致尊敬的顾客

关于产品目录等资料中的旧公司名称

NEC电子公司与株式会社瑞萨科技于2010年4月1日进行业务整合（合并），整合后的新公司暨“瑞萨电子公司”继承两家公司的所有业务。因此，本资料中虽还保留有旧公司名称等标识，但是并不妨碍本资料的有效性，敬请谅解。

瑞萨电子公司网址：http://www.renesas.com

2010年4月1日
瑞萨电子公司

【发行】瑞萨电子公司（http://www.renesas.com）

【业务咨询】http://www.renesas.com/inquiry

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

Software Manual

32

SH-1/SH-2/SH-DSP

瑞萨32位RISC单片机

SuperH™ RISC engine族

瑞萨电子

www.renesas.com

Rev.1.00 2009.01

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.

注意

本文只是参考译文，前页所载英文版“Cautions”具有正式效力。

关于利用本资料时的注意事项

1. 本资料是为了让用户根据用途选择合适的本公司产品的参考资料，对于本资料中所记载的技术信息，并非意味着对本公司或者第三者的知识产权及其他权利做出保证或对实施权力进行的承诺。
2. 对于因使用本资料所记载的产品数据、图、表、程序、算法及其他应用电路例而引起的损害或者对第三者的知识产权及其他权利造成侵犯，本公司不承担任何责任。
3. 不能将本资料所记载的产品和技术用于大规模破坏性武器的开发等目的、军事目的或其他的军需用途方面。另外，在出口时必须遵守日本的《外汇及外国贸易法》及其他出口的相关法令并履行这些法令中规定的必要手续。
4. 本资料所记载的产品数据、图、表、程序、算法以及其他应用电路例等所有信息均为本资料发行时的内容，本公司有可能在未做事先通知的情况下，对本资料所记载的产品或者产品规格进行更改。所以在购买和使用本公司的半导体产品之前，请事先向本公司的营业窗口确认最新的信息并经常留意本公司通过公司主页(http://www.renesas.com)等公开的最新信息。
5. 对于本资料中所记载的信息，制作时我们尽力保证出版时的精确性，但不承担因本资料的叙述不当而致使顾客遭受损失等的任何相关责任。
6. 在使用本资料所记载的产品数据、图、表等所示的技术内容、程序、算法及其他应用电路例时，不仅要对所使用的技术信息进行单独评价，还要对整个系统进行充分的评价。请顾客自行负责，进行是否适用的判断。本公司对于是否适用不负任何责任。
7. 本资料中所记载的产品并非针对万一出现故障或是错误运行就会威胁到人的生命或给人体带来危害的机器、系统(如各种安全装置或者运输交通用的、医疗、燃烧控制、航天器械、核能、海底中继用的机器和系统等)而设计和制造的，特别是对于品质和可靠性要求极高的机器和系统等（将本公司指定用于汽车方面的产品用于汽车时除外）。如果要用于上述的目的，请务必事先向本公司的营业窗口咨询。另外，对于用于上述目的而造成的损失等，本公司概不负责。
8. 除上述第7项内容外，不能将本资料中记载的产品用于以下用途。如果用于以下用途而造成的损失，本公司概不负责。
   1）生命维持装置。
   2）植埋于人体使用的装置。
   3）用于治疗（切除患部、给药等）的装置。
   4）其他直接影响到人的生命的装置。
9. 在使用本资料所记载的产品时，对于最大额定值、工作电源电压的范围、放热特性、安装条件及其他条件请在本公司规定的保证范围内使用。如果超出了本公司规定的保证范围使用时，对于由此而造成的故障和出现的事故，本公司将不承担任何责任。
10. 本公司一直致力于提高产品的质量和可靠性，但一般来说，半导体产品总会以一定的概率发生故障、或者由于使用条件不同而出现错误运行等。为了避免因本公司的产品发生故障或者错误运行而导致人身事故和火灾或造成社会性的损失，希望客户能自行负责进行冗余设计、采取延烧对策及进行防止错误运行等的安全设计（包括硬件和软件两方面的设计）以及老化处理等，这是作为机器和系统的出厂保证。特别是单片机的软件，由于单独进行验证很困难，所以要求在顾客制造的最终的机器及系统上进行安全检验工作。
11. 如果把本资料所记载的产品从其载体设备上卸下，有可能造成婴儿误吞的危险。顾客在将本公司产品安装到顾客的设备上时，请顾客自行负责将本公司产品设置为不容易剥落的安全设计。如果从顾客的设备上剥落而造成事故时，本公司将不承担任何责任。
12. 在未得到本公司的事先书面认可时，不可将本资料的一部分或者全部转载或者复制。
13. 如果需要了解关于本资料的详细内容，或者有其他关心的问题，请向本公司的营业窗口咨询。

产品使用时的注意事项

本文对适用于单片机所有产品的 “使用时的注意事项”进行说明。有关个别的使用时的注意事项请参照正文。此外，如果在记载上有与本手册的正文有差异之处，请以正文为准。

1. 未使用的引脚的处理
   【注意】将未使用的引脚按照正文的 “未使用引脚的处理”进行处理。
   CMOS产品的输入引脚的阻抗一般为高阻抗。如果在开路的状态下运行未使用的引脚，由于感应现象，外加LSI周围的噪声，在LSI内部产生穿透电流，有可能被误认为是输入信号而引起误动作。
   未使用的引脚，请按照正文的 “未使用引脚的处理”中的指示进行处理。

2. 通电时的处理
   【注意】通电时产品处于不定状态。
   通电时， LSI内部电路处于不确定状态，寄存器的设定和各引脚的状态不定。通过外部复位引脚对产品进行复位时，从通电到复位有效之前的期间，不能保证引脚的状态。
   同样，使用内部上电复位功能对产品进行复位时，从通电到达到复位产生的一定电压的期间，不能保证引脚的状态。

3. 禁止存取保留地址 （保留区）
   【注意】禁止存取保留地址 （保留区）
   在地址区域中，有被分配将来用作功能扩展的保留地址 （保留区）。因为无法保证存取这些地址时的运行，所以不能对保留地址 （保留区）进行存取。

4. 关于时钟
   【注意】复位时，请在时钟稳定后解除复位。
   在程序运行中切换时钟时，请在要切换成的时钟稳定之后进行。复位时，在通过使用外部振荡器（或者外部振荡电路）的时钟开始运行的系统中，必须在时钟充分稳定后解除复位。另外，在程序运行中，切换成使用外部振荡器 （或者外部振荡电路）的时钟时，在要切换成的时钟充分稳定后再进行切换。

5. 关于产品间的差异
   【注意】在变更不同型号的产品时，请对每一个产品型号进行系统评价测试。
   即使是同一个群的单片机，如果产品型号不同，由于内部ROM、版本模式等不同，在电特性范围内有时特性值、动作容限、噪声耐量、噪声辐射量等也不同。因此，在变更不认同型号的产品时，请对每一个型号的产品进行系统评价测试。

前言

SuperH RISC engine 族 CPU 是 RISC （Reduced Instruction Set Computer）结构的 CPU。基本指令以 1 条指令 1 个状态运行，实现了高性能的运算处理。并且内置乘法器，可进行与通用 DSP （Digital Signal Processor：数字信号处理器）相同的乘法和乘法累加运算。

瑞萨 SuperH RISC engine （以下简称 SuperH）族有 SH-1、 SH-2、 SH-3、 SH-DSP、 SH3-DSP、 SH-4 等 CPU 内核。

SH-1、 SH-2、 SH-3、 SH-4 的各 CPU 有 2 进制级高位兼容的指令体系。

SH-DSP 以 SH-2 CPU 为基础，是实现了与通用 DSP 相同信号处理性能的 32 位微控制器。 SH-DSP 强化了 SuperH RISC engine 中乘法和乘法累加运算的 DSP 功能，可实现 DSP 类的数据总线功能。 SH-DSP 与 SH-1、 SH-2 单片机在 2 进制级高位兼容 （参考**图 1**）。

该编程手册记载了 SH-1、SH-2、SH-DSP 的基本体系结构和指令详细内容，便于理解体系结构和指令的操作，同时也记载了 SuperH RISC engine 的特点——流水线运行。

有关 SH-3、 SH3-DSP、 SH-4，详细内容参照其他编程手册。

有关硬件的详细内容请参照各产品的硬件手册。

有关开发环境系统，请咨询本公司营业部。

SuperH 概略

SuperH RISC engine 族的指令形态如**图 1** 所示。

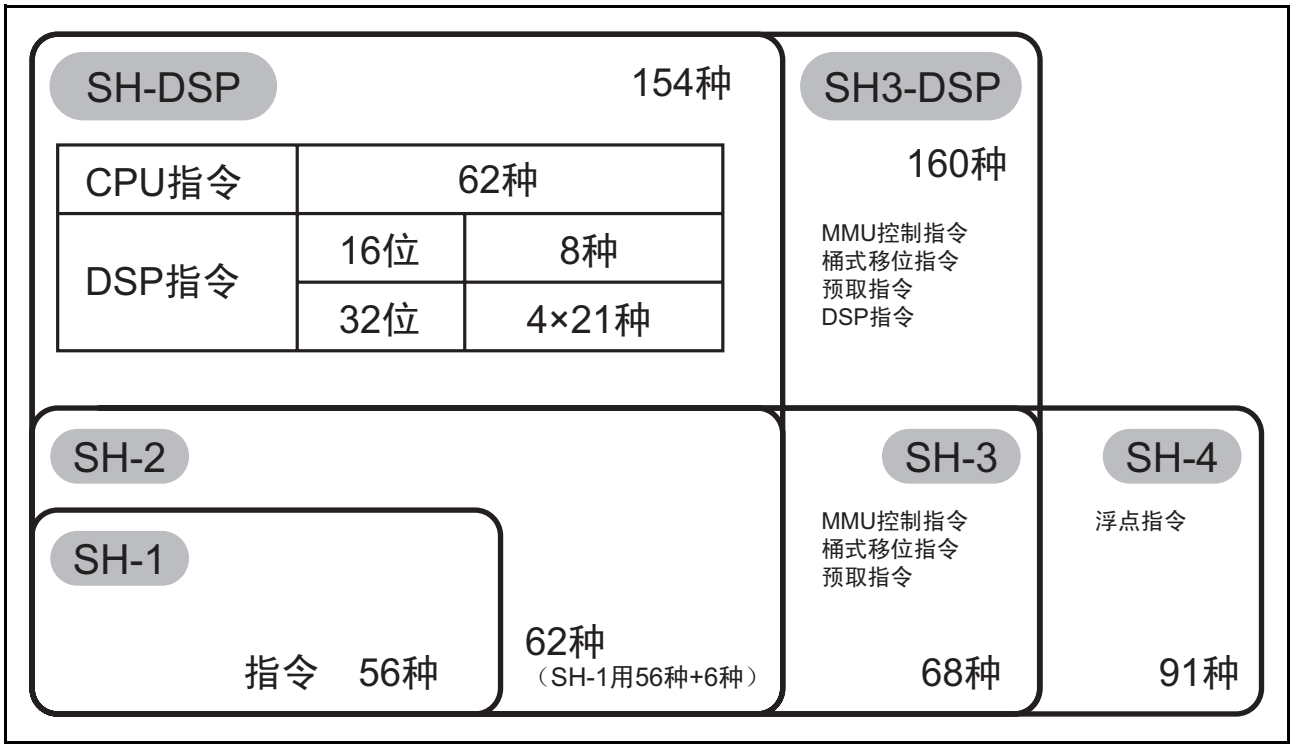


图 1　SuperH RISC engine 族的指令形态

手册的结构

该手册的结构如**表 1** 所示。

表 1　手册的结构

分类	各章标题	内容
概要	1.　特点	SuperH 的 CPU 和 DSP 的特点
体系结构	2.　寄存器结构	通用寄存器、控制寄存器、系统寄存器的种类和结构、 DSP 寄存器的种类和结构
	3.　数据格式	寄存器和存储器上的数据格式、 DSP 的数据格式
指令概要	4.　指令特点	CPU 指令和 DSP 指令的特点 • 寻址方式 • 指令格式 • 运算功能和数据传送
	5.　指令系统	以分类的顺序说明指令概要、各指令的操作
指令详细内容	6.　各指令说明	格式、操作概略、操作码、说明、注意、操作内容、使用示例
	7.　流水线运行	流水线的流程、各指令的操作和流水线的流程
附录	SuperH 系列的比较	SH-1、 SH-2、 SH-DSP 的比较

目　录

第 1 章 特点

1.1 SH-1、SH-2 的特点

SH-1 CPU 及 SH-2 CPU 具有 RISC 结构的指令系统，因为基本指令以 1 条指令 1 个状态 （1 个系统时钟周期）运行，所以可迅速提高指令执行速度。并且采用内部 32 位结构，强化了数据处理能力。

SH-1 CPU 及 SH-2 CPU 的特点如**表 1.1** 所示。

表 1.1 SH-1 CPU 及 SH-2 CPU 的特点

项目	特点
体系结构	• 瑞萨科技独创体系结构 • 内部 32 位结构
通用寄存器机	• 通用寄存器 32 位 ×16 个 • 控制寄存器 32 位 ×3 个 • 系统寄存器 32 位 ×4 个
指令系统	• RISC 结构的指令系统 — 指令长度固定为 16 位，可提高编码效率 — 加载存储结构 （在寄存器之间执行基本运算） — 采用延迟转移方式，减轻了转移时的流水线混乱 — 基于 C 语言的指令系统
指令执行时间	• 基本指令为 1 条指令 / 1 个状态 （1 条指令 / 1 个系统时钟周期）
地址空间	• 在体系结构为 4GB
内置乘法器 （SH-1 CPU）	• 内置乘法器，能在 1 ～ 3* 个状态执行 16×16→32 的乘法运算、 在 3/(2)* 个状态执行 16×16+42→42 的乘法累加运算
内置乘法器 （SH-2 CPU）	• 内置乘法器，能在 1 ～ 3 个状态执行 16×16→32 的乘法运算、 在 3/(2)* 个状态执行 16×16+64→64 的乘法累加运算、 在 2 ～ 4* 个状态执行 32×32→64 的乘法运算、 在 3/(2 ～ 4)* 个状态执行 32×32+64→64 的乘法累加运算
流水线	• 5 段流水线方式
处理状态	• 程序执行状态 • 异常处理状态 • 总线权释放状态 • 复位状态 • 低功耗状态
低功耗状态	• 睡眠模式 • 待机模式

【注】 * 表示普通执行状态。（ ）内的值表示因与前后指令竞争产生的执行状态。

1.2 SH-DSP 的特点

SH-DSP 以 SH-2 CPU 为基础，是实现了与通用 DSP 相同信号处理性能的 32 位微控制器。 SuperH 中有乘法运算与乘法累加运算的 DSP 类指令。 SH-DSP 是强化了 SuperH 的 DSP 功能的产物，可实现 DSP 类指令的所有数据总线功能。 SH-DSP 可与 SH-1、 SH-2 CPU 在目标代码级高位兼容。

SH-1、 SH-2 仅具有 16 位长度的指令， SH-DSP 中的指令长度基本也为 16 位，为了并发处理 DSP 类指令，可追加 32 位长度的 DSP 类指令。 SuperH 为标准的诺伊曼型体系结构，但 SH-DSP 具有扩展哈佛型体系结构的 DSP 数据总线。

SH-DSP 中追加的特点如**表 1.2** 所示。

表 1.2 SH-DSP 中追加的特点

项目	特点
DSP 单元	• 1 个周期的乘法器 • 16 位 ×16 位 →32 位 （带符号的定点） • 算术运算器 （ALU：Arithmetic Logic Unit） • 桶式移位器 • DSP 寄存器 • MSB 检测
DSP 寄存器	• 40 位数据寄存器 ×2 个 • 32 位数据寄存器 ×6 个 • DSP 状态寄存器 （DSR） • 在控制寄存器追加了模寄存器 （MOD、 32 位） • 在状态寄存器 （SR）追加了重复计数器 （RC）、模寻址指定 （DMX、DMY）及重复标志 （RF1、 RF0） • 在控制寄存器追加了重复起始寄存器 （RS、 32 位）、重复结束寄存器 （RE、32 位）
DSP 数据总线	• 扩展哈佛型体系结构 • 同时存取 2 条数据总线及 1 条指令总线
并发处理	• 最多可执行 4 个并发处理 • ALU 运算、乘法运算及 2 个加载或存储
地址运算器	• 2 个地址运算器 • 用于存取 2 个存储器的地址运算
DSP 数据寻址方式	• 递增、递减及变址 • 有模寻址或无模寻址
重复控制	• 零开销重复 （循环）控制
指令系统	• 16 位长度或 32 位长度 — 16位长度 （仅限加载或存储时） — 32位长度 （包含ALU运算及乘法运算时） • 追加存取 DSP 寄存器的 SH 单片机指令
流水线	• 5 段流水线方式 • 最后的第 5 个阶段可复用作 WB 阶段与 DSP 阶段

第 2 章 寄存器结构

SH-1、 SH-2 有通用寄存器 （32 位 ×16 个）、控制寄存器 （32 位 ×3 个）及系统寄存器 （32 位 ×4 个）。

SH-DSP 与 SH-1、 SH-2 在目标代码级高位兼容。因此， SH-DSP 除具有与 SH-1、 SH-2 相同的寄存器之外，还追加了几个寄存器。追加的寄存器有：控制寄存器的重复起始寄存器 （RS）、重复结束寄存器 （RE）及模寄存器 （MOD）共 3 个；系统寄存器的 DSP 状态寄存器 （DSR）、 8 个 DSP 数据寄存器即 A0、 A1、 X0、 X1、 Y0、 Y1、 M0、 M1。

为 CPU 类指令时，SH-DSP 的通用寄存器与 SH-1、SH-2 使用方法相同；为 DSP 类指令时， SH-DSP 的通用寄存器用作对存储器存取的地址寄存器、变址寄存器。

2.1 通用寄存器

SH-1、SH-2、SH-DSP 的通用寄存器（Rn）为 32 位长度，从 R0 到 R15 共有 16 个。通用寄存器用于数据处理和地址计算， R0 也可用作变址寄存器。几个指令可使用的寄存器固定为 R0。R15 可用作硬件堆栈指针（SP）。通过使用 R15 参考堆栈，执行异常处理时状态寄存器 （SR）和程序计数器 （PC）的压栈和出栈。

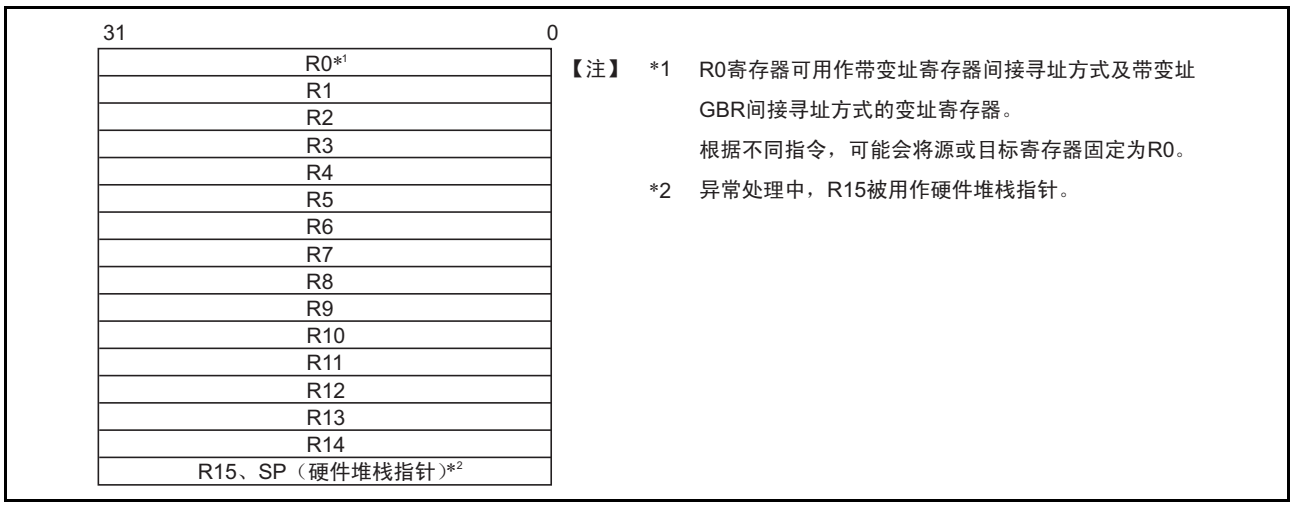


图 2.1 通用寄存器结构 （SH-1/SH-2）

在 SH-DSP，通过 DSP 类指令，将 16 个通用寄存器中的 8 个寄存器用于 X、 Y 数据存储器及使用主总线的数据存储器 （单数据）的寻址。

为了存取 X 存储器， R4、 R5 用作 X 地址寄存器［Ax］， R8 用作 X 变址寄存器［Ix］； 为了存取 Y 存储器， R6、 R7 用作 Y 地址寄存器 ［Ay］， R9 用作 Y 变址寄存器 ［Iy］； 为了使用主总线存取单数据， R2、 R3、 R4、 R5 用作单数据地址寄存器 ［As］， R8 用作单数据变址寄存器 ［Is］。

DSP 类指令可同时存取 X 和 Y 数据存储器。为了指定 X 和 Y 数据存储器的地址，有 2 组地址指针。

SH-DSP 的通用寄存器结构如图 2.2 所示。

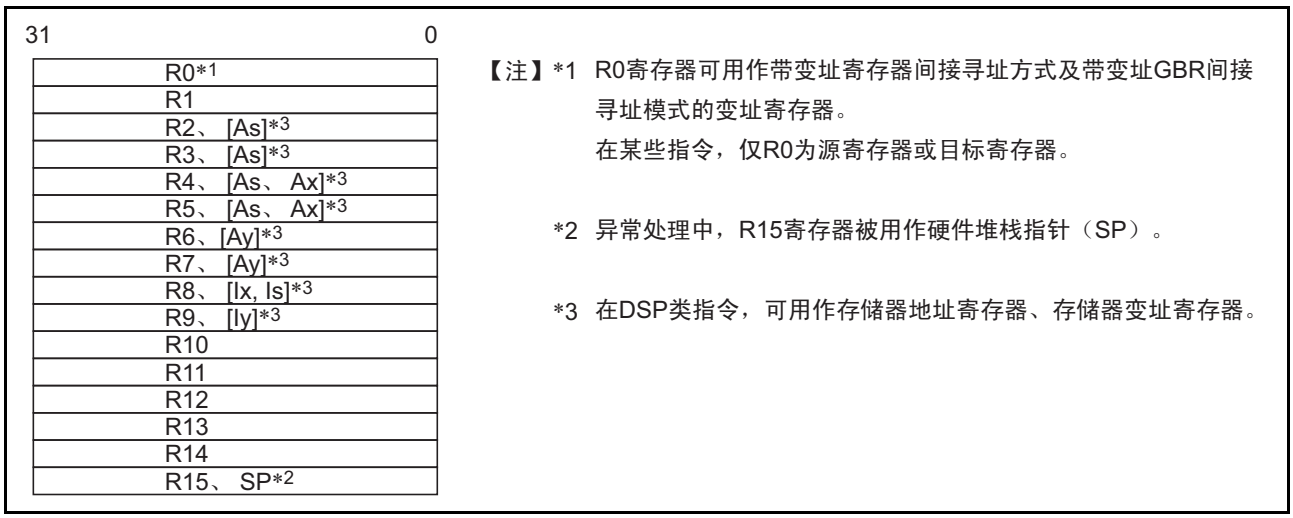


图 2.2 通用寄存器结构 （SH-DSP）

在汇编程序中，使用 R2、 R3、．．．、 R9 的符号名称。如果要使用表明 DSP 类指令寄存器作用的名称，则使用寄存器的别称 （alias），在汇编程序如下表示：

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

Ix 为 R8 的别称。其他寄存器的别称命名如下：

```
Ax0: .REG  (R4)
Ax1: .REG  (R5)
Ix:   .REG  (R8)
Ay0: .REG  (R6)
Ay1: .REG  (R7)
Iy:   .REG  (R9)
As0: .REG  (R4); 这是为了传送单数据而需要别称时的定义。
As1: .REG  (R5); 这是为了传送单数据而需要别称时的定义。
As2: .REG  (R2); 这是为了传送单数据而需要别称时的定义。
As3: .REG  (R3); 这是为了传送单数据而需要别称时的定义。
Is:   .REG  (R8); 这是为了传送单数据而需要别称时的定义。
```

2.2 控制寄存器

控制寄存器为 32 位长度，有状态寄存器 （SR:Status register）、全局基址寄存器 （GBR: Global base register）及向量基址寄存器 （VBR: Vector base register）共 3 个。

SR 寄存器表示各种指令的处理状态。

GBR 寄存器作为 GBR 间接寻址方式的基址，用于内部外围模块寄存器的数据传送等。GBR 间接寻址方式可用于内部外围模块寄存器区域的数据传送及逻辑运算。

VBR 寄存器用作包含中断的异常处理向量区域的基址。

SH-1、 SH-2 的控制寄存器如图 2.3 所示。

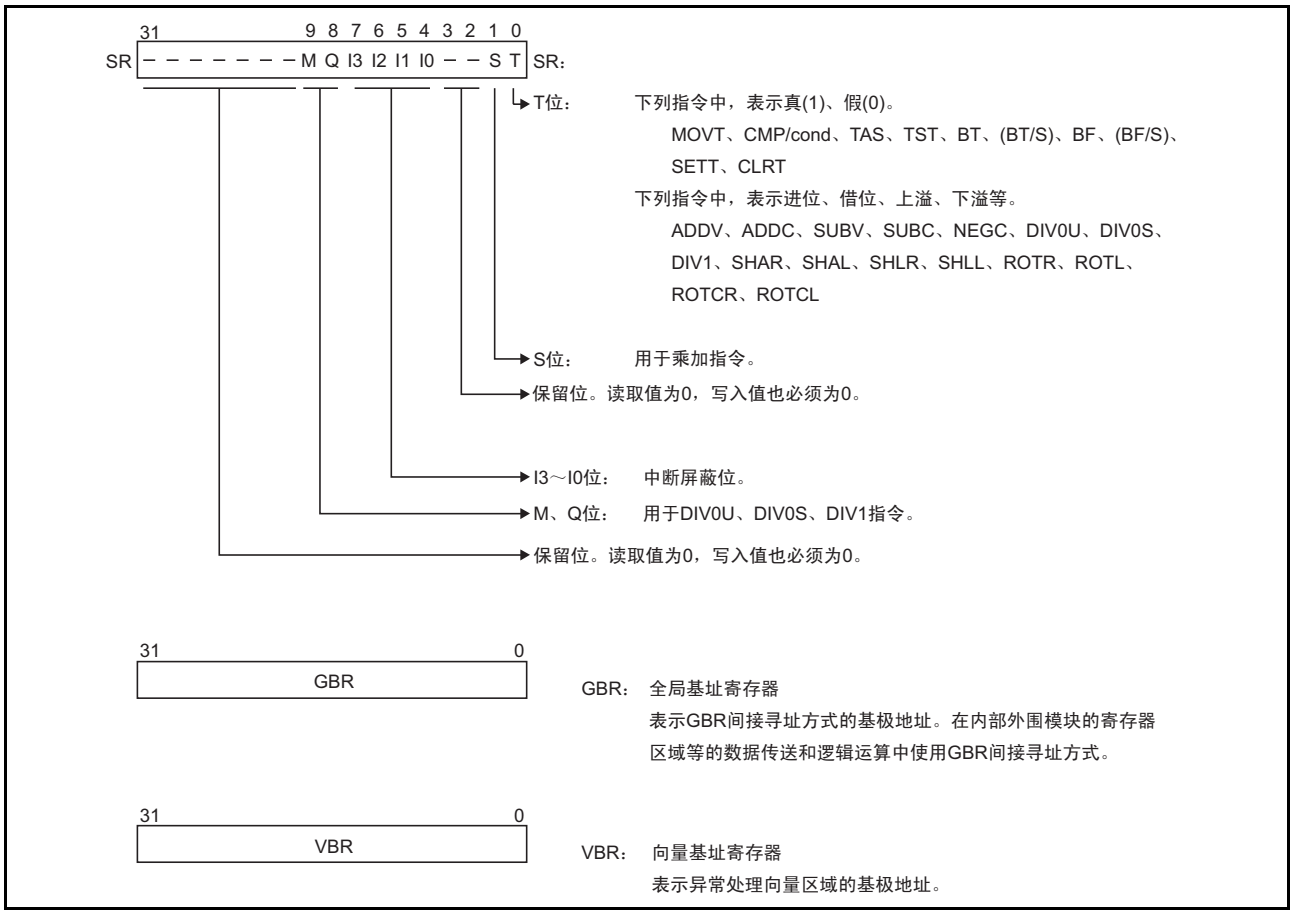


图 2.3 控制寄存器结构 （SH-1/SH-2）

SH-DSP 追加了重复起始寄存器 （RS: Repeat start register）、重复结束寄存器 （RE: Repeat end register）及模寄存器 （MOD: Modulo register）共 3 个。因此，SR 寄存器也新追加了 4 个控制位 （DMX 位、 DMY 位、 RF1 位、 RF0 位）和 12 位的 RC 计数器。

RS 寄存器和 RE 寄存器用于控制程序的重复 （循环）。在 SR 寄存器的重复计数器 （RC: Repeat counter）指定重复次数，在 RS 寄存器指定重复起始地址，在 RE 寄存器指定重复结束地址。但保存于 RS 寄存器和 RE 寄存器的地址值不一定与重复的物理起始地址、结束地址相同。

MOD 寄存器用于重复数据缓冲的模寻址。由 DMX 位或 DMY 位指定模寻址，在 MOD 寄存器的高 16 位指定模结束地址 （ME），在低 16 位指定模起始地址 （MS）。 DMX 和 DMY 位不可同时指定模寻址。为 X、Y 数据传送指令 （MOVX、 MOVY）时，可执行模寻址；为单数据传送指令 （MOVS）时，不可执行模寻址。

SH-DSP 中追加的控制寄存器和状态寄存器的位结构如**图 2.4** 所示。 SR 寄存器的位如**表 2.1** 所示。

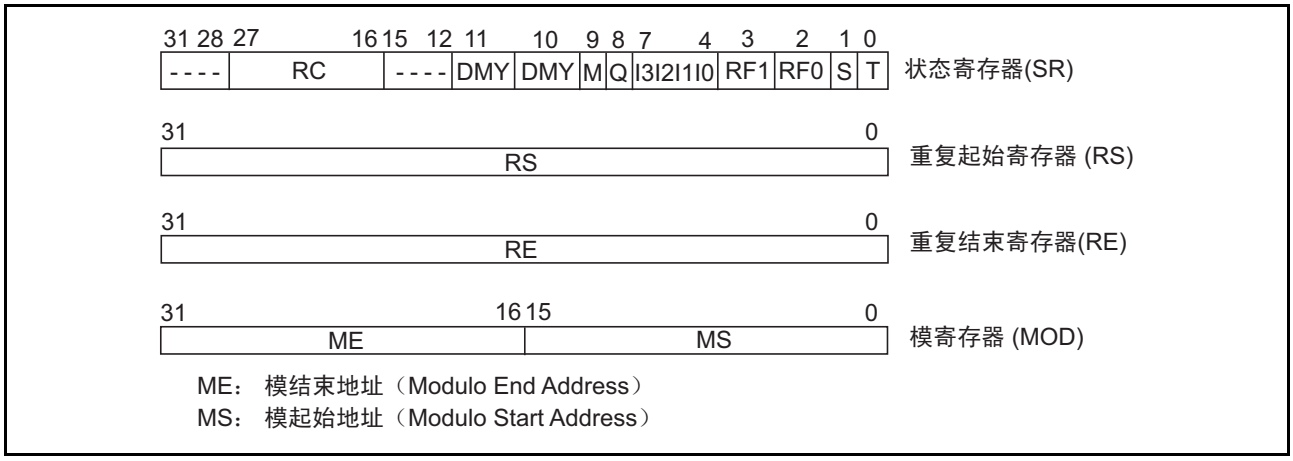


图 2.4 控制寄存器结构 （SH-DSP）

表 2.1 SR 寄存器的位

位	名称 （简称）	功能
27 ～ 16	重复计数器 （RC） *	指定重复 （循环）控制的重复次数 （2 ～ 4095）。
11	指定用于 Y 指针的模寻址 （DMY） *	1：对于 Y 存储器地址指针、 Ay （R6、 R7），模寻址方式有效。
10	指定用于 X 指针的模寻址 （DMX） *	1：对于 X 存储器地址指针、 Ax （R4、 R5），模寻址方式有效。
9	M 位	用于 DIV0S/U、 DIV1 指令。
8	Q 位	
7 ～ 4	中断请求屏蔽 （I3 ～ I0）	表示接受中断请求的级别 （0 ～ 15）。
3 ～ 2	重复标志 （RF1、 0） *	用于零开销重复 （循环）控制。 00：1 步重复 01：2 步重复 11：3 步重复 10：4 步及以上的重复
1	饱和运算位 （S）	用于 MAC 指令及 DSP 指令。 1：指定饱和运算 （防止上溢）。
0	T 位	MOVT、 CMP/cond、 TAS、 TST、 BT、 BF、 SETT、 CLRT 及 DT 指令时 0：表示假。 1：表示真。
		ADDV/C、 SUBV/C、 DIV0U/S、 DIV1、 NEGC、 SHAR/L、 SHLR/L、 ROTR/L 及 ROTCR/L 指令时 1：表示产生进位、借位、上溢或下溢。
31 ～ 28 15 ～ 12	保留位	0：读取值总是为 0， 写入值也必须为 0。

【注】 * 仅适用于 SH-DSP

具有存取 RS、 RE、 MOD 寄存器专用的加载 / 存储指令。例如，存取 RS 寄存器时如下：

```
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)
```

为了执行零开销重复控制，在 RS、 RE 寄存器设定地址的指令如下：

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

2.3 系统寄存器

系统寄存器为 32 位长度，有乘加寄存器 （MACH: Multiply and accumulate register high、 MACL: Multiply and accumulate register low）、过程寄存器 （PR: Procedure register）及程序计数器 （PC: Program counter）共 4 个。 MACH、MACL 寄存器保存乘法或乘法累加运算的结果 * ； PR 寄存器保存从子程序过程返回的目标地址； PC 计数器表示执行中的程序地址，控制处理流程； PC 计数器表示当前执行指令的 4 字节后的地址。

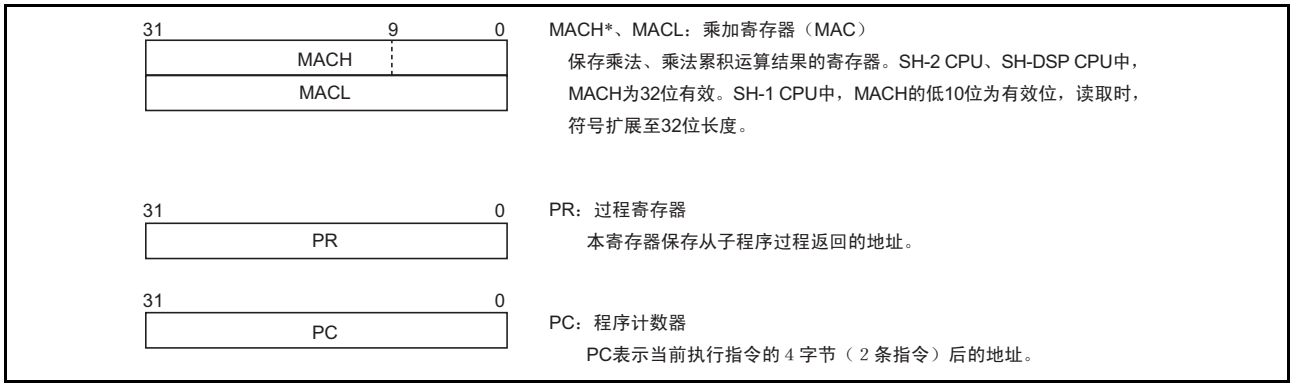


图 2.5 系统寄存器结构

【注】 * SH-DSP 中，仅在执行 SH-1、 SH-2 支持的指令时，使用该寄存器；
执行 SH-DSP 新追加的乘法指令 （PMULS）时，不使用该寄存器。

SH-DSP 中，还有 6 个寄存器用作系统寄存器，即：“2.4 DSP 寄存器”中描述的 DSP 单元寄存器 （DSP 寄存器）的 DSP 状态寄存器 （DSR）及 8 个数据寄存器中的 5 个 （A0、 X0、 X1、 Y0、 Y1）寄存器。其中，A0 寄存器为 40 位寄存器，从 A0 寄存器输出数据时，忽略保护位 （A0G）； 向 A0 寄存器输入数据时，数据的 MSB 被复制到保护位 （A0G）。

2.4 DSP 寄存器

SH-DSP 的 DSP 单元有作为 DSP 寄存器的 8 个数据寄存器和 1 个控制寄存器。

DSP 数据寄存器有 2 个 40 位长度的寄存器（A0、A1）和 6 个 32 位长度的寄存器（M0、M1、X0、X1、Y0、Y1）。A0、A1 寄存器中分别有 8 位保护位，即 A0G 与 A1G。

DSP 数据寄存器作为 DSP 指令的操作数，用于传送和处理 DSP 数据。存取 DSP 数据寄存器的指令有 3 种类型，即：DSP 数据处理及 X、Y 数据传送处理。

控制寄存器为 32 位长度的 DSP 状态寄存器（DSR:DSP Status Register），表示运算结果。DSR 寄存器中有表示运算结果的位、带符号的“大于”位（GT:signed Greater Than）、零位（Z:Zero value）、负值位（N:Negative value）、上溢位（V:overflow）、DSP 状态位（DC:DSP Condition）和控制 DC 位设定的状态选择位（CS:Condition Select）。

DC 位表示一种状态标志，与 SuperH CPU 内核的 T 位非常相似。对于带条件的 DSP 类指令，DSP 数据处理的执行受 DC 位控制。此控制仅与 DSP 单元的执行有关，仅更新 DSP 寄存器，与地址计算、加载 / 存储指令等 SuperH 单片机的 CPU 内核执行指令无关。控制位 CS（bit2 ～ 0）指定 DC 位的设定状态。

DSP 类指令包含无条件的 DSP 类指令和带条件的 DSP 类指令。无条件的 DSP 类的数据处理，除 PMULS、MOVX、MOVY、MOVS 指令之外，更新状态位和 DC 位。带条件的 DSP 类指令可根据 DC 位的状态执行，但不管执行与否都不更新 DSR 寄存器。

A0、X0、X1、Y0、Y1 等 5 个寄存器也用作系统寄存器。

DSP 寄存器如**图 2.6** 所示。DSR 寄存器位的功能如**表 2.2** 所示。

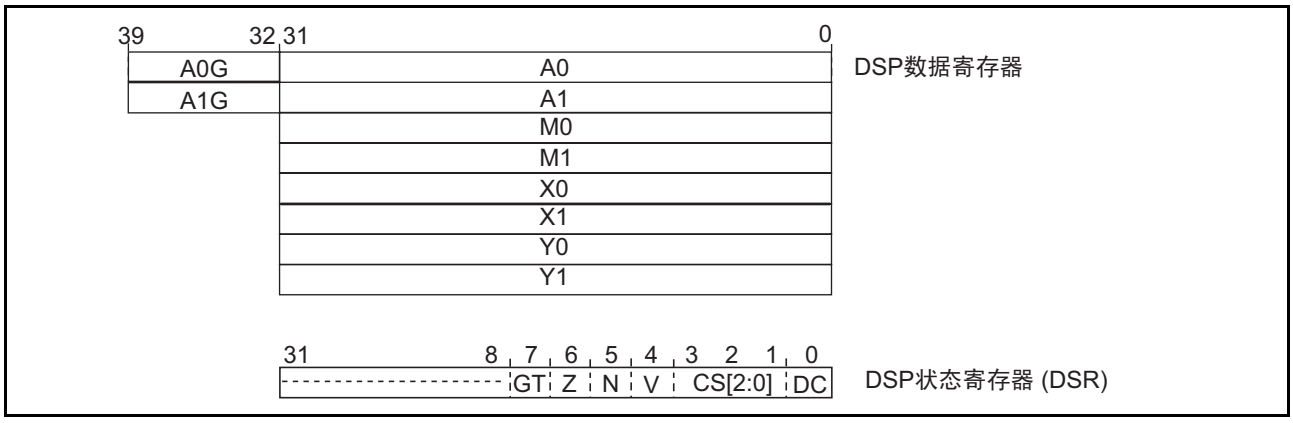


图 2.6 DSP 寄存器结构

表 2.2 DSR 寄存器的位

位	名称 （简称）	功能
31 ～ 8	保留位	0：读取值总是为 0， 写入值也必须为 0。
7	带符号的 “大于” 位 （GT）	表示运算结果为正 （零除外）或操作数 1 大于操作数 2。 1：运算结果为正或操作数 1 大于操作数 2
6	零位 （Z）	表示运算结果为零 （0）或操作数 1 等于操作数 2。 1：运算结果为零 （0）或相等
5	负值位 （N）	表示运算结果为负或操作数 1 小于操作数 2。 1：运算结果为负或操作数 1 小于操作数 2
4	上溢位 （V）	表示运算结果上溢。 1：运算结果上溢
3 ～ 1	状态选择位 （CS）	选择反映 DC 位的运算结果状态的模式。 不得指定 110、 111。 000：进位 / 借位模式 001：负值模式 010：零值模式 011：上溢模式 100：带符号的 “大于” 模式 101：带符号的 “大于等于” 模式
0	DSP 状态位 （DC）	显示 CS 位指定模式的运算结果状态。 0：指定的模式状态不成立 （不成立） 1：指定的模式状态成立

为 CPU 内核指令时， A0、 X0、 X1、 Y0、 Y1、 DSR 寄存器用作系统寄存器。

2.5 保护位和上溢处理的相关注意事项

DSP 单元的数据运算基本为 32 位运算，但常以 40 位（包含 8 位保护位）执行运算。保护位与 32 位的 MSB 值不匹配时，将运算结果作为上溢处理。此时， N 位与是否上溢无关，表示运算结果的正确状态。目标操作数为 32 位的寄存器也同样如此。总是假设为 8 位保护位，更新各状态标志。

即使使用保护位，产生不可正确显示结果的上溢时， N 标志也不可显示正确状态。详细内容参照 “4.8 ALU 定点运算 (3) DC 位”。

2.6 寄存器的初始值

复位后，寄存器的值如表 2.3 所示。

表 2.3 寄存器的初始值

分类	寄存器	初始值
通用寄存器	R0 ～ R14	不定
	R15 (SP)	向量地址表中 SP 的值
控制寄存器	SR	• I3 ～ I0 为 1111(H'F)，保留位为 0，其他不定 • RC、 DMY、 DMX、 RF1、 RF0 为 0（SH-DSP 追加位） SH-1、 SH-2、 SH-DSP 均为 H'0000 00F0
	RS RE	不定
	GBR	不定
	VBR	H'0000 0000
	MOD	不定
系统寄存器	MACH、 MACL、 PR	不定
	PC	向量地址表中 PC 的值
DSP 寄存器	A0、A0G、A1、A1G、M0、 M1、 X0、 X1、Y0、 Y1	不定
	DSR	H'0000 0000

第 3 章 数据格式

3.1 寄存器的数据格式

寄存器操作数的数据长度总是为长字 （32 位）。将存储器中的数据加载到寄存器时，如果存储器操作数的数据长度为字节 （8 位）或字 （16 位），则符号扩展为长字，并保存至寄存器。

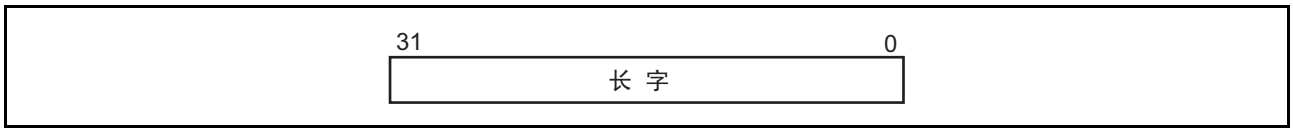


图 3.1 寄存器的数据格式

3.2 存储器的数据格式

有字节、字、长字的数据格式。

字节数据必须配置到任意地址，字数据必须从地址 2n 配置，长字数据必须从地址 4n 配置。如果从其边界以外存取，则产生地址错误。此时，不保证存取的结果。特别是，在硬件堆栈指针 （SP、 R15）所指的堆栈区以长字保存程序计数器 （PC）及状态寄存器 （SR），因此，硬件堆栈指针的值必须设定为 4n。有关地址错误，参照各产品的硬件手册。

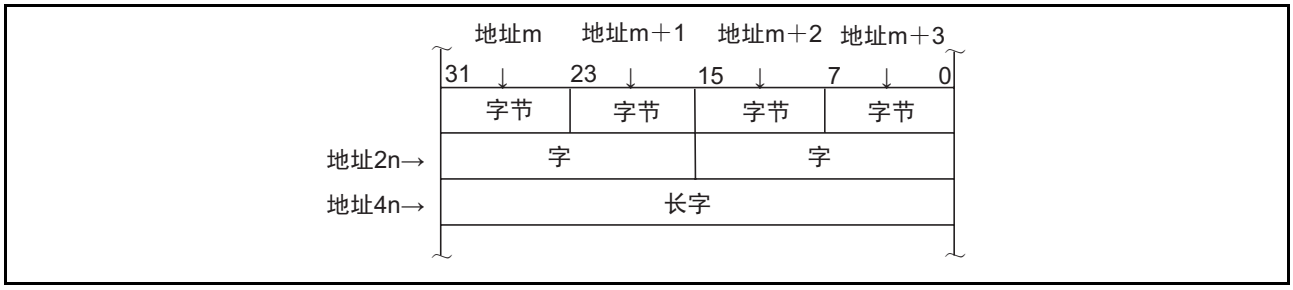


图 3.2 存储器的数据格式 （大端法）

产品内置小端法功能时，如下配置字节数据。各产品是否支持小端法，参照各产品的硬件手册。

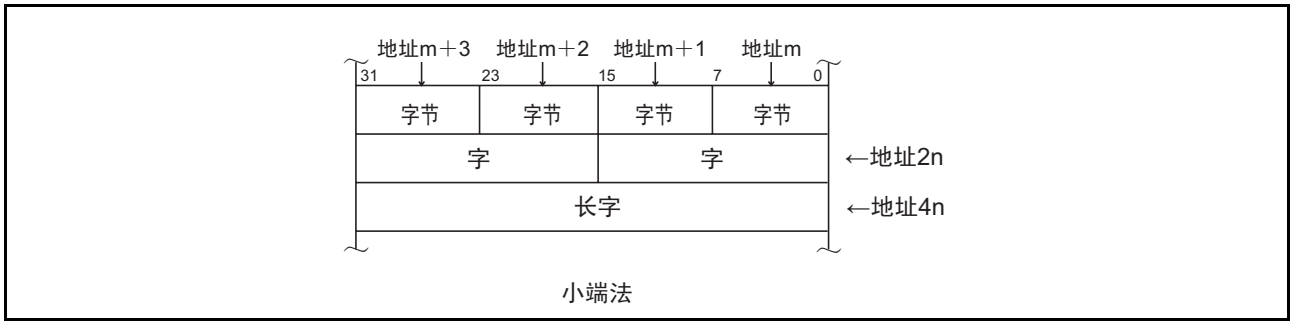


图 3.3 存储器的数据格式 （小端法）

3.3 立即数的数据格式

在操作码中配置字节的立即数。

在 MOV、 ADD、 CMP/EQ 指令，符号扩展立即数后，与寄存器以长字运算。在 TST、 AND、 OR、 XOR 指令，零扩展立即数后，以长字运算。因此，如果在 AND 指令使用立即数，则总是清除目标寄存器的高 24 位。

不在操作码中配置字及长字立即数，必须在存储器表中配置。以使用带位移量的 PC 相对寻址方式的立即数的数据传送指令 (MOV) 参照存储器表。

具体例子参照 “**第 4 章 指令特点**”的 “**4.1 8. 立即数**”。

3.4 DSP 类数据格式

SH-DSP 具有与 DSP 指令对应的 3 种不同数据格式：定点数据格式、整数数据格式及逻辑数据格式。

DSP 类的定点数据格式在 bit31 与 bit30 之间有 2 进制小数点，有带保护位、无保护位、乘法输入 3 种，各自的有效位长度与可显示值的范围不同。

DSP 类的整数数据格式在 bit16 与 bit15 之间有 2 进制小数点，有带保护位、无保护位、移位量 3 种，各自的有效位长度与可显示值的范围不同。

算术移位 （PSHA）的移位量为 7 位区域，可显示 –64 ～ +63 的值，但实际有效的移位量只有 –32 ～ +32 的值。同样，逻辑移位的移位量为 6 位区域，但实际有效的移位量只有 –16 ～ +16 的值。

DSP 类的逻辑数据格式无小数点。

数据格式及数据的有效长度由指令及 DSP 寄存器决定。

3 种 DSP 类的数据格式及其 2 进制小数点的位置，以及可参考的 CPU 类数据格式如**图 3.4** 所示。

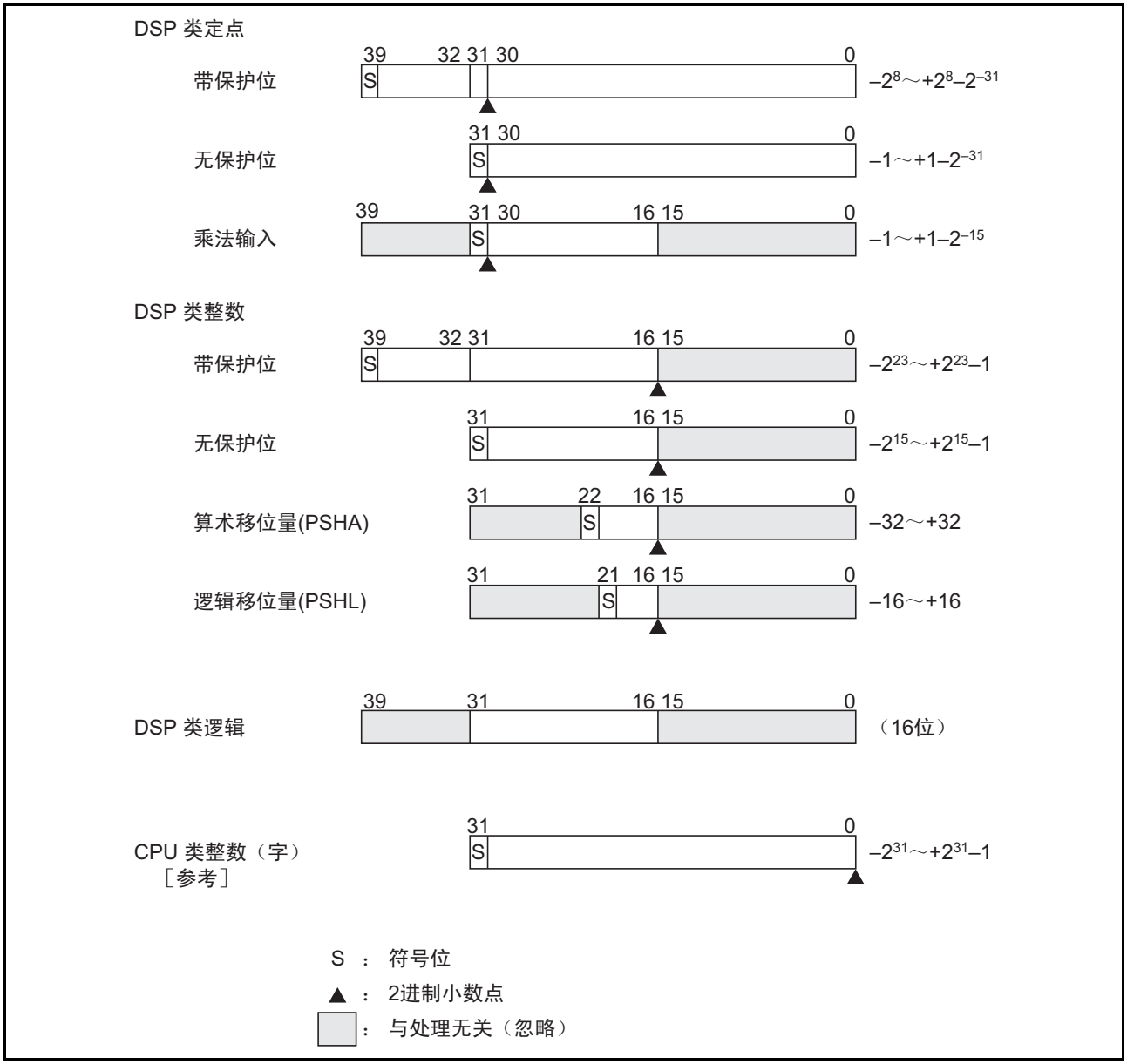


图 3.4 DSP 类数据格式

3.5 DSP 类指令与数据格式

DSP 数据格式与数据的有效长度由 DSP 类指令及 DSP 寄存器决定。存取 DSP 数据寄存器的指令有 DSP 数据处理、 X、 Y 数据传送处理、单数据传送处理 3 种。

(1) DSP 数据处理

DSP 定点数据处理中， A0、 A1 寄存器用作源寄存器时，保护位 （bit39 ～ 32）有效。除 A0、 A1 之外的寄存器（M0、 M1、 X0、 X1、 Y0、 Y1 寄存器）用作源寄存器时，符号扩展该寄存器数据后的数据为 bit39 ～ 32 的数据。 A0、 A1 寄存器用作目标寄存器时，保护位 （bit39 ～ 32）有效。除 A0、 A1 之外的寄存器用作目标寄存器时，忽略结果数据的 bit39 ～ 32。

DSP 整数数据处理与 DSP 定点数据处理相同。只是忽略源寄存器的低位字 （低 16 位、 bit15 ～ 0）。目标寄存器的低位字清 0。

DSP 逻辑数据处理的源寄存器的高位字 （高 16 位、 bit31 ～ 16）有效。忽略低位字与 A0、 A1 寄存器的保护位。目标寄存器的高位字有效。低位字与 A0、 A1 寄存器的保护位清 0。

(2) X、 Y 数据传送

MOVX.W、MOVY.W 指令通过 16 位的 X、 Y 数据总线存取 X、 Y 存储器。加载到寄存器的数据、从寄存器存储的数据，总是为高位字 （高 16 位、 bit31 ～ 16）。寄存器的低位字清 0。

(3) 单数据传送

MOVS.W、 MOVS.L 指令通过指令数据总线 （主总线）可存取任何存储器。所有 DSP 寄存器均与主总线连接，数据传送时为源寄存器或目标寄存器。数据传送有字及长字 2 种模式。在字模式，加载到 DSP 寄存器（除 A0G、 A1G 寄存器之外）的高位字，或从高位字存储；在长字模式，加载到 DSP 寄存器 （除 A0G、 A1G 寄存器之外）的 32 位，或从 32 位存储。

单数据传送过程中可将 A0G、 A1G 寄存器作为独立寄存器。在 A0G、 A1G 寄存器加载、存储的数据长度为 8 位。 A0G、 A1G 寄存器为源寄存器时，从寄存器仅存储数据的 8 位，并符号扩展高位。 A0G、 A1G 寄存器为目标寄存器时，在寄存器加载数据的低 8 位， A0、 A1 寄存器不清 0，保持在此之前的值。

DSP 类指令在寄存器的数据格式如**表 3.1**、**表 3.2** 所示。根据指令不同，有不可存取的寄存器。例如，PMULS 指令可将 A1 寄存器指定为源寄存器，但是不可将 A0 寄存器指定为源寄存器，详细内容参照指令说明。

数据传送时 DSP 寄存器与总线的关系如**图 3.5** 所示。

表 3.1 DSP 类指令的源寄存器的数据格式

寄存器	指令		保护位 39 32	寄存器位 31 16	15 0
A0、A1	DSP 运算	定点、PDMSB、PSHA		40 位数据	
		整数		24 位数据	
		逻辑、PSHL、PMULS		16 位数据	
	数据传送	MOVX/Y.W、MOVS.W		16 位数据	
		MOVS.L		32 位数据	
A0G、A1G	数据传送	MOVS.W	数据		
		MOVS.L	数据		
X0、X1 Y0、Y1 M0、M1	DSP 运算	定点、PDMSB、PSHA	符号 *	32 位数据	
		整数	符号 *	16 位数据	
		逻辑、PSHL、PMULS		16 位数据	
	数据传送	MOVS.W		16 位数据	
		MOVS.L		32 位数据	

【注】 * 符号扩展并保存在 ALU 的保护位。

表 3.2 DSP 类指令的目标寄存器的数据格式

寄存器	指令		保护位 39 32	寄存器位 31 16	15 0
A0、A1	DSP 运算	定点、PSHA、PMULS	（符号扩展）	40 位结果	
		整数、PDMSB	（符号扩展）	24 位结果	清 0
		逻辑、PSHL	清 0	16 位结果	清 0
	数据传送	MOVS.W	符号扩展	16 位结果	清 0
		MOVS.L	符号扩展	32 位结果	
A0G、A1G	数据传送	MOVS.W	数据	不更新	
		MOVS.L	数据	不更新	
X0、X1 Y0、Y1 M0、M1	DSP 运算	定点、PSHA、PMULS		32 位结果	
		整数、逻辑、PDMSB、PSHL		16 位结果	清 0
	数据传送	MOVX.W、MOVY.W、MOVS.W		16 位结果	清 0
		MOVS.L		32 位结果	

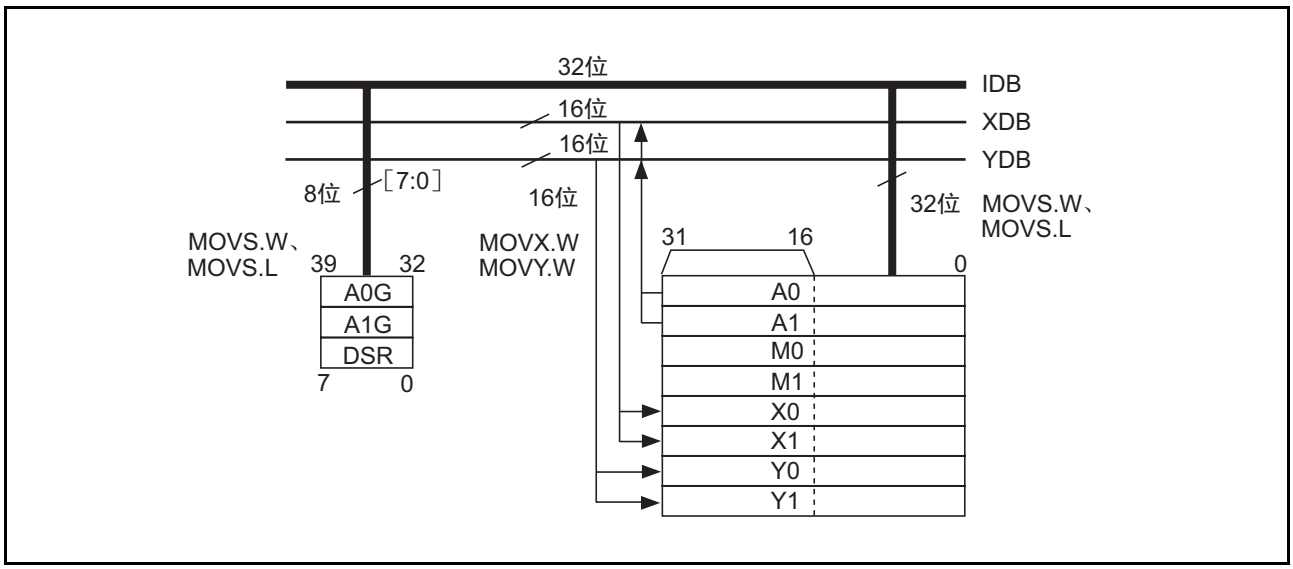


图 3.5 数据传送时 DSP 寄存器与总线的关系

第 4 章 指令特点

4.1 CPU 类指令

为 RISC 结构指令，其特点如下：

1. 16位固定长度指令
   指令长度均为16位固定长度。由此，可提高程序的编码效率。
2. 1条指令/1个状态
   采用流水线方式，基本指令能以1条指令/1个状态执行。
3. 数据长度
   运算的基本数据长度为长字。存储器的存取长度可选择字节/字/长字。存储器的字节和字数据经符号扩展后，以长字运算。立即数在算术运算时经符号扩展、在逻辑运算时经零扩展后，以长字运算。

表 4.1 字数据的符号扩展

SH-1/SH-2/SH-DSP 的 CPU	说明	其他 CPU 例
MOV.W @(disp,PC),R1 ADD R1,R0 ..... .DATA.W H'1234	符号扩展为 32 位，R1 为 H'00001234。然后用 ADD 指令运算。	ADD.W #H'1234,R0

【注】 通过 @(disp,PC) 参考立即数。

4. 加载存储结构
   在寄存器间执行基本运算。与存储器进行运算时，要先将数据加载至寄存器 （加载存储结构）。但是，操作AND等位的指令可直接对存储器执行运算。
5. 延迟转移
   无条件的转移指令等为延迟转移指令。延迟转移指令时，先执行紧接着延迟转移指令后的指令，再转移。这样可减轻转移时的流水线混乱。
   延迟转移中，转移操作产生在槽指令执行之后，但指令执行 （寄存器更新等）的顺序仍为延迟转移指令→延迟槽指令。例如，即使在延迟槽更改保存转移目标地址的寄存器，更改前的寄存器内容仍为转移目标地址。

表 4.2 延迟转移指令

SH-1/SH-2/SH-DSP 的 CPU	说明	其他 CPU 例
BRA TRGET ADD R1,R0	转移至 TRGET 前，执行 ADD。	ADD.W R1,R0 BRA TRGET

6. 乘法/乘法累加运算
   a. SH-1 CPU的乘法/乘法累加运算
      在1～3个状态执行16×16→32的乘法运算。在2～3个状态执行16×16+42→42的乘法累加运算。
   b. SH-2/SH- DSP CPU的乘法/乘法累加运算
      在1～2个状态执行16×16→32的乘法运算。在2～3个状态执行16×16+64→64的乘法累加运算。
      在2～4个状态执行32×32→64的乘法运算及32×32+64→64的乘法累加运算。
7. T位
   比较结果反映在状态寄存器 （SR）的T位，并根据其真假执行条件转移。仅根据最少所需指令更改T位，可提高处理速度。

表 4.3 T 位

SH-1/SH-2/SH-DSP 的 CPU	说明	其他 CPU 例
CMP/GE R1,R0 BT TRGET0 BF TRGET1	R0 ≥ R1 时，T 位置位。 R0 ≥ R1 时，转移至 TRGET0。 R0 < R1 时，转移至 TRGET1。	CMP.W R1,R0 BGE TRGET0 BLT TRGET1
ADD # － 1,R0 CMP/EQ #0,R0 BT TRGET	ADD 时，T 位无变化。 R0 = 0 时，T 位置位。 R0 = 0 时，转移。	SUB.W #1,R0 BEQ TRGET

8. 立即数

字节的立即数配置在操作码中，字与长字的立即数不配置在操作码中，而配置在存储器的表中。通过带位移量的PC相对寻址方式的立即数数据传送指令（MOV），参考存储器的表。

表 4.4 通过立即数参考

分类	SH-1/SH-2/SH-DSP 的 CPU	其他 CPU 例
8 位立即数	MOV #H'12,R0	MOV.B #H'12,R0
16 位立即数	MOV.W @(disp,PC),R0 …… .DATA.W H'1234	MOV.W #H'1234,R0
32 位立即数	MOV.L @(disp,PC),R0 …… .DATA.L H'12345678	MOV.L #H'12345678,R0

【注】 通过 @(disp,PC) 参考立即数。

9. 绝对地址

通过绝对地址参考数据时，事先在存储器的表中配置绝对地址值。执行指令时，通过加载立即数的方法将该值传送到寄存器，以寄存器间接寻址方式参考数据。

表 4.5 通过绝对地址参考

分类	SH-1/SH-2/SH-DSP 的 CPU	其他 CPU 例
绝对地址	MOV.L @(disp,PC),R1 MOV.B @R1,R0 …… .DATA.L H'12345678	MOV.B @H'12345678,R0

10. 16位 / 32位位移

通过16位或32位位移参考数据时，事先在存储器的表中配置位移量值。执行指令时，通过加载立即数的方法将该值传送到寄存器，以带变址的寄存器间接寻址方式参考数据。

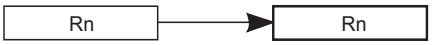
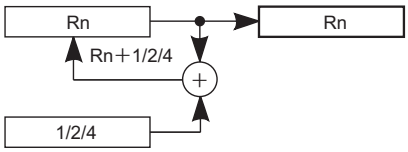
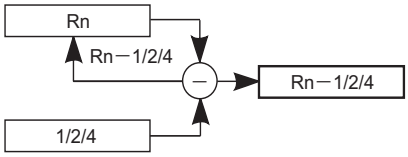
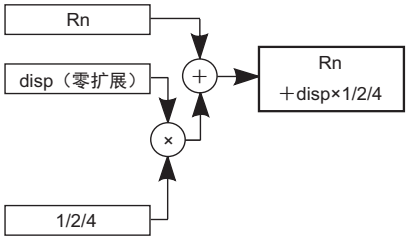
表 4.6 通过位移参考

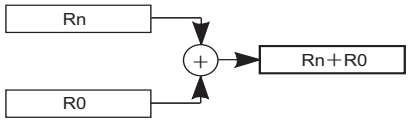
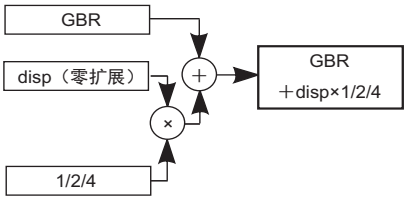
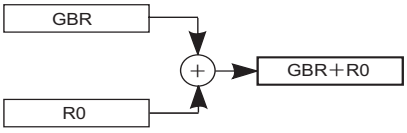
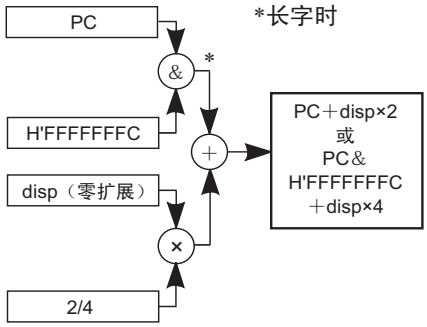
分类	SH-1/SH-2/SH-DSP 的 CPU	其他 CPU 例
16 位位移	MOV.W @(disp,PC),R0 MOV.W @(R0,R1),R2 …… .DATA.W H'1234	MOV.W @(H'1234,R1),R2

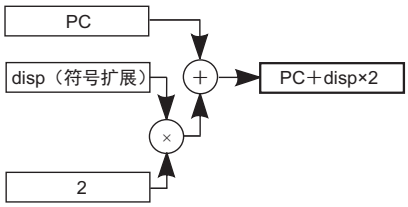
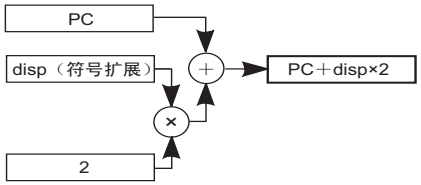
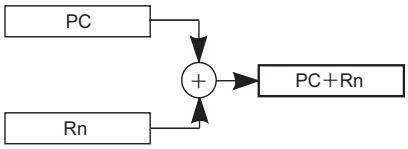
4.2 CPU 类指令的寻址方式

执行 CPU 类指令时，使用的寻址方式和有效地址的计算方法如下：

表 4.7 寻址方式和有效地址

寻址方式	指令格式	有效地址的计算方法	计算式
寄存器直接寻址	Rn	有效地址为寄存器 Rn。 （操作数为寄存器 Rn 的内容。）	—
寄存器间接寻址	@Rn	有效地址为寄存器 Rn 的内容。 	Rn
后增寄存器间接寻址	@Rn+	有效地址为寄存器 Rn 的内容。执行指令后，给 Rn 加常数。操作数长度是字节时常数为 1，是字时常数为 2，是长字时常数为 4。 	Rn 执行指令后 字节：Rn+1→Rn 字：Rn+2→Rn 长字：Rn+4→Rn
先减寄存器间接寻址	@-Rn	有效地址为先减去常数后的寄存器 Rn 的内容。操作数长度是字节时常数为 1，是字时常数为 2，是长字时常数为 4。 	字节：Rn－1→Rn 字：Rn－2→Rn 长字：Rn－4→Rn （用计算后的 Rn 执行指令）
带位移的寄存器间接寻址	@(disp:4,Rn)	有效地址为寄存器 Rn 加 4 位位移量 disp 后的内容。 disp 经零扩展后，根据操作数长度倍增，操作数长度是字节时为 1 倍，是字时为 2 倍，是长字时为 4 倍。 	字节：Rn+disp 字：Rn+disp×2 长字：Rn+disp×4

寻址方式	指令格式	有效地址的计算方法	计算式
带变址的寄存器间接寻址	@(R0,Rn)	有效地址为寄存器 Rn 加 R0 后的内容。 	Rn+R0
带位移量的 GBR 间接寻址	@(disp:8,GBR)	有效地址为寄存器 GBR 加 8 位位移量 disp 后的内容。 disp 经零扩展后，根据操作数长度倍增。操作数长度是字节时为 1 倍，是字时为 2 倍，是长字时为 4 倍。 	字节：GBR+disp 字：GBR+disp×2 长字：GBR+disp×4
带变址的 GBR 间接寻址	@(R0,GBR)	有效地址为寄存器 GBR 加 R0 后的内容。 	GBR+R0
带位移量的 PC 相对寻址	@(disp:8,PC)	有效地址为寄存器 PC 加 8 位位移量 disp 后的内容。 disp 经零扩展后，根据操作数长度倍增。操作数长度是字时为 2 倍，是长字时为 4 倍。并且为长字时，屏蔽 PC 的低 2 位。 	字：PC+disp×2 长字： PC&H'FFFFFFFC+disp×4

寻址方式	指令格式	有效地址的计算方法	计算式
PC 相对寻址	disp:8	有效地址为 8 位位移量 disp 经符号扩展后乘以 2，再加寄存器 PC 后的内容。 	PC+disp×2
	disp:12	有效地址为 12 位位移量 disp 经符号扩展后乘以 2，再加寄存器 PC 后的内容。 	PC+disp×2
	Rn*	有效地址为寄存器 PC 加 Rn 后的内容。 	PC+Rn
立即寻址	#imm:8	将 TST、 AND、 OR、 XOR 指令的 8 位立即数 imm 零扩展。	—
	#imm:8	将 MOV、 ADD、 CMP/EQ 指令的 8 位立即数 imm 符号扩展。	—
	#imm:8	将 TRAPA 指令的 8 位立即数 imm 零扩展后再乘以 4。	—

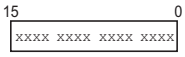
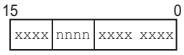
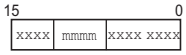
【注】 * 在 SH-2/SH-DSP 中可执行， SH-1 中没有该寻址方式。

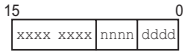
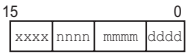
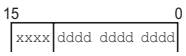
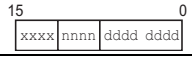
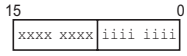
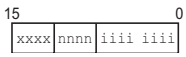
4.3 CPU 类指令的指令格式

表示指令格式、源操作数及目标操作数的含义。操作数的含义取决于操作码。符号说明如下：

xxxx ：操作码
mmmm ：源寄存器
nnnn ：目标寄存器
iiii ：立即数
dddd ：位移量

表 4.8 指令格式

指令格式		源操作数	目标操作数	指令例
0 格式		—	—	NOP
n 格式		—	nnnn：寄存器直接寻址	MOVT Rn
		控制寄存器 或系统寄存器	nnnn：寄存器直接寻址	STS MACH,Rn
		控制寄存器 或系统寄存器	nnnn：先减寄存器间接寻址	STC.L SR,@-Rn
m 格式		mmmm：寄存器直接寻址	控制寄存器 或系统寄存器	LDC Rm,SR
		mmmm：后增寄存器间接寻址	控制寄存器 或系统寄存器	LDC.L @Rm+,SR
		mmmm：寄存器间接寻址	—	JMP @Rm
		mmmm：使用 Rm 的 PC 相对寻址 *1	—	BRAF Rm
nm 格式		mmmm：寄存器直接寻址	nnnn：寄存器直接寻址	ADD Rm,Rn
		mmmm：寄存器直接寻址	nnnn：寄存器间接寻址	MOV.L Rm,@Rn
		mmmm： 后增寄存器间接寻址 （乘法累加运算） nnnn：*2 后增寄存器间接寻址 （乘法累加运算）	MACH,MACL	MAC.W @Rm+,@Rn+
		mmmm：后增寄存器间接寻址	nnnn：寄存器直接寻址	MOV.L @Rm+,Rn
		mmmm：寄存器直接寻址	nnnn：先减寄存器间接寻址	MOV.L Rm,@-Rn
		mmmm：寄存器直接寻址	nnnn：带变址的寄存器间接寻址	MOV.L Rm,@(R0,Rn)

指令格式		源操作数	目标操作数	指令例
md 格式		mmmmdddd： 带位移量的寄存器间接寻址	R0 （寄存器直接寻址）	MOV.B @(disp,Rm),R0
nd4 格式		R0 （寄存器直接寻址）	nnnndddd： 带位移量的寄存器间接寻址	MOV.B R0,@(disp,Rn)
nmd 格式		mmmm：寄存器直接寻址	nnnndddd： 带位移量的寄存器间接寻址	MOV.L Rm,@(disp,Rn)
		mmmmdddd： 带位移量的寄存器间接寻址	nnnn：寄存器直接寻址	MOV.L @(disp,Rm),Rn
d 格式		dddddddd： 带位移量的 GBR 间接寻址	R0 （寄存器直接寻址）	MOV.L @(disp,GBR),R0
		R0 （寄存器直接寻址）	dddddddd： 带位移量的 GBR 间接寻址	MOV.L R0,@(disp,GBR)
		dddddddd： 带位移量的 PC 相对寻址	R0 （寄存器直接寻址）	MOVA @(disp,PC),R0
		dddddddd：PC 相对寻址	—	BF label
d12 格式		ddddddddddddd： PC 相对寻址	—	BRA label (label=disp+PC)
nd8 格式		dddddddd： 带位移量的 PC 相对寻址	nnnn：寄存器直接寻址	MOV.L @(disp,PC),Rn
i 格式		iiiiiiii： 立即寻址	带变址的 GBR 间接寻址	AND.B #imm,@(R0,GBR)
		iiiiiiii： 立即寻址	R0 （寄存器直接寻址）	AND #imm,R0
		iiiiiiii： 立即寻址	—	TRAPA #imm
ni 格式		iiiiiiii： 立即寻址	nnnn：寄存器直接寻址	ADD #imm,Rn

