
STC	MOD,Rn	0000nnnn01010010	MOD→Rn	1	—
STC	RE,Rn	0000nnnn01110010	RE→Rn	1	—
STC	RS,Rn	0000nnnn01100010	RS→Rn	1	—
STC.L	MOD,@-Rn	0100nnnn01010011	Rn－4→Rn、MOD→(Rn)	2	—
STC.L	RE,@-Rn	0100nnnn01110011	Rn－4→Rn、RE→(Rn)	2	—
STC.L	RS,@-Rn	0100nnnn01100011	Rn－4→Rn、RS→(Rn)	2	—
STS	DSR,Rn	0000nnnn01101010	DSR→Rn	1	—
STS	A0,Rn	0000nnnn01111010	A0→Rn	1	—
STS	X0,Rn	0000nnnn10001010	X0→Rn	1	—
STS	X1,Rn	0000nnnn10011010	X1→Rn	1	—
STS	Y0,Rn	0000nnnn10101010	Y0→Rn	1	—
STS	Y1,Rn	0000nnnn10111010	Y1→Rn	1	—

命令	命令コード	動作	実行ステート	T ビット
STS.L DSR,@-Rn	0100nnnn01100010	Rn－4→Rn、DSR→(Rn)	1	—
STS.L A0,@-Rn	0100nnnn01110010	Rn－4→Rn、A0→(Rn)	1	—
STS.L X0,@-Rn	0100nnnn10000010	Rn－4→Rn、X0→(Rn)	1	—
STS.L X1,@-Rn	0100nnnn10010010	Rn－4→Rn、X1→(Rn)	1	—
STS.L Y0,@-Rn	0100nnnn10100010	Rn－4→Rn、Y0→(Rn)	1	—
STS.L Y1,@-Rn	0100nnnn10110010	Rn－4→Rn、Y1→(Rn)	1	—

表 A.3　SH-1 **に追加された** SH-2 **の** CPU **命令**

命令	動作	命令コード	実行ステート	T ビット
BF/S label	T＝0 **のとき** disp×2＋PC→PC、T＝1 **のとき** nop	10001111dddddddd	2／1*2	—
BRAF Rm	**遅延分岐、**Rm＋PC→PC	0000mmmm00100011	2	—
BSRF Rm	**遅延分岐、**PC→PR、Rm＋PC→PC	0000mmmm00000011	2	—
BT/S label	T＝1 **のとき** disp×2＋PC→PC、T＝0 **のとき** nop	10001101dddddddd	2／1*2	—
DMULS.L Rm,Rn	**符号付きで** Rn×Rm→MACH、MACL32×32→64 **ビット**	0011nnnnmmmm1101	2(〜4)*1	—
DMULU.L Rm,Rn	**符号なしで** Rn×Rm→MACH、MACL32×32→64 **ビット**	0011nnnnmmmm0101	2(〜4)*1	—
DT Rn	Rn－1→Rn、Rn **が** 0 **のとき** 1→T、Rn **が** 0 **以外のとき** 0→T	0100nnnn00010000	1	**比較結果**
MAC.L @Rm+,@Rn+	**符号付きで**(Rn)×(Rm)＋MAC→MAC	0000nnnnmmmm1111	2(〜4)*1	—
MUL.L Rm,Rn	Rn×Rm→MACL	0000nnnnmmmm0111	2(〜4)*1	