【注】 *1 在 SH-2/SH-DSP 中可使用，在 SH-1 中不可使用。

*2 乘加指令时， nnnn 为源寄存器。

4.4 DSP 类指令方式

DSP 类指令的运算与数据传送如下：

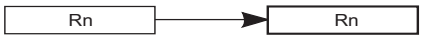
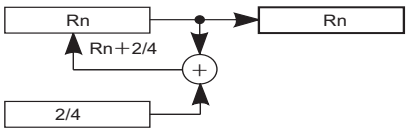
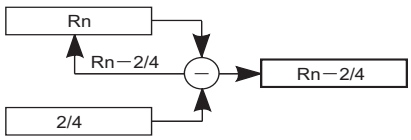
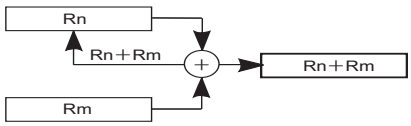
1. ALU定点运算：
   为定点数据40位 （带保护位）或32位 （无保护位）的定点算术运算，有加减运算和比较指令等。
2. ALU整数运算：
   为整数数据24位 （带保护位）或16位 （无保护位）的整数算术运算，有递增指令和递减指令。
3. ALU逻辑运算：
   为逻辑数据16位的逻辑运算，有逻辑 “与”、逻辑 “或”及逻辑 “异或”。
4. 定点乘法运算：
   为定点数据高16位的定点乘法运算 （算术运算），不更新DC位等状态位。
5. 移位运算：
   有算术移位运算和逻辑移位运算。算术移位运算为定点数据40位 （带保护位）或32位 （无保护位）的算术移位；逻辑移位运算为逻辑数据16位的逻辑运算。算术移位运算的移位量为– 32～+32 （负值右移，正值左移），逻辑移位运算的移位量为–16～+16。
6. MSB检测指令：
   计算数据正规化处理所需的移位量。通过整数数据24位 （带保护位）或16位 （无保护位），计算定点数据40位 （带保护位）或32位 （无保护位）的MSB位的位置。
7. 舍入运算：
   将定点数据40位 （带保护位）舍入为24位或将32位 （无保护位）舍入为16位。
8. 数据传送：
   有从X、 Y存储器加载、存储16位数据的X、 Y数据传送和从所有存储器加载、存储16位或32位数据的单数据传送。 X、 Y数据传送可同时并发执行2个处理，不更新DC位等状态位。

运算指令中具有无条件的运算指令和带条件的运算指令 （判断 DC 位后执行）。为带条件的运算指令时，不更新 DC 位等状态位。 DC 位等状态位在算术运算、逻辑运算、算术移位及逻辑移位中的设定各不相同。MSB 检测指令、舍入运算时的 DC 位等状态位，设定为算术运算。

算术运算中有上溢防止功能 （饱和运算）。如果由 SR 寄存器的 S 位指定饱和运算，则保存运算结果上溢时的最大值 （正）或最小值 （负）。 S 位的功能在 ALU、移位和乘法运算的所有算术运算中有效。

4.5 DSP 类指令的寻址方式

表 4.9 寻址方式和有效地址

寻址方式	指令格式	有效地址的计算方法	计算式	指令
寄存器间接寻址	@Rn	有效地址为寄存器 Rn 的内容。 	Rn	MOVS MOVX MOVY
后增寄存器间接寻址	@Rn+	有效地址为寄存器 Rn 的内容。执行指令后，给 Rn 加常数。操作数长度为字时常数是 2，为长字时常数是 4。 	Rn 执行指令后 字： Rn+2→Rn 长字： Rn+4→Rn	MOVS MOVX （仅限字） MOVY （仅限字）
先减寄存器间接寻址	@-Rn	有效地址为先减去常数后寄存器 Rn 的内容。操作数长度为字时常数是 2，为长字时常数是 4。 	字： Rn－ 2→Rn 长字： Rn－ 4→Rn （用计算后的 Rn, 执行指令）	MOVS
带变址的寄存器间接寻址	@Rn+Rm	有效地址为寄存器 Rn 加 Rm 后的内容。 	Rn+Rm	MOVS MOVX MOVY

4.6 DSP 数据寻址

DSP 类指令时，执行 2 种不同的存储器存取。一种是 X、 Y 数据传送指令 （MOVX.W、 MOVY.W），另一种是单数据传送指令 （MOVS.W、 MOVS.L）。这两种指令的数据寻址不同。数据传送指令的概要如**表 4.10** 所示。

表 4.10 数据传送指令的概要

	X、Y 数据传送处理 （MOVX.W、MOVY.W）	单数据传送处理 （MOVS.W、MOVS.L）
地址寄存器	Ax：R4、R5、Ay：R6、R7	As：R2、R3、R4、R5
变址寄存器	Ix：R8、Iy：R9	Is：R8
寻址	Nop/Inc(+2)/ 变址加法：后增	Nop/Inc(+2,+4)/ 变址加法：后增
	—	Dec(–2,–4)：先减
模寻址	可	不可
数据总线	XDB、YDB	主总线
数据长度	16 位 （字）	16 位 /32 位 （字 / 长字）
总线竞争	无	有
存储器	X、Y 数据存储器	所有的存储空间
源寄存器	Da：A0、 A1	Ds：A0/A1、 M0/M1、 X0/X1、 Y0/Y1、 A0G、 A1G
目标寄存器	Dx：X0/X1、 Dy：Y0/Y1	Ds：A0/A1、 M0/M1、 X0/X1、 Y0/Y1、 A0G、 A1G

4.6.1 X、Y 数据寻址

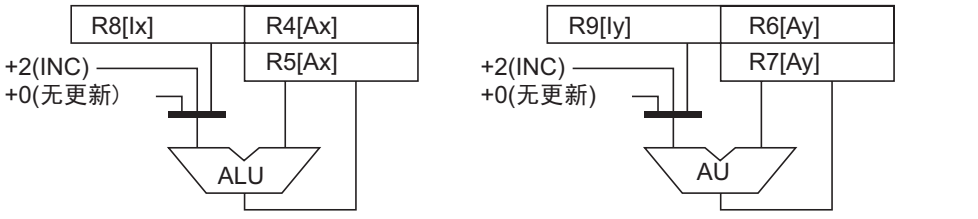
DSP 类指令时，使用 MOVX.W、 MOVY.W 指令，可同时存取 X、 Y 数据存储器。 DSP 类指令中有 2 个地址指针以便同时存取 X、 Y 数据寄存器。 DSP 类指令只可执行指针寻址，无立即寻址。地址寄存器分为 2 组：R4 和 R5 寄存器为 X 存储器的地址寄存器 （Ax）； R6 和 R7 寄存器为 Y 存储器的地址寄存器 （Ay）。X、 Y 数据传送指令有以下 3 种寻址方式：

1. 无更新地址寄存器：
   Ax 和 Ay 寄存器为地址指针，不可更新。
2. 加法变址寄存器：
   Ax 和 Ay 寄存器为地址指针，数据传送后分别加 Ix、 Iy 寄存器的值 （后增）。
3. 增量地址寄存器：
   Ax 和 Ay 寄存器为地址指针，数据传送后分别加 2 （后增）。

各地址指针中均有变址寄存器。 R8 寄存器为 X 存储器地址寄存器 （Ax）的变址寄存器 （Ix）， R9 寄存器为 Y 存储器地址寄存器 （Ay）的变址寄存器 （Iy）。

以字为单位处理 X、 Y 数据传送指令。以 16 位存取 X、 Y 数据存储器。因此，增量处理时，地址寄存器加 2 ；减量处理时，在变址寄存器设定 –2，并指定加法变址寄存器寻址。 X、 Y 数据寻址时，仅地址指针的 bit1 ～ 15 有效，必须给地址指针和变址寄存器的 bit0 写入 0。

X、 Y 数据传送的寻址如**图 4.1** 所示。使用 X、 Y 总线存取 X 存储器和 Y 存储器时，忽略 Ax （R4 或 R5）和 Ay （R6 或 R7）的高位字。 @Ay+ 和 @Ay+Iy 的结果保存于 Ay 的低位字，高位字保持原来的值。



【注】3种地址处理方法
（1）递增
（2）变址寄存器加法(Ix/Iy)
（3）无递增
所有处理均为后增方式
欲递减地址指针时，在变址寄存器设定-2/-4

AU： 为运行DSP 寻址而追加的加法器

图 4.1 X、 Y 数据传送的寻址

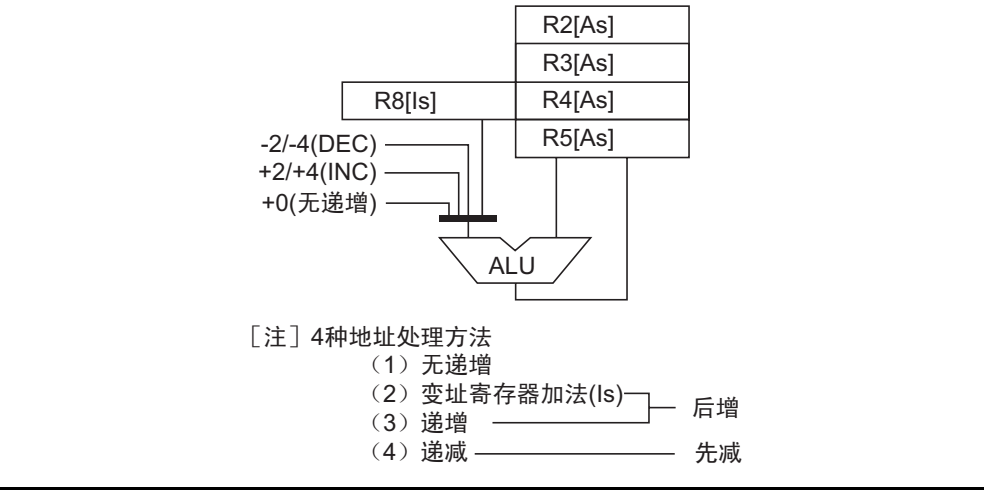
4.6.2 单数据寻址

DSP 类指令中有单数据传送指令 （MOVS.W、 MOVS.L）。将数据加载至 DSP 寄存器，再从 DSP 寄存器存储数据。该指令时， R2 ～ R5 寄存器可用作单数据传送的地址寄存器 （As）。

单数据传送指令有以下 4 种数据寻址方式：

1. 无增量地址寄存器：
   As寄存器为地址指针，不递增。
2. 加法变址寄存器：
   As寄存器为地址指针，数据传送后加Is寄存器的值 （后增）。
3. 增量地址寄存器：
   As寄存器为地址指针，数据传送后加2或加4 （后增）。
4. 减量地址寄存器：
   As寄存器为地址指针，数据传送前加–2或–4 （减去+2或+4）（先减）。

地址指针 （As）将 R8 寄存器用作变址寄存器 （Is）。单数据传送寻址如图 4.2 所示。



［注］4种地址处理方法
（1）无递增
（2）变址寄存器加法(Is) — 后增
（3）递增 — 后增
（4）递减 — 先减

图 4.2 单数据传送的寻址

4.6.3 模寻址

SH-DSP 与其他 DSP 相同，具有模寻址方式。该方式中，地址寄存器也同样可递增。如果地址指针的值为已设定的模结束地址，则地址指针指向模起始地址。

模寻址仅在 X、Y 数据传送指令（MOVX.W、MOVY.W）时有效。SR 寄存器的 DMX 位置位时，X 地址寄存器为模寻址方式； DMY 位置位时， Y 地址寄存器为模寻址方式。模寻址仅对 X、 Y 地址寄存器中的一个有效，两者不可同时为模寻址方式。因此，不得同时置位 DMX 和 DMY，万一同时置位，仅 DMY 有效。

MOD 寄存器用于指定模地址区域的起始和结束地址，并保存 MS （Modulo Start：模起始）和 ME （Modulo End：模结束）。 MOD 寄存器 （MS、 ME）的使用示例如下：

```
                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
```

在 MS、 ME 分别指定起始、结束地址，然后将 DMX 或 DMY 位置 1。将地址寄存器的内容与 ME 进行比较，如果与 ME 匹配，则将起始地址 MS 保存到地址寄存器。将地址寄存器低 16 位与 ME 进行比较，最大的模长度为 64K 字节，足以存取 X、 Y 数据存储器。模寻址的框图如图 4.3 所示。

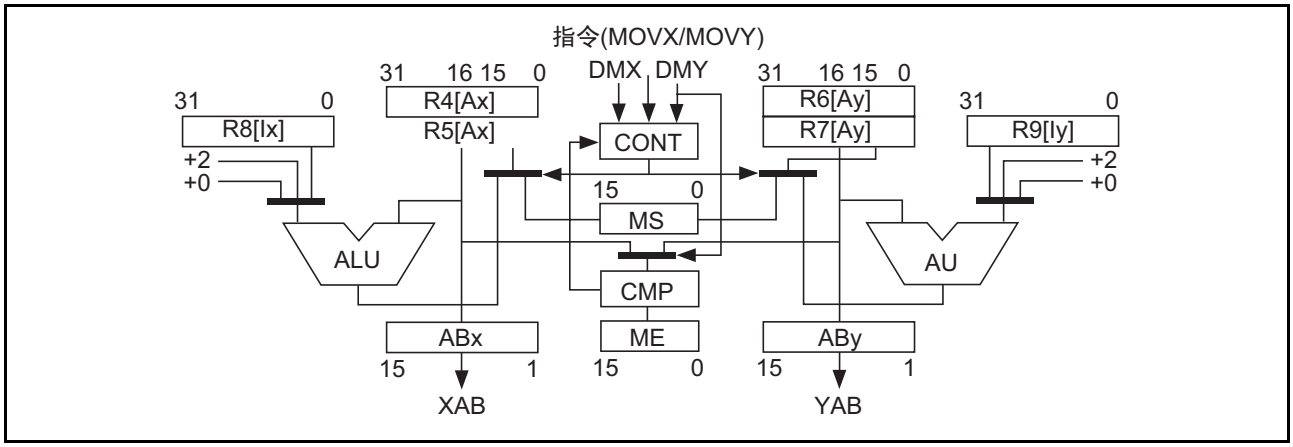


图 4.3 模寻址

模寻址的示例如下所示：

```
MS = H'C008; ME=H'C00C; R4=H'C008;
DMX=1; DMY=0;    （对地址寄存器 Ax(R4,R5) 设定模寻址）

通过以上设定， R4 寄存器的变化如下：
          R4: H'C008
Inc.      R4: H'C00A
Inc.      R4: H'C00C
Inc.      R4: H'C008  （已为模结束地址，因此返回模起始地址）
```

将模起始和结束地址的高 16 位设置为相同数据。这是因为模起始地址仅替换地址寄存器的低 16 位。

【注】 DSP 数据寻址中使用加法变址时，地址指针有时与 ME 不匹配，可能超出该值。此时，地址指针不返回模起始地址。

4.6.4 DSP 寻址操作

包括模寻址的流水线执行阶段 （EX）的 DSP 寻址操作如下：

```
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 类指令的指令格式

SH-DSP 中追加了用于数字信号处理的新指令，新指令分为以下 2 种：

1. 存储器与DSP寄存器的双、单数据传送指令 （16位长度）
2. 使用DSP引擎处理的并发处理指令 （32位长度）

各指令格式如**图 4.4** 所示。

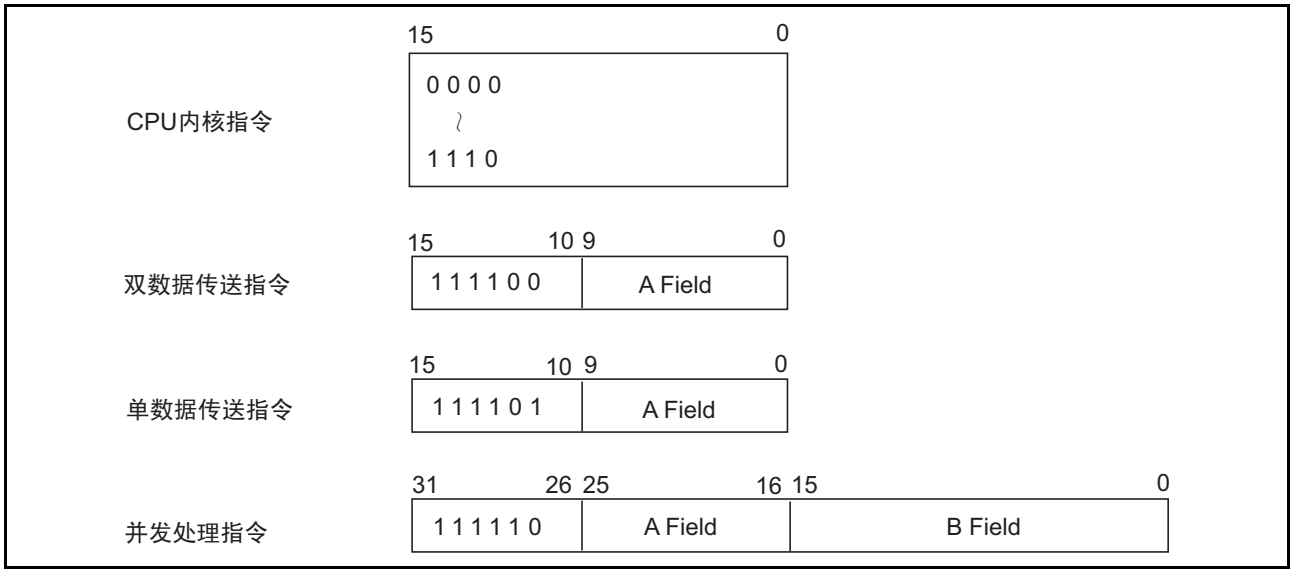


图 4.4 DSP 指令格式

4.7.1 双、单数据传送指令

双数据传送指令的指令格式如**表 4.11** 所示；单数据传送指令的指令格式如**表 4.12** 所示。

表 4.11 双数据传送指令格式

分类	助记符		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X 存储器数据传送	NOPX		1	1	1	1	0	0	0		0		0		0	0		
	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 存储器数据传送	NOPY		1	1	1	1	0	0		0		0		0			0	0
	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 4.12 单数据传送指令格式

分类	助记符		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
单数据传送	MOVS.W	@–As,Ds	1	1	1	1	0	1	As		Ds		0:(*1)		0	0	0	0
	MOVS.W	@As,Ds							0:R4				1:(*1)		0	1		
	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 系统保留码

4.7.2 并发处理指令

并发处理指令可高效执行使用 DSP 单元的数字信号处理。指令长度为 32 位，可同时并发执行 4 个处理（即， ALU 运算、乘法运算和 2 个数据传送）。

并发处理指令分为 A 字段和 B 字段。A 字段定义数据传送指令； B 字段定义 ALU 运算指令和乘法运算指令。此指令可单独定义、单独处理，且可同时并发执行。 A 字段的并发数据传送指令如**表 4.13** 所示； B 字段的 ALU 运算指令、乘法运算指令如**表 4.14** 所示。 A 字段指令与**表 4.11** 中的双数据传送相同。

表 4.13 A 字段的并发数据传送指令

分类	助记符	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15–0
X 存储器数据传送	NOPX	1	1	1	1	1	0	0		0		0		0	0			B字段
	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 存储器数据传送	NOPY								0		0		0			0	0	B字段
	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 字段的 ALU 运算指令、乘法运算指令

分类	助记符	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. 移位	PSHL #imm, Dz PSHA #imm, Dz	1 1 1 1 1 0	A字段	0 0 0 0 0 0 0 1	0	-16<=imm<=+16 -32<=imm<=+32		Dz
	保留			0 0 0 0 0 1	1			
6个操作数 并行指令	PMULS Se, Sf, Dg 保留 PSUB Sx, Sy, Du PMULS Se, Sf, Dg PADD Sx, Sy, Du PMULS Se, Sf, Dg			0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1	Se 0:X0 1:X1 2:Y0 3:A1	Sf 0:Y0 1:Y1 2:X0 3:A1	Sx Sy 0:X0 0:Y0 1:X1 1:Y1 2:A0 2:M0 3:A1 3:M1	Dg Du 0:M0 0:X0 1:M1 1:Y0 2:A0 2:A0 3:A1 3:A1
3个操作数 指令	保留 PSUBC Sx, Sy, Dz PADDC Sx, Sy, Dz PCMP Sx, Sy 保留 PABS Sx, Dz PRND Sx, Dz PABS Sy, Dz PRND Sy, Dz 保留			1 0 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1	0 0 0 1 1 0 1 1	0 0		Dz 0:(*1) 1:(*1) 2:(*1) 3:(*1) 4:(*1) 5:A1 6:(*1) 7:A0 8:X0 9:X1 A:Y0 B:Y1 C:M0 D:(*1) E:M1 F:(*1)
带条件的 3个操作数 指令	[if cc] PSHL Sx, Sy, Dz [if cc] PSHA Sx, Sy, Dz [if cc] PSUB Sx, Sy, Dz [if cc] PADD Sx, Sy, Dz 保留 [if cc] PAND Sx, Sy, Dz [if cc] PXOR Sx, Sy, Dz [if cc] POR Sx, Sy, Dz [if cc] PDEC Sx, Dz [if cc] PINC Sx, Dz [if cc] PDEC Sy, Dz [if cc] PINC Sy, Dz [if cc] PCLR Dz [if cc] PDMSB Sx, Dz 保留 [if cc] PDMSB Sy, Dz [if cc] PNEG Sx, Dz [if cc] PCOPY Sx, Dz [if cc] PNEG Sy, Dz [if cc] PCOPY Sy, Dz 保留 [if cc] PSTS MACH, Dz [if cc] PSTS MACL, Dz [if cc] PLDS Dz, MACH [if cc] PLDS Dz, MACL (*2) 保留			1 0 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1	0 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1	0 0	ifcc 01：无条件 10：DCT 11：DCF 0 0 ifcc 0 0		
	保留	1 1 1 1 1 1			0 *			

【注】 *1 系统保留码

*2 [if cc]：DCT（DC位真）、DCF（DC位假）或无（无条件指令）。

4.8 ALU 定点运算

(1) 运算功能

ALU 定点算术运算以 40 位 （基本精度 32 位加 8 位保护位）进行运算。源操作数为无保护位的寄存器时，将源操作数的符号位扩展为保护位后转存；目标操作数为无保护位的寄存器时，运算结果的低 32 位保存至目标寄存器。

在寄存器之间执行 ALU 定点算术运算。分别从 DSP 寄存器中独立选择源寄存器及目标寄存器，所选的寄存器有保护位时，执行包括保护位的运算。在流水线流程的最后 DSP 阶段执行此运算。

执行 ALU 算术运算时，固定为根据运算结果更新 DSR 寄存器的 DC、 N、 Z、 V、 GT 位。但为带条件的指令时，即使在指定状态也不更新状态位；为无条件的指令时，根据运算结果更新。

可通过 CS［2:0］位选择在 DC 位反映的状态。但与 CS 位设定无关， PADDC 指令与 PSUBC 指令的 DC 位被更新。为 PADDC 指令时， DC 位作为进位标志更新；为 PSUBC 指令时， DC 位作为借位标志更新。

ALU 定点算术运算流程如**图 4.5** 所示。

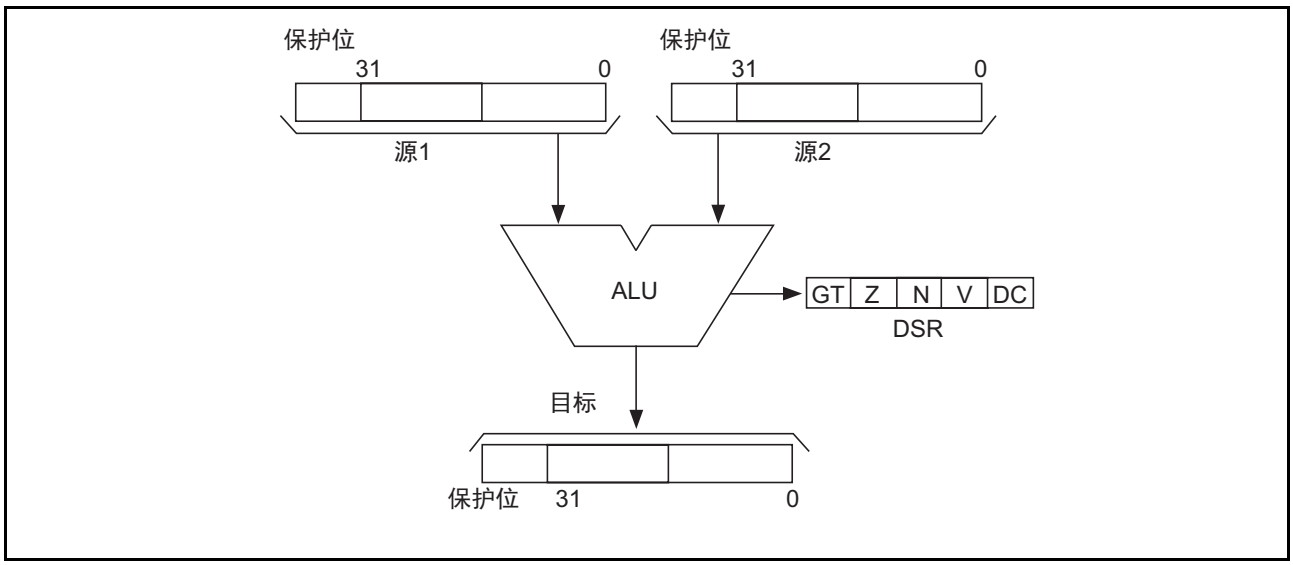


图 4.5 ALU 定点算术运算流程

如果将存储器读取的目标操作数与 ALU 运算的源操作数设为相同值，在与 ALU 运算相同的行编写数据传送指令的程序时，在存储器存取阶段 （MA）从存储器加载的数据不用作 ALU 运算指令的源操作数。此时，将先执行指令的结果用作 ALU 运算的源操作数，然后，作为数据加载指令的目标操作数进行更新。该流程如**图 4.6** 所示。

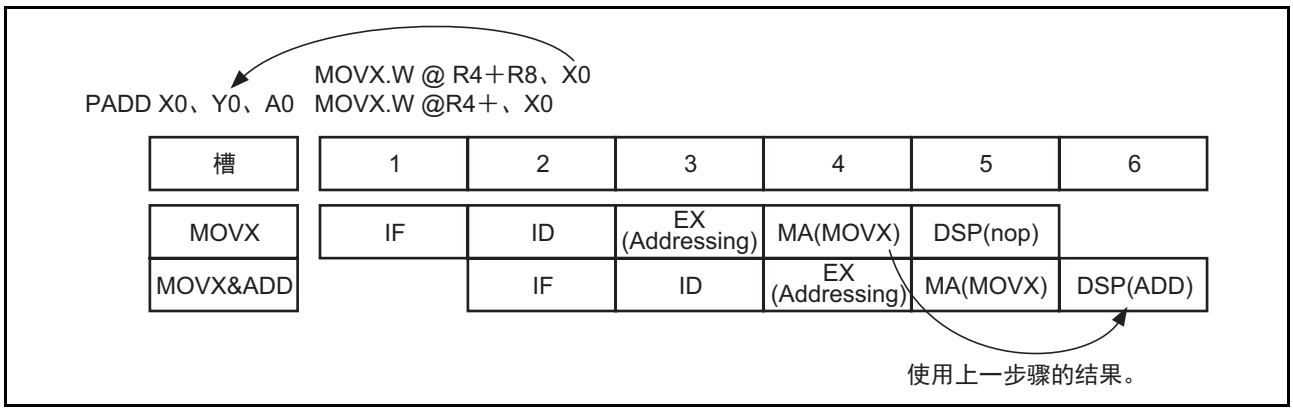


图 4.6 处理流程例

(2) 指令与操作数

ALU 定点算术运算的种类如**表 4.15** 所示；各操作数与寄存器的对应关系如**表 4.16** 所示。

表 4.15 ALU 定点算术运算的种类

助记符	功能	源 1	源 2	目标
PADD	加法	Sx	Sy	Dz （Du）
PSUB	减法	Sx	Sy	Dz （Du）
PADDC	带进位的加法	Sx	Sy	Dz
PSUBC	带借位的减法	Sx	Sy	Dz
PCMP	比较	Sx	Sy	—
PCOPY	数据复制	Sx	—	Dz
		—	Sy	Dz
PABS	绝对值	Sx	—	Dz
		—	Sy	Dz
PNEG	符号取反	Sx	—	Dz
		—	Sy	Dz
PCLR	清零	—	—	Dz

表 4.16 ALU 定点算术运算的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器

Du：与乘法组合时的操作数

(3) DC 位

按照 DSR 寄存器的 CS2 ～ CS0 位 （Condition Selection、状态选择）的设定， DC 位如下变化：

(a) 进位 / 借位模式： CS2 ～ CS0 ＝ 000

DC 位表示运算结果从 MSB （Most Significant Bit）位产生进位或借位，与保护位无关。 DSR 寄存器的初始状态为本模式。产生进位和借位的例子如**图 4.7** 所示。

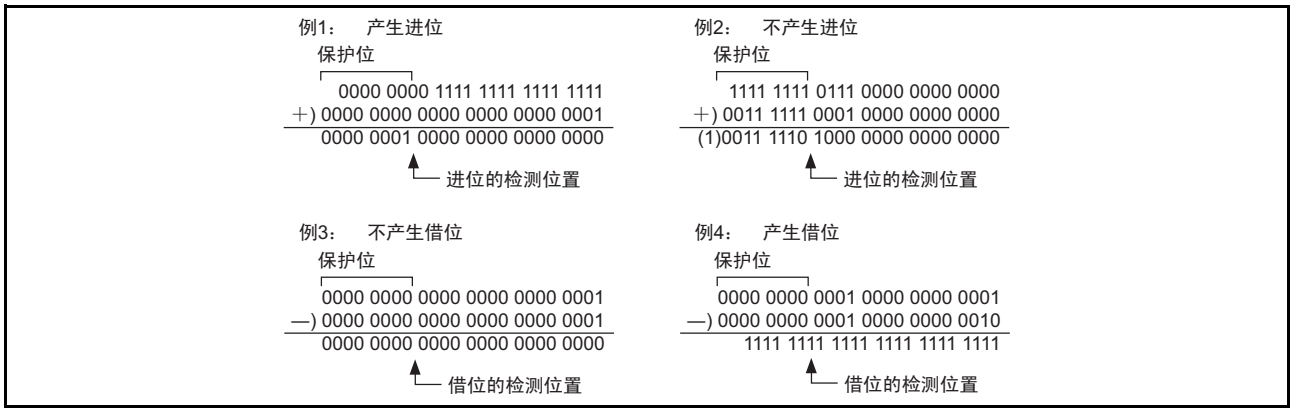


图 4.7 产生进位和借位的例子

(b) 负值模式： CS2 ～ CS0=001

DC 位与运算结果的 MSB 位的值相同。结果为负值时，DC 位变为 1；零或正值时，DC 位为 0。ALU 算术运算总是以 40 位来执行，因此，不用目标操作数的 MSB 位，而用包含运算结果保护位的 MSB 位判断符号位表示正还是负。正负判断例如图 4.8 所示。在该模式 DC 位的值与状态位 N 位的值相同。

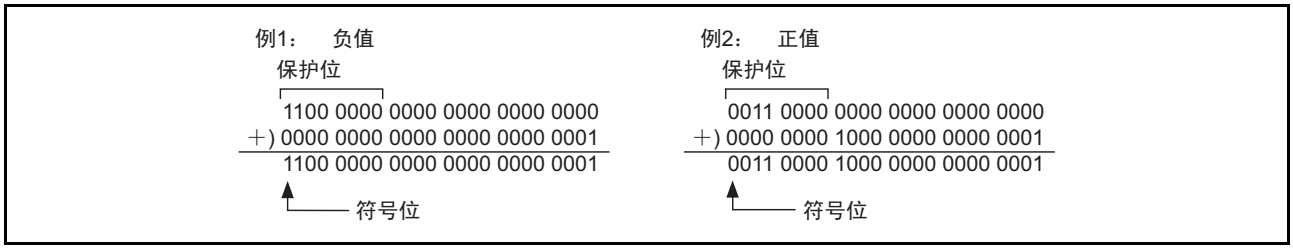


图 4.8 正负判断例

(c) 零值模式： CS2 ～ CS0=010

DC 位表示运算结果是否为零。结果为零时，DC 位变为 1；结果不为零时，DC 位为 0。在该模式 DC 位的值与状态位 Z 位的值相同。

(d) 上溢模式： CS2 ～ CS0=011

DC 位表示运算结果是否产生上溢。运算结果超出除保护位之外的目标寄存器的范围时， DC 位置 1。即使有保护位， DC 位也按无保护位判断上溢。因此，较大的数使用保护位时， DC 位总是被置 1。在该模式 DC 位的值与状态位 V 位的值相同。上溢判断例如图 4.9 所示。

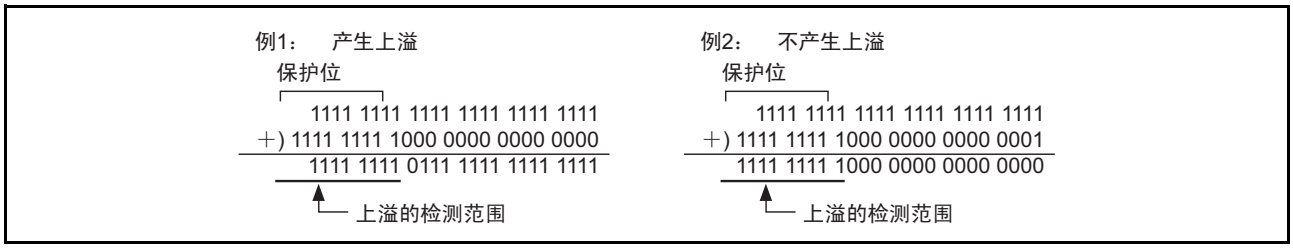


图 4.9 上溢判断例

(e) 带符号的 “大于” 模式： CS2 ～ CS0 = 100

DC 位表示比较指令 PCMP 的判断结果是否是源 1 数据 （带符号）大于源 2 数据 （带符号）。源 1 数据大于源 2 数据时，比较结果为正值，因此该模式与负值模式类似。但，即使源 1 数据大于源 2 数据，如果超出目标操作数的范围，比较结果的符号变为负值，此时，更新 DC 位。在该模式 DC 位的值与状态位 GT 位的值相同。本模式时的 DC 位用公式定义如下。但是，在运算结果超出含保护位字段的目标操作数表示范围时，VR 的值为真。

DC 位 = ～ {（N 位 ∧ VR） | Z 位}

如果在本模式执行 PCMP 指令，则 DC 位的值与表示 CPU 类指令的 CMP/GT 指令结果的 T 位相同。在本模式，即使不是 PCMP 指令，也根据上述定义更新 DC 位。

(f) 带符号的 “大于等于” 模式： CS2 ～ CS0 = 101

DC 位表示比较指令 PCMP 的执行结果是否是源 1 数据 （带符号）大于等于源 2 数据 （带符号）。因此，在本模式判断 DC 位之前，先执行 PCMP 指令。本模式除是否等于之外，类似于带符号的 “大于” 模式。本模式时的 DC 位用公式定义如下。但是，在运算结果超出含保护位字段的目标操作数表示范围时， VR 的值为真。

DC 位 = ～ （N 位 ∧ VR）

如果在本模式执行 PCMP 指令，则 DC 位的值与表示 SH 内核指令的 CMP/GE 指令结果的 T 位相同。在本模式，即使不是 PCMP 指令，也根据上述定义更新 DC 位。

(4) 状态位

状态位的设定如下：

N 位 （Negative bit、负值位）的值与通过 CS 位指定负值模式时 DC 位的值相同。运算结果为负值时， N 位置 1 ； 为零或正值时， N 位清 0。

Z 位 （Zero bit、零位）的值与通过 CS 位指定零值模式时 DC 位的值相同。运算结果为零时， Z 位置 1 ； 结果不为零时， Z 位清 0。

V 位 （Overflow bit、上溢位）的值与通过 CS 位指定上溢模式时 DC 位的值相同。运算结果超出除保护位之外的目标寄存器范围时， V 位置 1 ； 除此之外清 0。

GT 位 （Greater Than bit、带符号的 “大于” 位）的值与通过 CS 位指定带符号的 “大于” 模式时的 DC 位的值相同。比较结果为源 1 数据大于源 2 数据时， GT 位置 1 ； 除此之外清 0。

(5) 上溢防止功能 （饱和运算）

如果 SR 寄存器的 S 位置 1，则在 DSP 单元执行的所有 ALU 算术运算中可执行上溢防止功能。运算结果上溢时，保存最大值 （正）或最小值 （负）。

4.9 ALU 整数运算

ALU 整数运算基本为高位字（高 16 位、bit31 ～ 16）和 8 位保护位的 24 位运算。 ALU 整数算术运算时，忽略源操作数的低位字 （低 16 位、 bit15 ～ 0），将目标操作数的低位字清 0。源操作数中无保护位时，将符号位作为保护位扩展后保存；目标操作数中无保护位时，将除运算结果保护位之外的高位字保存到目标寄存器的高位字。

整数运算基本与 ALU 定点算术运算相同。整数运算的运算指令只有递增和递减指令 2 种。第 2 操作数实质为＋ 1 或－ 1。 16 位整数数据 （字数据）加载到 DSP 寄存器并保存到高位字。然后，使用 DSP 寄存器的高位字运算。有保护位时，保护位也有效。在流水线流程中命名为 DSP 阶段的最后阶段执行此运算。

执行 ALU 整数算术运算时，基本根据运算结果更新 DSR 寄存器的 DC、 N、 Z、 V、 GT 位，这一点与 ALU 定点运算相同。

为带条件的指令时，即使指定条件成立、并已执行指令，也不更新条件位和标志；为无条件的指令时，总是根据运算结果更新。

ALU 整数运算流程如**图 4.10** 所示。

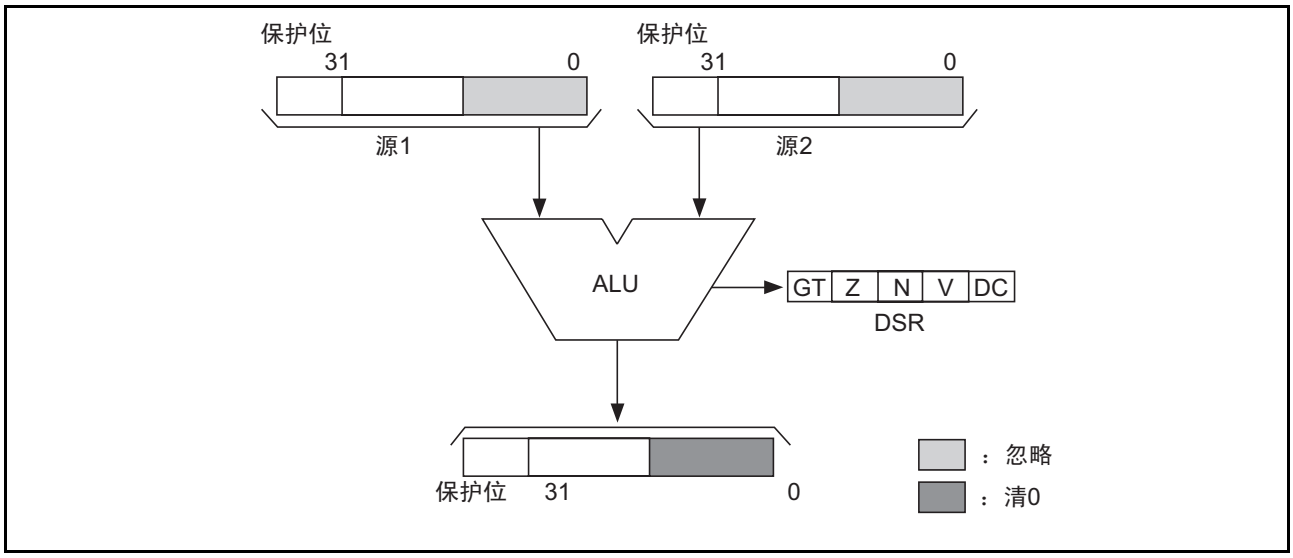


图 4.10 ALU 整数运算流程

ALU 整数运算的种类如**表 4.17** 所示；各操作数与寄存器的对应关系如**表 4.18** 所示。

表 4.17 ALU 整数运算的种类

助记符	功能	源 1	源 2	目标
PINC	递增 1	Sx	（+1）	Dz
		（+1）	Sy	Dz
PDEC	递减 1	Sx	（–1）	Dz
		（–1）	Sy	Dz

表 4.18 ALU 整数运算的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器。

为了执行上溢防止功能（饱和运算），SR 寄存器的 S 位必须置 1。可对以 DSP 类指令执行的 ALU 整数算术运算，指定上溢防止功能。运算结果上溢时，保存最大值（正）或最小值（负）。

4.10 ALU 逻辑运算

(1) 运算功能

ALU 逻辑运算也在寄存器之间执行。各个源操作数和目标操作数独立选择一个 DSP 寄存器。此运算仅使用各操作数的高位字，忽略源操作数的低位字和保护位，并将目标操作数的低位字和保护位清 0。在流水线流程中命名为 DSP 的最后阶段执行此运算。

执行 ALU 逻辑运算时，基本根据运算结果更新 DSR 寄存器的 DC 位、N、Z、V、GT 标志。为带条件的指令时，即使指定条件成立、并已执行指令，也不更新条件位和标志；为无条件的指令时，总是根据运算结果更新。根据 CS 位的指定，更新 DC 位。

ALU 逻辑运算流程如图 4.11 所示。

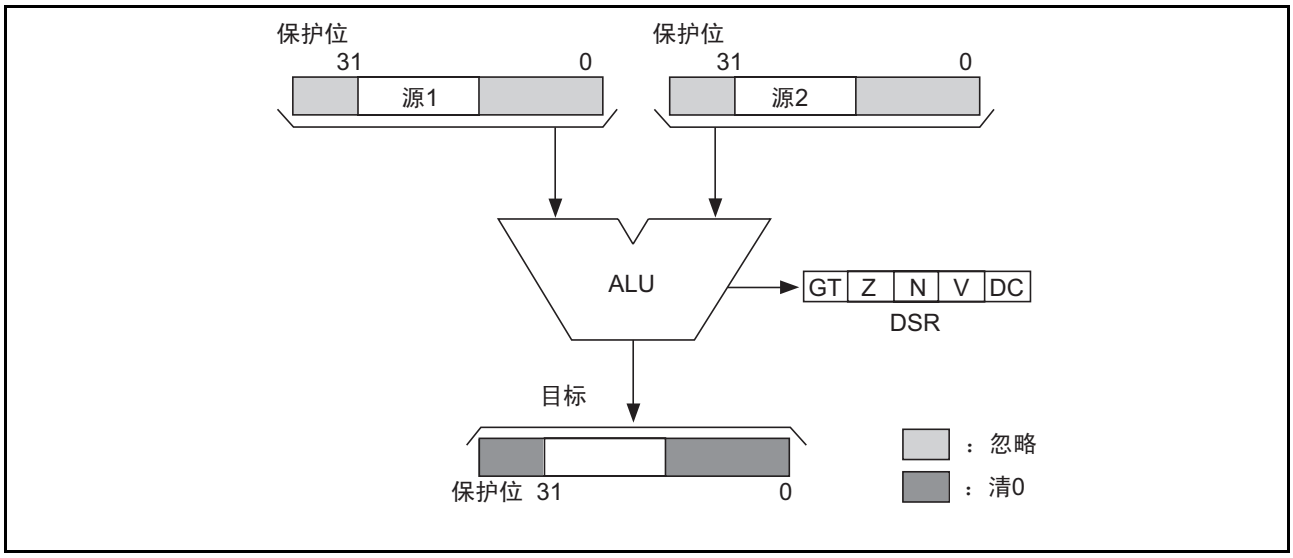


图 4.11 ALU 逻辑运算流程

(2) 指令与操作数

ALU 逻辑运算的种类如表 4.19 所示；各操作数与寄存器的对应关系与 ALU 定点运算相同，如表 4.20 所示。

表 4.19 ALU 逻辑运算的种类

助记符	功能	源 1	源 2	目标
PAND	逻辑 “与”	Sx	Sy	Dz
POR	逻辑 “或”	Sx	Sy	Dz
PXOR	逻辑 “异或”	Sx	Sy	Dz

表 4.20 ALU 逻辑运算的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器

(3) DC 位

逻辑运算的 DC 位设定如下：

(a) 进位 / 借位模式： CS2 ～ CS0=000

DC 位总是清 0。

(b) 负值模式： CS2 ～ CS0=001

DC 位为运算结果 bit31 的值。该模式的 DC 位的值与 N 位的值相同。

(c) 零值模式： CS2 ～ CS0=010

运算结果为零时， DC 位置 1 ；除此之外清 0。该模式的 DC 位的值与 Z 位的值相同。

(d) 上溢模式： CS2 ～ CS0=011

DC 位总是清 0。该模式的 DC 位的值与 V 位的值相同。

(e) 带符号的 “大于” 模式： CS2 ～ CS0=100

DC 位总是清 0。该模式的 DC 位的值与 GT 位的值相同。

(f) 带符号的 “大于等于” 模式： CS2 ～ CS0=101

DC 位总是清 0。

(4) 状态位

状态位的设定如下：

- N位为运算结果bit31的值。
- 运算结果为零时， Z位置1；除此之外清0。
- V位总是清0。
- GT位总是清0。

4.11 定点乘法运算

DSP 类指令的乘法运算为带符号的单精度乘法运算。该运算在 1 个周期完成处理。需要执行双精度乘法运算时，使用 CPU 类指令的双精度运算。

乘法运算的结果基本为 32 位运算结果。指定目标操作数为有保护位的寄存器时，执行符号扩展。

DSP 类指令的乘法运算不是整数运算，而是定点算术运算，因此，常数和被乘数的高位字分别输入到 MAC 运算器。如果为 CPU 类指令的乘法运算，源操作数和目标操作数的低位字输入到 MAC 运算器。因此，DSP 类指令与 CPU 类指令的运算结果不同。 CPU 类指令的乘法运算结果对准目标操作数的 LSB， DSP 类指令的定点乘法运算的运算结果对准 MSB，因此，定点乘法运算时，运算结果的 LSB 总是为 0。

定点乘法运算流程如**图 4.12** 所示。

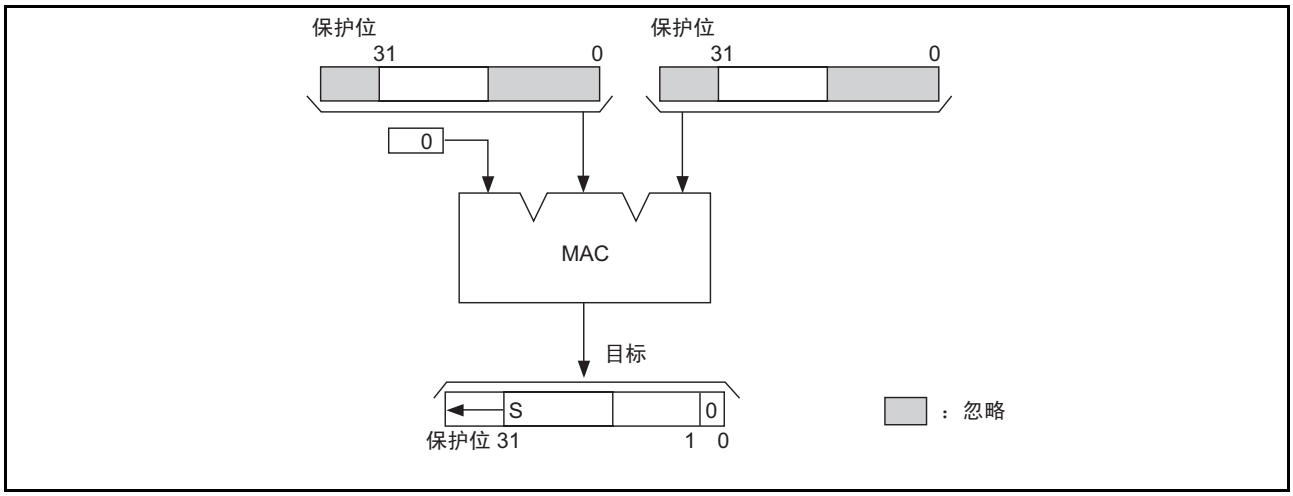


图 4.12 定点乘法运算流程

定点乘法运算的种类如**表 4.21** 所示；各操作数与寄存器的对应关系如**表 4.22** 所示。

表 4.21 定点乘法运算的种类

助记符	功能	源 1	源 2	目标
PMULS	带符号的乘法运算	Se	Sf	Dg

表 4.22 定点乘法运算的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器

DSP 类指令的定点乘法运算在 1 个周期完成 16 位 ×16 位的单精度运算处理；除此之外的乘法运算，与以往的 CPU 类指令的乘法运算相同。

乘法运算指令不更新 DSR 寄存器的各状态位及 DC、 N、 Z、 V、 GT 位。

上溢防止功能 （饱和运算）在 DSP 类指令的乘法运算中仍然有效。指定 SR 寄存器的 S 位置 1，运算结果值上溢或下溢时，分别为最大值或最小值。 DSP 类指令的定点乘法运算时，仅在 H'8000×H'8000 （（–1.0） ×（–1.0））时产生上溢。 S 位为 0 时，该运算结果为 H'8000 0000，这表示 –1.0，并不是正确值 +1.0。 S 位为 1 时，上溢防止功能起作用，该结果为 H'007FFF FFFF。

4.12 移位运算

移位运算的移位量由寄存器指定或由直接立即数指定，其他的源操作数和目标操作数由寄存器指定。移位运算有算术移位和逻辑移位两种，如**表 4.23** 所示。除立即操作数之外，各操作数与寄存器的对应关系，与 ALU 定点运算相同，如**表 4.24** 所示。

表 4.23 移位运算的种类

助记符	功能	源 1	源 2	目标
PSHA Sx、 Sy、 Dz	算术移位	Sx	Sy	Dz
PSHL Sx、 Sy、 Dz	逻辑移位	Sx	Sy	Dz
PSHA #imm、 Dz	带立即数的算术移位	Dz	imm1	Dz
PSHL #imm、 Dz	带立即数的逻辑移位	Dz	imm2	Dz

–32<=imm1<=+32、 –16<=imm2<=+16

表 4.24 移位运算的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器

4.12.1 算术移位运算

(1) 运算功能

ALU 算术移位运算为 32 位精度与 8 位保护位的 40 位运算，基本在寄存器之间执行。源操作数中无保护位时，将符号位作为保护位转存；目标操作数中无保护位时，将运算结果的低 32 位保存到目标寄存器。

本算术移位中，源 1 操作数和目标操作数的所有位都有效。指定移位量的源 2 操作数为整数数据。由寄存器或立即操作数指定源 2 操作数。有效移位量为 – 32 ～ +32。此处，负值代表向右移位、正值代表向左移位。源 2 操作数可指定 – 64 ～ +63，但有效移位量为 – 32 ～ +32，因此，指定无效值时，不能保证运算结果。用立即数指定移位量时，源 1 操作数必须与目标操作数相同。与定点运算相同，在流水线流程最后的 DSP 阶段执行此运算。

执行算术移位运算时，基本根据运算结果更新 DSR 寄存器的 DC、N、Z、V、GT 位，这一点与 ALU 定点运算相同。为带条件的指令时，即使指定条件成立、并已执行指令，也不更新状态位；为无条件的指令时，总是根据运算结果更新。

算术移位运算流程如**图 4.13** 所示。

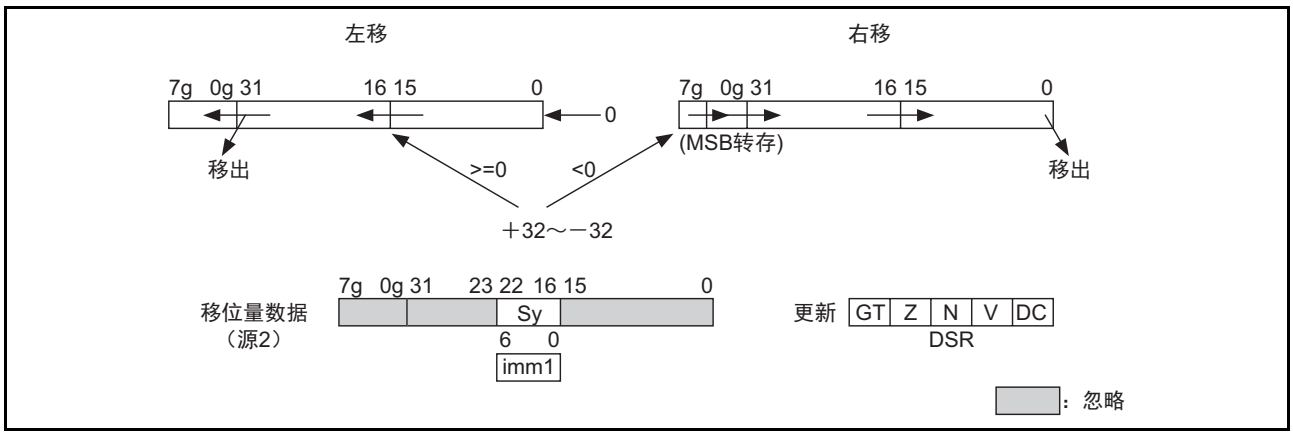


图 4.13 算术移位运算流程

(2) DC 位

根据 CS 位所指定的模式，如下更新 DC 位。

(a) 进位 / 借位模式： CS2 ～ CS0 ＝ 000

DC 位为运算结果最后移位时移出位的值。

(b) 负值模式： CS2 ～ CS0 ＝ 001

结果为负值时， DC 位置 1；为零或正值时，清 0。该模式的 DC 位的值与 N 位的值相同。

(c) 零值模式： CS2 ～ CS0 ＝ 010

运算结果为零时， DC 位置 1；除此之外清 0。该模式的 DC 位的值与 Z 位的值相同。

(d) 上溢模式： CS2 ～ CS0 ＝ 011

产生上溢时， DC 位置 1。该模式的 DC 位的值与 V 位的值相同。

(e) 带符号的 “大于” 模式： CS2 ～ CS0 ＝ 100

DC 位总是清 0。该模式的 DC 位的值与 GT 位的值相同。

(f) 带符号的 “大于等于” 模式： CS2 ～ CS0 ＝ 101

DC 位总是清 0。

(3) 状态位

状态位的更新如下：
N 位的值与 ALU 定点算术运算结果相同，运算结果为负值时， N 位置 1；为零或正值时，清 0。
Z 位的值与 ALU 定点算术运算结果相同，运算结果为零时， Z 位置 1；除此之外清 0。
V 位的值与 ALU 定点算术运算结果相同，产生上溢时， V 位置 1。
GT 位总是清 0。

(4) 上溢防止功能 （饱和运算）

将 SR 寄存器的 S 位置 1，在 DSP 单元执行的算术移位运算中可执行上溢防止功能。运算结果上溢时，保存最大值 （正）或最小值 （负）。

4.12.2 逻辑移位运算

(1) 运算功能

逻辑移位运算使用源 1 操作数和目标操作数的高位字。与 ALU 逻辑运算相同，忽略操作数的保护位和低位字。指定移位量的源 2 操作数为整数数据。由寄存器或立即操作数指定源 2 操作数。有效移位量为 –16 ～ +16。此处，负值代表向右移位、正值代表向左移位。源 2 操作数可指定 –32 ～ +31，但有效移位量为 –16 ～ +16，因此，如果指定无效值时，不能保证运算结果。由立即数指定移位量时，源 1 操作数必须与目标操作数相同。在流水线流程最后的 DSP 阶段执行此运算。

执行逻辑移位运算时，基本根据运算结果更新 DSR 寄存器的 DC、N、Z、V、GT 位，这一点与 ALU 逻辑运算相同。为带条件的指令时，即使指定条件成立、并已执行指令，也不更新状态位；为无条件的指令时，总是根据运算结果更新。

逻辑移位运算流程如**图 4.14** 所示。

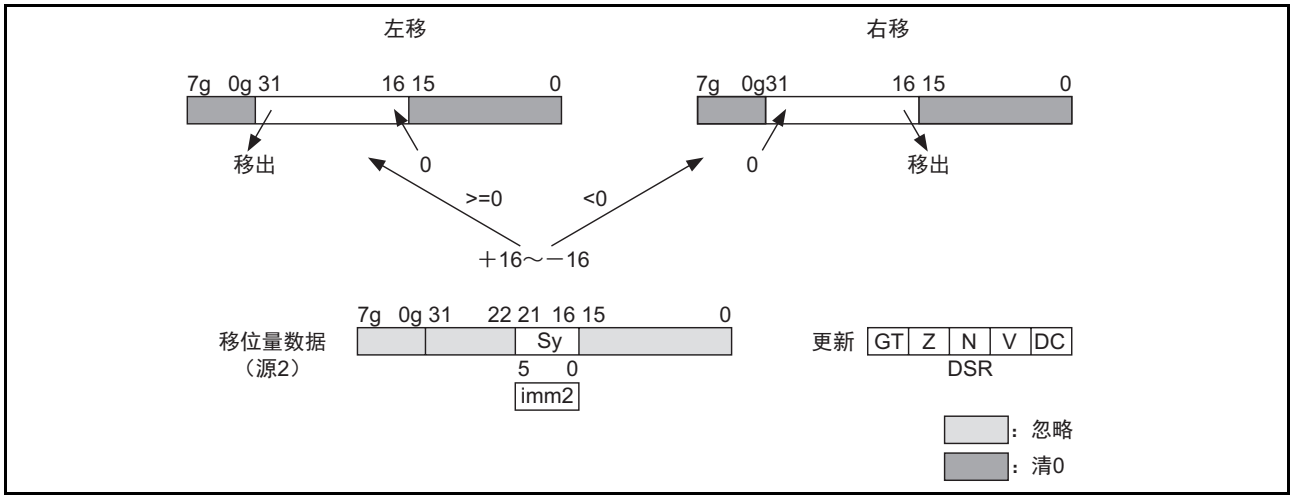


图 4.14 逻辑移位运算流程

(2) DC 位

根据 CS 位所指定的模式，如下更新 DC 位：

(a) 进位 / 借位模式： CS2 ～ CS0 = 000

DC 位为运算结果最后移位时移出位的值。

(b) 负值模式： CS2 ～ CS0 = 001

DC 位保存运算结果的 bit31 的值。该模式的 DC 位的值与 N 位的值相同。

(c) 零值模式： CS2 ～ CS0 = 010

运算结果均为零时， DC 位置 1；除此之外清 0。该模式的 DC 位的值与 Z 位的值相同。

(d) 上溢模式： CS2 ～ CS0 = 011

DC 位总是清 0。该模式的 DC 位的值与 V 位的值相同。

(e) 带符号的 “大于” 模式： CS2 ～ CS0 = 100

DC 位总是清 0。该模式的 DC 位的值与 GT 位的值相同。

(f) 带符号的 “大于等于” 模式： CS2 ～ CS0 = 101

DC 位总是清 0。

(3) 状态位

状态位的更新如下：

- N位的值与ALU逻辑运算结果相同，保存运算结果的bit31的值。
- Z位的值与ALU逻辑运算结果相同，运算结果均为零时， Z位置1；除此之外清0。
- V位总是清0。
- GT位总是清0。

4.13 MSB 检测指令

(1) 运算功能

MSB 检测指令 （PDMSB: Most Significant Bit detection）用来计算将数据正规化时的移位量。

与 ALU 整数运算相同，基本运算结果为高 16 位精度和 8 位保护位的 24 位有效。目标操作数为无保护位的寄存器时，保存到目标寄存器的高 16 位。

MSB 检测指令的对象为源操作数的所有位，可求出整数数据的运算结果，这是因为数据正规化时的移位量必须使用算术移位运算的整数数据。与定点运算相同，在流水线流程最后的 DSP 阶段执行此运算。

执行 PDMSB 指令时，基本根据运算结果更新 DSR 寄存器的 DC、 N、 Z、 V、 GT 位。为带条件的指令时，即使指定条件成立、并已执行指令，也不更新状态位；为无条件的指令时，总是根据运算结果更新。

MSB 检测指令流程如**图 4.15** 所示，源数据与目标数据之间的关系如**表 4.25** 所示。

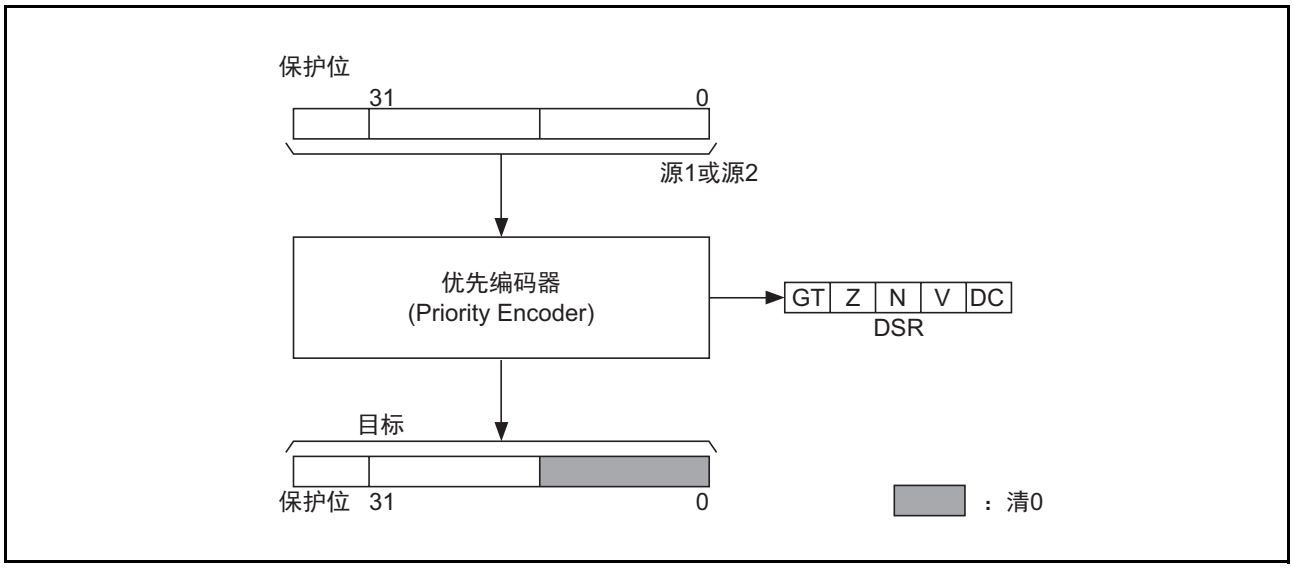


图 4.15 MSB 检测流程

表 4.25 源数据与目标结果之间的关系

源数据																结 果								
保护位						高位字							低位字				保护	高位字							
7g	6g	– – –		1g	0g	31	30	29	28	– – –			3	2	1	0	7g～0g	31～22	21	20	19	18	17	16	10进制数
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'、无影响。

(2) 指令与操作数

MSB 检测指令的种类如**表 4.26** 所示。各操作数与寄存器的对应关系，与 ALU 定点运算相同，如**表 4.27** 所示。

表 4.26 MSB 检测的种类

助记符	功能	源 1	源 2	目标
PDMSB	MSB 检测	Sx	—	Dz
		—	Sy	Dz

表 4.27 MSB 检测的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器

(3) DC 位

根据 CS 位所指定的模式，如下更新 DC 位：

(a) 进位 / 借位模式： CS2 ～ CS0=000

DC 位总是清 0。

(b) 负值模式： CS2 ～ CS0=001

运算结果为负值时， DC 位置 1 ；为零或正值时，清 0。该模式的 DC 位的值与 N 位的值相同。

(c) 零值模式： CS2 ～ CS0=010

运算结果为零时， DC 位置 1 ；除此之外清 0。该模式的 DC 位的值与 Z 位的值相同。

(d) 上溢模式： CS2 ～ CS0=011

DC 位总是清 0。该模式的 DC 位的值与 V 位的值相同。

(e) 带符号的 “大于” 模式： CS2 ～ CS0=100

运算结果为正值时， DC 位置 1 ；除此之外清 0。该模式的 DC 位的值与 GT 位的值相同。

(f) 带符号的 “大于等于” 模式： CS2 ～ CS0=101

运算结果为正值或零时， DC 位置 1 ；除此之外清 0。

(4) 状态位

状态位的更新如下：

- N位的值与ALU整数运算结果相同，运算结果为负值时， N位置1；为零或正值时，清0。
- Z位的值与ALU整数运算结果相同，运算结果为零时， Z位置1；除此之外清0。
- V位总是清0。
- GT位的值与ALU整数运算结果相同，运算结果为正值时， GT位置1；除此之外清0。

4.14 舍入处理

(1) 运算功能

SH-DSP 具有将 32 位数值舍入为 16 位的功能。有保护位时，将 40 位数值舍入为 24 位。如果执行舍入指令，则给源操作数加 H'0000 8000，然后将低位字清 0。

舍入处理使用源操作数和目标操作数的所有位数据。与定点运算相同，在流水线流程最后的 DSP 阶段执行该运算。

舍入处理指令为无条件的指令，因此，总是根据运算结果更新 DSR 寄存器的 DC、 N、 Z、 V、 GT 位。

舍入处理流程如图 4.16 所示，舍入处理的定义如图 4.17 所示。

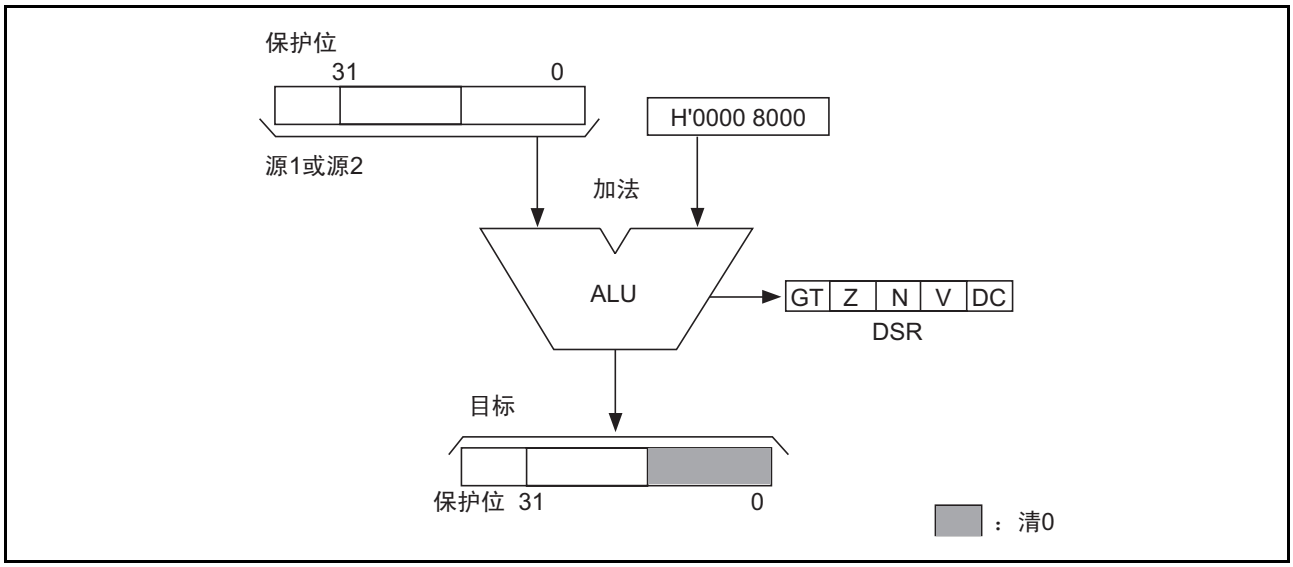


图 4.16 舍入处理流程

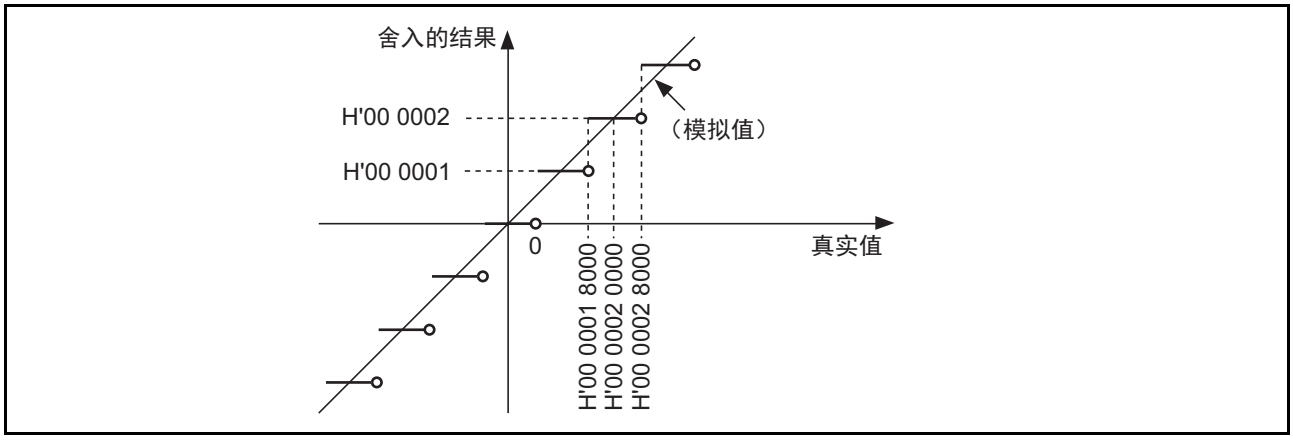


图 4.17 舍入处理的定义

(2) 指令与操作数

指令的种类如表 4.28 所示；操作数与寄存器的对应关系，与 ALU 定点运算相同，如表 4.29 所示。

表 4.28 定点乘法运算的种类

助记符	功能	源 1	源 2	目标
PRND	舍入处理	Sx	—	Dz
		—	Sy	Dz

表 4.29 定点乘法运算的操作数与寄存器的对应关系

操作数	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：可用于操作数的寄存器

(3) DC 位

根据 CS 位所指定的模式，如下更新 DC 位。状态位的更新与 ALU 定点算术运算相同。

(a) 进位 / 借位模式： CS2 ～ CS0=000

如果运算结果从 MSB 位产生进位或借位， DC 位置 1 ；除此之外清 0。

(b) 负值模式： CS2 ～ CS0=001

运算结果为负值时， DC 位置 1 ；为零或正值时，清 0。该模式的 DC 位的值与 N 位的值相同。

(c) 零值模式： CS2 ～ CS0=010

运算结果为零时， DC 位置 1 ；除此之外清 0。该模式的 DC 位的值与 Z 位的值相同。

(d) 上溢模式： CS2 ～ CS0=011

产生上溢时， DC 位置 1 ；除此之外清 0。该模式的 DC 位的值与 V 位的值相同。

(e) 带符号的 “大于” 模式： CS2 ～ CS0=100

运算结果为正值时， DC 位置 1 ；除此之外清 0。该模式的 DC 位的值与 GT 位的值相同。

(f) 带符号的 “大于等于” 模式： CS2 ～ CS0=101

运算结果为正值或零时， DC 位置 1 ；除此之外清 0。

(4) 状态位

状态位如下更新，状态位的更新与 ALU 定点算术运算相同。

- N位的值与ALU定点算术运算结果相同，运算结果为负值时， N位置1；为零或正值时，清0。
- Z位的值与ALU定点算术运算结果相同，运算结果为零时， Z位置1；除此之外清0。
- V位的值与ALU定点算术运算结果相同，产生上溢时， V位置1；除此之外清0。
- GT位的值与ALU定点算术运算结果相同，运算结果为正值时， GT位置1；除此之外清0。

(5) 上溢防止功能 （饱和运算）

如果将 SR 寄存器的 S 位置 1，则对在 DSP 单元执行的所有舍入处理，执行上溢防止功能。运算结果上溢时，保存最大值 （正）或最小值 （负）。

4.15 状态选择位 （CS）与 DSP 状态位 （DC）

DSP 类指令有无条件的指令和带条件的指令。无条件的指令的执行与 DSP 状态位 （DC）无关，执行带条件的指令时，需先判断 DC 位然后再决定是否执行。为无条件的指令时，根据 ALU 运算或移位运算的结果，更新 DSR 寄存器的 DC 位及状态位 （N、 Z、 V、 GT）；为带条件的指令时，与是否执行无关，不更新 DC 位及状态位 （N、 Z、 V、 GT）。按照状态选择位 （CS）的指定，更新 DC 位。根据算术运算、逻辑运算、算术移位、逻辑移位，更新结果也各不相同。 CS 位与 DC 位之间的关系如**表 4.30** 所示。

表 4.30 状态选择位 （CS）与 DSP 状态位 （DC）

CS 位			状态模式	说明
2	1	0		
0	0	0	进位 / 借位模式	ALU 算术运算的结果产生进位或借位时， DC 位置 1 ；除此之外清 0。 逻辑运算时， DC 位总是清 0。 移位运算 （PSHA、 PSHL 指令）时，将最后移位时被移出的位转存到 DC 位。
0	0	1	负值模式	ALU 算术运算或算术移位 （PSHA）运算时，包括保护位在内，运算结果的 MSB 位被转存到 DC 位。 ALU 逻辑运算或逻辑移位 （PSHL）运算时，除保护位之外，运算结果的 MSB 位被转存到 DC 位。
0	1	0	零值模式	ALU 运算或移位运算结果均为零 （0）时， DC 位置 1 ；除此之外清 0。
0	1	1	上溢模式	ALU 算术运算或算术移位 （PSHA）运算时，除保护位之外，运算结果超过目标寄存器值的范围时， DC 位置 1 ；除此之外清 0。 ALU 逻辑运算或逻辑移位 （PSHL）运算时， DC 位总是清 0。
1	0	0	带符号的 “大于” 模式	该模式类似于带符号的 “大于等于” 模式。运算结果为零 （0）时， DC 位清 0。即使包括保护位，运算结果仍超出可表示范围，设定 VR 为真状态时，执行如下计算： DC 位 = ～ {(N 位 ^ VR) | Z 位 } ； 算术运算时 DC 位 = 0 ； 逻辑运算时
1	0	1	带符号的 “大于等于” 模式	ALU 算术运算或算术移位 （PSHA）运算，并且结果没有上溢时，为负值模式的 DC 位取反后的值。即使包括保护位，运算结果仍超出可表示范围，则为与负值模式 DC 位相同的值。 ALU 逻辑运算或逻辑移位 （PSHL）运算时， DC 位总是清 0。即使包括保护位，如果运算结果超出可表示范围，设定 VR 为真状态时，执行如下计算： DC 位 = ～ (N 位 ^ VR) ；算术运算时 DC 位 = 0 ；逻辑运算时
1	1	0	保留码	
1	1	1		

4.16 上溢防止功能 （饱和运算）

上溢防止功能 （饱和运算）由 SR 寄存器的 S 位指定。该功能对 DSP 单元执行的所有算术运算及使用 CPU 类指令执行的乘法累加运算有效。除保护位之外，运算结果超出可表示的 2 的补码范围时，产生上溢。

DSP 类指令的定点算术运算中上溢的定义如**表 4.31** 所示；整数算术运算中上溢的定义如**表 4.32** 所示。

表 4.31 定点算术运算中上溢的定义

符号	上溢状态	最大值 / 最小值	16 进制表示
正	结果＞ $1–2^{-31}$	$1–2^{-31}$	007FFFFFFF
负	结果＜ –1	–1	FF80000000

表 4.32 整数算术运算中上溢的定义

符号	上溢状态	最大值 / 最小值	16 进制表示
正	结果＞ $2^{15}–1$	$2^{15}–1$	007FFF****
负	结果＜ $–2^{15}$	$–2^{15}$	FF8000****

【注】 *： ‘Don't care’、无影响。

如果指定上溢防止功能，则不会产生上溢，此时，上溢位 （V）不置位。由 CS 位指定上溢模式时， DC 位也不置位。

CPU 类指令的乘加指令 （MAC）使用 64 位寄存器 （MACH、 MACL）运算，因此，上溢值、最大值及最小值都与 DSP 类指令时的值不同。

4.17 数据传送

SH-DSP 的 DSP 单元在 DSP 寄存器与内部存储器之间最多可同时并发传送 2 个数据。 SH-DSP 有以下 3 种数据传送类型：

1. X、 Y存储器数据传送： 使用X总线和Y总线与X、 Y存储器进行数据传送
   a. 双数据传送：仅执行数据传送，也可只对任意一方传送
   b. 并发数据传送：与ALU运算或乘法运算并发处理的同时，进行数据传送
2. 单数据传送：使用主总线，与内部存储器进行数据传送

数据传送指令时，不更新 DSR 寄存器的状态位。

各功能如**表 4.33** 所示。

表 4.33 数据传送功能

类型	使用的总线	传送数据长度	与 ALU 运算的并发处理	数据传送的并发处理	指令长度
X、Y 存储器数据传送	X 总线 Y 总线	16 位	无 （双）	无 （X 总线或 Y 总线）	16 位
				有 （X 总线和 Y 总线）	16 位
			有 （并发）	无 （X 总线或 Y 总线）	32 位
				有 （X 总线和 Y 总线）	32 位
单数据传送	主总线	32 位 16 位	无	无	16 位

4.17.1 X、Y 存储器数据传送

X、Y 存储器数据传送可同时并发传送 2 个数据，还可同时并发执行数据传送与 DSP 数据运算。同时并发执行 DSP 数据运算和传送时，需要 32 位操作码，这叫做并发数据传送。仅执行 X、Y 存储器数据传送时，操作码为 16 位，这叫做双数据传送。

数据传送有 X 存储器数据传送和 Y 存储器数据传送。X 存储器数据加载到 X0 或 X1 寄存器，Y 存储器数据则加载到 Y0 或 Y1 寄存器。X0、X1、Y0、Y1 寄存器为目标寄存器。将数据传送到目标寄存器的高位字，低位字自动清 0。A0 或 A1 寄存器可用作源寄存器，将数据存储到 X、Y 寄存器。此数据传送均为字数据（16 位）。从源寄存器的高位字传送数据。

即使在同时并发执行的运算指令中指定带条件的指令，也不会影响数据传送指令。

X、Y 存储器数据传送仅存取 X、Y 存储器，不可存取其他存储区域。

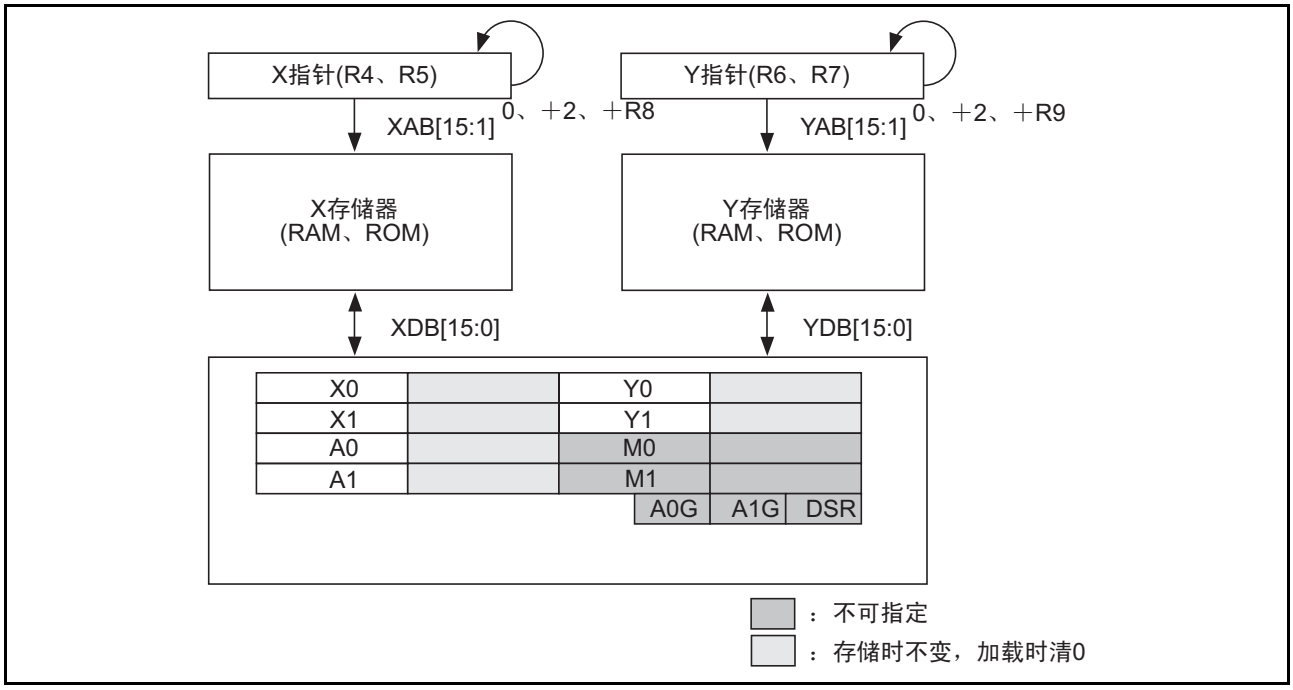


图 4.18 X、Y 存储器的数据传送流程

4.17.2 单数据传送

单数据传送仅传送 1 个数据，操作码为 16 位。单数据传送不可与 ALU 运算同时并发处理。存取 X 存储器的 X 指针和追加的 2 个指针有效，Y 指针无效。与 CPU 类指令相同，单数据传送可存取包括外部区域的所有存储区域。除 DSR 寄存器 * 之外的 DSP 寄存器可指定为源操作数、目标操作数。保护位寄存器、A0G、A1G 可作为独立的寄存器指定为操作数。单数据传送时，使用主总线代替 X 总线和 Y 总线，因此，在主总线会产生数据传送与取指令的竞争。

单数据传送使用字数据和长字数据。字数据传送时，寄存器的高位字有效。加载数据时，将数据加载到寄存器的高位字，低位字自动清 0。有保护位时，将符号位扩展后保存到保护位。从寄存器存储时，存储高位字。

传送长字时，32 位有效。加载时，如果有保护位，则将符号位扩展后保存到保护位。

从位寄存器存储保护时，符号扩展为高 24 位后，读取到主总线。保护位寄存器、A0G、A1G 寄存器作为 MOVS.W 指令的目标寄存器加载字数据时，将低位字节写入寄存器。

【注】 * DSR 寄存器被定义为系统寄存器，因此可使用 LDS、STS 指令传送数据。

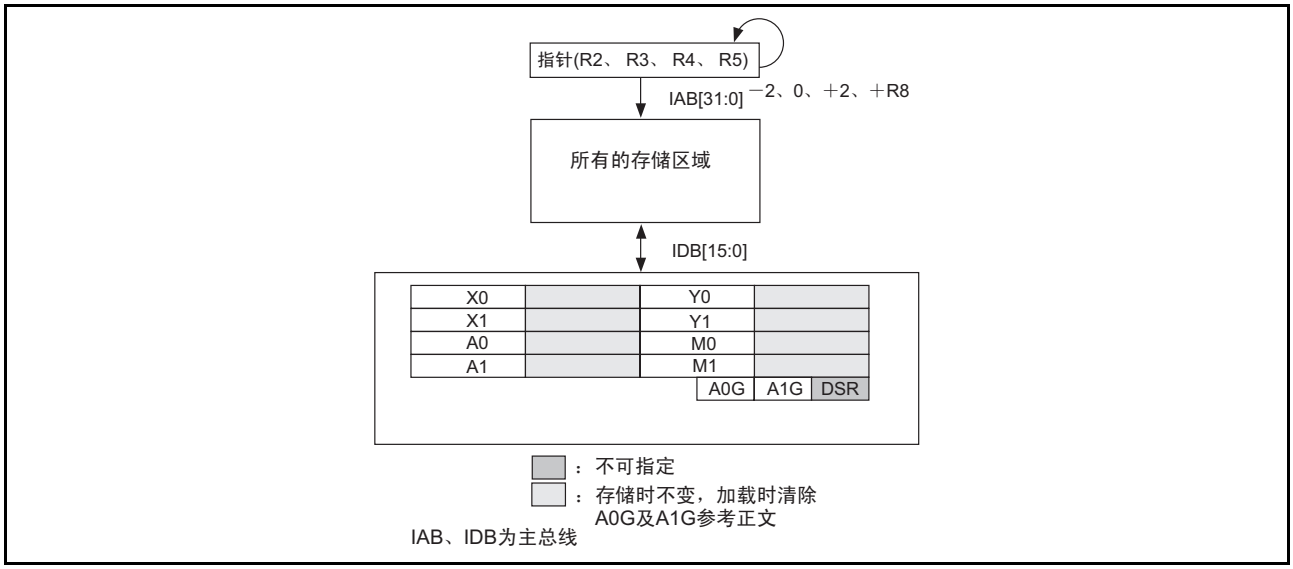


图 4.19 单数据传送流程 （字）

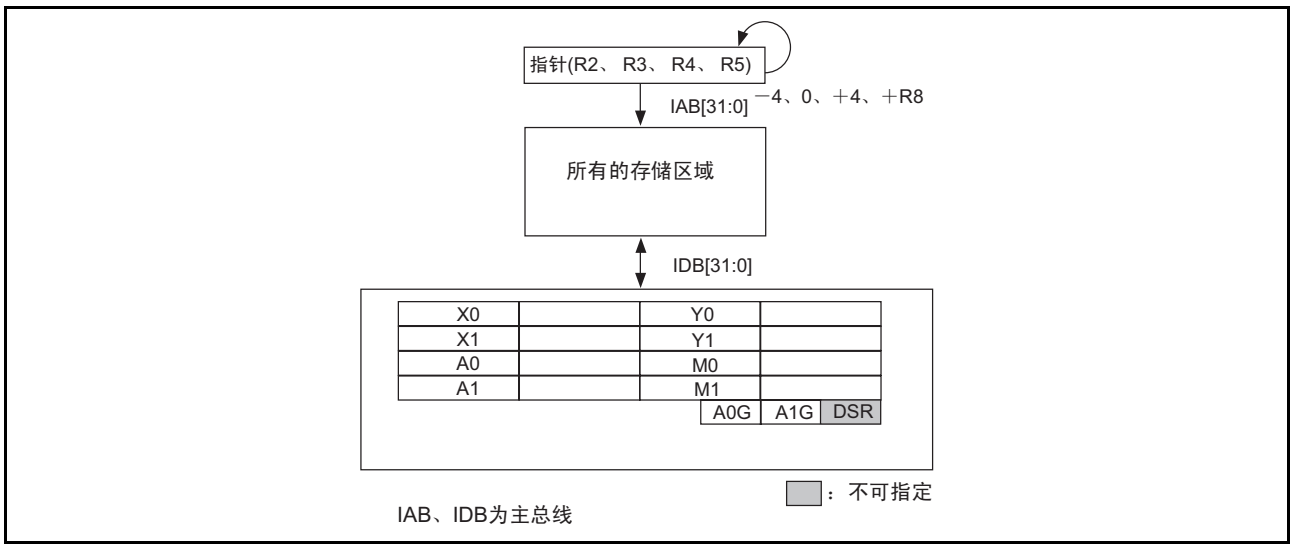


图 4.20 单数据传送流程 （长字）

在流水线的 MA 阶段执行数据传送， DSP 阶段执行 DSP 运算。如果数据运算指令后的下一个指令行为存储运算功能的指令，数据运算指令尚未结束就开始下一条数据存储指令，因此插入 1 个停止周期。在数据运算指令和数据传送指令之间插入一个指令，可避免此开销周期。此例如图 4.21 所示。

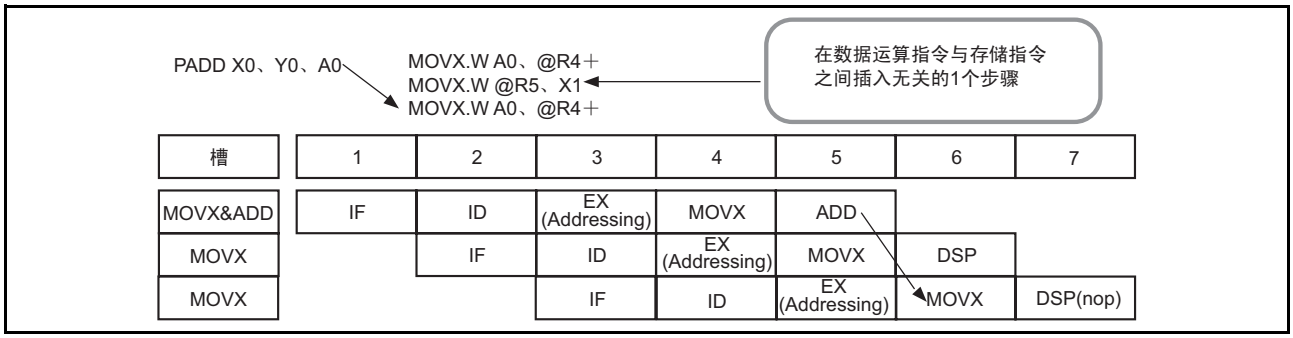


图 4.21 运算与数据存储的指令执行例

4.18 操作数竞争

如果至少 2 条并发处理指令对目标操作数指定相同寄存器，则产生数据竞争。数据竞争可考虑以下 3 种情况：

1. ALU运算和乘法运算指定了相同的目标操作数 （Du、 Dg）
2. X存储器加载与ALU运算指定了相同的目标操作数 （Dx、 Du、 Dz）
3. Y存储器加载与ALU运算指定了相同的目标操作数 （Dy、 Du、 Dz）

如果产生竞争，则不能保证其结果。产生竞争的操作数与寄存器的对应关系如**表 4.34** 所示。

有可检测此竞争的汇编程序，选择功能后使用汇编程序。

表 4.34 产生竞争的操作数与寄存器的对应关系

		DSP寄存器							
		X0	X1	Y0	Y1	M0	M1	A0	A1
X存储器加载	Ax								
	Ix								
	Dx	(*)	(*)						
Y存储器加载	Ay								
	Iy								
	Dy			(*)	(*)				
6个操作数ALU运算	Sx	*	*					*	*
	Sy			*	*	*	*		
	Du	(*)		(*)				(*)	(*)
3个操作数乘法	Se	*	*	*					*
	Sf	*		*	*				*
	Dg					*	*	(*)	(*)
3个操作数ALU运算	Sx	*	*					*	*
	Sy			*	*	*	*		
	Dz	(*)	(*)	(*)	(*)	*	*	*	*

↑ ↑ ↑ ↑ ↑ ↑

(Dx、Du、Dz 的竞争)(Dy、Du、Dz 的竞争) (Du、Dg 的竞争)

【注】 * 可对操作数设定的寄存器

(*) 操作数竞争

4.19 DSP 重复 （循环）控制

SH-DSP 的特殊结构可高效执行重复 （循环）控制。通过 SETRC 指令，将重复次数保存到重复计数器（RC、 12 位），并设定反复执行重复程序 （循环）直至 RC 变为 1 的执行模式。重复运行结束后， RC 的内容变为 0。

重复起始地址寄存器 （RS）保存重复循环的起始地址，重复结束寄存器 （RE）保存重复结束地址 （有例外。参照 “4.19.1 注意事项”）。重复计数器 （RC）保存重复次数。该重复控制执行顺序如下：

#1 在 RS 寄存器设定重复起始地址。
#2 在 RE 寄存器设定重复结束地址。
#3 在 RC 计数器设定重复次数。
#4 开始执行重复程序 （循环）。

使用以下指令，执行 #1 和 #2。

```
LDRS @（disp,PC）；及
LDRE @（disp,PC）；
```

使用 SETEC 指令，执行 #3 和 #4。 SETEC 指令的操作数由立即数或通用寄存器指定重复次数。

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

#imm 为 8 位， RC 计数器为 12 位，因此，要在 RC 计数器指定大于等于 256 的数值时，使用 Rm 寄存器设定。程序例如下所示：

```
                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;
```

该重复指令有以下限制：

1. SETRC指令与重复程序 （循环）的第一条指令之间至少要有1条指令。
2. 执行LDRS和LDRE指令后，执行SETRC指令。
3. 重复程序 （循环）大于等于4条指令时，如果重复起始地址 （上述例中为instr1的地址）不在长字边界，每重复一次都会产生1个停止周期 （执行等待周期）。
4. 重复程序 （循环）小于等于3条指令时，不可使用转移指令（BRA、 BSR、 BT、 BF、 BT/S、 BF/S、 BSRF、 RTS、 BRAF、 RTE、 JSR、 JMP）、重复控制指令（SETRC、 LDRS、 LDRE）、SR、 RS、 RE的加载指令及TRAPA。如果使用上述指令，则启动错误异常处理，并将表4.35所示的地址值压入R15指向的堆栈区。

表 4.35 压入的 PC 值 （1）

条件	位置	压入地址
RC>=2	任意	RptStart
RC=1	任意	非法指令的程序地址

5. 重复程序 （循环）大于等于4条指令时，转移指令 （BRA、 BSR、 BT、 BF、 BT/S、 BF/S、 BSRF、 RTS、 BRAF、 RTE、 JSR、 JMP）、重复控制指令（SETRC、LDRS、LDRE）、SR、RS、RE的加载指令及TRAPA不可用于重复程序 （循环）内的最后3条指令。如果使用上述指令，则启动错误异常处理，并将表4.36所示的地址值压入R15指向的堆栈区。重复控制指令 （SETRC、 LDRS、 LDRE）、SR、 RS、及RE的加载指令，不可用于重复模块的其他位置。如果使用上述指令，则不能正常运行。

表 4.36 压入的 PC 值 （2）

条件	位置	压入地址
RC>=2	instr n–2	非法指令的程序地址
	instr n–1	RptStart–4
	instr n	RptStart–2
RC=1	任意	非法指令的程序地址

6. 重复程序 （循环）小于等于3条指令时， PC相对指令 （MOVA （disp、 PC）、 R0等）仅可用于重复程序 （循环）的第1条指令 （上述例中为instr1）。
7. 重复程序 （循环）大于等于4条指令时， PC相对指令 （MOVA （disp、 PC）、 R0等）不可用于重复程序 （循环）的最后2条指令。
8. SH-DSP没有重复有效标志，但是RC计数器为0时，重复无效。 RC计数器不为0、且PC计数器与RE寄存器的内容匹配时，重复开始。 RC计数器设定为0时，重复程序 （循环）无效，但仅执行1次重复模块，与RC为1时相同，不返回重复程序 （循环）起始指令。 RC计数器设定为1时，仅执行1次重复模块，不返回重复程序 （循环）起始指令，但RC计数器变为0。
9. 重复程序 （循环）大于等于4条指令时，不可将重复结束地址及其前两个地址 （上述例中为instr n–2的地址）指定为转移指令的转移目标地址。如果执行，则重复控制不可正常运行。
10. 重复过程中，中断有所限制。详细内容参照图4.22，该图中各种情况的流程表示EX的各阶段。通常，在指令的EX阶段结束后，直接开始执行中断或总线错误异常的第一个EX阶段，图中表示为“A”。但，到下一个instr0的EX阶段，仅继续执行总线错误异常，指定为 “B”。在instr1的EX阶段，由于 “C”，中断和总线异常都不可继续执行，仅可继续instr2的EX阶段。

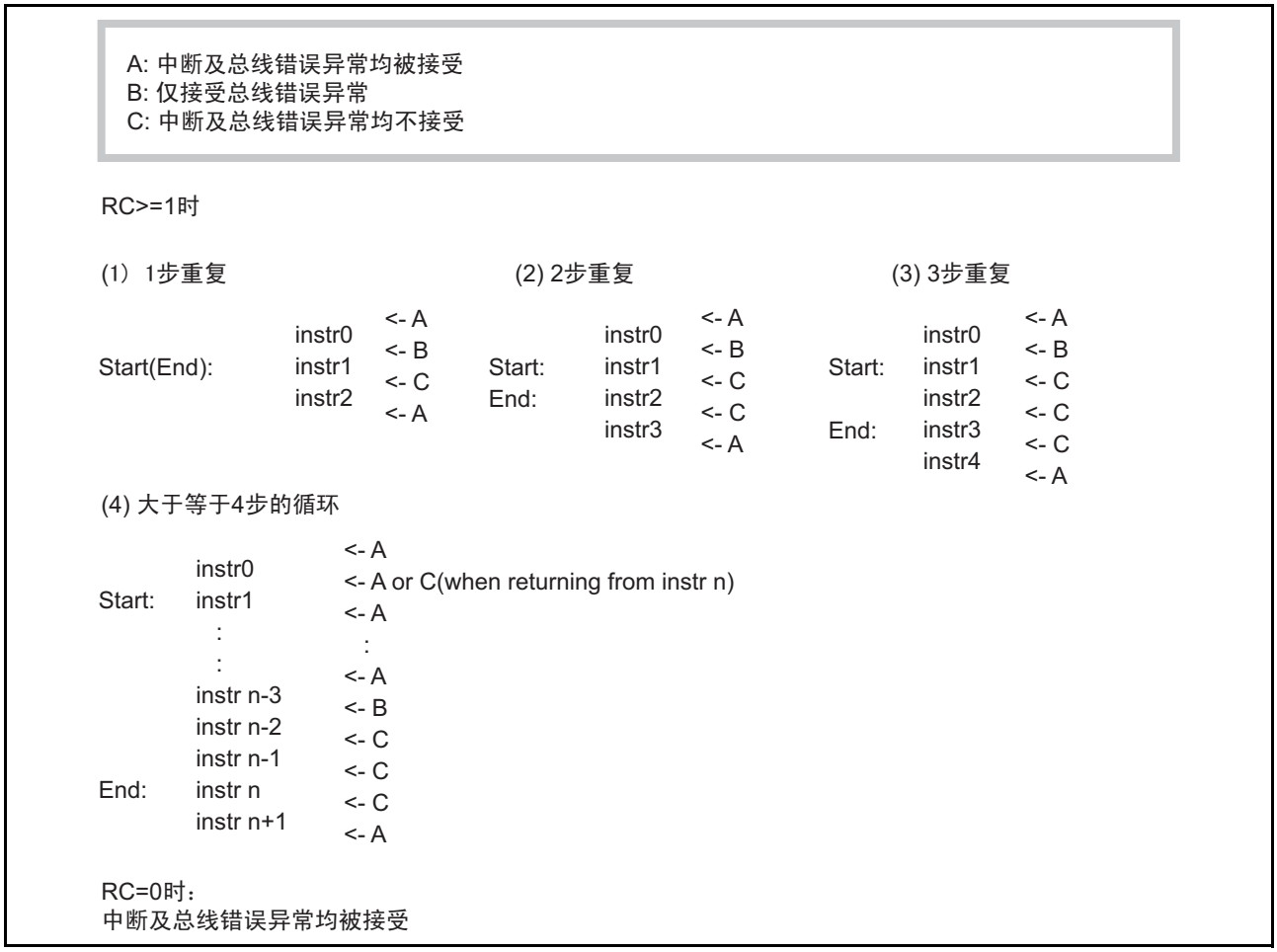


图 4.22 在重复模块接受中断的限制

4.19.1 注意事项

(1) 实际编程时

重复起始寄存器 （RS）和重复结束寄存器 （RE）分别保存重复起始地址和重复结束地址。此寄存器中保存的地址根据重复程序 （循环）内的指令数而变化。变化规则如下：

Repeat_Start：重复起始指令的地址
Repeat_Start0：重复结束指令的前 1 条指令的地址
Repeat_Start3：重复结束指令的 3 条前的指令的地址

表 4.37 RS 及 RE 设定规则

	重复程序 （循环）内的指令数			
	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

以本表为基础，各种情形下的实际重复程序 （循环）编程例如下所示：

情形 1：1 条重复指令时

```
                LDRS    RptStart0+8;        (RptStart)
                LDRE    RptStart0+4;        (RptStart)
                SETRC   RptCount;
                   - - - -
RptStart0:   instr0;
RptStart:    instr1;                              重复指令
             instr2;
```

情形 2：2 条重复指令时

```
                LDRS    RptStart0+6;          (RptStart)
                LDRE    RptStart0+4;          (RptEnd)
                SETRC   RptCount;
                   - - - -
RptStart0:  instr0;
RptStart:    instr1;                             重复指令 1
RptEnd:     instr2;                              重复指令 2
                instr3;
```

情形 3：3 条重复指令时

```
                LDRS    RptStart0+4;            (RptStart)
                LDRE    RptStart0+4;            (RptEnd)
                SETRC   RptCount;
                   - - - -
RptStart0: instr0;
RptStart:  instr1;                                重复指令 1
           instr2;                                重复指令 2
RptEnd:    instr3;                                重复指令 3
           instr4;
```

情形 4：大于等于 4 条重复指令时

```
                    LDRS     RptStart;
                    LDRE     RptEnd3+4;              (RptEnd)
                    SETRC    RptCount;
                        - - - -
RptStart0:  instr0;
RptStart:   instr1;                                  重复指令 1
            instr2;                                  重复指令 2
            instr3;                                  重复指令 3
-----------------------------------------------
RptEnd3:  instr N–3;                                 重复指令 N-3
          instr N–2;                                 重复指令 N-2
          instr N–1;                                 重复指令 N-1
RptEnd:   instr N;                                   重复指令 N
          instr N+1;
```

上述示例中，该重复程序 （循环）顺序可作为编程的模板使用。使用扩展指令 “REPEAT”，可简化此复杂的命名和偏移的问题，具体参照 (2)。

(2) 扩展指令 REPEAT

使用扩展指令 REPEAT，可简化表 4.36 及注 1 所述的命名和偏移。使用的名称如下所示：

RptStart: 重复程序 （循环）起始指令的地址

RptEnd: 重复程序 （循环）结束指令的地址

PptCount: 重复次数立即数

该指令如下使用：

Repeat count 可指定为立即数 #imm 或寄存器间接值 Rn。

情形 1：1 条重复指令时

```
REPEAT  RptStart,  RptStart,  RptCount
            - - - -
          instr0;
RptStart: instr1;                                  重复指令 1
          instr2;
```

情形 2：2 条重复指令时

```
REPEAT  RptStart,  RptEnd,  RptCount
              - - - -
          instr0;
RptStart: instr1;                    重复指令 1
RptEnd:   instr2;                    重复指令 2
```

情形 3：3 条重复指令时

```
REPEAT  RptStart,  RptEnd,  RptCount
            - - - -
          instr0;
RptStart: instr1;                                   重复指令 1
          instr2;                                   重复指令 2
RptEnd:   instr3;                                   重复指令 3
```

情形 4：大于等于 4 条重复指令时

```
REPEAT  RptStart,  RptEnd,  RptCount
              - - - -
            instr0;
RptStart:   instr1;                                  重复指令 1
            instr2;                                  重复指令 2
            instr3;                                  重复指令 3
-----------------------------------------------
            instr N-3;                               重复指令 N-3
            instr N-2;                               重复指令 N-2
            instr N-1;                               重复指令 N-1
RptEnd:     instr N;                                 重复指令 N
            instr N+1;
```

各情形时的扩展结果与 (1) 的情形编号相对应。

4.20 带条件的指令与数据传送

数据运算指令包含无条件的指令和带条件的指令。两者均可指定并发执行的数据传送指令，即使条件不成立也不会影响数据传送指令，总是执行数据传送指令。

带条件的指令与数据传送的例子如**图 4.23** 所示。

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

［条件为真时］
执行前：X0=H'33333333、 Y0=H'55555555、 A0=H'123456789A、
        R4=H'00008000、 R6=H'00008232、 R1=H'00000004
        (R4)=H'1111、 (R6)=H'2222
执行后：X0=H'11110000、 Y0=H'55555555、 A0=H'0088888888、
        R4=H'00008002、 R6=H'00008236、 R1=H'00000004
        (R4)=H'1111、 (R6)=H'1234

［条件为假时］
执行前：X0=H'33333333、 Y0=H'55555555、 A0=H'123456789A、
        R4=H'00008000、 R6=H'00008232、 R1=H'00000004
        (R4)=H'1111、 (R6)=H'2222
执行后：X0=H'11110000、 Y0=H'55555555、 A0=H'123456789A、
        R4=H'00008002、 R6=H'00008236、 R1=H'00000004
        (R4)=H'1111、 (R6)=H'1234
```

图 4.23 带条件的指令与数据传送的例子

第 5 章 指令系统

SH-DSP 的指令可分为 3 类：主要在 CPU 内核执行的 CPU 指令、主要在 DSP 单元执行的 DSP 数据传送指令和 DSP 运算指令。 CPU 指令中有几个支持 DSP 功能的指令。指令系统的说明也分为 3 类分别说明。

5.1 CPU 指令的指令系统

CPU 指令的分类如**表 5.1** 所示。

表 5.1 CPU 指令的分类

分类	指令的种类	操作码	功能	适用指令			指令数
				SH-1	SH-2	SH-DSP	
数据传送指令	5	MOV	传送数据 传送立即数 传送外围模块数据 传送结构体数据	○	○	○	39
		MOVA	传送有效地址	○	○	○	
		MOVT	传送 T 位	○	○	○	
		SWAP	交换高位和低位	○	○	○	
		XTRCT	抽出连接寄存器的中间部分	○	○	○	
算术运算指令	21	ADD	2 进制加法	○	○	○	33
		ADDC	带进位的 2 进制加法	○	○	○	
		ADDV	带上溢的 2 进制加法	○	○	○	
		CMP/cond	比较	○	○	○	
		DIV1	除法	○	○	○	
		DIV0S	带符号除法的初始化	○	○	○	
		DIV0U	无符号除法的初始化	○	○	○	
		DMULS	带符号的双精度乘法	—	○	○	
		DMULU	无符号的双精度乘法	—	○	○	
		DT	递减和测试	—	○	○	
		EXTS	符号扩展	○	○	○	
		EXTU	零扩展	○	○	○	
		MAC	乘法累加运算	○	○	○	
			双精度乘法累加运算	—	○	○	
		MUL	双精度乘法	—	○	○	
		MULS	带符号乘法	○	○	○	
		MULU	无符号乘法	○	○	○	
		NEG	符号取反	○	○	○	33
		NEGC	带借位的符号取反	○	○	○	
		SUB	2 进制减法	○	○	○	
		SUBC	带借位的 2 进制减法	○	○	○	
		SUBV	带下溢的 2 进制减法	○	○	○	

分类	指令的种类	操作码	功能	适用指令			指令数
				SH-1	SH-2	SH-DSP	
逻辑运算指令	6	AND	逻辑 “与” 运算	○	○	○	14
		NOT	位取反	○	○	○	
		OR	逻辑 “或” 运算	○	○	○	
		TAS	存储器测试和置位	○	○	○	
		TST	逻辑 “与” 运算的 T 位置位	○	○	○	
		XOR	逻辑 “异或”	○	○	○	
移位指令	10	ROTCL	带 T 位向左循环 1 位	○	○	○	
		ROTCR	带 T 位向右循环 1 位	○	○	○	
		ROTL	向左循环 1 位	○	○	○	
		ROTR	向右循环 1 位	○	○	○	
		SHAL	算术左移 1 位	○	○	○	
		SHAR	算术右移 1 位	○	○	○	
		SHLL	逻辑左移 1 位	○	○	○	
		SHLLn	逻辑左移 n 位	○	○	○	
		SHLR	逻辑右移 1 位	○	○	○	
		SHLRn	逻辑右移 n 位	○	○	○	
转移指令	9	BF	条件转移 （T = 0 时转移）	○	○	○	11
			带条件延迟转移 （T = 0 时转移）	—	○	○	
		BT	条件转移 （T = 1 时转移）	○	○	○	
			带条件延迟转移 （T = 1 时转移）	—	○	○	
		BRA	无条件转移	○	○	○	
		BRAF	无条件转移	—	○	○	
		BSR	转移到子程序过程	○	○	○	
		BSRF	转移到子程序过程	—	○	○	
		JMP	无条件转移	○	○	○	
		JSR	转移到子程序过程	○	○	○	
		RTS	从子程序过程返回	○	○	○	
系统控制指令	14	CLRMAC	清除 MAC 寄存器	○	○	○	71
		CLRT	清除 T 位	○	○	○	
		LDC	加载到控制寄存器	○	○	○	
		LDRE	加载到重复结束寄存器	—	—	○	
		LDRS	加载到重复开始寄存器	—	—	○	
		LDS	加载到系统寄存器	○	○	○	
		NOP	空操作	○	○	○	
系统控制指令	14	RTE	从异常处理返回	○	○	○	71
		SETRC	设定重复次数及重复控制标志	—	—	○	
		SETT	T 位置位	○	○	○	
		SLEEP	转移到低功耗状态	○	○	○	
		STC	从控制寄存器存储	○	○	○	
		STS	从系统寄存器存储	○	○	○	
		TRAPA	陷阱异常处理	○	○	○	
	合计 65						合计 182

按以下格式，分类说明 CPU 指令的操作码、操作及执行状态。

指令	操作	操作码	执行状态	T 位
用助记符表示。 符号说明 OP.Sz SRC、DEST OP：操作码 Sz：长度 SRC：源 DEST：目标 Rm：源寄存器 Rn：目标寄存器 imm：立即数 disp：位移量 *2	表示操作概略。 符号说明 →、←：传送方向 (xx)：存储器操作数 M/Q/T：SR 内的标志位 &：位“与” |：位“或” ∧：位“异或” ～：位“非” ＜＜ n：左移 n 位 ＞＞ n：右移 n 位	按照 MSB←→LSB 的顺序表示。 符号说明 mmmm：源寄存器 nnnn：目标寄存器 0000：R0 0001：R1 …… 1111：R15 iiii：立即数 dddd：位移量	无等待时的值。*1	表示执行指令后 T 位的值。 符号说明 —：不变化

【注】 *1 指令的执行状态

表中所示的执行状态为最小值。实际根据以下条件，指令执行的状态数会增加。

（1）取指令和数据存取产生竞争时

（2）加载指令（存储器→寄存器）的目标寄存器和紧接着的指令所用的寄存器相同时

*2 根据指令的操作数长度等被放大（×1、×2、×4）。

详细内容参照“第 6 章 指令说明”。

5.1.1 数据传送指令

指令		操作	操作码	执行状态	T 位	使用指令		
						SH-1	SH-2	SH-DSP
MOV	#imm,Rn	imm→ 符号扩展 →Rn	1110nnnniiiiiiii	1	—	○	○	○
MOV.W	@(disp,PC),Rn	(disp×2+PC)→ 符号扩展 →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)→ 符号扩展 →Rn	0110nnnnmmmm0000	1	—	○	○	○
MOV.W	@Rm,Rn	(Rm)→ 符号扩展 →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)→ 符号扩展 →Rn、Rm+1→Rm	0110nnnnmmmm0100	1	—	○	○	○
MOV.W	@Rm+,Rn	(Rm)→ 符号扩展 →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)→ 符号扩展 →R0	10000100mmmmdddd	1	—	○	○	○
MOV.W	@(disp,Rm),R0	(disp×2+Rm)→ 符号扩展 →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)→ 符号扩展 →Rn	0000nnnnmmmm1100	1	—	○	○	○
MOV.W	@(R0,Rm),Rn	(R0+Rm)→ 符号扩展 →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)→ 符号扩展 →R0	11000100dddddddd	1	—	○	○	○
MOV.W	@(disp,GBR),R0	(disp×2+GBR)→ 符号扩展 →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→ 交换低位 2 字节的高低字节 →Rn	0110nnnnmmmm1000	1	—	○	○	○
SWAP.W	Rm,Rn	Rm→ 交换高低字 →Rn	0110nnnnmmmm1001	1	—	○	○	○
XTRCT	Rm,Rn	Rm 和 Rn 的中间 32 位 →Rn	0010nnnnmmmm1101	1	—	○	○	○

5.1.2 算术运算指令

指令		操作	操作码	执行状态	T 位	适用指令		
						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、进位 →T	0011nnnnmmmm1110	1	进位	○	○	○
ADDV	Rm,Rn	Rn+Rm→Rn、上溢 →T	0011nnnnmmmm1111	1	上溢	○	○	○
CMP/EQ	#imm,R0	R0 = imm 时 1→T、≠ 时 0→T	10001000iiiiiiii	1	比较结果	○	○	○
CMP/EQ	Rm,Rn	Rn = Rm 时 1→T、≠ 时 0→T	0011nnnnmmmm0000	1	比较结果	○	○	○
CMP/HS	Rm,Rn	无符号 Rn ≥ Rm 时 1→T、 <时 0→T	0011nnnnmmmm0010	1	比较结果	○	○	○
CMP/GE	Rm,Rn	有符号 Rn ≥ Rm 时 1→T、 <时 0→T	0011nnnnmmmm0011	1	比较结果	○	○	○
CMP/HI	Rm,Rn	无符号 Rn > Rm 时 1→T、 ≤时 0→T	0011nnnnmmmm0110	1	比较结果	○	○	○
CMP/GT	Rm,Rn	有符号 Rn > Rm 时 1→T、 ≤时 0→T	0011nnnnmmmm0111	1	比较结果	○	○	○
CMP/PL	Rn	Rn > 0 时 1→T、≤ 0 时 0→T	0100nnnn00010101	1	比较结果	○	○	○
CMP/PZ	Rn	Rn ≥ 0 时 1→T、< 0 时 0→T	0100nnnn00010001	1	比较结果	○	○	○
CMP/STR	Rm,Rn	任意字节相等时 1→T、 不相等时 0→T	0010nnnnmmmm1100	1	比较结果	○	○	○
DIV1	Rm,Rn	单步除法 (Rn÷Rm)	0011nnnnmmmm0100	1	计算结果	○	○	○
DIV0S	Rm,Rn	Rn 的 MSB→Q、Rm 的 MSB→M、 M^Q→T	0010nnnnmmmm0111	1	计算结果	○	○	○
DIV0U		0→M/Q/T	0000000000011001	1	0	○	○	○
DMULS.L	Rm,Rn	带符号 Rn×Rm→MACH、 MACL 32×32→64 位	0011nnnnmmmm1101	2 ～ 4*1	—	—	○	○
DMULU.L	Rm,Rn	无符号 Rn×Rm→MACH、 MACL 32×32→64 位	0011nnnnmmmm0101	2 ～ 4*1	—	—	○	○
DT	Rn	Rn–1→Rn、Rn 为 0 时 1→T Rn 不为 0 时 0→T	0100nnnn00010000	1	比较结果	—	○	○
EXTS.B	Rm,Rn	将 Rm 从字节符号扩展 →Rn	0110nnnnmmmm1110	1	—	○	○	○
EXTS.W	Rm,Rn	将 Rm 从字符号扩展 →Rn	0110nnnnmmmm1111	1	—	○	○	○
EXTU.B	Rm,Rn	将 Rm 从字节零扩展 →Rn	0110nnnnmmmm1100	1	—	○	○	○
EXTU.W	Rm,Rn	将 Rm 从字零扩展 →Rn	0110nnnnmmmm1101	1	—	○	○	○
MAC.L	@Rm+,@Rn+	带符号 (Rn)×(Rm)+MAC→MAC 32×32+64→64 位	0000nnnnmmmm1111	3/(2 ～ 4)*1	—	○	○	○
MAC.W	@Rm+,@Rn+	带符号 (Rn)×(Rm)+MAC→MAC (SH-2) 16×16+64→64 位 (SH-1) 16×16+42→42 位	0100nnnnmmmm1111	3/(2)*1	—	○	○	○
MUL.L	Rm,Rn	Rn×Rm→MACL 32×32→32 位	0000nnnnmmmm0111	2 ～ 4*1	—	○	○	○
MULS.W	Rm,Rn	带符号 Rn×Rm→MAC 16×16→32 位	0010nnnnmmmm1111	1 ～ 3*1	—	○	○	○

指令	操作	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MULU.W Rm,Rn	无符号 Rn×Rm→MAC 16×16→32 位	0010nnnnmmmm1110	1 ～ 3*1	—	○	○	○
NEG Rm,Rn	0–Rm→Rn	0110nnnnmmmm1011	1	—	○	○	○
NEGC Rm,Rn	0–Rm-T→Rn、借位 →T	0110nnnnmmmm1010	1	借位	○	○	○
SUB Rm,Rn	Rn–Rm→Rn	0011nnnnmmmm1000	1	—	○	○	○
SUBC Rm,Rn	Rn–Rm-T→Rn、借位 →T	0011nnnnmmmm1010	1	借位	○	○	○
SUBV Rm,Rn	Rn–Rm→Rn、下溢 →T	0011nnnnmmmm1011	1	下溢	○	○	○

【注】 *1 表示普通执行状态。（ ）内的值表示因与前后指令竞争产生的执行状态。

5.1.3 逻辑运算指令

指令	操作	操作码	执行状态	T 位	适用指令		
					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) 为 0 时 1→T、不为 0 时 0→T。 与 (Rn) 的值无关 1→MSBof(Rn)	0100nnnn00011011	4	测试结果	○	○	○
TST Rm,Rn	Rn & Rm, 结果为 0 时 1→T、不为 0 时 0→T	0010nnnnmmmm1000	1	测试结果	○	○	○
TST #imm,R0	R0 & imm, 结果为 0 时 1→T、不为 0 时 0→T	11001000iiiiiiii	1	测试结果	○	○	○
TST.B #imm, @(R0,GBR)	(R0+GBR)&imm, 结果为 0 时 1→T、不为 0 时 0→T	11001100iiiiiiii	3	测试结果	○	○	○
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 移位指令

指令		操作	操作码	执行状态	T 位	适用指令		
						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 转移指令

指令		操作	操作码	执行状态	T 位	适用指令		
						SH-1	SH-2	SH-DSP
BF	label	T ＝ 0 时 disp×2+PC→ PC、T ＝ 1 时 nop	10001011dddddddd	3/1*2	—	○	○	○
BF/S	label	延迟转移， T ＝ 0 时 disp×2+PC→PC、T ＝ 1 时 nop	10001111dddddddd	2/1*2	—	—	○	○
BT	label	T ＝ 1 时 disp×2+PC→ PC、T ＝ 0 时 nop	10001001dddddddd	3/1*2	—	○	○	○
BT/S	label	延迟转移， T ＝ 1 时 disp×2+PC→ PC、T ＝ 0 时 nop	10001101dddddddd	2/1*2	—	—	○	○
BRA	label	延迟转移， disp×2+PC→ PC	1010dddddddddddd	2	—	○	○	○
BRAF	Rm	延迟转移， Rm+PC→ PC	0000mmmm00100011	2	—	—	○	○
BSR	label	延迟转移，PC→ PR、disp×2+PC→ PC	1011dddddddddddd	2	—	○	○	○
BSRF	Rm	延迟转移， PC→ PR、 Rm+PC→ PC	0000mmmm00000011	2	—	—	○	○
JMP	@Rm	延迟转移， Rm→ PC	0100mmmm00101011	2	—	○	○	○
JSR	@Rm	延迟转移， PC→ PR、 Rm→ PC	0100mmmm00001011	2	—	○	○	○
RTS		延迟转移， PR→ PC	0000000000001011	2	—	○	○	○

【注】 *2 不转移时，为 1 个状态。

5.1.6 系统控制指令

指令		操作	操作码	执行状态	T 位	适用指令		
						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		空操作	0000000000001001	1	—	○	○	○
RTE		延迟转移，堆栈区 →PC/SR	0000000000101011	4	LSB	○	○	○
SETRC	Rm	RE– RS 的运算结果 （重复状态） →RF1、 RF0 Rm[11:0]→RC (SR[27:16])	0100mmmm00010100	1	—	—	—	○

指令	操作	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SETRC #imm	RE–RS 的运算结果 （重复状态） →RF1、 RF0 imm→RC(SR[23:16])、 zeros→SR[27:24]	10000010iiiiiiii	1	1	—	—	○
SETT	1→T	0000000000011000	1	1	○	○	○
SLEEP	睡眠	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→ 堆栈区， (imm ・4+VBR)→PC	11000011iiiiiiii	8	—	○	○	○

【注】 *3 为转移到睡眠状态前的状态数。

［注意事项］

- 指令的执行状态

表中所示的执行状态为最小值。实际根据以下条件，指令执行状态数会增加。

1. 取指令和数据存取产生竞争时
2. 加载指令 （存储器→寄存器）的目标寄存器和紧接着的指令所用的寄存器相同时
3. 转移指令的转移目标地址为地址4n+2

5.1.7 支持 DSP 功能的 CPU 指令

为了支持 DSP 功能，在 CPU 内核的指令中追加了几个系统控制指令。追加了支持重复控制、模寻址的 RS、RE、MOD 寄存器，给 SR 寄存器追加了 RC 计数器，同时，为了存取此寄存器，还追加了 LDC、STC 指令。为了存取 DSP 寄存器的 DSR、 A0、 X0、 X1、 Y0 及 Y1 寄存器，追加了 LDS、 STS 指令。

给 SR 寄存器的重复计数器 （RC、 bit27 ～ 16）追加了设定值的 SETRC 指令。 SETRC 指令的操作数为立即数时，在 SR 寄存器的 bit23 ～ 16 保存 8 位立即数，并将 bit27 ～ 24 清 0。操作数为寄存器时，将寄存器的 bit11 ～ 0 的 12 位保存在 SR 寄存器的 bit27 ～ 16。

对于将重复开始地址、重复结束地址设定在 RS、 RE 寄存器的指令，除 LDC 指令外，还追加了 LDRS、LDRE 指令。

追加的指令如**表 5.2** 所示。

表 5.2 追加的 CPU 指令

指令		操作码	操作	执行状态	T 位
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	—

指令		操作码	操作	执行状态	T 位
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	0100nnnn10010010	Rn–4→Rn、 X1→(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])、 重复标志 →RF1、 RF0	1	—
SETRC	#imm	10000010iiiiiiii	imm→RC(SR[23:16])、 zeros→SR[27:24]、 重复标志 →RF1、 RF0	1	—
LDRS	@(disp,PC)	10001100dddddddd	disp×2+PC→RS	1	—
LDRE	@(disp,PC)	10001110dddddddd	disp×2+PC→RE	1	—

5.2 DSP 数据传送指令的指令系统

DSP 数据传送指令的分类如表 5.3 所示。

表 5.3 DSP 数据传送指令的分类

分类	指令分类	操作码	功能	指令数
双数据传送指令	4	NOPX	X 存储器空操作	14
		MOVX	传送 X 存储器数据	
		NOPY	Y 存储器空操作	
		MOVY	传送 Y 存储器数据	
单数据传送指令	1	MOVS	传送单数据	16
	合计 5			合计 30

数据传送指令分为 2 组：双数据传送和单数据传送。双数据传送时，结合 DSP 运算指令，可执行 DSP 并发处理指令。并发处理指令长 32 位，在 A 字段编入双数据传送指令。非并发处理指令的双数据传送和单数据传送指令为 16 位。

双数据传送时，可同时并发存取 X 存储器和 Y 存储器。分别从 X、 Y 存储器的数据存取各指定一条指令。Ax 指针用于存取 X 存储器， Ay 指针用于存取 Y 存储器。双数据传送时，仅可对 X、 Y 存储器存取。

单数据传送可存取任意区域。单数据传送时， Ax 指针和其他 2 个指针用作 As 指针。

5.2.1 双数据传送指令 （X 存储器数据）

指令	操作	操作数	执行状态	DC 位
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 双数据传送指令 （Y 存储器数据）

指令	操作	操作码	执行状态	DC 位
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 单数据传送指令

指令	操作	操作码	执行状态	DC 位
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】 * 对源操作数 Ds 指定保护位寄存器 A0G、 A1G （8 位寄存器）时，使用符号扩展后的数据。

DSP 数据传送的操作数和寄存器的对应关系如**表 5.4** 所示。CPU 内核的寄存器用作表示存储器地址的指针地址。

表 5.4 DSP 数据传送的操作数和寄存器的对应关系

操作数	SuperH （CPU 内核）寄存器									
	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										

操作数	DSP 寄存器									
	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：可设定的寄存器

5.3 DSP 运算指令的指令系统

DSP 运算指令是在 DSP 单元处理数字信号的指令。此指令为 32 位操作码，并发执行多个指令。操作码分为 A 字段和 B 字段，在 A 字段指定并发数据传送指令，在 B 字段指定单数据或双数据运算指令。可单独指定指令，也可单独并发执行。在 A 字段指定的并发数据传送指令与双数据传送指令完全相同。

B 字段的数据运算指令分为 3 类：双数据运算指令、带条件单数据运算指令、无条件单数据运算指令。DSP 运算指令的指令格式如**表 5.5** 所示。可从 DSP 寄存器单独选择各自的操作数。 DSP 运算指令的操作数和寄存器的对应关系如**表 5.6** 所示。

表 5.5 DSP 运算指令的指令格式

分类		指令格式	指令
双数据运算指令 （6 个操作数）		ALUop. Sx, Sy, Du MLTop. Se, Sf, Dg	PADD PMULS、 PSUB PMULS
带条件单数据运算指令	3 个操作数	ALUop. Sx, Sy, Dz DCT ALUop. Sx, Sy, Dz DCF ALUop. Sx, Sy, Dz	PADD、 PAND、 POR、 PSHA、 PSHL、 PSUB、 PXOR
	2 个操作数	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 个操作数	ALUop. Dz DCT ALUop. Dz DCF ALUop. Dz	PCLR
无条件的单数据运算指令	3 个操作数	ALUop. Sx, Sy, Du MLTop. Se, Sf, Dg	PADDC、 PSUBC、 PMULS
	2 个操作数	ALUop. Sx, Dz ALUop. Sy, Dz ALUop. Sx, Sy	PCMP、 PABS、 PRND
	1 个操作数	ALUop. Dz	PSHA #imm、 PSHL #imm

表 5.6 DSP 指令的操作数和寄存器的对应关系

寄存器	ALU、 BPU 指令				乘法指令		
	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	

写并发指令时，先写 B 字段的指令，然后写 A 字段的指令。并发处理程序例如图 5.1 所示。

```
        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] [;]
```

图 5.1 并发处理程序例

[ ]表示可省略。可省略空操作指令 NOPX、 NOPY。“;”为指令行的间隔符，可省略。使用间隔符“;”时，后面可用作注释栏。

总是通过无条件的 ALU 运算指令、移位运算指令更新 DSR 寄存器的各状态码（DC、 N、 Z、 V、 GT）。即使带条件指令条件成立，也不更新状态码。乘法指令也不更新状态码。通过指定 DSR 寄存器的 CS 位，决定 DC 位的定义。

DSP 运算指令的分类如表 5.7 所示。

表 5.7 DSP 运算指令的分类

分类		指令种类	操作码	功能	指令数
ALU 算术运算指令	ALU 定点运算指令	11	PABS	绝对值运算	28
			PADD	加法	
			PADD PMULS	加法和带符号乘法	
			PADDC	带进位的加法	
			PCLR	清除	
			PCMP	比较	
			PCOPY	转存	
			PNEG	符号取反	
			PSUB	减法	
			PSUB PMULS	减法和带符号乘法	
			PSUBC	带借位的减法	
	ALU 整数运算指令	2	PDEC	递减	12
			PINC	递增	
	MSB 检测指令	1	PDMSB	MSB 检测	6
	舍入运算指令	1	PRND	舍入运算	2
ALU 逻辑运算指令		3	PAND	逻辑 “与” 运算	9
			POR	逻辑 “或” 运算	
			PXOR	逻辑 “异或” 运算	
定点乘法指令		1	PMULS	带符号乘法	1
移位	算术移位运算指令	1	PSHA	算术移位	4
	逻辑移位运算指令	1	PSHL	逻辑移位	4
系统控制指令		2	PLDS	系统寄存器的加载	12
			PSTS	从系统寄存器存储	
		合计 25			合计 78

5.3.1 ALU 算术运算指令

(1) ALU 定点运算指令

指令	操作	操作码	执行状态	DC 位
PABS Sx,Dz	Sx ≥ 0 时 Sx→Dz Sx < 0 时 0 − Sx→Dz	111110********** 10001000xx00zzzz	1	更新
PABS Sy,Dz	Sy ≥ 0 时 Sy→Dz Sy < 0 时 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	DC = 1 时 Sx+Sy→Dz 0 时 nop	111110********** 10110010xxyyzzzz	1	—
DCF PADD Sx,Sy,Dz	DC = 0 时 Sx+Sy→Dz 1 时 nop	111110********** 10110011xxyyzzzz	1	—
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx+Sy→Du Se 的高位字 × Sf 的高位字 →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	DC = 1 时 H'00000000→Dz 0 时 nop	111110********** 100011100000zzzz	1	—
DCF PCLR Dz	DC = 0 时 H'00000000→Dz 1 时 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	DC = 1 时 Sx→Dz 0 时 nop	111110********** 11011010xx00zzzz	1	—
DCT PCOPY Sy,Dz	DC = 1 时 Sy→Dz 0 时 nop	111110********** 1111101000yyzzzz	1	—
DCF PCOPY Sx,Dz	DC = 0 时 Sx→Dz 1 时 nop	111110********** 11011011xx00zzzz	1	—
DCF PCOPY Sy,Dz	DC = 0 时 Sy→Dz 1 时 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	DC = 1 时 0–Sx→Dz 0 时 nop	111110********** 11001010xx00zzzz	1	—
DCT PNEG Sy,Dz	DC = 1 时 0–Sy→Dz 0 时 nop	111110********** 1110101000yyzzzz	1	—

指令	操作	操作码	执行状态	DC 位
DCF PNEG Sx,Dz	DC = 0 时 0–Sx→Dz 1 时 nop	111110********** 11001011xx00zzzz	1	—
DCF PNEG Sy,Dz	DC = 0 时 0–Sy→Dz 1 时 nop	111110********** 1110101100yyzzzz	1	—
PSUB Sx,Sy,Dz	Sx–Sy→Dz	111110********** 10100001xxyyzzzz	1	更新
DCT PSUB Sx,Sy,Dz	DC = 1 时 Sx–Sy→Dz 0 时 nop	111110********** 10100010xxyyzzzz	1	—
DCF PSUB Sx,Sy,Dz	DC = 0 时 Sx–Sy→Dz 1 时 nop	111110********** 10100011xxyyzzzz	1	—
PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx–Sy→Du Se 的高位字 × Sf 的高位字 →Dg	111110********** 0110eeffxxyygguu	1	更新
PSUBC Sx,Sy,Dz	Sx–Sy–DC→Dz	111110********** 10100000xxyyzzzz	1	更新

(2) ALU 整数运算指令

指令	操作	操作码	执行状态	DC 位
PDEC Sx,Dz	Sx 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字	111110********** 10001001xx00zzzz	1	更新
PDEC Sy,Dz	Sy 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字	111110********** 1010100100yyzzzz	1	更新
DCT PDEC Sx,Dz	DC = 1 时 Sx 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字 0 时 nop	111110********** 10001010xx00zzzz	1	—
DCT PDEC Sy,Dz	DC = 1 时 Sy 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字 0 时 nop	111110********** 1010101000yyzzzz	1	—
DCF PDEC Sx,Dz	DC = 0 时 Sx 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字 1 时 nop	111110********** 10001011xx00zzzz	1	—
DCF PDEC Sy,Dz	DC = 0 时 Sy 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字 1 时 nop	111110********** 1010101100yyzzzz	1	—
PINC Sx,Dz	Sx 的高位字 +1→Dz 的高位字， 清除 Dz 的低位字	111110********** 10011001xx00zzzz	1	更新
PINC Sy,Dz	Sy 的高位字 +1→Dz 的高位字， 清除 Dz 的低位字	111110********** 1011100100yyzzzz	1	更新
DCT PINC Sx,Dz	DC = 1 时 Sx 的高位字 +1→Dz 的高位字， 清除 Dz 的低位字 0 时 nop	111110********** 10011010xx00zzzz	1	—

指令	操作	操作码	执行状态	DC 位
DCT PINC Sy,Dz	DC = 1 时 Sy 的高位字 +1→Dz 的高位字， 清除 Dz 的低位字 0 时 nop	111110********** 1011101000yyzzzz	1	—
DCF PINC Sx,Dz	DC = 0 时 Sx 的高位字 +1→Dz 的高位字， 清除 Dz 的低位字 1 时 nop	111110********** 10011011xx00zzzz	1	—
DCF PINC Sy,Dz	DC = 0 时 Sy 的高位字 +1→Dz 的高位字， 清除 Dz 的低位字 1 时 nop	111110********** 1011101100yyzzzz	1	—

(3) MSB 检测指令

指令	操作	操作码	执行状态	DC 位
PDMSB Sx,Dz	Sx 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字	111110********** 10011101xx00zzzz	1	更新
PDMSB Sy,Dz	Sy 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字	111110********** 1011110100yyzzzz	1	更新
DCT PDMSB Sx,Dz	DC = 1 时 Sx 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字 0 时 nop	111110********** 10011110xx00zzzz	1	—
DCT PDMSB Sy,Dz	DC = 1 时 Sy 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字 0 时 nop	111110********** 1011111000yyzzzz	1	—
DCF PDMSB Sx,Dz	DC = 0 时 Sx 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字 1 时 nop	111110********** 10011111xx00zzzz	1	—
DCF PDMSB Sy,Dz	DC = 0 时 Sy 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字 1 时 nop	111110********** 1011111100yyzzzz	1	—

(4) 舍入运算指令

指令	操作	操作码	执行状态	DC 位
PRND Sx,Dz	Sx+H'00008000→Dz 清除 Dz 的低位字	111110********** 10011000xx00zzzz	1	更新
PRND Sy,Dz	Sy+H'00008000→Dz 清除 Dz 的低位字	111110********** 1011100000yyzzzz	1	更新

5.3.2 ALU 逻辑运算指令

指令	操作	操作码	执行状态	DC 位
PAND Sx,Sy,Dz	Sx & Sy→ Dz 清除 Dz 的低位字	111110********** 10010101xxyyzzzz	1	更新
DCT PAND Sx,Sy,Dz	DC = 1 时 Sx & Sy→Dz 清除 Dz 的低位字 0 时 nop	111110********** 10010110xxyyzzzz	1	—
DCF PAND Sx,Sy,Dz	DC = 0 时 Sx & Sy→Dz 清除 Dz 的低位字 1 时 nop	111110********** 10010111xxyyzzzz	1	—
POR Sx,Sy,Dz	Sx | Sy→Dz 清除 Dz 的低位字	111110********** 10110101xxyyzzzz	1	更新
DCT POR Sx,Sy,Dz	DC = 1 时 Sx | Sy→Dz 清除 Dz 的低位字 0 时 nop	111110********** 10110110xxyyzzzz	1	—
DCF POR Sx,Sy,Dz	DC = 0 时 Sx | Sy→Dz 清除 Dz 的低位字 1 时 nop	111110********** 10110111xxyyzzzz	1	—
PXOR Sx,Sy,Dz	Sx ^ Sy→Dz 清除 Dz 的低位	111110********** 10100101xxyyzzzz	1	更新
DCT PXOR Sx,Sy,Dz	DC = 1 时 Sx ^ Sy→Dz 清除 Dz 的低位字 0 时 nop	111110********** 10100110xxyyzzzz	1	—
DCF PXOR Sx,Sy,Dz	DC = 0 时 Sx ^ Sy→Dz 清除 Dz 的低位字 1 时 nop	111110********** 10100111xxyyzzzz	1	—

5.3.3 定点乘法指令

指令	操作	操作码	执行状态	DC 位
PMULS Se,Sf,Dg	Se 的高位字 × Sf 的高位字 →Dg	111110********** 0100eeff0000gg00	1	—

5.3.4 移位运算指令

(1) 算术移位运算指令

指令	操作	操作码	执行状态	DC 位
PSHA Sx,Sy,Dz	Sy ≥ 0 时 Sx << Sy→Dz Sy < 0 时 Sx >> Sy→Dz	111110********** 10010001xxyyzzzz	1	更新
DCT PSHA Sx,Sy,Dz	DC = 1 & Sy ≥ 0 时 Sx << Sy→Dz DC = 1 & Sy < 0 时 Sx >> Sy→Dz DC = 0 时 nop	111110********** 10010010xxyyzzzz	1	—
DCF PSHA Sx,Sy,Dz	DC = 0 & Sy ≥ 0 时 Sx << Sy→Dz DC = 0 & Sy < 0 时 Sx >> Sy→Dz DC = 1 时 nop	111110********** 10010011xxyyzzzz	1	—
PSHA #imm,Dz	imm ≥ 0 时 Dz << imm→Dz imm < 0 时 Dz >> imm→Dz	111110********** 00010iiiiiiizzzz	1	更新

(2) 逻辑移位运算指令

指令	操作	操作码	执行状态	DC 位
PSHL Sx,Sy,Dz	Sy ≥ 0 时 Sx << Sy→Dz, 清除 Dz 的低位字 Sy < 0 时 Sx >> Sy→Dz, 清除 Dz 的低位字	111110********** 10000001xxyyzzzz	1	更新
DCT PSHL Sx,Sy,Dz	DC = 1 & Sy ≥ 0 时 Sx << Sy→Dz, 清除 Dz 的低位字 DC = 1 & Sy < 0 时 Sx >> Sy→Dz, 清除 Dz 的低位字 DC = 0 时 nop	111110********** 10000010xxyyzzzz	1	—
DCF PSHL Sx,Sy,Dz	DC = 0 & Sy ≥ 0 时 Sx << Sy→Dz, 清除 Dz 的低位字 DC = 0 & Sy < 0 时 Sx >> Sy→Dz, 清除 Dz 的低位字 DC = 1 时 nop	111110********** 10000011xxyyzzzz	1	—
PSHL #imm,Dz	imm ≥ 0 时 Dz << imm→Dz, 清除 Dz 的低位字 imm < 0 时 Dz >> imm→Dz, 清除 Dz 的低位字	111110********** 00000iiiiiiizzzz	1	更新

5.3.5 系统控制指令

指令	操作	操作码	执行状态	DC 位
PLDS Dz,MACH	Dz→MACH	111110********** 1110110100000zzzz	1	—
PLDS Dz,MACL	Dz→MACL	111110********** 1111110100000zzzz	1	—
DCT PLDS Dz,MACH	DC = 1 时 Dz→MACH 0 时 nop	111110********** 1110111000000zzzz	1	—
DCT PLDS Dz,MACL	DC = 1 时 Dz→MACL 0 时 nop	111110********** 1111111000000zzzz	1	—
DCF PLDS Dz,MACH	DC = 0 时 Dz→MACH 1 时 nop	111110********** 1110111100000zzzz	1	—
DCF PLDS Dz,MACL	DC = 0 时 Dz→MACL 1 时 nop	111110********** 1111111100000zzzz	1	—
PSTS MACH,Dz	MACH→Dz	111110********** 1100110100000zzzz	1	—
PSTS MACL,Dz	MACL→Dz	111110********** 1101110100000zzzz	1	—
DCT PSTS MACH,Dz	DC = 1 时 MACH→Dz 0 时 nop	111110********** 1100111000000zzzz	1	—
DCT PSTS MACL,Dz	DC = 1 时 MACL→Dz 0 时 nop	111110********** 1101111000000zzzz	1	—
DCF PSTS MACH,Dz	DC = 0 时 MACH→Dz 1 时 nop	111110********** 1100111100000zzzz	1	—
DCF PSTS MACL,Dz	DC = 0 时 MACL→Dz 1 时 nop	111110********** 1101111100000zzzz	1	—

5.3.6 NOPX 和 NOPY 的操作码

没有与 DSP 运算指令同时并发处理的数据传送指令时，可在数据传送指令写 NOPX、 NOPY 指令或省略指令。无论是写或省略 NOPX、 NOPY 指令，操作码都相同。 NOPX 和 NOPY 的操作码例子如**表 5.8** 所示。

表 5.8 NOPX 和 NOPY 的操作码

指令			代码
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

第 6 章 指令说明

分别按照字母顺序说明 CPU 指令、 DSP 数据传送指令、 DSP 运算指令。

6.1 CPU 指令说明

以下面的格式按照字母顺序说明：

指令名称 指令功能 （英文） 指令分类
指令功能 延迟转移指令或中断禁止指令的表示

格式	操作概略	操作码	执行状态	T 位	适用指令
以汇编程序的输入格式表示。imm、disp 为数值、表达式或符号。	表示操作概略。	以 MSB←→LSB 顺序表示。	无等待时的值。	表示指令执行后 T 位的值。	表示指令适用于 SH-1、 SH-2、 SH-DSP 中的哪一个 CPU。

(1) 说明

操作的说明。

(2) 注意

说明使用指令时需特别注意的事项。

(3) 操作内容

表示以 C 语言操作的内容。在此，假设使用以下资源：

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

返回地址 Addr 各自长度的内容。从地址 2n 以外的地址读取字、从地址 4n 以外的地址读取长字时，检测为地址错误。

```
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);
```

以各自的长度将数据 Data 写入地址 Addr。向地址 2n 以外的地址写入字、向地址 4n 以外的地址写入长字时，检测为地址错误。

```
Delay_Slot(unsigned long Addr);
```

转移至执行地址 （Addr-4）的槽指令。例如 “Delay_Slot(4);”，意思不是转移至执行地址 4，而是转移至执行地址 0 的指令。如果要从该函数转移至执行下列指令，则在转移前检测出下列指令为槽非法指令。如果延迟槽指令为下列指令，则为槽非法指令。

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

```
unsigned long IS_32bit,Inst(unsigned long Addr);
```

如果地址 （Addr_4）的指令为 32 位长度，则返回 2 ； 为 16 位长度，则返回 0。

```
unsigned long R [16];
unsigned long SR,GBR,VBR;
unsigned long MACH,MACL,PR;
unsigned long PC;
```

各寄存器

```
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 结构定义

```
#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 中位的定义

```
Error( char *er );
```

错误显示函数

除此之外，假设 PC 表示当前执行指令的 4 字节后的地址，例如 “PC=4;”意思不是转移至执行地址 4，而是转移至执行地址 0 的指令。

(4) 使用示例

以汇编程序助记符举例，表示指令执行前后的状态。

欧文斜体字 （例：*.align*）表示汇编程序控制指令。汇编程序控制指令的含义如下。详细内容参照 “交叉汇编程序用户手册”。

.org	设定位置计数器
.data.w	确保字整数数据
.data.l	确保长字整数数据
.sdata	确保字符串数据
.align 2	调整 2 个字节边界
.align 4	调整 4 个字节边界
.arepeat 16	重复展开 16 次
.arepeat 32	重复展开 32 次
.aendr	指定次数重复展开结束

【注】 SH 系列交叉汇编程序 Ver 1.0 不支持带条件汇编程序功能。

【注】 *1 下列带位移量 （disp）的寻址方式中，本手册的汇编程序记述为执行根据操作数长度进行放大 （×1、 ×2、 ×4）前的值，这是为了明确 LSI 的操作，实际的汇编程序说明参照各汇编程序的记述规则。

@ (disp : 4, Rn)； 带位移量的寄存器间接寻址

@ (disp : 8, GBR)； 带位移量的 GBR 间接寻址

@ (disp : 8, PC)； 带位移量的 PC 相对寻址

disp : 8, disp : 12； PC 相对寻址

*2 在操作码 16 位中，未分配代码的指令作为一般非法指令处理，并产生非法指令异常处理。

例 H'FFFF ［一般非法指令］

*3 如果 BRA、 BT/S 等延迟转移指令的下一条指令为一般非法指令或转移指令 （将此称为槽非法指令），则产生非法指令异常处理。

例 1 ……

```
BRA   LABEL
.data. w   H'FFFF   ←   槽非法指令
……［H'FFFF 原为一般非法指令］
```

例 2 RTE

```
BT/S   LABEL   ←   槽非法指令
```

*4 延迟转移的转移运行产生在槽指令执行后。但仍按照延迟转移指令、延迟槽指令的顺序执行除更新寄存器等转移运行之外的指令。例如，即使在延迟槽变更保存转移目标地址的寄存器内容，转移目标地址仍为变更前寄存器的内容。

*5 如果小于等于 3 条指令或大于等于 4 条指令的重复程序 （循环）的最后 3 条指令中，存在一般非法指令、转移指令或更新 SR、 RS、 RE 寄存器的指令 （SETRC、 LDRS 等），则产生非法指令异常处理。详细内容参照 “4.19 DSP 重复 （循环）控制”。

6.1.1 ADD ADD binary 算术运算指令

2 进制加法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ADD Rm,Rn	Rn+Rm→Rn	0011nnnnmmmm1100	1	—	○	○	○
ADD #imm,Rn	Rn+imm→Rn	0111nnnniiiiiiii	1	—	○	○	○

(1) 说明

通用寄存器 Rn 的内容与 Rm 相加，结果保存在 Rn。
通用寄存器 Rn 与 8 位立即数也可相加。
8 位立即数进行符号扩展后为 32 位，因此可与减法运算并用。

(2) 操作内容

```
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) 使用示例

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

6.1.2 ADDC ADD with Carry 算术运算指令

带进位的 2 进制加法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ADDC Rm,Rn	Rn+Rm+T→Rn、进位 →T	0011nnnnmmmm1110	1	进位	○	○	○

(1) 说明

通用寄存器 Rn 的内容与 Rm 及 T 位相加，结果保存在 Rn。根据运算结果在 T 位反映进位。本指令用于超过 32 位的加法运算。

(2) 操作内容

```
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) 使用示例

```
CLRT          ; R0:R1(64 位 )+R2:R3(64 位 )=R0:R1(64 位 )
ADDC  R3,R1   ; 执行前 T=0,R1=H'00000001,R3=H'FFFFFFFF
              ; 执行后 T=1,R1=H'00000000
ADDC  R2,R0   ; 执行前 T=1,R0=H'00000000,R2=H'00000000
              ; 执行后 T=0,R0=H'00000001
```

6.1.3 ADDV ADD with （Vflag）overflow check 算术运算指令

带上溢的 2 进制加法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ADDV Rm,Rn	Rn+Rm→Rn、上溢 →T	0011nnnnmmmm1111	1	上溢	○	○	○

(1) 说明

通用寄存器 Rn 的内容与 Rm 相加，结果保存在 Rn。如果产生上溢，则 T 位置位。

(2) 操作内容

```
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) 使用示例

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

6.1.4 AND AND logical 逻辑运算指令

逻辑 “与” 运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

取通用寄存器 Rn 的内容与 Rm 的逻辑 “与”，结果保存在 Rn。

可执行通用寄存器 R0 与零扩展后的 8 位立即数的逻辑 “与” 运算，或带变址的 GBR 间接寻址方式下 8 位存储器与 8 位立即数的逻辑 “与” 运算。

(2) 注意

AND #imm,R0 时，总是清除运算结果 R0 的高 24 位。

(3) 操作内容

```
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) 使用示例

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

6.1.5 BF Branch if False 转移指令

条件转移

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BF label	T = 0 时 disp×2+PC→PC、 T = 1 时 nop	10001011dddddddd	3/1	—	○	○	○

(1) 说明

本指令为参考 T 位的带条件转移指令。 T=0 时，转移至转移目标地址； T=1 时，执行下一条指令。

转移目标为 PC 加位移量后的地址。但此时用于地址计算的 PC 为本指令 4 字节后的地址。8 位位移量符号扩展后乘以 2，因此与转移目标的相对距离在 –256 字节到 +254 字节范围内。未到达转移目标时，需通过与 BRA 指令的组合来解决。

(2) 注意

转移时为 3 个状态，不转移时为 1 个状态。

(3) 操作内容

```
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) 使用示例

```
        CLRT            ; 总是为 T=0
        BT    TRGET_T   ; 因为 T=0，所以不转移。
        BF    TRGET_F   ; 因为 T=0，所以转移到 TRGET_F。
        NOP             ;
        NOP             ; ←BF 指令时用于计算转移目标地址的 PC 位置
        ---
TRGET_F:                ; ←BF 指令的转移目标
```

6.1.6 BF/S Branch if False with delay Slot 转移指令

带条件的延迟转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BF/S label	T = 0 时 disp×2+PC→PC、 T = 1 时 nop	10001111dddddddd	2/1	—	—	○	○

(1) 说明

本指令为参考 T 位的带条件的延迟转移指令。T = 0 时，执行下一条指令后转移； T = 1 时，执行下一条指令。

转移目标为 PC 加位移量后的地址。但此时用于地址计算的 PC 为本指令 4 字节后的地址。8 位位移量符号扩展后乘以 2，因此与转移目标的相对距离在 –256 字节到 +254 字节范围内。未到达转移目标时，需通过与 BRA 指令组合来解决。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。

在本指令与紧接着的指令之间不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

转移时为 2 个状态，不转移时为 1 个状态。

(3) 操作内容

```
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) 使用示例

```
     CLRT                    ; 总是为 T=0
     BT/S TRGET_T            ; 因为 T=0，所以不转移。
     NOP                     ;
     BF/S TRGET_F            ; 因为 T=0，所以转移至 TRGET_F。
     ADD R0,R1               ; 在转移前执行。
     NOP                     ;←BF/S 指令时用于计算转移目标地址的 PC 位置
     ---
TRGET_F                      ;←BF/S 指令的转移目标
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令（寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.7 BRA BRAnch 转移指令

无条件转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BRA label	disp×2+PC→PC	1010dddddddddddd	2	—	○	○	○

(1) 说明

本指令为无条件的延迟转移指令。转移目标为 PC 加位移量后的地址，但此时用于地址计算的 PC 为本指令 4 字节后的地址。 12 位位移量符号扩展后乘以 2，因此与转移目标相对距离在 –4096 字节到 +4094 字节范围内。未到达转移目标时，需以 MOV 指令将转移目标地址传送至寄存器后变更为 JMP 指令。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。
在本指令与紧接着的指令之间不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