A.3　DSP データ転送命令

DSP データ転送命令を、アルファベット順に示します。

表 A.4　アルファベット順 DSP データ転送命令

命令	動作	命令コード	実行ステート	DC ビット	適用命令 SH-1	SH-2	SH-DSP
MOVS.L @-As,Ds	As－4→As、(As)→Ds	111101AADDDD0010	1	－	－	－	○
MOVS.L @As,Ds	(As)→Ds	111101AADDDD0110	1	－	－	－	○
MOVS.L @As+,Ds	(As)→Ds、As＋4→As	111101AADDDD1010	1	－	－	－	○
MOVS.L @As+Ix,Ds	(As)→Ds、As＋Is→As	111101AADDDD1110	1	－	－	－	○
MOVS.L Ds,@-As	As－4→As、Ds→(As)	111101AADDDD0011	1	－	－	－	○
MOVS.L Ds,@As	Ds→(As)	111101AADDDD0111	1	－	－	－	○
MOVS.L Ds,@As+	Ds→(As)、As＋4→As	111101AADDDD1011	1	－	－	－	○
MOVS.L Ds,@As+Is	Ds→(As)、As＋Is→As	111101AADDDD1111	1	－	－	－	○
MOVS.W @-As,Ds	As－2→As、(As)→MSW of Ds、0→LSW of Ds	111101AADDDD0000	1	－	－	－	○
MOVS.W @As,Ds	(As)→MSW of Ds、0→LSW of Ds	111101AADDDD0100	1	－	－	－	○
MOVS.W @As+,Ds	(As)→MSW of Ds、0→LSW of Ds、As＋2→As	111101AADDDD1000	1	－	－	－	○
MOVS.W @As+Is,Ds	(As)→MSW of Ds、0→LSW of Ds、As＋Is→As	111101AADDDD1100	1	－	－	－	○
MOVS.W Ds,@-As	As－2→As、MSW of Ds→(As)	111101AADDDD0001	1	－	－	－	○
MOVS.W Ds,@As	MSW of Ds→(As)	111101AADDDD0101	1	－	－	－	○
MOVS.W Ds,@As+	MSW of Ds→(As)、As＋2→As	111101AADDDD1001	1	－	－	－	○
MOVS.W Ds,@As+Is	MSW of Ds→(As)、As＋Is→As	111101AADDDD1101	1	－	－	－	○
MOVX.W @Ax,Dx	(Ax)→MSW of Dx、0→LSW of Dx	111100A*D*0*01**	1	－	－	－	○
MOVX.W @Ax+,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax＋2→Ax	111100A*D*0*10**	1	－	－	－	○
MOVX.W @Ax+Ix,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax＋Ix→Ax	111100A*D*0*11**	1	－	－	－	○
MOVX.W Da,@Ax	MSW of Da→(Ax)	111100A*D*1*01**	1	－	－	－	○
MOVX.W Da,@Ax+	MSW of Da→(Ax)、Ax＋2→Ax	111100A*D*1*10**	1	－	－	－	○
MOVX.W Da,@Ax+Ix	MSW of Da→(Ax)、Ax＋Ix→Ax	111100A*D*1*11**	1	－	－	－	○
MOVY.W @Ay,Dy	(Ay)→MSW of Dy、0→LSW of Dy	111100*A*D*0**01	1	－	－	－	○
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay＋2→Ay	111100*A*D*0**10	1	－	－	－	○
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay＋Iy→Ay	111100*A*D*0**11	1	－	－	－	○
MOVY.W Da,@Ay	MSW of Da→(Ay)	111100*A*D*1**01	1	－	－	－	○
MOVY.W Da,@Ay+	MSW of Da→(Ay)、Ay＋2→Ay	111100*A*D*1**10	1	－	－	－	○
MOVY.W Da,@Ay+Iy	MSW of Da→(Ay)、Ay＋Iy→Ay	111100*A*D*1**11	1	－	－	－	○
NOPX	No Operation	1111000*0*0*00**	1	－	－	－	○
NOPY	No Operation	111100*0*0*0**00	1	－	－	－	○

【注】 MSW：**オペランドの上位ワード**
LSW：**オペランドの下位ワード**

A.4　DSP 演算命令

DSP 演算命令を、アルファベット順に示します。

表 A.5　アルファベット順 DSP 演算命令

命令	動作	命令コード	実行ステート	DC ビット	適用命令 SH-1	SH-2	SH-DSP
PABS Sx,Dz	**もし** Sx≧0 **ならば** Sx→Dz **もし** Sx＜0 **ならば** 0－Sx→Dz	111110********** 10001000xx00zzzz	1	**更新**	－	－	○
PABS Sy,Dz	**もし** Sy≧0 **ならば** Sy→Dz **もし** Sy＜0 **ならば** 0－Sy→Dz	111110********** 1010100000yyzzzz	1	**更新**	－	－	○
PADD Sx,Sy,Dz	Sx＋Sy→Dz	111110********** 10110001xxyyzzzz	1	**更新**	－	－	○
DCT PADD Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx＋Sy→Dz **もし** 0 **ならば** nop.	111110********** 10110010xxyyzzzz	1	－	－	－	○
DCF PADD Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx＋Sy→Dz **もし** 1 **ならば** nop.	111110********** 10110011xxyyzzzz	1	－	－	－	○
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx＋Sy→Du Se **の上位ワード×**sf **の上位ワード→**Dg	111110********** 0111eeffxxyygguu	1	**更新**[*1]	－	－	○
PADDC Sx,Sy,Dz	Sx＋Sy＋DC→Dz	111110********** 10110000xxyyzzzz	1	**更新**	－	－	○
PAND Sx,Sy,Dz	Sx & Sy→Dz、Dz **の下位ワードクリア**	111110********** 10010101xxyyzzzz	1	**更新**	－	－	○
DCT PAND Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx&Sy→Dz、Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10010110xxyyzzzz	1	－	－	－	○
DCF PAND Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx&Sy→Dz、Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10010111xxyyzzzz	1	－	－	－	○
PCLR Dz	H'00000000→Dz	111110********** 1000110100000zzzz	1	**更新**	－	－	○
DCT PCLR Dz	**もし** DC＝1 **ならば** H'00000000→Dz **もし** 0 **ならば** nop.	111110********** 1000111000000zzzz	1	－	－	－	○
DCF PCLR Dz	**もし** DC＝0 **ならば** H'00000000→Dz **もし** 1 **ならば** nop.	111110********** 1000111100000zzzz	1	－	－	－	○
PCMP Sx,Sy	Sx－Sy	111110********** 10000100xxyy0000	1	**更新**	－	－	○
PCOPY Sx,Dz	Sx→Dz	111110********** 11011001xx00zzzz	1	**更新**	－	－	○
PCOPY Sy,Dz	Sy→Dz	111110********** 1111100100yyzzzz	1	**更新**	－	－	○
DCT PCOPY Sx,Dz	**もし** DC＝1 **ならば** Sx→Dz **もし** 0 **ならば** nop.	111110********** 11011010xx00zzzz	1	－	－	－	○
DCT PCOPY Sy,Dz	**もし** DC＝1 **ならば** Sy→Dz **もし** 0 **ならば** nop.	111110********** 1111101000yyzzzz	1	－	－	－	○