```
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) 使用示例

```
     BRA   TRGET              ; 转移至 TRGET。
     ADD   R0,R1              ; 在转移前执行。
     NOP                      ;←BRA 指令时用于计算转移目标地址的 PC 位置
     ---
TRGET:                        ;←BRA 指令的转移目标
```

【注】 延迟转移的转移运行产生在执行槽指令后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.8 BRAF BRAnch Far 转移指令
无条件转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BRAF Rm	Rm+PC→PC	0000mmmm00100011	2	—	—	○	○

(1) 说明

本指令为无条件的延迟转移指令。转移目标为 PC 加通用寄存器 Rm 内容的 32 位后的地址。此时，用于计算地址的 PC 为本指令 4 字节后的地址。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。

本指令与紧接着的指令之间不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

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

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

(4) 使用示例

```
     MOV.L #(TRGET-BSRF_PC),R0      ; 设定位移量。
     BRAF  R0                       ; 转移至 TRGET。
     ADD   R0,R1                    ; 在转移前执行。
BRAF_PC:                            ;←BRAF 指令时用于计算转移目标地址的 PC 位置
     NOP
     ---
TRGET:                              ;←BRAF 指令的转移目标
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令（寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.9 BSR Branch to SubRoutine 转移指令

向子程序过程的转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BSR label	PC→PR、 disp×2+PC→PC	1011dddddddddddd	2	—	○	○	○

(1) 说明

向指定地址的子程序过程延迟转移。 PC 的内容保存至 PR，并转移至 PC 加位移量后的地址。此时，用于计算地址的 PC 为本指令 4 字节后的地址。符号扩展 12 位位移量后乘以 2，因此与转移目标的相对距离在 –4096 字节到 +4094 字节范围内。未到达转移目标时，需以 MOV 指令将转移目标地址传送至寄存器后变更为 JSR 指令。与 RTS 组合后用于子程序过程调用。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。
本指令与紧接着的指令之间不接收地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

```
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) 使用示例

```
     BSR   TRGET      ; 转移至 TRGET。
     MOV   R3,R4      ; 在转移前执行。
     ADD   R0,R1      ;←BSR 指令时用于计算转移目标地址的 PC 位置
     ……               从过程返回的地址 （PR 的内容）。
     ……
TRGET:                ;← 过程入口
     MOV   R2,R3      ;
     RTS              ; 返回至上述 ADD 指令。
     MOV   #1,R0      ; 在转移前执行。
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.10 BSRF Branch to SubRoutine Far 转移指令

向子程序过程转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BSRF Rm	PC→PR、 Rm+PC→PC	0000mmmm00000011	2	—	—	○	○

(1) 说明

向指定地址的子程序过程延迟转移。PC 的内容保存至 PR。转移目标为 PC 加通用寄存器 Rm 内容的 32 位数据后的地址。此时，用于计算地址的 PC 为本指令 4 字节后的地址。与 RTS 组合用于子程序过程调用。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。

本指令与紧接着的指令之间，不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

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

(4) 使用示例

```
   MOV.L #(TRGET-BSRF_PC),R0          ; 设定位移量。
   BSRF  R0                           ; 转移至 TRGET。
   MOV   R3,R4                        ; 在转移前执行。
BSRF_PC:                              ;←BSRF 指令时用于计算转移目标地址的 PC 位置
   ADD   R0,R1                        ;
   ……
   ……
TRGET:                                ;← 过程入口
   MOV  R2,R3                         ;
   RTS                                ; 返回至上述 ADD 指令。
   MOV  #1,R0                         ; 在转移前执行。
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.11 BT Branch if True 转移指令

条件转移

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BT label	T = 1 时 disp×2+PC→PC、 T = 0 时 nop	10001001dddddddd	3/1	—	○	○	○

(1) 说明

本指令为参考 T 位的带条件的转移指令。 T=1 时，转移； T=0 时，执行下一条指令。

转移目标为 PC 加位移量后的地址。此时，用于计算地址的 PC 为本指令 4 字节后的地址。符号扩展 8 位位移量后乘以 2，因此与转移目标的相对距离在 –256 字节到 +254 字节范围内。未到达转移目标时，需与 BRA 等指令组合来解决。

(2) 注意

转移时为 3 个状态；不转移时为 1 个状态。

(3) 操作内容

```
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) 使用示例

```
      SETT              ; 总是为 T=1
      BF    TRGET_F     ; 因为 T=1，所以不转移。
      BT    TRGET_T     ; 因为 T=1，所以转移至 TRGET_T。
      NOP               ;
      NOP               ;←BT 指令时用于计算转移目标地址的 PC 位置
      ---
TRGET_T:                ;←BT 指令的转移目标
```

6.1.12 BT/S Branch if True with delay Slot 转移指令

带条件的延迟转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
BT/S label	T = 1 时 disp×2+PC→PC、 T = 0 时 nop	10001101dddddddd	2/1	—	—	○	○

(1) 说明

本指令为参考 T 位的带条件的延迟转移指令。 T=1 时，执行下一条指令后转移； T=0 时，执行下一条指令。

转移目标为 PC 加位移量后的地址。此时，用于计算地址的 PC 为本指令 4 字节后的地址。符号扩展 8 位位移量后乘以 2，因此与转移目标的相对距离在 –256 字节到 +254 字节范围内。未到达转移目标时，需与 BRA 等指令组合来解决。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。
本指令与紧接着的指令之间，不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。
转移时为 2 个状态；不转移时为 1 个状态。

(3) 操作内容

```
      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) 使用示例

```
      SETT              ; 总是为 T=1
      BF/S TRGET_F      ; 因为 T=1，所以不转移。
      NOP               ;
      BT/S TRGET_T      ; 因为 T=1，所以转移至 TRGET_T。
      ADD R0,R1         ; 在转移前执行。
      NOP               ;←BT/S 指令时用于计算转移目标地址的 PC 位置
      ---
  TRGET_T:              ;← BT/S 指令的转移目标
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.13 CLRMAC CLeaR MAC register 系统控制指令

MAC 寄存器的清除

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
CLRMAC	0→MACH、 MACL	0000000000101000	1	—	○	○	○

(1) 说明

清除 MACH、 MACL 寄存器。

(2) 操作内容

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

(3) 使用示例

```
CLRMAC            ;清除 MAC 寄存器后初始化。
MAC.W  @R0+,@R1+  ;乘法累加运算
MAC.W  @R0+,@R1+  ;
```

6.1.14 CLRT CLeaR Tbit 系统控制指令

T 位的清除

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
CLRT	0→T	0000000000001000	1	0	○	○	○

(1) 说明

清除 T 位。

(2) 操作内容

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

(3) 使用示例

```
CLRT        ;执行前 T=1
            ;执行后 T=0
```

6.1.15 CMP/cond CoMPare conditionally 算术运算指令

比较

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
CMP/EQ Rm,Rn	Rn = Rm 时， 1→T	0011nnnnmmmm0000	1	比较结果	○	○	○
CMP/GE Rm,Rn	有符号且 Rn ≥ Rm 时， 1→T	0011nnnnmmmm0011	1	比较结果	○	○	○
CMP/GT Rm,Rn	有符号且 Rn > Rm 时， 1→T	0011nnnnmmmm0111	1	比较结果	○	○	○
CMP/HI Rm,Rn	无符号且 Rn > Rm 时， 1→T	0011nnnnmmmm0110	1	比较结果	○	○	○
CMP/HS Rm,Rn	无符号且 Rn ≥ Rm 时， 1→T	0011nnnnmmmm0010	1	比较结果	○	○	○
CMP/PL Rn	Rn > 0 时， 1→T	0100nnnn00010101	1	比较结果	○	○	○
CMP/PZ Rn	Rn ≥ 0 时， 1→T	0100nnnn00010001	1	比较结果	○	○	○
CMP/STR Rm,Rn	任意字节相等时， 1→T	0010nnnnmmmm1100	1	比较结果	○	○	○
CMP/EQ #imm,R0	R0 = imm 时， 1→T	10001000iiiiiiii	1	比较结果	○	○	○

(1) 说明

比较通用寄存器 Rn 与 Rm，结果为如果指定条件 （cond）成立，则置位 T 位；如果条件不成立，则清除 T 位， Rn 的内容不变。可指定 8 个条件。 PZ 与 PL 条件时为 Rn 与 0 的比较。

EQ 条件可比较符号扩展后的 8 位立即数与 R0， R0 的内容不变。

助记符		说明
CMP/EQ	Rm,Rn	Rn = Rm 时 T = 1
CMP/GE	Rm,Rn	作为有符号值 Rn ≥ Rm 时 T = 1
CMP/GT	Rm,Rn	作为有符号值 Rn > Rm 时 T = 1
CMP/HI	Rm,Rn	作为无符号值 Rn > Rm 时 T = 1
CMP/HS	Rm,Rn	作为无符号值 Rn ≥ Rm 时 T = 1
CMP/PL	Rn	Rn > 0 时 T = 1
CMP/PZ	Rn	Rn ≥ 0 时 T = 1
CMP/STR	Rm,Rn	任意字节相等时 T = 1
CMP/EQ	#imm,R0	R0 = imm 时 T = 1

(2) 操作内容

```
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) 使用示例

```
CMP/GE   R0,R1              ;R0=H'7FFFFFFF,R1=H'80000000
BT TRGET_T                  ; 因为 T=0，所以不转移。
CMP/HS   R0,R1              ;R0=H'7FFFFFFF,R1=H'80000000
BT TRGET_T                  ; 因为 T=1，所以转移。
CMP/STR  R2,R3              ;R2="ABCD",R3="XYCZ"
BT TRGET_T                  ; 因为 T=1，所以转移。
```

6.1.16 DIV0S DIVide(step0) as Signed 算术运算指令

带符号除法的初始化

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
DIV0S Rm,Rn	Rn 的 MSB→Q、 Rm 的 MSB→M、 M^Q→T	0010nnnnmmmm0111	1	计算结果	○	○	○

(1) 说明

初始设定带符号的除法，在本指令后结合执行 1 位除法运算的 DIV1 指令等，重复执行除法运算求商。详细内容参照 DIV1 说明。

(2) 操作内容

```
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) 使用示例

参照 DIV1 的使用示例。

6.1.17 DIV0U DIVide (step0) as Unsigned 算术运算指令

无符号除法的初始化

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
DIV0U	0→M/Q/T	0000000000011001	1	0	○	○	○

(1) 说明

初始设定无符号的除法。在本指令后结合执行 1 位除法运算的 DIV1 指令等，重复执行除法运算求商。详细内容参照 DIV1 说明。

(2) 操作内容

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

(3) 使用示例

参照 DIV1 的使用示例。

6.1.18 DIV1 DIVide 1 step 算术运算指令

除法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
DIV1 Rm,Rn	单步除法 (Rn÷Rm)	0011nnnnmmmm0100	1	计算结果	○	○	○

(1) 说明

本指令为执行通用寄存器 Rn 的 32 位内容 （被除数）除以 Rm 内容 （除数）的 1 位除法 （单步除法）的指令。单独执行本指令或结合其他指令并重复执行求商。重复执行中，不得改写指定的寄存器与 M、 Q、 T 位。

单步除法，是指将被除数左移 1 位，然后减去除数，根据结果的正负将商的位反映到 Q 位。

通过除法求余数时，用 DIV1 指令求商后按照以下公式计算：

（被除数）— （除数）×（商）= （余数）

不支持被零除及上溢的检测。除法运算前，必须检查被零除及上溢除法，不支持余数运算。求除数与求得的商的乘积，从被除数减去所得的乘积为余数。

首先用 DIV0S 或 DIV0U 进行初始设定。重复执行 DIV1，其重复执行的次数为除数的位数。需要商大于等于 17 位时，将 ROTCL 放在 DIV1 之前。除法运算的顺序，参照下述的使用示例。

(2) 操作内容

```
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) 使用示例 1

```
                                ;R1(32 位 ) ÷ R0(16 位 )=R1(16 位 ): 无符号
    SHLL16          R0          ; 将除数设定为高 16 位，低 16 位设定为 0
    TST             R0,R0       ; 检查被零除
    BT              ZERO_DIV    ;
    CMP/HS          R0,R1       ; 检查上溢
    BT              OVER_DIV    ;
    DIV0U                       ; 初始化标志
    .arepeat        16          ;
    DIV1            R0,R1       ; 重复 16 次
    .aendr                      ;
    ROTCL           R1          ;
    EXTU.W          R1,R1       ;R1= 商
```

(4) 使用示例 2

```
                                ;R1:R2(64 位 ) ÷ R0(32 位 )=R2(32 位 ): 无符号
  TST           R0,R0           ; 检查被零除
  BT            ZERO_DIV        ;
  CMP/HS        R0,R1           ; 检查上溢
  BT            OVER_DIV        ;
  DIV0U                         ; 初始化标志
  .arepeat      32              ;
  ROTCL         R2              ; 重复 32 次
  DIV1          R0,R1           ;
  .aendr                        ;
  ROTCL         R2              ;R2= 商
```

(5) 使用示例 3

```
                                        ;R1(16 位 ) ÷ R0(16 位 )=R1(16 位 ): 带符号
  SHLL16                R0              ; 将除数设定为高 16 位，低 16 位设定为 0
  EXTS.W                R1,R1           ; 将被除数符号扩展为 32 位
  XOR                   R2,R2           ;R2=0
  MOV                   R1,R3           ;
  ROTCL                 R3              ;
  SUBC                  R2,R1           ; 被除数为负时，减 1。
  DIV0S                 R0,R1           ; 初始化标志
  .arepeat              16              ;
  DIV1                  R0,R1           ; 重复 16 次
  .aendr                                ;
  EXTS.W                R1,R1           ;
  ROTCL                 R1              ;R1= 商 ( 以 1 的补码表示 )
  ADDC                  R2,R1           ; 商的 MSB 为 1 时，加 1 后以 2 的补码表示
  EXTS.W                R1,R1           ;R1= 商 ( 以 2 的补码表示 )
```

(6) 使用示例 4

```
                                        ;R2(32 位 ) ÷ R0(32 位 )=R2(32 位 ): 带符号
  MOV                   R2,R3           ;
  ROTCL                 R3              ;
  SUBC                  R1,R1           ; 将被除数符号扩展为 64 位 (R1:R2)
  XOR                   R3,R3           ;R3=0
  SUBC                  R3,R2           ; 被除数为负时，减 1 后以 1 的补码表示
  DIV0S                 R0,R1           ; 初始化标志
  .arepeat              32              ;
  ROTCL                 R2              ; 重复 32 次
  DIV1                  R0,R1           ;
  .aendr                                ;
  ROTCL                 R2              ;R2= 商 ( 以 1 的补码表示 )
  ADDC                  R3,R2           ; 商的 MSB 为 1 时，加 1 后以 2 的补码表示
                                        ;R2= 商 ( 以 2 的补码表示 )
```

6.1.19 DMULS.L Double-length MULtiply as Signed 算术运算指令

带符号的双精度乘法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
DMULS.L Rm,Rn	带符号 Rn×Rm→MACH、MACL	0011nnnnmmmm1101	2～4	—	—	○	○

(1) 说明

通用寄存器 Rn 的内容与 Rm 进行 32 位乘法运算，并将得到的 64 位结果保存至 MACH 寄存器及 MACL 寄存器。运算为带符号的算术运算。

(2) 操作内容

```
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) 使用示例

```
DMULS.L     R0,R1        ;执行前 R0=H'FFFFFFFE,R1=H'00005555
                         ;执行后 MACH=H'FFFFFFFF,MACL=H'FFFF5556
STS         MACH,R0      ;得到运算结果 ( 高位 )
STS         MACL,R0      ;得到运算结果 ( 低位 )
```

6.1.20 DMULU.L Double-length MULtiply as Unsigned 算术运算指令

无符号的双精度乘法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
DMULU.L Rm,Rn	无符号 Rn×Rm→MACH、 MACL	0011nnnnmmmm0101	2 ～ 4	—	—	○	○

(1) 说明

通用寄存器 Rn 的内容与 Rm 进行 32 位乘法运算，并将得到的 64 位结果保存至 MACH 寄存器及 MACL 寄存器。运算为无符号的算术运算。

(2) 操作内容

```
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) 使用示例

```
DMULU.L     R0,R1          ; 执行前 R0=H'FFFFFFFE,R1=H'00005555
                           ; 执行后 MACH=H'00005554,MACL=H'FFFF5556
STS         MACH,R0        ; 得到运算结果 ( 高位 )
STS         MACL,R0        ; 得到运算结果 ( 低位 )
```

6.1.21 DT Decrement and Test 算术运算指令

递减与测试

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
DT Rn	Rn–1→Rn、 Rn 为 0 时 ,1→T Rn 不为 0 时 ,0→T	0100nnnn00010000	1	比较结果	—	○	○

(1) 说明

通用寄存器 Rn 的内容递减 1，并将结果与 0（零）比较。结果为 0 时， T 位置 1 ；结果不为 0 时， T 位清 0。

(2) 操作内容

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

(3) 使用示例

```
      MOV #4,R5          ; 设定循环次数。
LOOP:
      ADD R0,R1          ;
      DT RS              ; 将 R5 的值递减，判断是否为 0。
      BF LOOP            ; 如果 T=0，则转移至 LOOP （此例中循环 4 次）。
```

6.1.22 EXTS EXTend as Signed 算术运算指令

符号扩展

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
EXTS.B Rm,Rn	将 Rm 从字节进行符号扩展 →Rn	0110nnnnmmmm1110	1	—	○	○	○
EXTS.W Rm,Rn	将 Rm 从字进行符号扩展 →Rn	0110nnnnmmmm1111	1	—	○	○	○

(1) 说明

符号扩展通用寄存器 Rm 的内容，并将结果保存在 Rn。
指定字节时，将 Rm 的 bit7 的内容 传送至 Rn 的 bit8~bit31。
指定字时，将 Rm 的 bit15 的内容传送至 Rn 的 bit16~bit31。

(2) 操作内容

```
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) 使用示例

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

6.1.23 EXTU EXTend as Unsigned 算术运算指令

零扩展

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
EXTU.B Rm,Rn	将 Rm 从字节进行零扩展 →Rn	0110nnnnmmmm1100	1	—	○	○	○
EXTU.W Rm,Rn	将 Rm 从字进行零扩展 →Rn	0110nnnnmmmm1101	1	—	○	○	○

(1) 说明

零扩展通用寄存器 Rm 的内容，并将结果保存在 Rn。

指定字节时，将 0 传送至 Rn 的 bit8~bit31；指定字时，将 0 传送至 Rn 的 bit16~bit31。

(2) 操作内容

```
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) 使用示例

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

6.1.24 JMP JuMP 转移指令

无条件转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
JMP @Rm	Rm→PC	0100mmmm00101011	2	—	○	○	○

(1) 说明

无条件地延迟转移至寄存器间接寻址指定的地址。转移目标为通用寄存器 Rm 内容的 32 位数据表示的地址。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。
本指令与紧接着的指令之间，不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

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

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

(4) 使用示例

```
                MOV.L       JMP_TABLE,R0        ;R0=TRGET 的地址
                JMP         @R0                 ; 转移至 TRGET
                MOV         R0,R1               ; 在转移前执行
                .align      4
JMP_TABLE:      .data.l     TRGET               ; 跳转表
          ............
TRGET:          ADD         #1,R1               ;← 转移目标
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 (寄存器的更新等)。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.25 JSR Jump to SubRoutine 转移指令

向子程序过程的转移 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
JSR @Rm	PC→PR、Rm→PC	0100mmmm00001011	2	—	○	○	○

(1) 说明

延迟转移至寄存器间接寻址指定的地址的子程序过程。PC 的内容保存至 PR，并转移至通用寄存器 Rm 内容的 32 位数据表示的地址。保存的 PC 为本指令 4 字节后的地址。与 RTS 组合后，用于子程序过程调用。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。

本指令与紧接着的指令之间，不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

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

(4) 使用示例

```
                MOV.L       JSR_TABLE,R0    ;R0=TRGET 的地址
                JSR         @R0             ; 转移至 TRGET
                XOR         R1,R1           ; 在转移前执行
                ADD         R0,R1           ;← 从过程返回的目标
                .......                     (PR 的内容 )
                .align      4
JSR_TABLE:      .data.l     TRGET           ; 跳转表
TRGET:          NOP                         ;← 过程入口
                MOV R2,R3                   ;
                RTS                         ; 返回至上述 ADD 指令。
                MOV         #70,R1          ; 在 RTS 前执行。
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令（寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.26 LDC LoaD to Control register 系统控制指令

向控制寄存器加载 中断禁止指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

将源操作数保存至控制寄存器 SR、 GBR、 VBR、 MOD、 RE、 RS。

(2) 注意

本指令与紧接着的指令之间，不接受中断 （接收地址错误）。

(3) 操作内容

```
LDCSR(long m)          /* LDC Rm,SR */
{
    SR=R[m]&0x000003F3;                SH-1 CPU 与 SH-2 CPU 时
    SR=R[m]&0x0FFF0FFF;                SH-DSP 时
    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;     SH-1 CPU 与 SH-2 CPU 时
    SR=Read_Long(R[m])&0x0FFF0FFF;     SH-DSP 时
  R[m]+=4;
  PC+=2;
}

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) 使用示例

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

【注】 * 为 SH-DSP 中的执行结果。

6.1.27 LDRE LoaD effective address to RE register 系统控制指令

向重复结束寄存器加载

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
LDRE @(disp,PC)	disp×2+PC→RE	10001110dddddddd	1	—	—	—	○

(1) 说明

将源操作数的有效地址值保存至重复结束寄存器 RE。有效地址为 PC 加位移量后的地址。 PC 为本指令 4 字节后的地址。符号扩展 8 位位移量后乘以 2，因此在 –256 字节到 +254 字节范围内。

(2) 注意

对 RE 寄存器指定的有效地址值与实际的重复结束地址不同，详细内容参照**表 4.36**。如果在延迟转移指令之后紧接着配置本指令，则 PC 为转移目标的 “起始地址 +2”。

(3) 操作内容

```
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) 使用示例

```
      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 系统控制指令

向重复起始寄存器加载

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
LDRS @(disp,PC)	disp×2+PC→RS	10001100dddddddd	1	—	—	—	○

(1) 说明

将源操作数的有效地址值保存至重复起始寄存器 RS。有效地址为 PC 加位移量后的地址。 PC 为本指令 4 字节后的地址。符号扩展 8 位位移量后乘以 2，因此在 –256 字节到 +254 字节范围内。

(2) 注意

重复 （循环）程序小于等于 3 条指令时，对 RS 寄存器指定的有效地址值与实际的重复起始地址不同，详细内容参照**表 4.36**。如果在延迟转移指令之后紧接着配置本指令，则 PC 为转移目标的 “起始地址 +2”。

(3) 操作内容

```
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) 使用示例

```
      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 系统控制指令

向系统寄存器加载 中断禁止指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

将源操作数保存至系统寄存器 MACH、MACL、PR 或 DSP 寄存器 DSR、A0、X0、X1、Y0、Y1。指定 A0 为目标时，将 MSB 数据复制到 A0G。

(2) 注意

本指令与紧接着的指令之间，不接受中断 （接受地址错误）。

SH-1 CPU 中， MACH 保存低 10 位。

SH-2 CPU、 SH-DSP 中， MACH 保存 32 位。

(3) 操作内容

```
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];
    PC+=2;
```

SH-1 CPU 时（SH-2 CPU 及 SH-DSP 中不需要这 2 行。）

```
}

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;
}

LDSMA0(long m)        /* LDS.L @Rm+,A0 */
```

SH-1 CPU 时（SH-2 CPU 及 SH-DSP 中不需要这 2 行。）

```
{
    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) 使用示例

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

6.1.30 MAC.L Multiply and ACcumulate Long 算术运算指令

双精度乘法累加运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MAC.L @Rm+,@Rn+	带符号 (Rn)×(Rm)+MAC→MAC	0000nnnnmmmm1111	3/(2 ～ 4)	—	—	○	○

(1) 说明

对通用寄存器 Rm 及 Rn 的内容作为地址的 32 位操作数执行带符号的乘法运算，将 64 位结果加 MAC 寄存器的内容，结果保存至 MAC 寄存器。每读取一次操作数， Rm 和 Rn 分别加 4。

S 位为 0 时，将 64 位结果保存至连接的 MACH、 MACL 寄存器。

S 位为 1 时，与 MAC 寄存器的加法运算从 LSB 到第 48 位为饱和运算。饱和运算中，仅限 MAC 寄存器的低 48 位有效，结果限制在 H'FFFF800000000000 （最小值）到 H'00007FFFFFFFFFFF （最大值）范围内。

(2) 操作内容

```
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) 使用示例

```
        MOVA        TBLM,R0         ;得到表地址
        MOV         R0,R1;
        MOVA        TBLN,R0         ;得到表地址
        CLRMAC                      ;初始化 MAC 寄存器
        MAC.L       @R0+,@R1+       ;
        MAC.L       @R0+,@R1+       ;
        STS         MACL,R0         ;在 R0 得到结果
    ............
        .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 算术运算指令

乘法累加运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MAC.W @Rm+,@Rn+	带符号且	0100nnnnmmmm1111	3/(2)	—	—	○	○
MAC @Rm+,@Rn+	(Rn)×(Rm)+MAC→MAC				○	○	○

(1) 说明

对通用寄存器 Rm 及 Rn 的内容作为地址的 16 位操作数执行带符号的乘法运算，将 32 位结果加 MAC 寄存器的内容，结果保存至 MAC 寄存器。每读取一次操作数， Rm 和 Rn 分别加 2。

S 位为 0 时，为 16×16+64→64 位的乘法累加运算，将 64 位结果保存至连接的 MACH、 MACL 寄存器。

S 位为 1 时，为 16×16+32→32 位的乘法累加运算，与 MAC 寄存器的加法运算为饱和运算。饱和运算中，仅限 MACL 寄存器有效，结果限制在 H'80000000 （最小值）到 H'7FFFFFFF （最大值）范围内。如果产生上溢，则 MACH 寄存器的 LSB 置 1。结果向负方向上溢时，将 H'80000000 （最小值）保存至 MACL 寄存器；结果向正方向上溢时，将 H'7FFFFFFF （最大值）保存至 MACL 寄存器。

(2) 注意

S 位为 0 时，在 SH-2 CPU 及 SH-DSP，执行 16×16+64→64 位的乘法累加运算；在 SH-1 CPU 执行 16×16+42→42 位的乘法累加运算。

(3) 操作内容

```
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 时 （SH-2 CPU 及 SH-DSP 中不需要这 2 行。）

SH-1 CPU 时 （SH-2 CPU 及 SH-DSP 中不需要这 2 行。）

(4) 使用示例

```
        MOVA    TBLM,R0         ; 得到表地址
        MOV     R0,R1           ;
        MOVA    TBLN,R0         ; 得到表地址
        CLRMAC                  ; 初始化 MAC 寄存器
        MAC.W   @R0+,@R1+       ;
        MAC.W   @R0+,@R1+       ;
        STS     MACL,R0         ; 在 R0 获得结果
    ............
        .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 数据传送指令

数据传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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)→ 符号扩展 →Rn	0110nnnnmmmm0000	1	—	○	○	○
MOV.W @Rm,Rn	(Rm)→ 符号扩展 →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)→ 符号扩展 →Rn、 Rm+1→Rm	0110nnnnmmmm0100	1	—	○	○	○
MOV.W @Rm+,Rn	(Rm)→ 符号扩展 →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) → 符号扩展 →Rn	0000nnnnmmmm1100	1	—	○	○	○
MOV.W @(R0,Rm),Rn	(R0+Rm)→ 符号扩展 →Rn	0000nnnnmmmm1101	1	—	○	○	○
MOV.L @(R0,Rm),Rn	(R0+Rm)→Rn	0000nnnnmmmm1110	1	—	○	○	○

(1) 说明

将源操作数传送至目标。操作数为存储器时，可将传送数据长度指定在字节 / 字 / 长字范围内。源操作数为存储器时，将加载的数据符号扩展为长字后保存至寄存器。

(2) 操作内容

```
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) 使用示例

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

6.1.33 MOV MOVe immediate data 数据传送指令

立即数的传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MOV #imm,Rn	imm→ 符号扩展 →Rn	1110nnnniiiiiiii	1	—	○	○	○
MOV.W @(disp,PC),Rn	(disp×2+PC)→	1001nnnnddddddddd	1	—	○	○	○
MOV.L @(disp,PC),Rn	符号扩展 →Rn						
	(disp×4+PC)→Rn	1101nnnnddddddddd	1	—	○	○	○

(1) 说明

将符号扩展为长字的立即数保存至通用寄存器 Rn。数据为字或长字时，参考在 PC 加位移量后的地址所指定的表内保存的数据。

数据为字时，零扩展 8 位位移量后乘以 2，因此与表的相对距离在 PC+510 字节的范围内。 PC 为本指令的 2 条指令后的起始地址。

数据为长字时，零扩展 8 位位移量后乘以 4，因此与操作数的相对距离在 PC+1020 字节范围内。 PC 为本指令的 2 条指令后的起始地址，将低 2 位修改为 B'00 的值用于地址计算。

(2) 注意

此表最好配置在模块后端或无条件转移指令的 1 条指令后。由于 510 字节 / 1020 字节内没有无条件转移指令，而不能进行最佳配置时，通过 BRA 指令跳过该表。如果在延迟转移指令之后紧接着配置本指令，则 PC 为转移目标的 “起始地址 +2”。

(3) 操作内容

```
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) 使用示例

```
地址
1000          MOV      #H'80,R1          ;R1=H'FFFFFF80
1002          MOV.W    IMM,R2            ;R2=H'FFFF9ABC  IMM 表示 @(H'08,PC)
1004          ADD      #-1,R0            ;
1006          TST      R0,R0             ;←MOV.W 指令时用于计算地址的 PC 位置
1008          MOVT     R13               ;
100A          BRA      NEXT              ; 延迟转移指令
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 的转移目标
1014          CMP/EQ   #0,R0             ;←MOV.L 指令时用于计算地址的 PC 位置
              .align   4                 ;
1018          .data.l  H'12345678        ;
```

6.1.34 MOV MOVe peripheral data 数据传送指令

外围模块数据的传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MOV.B @(disp,GBR),R0	(disp+GBR)→ 符号扩展 →R0	11000100dddddddd	1	—	○	○	○
MOV.W @(disp,GBR), R0	(disp×2+GBR)→ 符号扩展 →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) 说明

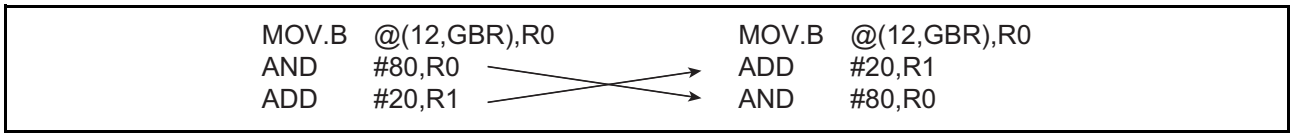
将源操作数传送至目标。适用于存取内部外围模块区域内的数据。可指定数据长度为字节、字或长字，寄存器固定为 R0。在 GBR 设定内部外围模块的基址。

内部外围模块的数据为字节长度时，仅零扩展 8 位位移量，因此可指定在 +255 字节范围内。内部外围模块的数据为字长度时，零扩展 8 位位移量后乘以 2，因此可指定在 +510 字节范围内。内部外围模块的数据为长字长度时，零扩展 8 位位移量后乘以 4，因此可指定在 +1020 字节范围内。未到达存储器操作数时，将 GBR 传送至通用寄存器后，需使用上述 @(R0,Rn) 模式。

源操作数为存储器时，将加载的数据符号扩展为长字后保存至寄存器。

(2) 注意

加载时，目标寄存器固定为 R0，因此，即使紧接着的指令要参考 R0，也必须等到加载指令执行结束。可通过改变指令顺序优化处理。



(3) 操作内容

```
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) 使用示例

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

6.1.35 MOV MOVe structure data 数据传送指令

结构体数据的传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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)→ 符号扩展 →R0	10000100mmmmdddd	1	—	○	○	○
MOV.W @(disp,Rm),R0	(disp×2+Rm)→ 符号扩展 →R0	10000101mmmmdddd	1	—	○	○	○
MOV.L @(disp,Rm),Rn	(disp×4+Rm)→Rn	0101nnnnmmmmdddd	1	—	○	○	○

(1) 说明

将源操作数传送至目标。适用于存取结构体和堆栈内的数据。可指定数据长度为字节、字或长字，为字节或字时，寄存器固定为 R0。

数据为字节长度时，仅零扩展 4 位位移量，因此可指定在 +15 字节范围内；数据为字长度时，零扩展 4 位位移量后乘以 2，因此可指定在 +30 字节范围内；数据为长字长度时，零扩展 4 位位移量后乘以 4，因此可指定在 +60 字节的范围。未到达存储器操作数时，需使用上述 @(R0,Rn) 模式。

源操作数为存储器时，将加载的数据符号扩展为长字后保存至寄存器。

(2) 注意

加载字节 / 字数据时，目标寄存器固定为 R0，因此，即使紧接着的指令要参考 R0，也必须等到加载指令执行结束。可通过改变指令顺序优化处理。

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

(3) 操作内容

```
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) 使用示例

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

6.1.36 MOVA MOVe effective Address 数据传送指令

有效地址的传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MOVA @(disp,PC),R0	disp×4 + PC→R0	11000111dddddddd	1	—	○	○	○

(1) 说明

将源操作数的有效地址保存至通用寄存器 R0。零扩展 8 位位移量后乘以 4，因此与操作数的相对距离在 PC+1020 字节范围内。 PC 为本指令 4 字节后的地址，将低 2 位修改为 B'00 的值用于地址计算。

(2) 注意

本指令配置在紧接着延迟转移指令之后时， PC 为转移目标的 “起始地址 +2”。

(3) 操作内容

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

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

(4) 使用示例

```
地址          .org        H'1006
1006          MOVA        STR,R0          ;STR 的地址 →R0
1008          MOV.B       @R0,R1          ;R1="X" ←PC 低 2 位修改后的位置
100A          ADD         R4,R5           ;← 计算 MOVA 指令的地址时， PC 的原位置
              .align      4
100C  STR:.sdata          "XYZP12"
    ・・・・・・・・
2002          BRA         TRGET           ; 延迟转移指令
2004          MOVA        @(0,PC),R0      ;TRGET 的地址 +2→R0
2006          NOP         ;
```

6.1.37 MOVT MOVe Tbit 数据传送指令

T 位的传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MOVT Rn	T→Rn	0000nnnn00101001	1	—	○	○	○

(1) 说明

将 T 位保存至通用寄存器 Rn。 T=1 时 Rn=1； T=0 时 Rn=0。

(2) 操作内容

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

(3) 使用示例

```
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 算术运算指令

双精度乘法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MUL.L Rm,Rn	Rn×Rm→MACL	0000nnnnmmmm0111	2 ～ 4	—	—	○	○

(1) 说明

将通用寄存器 Rn 的内容与 Rm 进行 32 位乘法运算，结果的低 32 位保存至 MACL 寄存器， MACH 的内容不变。

(2) 操作内容

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

(3) 使用示例

```
MUL.L       R0,R1       ; 执行前      R0=H'FFFFFFFE,R1=H'00005555
                        ; 执行后      MACL=H'FFFF5556
STS         MACL,R0     ; 得到运算结果
```

6.1.39 MULS.W MULtiply as Signed Word 算术运算指令

带符号的乘法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MULS.W Rm,Rn MULS Rm,Rn	带符号且 Rn×Rm→MACL	0010nnnnmmmm1111	1 ～ 3	—	○	○	○

(1) 说明

将通用寄存器 Rn 的内容与 Rm 进行 16 位乘法运算， 32 位结果保存至 MACL 寄存器。该运算为带符号的算术运算。 MACH 的内容不变。

(2) 操作内容

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

(3) 使用示例

```
MULS        R0,R1       ; 执行前 R0=H'FFFFFFFE,R1=H'00005555
                        ; 执行后 MACL=H'FFFF5556
STS         MACL,R0     ; 得到运算结果
```

6.1.40 MULU.W MULtiply as Unsigned Word 算术运算指令

无符号的乘法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MULU.W Rm,Rn MULU Rm,Rn	无符号 Rn×Rm→MACL	0010nnnnmmmm1110	1 ～ 3	—	○	○	○

(1) 说明

将通用寄存器 Rn 的内容与 Rm 进行 16 位乘法运算， 32 位结果保存至 MACL 寄存器。该运算为无符号的算术运算。 MACH 的内容不变。

(2) 操作内容

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

(3) 使用示例

```
MULU        R0,R1           ;执行前   R0=H'00000002,R1=H'FFFFAAAA
                            ;执行后   MACL=H'00015554
STS         MACL,R0         ;得到运算结果
```

6.1.41 NEG NEGate 算术运算指令

符号取反

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
NEG Rm,Rn	0–Rm→Rn	0110nnnnmmmm1011	1	—	○	○	○

(1) 说明

取通用寄存器 Rm 内容的 2 的补码，结果保存在 Rn。即从 0 减去 Rm，结果保存在 Rn。

(2) 操作内容

```
NEG(long m, long n)         /* NEG Rm,Rn */
{
     R[n]=0-R[m];
     PC+=2;
}
```

(3) 使用示例

```
NEG         R0,R1       ;执行前 R0=H'00000001
                        ;执行后 R1=H'FFFFFFFF
```

6.1.42 NEGC NEGate with Carry 算术运算指令

带借位的符号取反

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
NEGC Rm,Rn	0–Rm–T→Rn、借位→T	0110nnnnmmmm1010	1	借位	○	○	○

(1) 说明

从 0 减去通用寄存器 Rm 的内容及 T 位，结果保存在 Rn。根据运算结果在 T 位反映借位。本指令用于超过 32 位值的符号取反。

(2) 操作内容

```
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) 使用示例

```
CLRT            ;R0:R1(64 位 ) 的符号取反
NEGC  R1,R1     ; 执行前   R1=H'00000001,T=0
                ; 执行后   R1=H'FFFFFFFF,T=1
NEGC  R0,R0     ; 执行前   R0=H'00000000,T=1
                ; 执行后   R0=H'FFFFFFFF,T=1
```

6.1.43 NOP No OPeration 系统控制指令

空操作

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
NOP	空操作	0000000000001001	1	—	○	○	○

(1) 说明

仅执行 PC 递增，转移至执行下一条指令。

(2) 操作内容

```
NOP( )        /* NOP */
{
    PC+=2;
}
```

(3) 使用示例

```
NOP                ;经过 1 个周期。
```

6.1.44 NOT NOT-logical complement 逻辑运算指令

位取反

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
NOT Rm,Rn	～ Rm→Rn	0110nnnnmmmm0111	1	—	○	○	○

(1) 说明

取通用寄存器 Rm 内容的 1 的补码，将结果保存在 Rn。即 Rm 的位取反后保存至 Rn。

(2) 操作内容

```
NOT(long m, long n)       /* NOT Rm,Rn */
{
     R[n]=~R[m];
     PC+=2;
}
```

(3) 使用示例

```
NOT     R0,R1             ;执行前   R0=H'AAAAAAAA
                          ;执行后   R1=H'55555555
```

6.1.45 OR OR logical 逻辑运算指令

逻辑 “或” 运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

取通用寄存器 Rn 的内容与 Rm 的逻辑 “或”，结果保存在 Rn。

可执行通用寄存器 R0 与零扩展后的 8 位立即数的逻辑 “或” 运算，或对带变址的 GBR 间接寻址方式的 8 位存储器与 8 位立即数的逻辑 “或” 运算。

(2) 操作内容

```
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) 使用示例

```
OR          R0,R1           ;执行前 R0=H'AAAA5555,R1=H'55550000
                            ;执行后 R1=H'FFFF5555
OR          #H'F0,R0        ;执行前 R0=H'00000008
                            ;执行后 R0=H'000000F8
OR.B        #H'50,@(R0,GBR) ;执行前 @(R0,GBR)=H'A5
                            ;执行后 @(R0,GBR)=H'F5
```

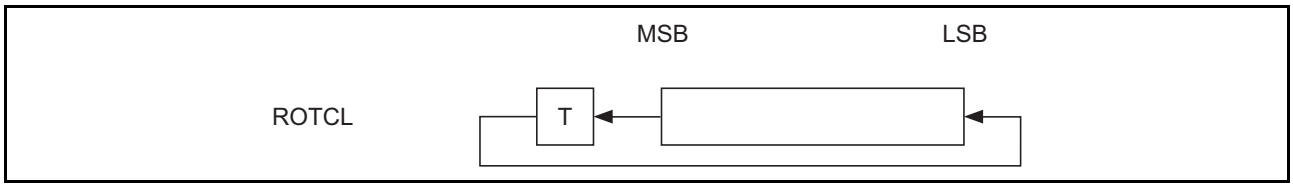
6.1.46 ROTCL ROTate with Carry Left 移位指令

带 T 位向左循环 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ROTCL Rn	T←Rn←T	0100nnnn00100100	1	MSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容 （包含 T 位）向左循环 1 位，结果保存在 Rn。循环后将移出操作数的位传送至 T 位。



(2) 操作内容

```
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) 使用示例

```
ROTCL      R0       ;执行前   R0=H'80000000,T=0
                    ;执行后   R0=H'00000000,T=1
```

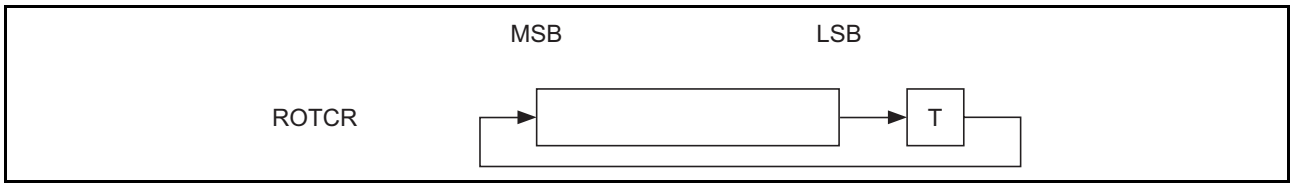
6.1.47 ROTCR ROTate with Carry Right 移位指令

带 T 位向右循环 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ROTCR Rn	T→Rn→T	0100nnnn00100101	1	LSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容（包含 T 位）向右循环 1 位，结果保存在 Rn。循环后将移出操作数的位传送至 T 位。



(2) 操作内容

```
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) 使用示例

```
ROTCR  R0           ;执行前  R0=H'00000001,T=1
                    ;执行后  R0=H'80000000,T=1
```

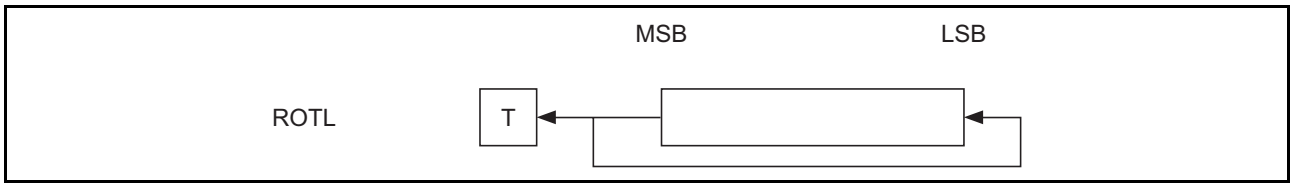
6.1.48 ROTL ROTate Left 移位指令

向左循环 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ROTL Rn	T←Rn←MSB	0100nnnn00000100	1	MSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容向左循环 1 位，结果保存在 Rn。循环后将移出操作数的位传送至 T 位。



(2) 操作内容

```
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) 使用示例

```
ROTL       R0      ;执行前 R0=H'80000000,T=0
                   ;执行后 R0=H'00000001,T=1
```

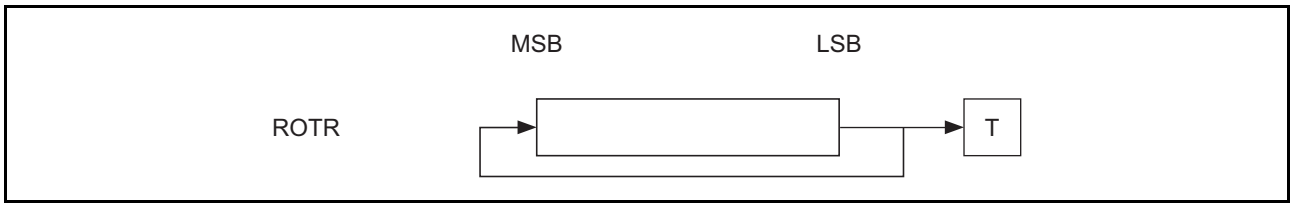
6.1.49 ROTR ROTate Right 移位指令

向右循环 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
ROTR Rn	LSB→Rn→T	0100nnnn00000101	1	LSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容向右循环 1 位，结果保存在 Rn。循环后将移出操作数的位传送至 T 位。



(2) 操作内容

```
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) 使用示例

```
ROTR      R0    ;执行前 R0=H'00000001,T=0
                ;执行后 R0=H'80000000,T=1
```

6.1.50 RTE ReTurn from Exception 系统控制指令

从异常处理返回 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
RTE	堆栈区 → PC/SR	0000000000101011	4	LSB	○	○	○

(1) 说明

从中断程序返回。即从堆栈返回 PC 与 SR 后，从返回的 PC 所示地址继续执行处理。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。
本指令与紧接着的指令之间，不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

```
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 及 SH-2 CPU 时

SH-DSP 时

(4) 使用示例

```
RTE                 ; 返回到原来的程序。
ADD #8,R14          ; 在转移前执行。
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.51 RTS ReTurn from SubRoutine 转移指令

从子程序过程返回 延迟转移指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
RTS	PR→PC	0000000000001011	2	—	○	○	○

(1) 说明

从子程序过程返回。即从 PR 返回 PC，从返回的 PC 所示地址继续处理。可根据本指令，从 BSR、 BSRF 及 JSR 指令调用的子程序过程返回至调用源。

(2) 注意

本指令为延迟转移指令，因此先执行紧接着本指令之后的指令，再转移。
本指令与紧接着的指令之间，不接受地址错误与中断。紧接着的指令为转移指令时，视为槽非法指令。

(3) 操作内容

```
RTS( )    /* RTS */
{
    unsigned long temp;

    temp=PC;
    PC=PR+4;
    Delay_Slot(temp+2);
}
```

(4) 使用示例

```
        MOV.L       TABLE,R3        ;R3=TRGET 的地址
        JSR         @R3             ; 转移至 TRGET。
        NOP                         ; 在 JSR 前执行。
        ADD         R0,R1           ;← 从子程序返回的目标 (PR 的内容 )
        ・・・・・・
TABLE:.data.l       TRGET           ; 跳转表
        ・・・・・・
TRGET:MOV           R1,R0           ;← 过程入口
        RTS                         ;PR 的内容 →PC
        MOV         #12,R0          ; 在转移前执行。
```

【注】 延迟转移的转移运行产生在槽指令执行后，但仍按照延迟转移指令 → 延迟槽指令的顺序执行指令 （寄存器的更新等）。例如，即使在延迟槽变更保存转移目标地址的寄存器，变更前的寄存器内容仍为转移目标地址。

6.1.52 SETRC SET repeat count to RC 系统控制指令

RC 计数器的设定及重复控制标志的设定

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SETRC Rm	Rm［11:0］→ RC(SR［27:16］)、重复控制标志 →RF1、 RF0	0100mmmm00010100	1	—	—	—	○
SETRC #imm	imm→RC(SR［23:16］)zeros→ SR［27:24］、重复控制标志 →RF1、 RF0	10000010iiiiiii0	1	—	—	—	○

(1) 说明

在 SR 寄存器的 RC 计数器设定重复次数。操作数为寄存器时，通过低 12 位指定重复次数；为立即数时，通过 8 位指定重复次数，此时 RC 的高 4 位为 0。

在 SR 寄存器的 RF1、 RF0 位设定重复控制标志。

使用 SETRC 指令时有一些限制，详细内容参照 “4.19 DSP 重复 （循环）控制”。

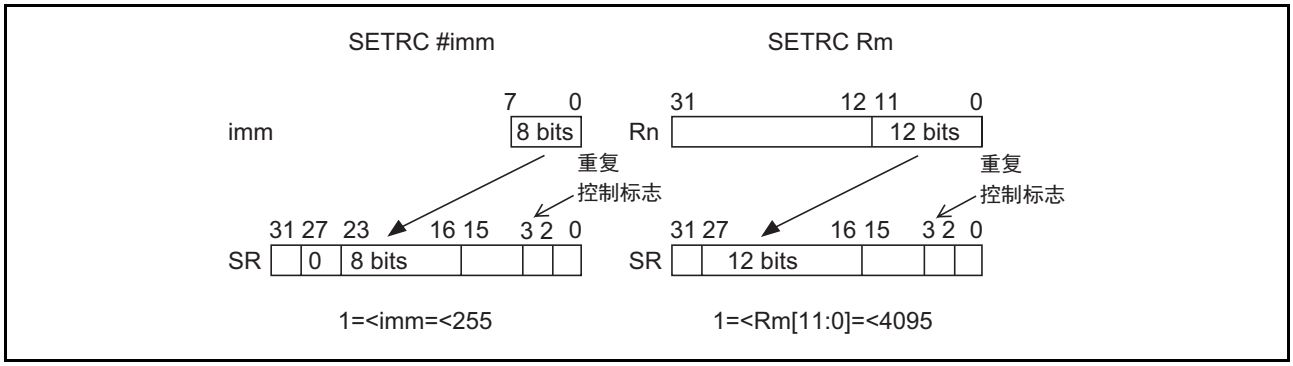
(2) 操作内容

```
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) 使用示例

```
        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 系统控制指令

T 位置位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SETT	1→T	0000000000011000	1	1	○	○	○

(1) 说明

置位 T 位。

(2) 操作内容

```
SETT( )                 /* SETT */
{
    T=1;
    PC+=2;
}
```

(3) 使用示例

```
SETT        ; 执行前    T=0
            ; 执行后    T=1
```

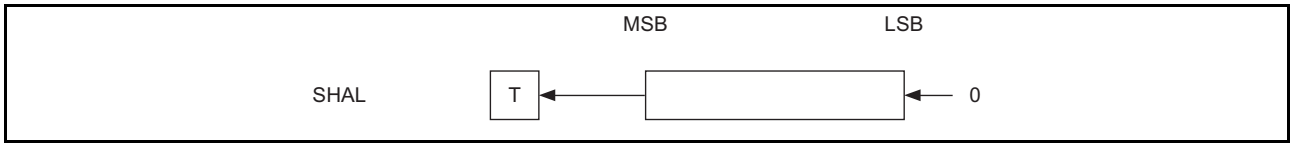
6.1.54 SHAL SHift Arithmetic Left 移位指令

算术左移 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SHAL Rn	T←Rn←0	0100nnnn00100000	1	MSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容算术左移 1 位，结果保存在 Rn。移位后将移出操作数的位传送至 T 位。



(2) 操作内容

```
SHAL(long n)        /* SHAL Rn(Same as SHLL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

(3) 使用示例

```
SHAL        R0          ; 执行前      R0=H'80000001,T=0
                        ; 执行后      R0=H'00000002,T=1
```

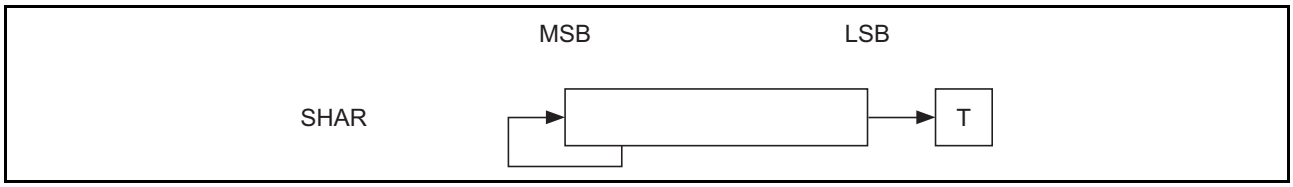
6.1.55 SHAR SHift Arithmetic Right 移位指令

算术右移 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SHAR Rn	MSB→Rn→T	0100nnnn00100001	1	LSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容算术右移 1 位，结果保存在 Rn。移位后将移出操作数的位传送至 T 位。



(2) 操作内容

```
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) 使用示例

```
SHAR    R0          ;执行前 R0=H'80000001,T=0
                    ;执行后 R0=H'C0000000,T=1
```

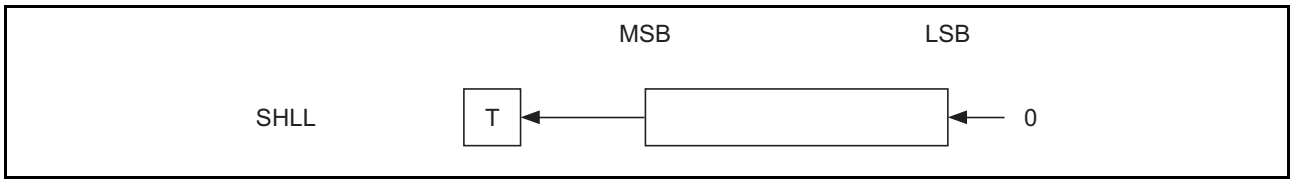
6.1.56 SHLL SHift Logical Left 移位指令

逻辑左移 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SHLL Rn	T←Rn←0	0100nnnn00000000	1	MSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容逻辑左移 1 位，结果保存在 Rn。移位后将移出操作数的位传送至 T 位。



(2) 操作内容

```
SHLL(long n)        /* SHLL Rn  (Same as SHAL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

(3) 使用示例

```
SHLL   R0          ;执行前 R0=H'80000001,T=0
                   ;执行后 R0=H'00000002,T=1
```

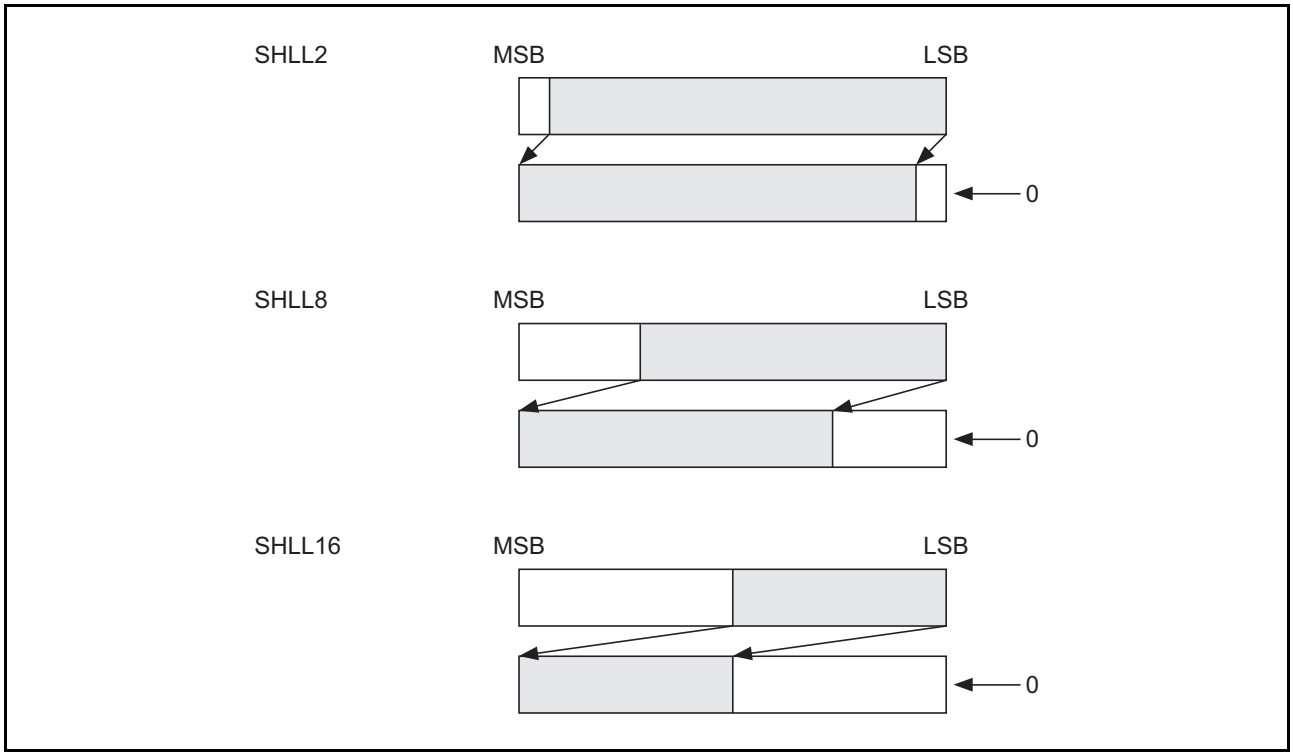
6.1.57 SHLLn n bits SHift Logical Left 移位指令

逻辑左移 n 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

将通用寄存器 Rn 的内容逻辑左移 2/8/16 位，结果保存在 Rn。舍弃移位后移出操作数的位。



(2) 操作内容

```
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) 使用示例

```
SHLL2   R0      ;执行前  R0=H'12345678
                ;执行后  R0=H'48D159E0
SHLL8   R0      ;执行前  R0=H'12345678
                ;执行后  R0=H'34567800
SHLL16  R0      ;执行前  R0=H'12345678
                ;执行后  R0=H'56780000
```

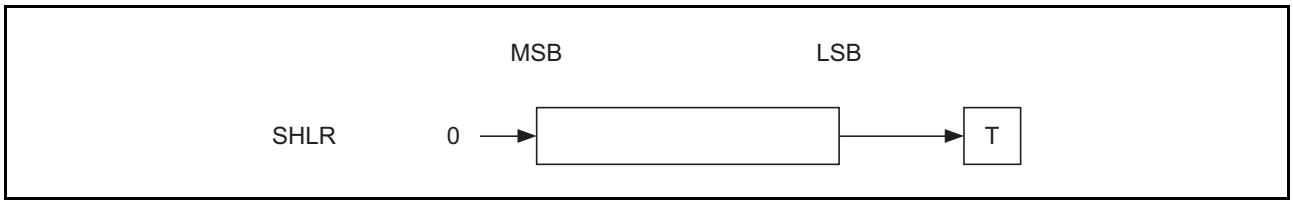
6.1.58 SHLR SHift Logical Right 移位指令

逻辑右移 1 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SHLR Rn	0→Rn→T	0100nnnn00000001	1	LSB	○	○	○

(1) 说明

将通用寄存器 Rn 的内容逻辑右移 1 位，结果保存在 Rn。移位后将移出操作数的位传送至 T 位。



(2) 操作内容

```
SHLR(long n)          /* SHLR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

(3) 使用示例

```
SHLR  R0              ;执行前  R0=H'80000001,T=0
                      ;执行后  R0=H'40000000,T=1
```

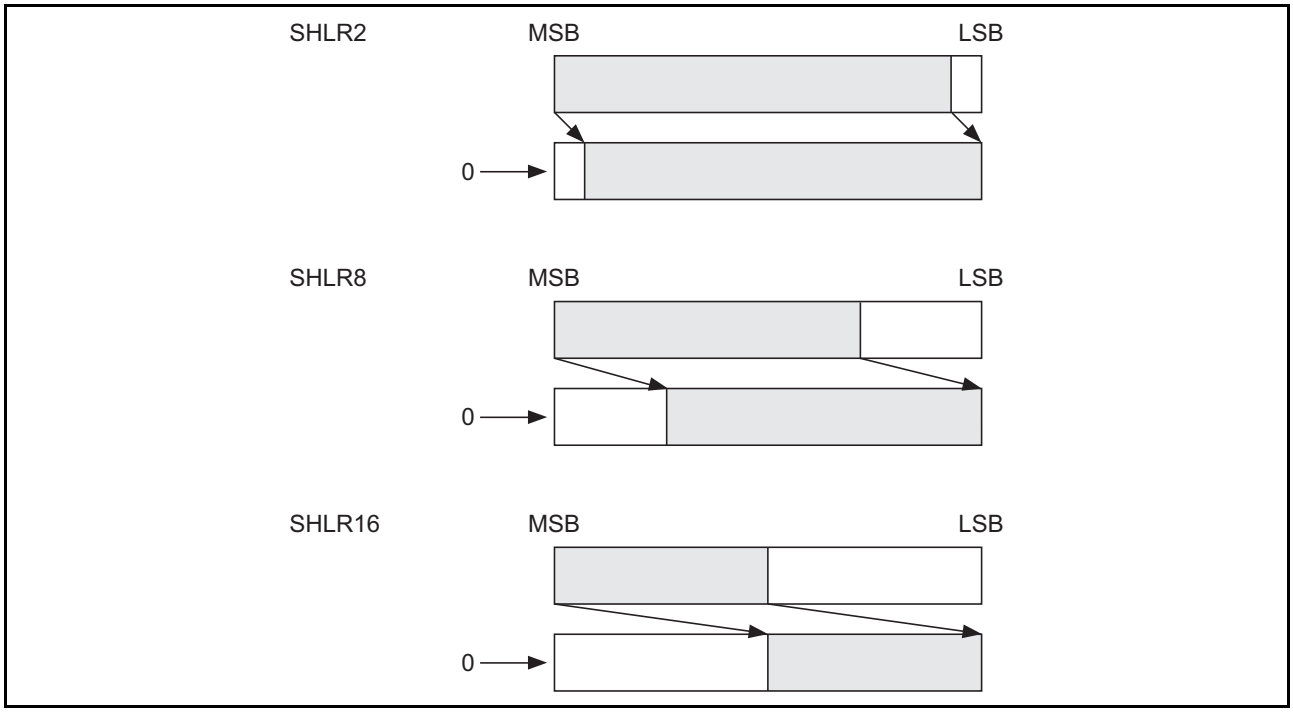
6.1.59 SHLRn n bits SHift Logical Right 移位指令

逻辑右移 n 位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

将通用寄存器 Rn 的内容逻辑右移 2/8/16 位，结果保存在 Rn。舍弃移位后移出操作数的位。



(2) 操作内容

```
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) 使用示例

```
SHLR2   R0              ;执行前 R0=H'12345678
                        ;执行后 R0=H'048D159E
SHLR8   R0              ;执行前 R0=H'12345678
                        ;执行后 R0=H'00123456
SHLR16  R0              ;执行前 R0=H'12345678
                        ;执行后 R0=H'00001234
```

6.1.60 SLEEP SLEEP 系统控制指令

向低功耗状态转移

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SLEEP	睡眠	0000000000011011	3	—	○	○	○

(1) 说明

将 CPU 设定为低功耗状态。

在低功耗状态，保持 CPU 的内部状态，停止执行下一条指令，等待产生中断请求，产生请求后，退出低功耗状态。

(2) 注意

执行状态栏的 3 为转移至睡眠模式所需的状态数。

(3) 操作内容

```
SLEEP( )          /* SLEEP */
{
    PC-=2;
    wait_for_exception;
}
```

(4) 使用示例

```
SLEEP        ;转移至低功耗状态
```

6.1.61 STC STore Control register 系统控制指令

从控制寄存器存储 中断禁止指令

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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) 说明

将控制寄存器 SR、 GBR、 VBR、 MOD、 RE、 SE 的数据保存至目标。

(2) 注意

本指令与紧接着的指令之间，不接受中断 （接受地址错误）。

(3) 操作内容

```
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) 使用示例

```
STC     SR,R0         ;执行前      R0=H'FFFFFFFF,SR=H'00000000
                      ;执行后      R0=H'00000000
STC.L   GBR,@-R15     ;执行前      R15=H'10000004
                      ;执行后      R15=H'10000000,@R15=GBR
```

6.1.62 STS STore System register 系统控制指令

从系统寄存器存储 中断禁止指令

格式		操作概略	操作码	执行状态	T 位	适用指令		
						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) 说明

将系统寄存器 MACH、 MACL、 PR 或 DSP 寄存器 DSR、 A0、 X0、 X1、 Y0、 Y1 保存至目标。

(2) 注意

本指令与紧接着的指令之间，不接受中断 （接受地址错误）。

SH-1 CPU 从 MACH 存储时，传送 bit9 的内容并保存在目标的高 22 位 （bit31 ～ bit10）； SH-2 CPU 及 SH-DSP 原样保存 MACH 的 32 位。

(3) 操作内容

```
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;
```

SH-1 CPU 时 （SH-2 CPU 及 SH-DSP 中不需要这 2 行。）

```
}

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;
}

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)                    SH-1 CPU 时
    Write_Long(R[n],MACH0x000003FF);
    else Write_Long(R[n], MACH|0xFFFFFC00);

    Write_Long(R[n],MACH);                      SH-2 CPU 及 SH-DSP 时
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) 使用示例

```
STS     MACH,R0   ;执行前    R0=H'FFFFFFFF,MACH=H'00000000
                  ;执行后    R0=H'00000000
STS.L   PR,@-R15  ;执行前    R15=H'10000004
                  ;执行后    R15=H'10000000,@R15=PR
```

6.1.63 SUB SUBtract binary 算术运算指令

2 进制减法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SUB Rm,Rn	Rn–Rm→Rn	0011nnnnmmmm1000	1	—	○	○	○

(1) 说明

通用寄存器 Rn 的内容减去 Rm，结果保存在 Rn。使用 ADD #imm,Rn，与立即数进行减法运算。

(2) 操作内容

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

(3) 使用示例

```
SUB        R0,R1          ;执行前   R0=H'00000001,R1=H'80000000
                          ;执行后   R1=H'7FFFFFFF
```

6.1.64 SUBC SUBtract with Carry 算术运算指令

带借位的 2 进制减法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SUBC Rm,Rn	Rn–Rm–T→Rn、借位 →T	0011nnnnmmmm1010	1	借位	○	○	○

(1) 说明

通用寄存器 Rn 的内容减去 Rm 与 T 位，结果保存在 Rn。根据运算结果在 T 位反映借位。本指令用于超过 32 位的减法运算。

(2) 操作内容

```
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) 使用示例

```
CLRT            ; R0:R1(64 位 )-R2:R3(64 位 )=R0:R1(64 位 )
SUBC    R3,R1   ;执行前    T=0,R1=H'00000000,R3=H'00000001
                ;执行后    T=1,R1=H'FFFFFFFF
SUBC    R2,R0   ;执行前    T=1,R0=H'00000000,R2=H'00000000
                ;执行后    T=1,R0=H'FFFFFFFF
```

6.1.65 SUBV SUBtract with (Vflag) underflow check 算术运算指令

带下溢的 2 进制减法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SUBV Rm,Rn	Rn–Rm→Rn、下溢 →T	0011nnnnmmmm1011	1	下溢	○	○	○

(1) 说明

通用寄存器 Rn 的内容减去 Rm，结果保存在 Rn。如果产生下溢，则 T 位置位。

(2) 操作内容

```
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) 使用示例

```
SUBV    R0,R1       ;执行前   R0=H'00000002,R1=H'80000001
                    ;执行后   R1=H'7FFFFFFF,T=1
SUBV    R2,R3       ;执行前   R2=H'FFFFFFFE,R3=H'7FFFFFFE
                    ;执行后   R3=H'80000000,T=1
```

6.1.66 SWAP SWAP register halves 数据传送指令

高位 / 低位的交换

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
SWAP.B Rm,Rn	Rm→ 低位 2 字节的高位 / 低位字节交换 →Rn	0110nnnnmmmm1000	1	—	○	○	○
SWAP.W Rm,Rn	Rm→ 高位 / 低位字交换 →Rn	0110nnnnmmmm1001	1	—	○	○	○

(1) 说明

交换通用寄存器 Rm 内容的高位 / 低位，结果保存在 Rn。

指定字节时，将 Rm 的 bit0 到 bit7 的 8 位与 bit8 到 bit15 的 8 位交换。将 Rm 的高 16 位原样传送至 Rn 的高 16 位。

指定字时，将 Rm 的 bit0 到 bit15 的 16 位与 bit16 到 bit31 的 16 位交换。

(2) 操作内容

```
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) 使用示例

```
SWAP.B     R0,R1     ;执行前  R0=H'12345678
                     ;执行后  R1=H'12347856
SWAP.W     R0,R1     ;执行前  R0=H'12345678
                     ;执行后  R1=H'56781234
```

6.1.67 TAS Test And Set 逻辑运算指令

存储器测试与置位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
TAS.B @Rn	(Rn) 为 0 时 1→T、 1→MSBof(Rn)	0100nnnn00011011	4	测试结果	○	○	○

(1) 说明

将通用寄存器 Rn 的内容作为地址，读取该地址显示的字节数据，该数据为 0 时， T=1 ；该数据不为 0 时， T=0。此后，将 bit7 置 1 后写入相同地址。在此期间，不释放总线权。

(2) 操作内容

```
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) 使用示例

```
    _LOOP     TAS.B     @R7       ;R7=1000
              BF        _LOOP     ; 重复直至地址 1000 变为 0。
```

6.1.68 TRAPA TRAP Always 系统控制指令

陷阱异常处理

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
TRAPA #imm	PC/SR→堆栈区、 (imm×4+VBR)→PC	11000011iiiiiiii	8	—	○	○	○

(1) 说明

开始陷阱异常处理。即，将 PC 与 SR 保存至堆栈区，并转移至指定向量所表示的地址。向量为 8 位立即数经零扩展后乘以 4 的存储器地址。 PC 为下一条指令的起始地址。

与 RTE 组合用于系统调用。

(2) 操作内容

```
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) 使用示例

```
地址
VBR+H'80        .data.l         10000000;
                .......
                TRAPA           #H'20     ; 转移至地址 VBR+H'80 内容的地址。
                TST             #0,R0     ; ← 从陷阱程序返回的地址
                .......                     （压栈后 PC 的内容）。
                .......
10000000        XOR             R0,R0     ; ← 陷阱程序入口
10000002        RTE                       ; 返回上述 TST。
10000004        NOP                       ; 在 RTE 前执行。
```

6.1.69 TST TeST logical 逻辑运算指令

逻辑 “与”运算的 T 位置位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
TST Rm,Rn	Rn&Rm、结果为 0 时 1→T	0010nnnnmmmm1000	1	测试结果	○	○	○
TST #imm,R0	R0&imm、结果为 0 时 1→T	11001000iiiiiiii	1	测试结果	○	○	○
TST.B #imm,@(R0,GBR)	(R0+GBR)&imm、结果为 0 时 1→T	11001100iiiiiiii	3	测试结果	○	○	○

(1) 说明

取通用寄存器 Rn 的内容及 Rm 的逻辑 “与”，结果为 0 时，置位 T 位；结果不为 0 时，清除 T 位。 Rn 的内容不变。

可取通用寄存器 R0 与零扩展后的 8 位立即数的逻辑 “与”或带变址的 GBR 间接寻址方式的 8 位存储器与 8 位立即数的逻辑 “与”。 R0 或存储器的内容不变。

(2) 操作内容

```
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) 使用示例

```
TST       R0,R0           ;执行前  R0=H'00000000
                          ;执行后  T=1
TST       #H'80,R0        ;执行前  R0=H'FFFFFF7F
                          ;执行后  T=1
TST.B     #H'A5,@(R0,GBR) ;执行前  @(R0,GBR)=H'A5
                          ;执行后  T=0
```

6.1.70 XOR eXclusive OR logical 逻辑运算指令

逻辑 “异或”运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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)说明

取通用寄存器 Rn 的内容与 Rm 的逻辑 “异或”，将结果保存在 Rn。

可取通用寄存器 R0 与零扩展后的 8 位立即数的逻辑 “异或”或带变址的 GBR 间接寻址方式的 8 位存储器与 8 位立即数的逻辑 “异或”。

(2) 操作内容

```
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) 使用示例

```
XOR     R0,R1             ;执行前  R0=H'AAAAAAAA,R1=H'55555555
                          ;执行后  R1=H'FFFFFFFF
XOR     #H'F0,R0          ;执行前  R0=H'FFFFFFFF
                          ;执行后  R0=H'FFFFFF0F
XOR.B   #H'A5,@(R0,GBR)   ;执行前  @(R0,GBR)=H'A5
                          ;执行后  @(R0,GBR)=H'00
```

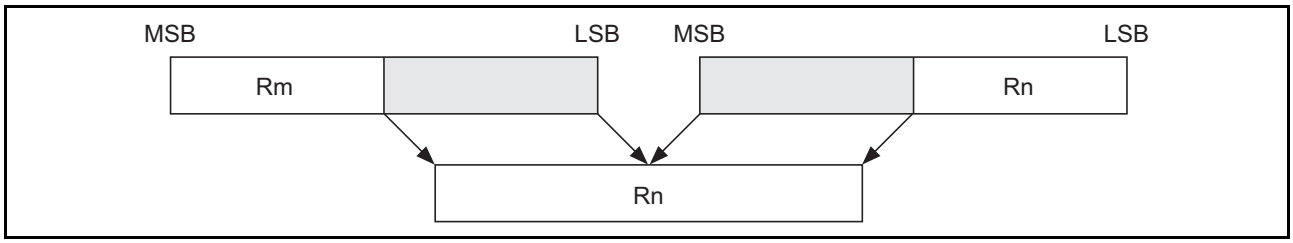
6.1.71 XTRCT eXTRaCT 数据传送指令

抽出连接寄存器的中间部分

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
XTRCT Rm,Rn	Rm: Rn 的中间 32 位 →Rn	0010nnnnmmmm1101	1	—	○	○	○