命令	動作	命令コード	実行ステート	DC ビット	適用命令 SH-1	SH-2	SH-DSP
DCF PCOPY Sx,Dz	**もし** DC=0 **ならば** Sx→Dz **もし** 1 **ならば** nop.	111110********** 11011011xx00zzzz	1	−	−	−	○
DCF PCOPY Sy,Dz	**もし** DC=0 **ならば** Sy→Dz **もし** 1 **ならば** nop.	111110********** 1111101100yyzzzz	1	−	−	−	○
PDEC Sx,Dz	Sx **の上位ワード**−1→Dz **の上位ワード、** Dz **の下位ワードをクリア**	111110********** 10001001xx00zzzz	1	**更新**	−	−	○
PDEC Sy,Dz	Sy **の上位ワード**−1→Dz **の上位ワード、** Dz **の下位ワードをクリア**	111110********** 10101001xx00zzzz	1	**更新**	−	−	○
DCT PDEC Sx,Dz	**もし** DC=1 **ならば** Sx **の上位ワード**−1→Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10001010xx00zzzz	1	−	−	−	○
DCT PDEC Sy,Dz	**もし** DC=1 **ならば** Sy **の上位ワード**−1→Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10101010xx00zzzz	1	−	−	−	○
DCF PDEC Sx,Dz	**もし** DC=0 **ならば** Sx **の上位ワード**−1→Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10001011xx00zzzz	1	−	−	−	○
DCF PDEC Sy,Dz	**もし** DC=0 **ならば** Sy **の上位ワード**−1→Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10101011xx00zzzz	1	−	−	−	○
PDMSB Sx,Dz	Sx **データの** MSB **位置→**Dz **の上位ワード、**Dz **の下位ワードクリア**	111110********** 10011101xx00zzzz	1	**更新**	−	−	○
PDMSB Sy,Dz	Sy **データの** MSB **位置→**Dz **の上位ワード、**Dz **の下位ワードクリア**	111110********** 1011110100yyzzzz	1	**更新**	−	−	○
DCT PDMSB Sx,Dz	**もし** DC=1 **ならば** Sx **データの** MSB **位置→**Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10011110xx00zzzz	1	−	−	−	○
DCT PDMSB Sy,Dz	**もし** DC=1 **ならば** Sy **データの** MSB **位置→**Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 1011111000yyzzzz	1	−	−	−	○
DCF PDMSB Sx,Dz	**もし** DC=0 **ならば** Sx **データの** MSB **位置→**Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10011111xx00zzzz	1	−	−	−	○
DCF PDMSB Sy,Dz	**もし** DC=0 **ならば** Sy **データの** MSB **位置→** Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 1011111100yyzzzz	1	−	−	−	○
PINC Sx,Dz	Sx **の上位ワード**+1→ Dz **の上位ワード、**Dz **の下位ワードクリア**	111110********** 10011001xx00zzzz	1	**更新**	−	−	○
PINC Sy,Dz	Sy **の上位ワード**+1→ Dz **の上位ワード、**Dz **の下位ワードクリア**	111110********** 1011100100yyzzzz	1	**更新**	−	−	○
DCT PINC Sx,Dz	**もし** DC=1 **ならば** Sx **の上位ワード**+1→Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10011010xx00zzzz	1	−	−	−	○

命令		動作	命令コード	実行	DC	適用命令		
				ステート	ビット	SH-1	SH-2	SH-DSP
DCT	PINC Sy,Dz	**もし** DC＝1 **ならば** Sy **の上位ワード**＋1→ Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 1011101000yyzzzz	1	−	−	−	○
DCF	PINC Sx,Dz	**もし** DC＝0 **ならば** Sx **の上位ワード**＋1→ Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10011011xx00zzzz	1	−	−	−	○
DCF	PINC Sy,Dz	**もし** DC＝0 **ならば** Sy **の上位ワード**＋1→ Dz **の上位ワード、**Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 1011101100yyzzzz	1	−	−	−	○
	PLDS Dz,MACH	Dz→MACH	111110********** 111011010000zzzz	1	−	−	−	○
	PLDS Dz,MACL	Dz→MACL	111110********** 111111010000zzzz	1	−	−	−	○
DCT	PLDS Dz,MACH	**もし** DC＝1 **ならば** Dz→MACH **もし** 0 **ならば** nop.	111110********** 111011100000zzzz	1	−	−	−	○
DCT	PLDS Dz,MACL	**もし** DC＝1 **ならば** Dz→MACL **もし** 0 **ならば** nop.	111110********** 111111100000zzzz	1	−	−	−	○
DCF	PLDS Dz,MACH	**もし** DC＝0 **ならば** Dz→MACH **もし** 1 **ならば** nop.	111110********** 111011110000zzzz	1	−	−	−	○
DCF	PLDS Dz,MACL	**もし** DC＝0 **ならば** Dz→MACL **もし** 1 **ならば** nop.	111110********** 111111110000zzzz	1	−	−	−	○
	PMULS Se,Sf,Dg	Se **の上位ワード**×sf **の上位ワード**→Dg	111110********** 0100eeff0000gg00	1	−	−	−	○
	PNEG Sx,Dz	0−Sx→Dz	111110********** 11001001xx00zzzz	1	**更新**	−	−	○
	PNEG Sy,Dz	0−Sy→Dz	111110********** 1110100100yyzzzz	1	**更新**	−	−	○
DCT	PNEG Sx,Dz	**もし** DC＝1 **ならば** 0−Sx→Dz **もし** 0 **ならば** nop.	111110********** 11001010xx00zzzz	1	−	−	−	○
DCT	PNEG Sy,Dz	**もし** DC＝1 **ならば** 0−Sy→Dz **もし** 0 **ならば** nop.	111110********** 1110101000yyzzzz	1	−	−	−	○
DCF	PNEG Sx,Dz	**もし** DC＝0 **ならば** 0−Sx→Dz **もし** 1 **ならば** nop.	111110********** 11001011xx00zzzz	1	−	−	−	○
DCF	PNEG Sy,Dz	**もし** DC＝0 **ならば** 0−Sy→Dz **もし** 1 **ならば** nop.	111110********** 1110101100yyzzzz	1	−	−	−	○
	POR Sx,Sy,Dz	Sx	Sy→Dz、Dz **の下位ワードクリア**	111110********** 10110101xxyyzzzz	1	**更新**	−	−	○
DCT	POR Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx|Sy→Dz、Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10110110xxyyzzzz	1	−	−	−	○
DCF	POR Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx|Sy→Dz、Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10110111xxyyzzzz	1	−	−	−	○

命令		動作	命令コード	実行ステート	DC ビット	適用命令 SH-1	SH-2	SH-DSP
	PRND Sx,Dz	Sx＋H'00008000→Dz Dz **の下位ワードクリア**	111110********** 10011000xx00zzzz	1	**更新**	−	−	○
	PRND Sy,Dz	Sy＋H'00008000→Dz Dz **の下位ワードクリア**	111110********** 1011100000yyzzzz	1	**更新**	−	−	○
	PSHA Sx,Sy,Dz	**もし** Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** Sy＜0 **ならば** Sx＞＞Sy→Dz	111110********** 10010001xxyyzzzz	1	**更新**	−	−	○
DCT	PSHA Sx,Sy,Dz	**もし** DC＝1 & Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** DC＝1 & Sy＜0 **ならば** Sx＞＞Sy→Dz **もし** DC＝0 **ならば** nop.	111110********** 10010010xxyyzzzz	1	−	−	−	○
DCF	PSHA Sx,Sy,Dz	**もし** DC＝0 & Sy≧0 **ならば** Sx＜＜Sy→Dz **もし** DC＝0 & Sy＜0 **ならば** Sx＞＞Sy→Dz **もし** DC＝1 **ならば** nop.	111110********** 10010011xxyyzzzz	1	−	−	−	○
	PSHA #imm,Dz	**もし** imm≧0 **ならば** Dz＜＜imm→Dz **もし** imm＜0 **ならば** Dz＞＞imm→Dz	111110********** 00010iiiiiiizzzz	1	**更新**	−	−	○
	PSHL Sx,Sy,Dz	**もし** Sy≧0 **ならば** Sx＜＜Sy→Dz、 Dz **の下位ワードクリア** **もし** Sy＜0 **ならば** Sx＞＞Sy→Dz、 Dz **の下位ワードクリア**	111110********** 10000001xxyyzzzz	1	**更新**	−	−	○
DCT	PSHL Sx,Sy,Dz	**もし** DC＝1 & Sy≧0 **ならば** Sx＜＜Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝1 & Sy＜0 **ならば** Sx＞＞Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝0 **ならば** nop.	111110********** 10000010xxyyzzzz	1	−	−	−	○
DCF	PSHL Sx,Sy,Dz	**もし** DC＝0 & Sy≧0 **ならば** Sx＜＜Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝0 & Sy＜0 **ならば** Sx＞＞Sy→Dz、Dz **の下位ワードクリア** **もし** DC＝1 **ならば** nop.	111110********** 10000011xxyyzzzz	1	−	−	−	○
	PSHL #imm,Dz	**もし** imm≧0 **ならば** Dz＜＜imm→Dz、Dz **の下位ワードクリア** **もし** imm＜0 **ならば** Dz＞＞imm→Dz、Dz **の下位ワードクリア**	111110********** 00000iiiiiiizzzz	1	**更新**	−	−	○
	PSTS MACH,Dz	MACH→Dz	111110********** 1100110100000zzzz	1	−	−	−	○
	PSTS MACL,Dz	MACL→Dz	111110********** 1101110100000zzzz	1	−	−	−	○
DCT	PSTS MACH,Dz	**もし** DC＝1 **ならば** MACH→Dz **もし** 0 **ならば** nop.	111110********** 1100111000000zzzz	1	−	−	−	○
DCT	PSTS MACL,Dz	**もし** DC＝1 **ならば** MACL→Dz **もし** 0 **ならば** nop.	111110********** 1101111000000zzzz	1	−	−	−	○
DCF	PSTS MACH,Dz	**もし** DC＝0 **ならば** MACH→Dz **もし** 1 **ならば** nop.	111110********** 1100111100000zzzz	1	−	−	−	○