(1) 说明

抽出连接通用寄存器 Rm 与 Rn 的 64 位的中间 32 位，结果保存在 Rn。



(2) 操作内容

```
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) 使用示例

```
XTRCT    R0,R1          ;执行前  R0=H'01234567,R1=H'89ABCDEF
                        ;执行后  R1=H'456789AB
```

6.2 DSP 数据传送指令说明

说明 X、 Y 数据传送 （MOVX.W、 MOVY.W）、单数据传送 （MOVS.W、 MOVS.L）的运行。

(1) X、 Y 数据传送 （MOVX.W、 MOVY.W）

本指令使用 XDB 总线、YDB 总线存取 X、Y 存储器，不可存取 X、Y 存储器以外的区域。以字为单位存取存储器。由于使用独立的总线，与取指令 （使用主总线）不产生存取竞争。

同时并发执行的数据运算指令即使为带条件指令，也与条件无关，执行 X、 Y 数据传送指令。

X、 Y 数据传送的加载、存储运行如图 6.1 所示。

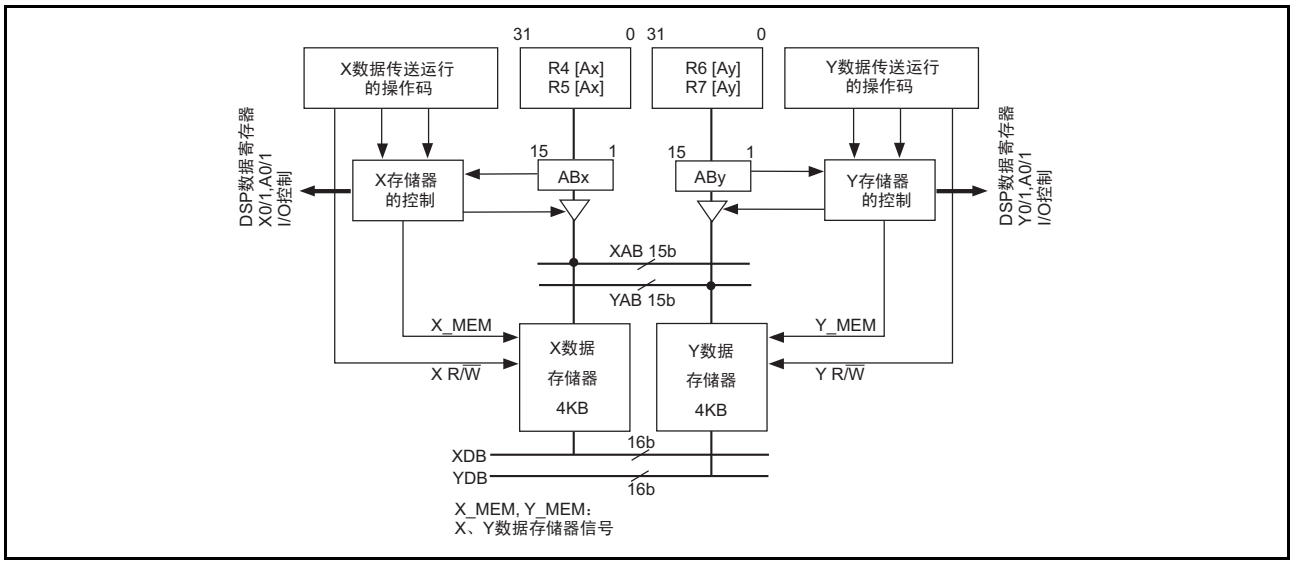


图 6.1 X、 Y 数据传送的加载、存储运行

X 存储器数据传送如下运行， Y 存储器数据传送相同。

```
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) 单数据传送 （MOVS.W、 MOVS.L）

单数据传送是加载、存储至 DSP 寄存器的指令，与系统寄存器的加载、存储指令相似。使用主总线传送存储器与 DSP 寄存器之间的数据。与 CPU 内核的指令相同，产生数据存取与指令存取的竞争。

单数据传送可传送字长度、长字长度的数据。

单数据传送的加载、存储运行如图 6.2 所示。

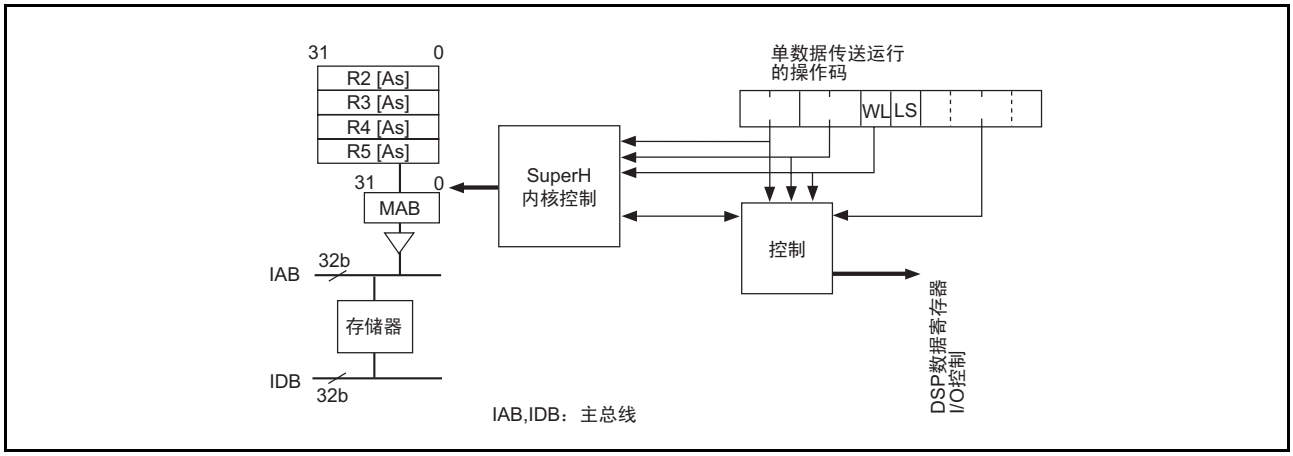


图 6.2 单数据传送的加载、存储运行

以字母顺序显示 DSP 数据传送指令。

指令名称 指令功能 （英文） 指令分类
指令功能 延迟转移指令或中断禁止指令的表示

格式	操作概略	操作码	执行状态	T 位	适用指令
表示汇编的输入格式。	表示操作概略。	按 MSB⇔LSB 顺序表示。	DSP 指令均为 1。	表示指令执行后的 DC 位的状态。	表示指令适用于 SH-1、 SH-2、 SH-DSP 中的哪个 CPU。

- 格式

[if cc] OP.Sz SRC1,SRC2,DEST

[if cc] ：条件 （无条件、 DCT 或 DCF）
OP ：操作助记符
Sz ：长度
SRC1 ：源 1 操作数
SRC2 ：源 2 操作数
DEST ：目标操作数

• 操作概略

→,←	：传送方向
(xx)	：存储器操作数
DC	：DSR 内的标志位
&	：位 “与”
|	：位 “或”
^	：位 “异或”
～	：位 “非”
<<n	：左移 n 位
>>n	：右移 n 位
MSW	：高位字 （bit 31 ～ 16）
LSW	：低位字 （bit 15 ～ 0）
[n1:n2]	：bit n1 ～ n2

• 操作码

表示源寄存器、目标寄存器。

X 数据传送指令

A(Ax): 0=R4, 1=R5
D(目标、 Dx): 0=X0, 1=X1
D(源、 Da): 0=A0, 1=A1

Y 数据传送指令

A(Ay): 0=R6, 1=R7
D(目标、 Dy): 0=Y0, 1=Y1
D(源、 Da): 0=A0, 1=A1

单数据传送指令

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 运算指令

iiiiiii(imm): PSHA 时 –32 ～ +32， PSHL 时 –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位

更新	：根据运算结果和指定的状态选择位 （CS）更新。
—	：不更新。

1. 说明
   说明指令的操作。

2. 注意
   说明使用指令时特别需要注意的事项。

3. 操作内容
   表示以C语言操作的内容。

4. 使用示例
   以汇编助记符举例，表示指令执行前后的状态。

6.2.1 MOVS MOVe Single data between memory and dsp register DSP 数据传送指令

单数据传送

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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.WDs,@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	—	—	—	○

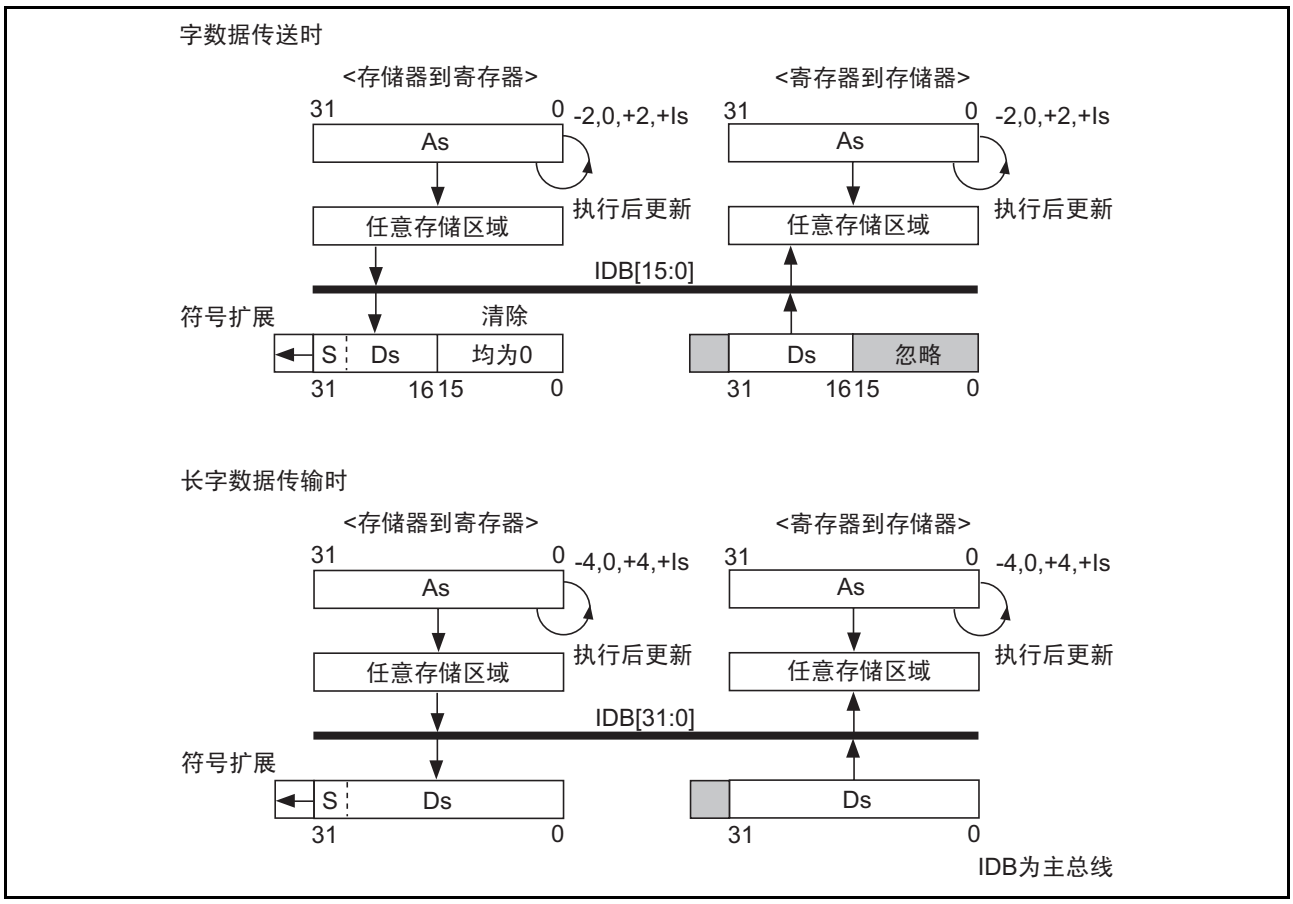
(1) 说明

向目标操作数传送源操作数的数据。从存储器向寄存器、从寄存器向存储器传送。可指定字长度、长字长度。如果为字传送，源操作数为存储器、目标操作数为寄存器时，字数据加载至寄存器的高位字，低位字清 0。源操作数为寄存器、目标操作数为存储器时，寄存器的高位字作为字数据存储。长字传送时传送长字数据。寄存器为目标操作数且有保护位时，符号扩展后保存在保护位。

(2) 注意

保护位寄存器 A0G、 A1G 的其中一个为存储处理的源操作数时，最低 8 位 （bit 7 ～ 0）输出数据，高 24 位 （bit 31 ～ 8）符号扩展。

(3) 操作内容



(4) 使用示例

```
MOVS.W   @R4 + ,A0          ; 执行前：R4=H'00000400,(R4)=H'8765,
                                    A0=H'123456789A
                              执行后：R4=H'00000402,A0=H'FF87650000
MOVS.L   A1,@-R3            ; 执行前：R3=H'00000800,A1=H'123456789A
                              执行后：R3=H'000007FC,
                                    (H'000007FC)=H'3456789A
```

6.2.2 MOVX MOVe between X memory and dsp register DSP 数据传送指令

X 存储器数据传送

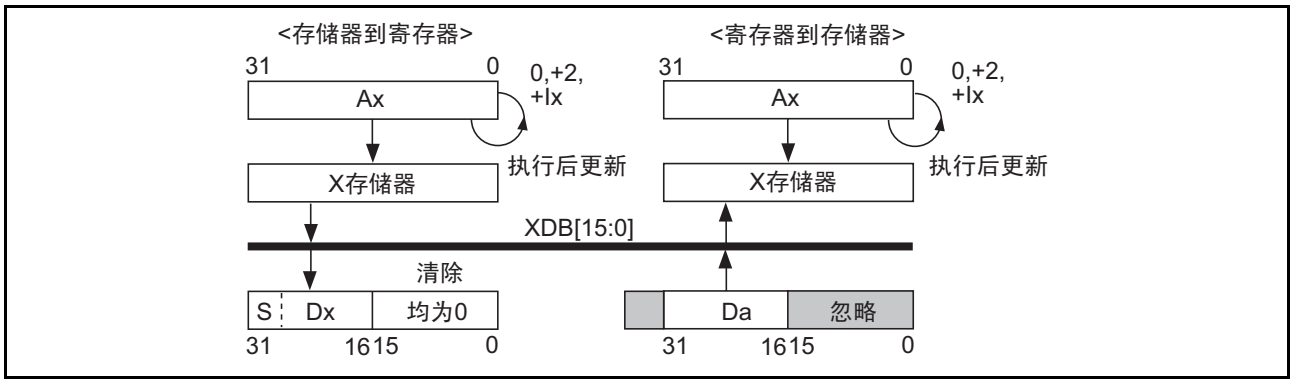
格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MOVX.W @Ax,Dx	(Ax)→MSW of Dx、0→LSW of Dx	111100A*D*0*01**	1	—	—	—	○
MOVX.W @Ax+,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax+2→Ax	111100A*D*0*10**	1	—	—	—	○
MOVX.W @Ax+Ix,Dx	(Ax)→MSW of Dx、0→LSW of Dx、Ax+Ix→Ax	111100A*D*0*11**	1	—	—	—	○
MOVX.W Da,@Ax	MSW of Da→(Ax)	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	—	—	—	○

操作码的 “*” 部分为 MOVY 指令指定区域。

(1) 说明

向目标操作数传送源操作数的数据。从存储器向寄存器、从寄存器向存储器传送。仅可指定 X 存储器和字长度。源操作数为存储器、目标操作数为寄存器时，字数据加载至寄存器的高位字，低位字清 0。源操作数为寄存器、目标操作数为存储器时，寄存器的高位字作为字数据存储。

(2) 操作内容



(3) 使用示例

```
MOVX.W @R4 + ,X0          ; 执行前：R4=H'08010000,(R4)=H'5555, X0=H'12345678
                            执行后：R4=H'08010002,X0=H'55550000
```

6.2.3 MOVY MOVe between Y memory and dsp register DSP 数据传送指令

Y 存储器数据传送

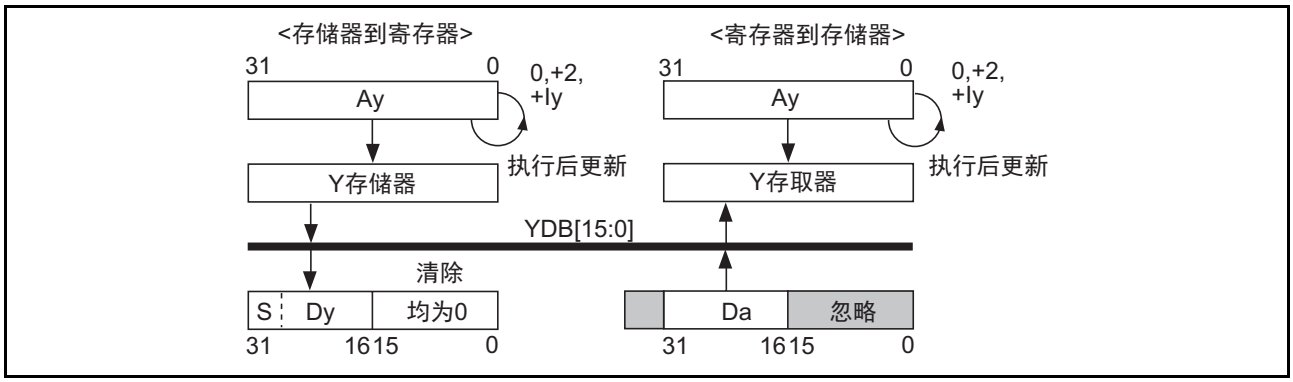
格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
MOVY.W @Ay,Dy	(Ay)→MSW of Dy、0→LSW of Dy	111100*A*D*0**01	1	—	—	—	○
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay+2→Ay	111100*A*D*0**10	1	—	—	—	○
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy、0→LSW of Dy、Ay+Iy→Ay	111100*A*D*0**11	1	—	—	—	○
MOVY.W Da,@Ay	MSW of Da→(Ay)	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	—	—	—	○

操作码的 “*” 部分为 MOVX 指令的指定区域。

(1) 说明

向目标操作数传送源操作数的数据。从存储器向寄存器、从寄存器向存储器传送。仅可指定 Y 存储器和字长度。源操作数为存储器、目标操作数为寄存器时，字数据加载至寄存器的高位字，低位字清 0。源操作数为寄存器、目标操作数为存储器时，寄存器的高位字作为字数据存储。

(2) 操作内容



(3) 使用示例

```
MOVY.W A0,@R6 + R9          ; 执行前：R6=H'08020000, R9=H'00000006,
                                    A0=H'123456789A
                            执行后：R6=H'08020006, (H'08020000)=H'3456
```

6.2.4 NOPX No access OPeration for X memory DSP 数据传送指令

X 存储器空操作

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
NOPX	No Operation	1111000*0*0*00**	1	—	—	—	○

操作码的 “*” 部分为 MOVY 指令的指定区域。

(1) 说明

X 存储器无存取操作。
该助记符可省略。

6.2.5 NOPY No access OPeration for Y memory DSP 数据传送指令

Y 存储器空操作

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
NOPY	No Operation	111100*0*0*0**00	1	—	—	—	○

操作码的 “*” 部分为 MOVX 指令的指定区域。

(1) 说明

Y 存储器无存取操作。
该助记符可省略。

6.3 DSP 运算指令说明

以与 CPU 指令相同的形式，说明 DSP 指令。说明使用 C 语言的操作内容时，假设使用以下的 DSP 资源。

(1) DSP 寄存器

DSP 寄存器名称以下述 DSP_Register_Set 的联合名为基础定义。该联合名由 11 个长字构成，各个长字对应 11 个 DSP 寄存器（A0、 A1、 M0、 M1、 X0、 X1、 Y0、 Y1、 AG0、 AG1、 DSR）。

```
/* Union DSP_Register_Set 的定义  */

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;
```

使用上述联合名 DSP_Register_Set，如下定义 DSP 寄存器的名称。

```
/* DSP Register 名称的定义 */

#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
```

DSR 寄存器的各位也同样使用联合名 DSP_Register_Set，如下定义。

```
#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 的输入 / 输出和表示运算结果的变量

ALU 的输入 / 输出以下述 DSP_ALU_Set 的联合名为基础定义。该联合名由 6 个长字构成，其中 3 个长字对应 2 个输入和 1 个输出 （src1、 src2、 dst），其余 3 个长字用于这 2 个输入和 1 个输出的保护位 （src1g、 src2g、 dstg）。

```
/* Union DSP_ALU_Set 的定义 */

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;
```

使用上述联合名 DSP_ALU_Set，如下定义 ALU 输入 / 输出的名称。

```
/* DSP 运算指令中 ALU 输入 / 输出的定义 */

#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
```

表示运算结果的变量也使用上述定义，如下进行定义。此变量在各指令操作内容的说明中，用于计算 DSR 寄存器内的 DC 位。

```
/* 表示 DSP 运算结果变量的定义 */

#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) 乘法器的输入 / 输出

乘法器的输入 / 输出以下述 DSP_MUL_Set 的联合名为基础定义。该联合名由 4 个长字构成。 2 个输入各分配 1 个长字，两者均仅可使用高 16 位 （usi ［0］、 usi ［2］）。输出对应包含保护位的 2 个长字 （dst、dstg）。

```
* Union DSP_MUL_Set 的定义  */

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;
```

使用上述联合名 DSP_MUL_Set，如下定义乘法器输入 / 输出的名称。

```
/* DSP 运算指令中乘法器输入 / 输出的定义 */

#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) 其他指令操作内容说明中使用的变量等

说明可指定 DCT、 DCF 条件的 DSP 运算指令的操作内容时，使用以下变量。

```
#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))
```

上述定义中，EX_DCT 和 EX_DCF 分别为指令指定 DCT、DCF 条件时，为真变量。DSPDCBIT 详细内容参照 “(1) DSP 寄存器”。

SR 寄存器的饱和位为 1 时， DSP 的算术运算执行饱和处理。说明操作内容时，称该饱和位为 SBIT。

为了简化说明操作内容，定义可通用的以下函数。

```
/* DSP 运算指令的说明中可通用的函数 */

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 */
    }
}
```

以下 6 个函数用于更新 DSR 寄存器。根据 DSP 运算指令的运算结果和指示的状态选择位 （CS）更新 DSR 寄存器内的 DC 位，其他位仅根据 DSP 运算指令的运算结果更新。

```
/* 无条件地根据借位标志更新 DC 位 (DSPDCBIT) 的函数 */
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;
}

/* 无条件地根据进位标志更新 DC 位 (DSPDCBIT) 的函数 */
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;
}

/* 减法时，更新 DC 位 (DSPDCBIT) 的函数 */
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;
}

/* 加法时，更新 DC 位 (DSPDCBIT) 的函数 */
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;
}
```

```
/* 逻辑运算时，更新 DC 位 (DSPDCBIT) 的函数 */
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;
}
```

以字母顺序表示 DSP 运算指令的详细指令。

6.3.1 PABS ABSolute DSP 算术运算指令

绝对值运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PABS Sx,Dz	Sx ≥ 0 时，则 Sx→Dz Sx < 0 时，则 0–Sx→Dz	111110********** 10001000xx00zzzz	1	更新	—	—	○
PABS Sy,Dz	Sy ≥ 0 时，则 Sy→Dz Sy < 0 时，则 0–Sy→Dz	111110********** 1010100000yyzzzz	1	更新	—	—	○

(1) 说明

求绝对值。 Sx、 Sy 操作数的内容为正值时，保存至 Dz 操作数。为负值时，符号取反后保存至 Dz 操作数。

根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。

(2) 操作内容

```
/*        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) 使用示例

```
PABS X0,M0  NOPX  NOPY      ; 执行前：X0=H'33333333, M0=H'12345678
                             执行后：X0=H'33333333, M0=H'33333333
PABS X1,X1  NOPX  NOPY      ; 执行前：X1=H'DDDDDDDD
                            执行后：X1=H'22222223
                            根据 CS［2:0］的状态更新 DC 位。
```

6.3.2 [if cc] PADD ADDition with Condition DSP 算术运算指令

带条件的加法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PADD Sx,Sy,Dz	Sx+Sy→Dz	111110**********	1	更新	—	—	○
		10110001xxyyzzzz					
DCT PADD Sx,Sy,Dz	DC = 1 时，则 Sx+Sy→Dz	111110**********	1	—	—	—	○
	为 0 时，则 nop.						
DCF PADD Sx,Sy,Dz	DC = 0 时，则 Sx+Sy→Dz	10110010xxyyzzzz					
	为 1 时，则 nop.	111110**********					
		10110011xxyyzzzz	1	—	—	—	○

(1) 说明

Sx、 Sy 操作数的内容相加，结果保存至 Dz 操作数。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*   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) 使用示例

```
PADD X0,Y0,A0  NOPX  NOPY    ; 执行前：X0=H'22222222, Y0=H'33333333,
                                       A0=H'123456789A
                               执行后：X0=H'22222222, Y0=H'33333333,
                                       A0=H'0055555555
                               无条件执行时，根据运算前 CS［2:0］位的状态更新 DC 位。
```

6.3.3 PADD PMULS ADDition & MULtiply Signed by Signed DSP 算术运算指令

加法和带符号的乘法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx+Sy→Du Se 的高位字 ×Sf 的高位字 →Dg	111110********** 0111eeffxxyygguu	1	更新	—	—	○

(1) 说明

Sx、 Sy 操作数的内容相加，结果保存至 Du 操作数。 Se、 Sf 操作数的高位字的内容 （带符号）相乘，结果保存至 Dg 操作数。这 2 个处理可同时并发执行。

根据 ALU 运算的结果和 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也根据 ALU 运算的结果更新。

(2) 注意

因 PMULS 为定点乘法运算，即使源数据相同，也与 MULS 的运算结果不同。

(3) 操作内容

```
/*   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) 使用示例

```
PADD A0,M0,A0 PMULS X0,Y0,M0  NOPX  NOPY;
```

执行前：X0=H'00020000,Y0=H'00030000,
M0=H'22222222,A0=H'0055555555

执行后：X0=H'00020000,Y0=H'00030000,
M0=H'0000000C,A0=H'0077777777

根据 CS［2:0］的状态及 PADD 运行结果更新 DC 位。

6.3.4 PADDC ADDition with Carry DSP 算术运算指令

带进位的加法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PADDC Sx,Sy,Dz	Sx+Sy+DC→Dz	111110********** 10110000xxyyzzzz	1	进位	—	—	○

(1) 说明

Sx、 Sy 操作数的内容与 DC 位相加，结果保存至 Dz 操作数。
DSR 寄存器的 DC 位作为进位标志更新。 DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。

(2) 注意

执行 PADDC 指令后的 DC 位作为进位标志更新，与 CS 位无关。

(3) 操作内容

```
/*     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) 使用示例

CS[2:0]=***: 与 CS 位的状态无关，总是作为 Carry or Borrow Mode 运行。

```
PADDC X0,Y0,M0  NOPX  NOPY;
```

执行前： X0=H'B3333333, Y0=H'55555555
M0=H'12345678, DC=0

执行后： X0=H'B3333333, Y0=H'55555555
M0=H'08888888, DC=1

```
PADDC X0,Y0,M0  NOPX  NOPY;
```

执行前： X0=H'33333333, Y0=H'55555555
M0=H'12345678, DC=1

执行后： X0=H'33333333, Y0=H'55555555
M0=H'88888889, DC=0

根据 CS[2:0] 位的状态更新 DC 位。

6.3.5 [if cc] PAND logical AND DSP 逻辑运算指令

带条件的逻辑 “与”运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PAND Sx,Sy,Dz	Sx & Sy→Dz，清除 Dz 的低位字	111110********** 10010101xxyyzzzz	1	更新	—	—	○
DCT PAND Sx,Sy,Dz	DC = 1 时， Sx&Sy→Dz，清除 Dz 的低位字 为 0 时， nop	111110********** 10010110xxyyzzzz	1	—	—	—	○
DCF PAND Sx,Sy,Dz	DC = 0 时，Sx&Sy→Dz，清除 Dz 的低位字 为 1， nop	111110********** 10010111xxyyzzzz	1	—	—	—	○

(1) 说明

执行 Sx 操作数高位字内容和 Sy 操作数高位字内容的逻辑 “与”运算，结果保存至 Dz 操作数的高位字，Dz 操作数的低位字清 0。 Dz 为有保护位的寄存器时，保护位也清 0。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 注意

更新 DC 位时忽略目标寄存器低位字的内容和保护位的内容。

(3) 操作内容

```
/*      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) 使用示例

```
PAND X0,Y0,A0  NOPX  NOPY ;
```

执行前： X0=H'33333333, Y0=H'55555555
A0=H'123456789A

执行后： X0=H'33333333, Y0=H'55555555
A0=H'0011110000

无条件执行时，根据 CS［2:0］的状态更新 DC 位。

6.3.6 [if cc] PCLR CLeaR DSP 算术运算指令

带条件的清除

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PCLR Dz	H'00000000→Dz	111110********** 1000110100000zzzz	1	更新	—	—	○
DCT PCLR Dz	DC = 1 时， H'00000000→Dz 为 0 时， nop	111110********** 1000111000000zzzz	1	—	—	—	○
DCF PCLR Dz	DC = 0 时， H'00000000→Dz 为 1 时， nop	111110********** 1000111100000zzzz	1	—	—	—	○

(1) 说明

Dz 操作数的内容清 0。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位。 DSR 寄存器的 Z 位置 1， N、 V、 GT 位清 0。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*      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) 使用示例

```
PCLR A0  NOPX  NOPY ;          执行前 :A0=H'FF87654321
                               执行后 :A0=H'0000000000
                               无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.7 PCMP CoMPare two data DSP 算术运算指令

比较

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PCMP Sx,Sy	Sx–Sy	111110********** 10000100xxyy0000	1	更新	—	—	○

(1) 说明

Sx 操作数的内容减 Sy 操作数的内容。

根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。

(2) 操作内容

```
/*    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) 使用示例

```
PCMP X0,Y0  NOPX  NOPY          ; 执行前 :X0=H'22222222, Y0=H'33333333
                                  执行后 :X0=H'22222222, Y0=H'33333333
                                         N=1,Z=0,V=0,GT=0
                                  根据 CS［2:0］的状态更新 DC 位。
```

6.3.8 [if cc] PCOPY COPY with Condition DSP 算术运算指令

带条件的转存

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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	DC = 1 时， Sx→Dz 为 0 时， nop	111110********** 11011010xx00zzzz	1	—	—	—	○
DCT PCOPY Sy,Dz	DC = 1 时， Sy→Dz 为 0 时， nop	111110********** 1111101000yyzzzz	1	—	—	—	○
DCF PCOPY Sx,Dz	DC = 0 时， Sx→Dz 为 1 时， nop	111110********** 11011011xx00zzzz	1	—	—	—	○
DCF PCOPY Sy,Dz	DC = 0 时， Sy→Dz 为 1 时， nop	111110********** 1111101100yyzzzz	1	—	—	—	○

(1) 说明

Sx、 Sy 操作数的内容保存至 Dz 操作数。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*  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) 使用示例

```
PCOPY X0,A0  NOPX  NOPY              ; 执行前 :X0=H'55555555, A0=H'FFFFFFFFFF
                                       执行后 :X0=H'55555555, A0=H'0055555555
                                     无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.9 [if cc] PDEC DECrement by 1 DSP 算术运算指令

带条件的递减

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PDEC Sx,Dz	Sx 的高位字 – 1→Dz 的高位字，清除 Dz 的低位字	111110********** 10001001xx00zzzz	1	更新	—	—	○
PDEC Sy,Dz	Sy 的高位字 – 1→Dz 的高位字，清除 Dz 的低位字	111110********** 1010100100yyzzzz	1	更新	—	—	○
DCT PDEC Sx,Dz	DC = 1 时， Sx 的高位字 – 1→Dz 的高位字，清除 Dz 的低位字 为 0 时， nop	111110********** 10001010xx00zzzz	1	—	—	—	○
DCT PDEC Sy,Dz	DC = 1 时， Sy 的高位字 – 1→Dz 的高位字，清除 Dz 的低位字 为 0 时， nop	111110********** 1010101000yyzzzz	1	—	—	—	○
DCF PDEC Sx,Dz	DC = 0 时， Sx 的高位字 – 1→Dz 的高位字，清除 Dz 的低位字 为 1 时， nop	111110********** 10001011xx00zzzz	1	—	—	—	○
DCF PDEC Sy,Dz	DC = 0 时， Sy 的高位字 – 1→Dz 的高位字，清除 Dz 的低位字 为 1 时， nop	111110********** 1010101100yyzzzz	1	—	—	—	○

(1) 说明

Sx、 Sy 操作数高位字的内容减 1，结果保存至 Dz 操作数的高位字， Dz 操作数的低位字清 0。指定 DCT、DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 注意

更新 DC 位时忽略目标寄存器低位字的内容。

(3) 操作内容

```
/*    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) 使用示例

```
PDEC X0,M0  NOPX  NOPY          ;执行前： X0=H'0052330F, M0=H'12345678
                                 执行后： X0=H'0052330F, M0=H'00510000
PDEC X1,X1  NOPX  NOPY          ;执行前： X1=H'FC342855
                                 执行后： X1=H'FC330000
                                 无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.10 [if cc] PDMSB Detect MSB with Condition DSP 算术运算指令

带条件的 MSB 检测

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PDMSB Sx,Dz	Sx 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字	111110********** 10011101xx00zzzz	1	更新	—	—	○
PDMSB Sy,Dz	Sy 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字	111110********** 1011110100yyzzzz	1	更新	—	—	○
DCT PDMSB Sx,Dz	DC = 1 时， Sx 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 0 时， nop	111110********** 10011110xx00zzzz	1	—	—	—	○
DCT PDMSB Sy,Dz	DC = 1 时， Sy 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 0 时， nop	111110********** 1011111000yyzzzz	1	—	—	—	○
DCF PDMSB Sx,Dz	DC = 0 时， Sx 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 1 时， nop	111110********** 10011111xx00zzzz	1	—	—	—	○
DCF PDMSB Sy,Dz	DC = 0 时， Sy 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 1 时， nop	111110********** 1011111100yyzzzz	1	—	—	—	○

(1) 说明

检测 Sx、 Sy 操作数位排列最先改变的位置，将其位的位置保存至 Dz 操作数。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*     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) 使用示例

```
PDMSB X0,M0  NOPX  NOPY    ;        执行前： X0=H'0052330F, M0=H'12345678
                                    执行后： X0=H'0052330F, M0=H'00080000
PDMSB X1,X1  NOPX  NOPY    ;        执行前： X1=H'FC342855
                                    执行后： X1=H'00050000
                                    无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.11 [if cc] PINC INCrement by 1 with Condition DSP 算术运算指令

带条件的递增

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PINC Sx,Dz	Sx 的高位字 +1→Dz 的高位字，清除 Dz 的低位字	111110********** 10011001xx00zzzz	1	更新	—	—	○
PINC Sy,Dz	Sy 的高位字 +1→Dz 的高位字，清除 Dz 的低位字	111110********** 1011100100yyzzzz	1	更新	—	—	○
DCT PINC Sx,Dz	DC = 1 时， Sx 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 0 时， nop	111110********** 10011010xx00zzzz	1	—	—	—	○
DCT PINC Sy,Dz	DC = 1 时， Sy 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 0 时， nop	111110********** 1011101000yyzzzz	1	—	—	—	○
DCF PINC Sx,Dz	DC = 0 时， Sx 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 1 时， nop	111110********** 10011011xx00zzzz	1	—	—	—	○
DCF PINC Sy,Dz	DC = 0 时， Sy 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 1 时， nop	111110********** 1011101100yyzzzz	1	—	—	—	○

(1) 说明

Sx、 Sy 操作数高位字的内容加 1，结果保存至 Dz 操作数的高位字， Dz 操作数的低位字清 0。指定 DCT、DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 注意

更新 DC 位时忽略目标寄存器低位字。

(3) 操作内容

```
/*   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) 使用示例

```
PINC X0,M0  NOPX  NOPY  ;
PINC X1,X1  NOPX  NOPY  ;
```

执行前：X0=H'0052330F, M0=H'12345678

执行后：X0=H'0052330F, M0=H'00530000

执行前：X1=H'FC342855

执行后：X1=H'FC350000

无条件执行时，根据 CS［2:0］的状态更新 DC 位。

6.3.12 [if cc] PLDS LoaD System register DSP 系统控制指令

加载至带条件的系统寄存器

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PLDS Dz,MACH	Dz→MACH	111110********** 111011010000zzzz	1	—	—	—	○
PLDS Dz,MACL	Dz→MACL	111110********** 111111010000zzzz	1	—	—	—	○
DCT PLDS Dz,MACH	DC = 1 时，Dz→MACH 为 0 时， nop	111110********** 111011100000zzzz	1	—	—	—	○
DCT PLDS Dz,MACL	DC = 1 时， Dz→MACL 为 0 时， nop	111110********** 111111100000zzzz	1	—	—	—	○
DCF PLDS Dz,MACH	DC = 0 时， Dz→MACH 为 1 时， nop	111110********** 111011110000zzzz	1	—	—	—	○
DCF PLDS Dz,MACL	DC = 0 时， Dz→MACL 为 1 时， nop	111110********** 111111110000zzzz	1	—	—	—	○

(1) 说明

Dz 操作数的内容保存至 MACH、 MACL 寄存器。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

DSR 寄存器的 DC、 N、 Z、 V、 GT 位均不更新。

(2) 注意

可指定 PLDS 与 MOVX、 MOVY 并发执行，但执行可能需要 2 个周期。

(3) 操作内容

```
/*      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) 使用示例

```
  PLDS A0,MACH  NOPX  NOPY            ; 执行前： A0=H'123456789A,
                                              MACH=H'66666666
                                        执行后： A0=H'123456789A,
                                              MACH=H'3456789A
```

6.3.13 PMULS MULtiply Signed by Signed DSP 算术运算指令

带符号乘法运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PMULS Se,Sf,Dg	Se 的高位字 ×Sf 的高位字 →Dg	111110********** 0100eeff0000gg00	1	—	—	—	○

(1) 说明

对 Se、 Sf 操作数高位字的内容进行带符号乘法运算，结果保存至 Dg 操作数。
DSR 寄存器的 DC、 N、 Z、 V、 GT 位均不更新。

(2) 注意

PMULS 为定点乘法运算，即使源数据相同，也与整数乘法 MULS 的运算结果不同。

【注】 视为整数数据时，运算结果为 2 倍的值。

(3) 操作内容

```
/*      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) 使用示例

```
PMULS X0,Y0,M0  NOPX  NOPY  ; 执行前： X0=H'00010000,Y0=H'00020000,
                                        （2-15）          （2-14）
                              M0=H'33333333
                            执行后： X0=H'00010000,Y0=H'00020000,
                              M0=H'00000004← 视为整数数据时，
                                     （2-29）  为 2 倍的值。
PMULS X1,Y1,A0  NOPX  NOPY  ; 执行前： X1=H'FFFE2222,Y1=H'0001AAAA,
                              A0=H'4444444444
                            执行后： X1=H'FFFE2222,Y1=H'0001AAAA,
                              A0=H'FFFFFFFFFC

                            （  ）：定点值
```

6.3.14 [if cc] PNEG NEGate DSP 算术运算指令

带条件的符号取反

格式	操作概略	操作码	执行状态	T 位	适用指令		
					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	DC = 1 时， 0– Sx→Dz 为 0 时， nop	111110********** 11001010xx00zzzz	1	—	—	—	○
DCT PNEG Sy,Dz	DC = 1 时， 0– Sy→Dz 为 0 时， nop	111110********** 1110101000yyzzzz	1	—	—	—	○
DCF PNEG Sx,Dz	DC =时， 0– Sx→Dz 为 1 时， nop	111110********** 11001011xx00zzzz	1	—	—	—	○
DCF PNEG Sy,Dz	DC = 0 时， 0– Sy→Dz 为 1 时， nop	111110********** 1110101100yyzzzz	1	—	—	—	○

(1) 说明

符号取反。 0 减去 Sx、 Sy 操作数的内容，结果保存至 Dz 操作数。指定 DCT、 DCF 条件时，如果条件为真，则执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*    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) 使用示例

```
PNEG X0,A0  NOPX  NOPY              ;执行前： X0=H'55555555,A0=H'A987654321
                                     执行后： X0=H'55555555,A0=H'FFAAAAAAAB
PNEG Y1,Y1  NOPX  NOPY              ;执行前： Y1=H'99999999
                                     执行后： Y1=H'66666667
                                    无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.15 [if cc] POR logical OR DSP 逻辑运算指令

带条件的逻辑 “或” 运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
POR Sx,Sy,Dz	Sx | Sy→Dz，清除 Dz 的低位字	111110********** 10110101xxyyzzzz	1	更新	—	—	○
DCT POR Sx,Sy,Dz	DC = 1 时，Sx|Sy→Dz，清除 Dz 的低位字 为 0 时， nop	111110********** 10110110xxyyzzzz	1	—	—	—	○
DCF POR Sx,Sy,Dz	DC = 0 时，Sx|Sy→Dz，清除 Dz 的低位字 为 1 时， nop	111110********** 10110111xxyyzzzz	1	—	—	—	○

(1) 说明

执行 Sx 操作数高位字的内容和 Sy 操作数高位字的内容的逻辑 “或” 运算，结果保存至 Dz 操作数的高位字， Dz 操作数的低位字清 0。 Dz 操作数为有保护位的寄存器时，保护位也清 0。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 注意

更新 DC 位时忽略目标寄存器低位字的内容和保护位的内容。

(3) 操作内容

```
/*    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) 使用示例

```
POR X0,Y0,A0  NOPX  NOPY  ;
```

执行前 :X0=H'33333333,Y0=H'55555555
A0=H'123456789A

执行后 :X0=H'33333333, Y0=H'55555555
A0=H'0077770000

无条件执行时，根据 CS［2:0］的状态更新 DC 位。

6.3.16 PRND RouNDing DSP 算术运算指令

舍入运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PRND Sx,Dz	Sx+H'00008000→Dz 清除 Dz 的低位字	111110********** 10011000xx00zzzz	1	更新	—	—	○
PRND Sy,Dz	Sy+H'00008000→Dz 清除 Dz 的低位字	111110********** 1011100000yyzzzz	1	更新	—	—	○

(1) 说明

执行舍入。 Sx、 Sy 操作数的内容加立即数 H'00008000，结果的高位字保存至 Dz 操作数， Dz 操作数的低位字清 0。

根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。

(2) 操作内容

```
/*      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) 使用示例

```
PRND X0,M0  NOPX  NOPY  ;
PRND X1,X1  NOPX  NOPY  ;
```

执行前 :X0=H'0052330F, M0=H'12345678
执行后 :X0=H'0052330F, M0=H'00520000
执行前 :X1=H'FC34C087
执行后 :X1=H'FC350000
根据 CS［2:0］的状态更新 DC 位。

6.3.17 [if cc] PSHA SHift Arithmetically with Condition DSP 算术移位指令

带条件的算术移位

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PSHA Sx,Sy,Dz	Sy ≥ 0 时， Sx << Sy→Dz Sy < 0 时， Sx >> | Sy | →Dz	111110********** 10010001xxyyzzzz	1	更新	—	—	○
DCT PSHA Sx,Sy,Dz	DC = 1 & Sy ≥ 0 时，Sx << Sy→Dz DC = 1 & Sy < 0 时，Sx >> | Sy | →Dz DC = 0 时， nop	111110********** 10010010xxyyzzzz	1	—	—	—	○
DCF PSHA Sx,Sy,Dz	DC = 0 & Sy ≥ 0 时，Sx << Sy→Dz DC = 0 & Sy < 0 时，Sx >> | Sy | →Dz DC = 1 时， nop	111110********** 10010011xxyyzzzz	1	—	—	—	○
PSHA #imm,Dz	imm ≥ 0 时， Dz << imm→Dz imm < 0 时， Dz >> | imm | →Dz	111110********** 00010iiiiiiizzzz	1	更新	—	—	○

(1) 说明

将 Sx 或 Dz 操作数的内容算术移位，结果保存至 Dz 操作数。由 Sy 操作数或立即数 imm 操作数指定移位量。移位量为正值时向左移位，为负值时向右移位。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*     PSHA Sx,Sy,Dz      */
       <由寄存器操作数指定时>

{
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      */
        ＜由立即操作数指定时＞

{
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) 使用示例

```
PSHA X0,Y0,A0  NOPX  NOPY       ; 执行前 :X0=H'88888888, Y0=H'00020000,
                                          A0=H'123456789A
                                  执行后 :X0=H'88888888, Y0=H'00020000,
                                          A0=H'FE22222222
PSHA X0,Y0,X0  NOPX  NOPY       ; 执行前 :X0=H'33333333, Y0=H'FFFF0000,
                                  执行后 :X0=H'19999999,Y0=H'FFFE0000,
PSHA #-5,A1  NOPX  NOPY         ; 执行前 :A1=H'AAAAAAAAAA
                                  执行后 :A1=H'FD55555555
                                  无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.18 [if cc] PSHL SHift Logically with condition DSP 逻辑移位指令

带条件的逻辑移位

格式		操作概略	操作码	执行状态	T 位	适用指令		
						SH-1	SH-2	SH-DSP
	PSHL Sx,Sy,Dz	Sy ≥ 0 时， Sx ＜＜ Sy→Dz，清除 Dz 的低位字 Sy ＜ 0 时， Sx ＞＞ Sy→Dz，清除 Dz 的低位字	111110********** 10000001xxyyzzzz	1	更新	—	—	○
DCT	PSHL Sx,Sy,Dz	DC ＝ 1&Sy ≥ 0 时，Sx ＜＜ Sy→Dz，清除 Dz 的低位字 DC ＝ 1 & Sy ＜ 0 时，Sx ＞＞ Sy→Dz，清除 Dz 的低位字 DC ＝ 0 时， nop	111110********** 10000010xxyyzzzz	1	—	—	—	○
DCF	PSHL Sx,Sy,Dz	DC ＝ 0 & Sy ≥ 0 时，Sx ＜＜ Sy→Dz，清除 Dz 的低位字 DC ＝ 0 & Sy ＜ 0 时，Sx ＞＞ Sy→Dz，清除 Dz 的低位字 DC ＝ 1 时， nop	111110********** 10000011xxyyzzzz	1	—	—	—	○
	PSHL #imm,Dz	imm ≥ 0 时， Dz ＜＜ imm→Dz，清除 Dz 的低位字 imm ＜ 0 时， Dz ＞＞ imm→Dz，清除 Dz 的低位字	111110********** 00000iiiiiiizzzz	1	更新	—	—	○

(1) 说明

将 Sx 或 Dz 操作数高位字的内容逻辑移位，结果保存至 Dz 操作数的高位字， Dz 操作数的低位字清 0。Dz 操作数为有保护位的寄存器时，保护位也清 0。由 Sy 操作数或立即数 imm 操作数指定移位量。移位量为正值时向左移位，为负值时向右移位。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*      PSHL Sx,Sy,Dz      */
        ＜由寄存器操作数指定时＞

{
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      */
         ＜由立即操作数指定时＞

{
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) 使用示例

```
PSHL X0,Y0,A0  NOPX  NOPY          ;执行前： X0=H'22222222, Y0=H'00030000,
                                            A0=H'123456789A
                                    执行后： X0=H'22222222, Y0=H'00030000,
                                            A0=H'0011100000
PSHL X1,Y1,X1  NOPX  NOPY          ;执行前： X1=H'CCCCCCCC, Y1=H'FFFE0000
                                    执行后： X1=H'33330000, Y1=H'FFFE0000
PSHL #7,A1  NOPX  NOPY             ;执行前： A1=H'55555555
                                    执行后： A1=H'AA800000
                                    无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.19 [if cc] PSTS STore System register DSP 系统控制指令

带条件从系统寄存器存储

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PSTS MACH,Dz	MACH→Dz	111110********** 1100110100000zzzz	1	—	—	—	○
PSTS MACL,Dz	MACL→Dz	111110********** 1101110100000zzzz	1	—	—	—	○
DCT PSTS MACH,Dz	DC = 1 时，MACH→Dz 为 0 时， nop	111110********** 1100111000000zzzz	1	—	—	—	○
DCT PSTS MACL,Dz	DC = 1 时， MACL→Dz 为 0 时， nop	111110********** 1101111000000zzzz	1	—	—	—	○
DCF PSTS MACH,Dz	DC = 0 时，MACH→Dz 为 1 时， nop	111110********** 1100111100000zzzz	1	—	—	—	○
DCF PSTS MACL,Dz	DC = 0 时， MACL→Dz 为 1 时， nop	111110********** 1101111100000zzzz	1	—	—	—	○

(1) 说明

MACH、 MACL 寄存器的内容保存至 Dz 操作数。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

DSR 寄存器的 DC、 N、 Z、 V、 GT 位均不更新。

(2) 注意

可指定 PSTS 与 MOVX、 MOVY 并发执行，但执行可能需要 2 个周期。

(3) 操作内容

```
/*      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) 使用示例

```
PSTS MACH,A0  NOPX  NOPY        ;执行前 :A0=H'123456789A,
                                        MACH=H'88888888
                                 执行后 :A0=H'FF88888888,
                                        MACH=H'88888888
```

6.3.20 [if cc] PSUB SUBtract with Condition DSP 算术运算指令

带条件的减法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PSUB Sx,Sy,Dz	Sx–Sy→Dz	111110********** 10100001xxyyzzzz	1	更新	—	—	○
DCT PSUB Sx,Sy,Dz	DC = 1 时， Sx–Sy→Dz 为 0 时， nop	111110********** 10100010xxyyzzzz	1	—	—	—	○
DCF PSUB Sx,Sy,Dz	DC = 0 时， Sx–Sy→Dz 为 1 时， nop	111110********** 10100011xxyyzzzz	1	—	—	—	○

(1) 说明

Sx 操作数的内容减 Sy 操作数的内容，结果保存至 Dz 操作数。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 操作内容

```
/*        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) 使用示例

```
PSUB X0,Y0,A0  NOPX  NOPY        ; 执行前：  X0=H'55555555, Y0=H'33333333,
                                             A0=H'123456789A
                                   执行后：  X0=H'55555555, Y0=H'33333333,
                                             A0=H'0022222222
                                   无条件执行时，根据 CS［2:0］的状态更新 DC 位。
```

6.3.21 PSUB SUBtraction & MULtiply Signed DSP 算术运算指令
PMULS by Signed

减法和带符号乘法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx–Sy→Du Se 的高位字 ×Sf 的高位字 →Dg	111110********** 0110eeffxxyygguu	1	更新	—	—	○

(1) 说明

Sx 操作数的内容减 Sy 操作数的内容，结果保存至 Du 操作数。对 Se、 Sf 操作数的高位字的内容进行带符号乘法运算，结果保存至 Dg 操作数。这 2 个处理可同时并发执行。

根据 ALU 运算的结果和 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也根据 ALU 运算的结果更新。

(2) 操作内容

```
/*          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) 使用示例

```
PSUB A0,M0,A0  PMULS X0,Y0,M0  NOPX  NOPY ; 执行前：  X0=H'00020000,
                                                    Y0=H'FFFE0000,
                                                    M0=H'33333333,
                                                    A0=H'0022222222
                                           执行后：  X0=H'00020000,
                                                    Y0=H'FFFE0000,
                                                    M0=H'FFFFFFF8,
                                                    A0=H'0055555555
```

6.3.22 PSUBC SUBtract with Carry DSP 算术运算指令

带借位的减法

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PSUBC Sx,Sy,Dz	Sx–Sy–DC→Dz	111110********** 10100000xxyyzzzz	1	借位	—	—	○

(1) 说明

Sx 操作数的内容减 Sy 操作数的内容和 DC 位，结果保存至 Dz 操作数。
DSR 寄存器的 DC 位作为借位标志更新， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。

(2) 注意

PADDC 指令执行后的 DC 位与 CS 位无关，作为借位标志更新。

(3) 操作内容

```
/*   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) 使用示例

```
CS[2:0]=***: Always Carry or Borrow Mode
PSUBC X0,Y0,M0  NOPX  NOPY ;        执行前： X0=H'33333333, Y0=H'55555555
                                             M0=H'12345678, DC=0
                                    执行后： X0=H'33333333, Y0=H'55555555
                                             M0=H'DDDDDDDE, DC=1
PSUBC X0,Y0,M0  NOPX  NOPY ;        执行前： X0=H'33333333, Y0=H'55555555
                                             M0=H'12345678, DC=1
                                    执行后： X0=H'33333333, Y0=H'55555555
                                             M0=H'DDDDDDDD, DC=1
```

6.3.23 [if cc] PXOR logical eXclusive OR DSP 逻辑运算指令

带条件的逻辑“异或”运算

格式	操作概略	操作码	执行状态	T 位	适用指令		
					SH-1	SH-2	SH-DSP
PXOR Sx,Sy,Dz	Sx^Sy→Dz，清除 Dz 的低位字	111110********** 10100101xxyyzzzz	1	更新	—	—	○
DCT PXOR Sx,Sy,Dz	DC = 1 时，Sx^Sy→Dz，清除 Dz 的低位字 为 0 时， nop	111110********** 10100110xxyyzzzz	1	—	—	—	○
DCF PXOR Sx,Sy,Dz	DC = 0 时，Sx^Sy→Dz，清除 Dz 的低位字 为 1 时， nop	111110********** 10100111xxyyzzzz	1	—	—	—	○

(1) 说明

执行 Sx 操作数高位字的内容和 Sy 操作数高位字的内容的逻辑 “异或”运算，结果保存至 Dz 操作数的高位字， Dz 操作数的低位字清 0。 Dz 操作数为有保护位的寄存器时，保护位也清 0。指定 DCT、 DCF 条件时，如果条件为真，执行指令；条件为假，则不执行。

未指定条件时，根据 CS 位的指定更新 DSR 寄存器的 DC 位， DSR 寄存器的 N、 Z、 V、 GT 位也随之更新。指定条件时，即使条件为真、并已执行指令，也不更新 DC、 N、 Z、 V、 GT 位。

(2) 注意

更新 DC 位时忽略目标寄存器低位字的内容和保护位的内容。

(3) 操作内容

```
/* 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) 使用示例

```
PXOR X0,Y0,A0  NOPX  NOPY ;   执行前 :X0=H'33333333,Y0=H'55555555
                                    A0=H'123456789A
                              执行后 :X0=H'33333333,Y0=H'55555555
                                    A0=H'0066660000
```

无条件执行时，根据 CS［2:0］的状态更新 DC 位。

第 7 章 流水线运行

本章说明各指令的流水线运行，该运行提供计算 CPU 指令执行状态数 （系统时钟周期数）所需的信息。

7.1 流水线的基本结构

7.1.1 5 段流水线

流水线由以下 5 个阶段构成：

- IF： 取指令……从保存程序的存储器获取指令。
- ID： 指令解码……解码获取的指令。
- EX： 指令执行……根据解码结果，执行数据运算或地址计算。
- MA： 存储器存取……存取存储器的数据。
  在执行存储器存取指令时产生，但有部分例外。
- WB/DSP （W/D）：
  回写 （CPU 内核）或 DSP （DSP 单元）
  回写……将存储器存取的结果 （数据）返回寄存器。
  在执行存储器加载指令时产生，但有部分例外。
  DSP……使用 DSP 单元的 ALU、 MAC 来运算。
  将存储器存取的结果 （数据）返回寄存器。写入存储器或空操作 （NOP）时，不产生 WB/DSP。

执行指令的同时，指令的各个阶段依次执行，构成流水线。在某一瞬间， 5 条指令将同时执行。流水线的基本流程如图 7.1 所示。 1 个阶段的执行周期称为槽，用 “←→” 表示。

所有指令都存在 IF、 ID、 EX 等 3 个阶段，但有些指令可能没有 MA、 WB/DSP。流水线的流程因指令种类不同而变化。 IF 与 MA 之间也会产生竞争，此时，流水线的流程会发生变化。

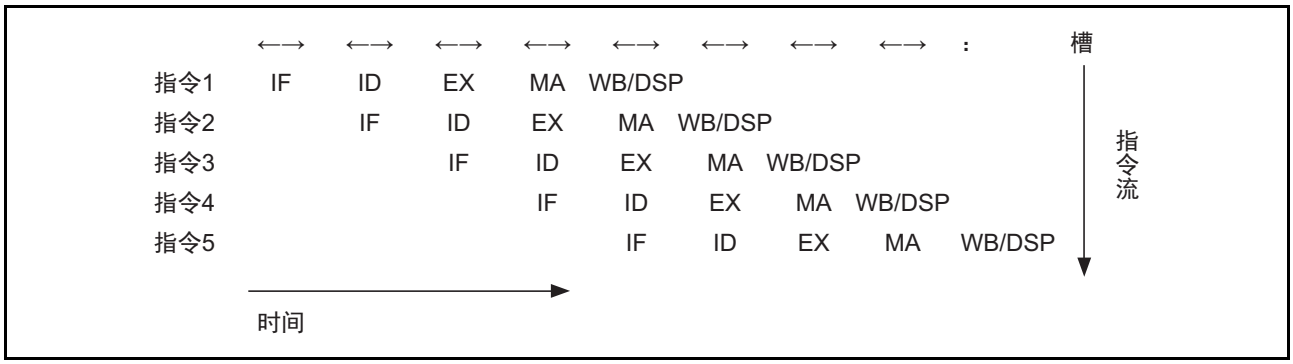


图 7.1 流水线的基本结构

7.1.2 槽与流水线的流程

1 个阶段的执行周期称为槽，该槽具有以下规则：

1. 指令的各个阶段（IF、ID、EX、MA、WB/DSP）必须在1个槽内执行。即1个槽内不可执行2个或2个以上阶段。由于在MA之后立即执行WB，因此，也会有在同一槽内执行某条指令的MA与WB。
2. 1个槽内最多可设定其他指令的1个不同阶段，即在1个槽内不可能执行其他指令的相同阶段。

不可能存在的流水线流程如图7.2、图7.3所示。

图 7.2 不可能存在的流水线流程 （1）

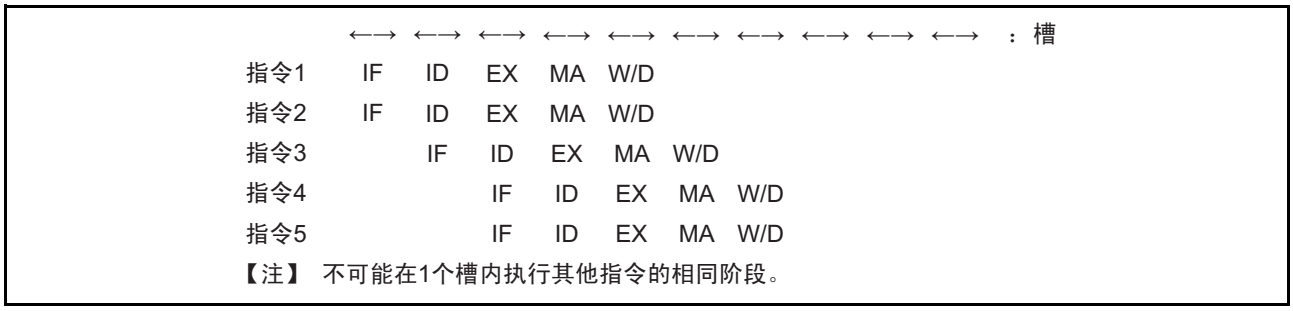


图 7.3 不可能存在的流水线流程 （2）

7.1.3 执行 1 个槽所需的状态数

执行 1 个槽所需的状态数 （系统时钟周期数） S 根据以下条件计算：

S= （1 个槽内所包含的各指令阶段的最长状态数）
即：最长阶段导致其他短阶段的指令停止
各阶段的执行状态数如下……

- IF： 用于取指令的存储器存取时钟数
- ID： 总是为1个状态
- EX： 总是为1个状态
- MA： 用于数据存取的存储器存取时钟数
- WB/DSP： 总是为1个状态

例如，执行指令 1、指令 2 的 IF （用于取指令的存储器存取）需要 2 个周期，执行指令 1 的 MA （用于数据存取的存储器存取）需要 3 个周期，其他需要 1 个周期，这时流水线流程如**图 7.4** 所示。“－”表示停止。

槽	←——→		←——→		←→	←————→			←→	←→
状态数	（2）		（2）		（1）	（3）			（1）	（1）
指令1	IF	IF	ID	－	EX	MA	MA	MA	W/D	
指令2			IF	IF	ID	EX	－	－	MA	W/D

图 7.4 需要多个周期的槽

7.1.4 指令执行状态数

各指令的执行状态数通过数 EX 阶段的执行间隔算出。从开始执行指令 1 的 EX 阶段到开始执行下一条指令 2 的 EX 阶段前的状态数为指令 1 的执行时间。指令执行状态数的计算方法例如**图 7.5** 所示。

例如，为**图 7.5** 所示的流水线流程时，指令 1 与指令 2 的 EX 阶段间隔为 5 个状态，因此指令 1 的执行时间为 5 个状态。指令 2 与指令 3 的 EX 阶段间隔为 1 个状态，因此指令 2 的执行时间为 1 个状态。如果程序在指令 3 结束，则假设指令 3 的下一条指令为 MOV Rm,Rn，即指令 4，根据指令 3 与指令 4 的 EX 阶段间隔计算指令 3 的执行时间。此例中，指令 3 的执行时间为 1 个状态。指令 1 ～指令 3 的执行时间总计为 5+1+1=7 个状态。

此例中，指令 1 的 MA 与指令 4 的 IF 竞争。 MA 与 IF 竞争时的运行，详细内容参照 **“7.2.1 取指令（IF）与存储器存取 （MA）的竞争”**。

槽	←——→		←——→		←——→	←————→			←→	←→
状态数	（2）		（2）		（2）	（4）			（1）	（1）
指令1	IF	IF	ID	－	EX	－	MA	MA	MA	W/D
指令2			IF	IF	ID	－	－	－	－	EX
指令3					IF	IF	－	－	－	ID
（指令4：	MOV	Rm,Rn							IF	ID

EX	MA	
EX		
EX	MA	
EX	）	

图 7.5 指令执行状态数的计算方法例

7.2 产生竞争

以下 4 种情况下产生竞争。如果产生竞争，则拆分该槽， 1 个槽至少需要 2 个状态。

1. 取指令 （IF）与存储器存取 （MA）的竞争
2. 使用先行指令目标寄存器时的竞争
3. 乘法器存取的竞争
4. DSP运算或存储器加载 （WB/DSP）与存储器存储 （MA）的竞争

7.2.1 取指令 （IF）与存储器存取 （MA）的竞争

(1) IF 与 MA 竞争时的基本操作 （通用）

IF 阶段与 MA 阶段都要存取存储器，因此不可同时运行。如果 IF 阶段与 MA 阶段欲在同一槽内存取存储器，则拆分该槽。有 WB 时，在 MA 结束后立即执行 WB。 IF 与 MA 竞争时的运行如图 7.6 所示。

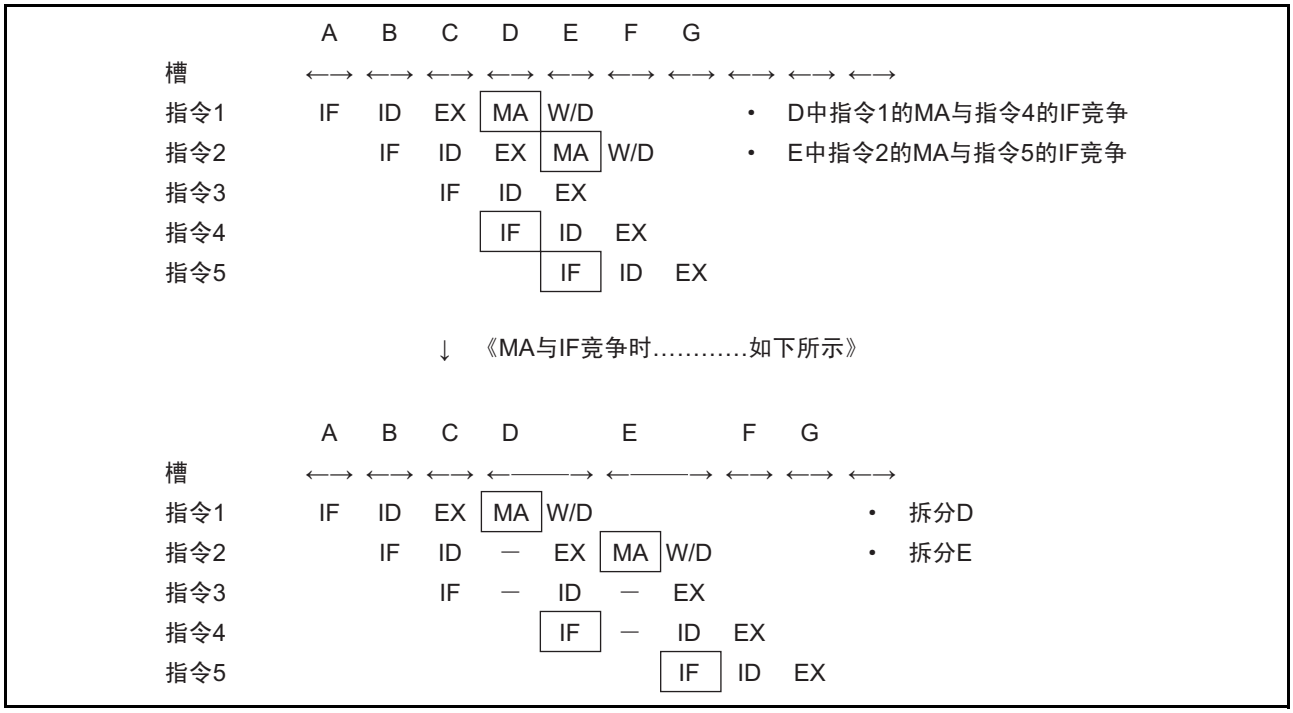


图 7.6 IF 与 MA 竞争时的运行

拆分 MA 与 IF 产生竞争的槽，在槽的前半部分优先执行 MA，在槽的后半部分同时执行其他的 EX、 ID、IF。 MA 之后有 WB 时，在 MA 结束后立即执行 WB。例如，图 7.6 的情况，在槽 D 优先执行指令 1 的 MA，在槽的后半部分同时执行指令 2 的 EX、指令 3 的 ID 及指令 4 的 IF 等 3 个阶段。在槽 E 优先执行指令 2 的 MA，在槽的后半部分同时执行指令 3 的 EX、指令 4 的 ID 及指令 5 的 IF。

MA 与 IF 产生竞争的槽所需状态数为 MA 与 IF 的存储器存取周期数之和。

(2) 配置于内部 ROM/RAM 或内部高速缓存的指令的位置与 IF 的关系 （SH-1/2）

在 SuperH 单片机的内部 ROM/RAM 或内部高速缓存配置指令时， SuperH 单片机以 32 位存取内部 ROM/RAM 或内部高速缓存。 SuperH 单片机的指令长度均固定为 16 位，因此基本上 IF 阶段的 1 次存取可取 2 条指令。

此时，在 IF 执行 1 次取指令可取 2 条指令，因此在下一条指令的 IF，不产生从存储器取指令的总线周期。在下下一条指令的 IF 也可取 2 条指令，且在其下一条指令的 IF 不产生总线周期。

即，在配置于内部 ROM/RAM 或内部高速缓存的指令中，从长字边界配置的指令 （指令地址低 2 位为 00 的位置：A1 ＝ 0、 A0 ＝ 0）的 IF 可取 2 条指令。因此，其下一条指令的 IF 不产生总线周期。不产生该总线周期的 IF 用小写字母 “if” 表示。 if 必须为 1 个状态。

另一方面，通过转移，从字边界配置的指令 （指令地址低 2 位为 10 的位置：A1=1、 A0=0）取指令时，该 IF 的总线周期只能取 1 条指令。因此，虽然其下一条指令的 IF 产生总线周期，但从该 IF 仍可取 2 条指令。以上运行如图 7.7 所示。

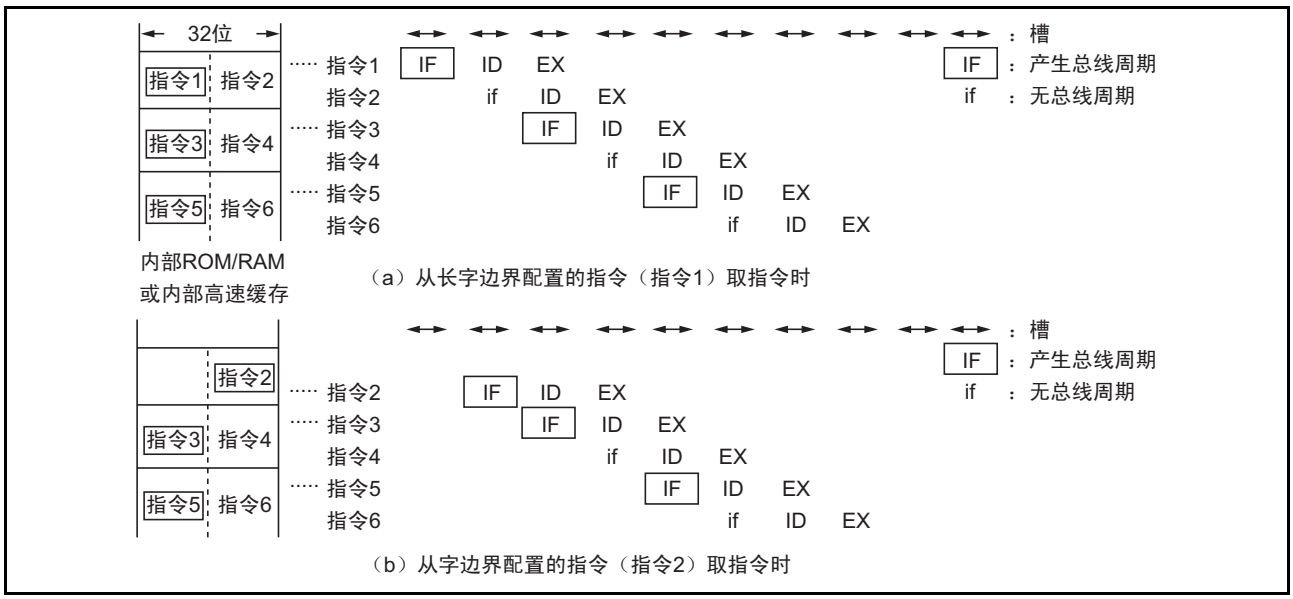


图 7.7 配置于内部 ROM/RAM 或内部高速缓存的指令的位置与 IF 的关系

(3) 配置于内部 ROM/RAM 或内部高速缓存的指令的位置和 IF 与 MA 竞争的关系 （SH-1/2）

在内部 ROM/RAM 或内部高速缓存配置指令时，如上述 （2）所示，存在不产生总线周期的取指令阶段（用小写字母 “if”表示），该 if 与 MA 竞争时，与 IF 和 MA 的竞争不同，不拆分槽。即， if 与 MA 可同时执行。执行该槽所需的状态数为 MA 的存储器存取周期数。详细内容如图 7.8 所示。

编程时，尽量避免 MA 与 IF 的竞争，如果写程序时使 MA 与 if 产生竞争，则可提高指令执行速度。即，如果将具有 IF、 ID、 EX、 MA、（WB）等 4 （5）段流水线的指令从内部 ROM/RAM 或内部高速缓存的长字边界 （指令地址低 2 位为 00 的地址：A1=0、 A0=0）配置，则该指令的 MA 阶段与其后的 if 在同一槽内，因此不会停止。

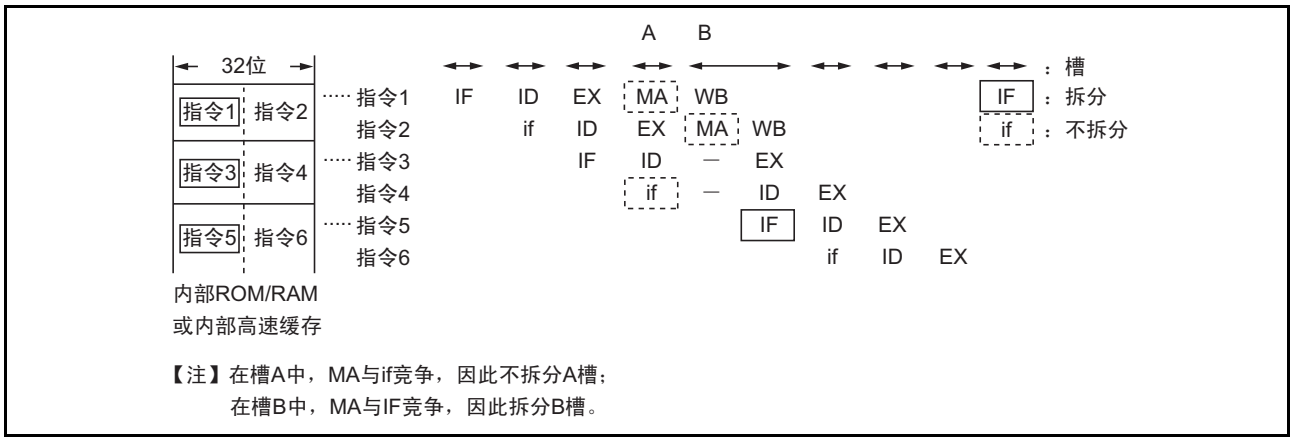


图 7.8 配置于内部 ROM/RAM 或内部高速缓存的指令的位置和 IF 与 MA 竞争的关系

(4) 配置指令的位置与 IF 的关系 （SH-DSP）

取指令时， SH-DSP 以 32 位存取。因此，基本上通过 IF 阶段的 1 次存取，如果指令长度为 16 位，则取 2 条指令；如果指令长度为 32 位，则取 1 条指令。

首先，说明转移至从长字边界配置的指令 （指令地址低 2 位为 00 的位置）。此时，如图 7.9 （a）所示，在最初 3 个 IF 阶段 （IF1、 IF2、 IF3）产生总线周期，从存储器取 6 字长度的指令。在此后的 IF 阶段是产生从存储器取指令的总线周期，还是不从存储器取指令且不产生总线周期 （该 IF 用小写字母 “if”表示。 If 必须为 1 个状态。），取决于以下 3 条规则：

a. 2 条指令前的指令从长字边界配置时， IF阶段产生总线周期。
b. 2 条指令前的指令长度为32位时， IF阶段产生总线周期。
c. 2 条指令前的指令为上述a、 b之外时， IF阶段不产生总线周期。

例如，图 7.9 （a）的 if4，此时 2 条指令前的指令 （指令 2）未在长字边界配置，且长度为 16 位，因此不产生总线周期。 IF5 时，2 条指令前的指令 （指令 3）在长字边界配置，因此产生总线周期， IF6 时， 2 条指令前的指令 （指令 4）长度为 32 位，因此产生总线周期。

其次，说明转移至字边界配置的指令 （指令地址低 2 位为 10）。此时，如图 7.9 （b）所示，在转移目标指令 （指令 2）的 ID 阶段前，有 2 个 IF 阶段 （IF1、 IF2）。在这 2 个 IF 阶段与下一个 IF 阶段 （IF3）产生总线周期，从存储器取 5 字长度的指令。是否在此后的 IF 阶段产生从存储器取指令的总线周期，取决于前文所述的 3 条规则。

在 ID 阶段前的 IF 阶段获取的指令，有时与 ID 阶段的指令不相同。例如，图 7.9 （a）中，在指令 2 的 ID 阶段前的 IF 阶段 （IF2）取的指令为指令 3 与指令 4。