命令		動作	命令コード	実行ステート	DC ビット	適用命令 SH-1	SH-2	SH-DSP
DCF	PSTS MACL,Dz	**もし** DC＝0 **ならば** MACL→Dz **もし** 1 **ならば** nop.	111110********** 110111110000zzzz	1	−	−	−	○
	PSUB Sx,Sy,Dz	Sx－Sy→Dz	111110********** 10100001xxyyzzzz	1	**更新**	−	−	○
DCT	PSUB Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx-Sy→Dz **もし** 0 **ならば** nop.	111110********** 10100010xxyyzzzz	1	−	−	−	○
DCF	PSUB Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx-Sy→Dz **もし** 1 **ならば** nop.	111110********** 10100011xxyyzzzz	1	−	−	−	○
	PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx－Sy→Du Se **の上位ワード×**sf **の上位ワード→**Dg	111110********** 0110eeffxxyygguu	1	**更新**[*2]	−	−	○
	PSUBC Sx,Sy,Dz	Sx－Sy－DC→Dz	111110********** 10100000xxyyzzzz	1	**更新**	−	−	○
	PXOR Sx,Sy,Dz	Sx ^ Sy→Dz、Dz **の下位ワードクリア**	111110********** 10100101xxyyzzzz	1	**更新**	−	−	○
DCT	PXOR Sx,Sy,Dz	**もし** DC＝1 **ならば** Sx^Sy→Dz、Dz **の下位ワードクリア** **もし** 0 **ならば** nop.	111110********** 10100110xxyyzzzz	1	−	−	−	○
DCF	PXOR Sx,Sy,Dz	**もし** DC＝0 **ならば** Sx^Sy→Dz、Dz **の下位ワードクリア** **もし** 1 **ならば** nop.	111110********** 10100111xxyyzzzz	1	−	−	−	○

【注】 *1　PADD **の演算結果に基いて更新されます。**
*2　PSUB **の演算結果に基いて更新されます。**

B.　SH-DSP と SH-2、SH-1 との比較

・機能の比較

SH-DSP、SH-2、SH-1 の機能の比較を表 B.1 に示します。

表 B.1　SH-DSP、SH-2、SH-1 の機能比較

		SH-DSP	SH-2	SH-1
命令数	CPU 命令	65 種　182 命令	62 種　142 命令	56 種　133 命令
	DSP 命令	26 種　135 命令	―	―
追加された SH 命令		（SH-2 に対して） 3 種　40 命令 （表 A.2 参照）	（SH-1 に対して） 6 種　9 命令 （表 A.3 参照）	―
MAC.W の機能		16×16＋64→64 ビット パイプラインは 7 段 （IF、ID、EX、MA、MA、mm、mm）	16×16＋64→64 ビット パイプラインは 7 段 （IF、ID、EX、MA、MA、mm、mm）	16×16＋42→42 ビット パイプラインは 8 段 （IF、ID、EX、MA、MA、mm、mm、mm）
MULS.W MULU.W の機能		パイプラインは 6 段 （IF、ID、EX、MA、mm、mm）	パイプラインは 6 段 （IF、ID、EX、MA、mm、mm）	パイプラインは 7 段 （IF、ID、EX、MA、mm、mm、mm）
PMULS の機能		16×16→40 ビット パイプラインは 5 段 （IF、ID、EX、MA、WB/DSP）	―	―

ルネサス32 ビットRISC マイクロコンピュータ
ソフトウェアマニュアル
SH-1/SH-2/SH-DSP

発行年月日　1993 年 3 月　　第 1 版
　　2005 年 1 月 11 日　Rev.7.00
発　　行　株式会社ルネサス テクノロジ 営業企画統括部
　　〒100-0004 東京都千代田区大手町 2-6-2
編　　集　株式会社ルネサス小平セミコン 技術ドキュメント部

株式会社ルネサス テクノロジ　営業企画統括部　〒100-0004 東京都千代田区大手町2-6-2 日本ビル

営業お問合せ窓口
株式会社ルネサス販売

http://www.renesas.com

本社	〒100-0004	千代田区大手町2-6-2 (日本ビル)	(03) 5201-5350
京浜支社	〒212-0058	川崎市幸区鹿島田890-12 (新川崎三井ビル)	(044) 549-1662
西東京支社	〒190-0023	立川市柴崎町2-2-23 (第二高島ビル2F)	(042) 524-8701
札幌支店	〒060-0002	札幌市中央区北二条西4-1 (札幌三井ビル5F)	(011) 210-8717
東北支社	〒980-0013	仙台市青葉区花京院1-1-20 (花京院スクエア13F)	(022) 221-1351
いわき支店	〒970-8026	いわき市平小太郎町4-9 (損保ジャパンいわき第二ビル3F)	(0246) 22-3222
茨城支店	〒312-0034	ひたちなか市堀口832-2 (日立システムプラザ勝田1F)	(029) 271-9411
新潟支店	〒950-0087	新潟市東大通1-4-2 (新潟三井物産ビル3F)	(025) 241-4361
松本支社	〒390-0815	松本市深志1-2-11 (昭和ビル7F)	(0263) 33-6622
中部営業本部	〒460-0008	名古屋市中区栄3-13-20 (栄センタービル4F)	(052) 261-3000
浜松支店	〒430-7710	浜松市板屋町111-2（浜松アクトタワー10F）	(053) 451-2131
西部営業本部	〒541-0044	大阪市中央区伏見町4-1-1 (明治安田生命大阪御堂筋ビル)	(06) 6233-9500
北陸支社	〒920-0031	金沢市広岡3-1-1 (金沢パークビル8F)	(076) 233-5980
広島支店	〒730-0036	広島市中区袋町5-25 (広島袋町ビルディング8F)	(082) 244-2570
鳥取支店	〒680-0822	鳥取市今町2-251 (日本生命鳥取駅前ビル)	(0857) 21-1915
九州支社	〒812-0011	福岡市博多区博多駅前2-17-1 (ヒロカネビル本館5F)	(092) 481-7695
鹿児島支店	〒890-0053	鹿児島市中央町12-2 (明治安田生命鹿児島中央町ビル)	(099) 284-1748

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：カスタマサポートセンタ　E-Mail: csc@renesas.com

Colophon 3.0

SH-1/SH-2/SH-DSP
ソフトウェアマニュアル

ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668

RJJ09B0228-0700