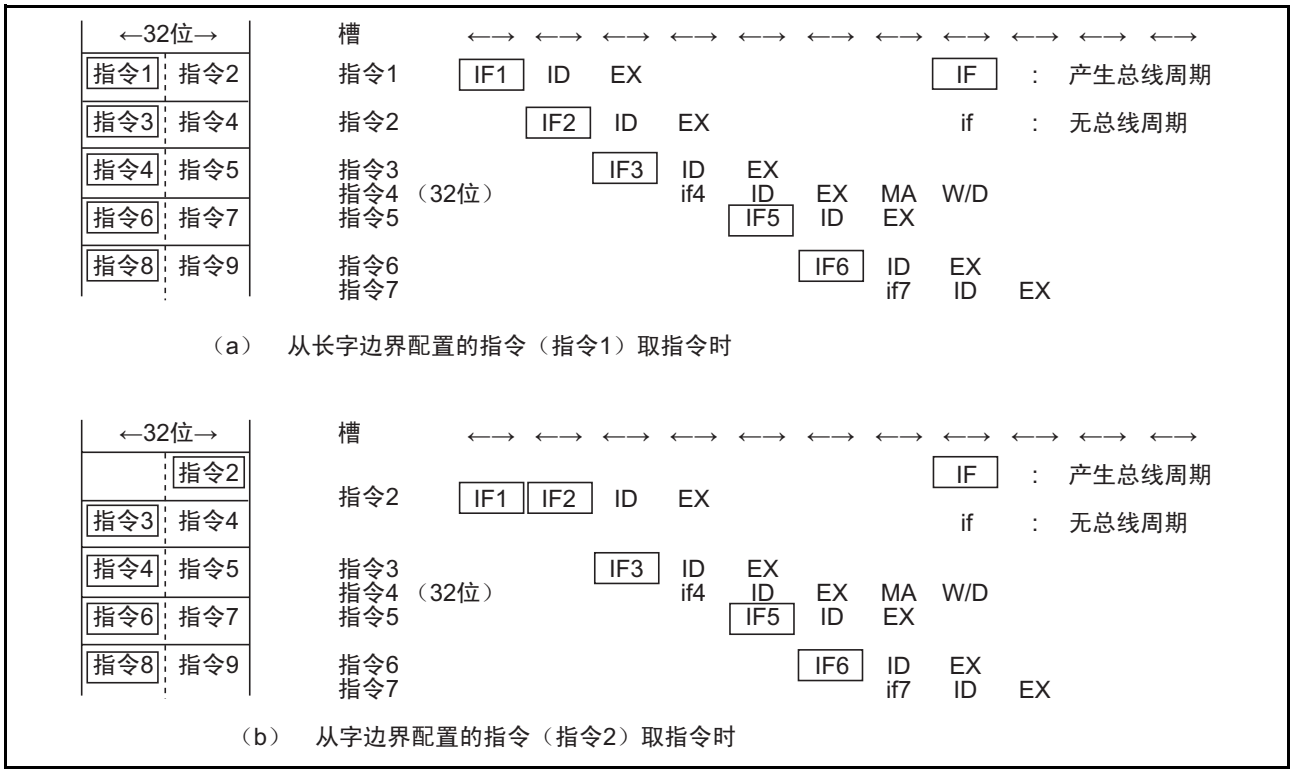


图 7.9 配置于内部 ROM/RAM 或内部高速缓存的指令的位置与 IF 的关系

(5) 配置指令的位置和 IF 与 MA 竞争的关系 （SH-DSP）

取指令时，如上述 （2）所示，存在不产生总线周期的取指令阶段 （用小写字母 “if” 表示）。该 if 与 MA 竞争时，与 IF 和 MA 的竞争不同，不拆分槽。即， if 与 MA 可同时执行。执行该槽所需的状态数为 MA 的存储器存取周期数。详细内容如图 7.10 所示。

编程时，尽量避免 MA 与 IF 的竞争，如果写程序时使 MA 与 if 产生竞争，则可提高指令执行速度。

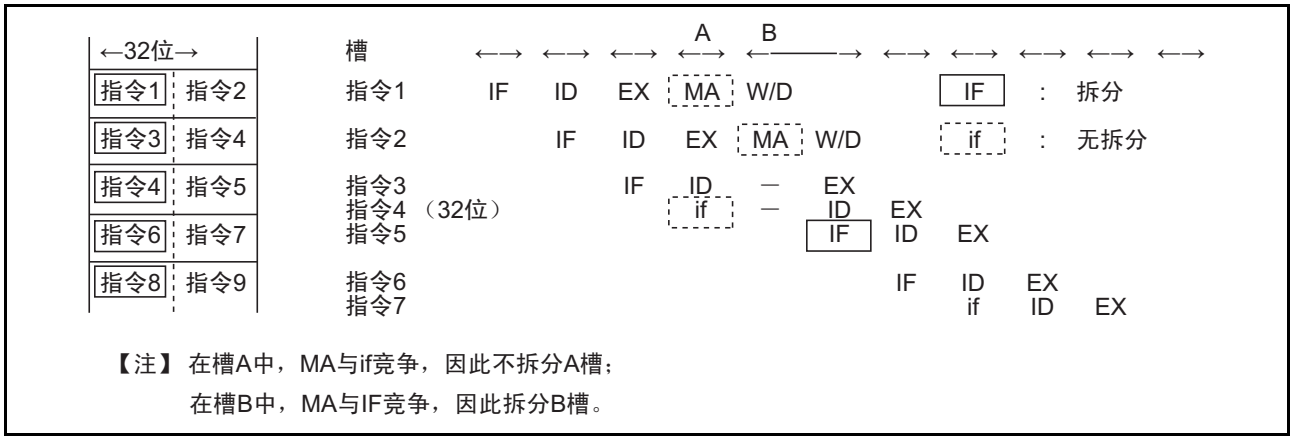


图 7.10 配置于内部 ROM/RAM 或内部高速缓存的指令的位置和 IF 与 MA 竞争的关系

7.2.2 使用先行指令目标寄存器时的竞争

(1) 加载指令与下一条指令的关系

在流水线最后的 WB/DSP 阶段，从存储器向 CPU 寄存器的加载指令将数据返回目标寄存器。如果注意该加载指令 （作为加载指令 1）与紧接其之后的 CPU 运算或存储指令 （作为指令 2），则会发现指令 2 的 EX 阶段开始于加载指令 1 的 WB/DSP 阶段结束前。

此时，指令 2 要使用加载指令 1 的目标寄存器时，该寄存器的内容尚未准备好，因此需要拆分指令 1 的 MA 与指令 2 的 EX 所在的槽。加载指令 1 的目标寄存器即使不是指令 2 的源寄存器而是目标寄存器，也要拆分槽。

加载指令 1 的目标寄存器为状态寄存器 （SR），指令 2 获取并使用其中的标志时 （例如， ADDC 等）也要拆分槽。

但，以下情况不拆分：

1. 指令2为加载指令，且其目标寄存器与加载指令1的目标寄存器相同时。
2. 指令2为MAC @Rm+,@Rn+，且Rm与加载指令1的目标寄存器相同时。

产生拆分的槽的状态数为 MA 的周期数＋ IF （或 if）的周期数。详细内容如图 7.11 及图 7.12 所示。

因此，编程时，如果在加载指令后紧接着配置使用其结果的指令，会降低执行速度；如果在加载指令的 2 条指令后配置使用加载指令结果的指令，则不会降低速度。

槽									
加载指令1 （MOV @Ra,Rb）	IF	ID	EX	MA	W/D				
指令2 （ADD Rb,Rc）		IF	ID	—	EX				
指令3			IF	—	ID	EX	MA	W/D	
指令4					IF	ID	EX	MA	W/D

图 7.11 存储器加载指令对流水线的影响

通过先行指令给寄存器加载数据，后续的存储器存取指令将该寄存器作为地址指针使用时，会延长存储器存取直至先行指令的 MA 阶段的数据加载结束。

槽									
加载指令1 （MOV @Ra,Rb）	IF	ID	EX	MA	W/D				
指令2 （MOV @Rb,Rc）		IF	ID	—	EX	MA	W/D		
指令3			IF	—	ID	EX	MA	W/D	
指令4					IF	ID	EX	MA	W/D

图 7.12 存储器加载指令对流水线的影响

由于在 DSP 单元，所有运算指令均在 WB/DSP 阶段执行，因此不产生传送与运算的竞争。详细内容如图 7.13 所示。但，先行 MOV 指令的目标寄存器用作后续指令的地址指针时，与图 7.12 相同，会产生竞争。

槽									
指令1（MOVX @Ra,X0）	IF	ID	EX	MA	W/D				
指令2（PADD X0,Y0,A0）		IF	ID	EX	MA	W/D			
指令3			IF	ID	EX	MA	W/D		
指令4				IF	ID	EX	MA	W/D	

图 7.13 DSP 单元的存储器加载指令对流水线的影响

(2) 数据运算指令与存储指令的关系

在 DSP 单元执行 DSP 运算，通过下一条指令将其结果存储在存储器时，会产生与存储器加载指令相同的竞争。此时，会延长后续指令的 MA 阶段的数据存储直至先行指令的 WB/DSP 阶段的数据运算结束。

CPU 内核的运算均在 EX 阶段执行，因此不产生停止周期。

DSP 单元的数据运算指令与存储指令的关系如图 7.14 所示，与 CPU 内核的关系如图 7.15 所示。

槽	←→	←→	←→	←→	←——→	←→	←→	←→
指令1（PADD X0,Y0,A0）	IF	ID	EX	MA	W/D			
指令2（MOVX A0,@Ra）		IF	ID	EX	—	MA	W/D	
指令3			IF	ID	—	EX	MA	W/D
指令4				IF	—	ID	EX	MA	W/D

图 7.14 DSP 单元的运算指令与存储指令的关系

槽	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→
指令1（ADD Ra,Rb）	IF	ID	EX	MA	W/D					
指令2（MOV Rb,@Rc）		IF	ID	EX	MA	W/D				
指令3			IF	ID	EX	MA	W/D			
指令4				IF	ID	EX	MA	W/D		

图 7.15 CPU 内核的运算指令与存储指令的关系

(3) 加载指令与存储指令的关系

从存储器加载至目标寄存器，且该寄存器指定为后续存储指令的源操作数时，在 WB/DSP 阶段执行先行指令的加载，在 MA 阶段执行后续指令的存储。此阶段的执行周期完全相同，但不产生竞争。 CPU 内核与 DSP 单元均使用相同的数据传送方法，在输入到内部总线的数据保存至目标寄存器的同时，相同的数据再次输出至内部总线。

槽	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→
指令1（MOV.L @Ra,Rn）	IF	ID	EX	MA	W/D					
指令2（MOV.L Rn,@Rb）		IF	ID	EX	MA	W/D				
指令3			IF	ID	EX	MA	W/D			
指令4				IF	ID	EX	MA	W/D		

图 7.16 CPU 内核的加载指令与存储指令的关系

槽	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→
指令1（MOVS.L @R4,Ds）	IF	ID	EX	MA	W/D					
指令2（MOVS.L Ds,@R5）		IF	ID	EX	MA	W/D				
指令3			IF	ID	EX	MA	W/D			
指令4				IF	ID	EX	MA	W/D		

图 7.17 DSP 单元的加载指令与存储指令的关系

(4) MAC 指令与 STS 指令的关系

MAC.W 指令中分别有 2 个 MA、 mm （乘法器存取）阶段。 MAC.W 指令后，紧接将 MACL、 MACH 寄存器存储至 Rn 寄存器的 STS 指令时，在 MAC.W 指令的 mm 阶段结束后，执行 STS 指令的 MA 阶段。
MAC.W 指令后，紧接将 MACL、 MACH 寄存器存储至存储器的 STS.L 指令时，同样也在 MAC.W 指令的 mm 阶段结束后，执行 STS.L 指令的 MA 阶段。

MAC.L 指令也相同，紧接 STS 指令、 STS.L 指令时，在 mm 阶段结束 （MAC.L 指令的 mm 阶段有 4 个）后，执行 STS、 STS.L 指令的 MA 阶段。

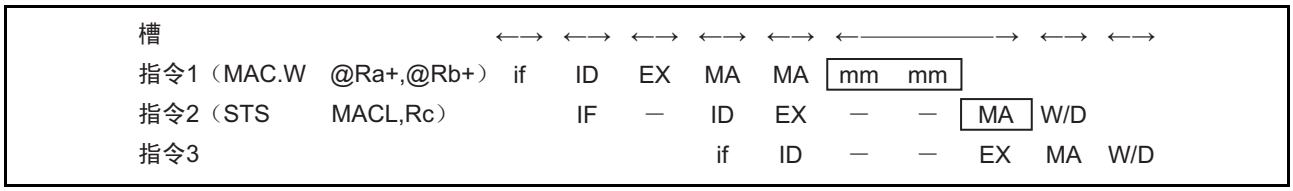


图 7.18 MAC.W 指令与 STS 指令的关系

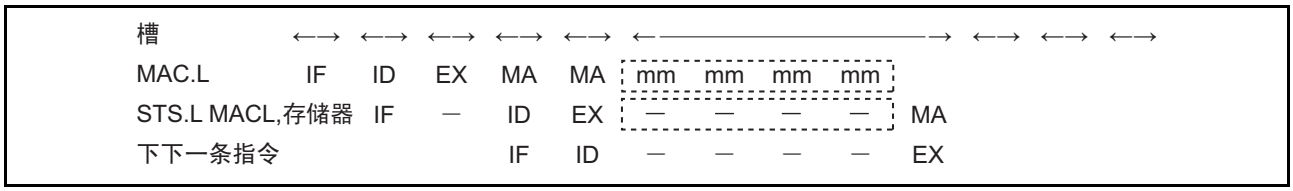


图 7.19 MAC.L 指令与 STS.L 指令的关系

7.2.3 由乘法器存取产生的竞争

乘法类指令 （乘加指令、乘法指令）及存取乘加寄存器 （MACH、 MACL）的指令，在乘法器存取时产生竞争。

乘法类指令中，乘法器在 MA 结束后与槽无关，为双精度指令 （64 位）时在 4 个状态之间运行，为单精度指令 （32 位）时在 2 个状态之间运行。乘法类指令 （乘加指令、乘法指令）的 MA （有 2 个时，为第 2 个 MA）与其之前乘法类指令的乘法器存取 （mm）产生竞争时，延长该 MA 的总线周期直至 mm 结束，延长的 MA 为 1 个槽。

双精度指令的下一条指令的 ID 停止 1 个槽的期间。

乘法类指令及乘加寄存器存取指令具有 MA，因此也会产生与 IF 的竞争。

乘法器存取竞争例如图 7.20 所示，此处未考虑 MA 与 IF 的竞争。

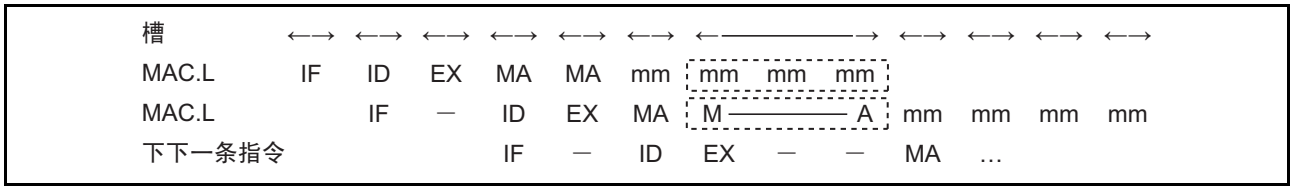


图 7.20 乘法器存取竞争例 （MAC.L 指令与 MAC.L 指令的竞争）

7.2.4 DSP 寄存器之间传送与存储器加载 / 存储运行的竞争

DSP 单元具有在寄存器 （MACH、 MACL）（用于执行以往 SuperH 单片机 （SH-1、 SH-2）的乘法 / 乘法累加运算指令）与寄存器 （A0、 A1、 M0、 M1、 X0、 X1、 Y0、 Y1）（用于 DSP 运算指令）之间执行数据传送的指令 （PSTS、 PLDS）。如果该指令与通过 MOVX.W 的存储器加载并发执行，则产生竞争。如果在该指令后立即通过 MOVX.W、 MOVS.W 或 MOVS.L 执行存储器存储，也会产生竞争。

从定义上讲，该指令可分配在与其他 DSP 运算指令相同的操作码区域，因此可与双数据传送指令 (MOVX.W、 MOVY.W）并发执行。但运行的硬件与 MOVX.W 复用，因此，如果 PSTS 或 PLDS 与 MOVX.W （加载）并发执行，则会在 WB/DSP 阶段产生竞争。此时，先执行 PSTS 或 PLDS， MOVX.W （加载）停止 1 个槽的期间。

MOVS.W、 MOVS.L 与 MOVX.W 均使用相同硬件，但无法与 DSP 运算指令并发执行，因此不可能产生上述竞争。但在该指令后立即通过 MOVX.W、 MOVS.W 或 MOVS.L 执行存储运行时，PSTS 或 PLDS 的 WB/DSP 阶段与存储操作的 MA 阶段产生竞争。

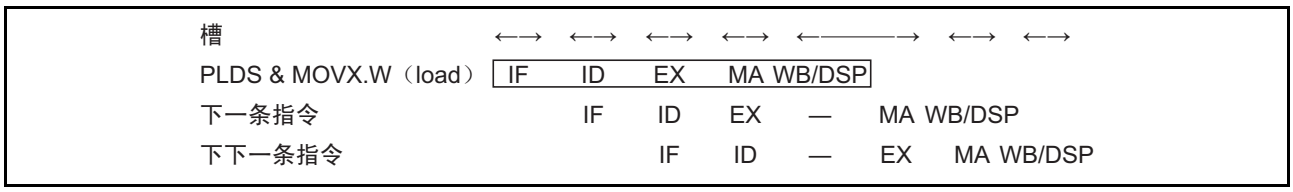


图 7.21 在 DSP 寄存器之间传送与存储器加载并发运行的竞争

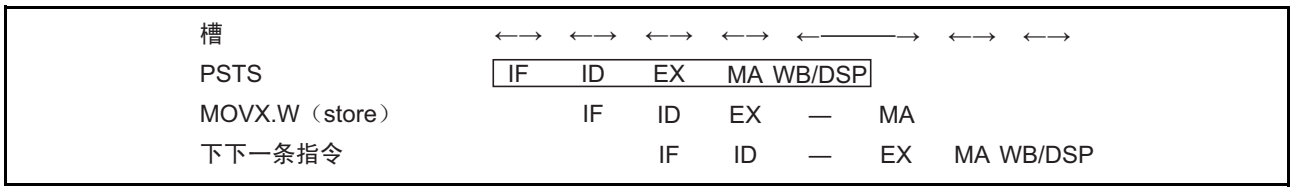


图 7.22 DSP 寄存器之间传送后紧接的存储器存储运行的竞争

7.3 编程准则

7.3.1 竞争种类与指令的对应关系

竞争种类与指令的对应关系汇总如下：

1. 此指令不产生竞争
2. 此指令有存储器存取 （MA），与取指令 （IF）竞争
3. 此指令使用X总线或Y总线，将之前DSP运算结果存储到存储器
4. 此指令有存储器存取 （MA），与取指令 （IF）竞争；有回写 （WB/DSP）；与存储器加载竞争
5. 此指令有存储器存取 （MA），与取指令 （IF）竞争；存取乘法器 （mm），与乘法器竞争
6. 此指令有存储器存取 （MA），与取指令 （IF）竞争；将之前DSP运算结果存储到存储器
7. 此指令有存储器存取 （MA），与取指令 （IF）竞争；存取乘法器 （mm），与乘法器竞争；传送之前DSP运算结果；有回写 （WB/DSP），与存储器加载竞争
8. 此指令与MOV.X、 MOVS.W或MOVS.L指令竞争

竞争种类与指令的对应关系如**表 7.1** 所示。

表 7.1 竞争种类与指令的对应关系

竞争	执行状态	阶段数	指令
无	1	3	寄存器之间的传送指令 寄存器之间的运算 （乘法类指令除外） 寄存器之间的逻辑运算指令 移位指令 系统控制 ALU 指令
	2	3	无条件转移
	3/1	3	条件转移
	2/1	3	带延迟的条件转移指令
	3	3	SLEEP 指令
	4	5	RTE 指令
	8	9	TRAP 指令
	1	5	DSP 运算指令、 MOVX.W （加载）、 MOVY.W （加载）指令
• MA 与 IF 竞争	1	4	存储器存储指令 STS.L 指令 （PR）
	2	4	STC.L 指令
	3	6	存储器逻辑运算
	4	6	TAS 指令
	1	5	MOVS.W （加载）、 MOVS.L （加载）指令
• 与 DSP 运算竞争	1	4	MOVX.W （存储）、 MOVY.W （存储）指令
• MA 与 IF 竞争 • 与存储器加载竞争	1	5	存储器加载指令 LDS.L 指令 （PR）
	3	5	LDC.L 指令
• MA 与 IF 竞争 • 与乘法器竞争	1	4	寄存器 →MAC 传送指令 （MACH/MACL） 存储器 →MAC 传送指令 （MACH/MACL） MAC→ 存储器传送指令 （MACH/MACL）
	1(～ 3)*	6	乘法指令 （PMULS 除外）
	2(～ 3)*	7	乘加指令
	2(～ 4)*	9	双精度乘加指令
	2(～ 4)*	9	双精度乘法指令
• MA 与 IF 竞争 • 与 DSP 运算竞争	1	4	MOVS.W （存储）、 MOVS.L （存储）指令
• MA 与 IF 竞争 • 与乘法器竞争 • 与 DSP 运算竞争 • 与存储器加载竞争	1	5	STS 指令 （PR 除外）
• 与 MOVX.W、 MOVY.W、 MOVS.W 或 MOVS.L 指令竞争	1	5	PLDS、 PSTS 指令

【注】 * 表示普通执行状态。( ) 内的值表示因与前后指令竞争产生的执行状态。

7.3.2 指令执行速度的提高

如果编程时尽量避免竞争，则可提高指令执行速度，此时如下编程：

1. 为了不产生存储器存取 （MA）与取指令 （IF）的竞争，将有MA的指令配置在内部存储器的长字边界 （指令地址低2位为00的位置：A1＝0、 A0＝0）。
2. 在紧接着存储器加载指令后的CPU内核的运算指令中，配置不使用与加载指令的目标寄存器相同寄存器的指令。由此，不产生因回写 （WB/DSP）而导致的存储器加载竞争。
3. 不要连续配置使用乘法器的指令 （PMULS指令除外）。存取从乘法器获取结果的MACH、 MACL寄存器时，也不要连续配置使用乘法器的指令。由此，不产生因乘法器存取 （mm）而导致的乘法器竞争。
4. 在DSP单元执行数据运算后，不要立即将保存运算结果的寄存器的内容数据传送至存储器或CPU内核的寄存器。通过插入其他指令，可避免产生竞争。
5. 在DSP单元执行PLDS/PSTS指令后，不要立即通过MOVX.W、 MOVY.W、 MOVS.W或MOVS.L进行存储器存储，且不要并发执行PLDS/PSTS指令与通过MOVX.W执行的存储器存储指令。

7.3.3 状态数

将基本指令设计为以 1 条指令 1 个状态执行。 1 个状态的指令，包含不产生竞争的指令与产生竞争的指令。在 1 个状态执行寄存器之间的运算及传送，不产生竞争。

有些指令即使不产生竞争也至少有 2 个状态。如通过转移指令等改变转移目标地址的指令、存储器逻辑运算指令或如果干条系统控制指令一样至少执行 2 次存储器存取的指令、如乘法指令、乘加指令一样具有存储器存取及乘法器存取的指令 （PMULS 除外），都至少有 2 个状态。

至少有 2 个状态的指令，也包含不产生竞争的指令与产生竞争的指令。

为了编写高效率的程序，在避免竞争提高指令执行速度的同时，还需要考虑使用状态数较少的指令。

7.4 各指令的流水线运行

以下说明各指令的流水线运行。综合上述规则，可计算程序流水线的流程及指令执行状态数。

在以下流水线的图中，“指令 A”为要说明的指令。不区分取指令阶段的 IF 与 if，均写作 IF。如果该 IF 与 MA 竞争，则拆分槽，特殊情况外除外，拆分状况都不在下图表示。判断要拆分槽时，必须以 “7.2.1 取指令 （IF）与存储器存取 （MA）的竞争”所述规则为基础，考虑流水线运行。

指令阶段数与执行状态数用以下格式表示。

指令阶段数与执行状态数的表格式

分类	区分	指令	执行状态	阶段数	竞争
根据功能分类。	根据不同的操作，区分指令。	用助记符表示对应的指令。	没有竞争时的执行状态数。	指令的阶段数。	表示产生的竞争。

表 7.2 指令的阶段数与执行状态数

分类	区分	指令	执行状态	阶段数	竞争

数据传送指令	寄存器－寄存器之间的传送指令	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	—

	存储器加载指令	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	• 如果在该指令后紧接着设置使用该指令目标寄存器的 CPU 运算指令，则产生竞争。 • MA 与 IF 竞争。
	存储器加载指令	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 与 IF 竞争。

分类	区分	指令	执行状态	阶段数	竞争

算术运算指令	寄存器之间的算术运算指令
	（乘法类指令除外）
		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	—

	乘法指令	MULS.W Rm,Rn MULU.W Rm,Rn	1(～ 3)*1	6/7*3	• 该指令后出现使用乘法器的指令时，产生乘法器竞争。 • MA 与 IF 竞争。
	乘加指令	MAC.W @Rm+,@Rn+	2(～ 3)*1	7/8*3	• 该指令后出现使用乘法器的指令时，产生乘法器竞争。 • MA 与 IF 竞争。
	双精度乘加指令	MAC.L @Rm+,@Rn+	2(～ 4)*1	9	• 该指令后出现使用乘法器的指令时，产生乘法器竞争。 • MA 与 IF 竞争。
	双精度乘法指令	DMULS.L Rm,Rn DMULU.L Rm,Rn MUL.L Rm,Rn	2(～ 4)*1	9	• 该指令后出现使用乘法器的指令时，产生乘法器竞争。 • MA 与 IF 竞争。

分类	区分	指令	执行状态	阶段数	竞争
逻辑运算指令	寄存器－寄存器之间的逻辑运算指令	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	—
	存储器逻辑运算指令	AND.B #imm,@(R0,GBR) OR.B #imm,@(R0,GBR) TST.B #imm,@(R0,GBR) XOR.B #imm,@(R0,GBR)	3	6	• MA 与 IF 竞争。
	TAS 指令	TAS.B @Rn	4	6	• MA 与 IF 竞争。
移位指令	移位指令	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	—
转移指令	条件转移指令	BF label BT label	3/1*2	3	—
	带延迟的条件转移指令	BF/S label BT/S label	2/1*2	3	—
	无条件转移指令	BRA label BRAF Rm BSR label BSRF Rm JMP @Rm JSR @Rm RTS	2	3	—

分类	区分	指令	执行状态	阶段数	竞争

系统控制指令	系统控制 ALU 指令	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 指令 （PR）	LDS.L @Rm+,PR	1	5	• 如果在该指令后紧接着设置使用该指令目标寄存器的指令，则产生竞争。 • MA 与 IF 竞争。
	STS.L 指令 （PR）	STS.L PR,@-Rn	1	4	• MA 与 IF 竞争。
	LDC.L 指令	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	• 如果在该指令后紧接着设置使用该指令目标寄存器的指令，则产生竞争。 • MA 与 IF 竞争。
	STC.L 指令	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 与 IF 竞争。

分类	区分	指令	执行状态	阶段数	竞争

系统控制指令	寄存器 →MAC 传送指令	CLRMAC
		LDS Rm,MACH
		LDS Rm,MACL
			1	4	• 与乘法器竞争。
					• MA 与 IF 竞争。

	寄存器 →DSP 传送指令	LDS Rm,DSR
		LDS Rm,A0
		LDS Rm,X0
		LDS Rm,X1
		LDS Rm,Y0
		LDS Rm,Y1
			1	4	—

	存储器 →MAC 传送指令	LDS.L @Rm+,MACH LDS.L @Rm+,MACL	1	4	• 与乘法器竞争。 • MA 与 IF 竞争。
	存储器 →DSP 传送指令	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→ 寄存器传送指令	STS MACH,Rn STS MACL,Rn	1	5	• 与乘法器竞争。 • 如果在该指令后紧接着设置使用该指令目标寄存器的指令，则产生竞争。 • MA 与 IF 竞争。 • 与 DSP 运算的竞争。
	DSP→ 寄存器传送指令	STS DSR,Rn STS A0,Rn STS X0,Rn STS X1,Rn STS Y0,Rn STS Y1,Rn			
	MAC→ 存储器传送指令	STS.L MACH,@-Rn STS.L MACL,@-Rn	1	4	• 与乘法器竞争。 • MA 与 IF 竞争。
	DSP→ 存储器传送指令	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 指令	RTE	4	5	—
	TRAP 指令	TRAPA #imm	8	9	—
	SLEEP 指令	SLEEP	3	3	—

分类	区分	指令	执行状态	阶段数	竞争
DSP 数据传送指令	X 存储器加载指令	NOPX MOVX.W @Ax,Dx MOVX.W @Ax+,Dx MOVX.W @Ax+Ix,Dx	1	5	—
	X 存储器存储指令	MOVX.W Da, @Ax MOVX.W Da, @Ax+ MOVX.W Da, @Ax+Ix	1	4	• 与 DSP 运算竞争。
	Y 存储器加载指令	NOPY MOVY.W @Ay,Dy MOVY.W @Ay+,Dy MOVY.W @Ay+Iy,Dy	1	5	—
	Y 存储器存储指令	MOVY.W Da, @Ay MOVY.W Da, @Ay+ MOVY.W Da, @Ay+Iy	1	4	• 与 DSP 运算竞争。
	单加载指令	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 与 IF 竞争。
	单存储指令	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 与 IF 竞争。 • 与 DSP 运算的竞争。
DSP 运算指令	ALU 算术运算指令	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	—

分类	区分	指令	执行状态	阶段数	竞争
DSP 运算指令	ALU 算术运算指令	PDMSB Sx,Dz 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	1	5	—

分类	区分	指令	执行状态	阶段数	竞争
DSP 运算指令	ALU 逻辑运算指令	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	—
	移位指令	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	—
	乘法指令	PMULS Se,Sf,Dg	1	5	—
	寄存器之间的传送指令	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	• 与 MOVX.W,MOVY.W, MOVS.W,MOVS.L 竞争。

【注】 *1 表示普通执行状态。
（ ）内的值表示因与后续指令竞争产生的执行状态数。

*2 不转移时为 1 个状态。

*3 为 SH-1 CPU 的阶段数。

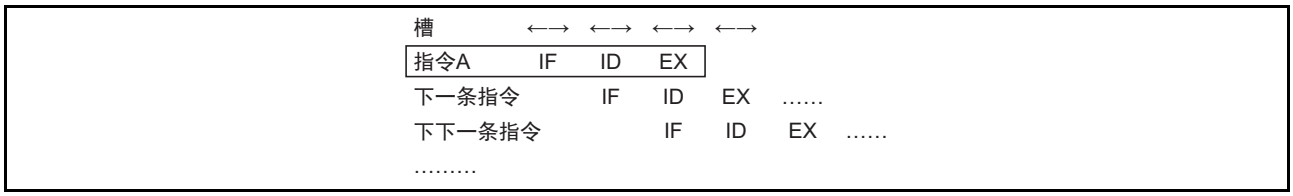
7.4.1 数据传送指令

(1) 寄存器－寄存器之间的传送指令 （通用）

- 指令种类

MOV	#imm,Rn	SWAP.B	Rm,Rn
MOV	Rm,Rn	SWAP.W	Rm,Rn
MOVA	@(disp,PC),R0	XTRACT	Rm,Rn
MOVT	Rn		

- 流水线



- 运行说明

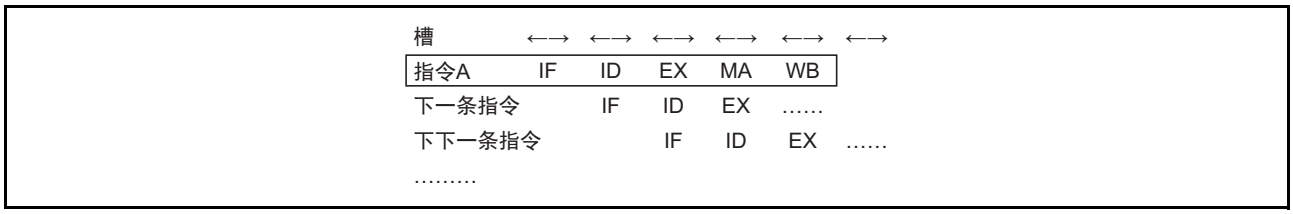
流水线完成 IF、 ID、 EX 等 3 个阶段后结束。在 EX 阶段，通过 ALU 传送数据。

(2) 存储器加载指令 （通用）

- 指令种类

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

- 流水线



- 运行说明

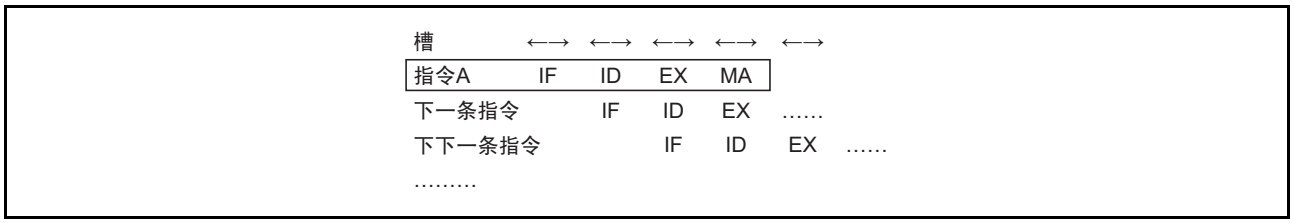
流水线完成 IF、ID、EX、MA、WB 等 5 个阶段后结束。如果在该指令后紧接着设置使用该指令目标寄存器的指令，则产生竞争 （详细内容参照 “7.2.2 使用先行指令目标寄存器时的竞争”）。

(3) 存储器存储指令 （通用）

- 指令种类

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)		

- 流水线



- 运行说明

流水线完成 IF、 ID、 EX、 MA 等 4 个阶段后结束。数据不返回到寄存器，因此没有 WB 阶段。

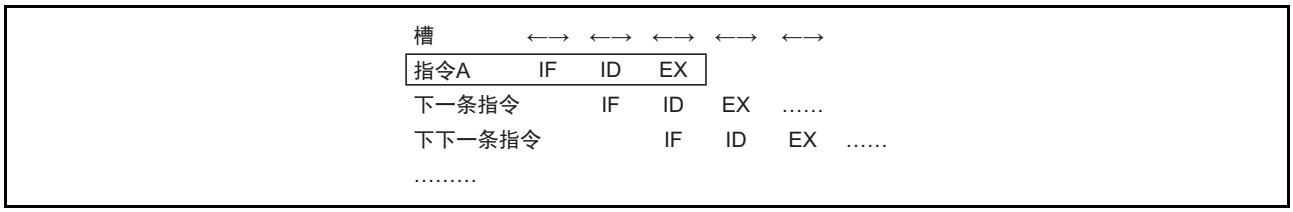
7.4.2 算术运算指令

(1) 寄存器之间的算术运算指令 （乘法类指令除外）（通用或 SH-2、 SH-DSP）

- 指令种类

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

- 流水线



- 运行说明

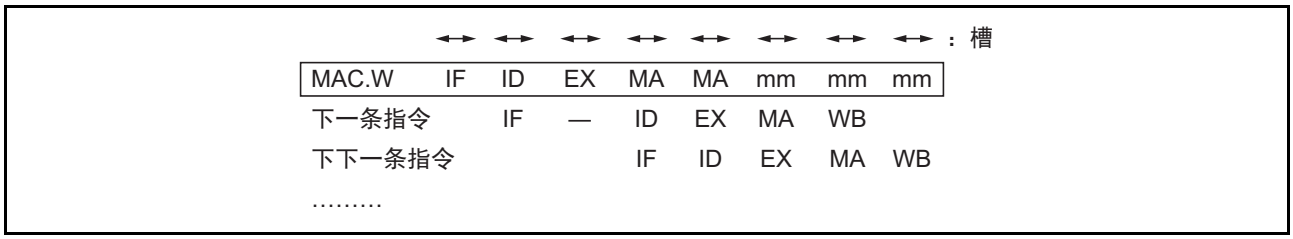
流水线完成 IF、 ID、 EX 等 3 个阶段后结束。在 EX 阶段，通过 ALU 完成数据运算。

(2) 乘加指令 （SH-1）

【指令种类】

MAC.W @Rm+,@Rn+

【流水线】



【运行说明】

流水线完成 IF、ID、EX、MA、MA、mm、mm、mm 等 8 个阶段后结束。第 2 个 MA 阶段在读取存储器的同时进行乘法器存取。 mm 表示乘法器正在运行的状态，最后一个 MA 结束后与槽无关，在 3 个状态期间执行 mm 运行。 MAC.W 指令的下一条指令的 ID 停止 1 个槽的期间。 MAC.W 指令的 2 个 MA 均与 IF 竞争时，如 “7.2.1 取指令 （IF）与存储器存取 （MA）的竞争”所述，拆分槽。

在 MAC.W 指令后出现不使用乘法器的指令时，MAC.W 指令可视为 IF、ID、EX、MA、MA 等 5 段流水线指令。即，此时下一条指令 ID 只停止 1 个槽的期间，之后与通常的流水线运行相同。

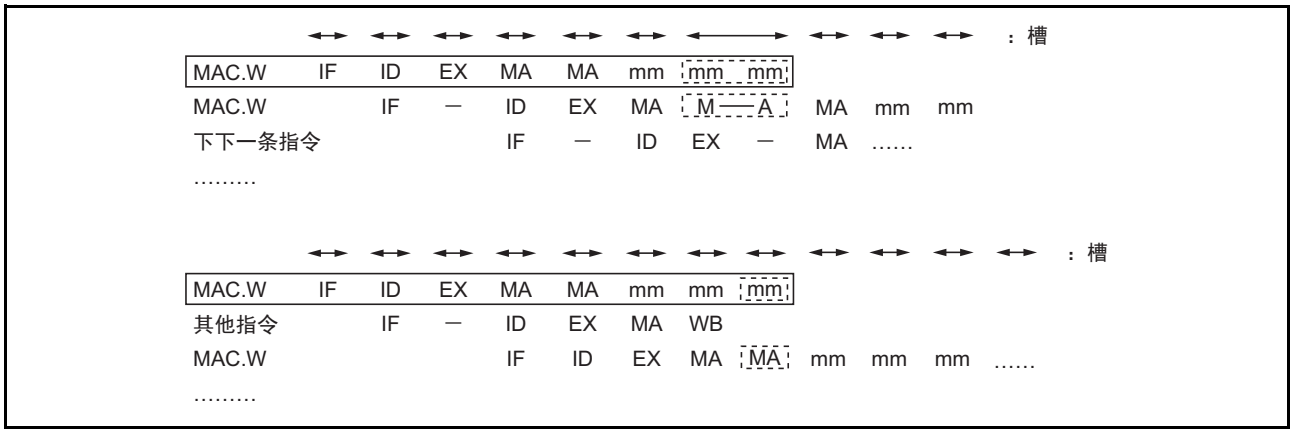
但，在 MAC.W 指令后出现使用乘法器的指令时，产生乘法器竞争，因此与通常的运行不同。此时有以下几种情况：

1. MAC.W指令后紧接着连续出现MAC.W指令时
2. MAC.W指令后紧接着连续出现MULS.W指令时
3. MAC.W指令后紧接着出现STS （寄存器）指令时
4. MAC.W指令后紧接着出现STS.L （存储器）指令时
5. MAC.W指令后紧接着出现LDS （寄存器）指令时
6. MAC.W指令后紧接着出现LDS.L （存储器）指令时

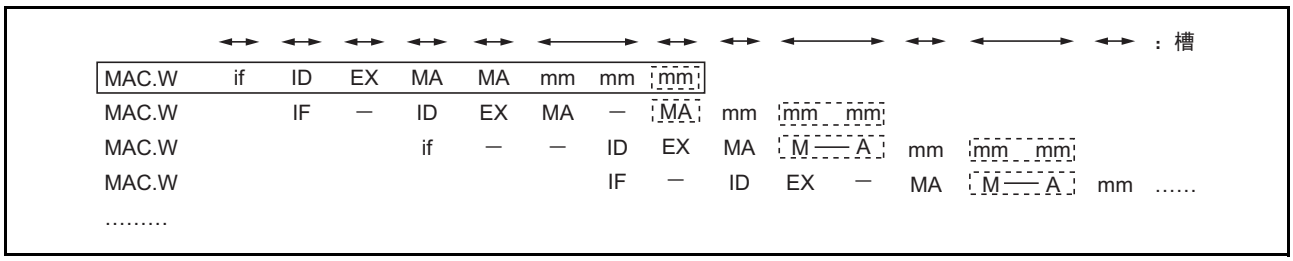
1. MAC.W指令后紧接着连续出现MAC.W指令时

   MAC.W指令的第2个MA与之前乘法类指令产生的mm竞争时，延长该MA的总线周期直至mm结束（下图的M—— A），延长的MA为1个槽。

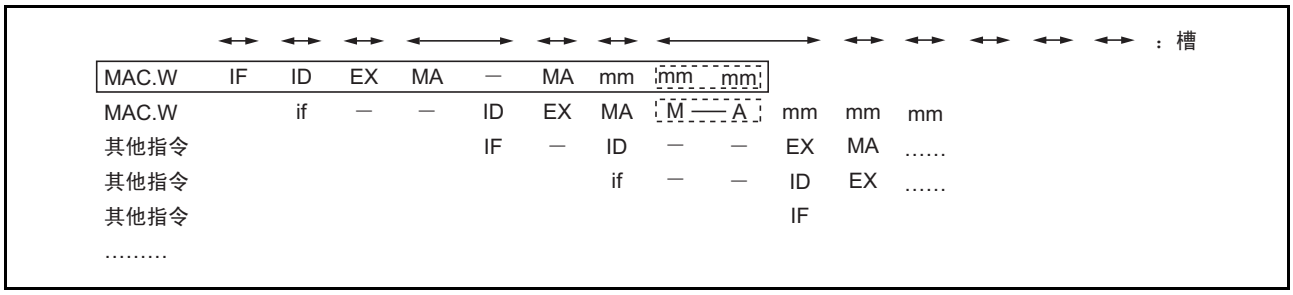
   如果在MAC.W指令与MAC.W指令之间至少插入1条与乘法器无关的指令，则不会出现因MAC.W指令之间的乘法器竞争而产生的停止。



因 MAC.W 连续，即使在 MA 与 IF 竞争时出现指令执行滞后，也有可能不会出现乘法器的竞争。详细内容参照下图，该图考虑了 MA 与 IF 的竞争。

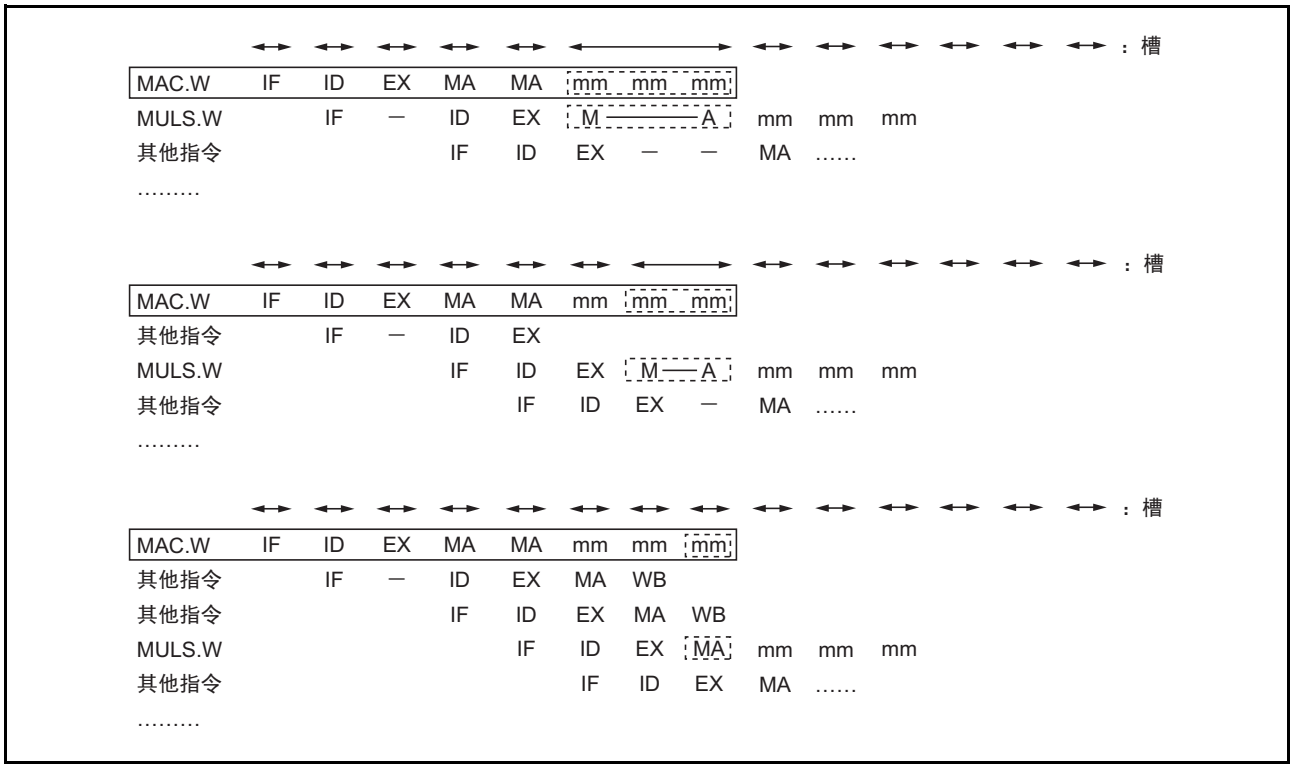


MAC.W指令的第2个MA延长至mm结束时，如果该MA与IF竞争，则照常拆分槽。详细内容参照下图，该图考虑了MA与IF的竞争。



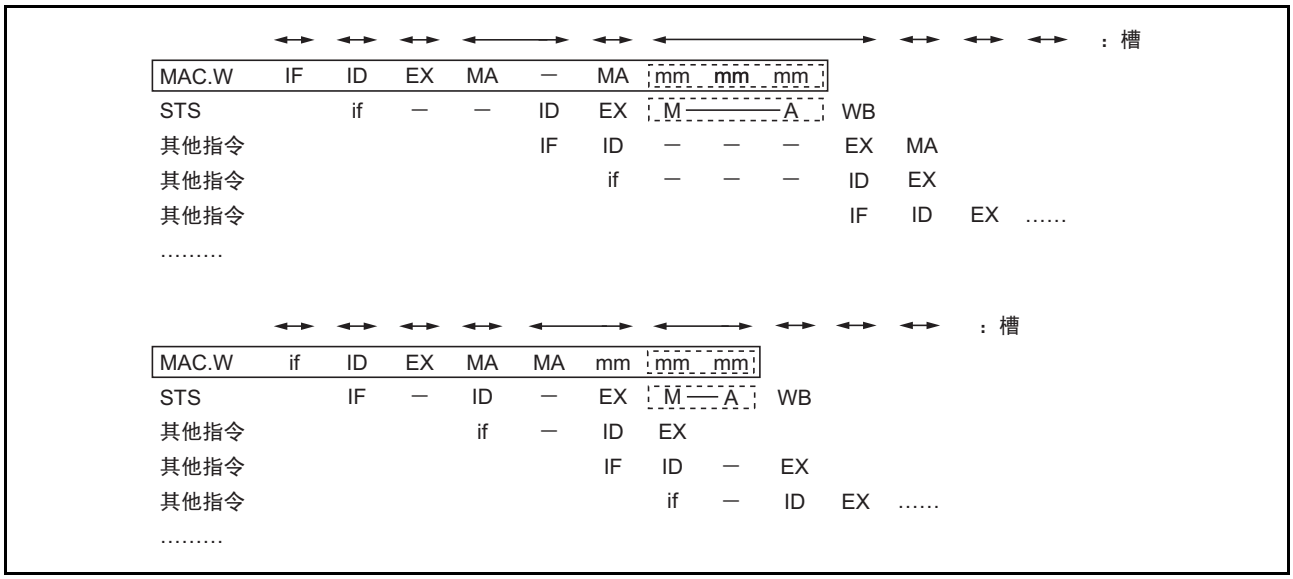
2. MAC.W指令后紧接着连续出现MULS.W指令时

MULS.W指令有用于乘法器存取的MA阶段。 MAC.W指令乘法器 （mm）的运行过程中，如果MULS.W的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在MAC.W与MULS.W之间至少插入2条与乘法器无关的指令，则不会出现因MAC.W与MULS.W竞争而产生的停止。如果MULS.W的MA也与IF竞争，则拆分槽。



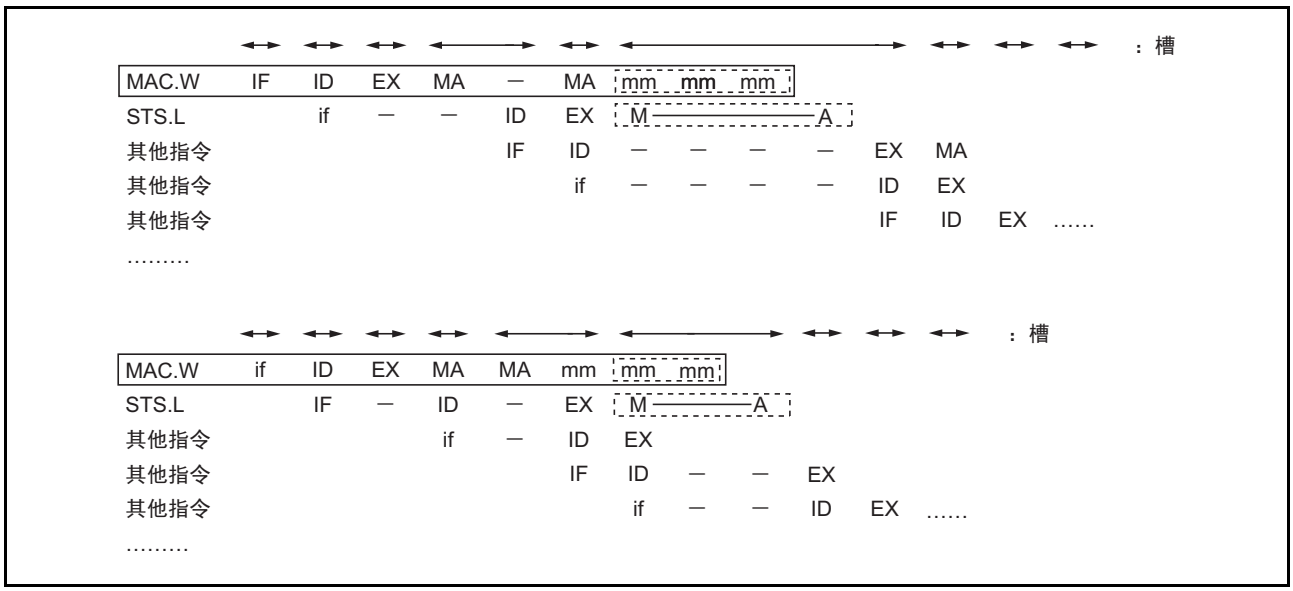
3. MAC.W指令后紧接着出现STS （寄存器）指令时

通过STS指令将MAC寄存器的内容存储至通用寄存器时，如后文所述， STS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束 （下图的M——A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



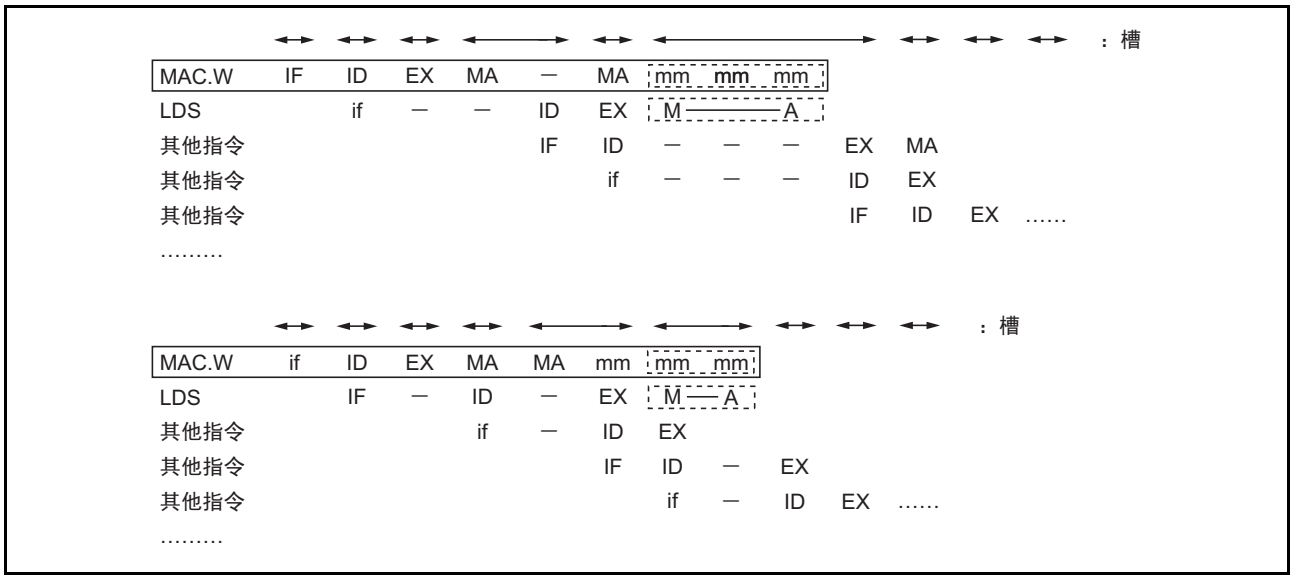
4. MAC.W指令后紧接着出现STS.L （存储器）指令时

通过STS指令将MAC寄存器的内容存储至存储器时，如后文所述， STS指令进入乘法器存取及存储器写入的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束后经过1个状态 （下图的M——A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



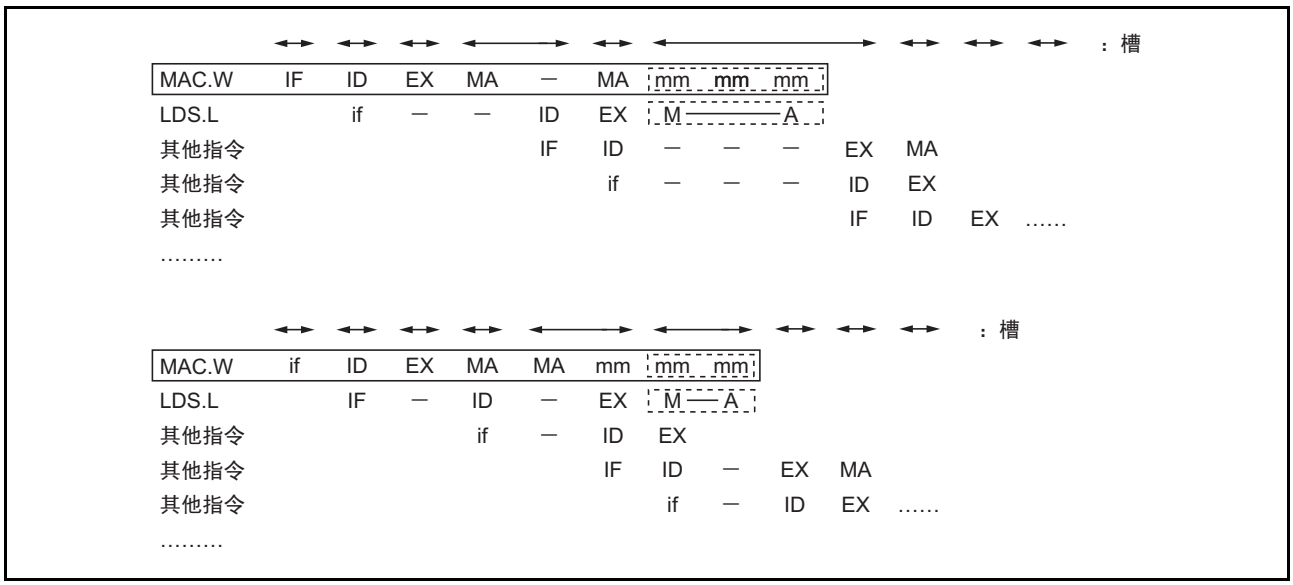
5. MAC.W指令后紧接着出现LDS （寄存器）指令时

通过LDS指令从通用寄存器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



6. MAC.W指令后紧接着出现LDS.L （存储器）指令时

通过LDS指令从存储器加载MAC寄存器的内容时，如后文所述， LDS指令进入存储器存取与乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束（下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



(3) 乘加指令 （SH-2、 SH-DSP）

- 指令种类

MAC.W @Rm+,@Rn+

- 流水线

槽	←→	←→	←→	←→	←→	←→	←→	←→
指令A	IF	ID	EX	MA	MA	mm	mm	
下一条指令		IF	—	ID	EX	MA	WB	
下下一条指令				IF	ID	EX	MA	WB
.....								

- 运行说明

流水线完成 IF、 ID、 EX、 MA、 MA、 mm、 mm 等 7 个阶段后结束。第 2 个 MA 在读取存储器的同时存取乘法器。 mm 表示乘法器正在运行的状态，最后 1 个 MA 结束后与槽无关，在 2 个状态执行 mm 运行。MAC.W 指令的下一条指令的 ID 停止 1 个槽的期间。 MAC.W 指令的 2 个 MA 均与 IF 竞争时，如 “7.2.1 取指令 （IF）与存储器存取 （MA）的竞争”所述，拆分槽。

MAC.W 指令后出现不使用乘法器的指令时，MAC.W 指令可视为 IF、 ID、 EX、 MA、 MA 等 5 段流水线指令。即，此时下一条指令的 ID 只停止 1 个槽的期间，之后与通常的流水线运行相同。

但， MAC.W 指令后出现使用乘法器的指令时，产生乘法器竞争，因此与通常的运行不同。此时有以下几种情况：

1. MAC.W指令后紧接着连续出现MAC.W指令时
2. MAC.W指令后紧接着连续出现MAC.L指令时
3. MAC.W指令后紧接着连续出现MULS.W指令时
4. MAC.W指令后紧接着连续出现DMULS.L指令时
5. MAC.W指令后紧接着出现STS （寄存器）指令时
6. MAC.W指令后紧接着出现STS.L （存储器）指令时
7. MAC.W指令后紧接着出现LDS （寄存器）指令时
8. MAC.W指令后紧接着出现LDS.L （存储器）指令时

1. MAC.W指令后紧接着连续出现MAC.W指令时

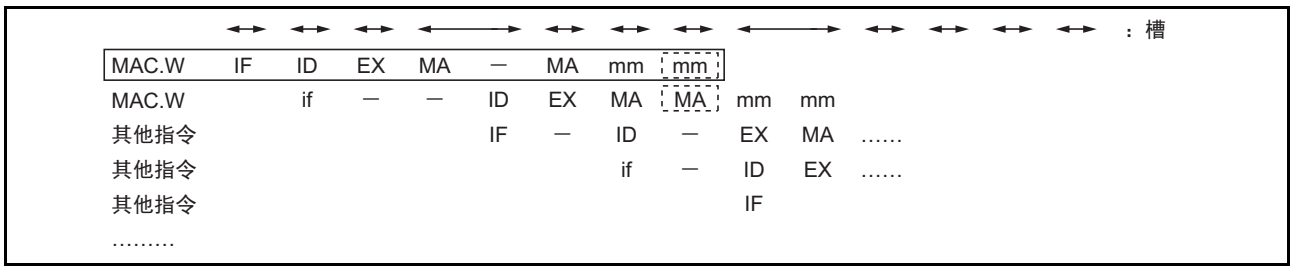
MAC.W指令的第2个MA不与之前乘法类指令产生的mm竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	: 槽
MAC.W	IF	ID	EX	MA	MA	mm	mm					
MAC.W		IF	—	ID	EX	MA	MA	mm	mm			
下下一条指令				IF	—	ID	EX	MA	.....			
.....												

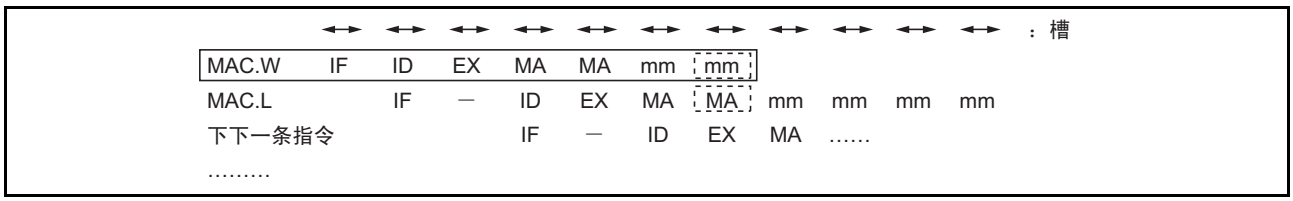
因MAC.W连续，在MA与IF竞争时有可能出现指令执行滞后。详细内容参照下图，该图考虑了MA与IF的竞争。

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	: 槽
MAC.W	if	ID	EX	MA	MA	mm	mm							
MAC.W		IF	—	ID	EX	MA	—	MA	mm	mm				
MAC.W				if	—	—	ID	EX	MA	MA	mm	mm		
MAC.W							IF	—	ID	EX	MA	MA	mm	.....
.....														

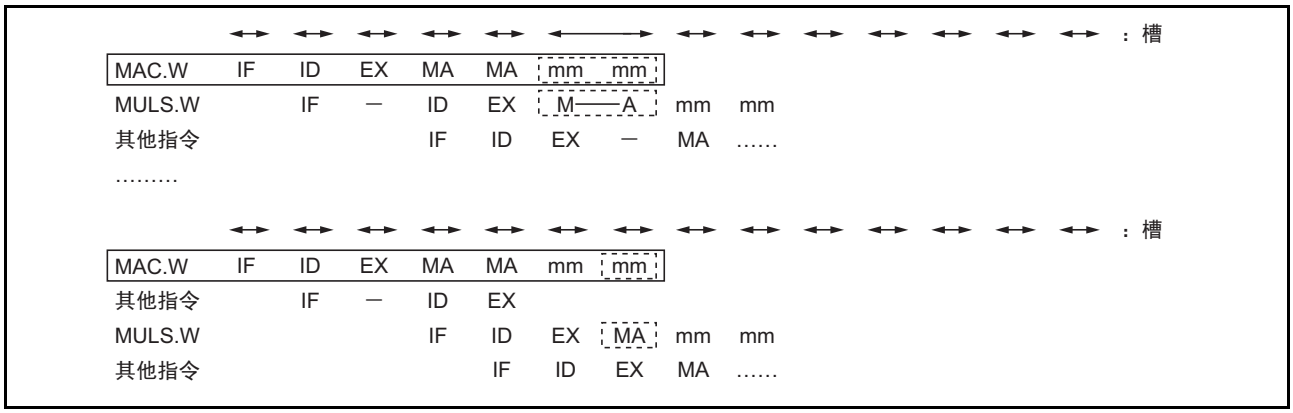
如果MAC.W指令的第2个MA与IF竞争，则照常拆分槽。详细内容参照下图，该图考虑了MA与IF的竞争。



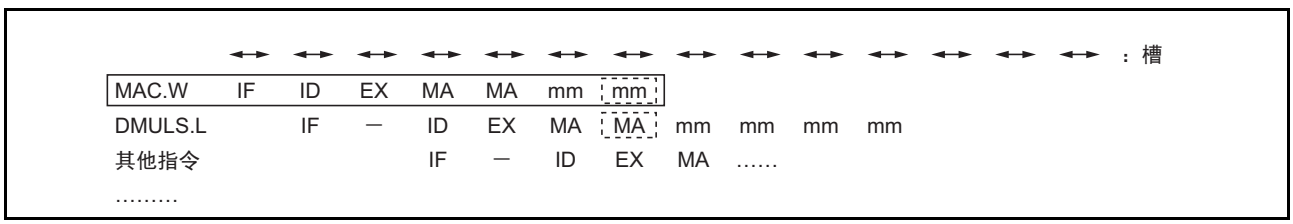
2. MAC.W指令后紧接着连续出现MAC.L指令时
MAC.W指令的第2个MA不与之前乘法类指令产生的mm竞争。



3. MAC.W指令后紧接着连续出现MULS.W指令时
MULS.W指令有乘法器存取的MA阶段。 MAC.W指令的乘法器运行过程中 （mm），如果MULS.W的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在MAC.W与MULS.W之间至少插入1个与乘法器无关的指令，则不会出现因MAC.W与MULS.W竞争而产生的停止。 MAC.W指令的乘法器运行过程中 （mm）， MULS.W的MA不产生竞争。如果MULS.W的MA也与IF竞争，则拆分槽。

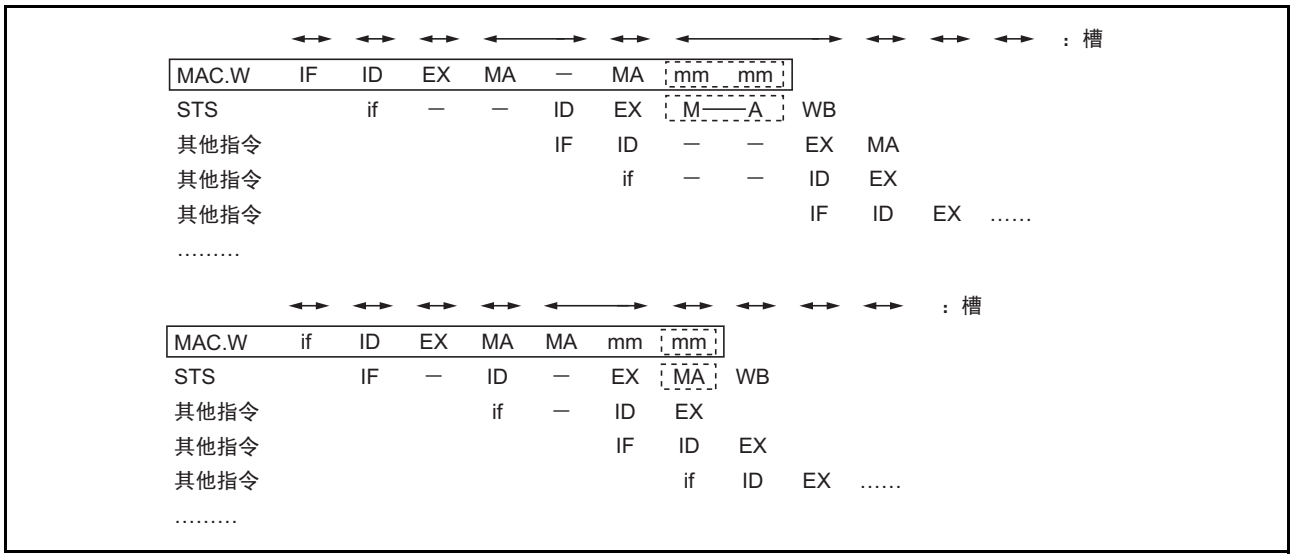


4. MAC.W指令后紧接着连续出现DMULS.L指令时
MULS.W指令有乘法器存取的MA阶段。 MAC.W指令的乘法器运行过程中 （mm）， MULS.W的MA不产生竞争。如果MULS.W的MA也与IF竞争，则拆分槽。



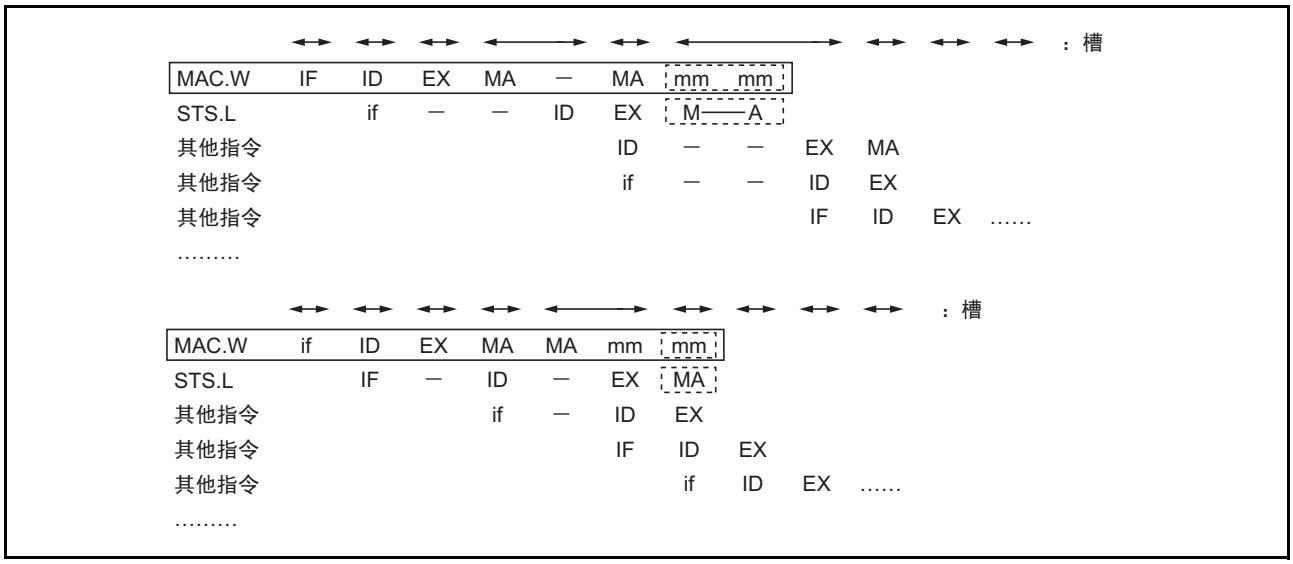
5. MAC.W指令后紧接着出现STS （寄存器）指令时

通过STS指令将MAC寄存器的内容存储至通用寄存器时，如后文所述， STS指令进入乘法器存取MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。

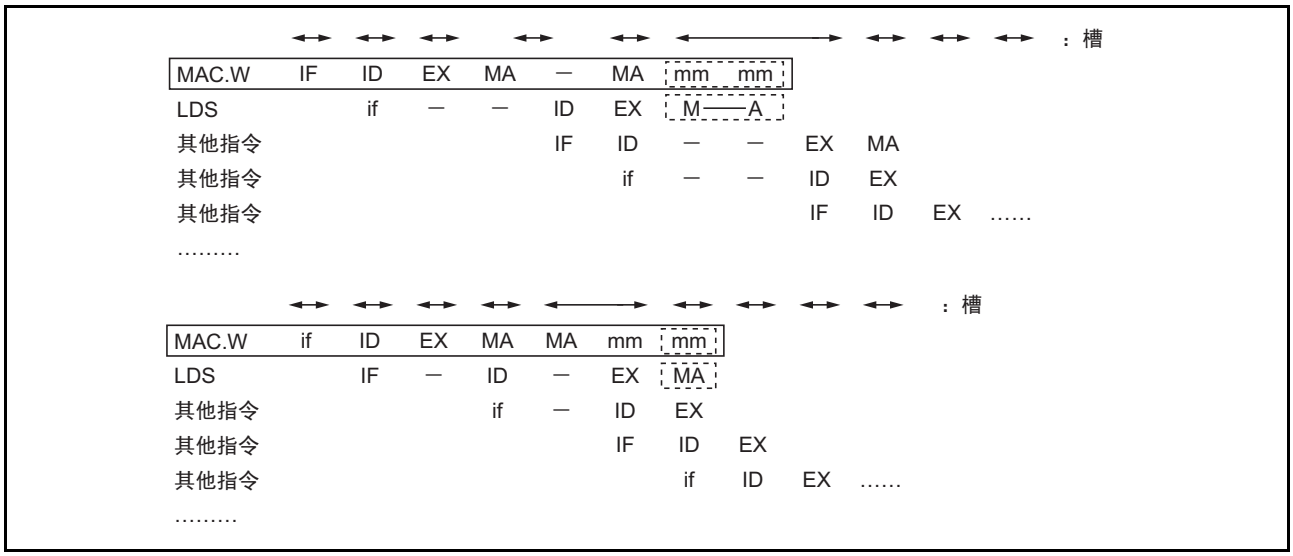


6. MAC.W指令后紧接着出现STS.L （存储器）指令时

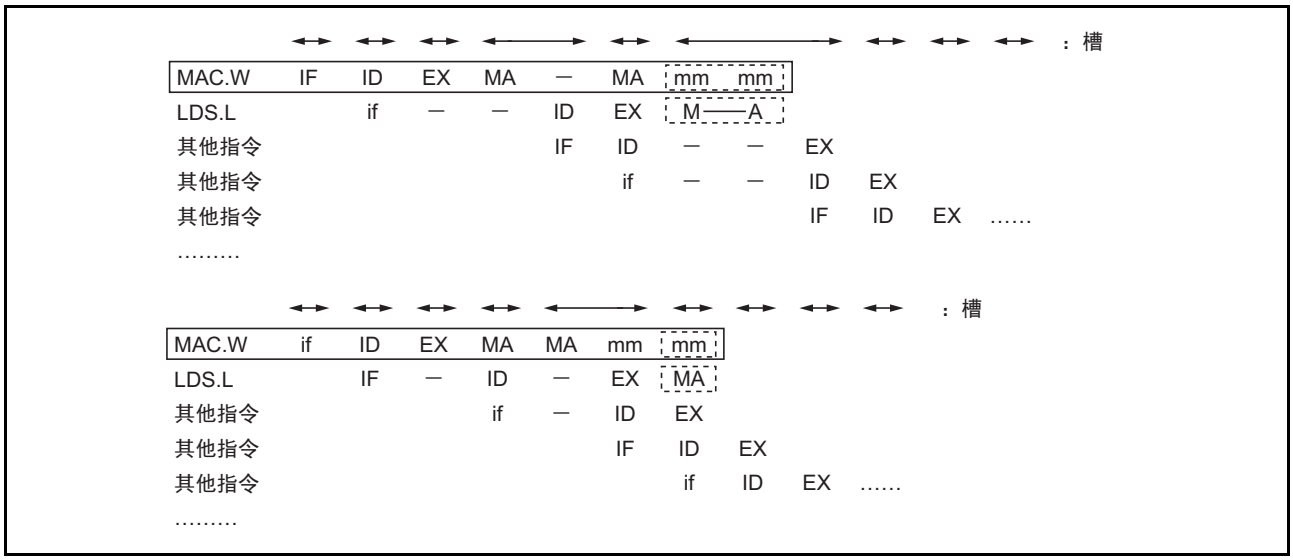
通过STS指令将MAC寄存器的内容存储至存储器时，如后文所述， STS指令进入乘法器存取及存器写入的MA阶段。详细内容如下图所示，该图考虑了MA与IF的竞争。



7. MAC.W指令后紧接着出现LDS （寄存器）指令时
通过LDS指令从通用寄存器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M——A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



8. MAC.W指令后紧接着出现LDS.L （存储器）指令时
通过LDS指令从存储器加载MAC寄存器的内容时，如后文所述， LDS指令进入存储器存取及乘法存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束（下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



(4) 双精度乘加指令 （SH-2、 SH-DSP）

- 指令种类

MAC.L @Rm+,@Rn+

- 流水线

槽	←→	←→	←→	←→	←→	←→	←→	←→	←→
指令A	IF	ID	EX	MA	MA	mm	mm	mm	mm
下一条指令		IF	—	ID	EX	MA	WB		
下下一条指令				IF	ID	EX	MA	WB	
.....									

- 运行说明

流水线完成 IF、ID、EX、MA、MA、mm、mm、mm、mm 等 9 个阶段后结束。第 2 个 MA 在读取存储器的同时存取乘法器。 mm 表示乘法器正在运行的状态，最后 1 个 MA 结束后与槽无关，在 4 个状态执行 mm 运行。 MAC.L 指令的下一条指令 ID 停止 1 个槽的期间。 MAC.L 指令的 2 个 MA 均与 IF 竞争时，如 “7.2.1 取指令 （IF）与存储器存取 （MA）的竞争”所述，拆分槽。

MAC.L 指令后出现不使用乘法器的指令时，MAC.L 指令可视为 IF、ID、EX、MA、MA 等 5 段流水线指令。即，此时下一条指令的 ID 只停止 1 个槽的期间，之后与通常的流水线运行相同。

但， MAC.L 指令后出现使用乘法器的指令时，产生乘法器竞争，因此与通常的运行不同。此时有以下几种情况：

1. MAC.L指令后紧接着连续出现MAC.L指令时
2. MAC.L指令后紧接着连续出现MAC.W指令时
3. MAC.L指令后紧接着连续出现DMULS.L指令时
4. MAC.L指令后紧接着连续出现MULS.W指令时
5. MAC.L指令后紧接着出现STS （寄存器）指令时
6. MAC.L指令后紧接着出现STS.L （存储器）指令时
7. MAC.L指令后紧接着出现LDS （寄存器）指令时
8. MAC.L指令后紧接着出现LDS.L （存储器）指令时

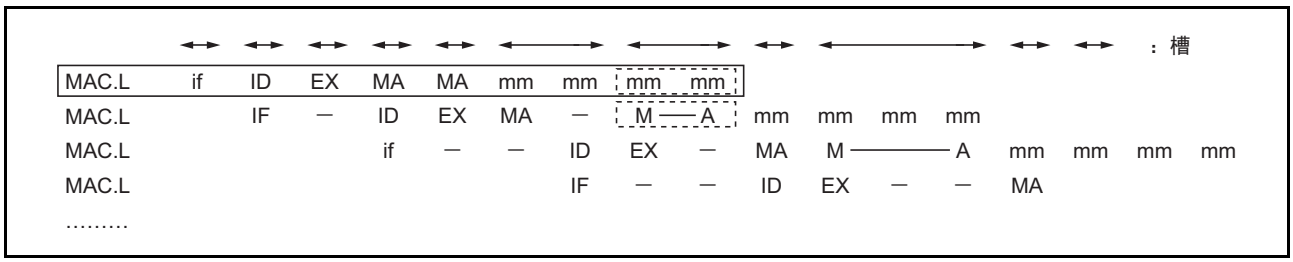
1. MAC.L指令后紧接着连续出现MAC.L指令时

MAC.L指令的第2个MA与之前乘法类指令产生的mm竞争时，延长该MA的总线周期直至mm结束（下图的M—— A），延长的MA为1个槽。如果在MAC.L指令与MAC.L指令之间至少插入2条与乘法器无关的指令，则不会出现因MAC.L指令之间的乘法器竞争而产生的停止。

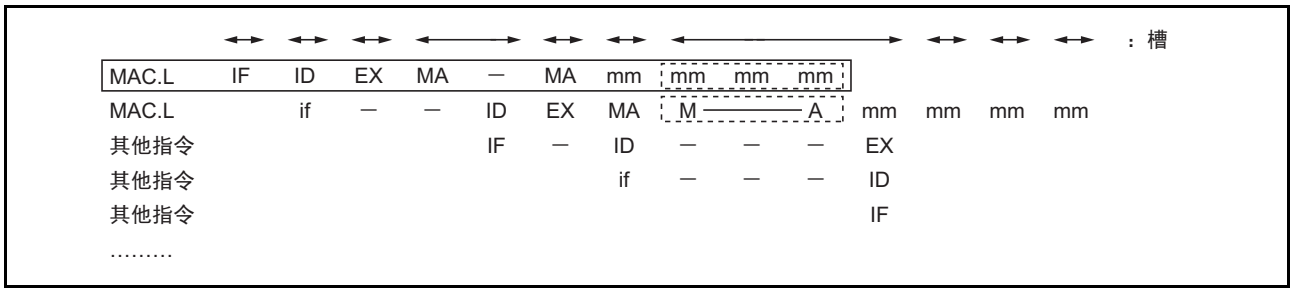
	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	：槽
MAC.L	IF	ID	EX	MA	MA	mm	mm	mm	mm					
MAC.L		IF	—	ID	EX	MA	M	—	—	A	mm	mm	mm	mm
下下一条指令				IF	—	ID	EX	—	—	MA	.....			
.....														

	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	←→	：槽
MAC.L	IF	ID	EX	MA	MA	mm	mm	mm	mm					
其他指令		IF	—	ID	EX	MA	WB							
其他指令				IF	ID	EX	MA	WB						
MAC.L					IF	ID	EX	MA	MA	mm	mm	mm	mm	
.....														

因MAC.L连续，即使在MA与IF竞争时出现指令执行滞后，也有可能不会出现乘法器的竞争。详细内容参照下图，该图考虑了MA与IF的竞争。



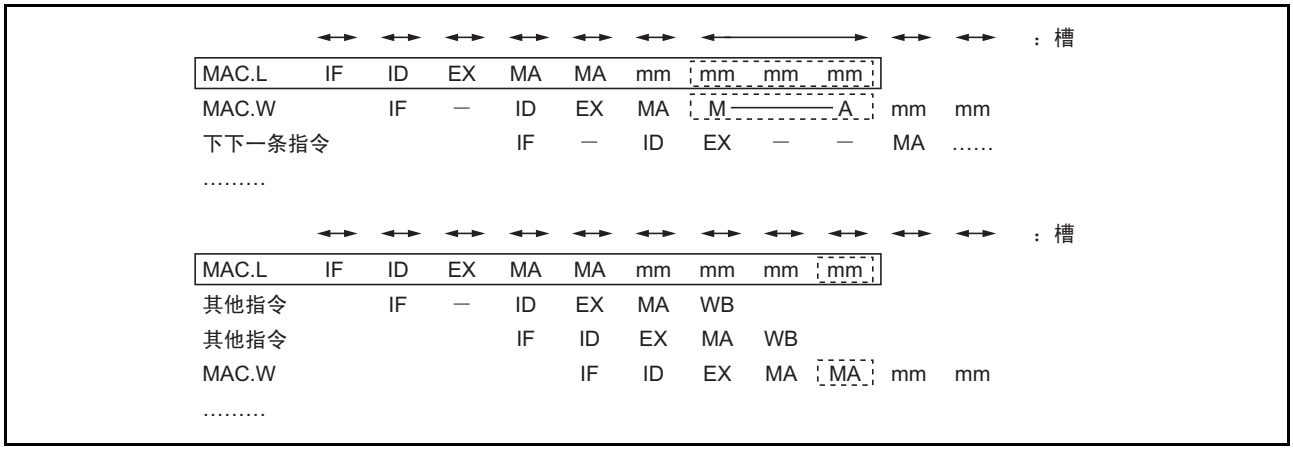
如果MAC.L指令的第2个MA延长至mm结束，且该MA与IF竞争，则照常拆分槽。详细内容参照下图，该图考虑了MA与IF的竞争。



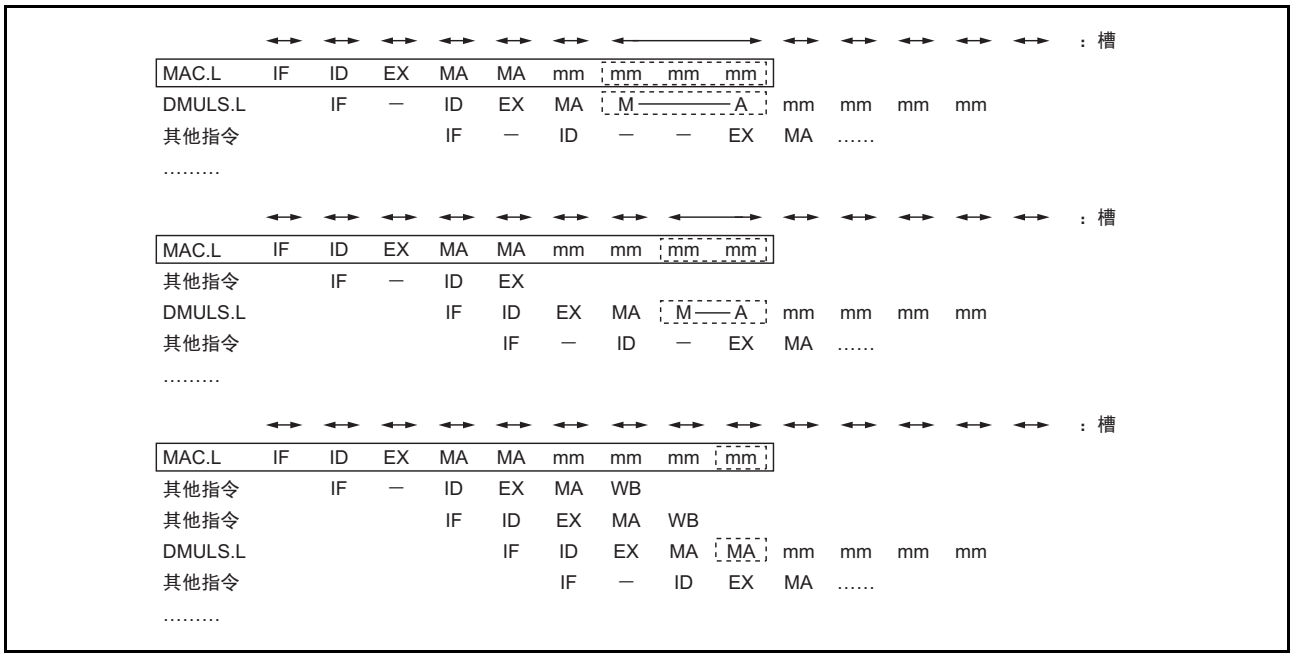
2. MAC.L指令后紧接着连续出现MAC.W指令时

MAC.W指令的第2个MA与之前乘法类指令产生的mm竞争时，延长该MA的总线周期直至mm结束（下图的M—— A），延长的MA为1个槽。

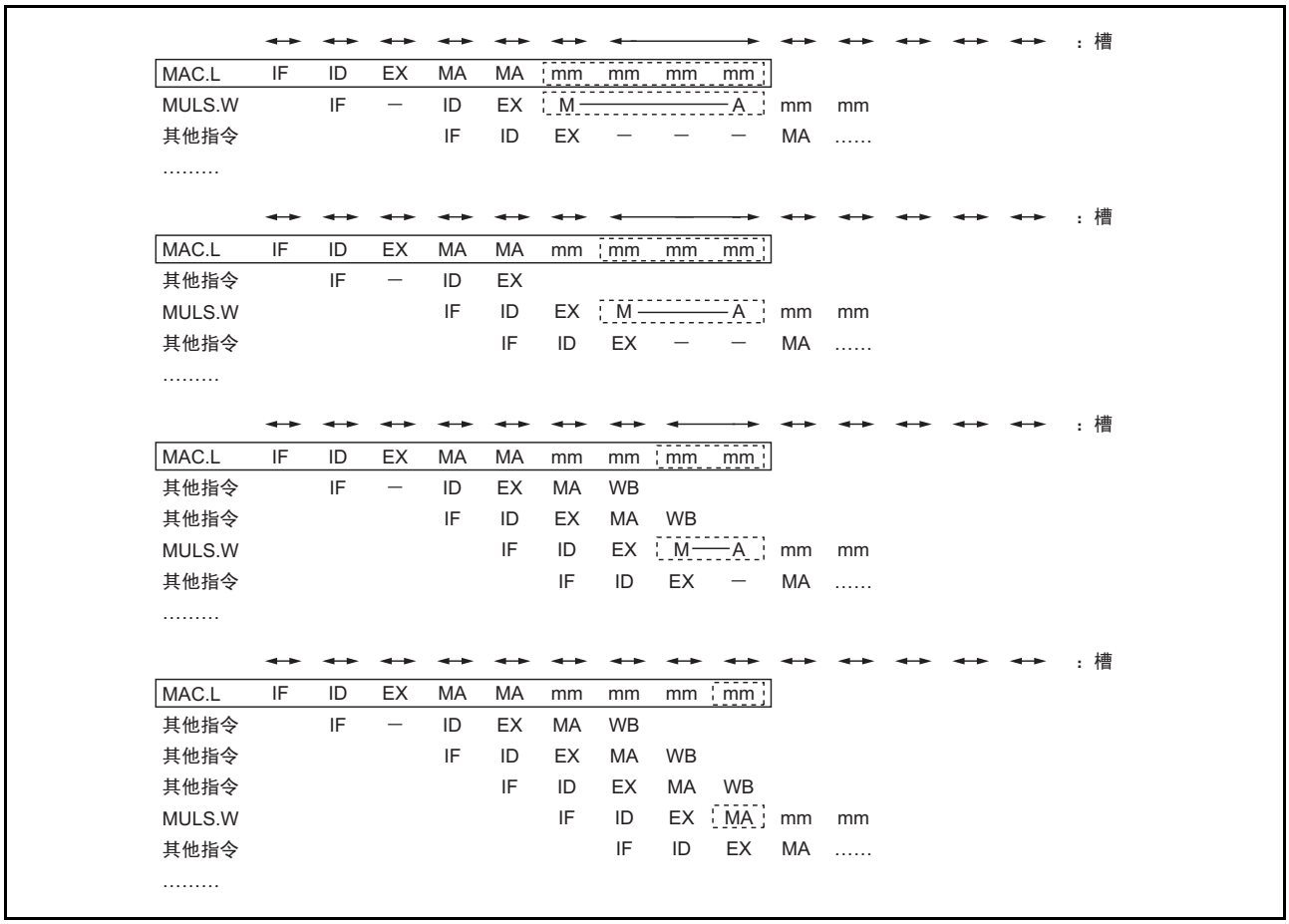
如果在MAC.L指令与MAC.W指令之间至少插入2条与乘法器无关的指令，则不会出现因MAC.L指令与MAC.W指令的乘法器竞争而产生的停止。



3. MAC.L指令后紧接着连续出现DMULS.L指令时
DMULS.L指令有乘法器存取的MA阶段。 MAC.L指令的乘法器运行过程中 （mm），如果DMULS.L指令的第2个MA产生竞争，则延长该MA直至mm结束 （下图的M——A），并形成1个槽。如果在MAC.L与DMULS.L之间至少插入2条与乘法器无关的指令，则不会出现因MAC.L与DMULS.L竞争而产生的停止。如果DMULS.L的MA也与IF竞争，则拆分槽。

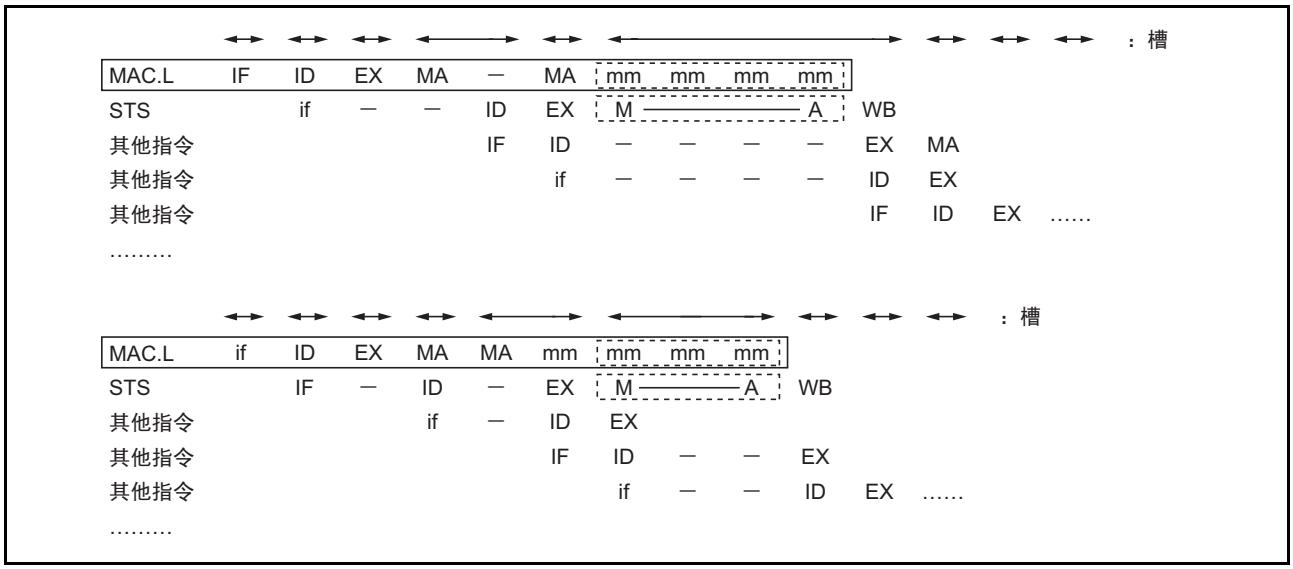


4. MAC.L指令后紧接着连续出现MULS.W指令时
MULS.W指令有乘法器存取的MA阶段。 MAC.L指令的乘法器运行过程中 （mm），如果MULS.W的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在MAC.L与MULS.W之间至少插入3条与乘法器无关的指令，则不会出现因MAC.L与MULS.W竞争而产生的停止。如果MULS.W的MA也与IF竞争，则拆分槽。



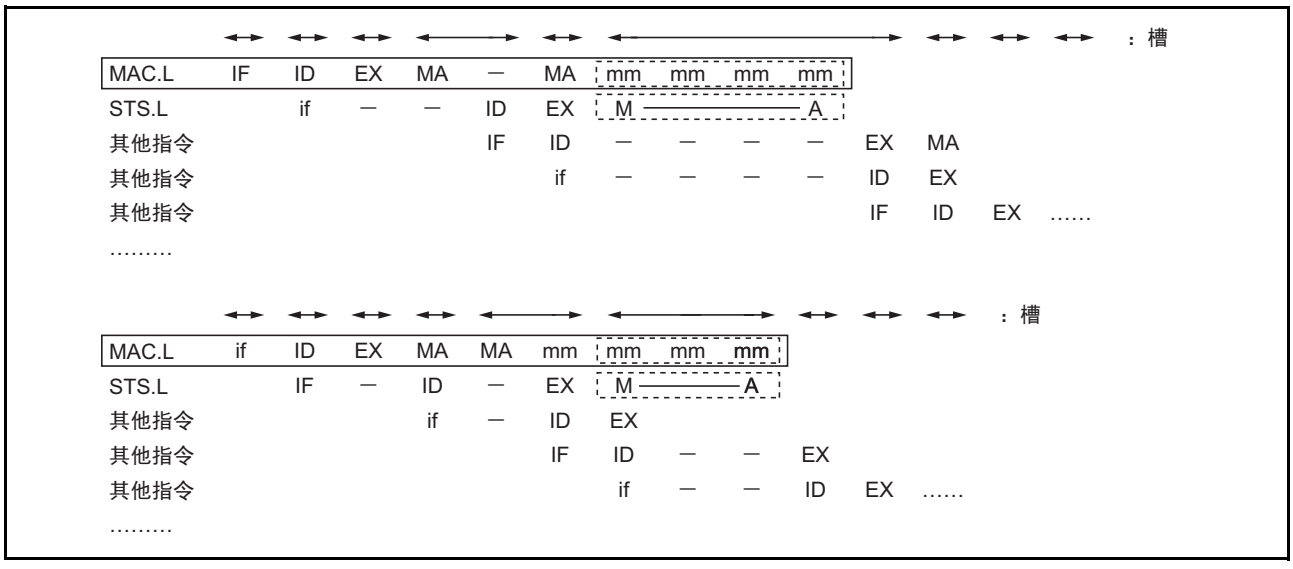
5. MAC.L指令后紧接着出现STS （寄存器）指令时

通过STS指令将MAC寄存器的内容存储至通用寄存器时，如后文所述， STS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。

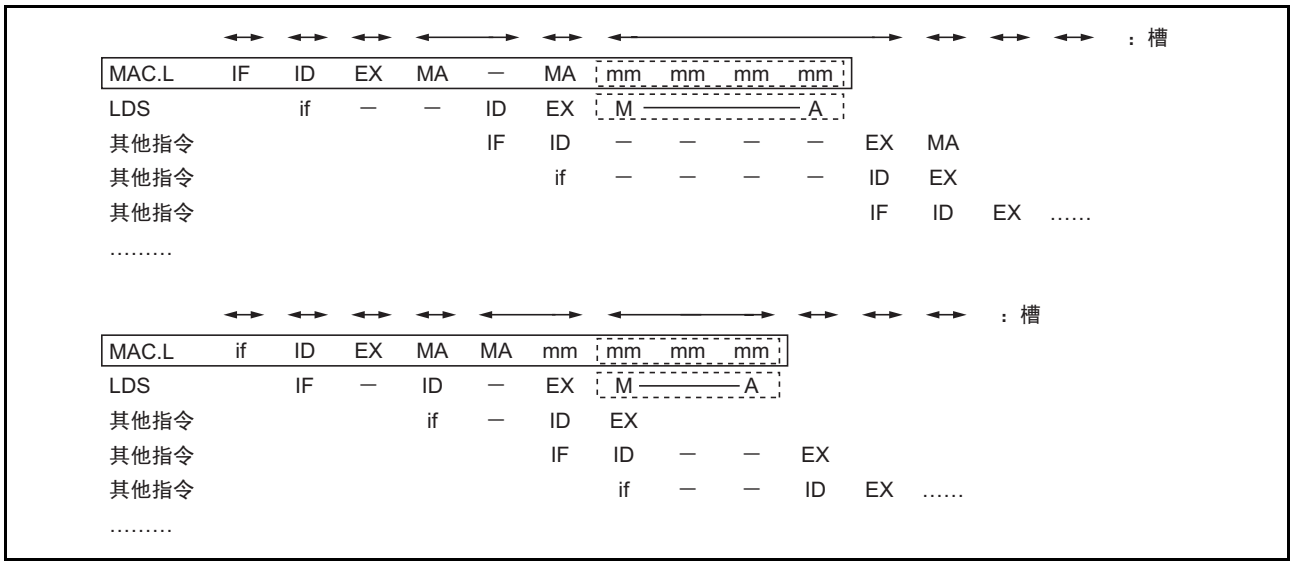


6. MAC.L指令后紧接着出现STS.L （存储器）指令时

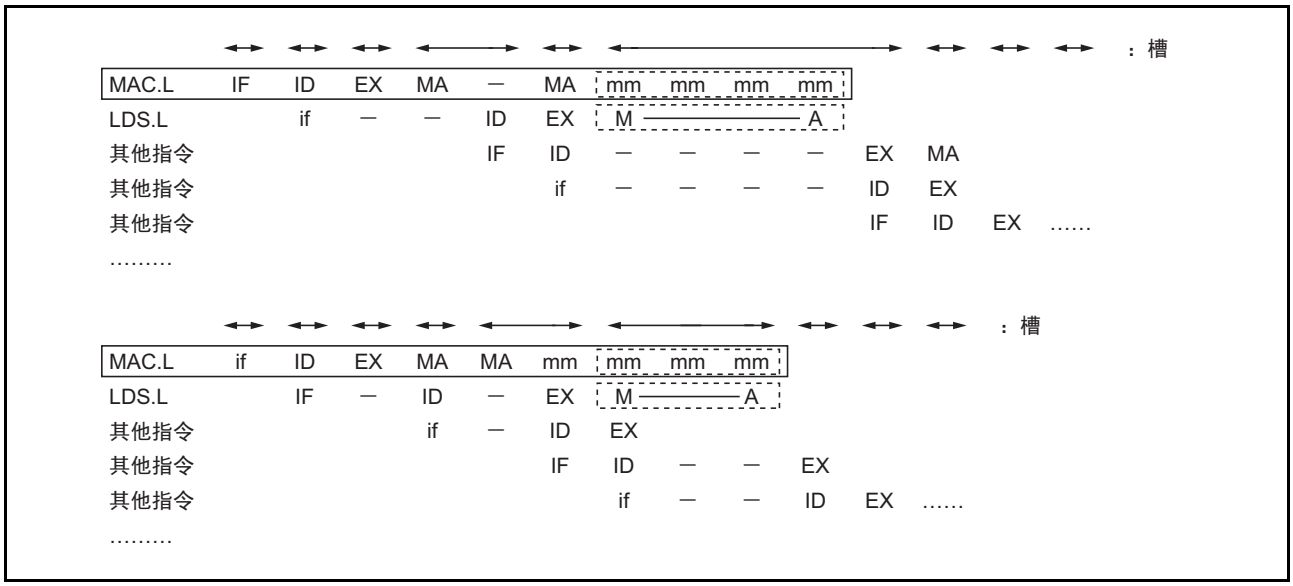
通过STS指令将MAC寄存器的内容存储至存储器时，如后文所述， STS指令进入乘法器存取及存储器写入的MA阶段，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



7. MAC.L指令后紧接着出现LDS （寄存器）指令时
通过LDS指令从通用寄存器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M——A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



8. MAC.L指令后紧接着出现LDS.L （存储器）指令时
通过LDS指令从存储器加载MAC寄存器的内容时，如后文所述， LDS指令进入存储器存取及乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束（下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。

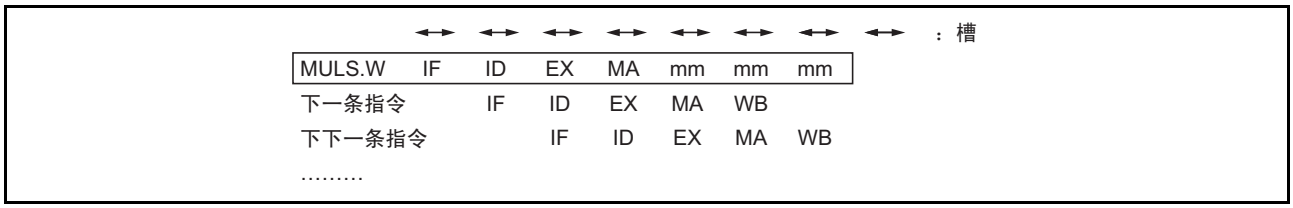


(5) 乘法指令 （SH-1）

【指令种类】

MULS.W Rm,Rn
MULU.W Rm,Rn

【流水线】



【运行说明】

流水线完成 IF、ID、EX、MA、mm、mm、mm 等 7 个阶段后结束。MA 用来存取乘法器。mm 表示乘法器正在运行的状态， MA 结束后与槽无关，在 3 个状态执行 mm 运行。 MULS.W 指令的 MA 也与 IF 竞争时，如 “7.2.1 取指令 （IF）与存储器存取 （MA）的竞争”所述，拆分槽。

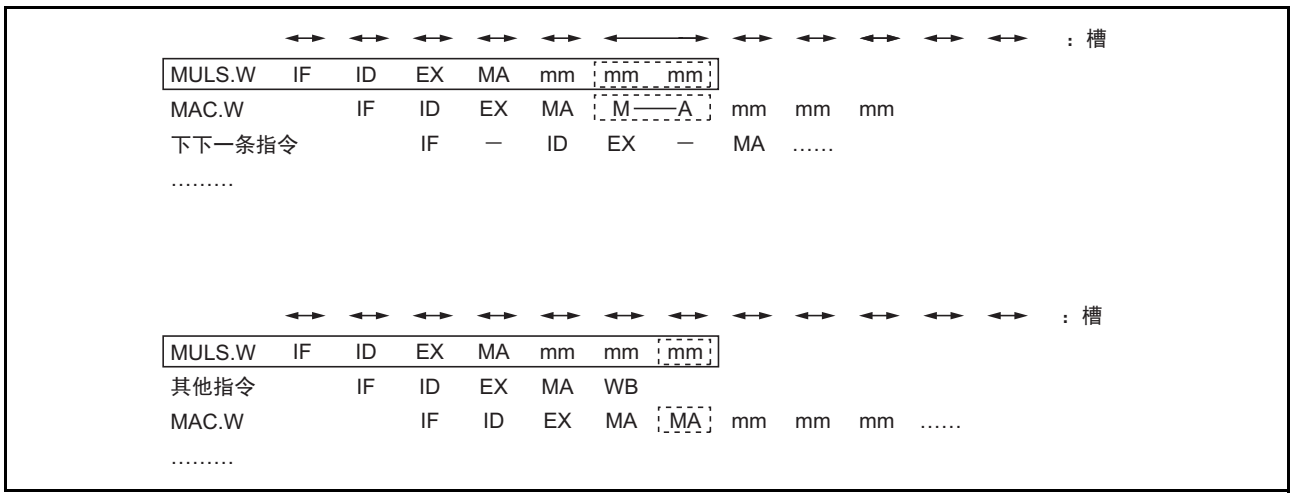
MULS.W 指令后出现不使用乘法器的指令时， MULS.W 指令可视为 IF、 ID、 EX、 MA 等 4 段流水线指令。即，此时与通常的流水线运行相同。

但， MULS.W 指令后出现使用乘法器的指令时，产生乘法器竞争，因此与通常的运行不同。此时有以下几种情况：

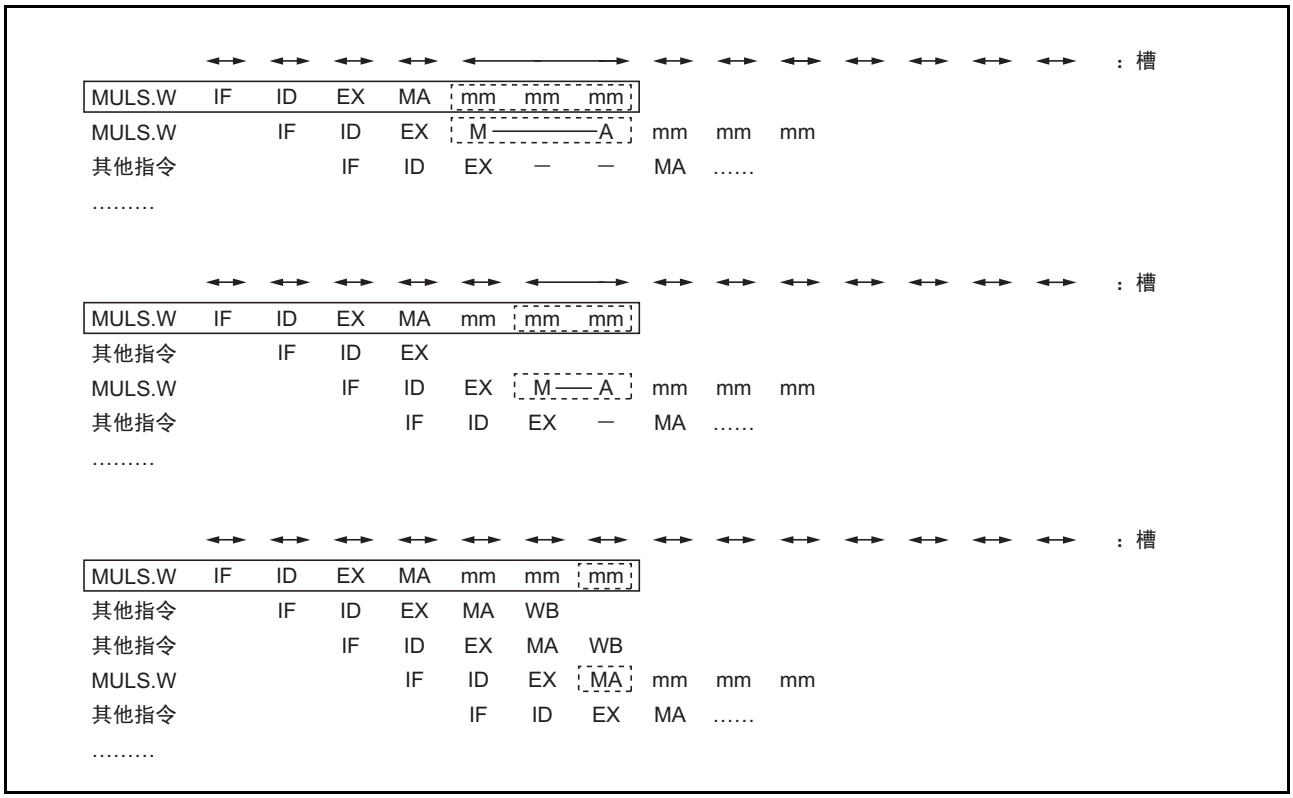
1. MULS.W指令后紧接着连续出现MAC.W指令时
2. MULS.W指令后紧接着连续出现MULS.W指令时
3. MULS.W指令后紧接着出现STS （寄存器）指令时
4. MULS.W指令后紧接着出现STS.L （存储器）指令时
5. MULS.W指令后紧接着出现LDS （寄存器）指令时
6. MULS.W指令后紧接着出现LDS.L （存储器）指令时

1. MULS.W指令后紧接着连续出现MAC.W指令时

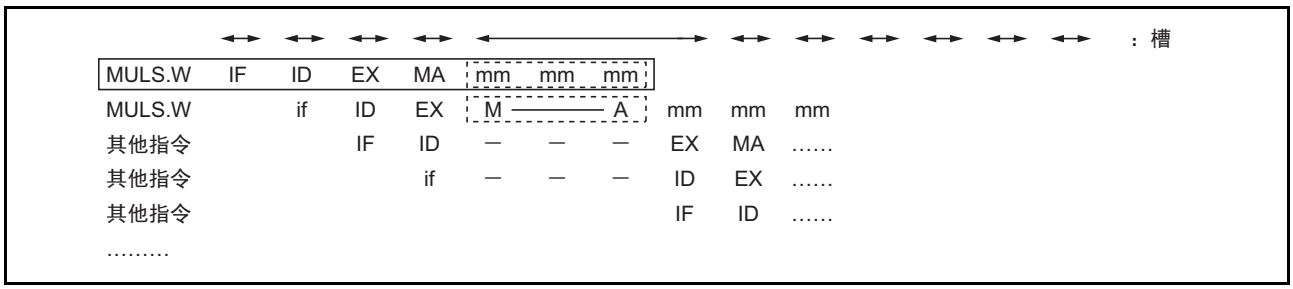
   MAC.W指令的第2个MA与之前乘法类指令产生的mm竞争时，延长该MA的总线周期直至mm结束（下图的M—— A），延长的MA为1个槽。如果在MULS.W指令与MAC.W指令之间至少插入1条与乘法器无关的指令，则不会出现因MULS.W与MAC.W指令之间的乘法器竞争而产生的停止。



2. MULS.W指令后紧接着连续出现MULS.W指令时
MULS.W指令有乘法器存取的MA阶段。 MULS.W指令的乘法器运行过程中 （mm），如果其他MULS.W的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在MULS.W与MULS.W之间至少插入2条与乘法器无关的指令，则不会出现因MULS.W与MULS.W竞争而产生的停止。如果MULS.W的MA也与IF竞争，则拆分槽。

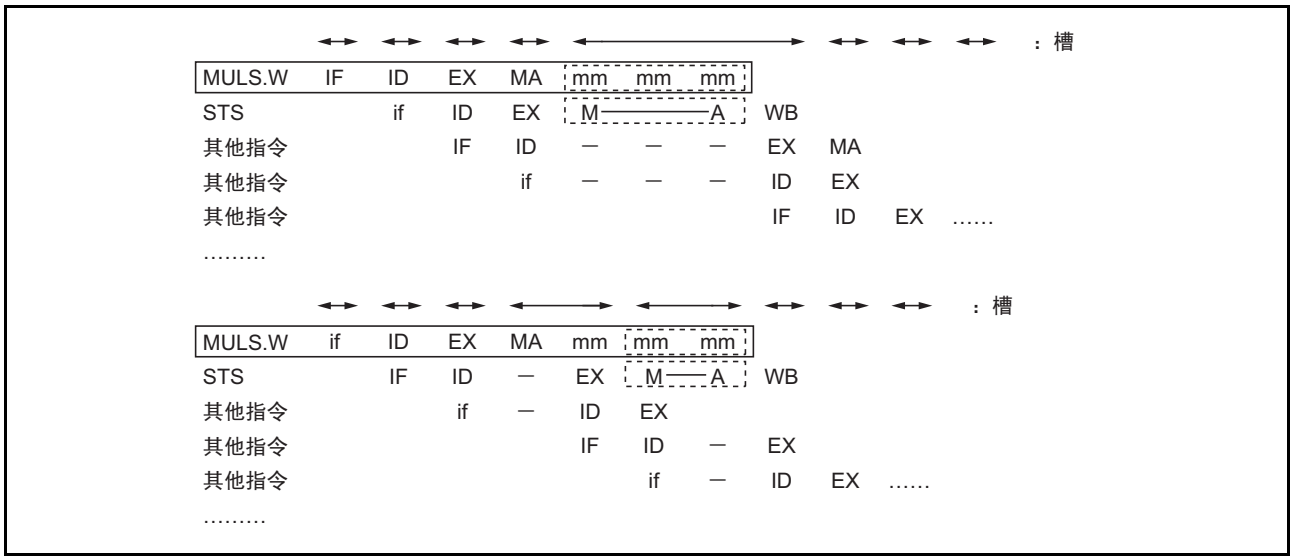


MULS.W 指令的 MA 延长至 mm 结束时，如果该 MA 与 IF 竞争，则照常拆分槽。详细内容参照下图，该图考虑了 MA 与 IF 的竞争。



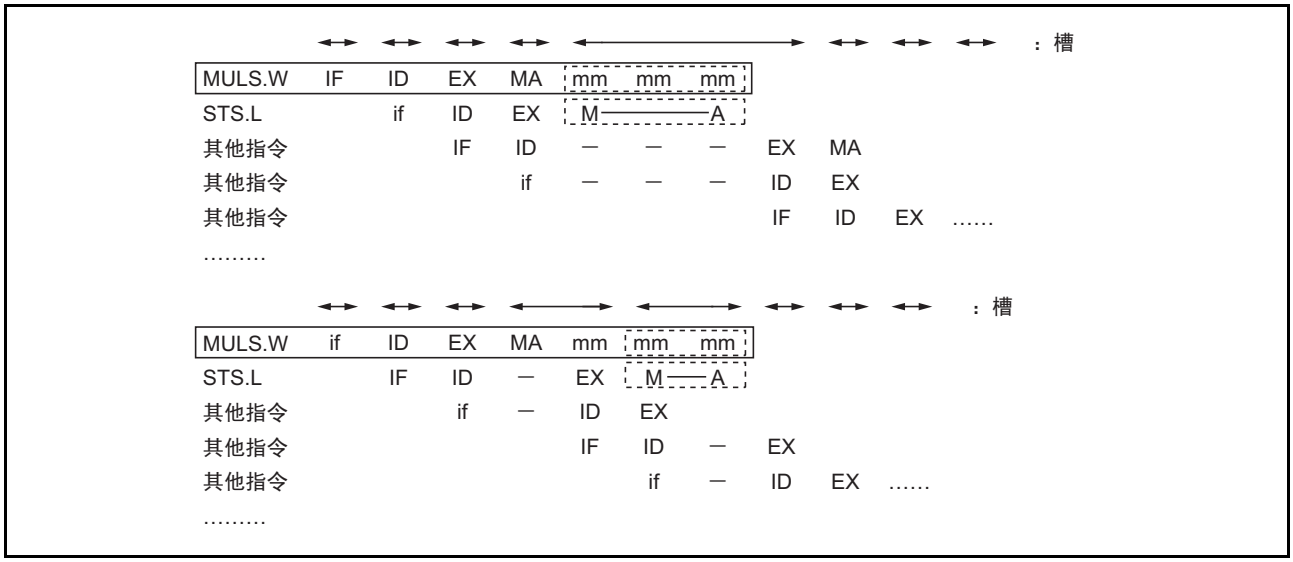
3. MULS.W指令后紧接着出现STS （寄存器）指令时

通过STS指令将MAC寄存器的内容存储至通用寄存器时，如后文所述， STS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。

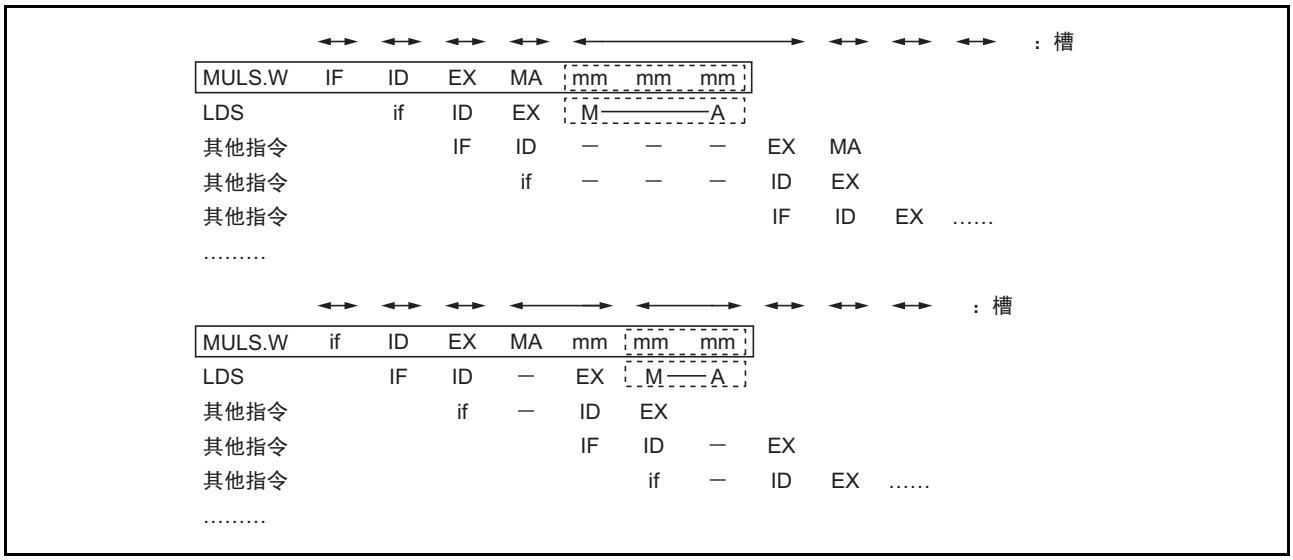


4. MULS.W指令后紧接着出现STS.L （存储器）指令时

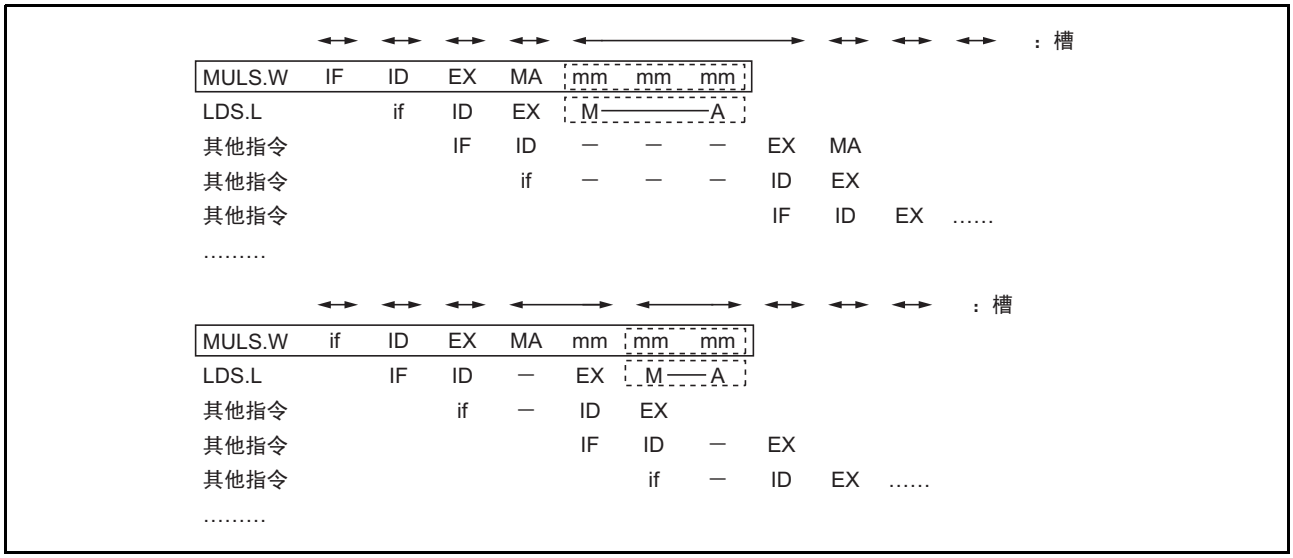
通过STS指令将MAC寄存器的内容存储至存储器时，如后文所述， STS指令进入乘法器存取及存储器写入的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束后经过1个状态 （下图的M—— A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



5. MULS.W指令后紧接着出现LDS （寄存器）指令时
通过LDS指令从通用寄存器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



6. MULS.W指令后紧接着出现LDS.L （存储器）指令时
通过LDS指令从存储器加载MAC寄存器的内容时，如后文所述， LDS指令进入存储器存取及乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束（下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。

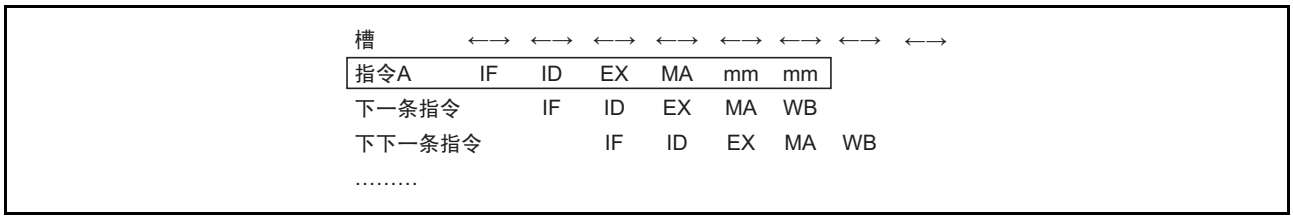


(6) 乘法指令 （SH-2、 SH-DSP）

- 指令种类

MULS.W Rm,Rn
MULU.W Rm,Rn

- 流水线



- 运行说明

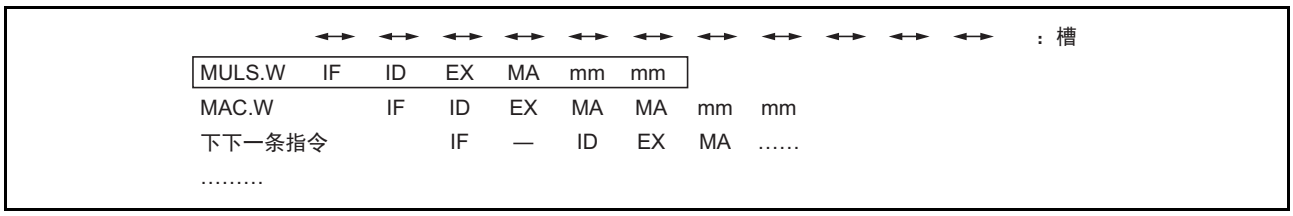
流水线完成 IF、ID、EX、MA、mm、mm 等 6 个阶段后结束。MA 进行乘法器存取。mm 表示乘法器正在运行的状态， MA 结束后与槽无关，在 2 个状态执行 mm 运行。 MULS.W 指令的 MA 也与 IF 竞争时，如 **“7.2.1 取指令 （IF）与存储器存取 （MA）的竞争”**所述，拆分槽。

MULS.W 指令后出现不使用乘法器的指令时， MULS.W 指令可视为 IF、 ID、 EX、 MA 等 4 段流水线指令。即，此时与通常的流水线运行相同。

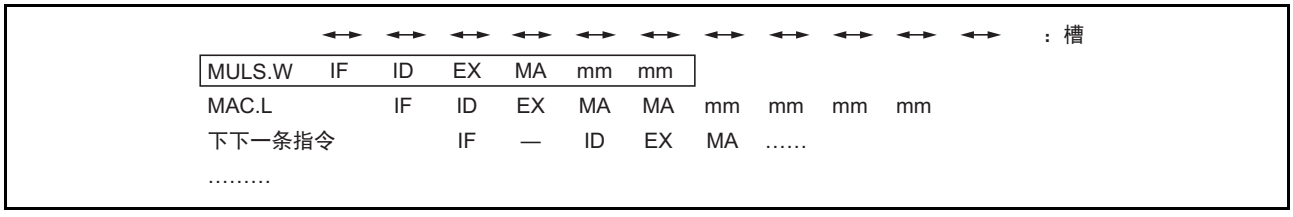
但， MULS.W 指令后出现使用乘法器的指令时，产生乘法器竞争，因此与通常的运行不同。此时有以下几种情况：

1. MULS.W指令后紧接着连续出现MAC.W指令时
2. MULS.W指令后紧接着连续出现MAC.L指令时
3. MULS.W指令后紧接着连续出现MULS.W指令时
4. MULS.W指令后紧接着连续出现DMULS.L指令时
5. MULS.W指令后紧接着出现STS （寄存器）指令时
6. MULS.W指令后紧接着出现STS.L （存储器）指令时
7. MULS.W指令后紧接着出现LDS （寄存器）指令时
8. MULS.W指令后紧接着出现LDS.L （存储器）指令时

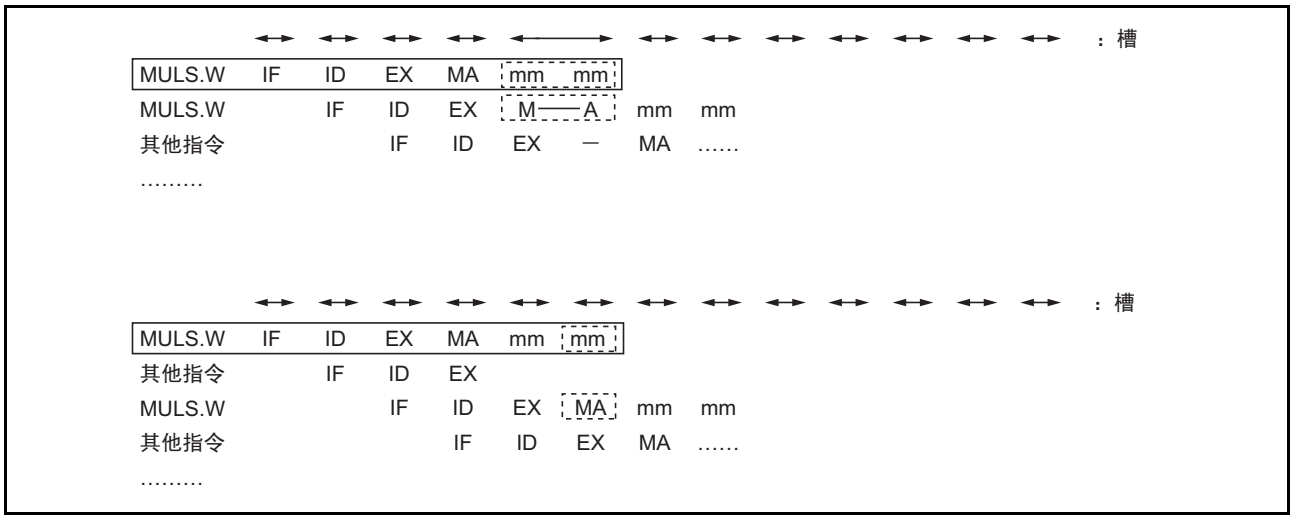
1. MULS.W指令后紧接着连续出现MAC.W指令时
   MAC.W指令的第2个MA不与之前乘法类指令产生的mm竞争。



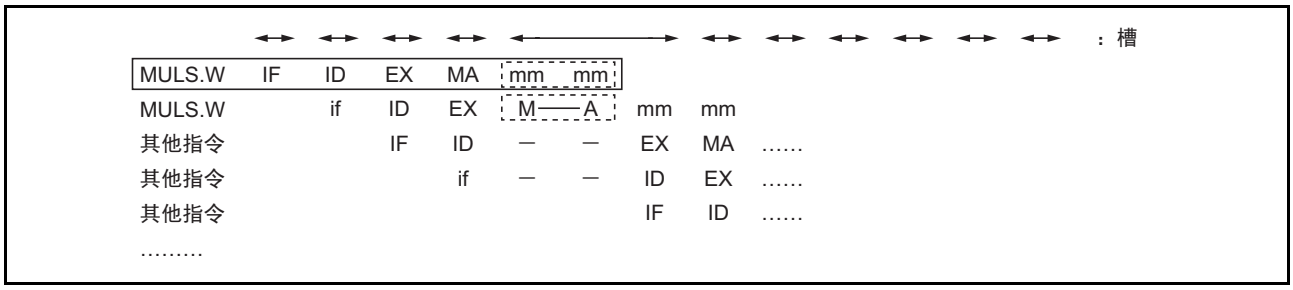
2. MULS.W指令后紧接着连续出现MAC.L指令时
   MAC.W指令的第2个MA不与之前乘法类指令产生的mm竞争。



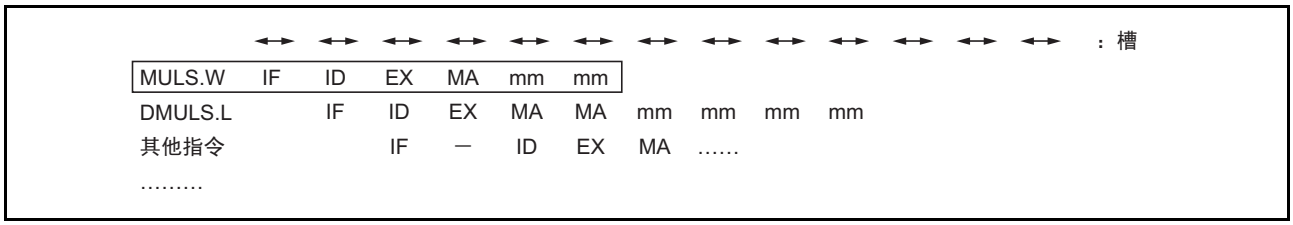
3. MULS.W指令后紧接着连续出现MULS.W指令时
MULS.W指令有乘法器存取的MA阶段。 MULS.W指令的乘法器运行过程中 （mm），如果其他MULS.W的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在MULS.W与MULS.W之间至少插入1条与乘法器无关的指令，则不会出现因MULS.W与MULS.W竞争而产生的停止。如果MULS.W的MA也与IF竞争，则拆分槽。



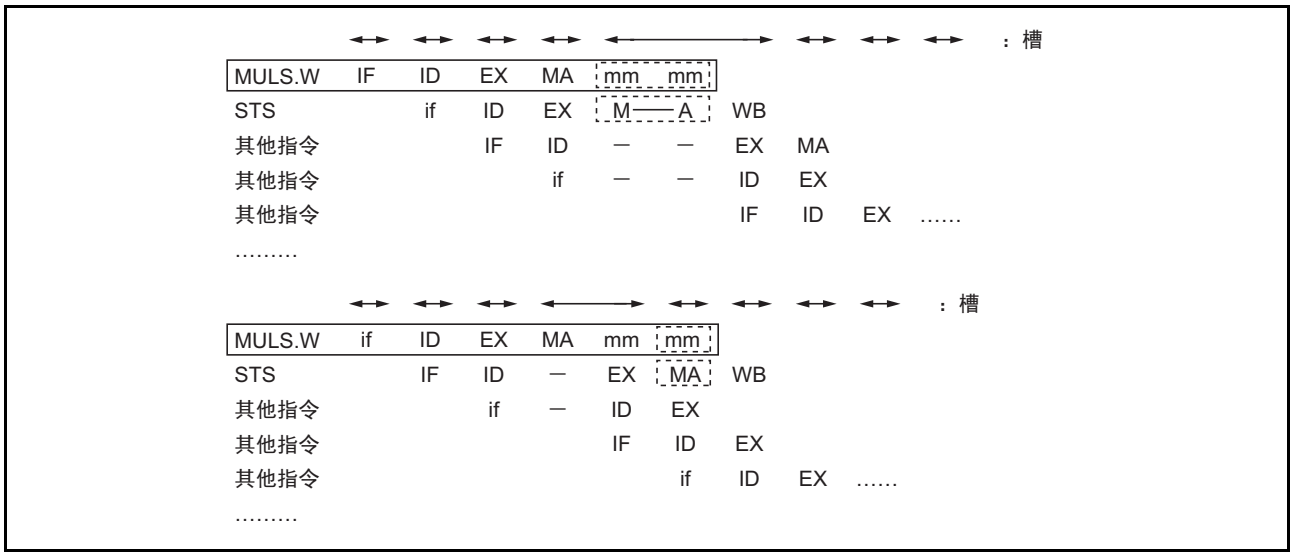
延长MULS.W指令的MA直至mm结束时，如果该MA与IF竞争，则照常拆分槽。详细内容参照下图，该图考虑了MA与IF的竞争。



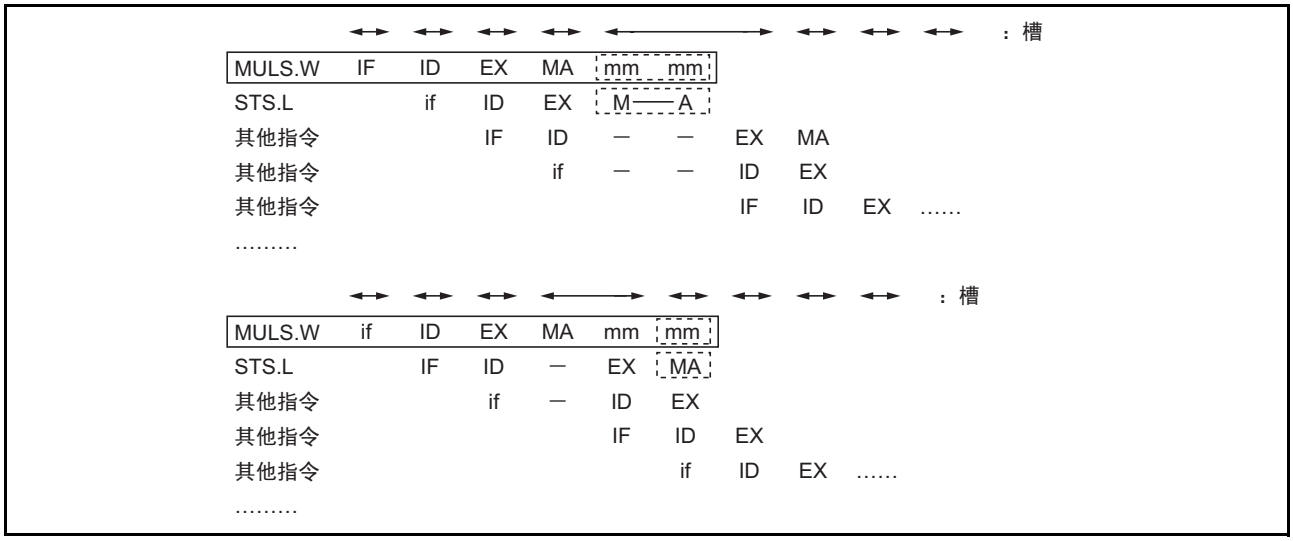
4. MULS.W指令后紧接着连续出现DMULS.L指令时
DMULS.L指令的第2个MA进行乘法器存取，但不与MULS.W指令产生的mm竞争。



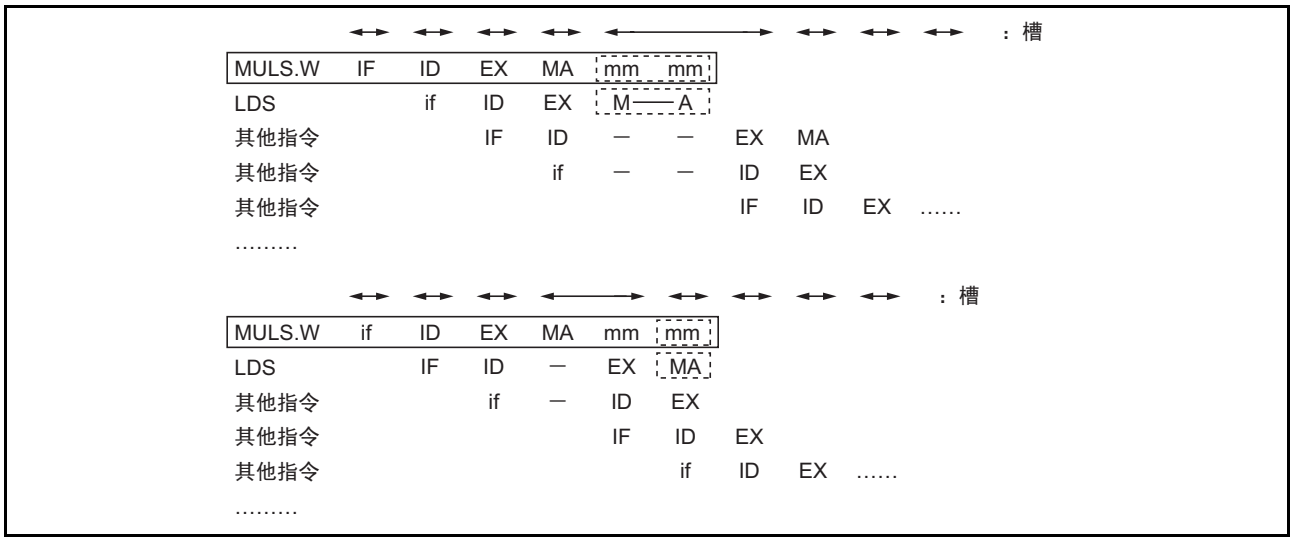
5. MULS.W指令后紧接着出现STS （寄存器）指令时
通过STS指令将MAC寄存器的内容存储至通用寄存器时，如后文所述， STS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



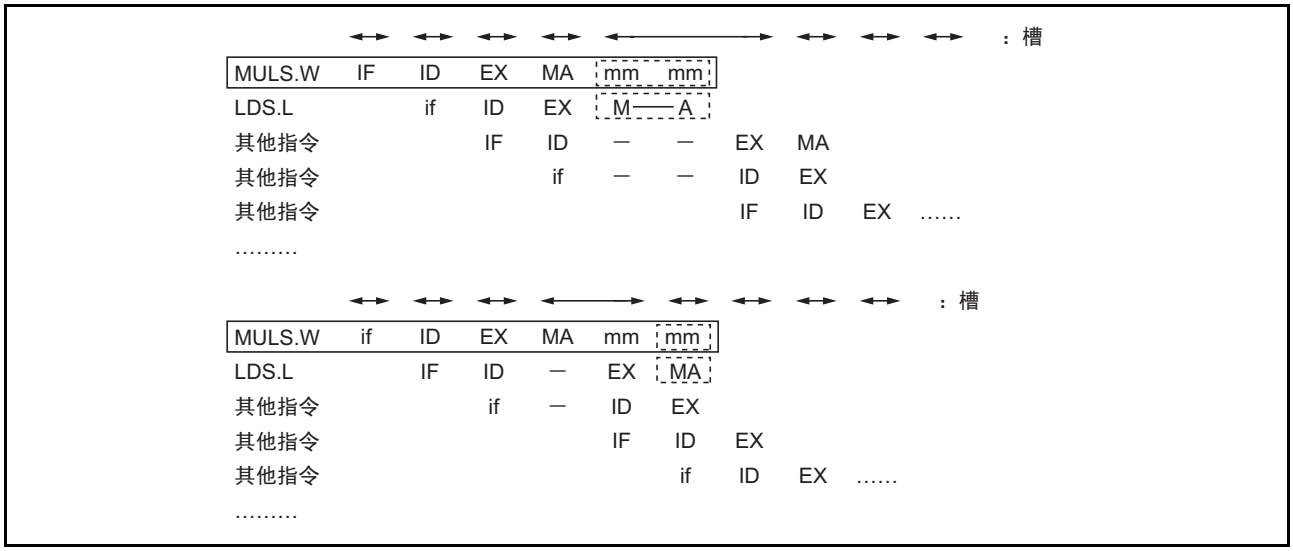
6. MULS.W指令后紧接着出现STS.L （存储器）指令时
通过STS指令将MAC寄存器的内容存储至存储器时，如后文所述， STS指令进入乘法器存取及存储器写入的MA阶段，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



7. MULS.W指令后紧接着出现LDS （寄存器）指令时
通过LDS指令从通用寄存器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



8. MULS.W指令后紧接着出现LDS.L （存储器）指令时
通过LDS指令从存储器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



(7) 双精度乘法指令 （SH-2、 SH-DSP）

- 指令种类

DMULS.L Rm,Rn
DMULU.L Rm,Rn
MUL.L Rm,Rn

- 流水线

槽	←→	←→	←→	←→	←→	←→	←→	←→	←→
指令A	IF	ID	EX	MA	MA	mm	mm	mm	mm
下一条指令		IF	—	ID	EX	MA	WB		
下下一条指令				IF	ID	EX	MA	WB	
……									

- 运行说明

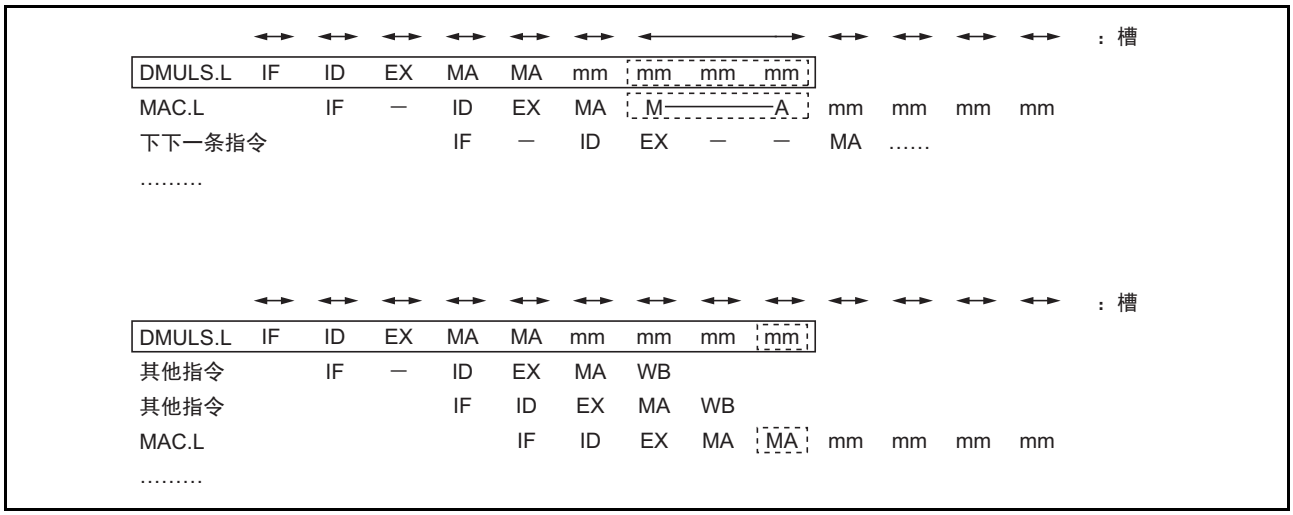
流水线完成 IF、ID、EX、MA、MA、mm、mm、mm、mm 等 9 个阶段后结束。第 2 个 MA 进行乘法器存取。 mm 表示乘法器正在运行的状态， MA 结束后与槽无关，在 4 个状态执行 mm 运行。 DMULS.L 指令的下一条指令的 ID 停止 1 个槽的期间（参考乘加指令）。DMULS.L 指令的 2 个 MA 均与 IF 竞争时，如“**7.2.1 取指令 （IF）与存储器存取 （MA）的竞争**”所述，拆分槽。

DMULS.L 指令后出现不使用乘法器的指令时， DMULS.L 指令可视为 IF、 ID、 EX、 MA、 MA 等 5 段流水线指令。即，此时与通常的流水线运行相同。

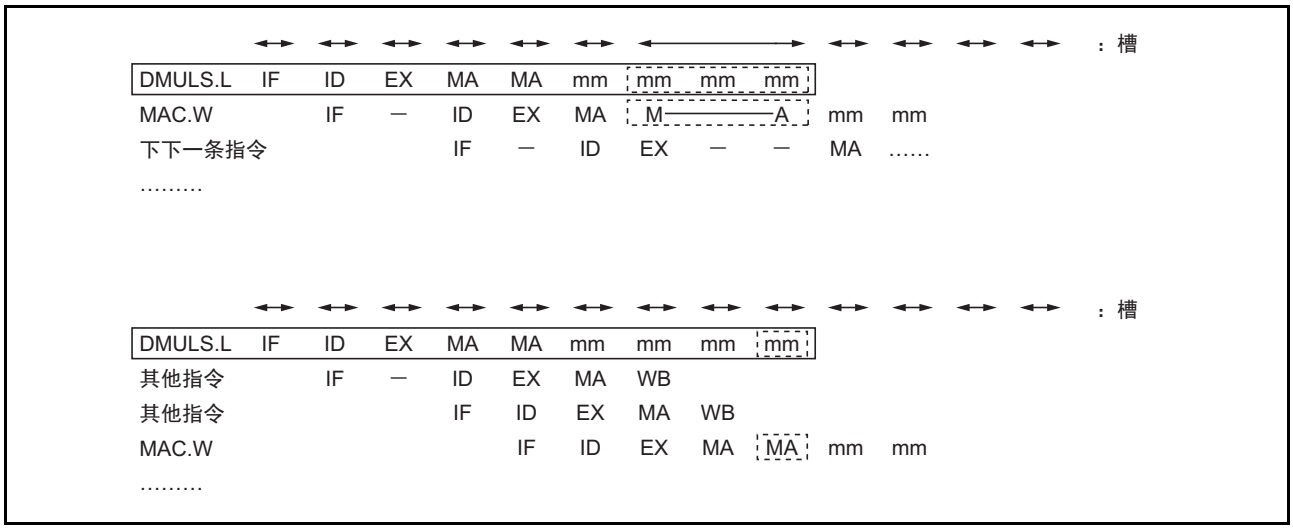
但， DMULS.L 指令后出现使用乘法器的指令时，产生乘法器竞争，因此与通常的运行不同。此时有以下几种情况：

1. DMULS.L指令后紧接着连续出现MAC.L指令时
2. DMULS.L指令后紧接着连续出现MAC.W指令时
3. DMULS.L指令后紧接着连续出现DMULS.L指令时
4. DMULS.L指令后紧接着连续出现MULS.W指令时
5. DMULS.L指令后紧接着出现STS （寄存器）指令时
6. DMULS.L指令后紧接着出现STS.L （存储器）指令时
7. DMULS.L指令后紧接着出现LDS （寄存器）指令时
8. DMULS.L指令后紧接着出现LDS.L （存储器）指令时

1. DMULS.L指令后紧接着连续出现MAC.L指令时
   MAC.L指令的第2个MA与之前乘法类指令产生的mm竞争时，延长该MA的总线周期直至mm结束（下图的M—— A），延长的MA为1个槽。
   如果在DMULS.L指令与MAC.L指令之间至少插入2条与乘法器无关的指令，则不会出现因DMULS.L与MAC.L指令之间的乘法器竞争而产生的停止。

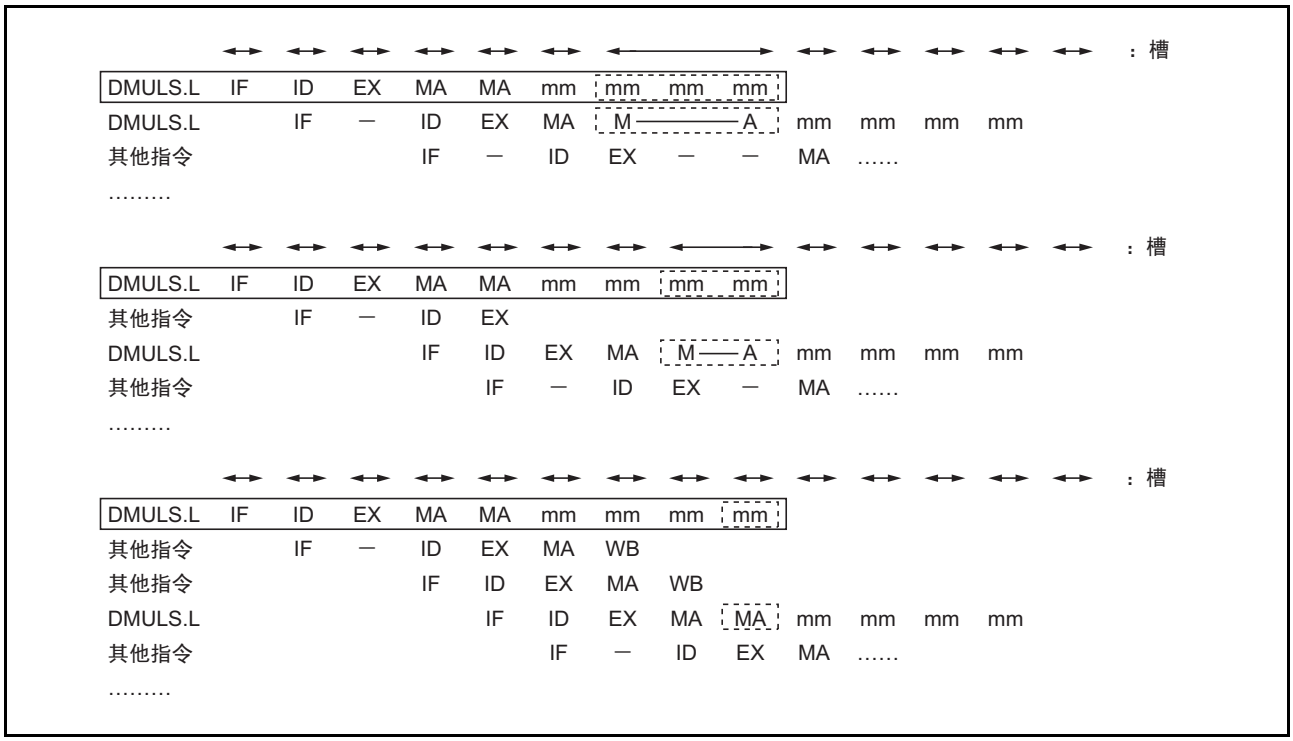


2. DMULS.L指令后紧接着连续出现MAC.W指令时
   MAC.W指令的第2个MA与之前乘法类指令产生的mm竞争时，延长该MA的总线周期直至mm结束（下图的M—— A），延长的MA为1个槽。
   如果在DMULS.L指令与MAC.W指令之间至少插入2条与乘法器无关的指令，则不会出现因DMULS.L与MAC.W指令之间的乘法器竞争而产生的停止。

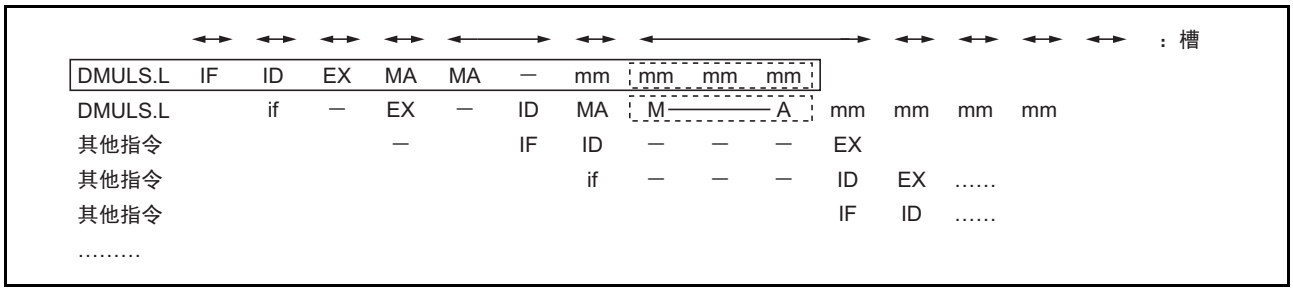


3. DMULS.L指令后紧接着连续出现DMULS.L指令时

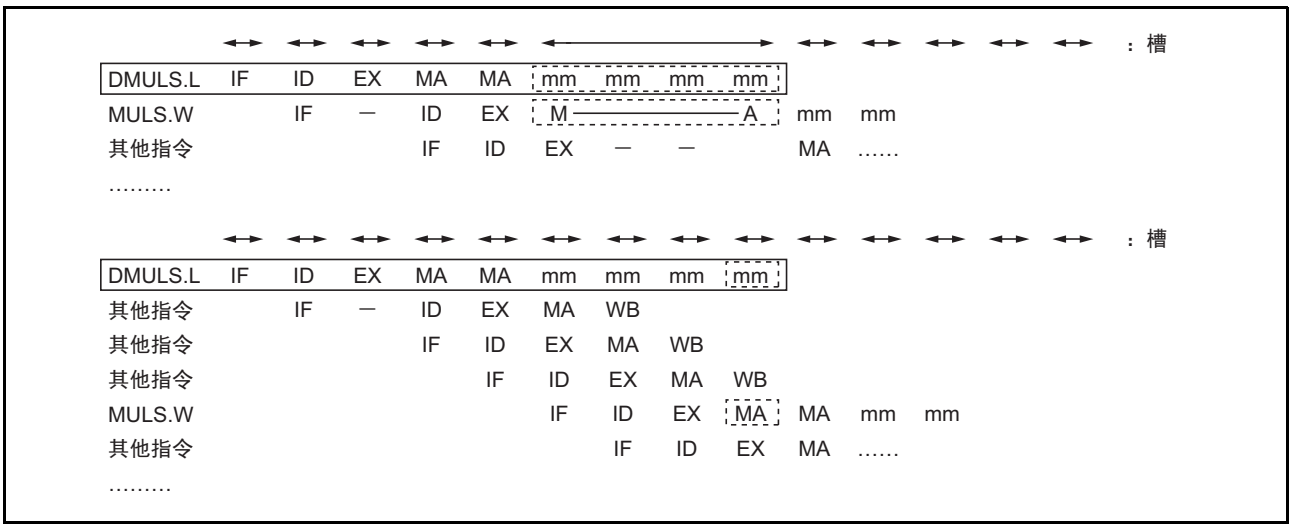
DMULS.L指令有乘法器存取的MA阶段。 DMULS.L指令的乘法器运行过程中 （mm），如果其他DMULS.L的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在DMULS.L与DMULS.L之间至少插入2条与乘法器无关的指令，则不会出现因DMULS.L与DMULS.L竞争而产生的停止。如果DMULS.L的MA也与IF竞争，则拆分槽。



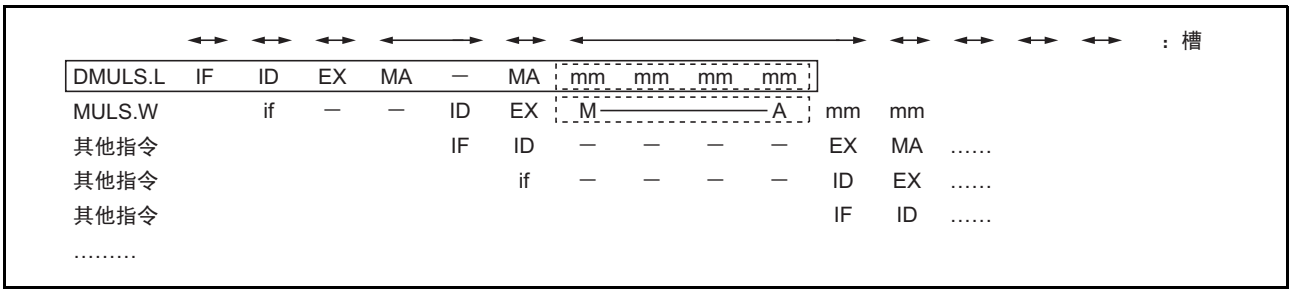
延长DMULS.L指令的MA直至mm结束时，如果该MA与IF竞争，则照常拆分槽。详细内容参照下图，该图考虑了MA与IF的竞争。



4. DMULS.L指令后紧接着连续出现MULS.W指令时
MULS.W指令有乘法器存取的MA阶段。 DMULS.L指令的乘法器运行过程中 （mm），如果其他MULS.W的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽。如果在DMULS.L与MULS.W之间至少插入3条与乘法器无关的指令，则不会出现因DMULS.L与DMULS.L竞争而产生的停止。如果MULS.W的MA也与IF竞争，则拆分槽。

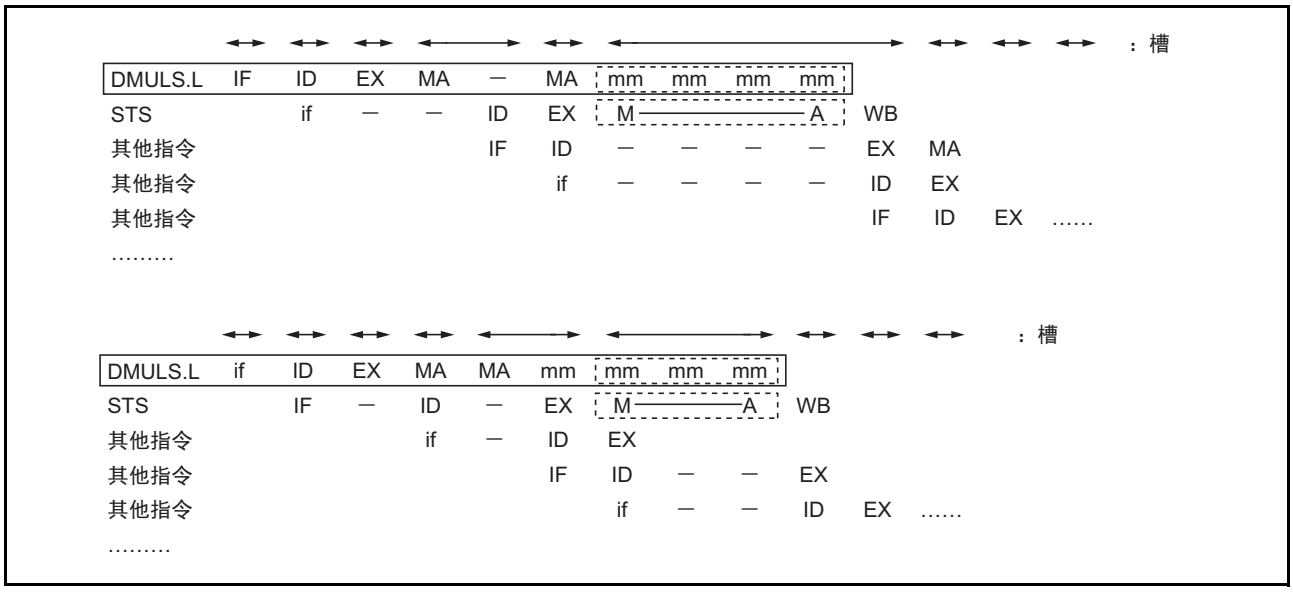


延长DMULS.L指令的MA直至mm结束时，如果该MA与IF竞争，则照常拆分槽。详细内容参照下图，该图考虑了MA与IF的竞争。



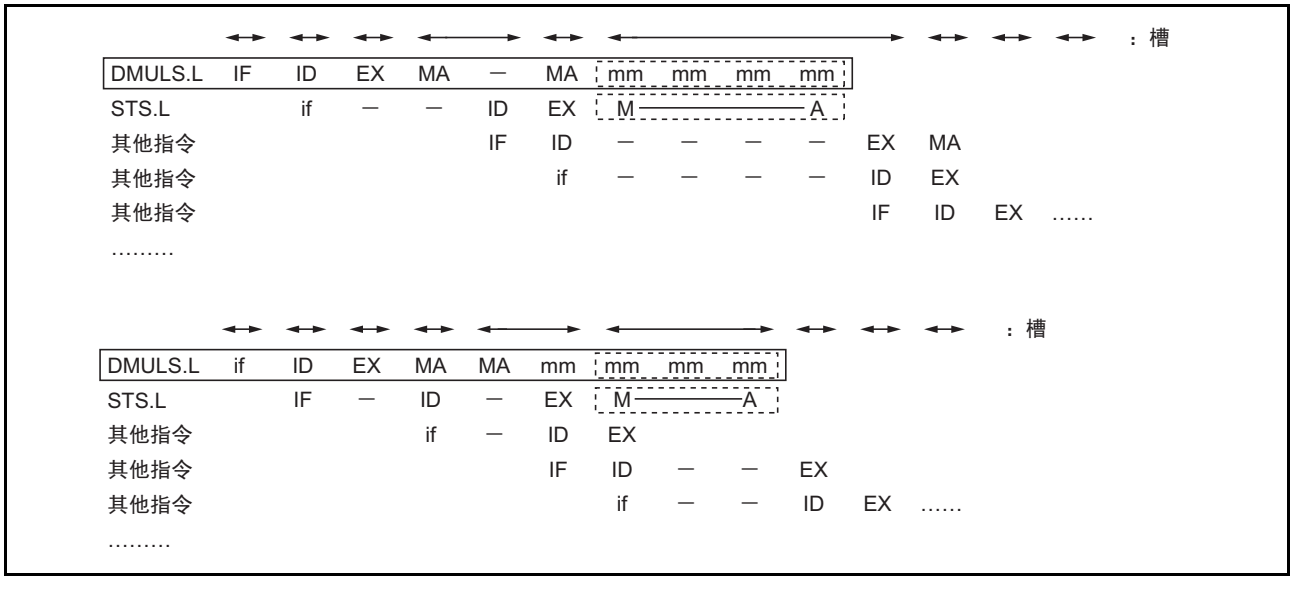
5. DMULS.L指令后紧接着出现STS （寄存器）指令时

通过STS指令将MAC寄存器的内容存储至通用寄存器时，如后文所述， STS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果STS的MA产生竞争，则延长该MA直至mm结束 （下图的M——A），并形成1个槽，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



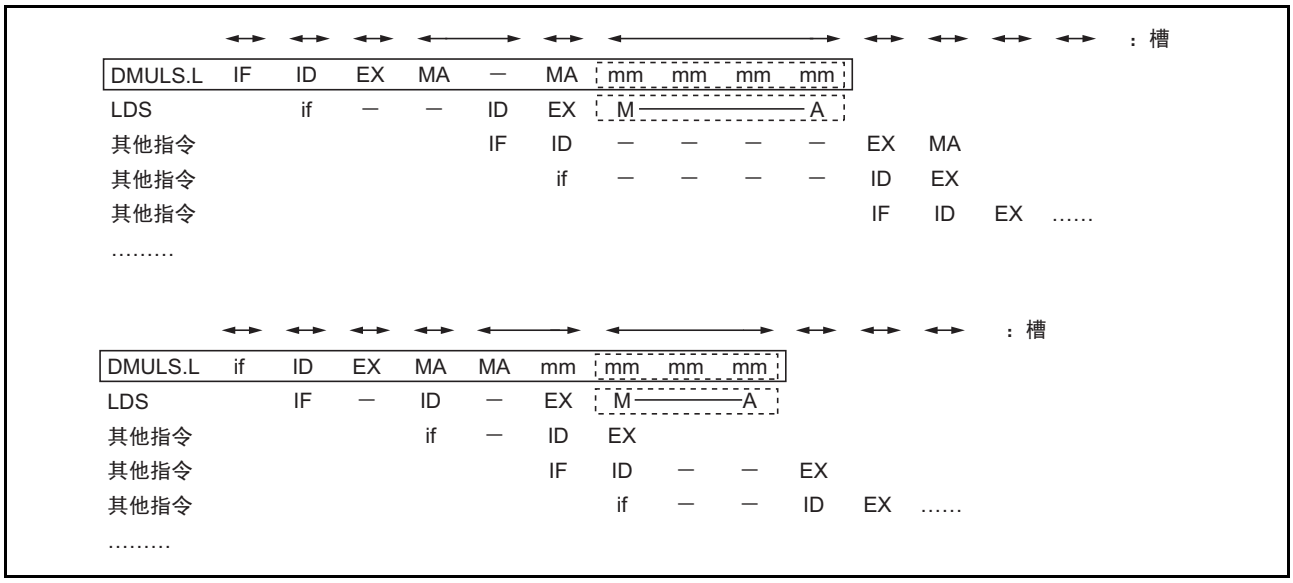
6. DMULS.L指令后紧接着出现STS.L （存储器）指令时

通过STS指令将MAC寄存器的内容存储至存储器时，如后文所述， STS指令进入乘法器存取及存储器写入的MA阶段，且该STS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



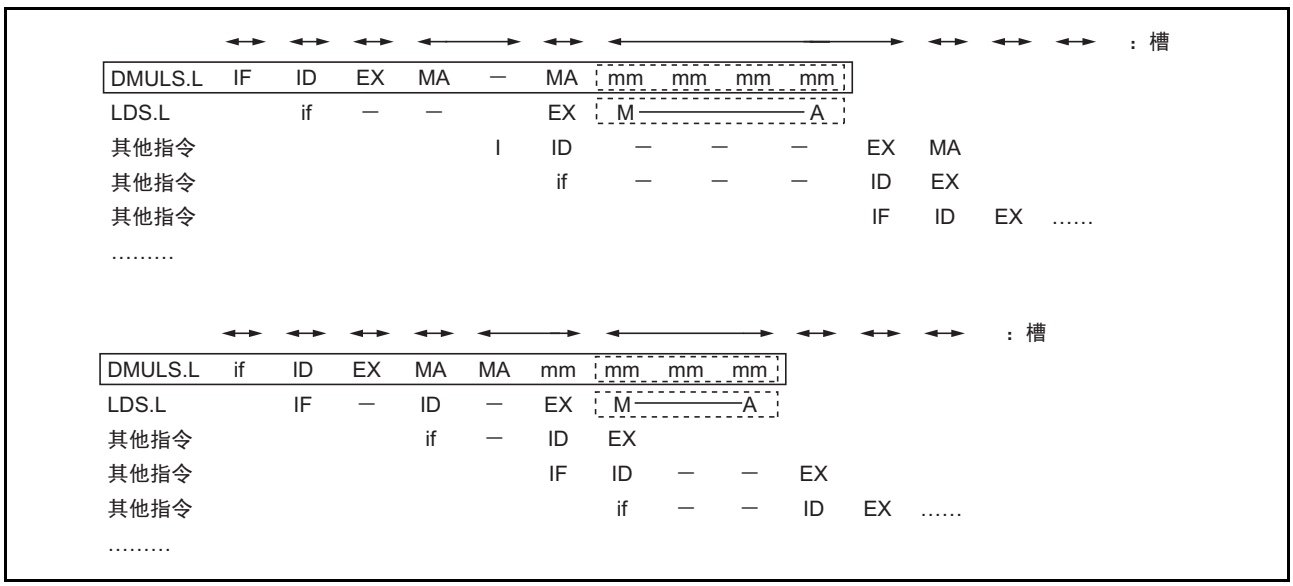
7. DMULS.L指令后紧接着出现LDS （寄存器）指令时

通过LDS指令从通用寄存器加载MAC寄存器的内容时，如后文所述， LDS指令进入乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束 （下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



8. DMULS.L指令后紧接着出现LDS.L （存储器）指令时

通过LDS指令从存储器加载MAC寄存器的内容时，如后文所述， LDS指令进入存储器存取及乘法器存取的MA阶段。乘法器运行过程中 （mm），如果LDS的MA产生竞争，则延长该MA直至mm结束（下图的M—— A），并形成1个槽，且该LDS的MA与IF竞争。详细内容如下图所示，该图考虑了MA与IF的竞争。



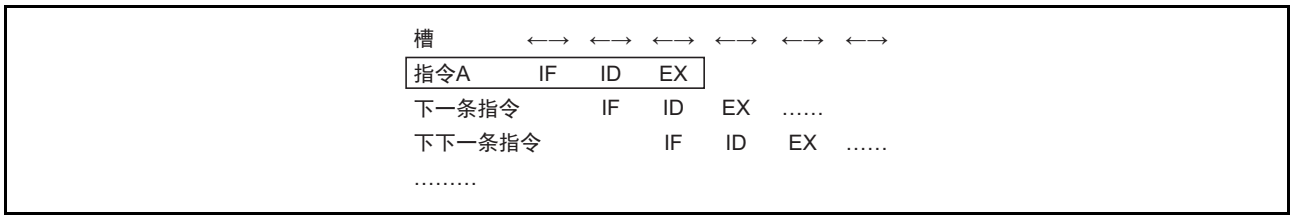
7.4.3 逻辑运算指令

(1) 寄存器－寄存器之间的逻辑运算指令 （通用）

- 指令种类

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		

- 流水线



- 运行说明

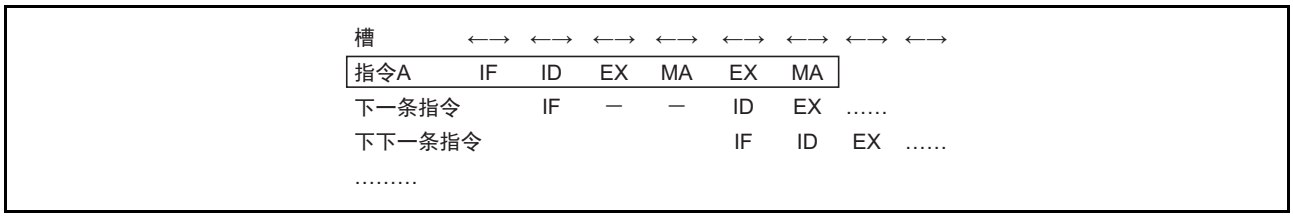
流水线完成 IF、 ID、 EX 等 3 个阶段后结束。在 EX 阶段，通过 ALU 完成数据运算。

(2) 存储器逻辑运算指令 （通用）

- 指令种类

AND.B	#imm,@(R0,GBR)
OR.B	#imm,@(R0,GBR)
TST.B	#imm,@(R0,GBR)
XOR.B	#imm,@(R0,GBR)

- 流水线



- 运行说明

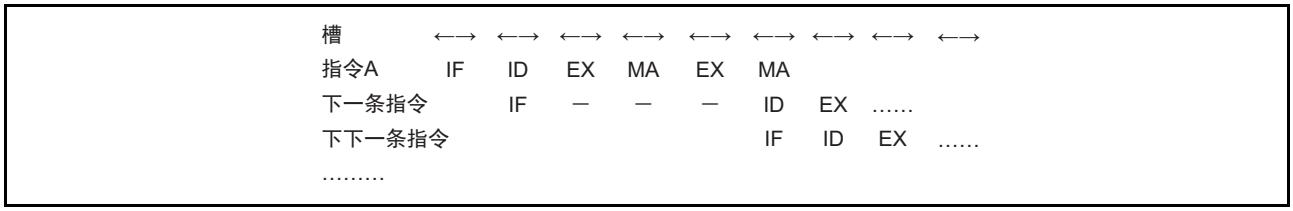
流水线完成 IF、 ID、 EX、 MA、 EX、 MA 等 6 个阶段后结束。下一条指令的 ID 停止 2 个槽的期间。当然，此指令的 MA 与 IF 竞争。

(3) TAS 指令 （通用）

- 指令种类

TAS.B @Rn

- 流水线



- 运行说明

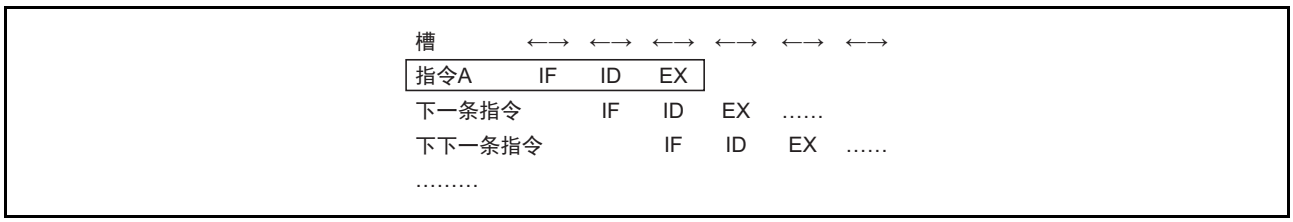
流水线完成 IF、 ID、 EX、 MA、 EX、 MA 等 6 个阶段后结束。下一条指令的 ID 停止 3 个槽的期间。当然， TAS 指令的 MA 与 IF 竞争。

7.4.4 移位指令

- 指令种类

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		

- 流水线



- 运行说明

流水线完成 IF、 ID、 EX 等 3 个阶段后结束。在 EX 阶段，通过 ALU 完成数据运算。

7.4.5 转移指令

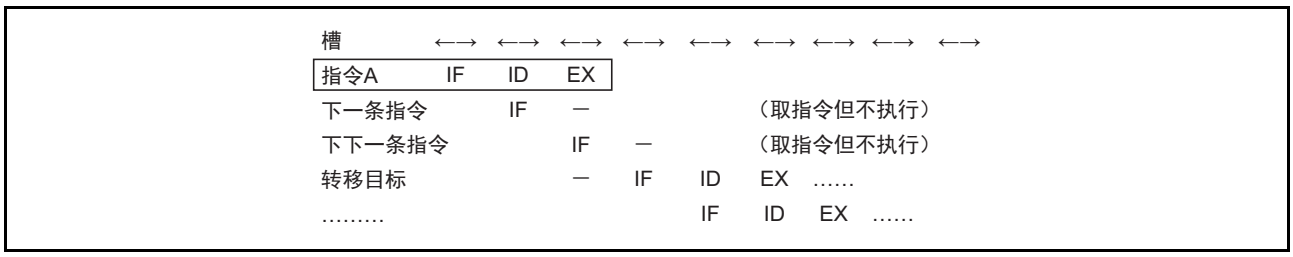
(1) 条件转移指令 （通用）

- 指令种类

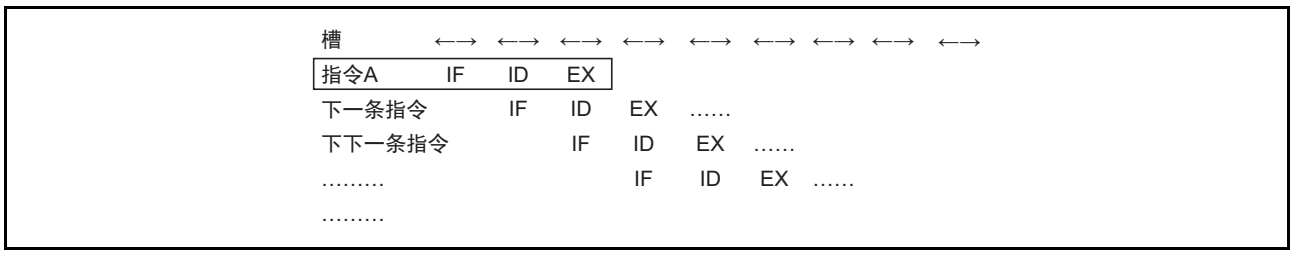
```
BF      label
BT      label
```

- 流水线

1. 条件成立时



2. 条件不成立时



- 运行说明

流水线完成 IF、ID、EX 等 3 个阶段后结束。在 ID 阶段进行条件判断。条件转移指令不是延迟转移指令。

1. 条件成立时
   在EX阶段计算转移目标地址。取条件转移指令 （指令A）的下一条及下下一条指令，但不执行。从指令A的EX阶段所在槽的下一个槽开始取转移目标指令。
2. 条件不成立时
   在ID阶段判断条件不成立后，在EX阶段不执行任何运行，而继续取下一条指令并执行。

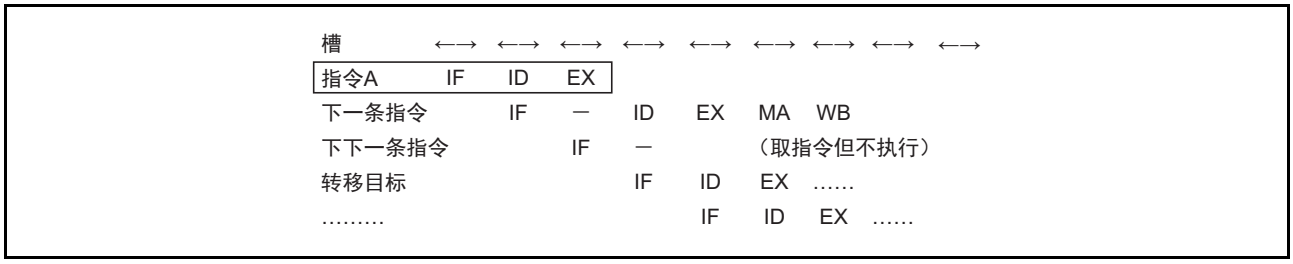
(2) 带延迟的条件转移指令 （SH-2、 SH-DSP）

- 指令种类

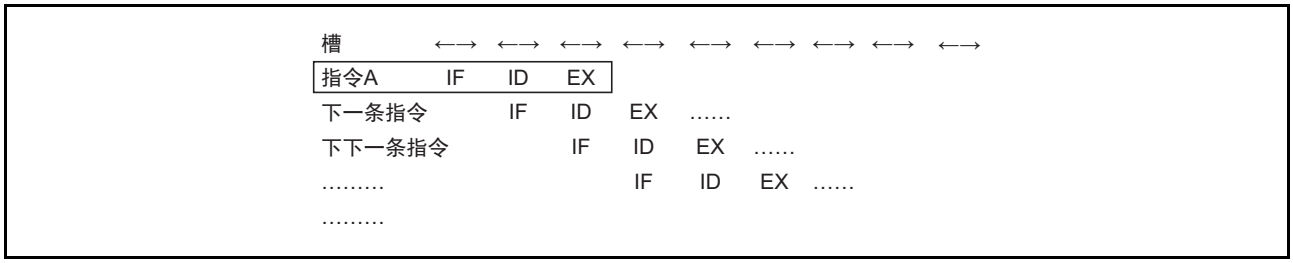
```
BF/S    label
BT/S    label
```

- 流水线

1. 条件成立时



2. 条件不成立时



- 运行说明

流水线完成 IF、 ID、 EX 等 3 个阶段后结束。在 ID 阶段进行条件判断。

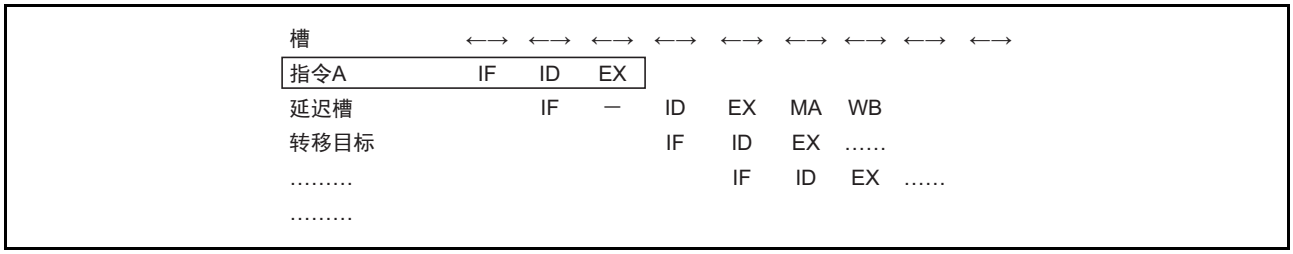
1. 条件成立时
   在EX阶段计算转移目标地址。取条件转移指令 （指令A）的下一条指令并执行，即使取下下一条指令也不执行。从指令A的EX阶段所在槽的下一个槽开始取转移目标指令。
2. 条件不成立时
   在ID阶段判断条件不成立后，在EX阶段不执行任何运行，而继续取下一条指令并执行。

(3) 无条件转移指令 （通用或 SH-2、 SH-DSP）

- 指令种类

BRA	label	JMP	@Rm
BRAF	Rm （SH-2、 SH-DSP）	JSR	@Rm
BSR	label	RTS	
BSRF	Rm （SH-2、 SH-DSP ）		

- 流水线



- 运行说明

流水线完成 IF、 ID、 EX 等 3 个阶段后结束。无条件转移指令为延迟转移指令。

在 EX 阶段计算转移目标地址。取无条件转移指令 （指令 A）的下一条指令即延迟槽指令后执行，而不能像条件转移指令那样取后不执行。但，该延迟槽指令的 ID 阶段停止 1 个槽的期间。从指令 A 的 EX 阶段所在槽的下一个槽开始取转移目标指令。

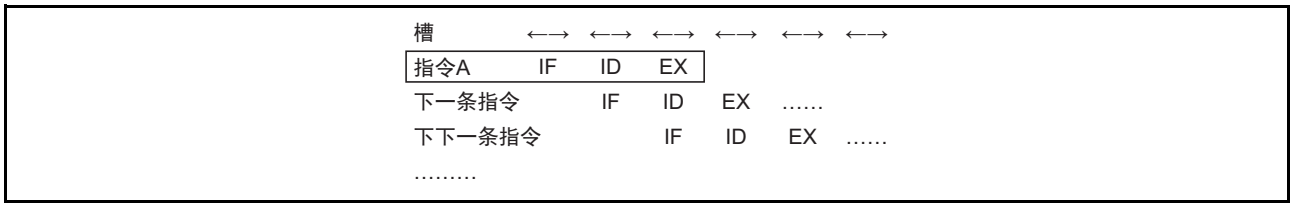
7.4.6 系统控制指令

(1) 系统控制 ALU 指令 （通用或 SH-DSP）

- 指令种类

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）		

- 流水线



- 运行说明

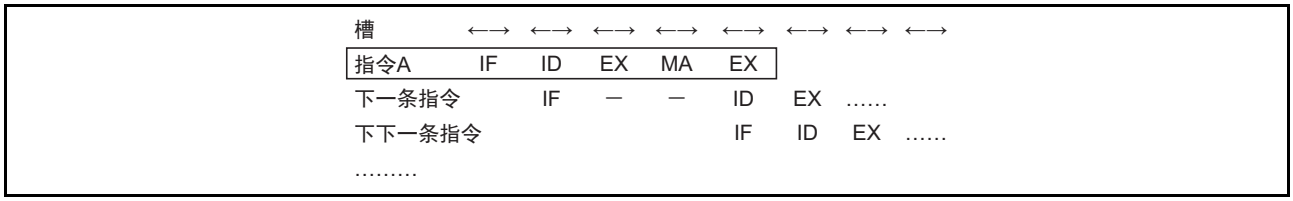
流水线完成 IF、 ID、 EX 等 3 个阶段后结束。在 EX 阶段，通过 ALU 完成数据运算。

(2) LDC.L 指令 （通用或 SH-DSP）

- 指令种类

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）

- 流水线



- 运行说明

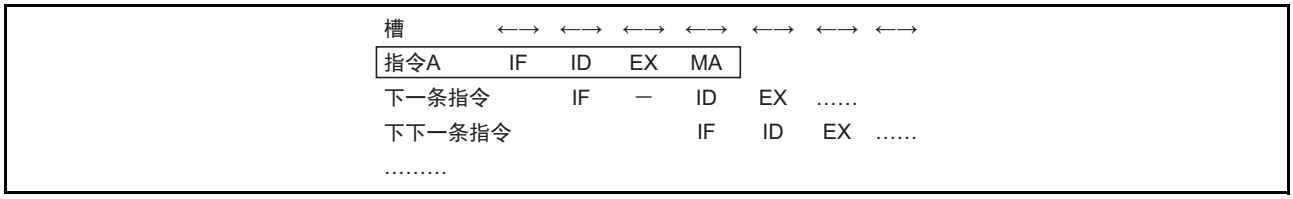
流水线完成 IF、 ID、 EX、 MA、 EX 等 5 个阶段后结束。下一条指令的 ID 停止 2 个槽的期间。

(3) STC.L 指令 （通用或 SH-DSP）

- 指令种类

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）

- 流水线



- 运行说明

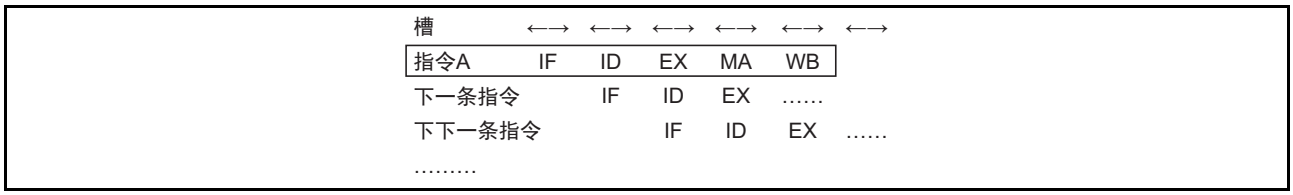
流水线完成 IF、 ID、 EX、 MA 等 4 个阶段后结束。下一条指令的 ID 停止 1 个槽的期间。

(4) LDS.L 指令 （通用）

- 指令种类

LDS.L @Rm+,PR

- 流水线



- 运行说明

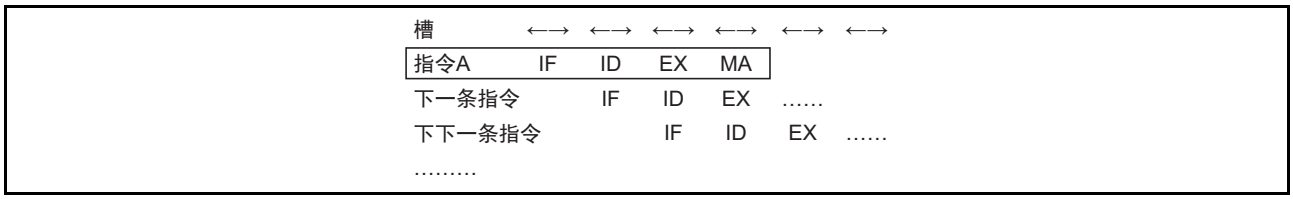
流水线完成 IF、 ID、 EX、 MA、 WB 等 5 个阶段后结束，与通常的加载指令相同。

(5) STS.L 指令 （通用）

- 指令种类

STS.L PR,@-Rn

- 流水线



- 运行说明

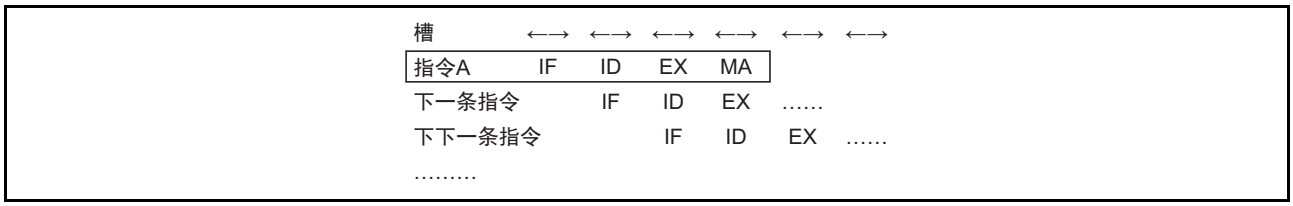
流水线完成 IF、 ID、 EX、 MA 等 4 个阶段后结束，与通常的存储指令相同。

(6) 寄存器 →MAC、 DSP 传送指令 （通用或 SH-DSP）

- 指令种类

```
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）
```

- 流水线



- 运行说明

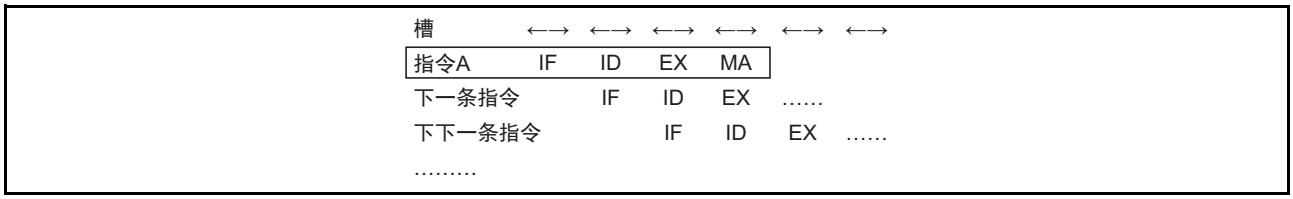
流水线完成 IF、 ID、 EX、 MA 等 4 个阶段后结束。 MA 为乘法器存取的阶段，且该 MA 与 IF 产生竞争。即，与通常的存储指令相同，但会引起与乘法器的竞争，因此请参照乘法指令、乘加指令、双精度乘法指令及双精度乘加指令的内容。

(7) 存储器 →MAC、 DSP 传送指令 （通用或 SH-DSP）

- 指令种类

```
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）
```

- 流水线



- 运行说明

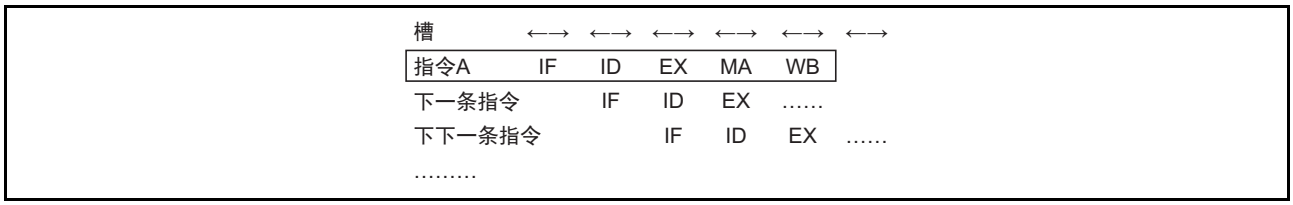
流水线完成 IF、 ID、 EX、 MA 等 4 个阶段后结束。 MA 为存储器存取及乘法器存取的阶段，且该 MA 与 IF 产生竞争。即，与通常的加载指令相同，但会引起与乘法器的竞争，因此请参照乘法指令、乘加指令、双精度乘法指令及双精度乘加指令的内容。

(8) MAC、 DSP→ 寄存器传送指令 （通用或 SH-DSP）

- 指令种类

```
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）
```

- 流水线



- 运行说明

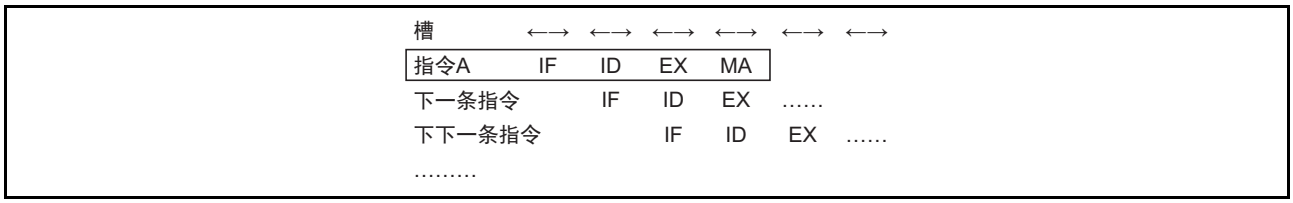
流水线完成 IF、 ID、 EX、 MA、 WB 等 5 个阶段后结束。 MA 为乘法器存取的阶段，且该 MA 与 IF 产生竞争。即，与通常的加载指令相同，但会引起与乘法器的竞争，因此请参照乘法指令、乘加指令、双精度乘法指令及双精度乘加指令的内容。

(9) MAC、 DSP→ 存储器传送指令 （通用或 SH-DSP）

- 指令种类

```
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）
```

- 流水线



- 运行说明

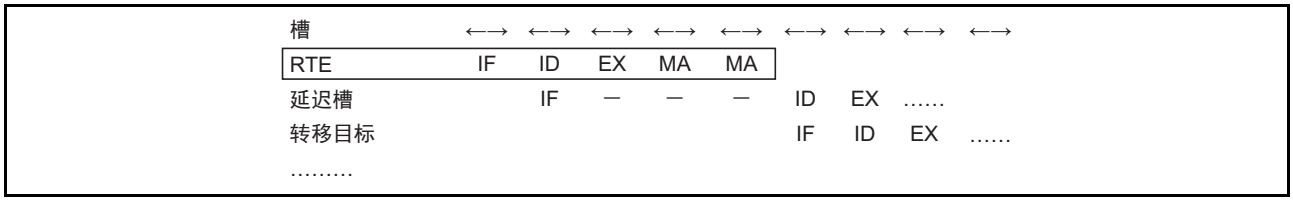
流水线完成 IF、 ID、 EX、 MA 等 4 个阶段后结束。 MA 为存储器存取及乘法器存取的阶段，且该 MA 与 IF 产生竞争。即，与通常的存储指令相同，但会引起与乘法器的竞争，因此请参照乘法指令、乘加指令、双精度乘法指令及双精度乘加指令的内容。

(10) RTE 指令 （通用）

- 指令种类

RTE

- 流水线



- 运行说明

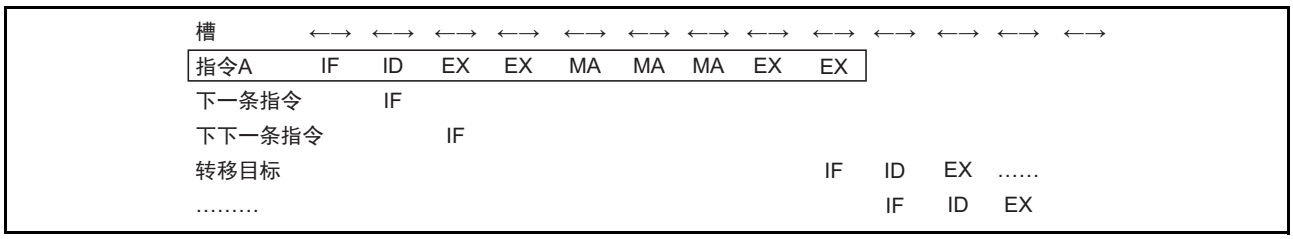
流水线完成 IF、 ID、 EX、 MA、 MA 等 5 个阶段后结束。此 MA 不与 IF 产生竞争。 RTE 为延迟转移指令，延迟槽指令的 ID 停止 3 个槽的期间。从 RTE 中 MA 的下一个槽开始执行转移目标指令的 IF。

(11) TRAP 指令 （通用）

- 指令种类

TRAPA #imm

- 流水线



- 运行说明

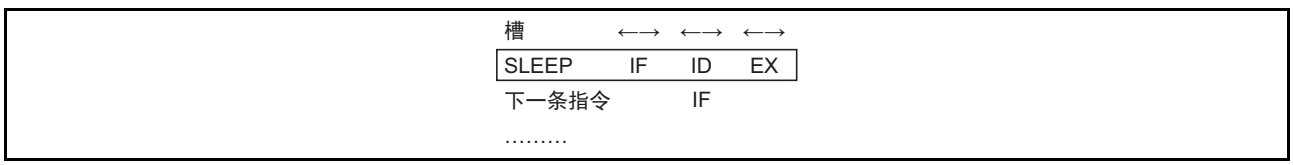
流水线完成 IF、ID、EX、EX、MA、MA、MA、EX、EX 等 9 个阶段后结束。此 MA 不与 IF 产生竞争。 TRAP 指令不是延迟转移指令。虽然取 TRAP 指令的下一条及下下一条指令，但不执行。从 TRAP 指令第 9 个阶段 EX 的所在槽开始执行转移目标指令的 IF。

(12) SLEEP 指令 （通用）

- 指令种类

SLEEP

- 流水线



- 运行说明

流水线完成 IF、 ID、 EX 等 3 个阶段后结束，在执行下一条指令的 IF 前发行。执行 SLEEP 指令后进入睡眠模式或待机模式。

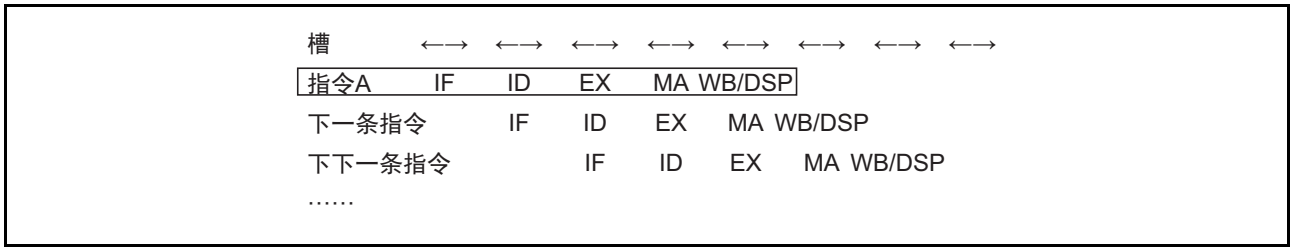
7.4.7 DSP 数据传送指令

(1) X 存储器加载指令 （SH-DSP）

- 指令种类

```
NOPX
MOVX.W    @Ax,Dx
MOVX.W    @Ax+,Dx
MOVX.W    @Ax+Ix,Dx
```

- 流水线



- 运行说明

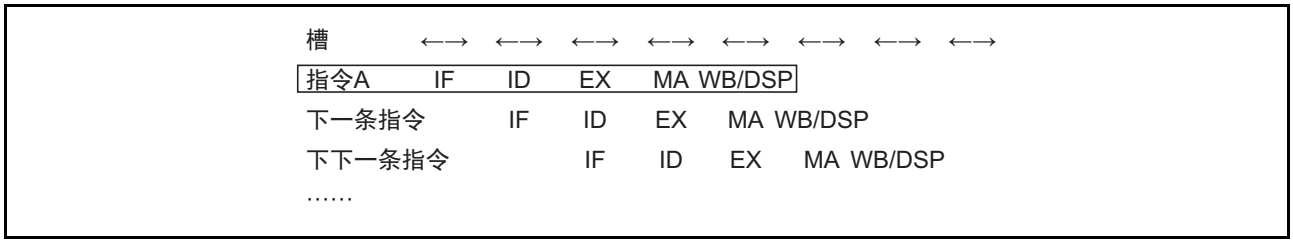
流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。由于该数据传送通过 X 总线执行，因此不与其他指令的 IF 产生竞争。

(2) Y 存储器加载指令 （SH-DSP）

- 指令种类

```
NOPY
MOVY.W    @Ay,Dy
MOVY.W    @Ay+,Dy
MOVY.W    @Ay+Iy,Dy
```

- 流水线



- 运行说明

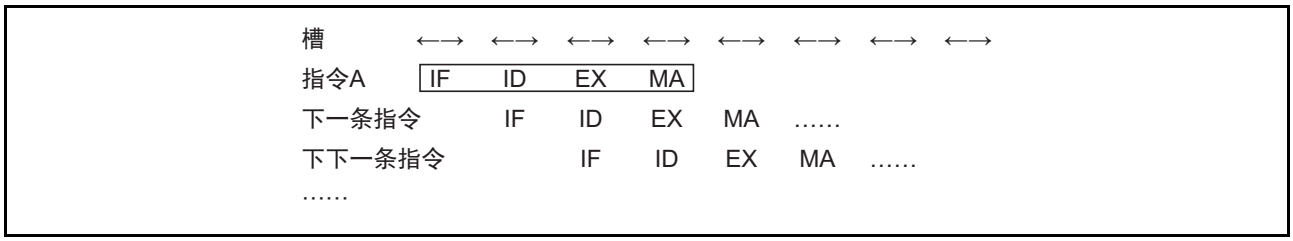
流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。由于该数据传送通过 Y 总线执行，因此不与其他指令的 IF 产生竞争。

(3) X 存储器存储指令 （SH-DSP）

- 指令种类

```
MOVX.W    Da, @Ax
MOVX.W    Da, @Ax+
MOVX.W    Da, @Ax+Ix
```

- 流水线



- 运行说明

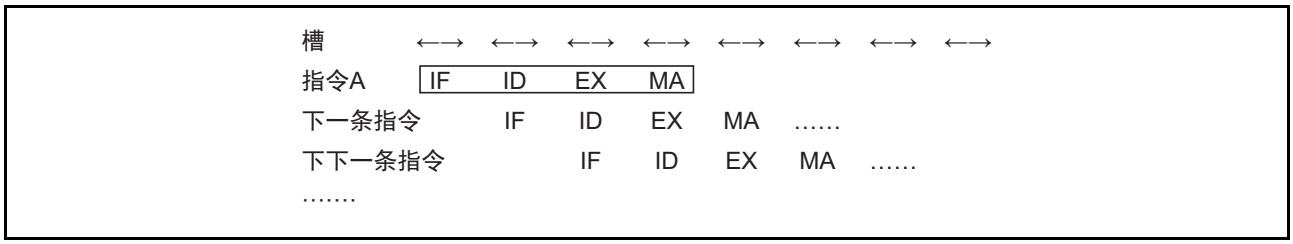
流水线完成 IF、ID、EX、MA 等 4 个阶段后结束。如果要在 DSP 运算指令后紧接着由该指令存储 DSP 运算结果，则产生竞争 （详细内容参照 **“7.2.2 使用先行指令目标寄存器时的竞争”**（2））。

(4) Y 存储器存储指令 （SH-DSP）

- 指令种类

```
MOVY.W    Da, @Ay
MOVY.W    Da, @Ay+
MOVY.W    Da, @Ay+Iy
```

- 流水线



- 运行说明

流水线完成 IF、ID、EX、MA 等 4 个阶段后结束。如果要在 DSP 运算指令后紧接着由该指令存储 DSP 运算结果，则产生竞争 （详细内容参照 **“7.2.2 使用先行指令目标寄存器时的竞争”**（2））。

(5) 单加载指令 （SH-DSP）

- 指令种类

```
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
```

- 流水线

```
槽          ←→  ←→  ←→  ←→  ←→  ←→  ←→  ←→
指令A       IF  ID  EX  MA  WB/DSP
下一条指令      IF  ID  EX  MA  WB/DSP
下下一条指令        IF  ID  EX  MA  WB/DSP
.....
```

- 运行说明

流水线完成 IF、ID、EX、MA、WB/DSP 等 5 个阶段后结束。即使设置使用该指令目标寄存器的指令，也不产生竞争。

(6) 单存储指令 （SH-DSP）

- 指令种类

```
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
```

- 流水线

```
槽          ←→  ←→  ←→  ←→  ←→  ←→  ←→  ←→
指令A       IF  ID  EX  MA
下一条指令      IF  ID  EX  MA  ……
下下一条指令        IF  ID  EX  MA  ……
.....
```

- 运行说明

流水线完成 IF、ID、EX、MA 等 4 个阶段后结束。如果要在 DSP 运算指令后紧接着由该指令存储 DSP 运算结果，则产生竞争 （详细内容参照 **“7.2.2 使用先行指令目标寄存器时的竞争”**（2））。

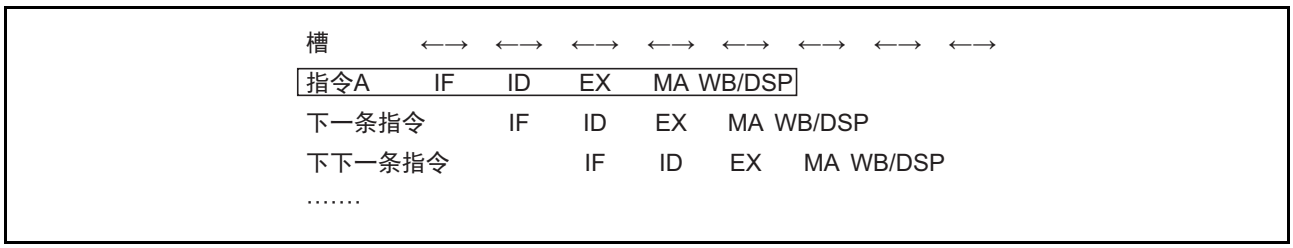
7.4.8 DSP 运算指令

(1) ALU 算术运算指令 （SH-DSP）

- 指令种类

	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		

- 流水线



- 运行说明

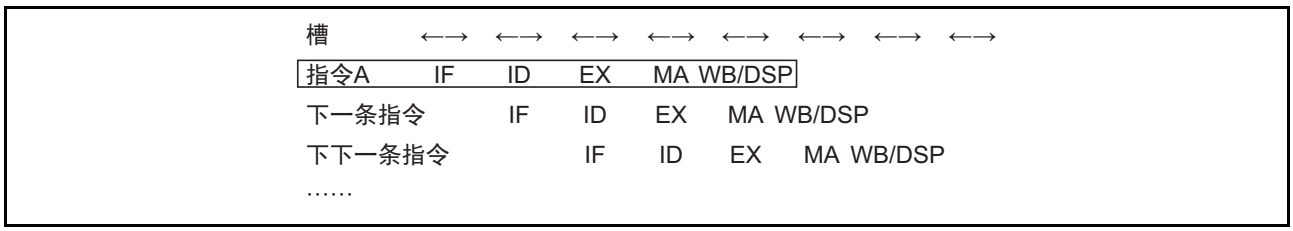
流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。执行带条件的运算指令时，如果条件不成立，则 WB/DSP 阶段为空操作 （NOP），但不改变流水线。

(2) ALU 逻辑运算指令 （SH-DSP）

- 指令种类

```
        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
```

- 流水线



- 运行说明

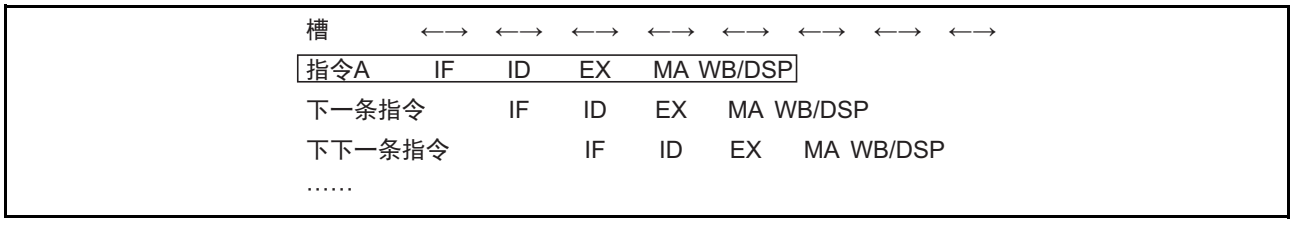
流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。执行带条件的运算指令时，如果条件不成立，则 WB/DSP 阶段为空操作 （NOP），但不改变流水线。

(3) ALU 逻辑运算指令 （SH-DSP）

- 指令种类

```
        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
```

- 流水线



- 运行说明

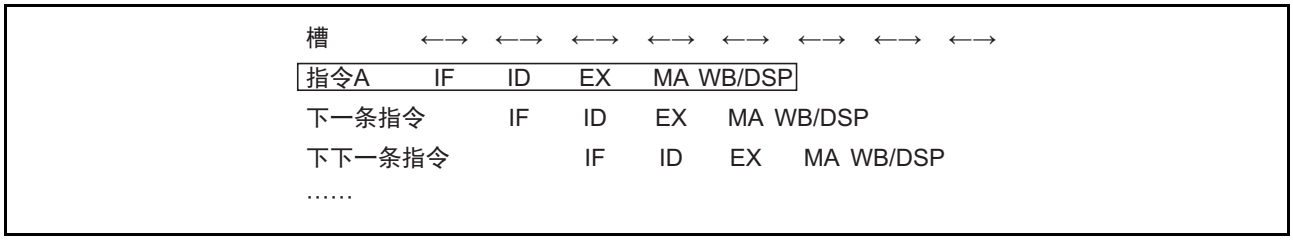
流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。执行带条件的运算指令时，如果条件不成立，则 WB/DSP 阶段为空操作 （NOP），但不改变流水线。

(4) 带符号的乘法指令 （SH-DSP）

- 指令种类

PMULS Se,Sf,Dg

- 流水线



- 运行说明

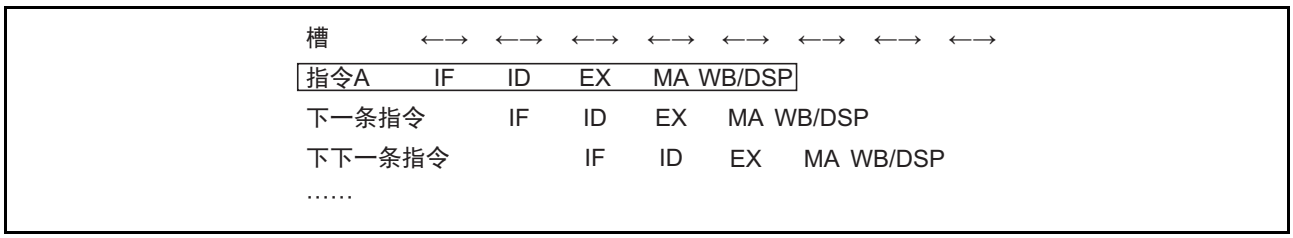
流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。

(5) 寄存器之间的传送指令 （SH-DSP）

- 指令种类

	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

- 流水线



- 运行说明

流水线完成 IF、 ID、 EX、 MA、 WB/DSP 等 5 个阶段后结束。执行带条件的运算指令时，如果条件不成立，则 WB/DSP 阶段为空操作 （NOP），但不改变流水线。如果该指令与通过 MOVX.W、 MOVY.W、 MOVS.W 或 MOVS.L 的存储器加载并发执行，则产生竞争。如果在该指令后紧接着通过 MOVX.W、 MOVY.W、 MOVS.W 或 MOVS.L 执行存储器存储，也会产生竞争。（详细内容参照 “7.2.4 DSP 寄存器之间传送与存储器加载 / 存储运行的竞争”）。

7.4.9 异常处理

(1) 中断异常处理 （通用）

- 指令种类

中断异常处理

- 流水线

- 运行说明

在指令的 ID 阶段接受中断，将该 ID 阶段以后的部分替换为中断异常处理顺序。

流水线完成 IF、ID、EX、EX、MA、MA、EX、MA、EX、EX 等 10 个阶段后结束。中断异常处理不是延迟转移。中断异常处理中被取的指令变为无效 （IF）。从中断异常处理最后的 EX 所在槽开始执行转移目标指令的 IF。

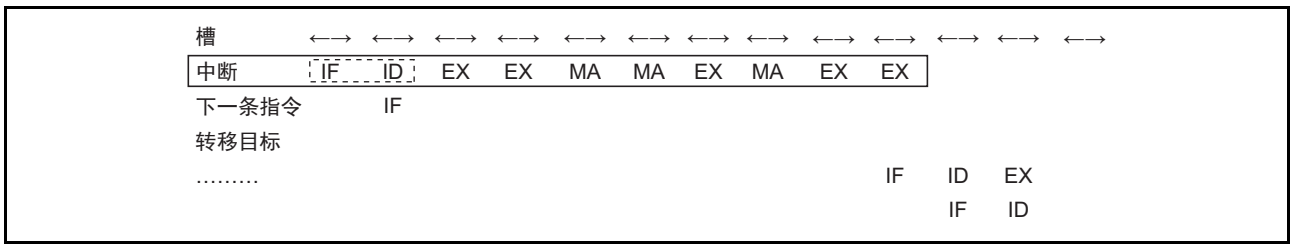
中断源有 NMI、用户断点、 IRQ 及内部外围模块引起的中断。

(2) 地址错误异常处理 （通用）

- 指令种类

地址错误异常处理

- 流水线



- 运行说明

在指令的 ID 阶段接受地址错误，将该 ID 阶段以后的部分替换为地址错误异常处理顺序。

流水线完成 IF、ID、EX、EX、MA、MA、EX、MA、EX、EX 等 10 个阶段后结束。地址错误异常处理不是延迟转移。地址错误异常处理中被取的指令变为无效 （IF）。从地址错误异常处理最后的 EX 所在槽开始执行转移目标指令的 IF。

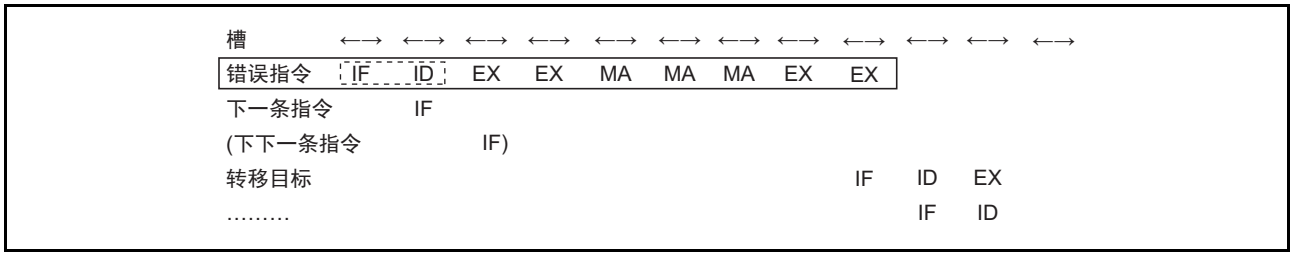
地址错误的产生源有取指令、读取或写入数据，详细内容参照硬件手册。

(3) 非法指令异常处理 （通用）

- 指令种类

非法指令异常处理

- 流水线



- 运行说明

在指令的 ID 阶段接受非法指令，将该 ID 阶段以后的部分替换为非法指令异常处理顺序。

流水线完成 IF、ID、EX、EX、MA、MA、MA、EX、EX 等 9 个阶段后结束。非法指令异常处理不是延迟转移。非法指令异常处理中被取的指令变为无效 （IF）。只在下一条指令执行时产生 IF，或在下下一条指令执行时也产生 IF，取决于将要执行的指令。从非法指令异常处理最后的 EX 所在槽开始执行转移目标指令的 IF。

非法指令异常处理源有一般非法指令以及槽非法指令。如果将配置于紧接着延迟转移指令之后的槽 （延迟槽）以外的未定义码解码，则为一般非法指令异常处理。如果将配置于延迟槽的未定义码解码，或在延迟槽配置改写程序计数器的指令并解码，则为槽非法指令异常处理。

附录

附录 A. 操作码

附录 A.1 CPU 指令说明

按字母顺序表示 CPU 内核执行的指令。

表 A.1 按字母顺序表示 CPU 指令

指令		操作码	操作	执行状态	T 位
ADD	#imm,Rn	0111nnnniiiiiiii	Rn+imm→Rn	1	—
ADD	Rm,Rn	0011nnnnmmmm1100	Rn+Rm→Rn	1	—
ADDC	Rm,Rn	0011nnnnmmmm1110	Rn+Rm+T→Rn、进位 →T	1	进位
ADDV	Rm,Rn	0011nnnnmmmm1111	Rn+Rm→Rn、上溢 →T	1	上溢
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 时 disp×2+PC→PC、 T = 1 时 nop	3/1*[1]	—
BF/S	label	10001111dddddddd	T = 0 时 disp×2+PC→PC、 T = 1 时 nop	2/1*[1]	—
BRA	label	1010dddddddddddd	延迟转移， disp×2+PC→PC	2	—
BRAF	Rm	0000mmmm00100011	延迟转移， Rm+PC→PC	2	—
BSR	label	1011dddddddddddd	延迟转移， PC→PR、 disp×2+PC→PC	2	—
BSRF	Rm	0000mmmm00000011	延迟转移， PC→PR、 Rm+PC→PC	2	—
BT	label	10001001dddddddd	T = 1 时 disp×2+PC→PC、 T = 0 时 nop	3/1*[1]	—
BT/S	label	10001101dddddddd	T = 1 时 disp×2+PC→PC、 T = 0 时 nop	2/1*[1]	—
CLRMAC		0000000000101000	0→MACH、 MACL	1	—
CLRT		0000000000001000	0→T	1	0
CMP/EQ	#imm,R0	10001000iiiiiiii	R0 = imm 时 1→T	1	比较结果
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	Rn = Rm 时 1→T	1	比较结果
CMP/GE	Rm,Rn	0011nnnnmmmm0011	有符号 Rn ≥ Rm 时 1→T	1	比较结果
CMP/GT	Rm,Rn	0011nnnnmmmm0111	有符号 Rn > Rm 时 1→T	1	比较结果
CMP/HI	Rm,Rn	0011nnnnmmmm0110	无符号 Rn > Rm 时 1→T	1	比较结果
CMP/HS	Rm,Rn	0011nnnnmmmm0010	无符号 Rn ≥ Rm 时 1→T	1	比较结果
CMP/PL	Rn	0100nnnn00010101	Rn > 0 时 1→T	1	比较结果
CMP/PZ	Rn	0100nnnn00010001	Rn ≥ 0 时 1→T	1	比较结果
CMP/STR	Rm,Rn	0010nnnnmmmm1100	任意字节相等时 1→T	1	比较结果
DIV0S	Rm,Rn	0010nnnnmmmm0111	Rn 的 MSB→Q、 Rm 的 MSB→M、 M^Q→T	1	计算结果
DIV0U		0000000000011001	0→M/Q/T	1	0

指令		操作码	操作	执行状态	T 位
DIV1	Rm,Rn	0011nnnnmmmm0100	单步除法 (Rn÷Rm)	1	计算结果
DMULS.L	Rm,Rn	0011nnnnmmmm1101	带符号 Rn×Rm→MACH、 MACHL	2 ～ 4*2	—
DMULU.L	Rm,Rn	0011nnnnmmmm0101	无符号 Rn×Rm→ MACH、 MACL	2 ～ 4*2	—
DT	Rn	0100nnnn00010000	Rn–1→Rn， Rn 为 0 时 1→TRn 不为 0 时， 0→T	1	比较结果
EXTS.B	Rm,Rn	0110nnnnmmmm1110	将 Rm 从字节执行符号扩展 →Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	将 Rm 从字执行符号扩展 →Rn	1	—
EXTU.B	Rm,Rn	0110nnnnmmmm1100	将 Rm 从字节执行零扩展 →Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	将 Rm 从字执行零扩展 →Rn	1	—
JMP	@Rm	0100mmmm00101011	延迟转移， Rm→PC	2	—
JSR	@Rm	0100mmmm00001011	延迟转移， 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	—

指令		操作码	操作	执行状态	T 位
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	带符号 (Rn)×(Rm)+MAC→MAC	3/(2 ～ 4)*2	—
MAC.W	@Rm+,@Rn+	0100nnnnmmmm1111	带符号 (Rn)×(Rm)+MAC→MAC	3/(2)*2	—
MOV	#imm,Rn	1110nnnniiiiiiii	imm→ 符号扩展 →Rn	1	—
MOV	Rm,Rn	0110nnnnmmmm0011	Rm→Rn	1	—
MOV.B	@(disp,GBR),R0	11000100dddddddd	(disp+GBR)→ 符号扩展 →R0	1	—
MOV.B	@(disp,Rm),R0	10000100mmmmdddd	(disp+Rm)→ 符号扩展 →R0	1	—
MOV.B	@(R0,Rm),Rn	0000nnnnmmmm1100	(R0+Rm)→ 符号扩展 →Rn	1	—
MOV.B	@Rm+,Rn	0110nnnnmmmm0100	(Rm)→ 符号扩展 →Rn、 Rm+1→Rm	1	—
MOV.B	@Rm,Rn	0110nnnnmmmm0000	(Rm)→ 符号扩展 →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)→ 符号扩展 →R0	1	—
MOV.W	@(disp,PC),Rn	1001nnnndddddddd	(disp×2+PC)→ 符号扩展 →Rn	1	—
MOV.W	@(disp,Rm),R0	10000101mmmmdddd	(disp×2+Rm)→ 符号扩展 →R0	1	—
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0+Rm)→ 符号扩展 →Rn	1	—
MOV.W	@Rm+,Rn	0110nnnnmmmm0101	(Rm)→ 符号扩展 →Rn、 Rm+2→Rm	1	—
MOV.W	@Rm,Rn	0110nnnnmmmm0001	(Rm)→ 符号扩展 →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	—

指令		操作码	操作	执行状态	T 位
MULS.W	Rm,Rn	0010nnnnmmmm1111	带符号 Rn×Rm→MAC	1 ～ 3*2	—
MULU.W	Rm,Rn	0010nnnnmmmm1110	无符号 Rn×Rm→MAC	1 ～ 3*2	—
NEG	Rm,Rn	0110nnnnmmmm1011	0– Rm→Rn	1	—
NEGC	Rm,Rn	0110nnnnmmmm1010	0– Rm– T→Rn、借位 →T	1	借位
NOP		0000000000001001	空操作	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	延迟转移，堆栈区 →PC/SR	4	LSB
RTS		0000000000001011	延迟转移， 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	睡眠	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	—

指令		操作码	操作	执行状态	T 位
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、借位 ⟶T	1	借位
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn–Rm→Rn、下溢 ⟶T	1	下溢
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm→ 交换低位 2 字节的高低字节 →Rn	1	—
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm→ 交换高低字 →Rn	1	—
TAS.B	@Rn	0100nnnn00011011	(Rn) 为 0 时 1→T、 1→MSBof(Rn)	4	测试结果
TRAPA	#imm	11000011iiiiiiii	PC/SR→ 堆栈区、 (imm×4+VBR)→PC	8	—
TST	#imm,R0	11001000iiiiiiii	R0 & imm，结果为 0 时 1→T	1	测试结果
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm，结果为 0 时 1→T	1	测试结果
TST.B	#imm,@(R0,GBR)	11001100iiiiiiii	(R0+GBR)&imm，结果为 0 时 1→T	3	测试结果
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 的中间 32 位 →Rn	1	—

【注】 *1 不转移时，为 1 个状态。

*2 表示普通执行状态。根据与前后指令的竞争关系，改变执行状态数。

附录 A.2 追加的 CPU 指令

SH-2 追加的 SH-DSP 的 CPU 指令（3 种、 40 条指令）如表 A.2 所示。

SH-1 追加的 SH-2 的 CPU 指令（6 种、 9 条指令）如表 A.3 所示。

表 A.2 SH-2 追加的 SH-DSP 的 CPU 指令

指令		操作码	操作	执行状态	T 位
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］)、重复控制标志 →RF1、 RF0	1	—
SETRC	#imm	10000010iiiiiiii	imm→RC(SR［23:16］)、 zeros→SR［27:24］、 重复控制标志 →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	—
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 追加的 SH-2 的 CPU 指令

指令	操作	操作码	执行状态	T 位
BF/S label	T = 0 时 disp×2+PC→PC、 T = 1 时 nop	10001111dddddddd	2/1*2	—
BRAF Rm	延迟转移， Rm+PC→PC	0000mmmm00100011	2	—
BSRF Rm	延迟转移， PC→PR、 Rm+PC→PC	0000mmmm00000011	2	—
BT/S label	T = 1 时 disp×2+PC→PC、 T = 0 时 nop	10001101dddddddd	2/1*2	—
DMULS.L Rm,Rn	带符号 Rn×Rm→MACH、MACL32×32→64 位	0011nnnnmmmm1101	2(～ 4)*1	—
DMULU.L Rm,Rn	无符号 Rn×Rm→MACH、MACL32×32→64 位	0011nnnnmmmm0101	2(～ 4)*1	—
DT Rn	Rn–1→Rn、 Rn 为 0 时 1→T、 Rn 不为 0 时 0→T	0100nnnn00010000	1	比较结果
MAC.L @Rm+,@Rn+	带符号 (Rn)×(Rm)+MAC→MAC	0000nnnnmmmm1111	2(～ 4)*1	—
MUL.L Rm,Rn	Rn×Rm→MACL	0000nnnnmmmm0111	2(～ 4)*1	

附录 A.3 DSP 数据传送指令

按字母顺序表示 DSP 数据传送指令。

表 A.4 按字母顺序表示 DSP 数据传送指令

指令	操作	操作码	执行状态	DC 位	适用指令		
					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：操作数的高位字
LSW：操作数的低位字

附录 A.4 DSP 运算指令

按字母顺序表示 DSP 运算指令。

表 A.5 按字母顺序表示 DSP 运算指令

指令	操作	操作码	执行状态	DC 位	适用指令		
					SH-1	SH-2	SH-DSP
PABS Sx,Dz	Sx ≥ 0 时 Sx→Dz Sx < 0 时 0–Sx→Dz	111110********** 10001000xx00zzzz	1	更新	—	—	○
PABS Sy,Dz	Sy ≥ 0 时 Sy→Dz Sy < 0 时 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	DC = 1 时 Sx+Sy→Dz 为 0 时 nop	111110********** 10110010xxyyzzzz	1	—	—	—	○
DCF PADD Sx,Sy,Dz	DC = 0 时 Sx+Sy→Dz 为 1 时 nop	111110********** 10110011xxyyzzzz	1	—	—	—	○
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx+Sy→Du Se 的高位字 ×sf 的高位字 →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，清除 Dz 的低位字	111110********** 10010101xxyyzzzz	1	更新	—	—	○
DCT PAND Sx,Sy,Dz	DC = 1 时 Sx&Sy→Dz，清除 Dz 的低位字 为 0 时 nop	111110********** 10010110xxyyzzzz	1	—	—	—	○
DCF PAND Sx,Sy,Dz	DC = 0 时 Sx&Sy→Dz，清除 Dz 的低位字 为 1 时 nop	111110********** 10010111xxyyzzzz	1	—	—	—	○
PCLR Dz	H'00000000→Dz	111110********** 1000110100000zzzz	1	更新	—	—	○
DCT PCLR Dz	DC = 1 时 H'00000000→Dz 为 0 时 nop	111110********** 1000111000000zzzz	1	—	—	—	○
DCF PCLR Dz	DC = 0 时 H'00000000→Dz 为 1 时 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	DC = 1 时 Sx→Dz 为 0 时 nop	111110********** 11011010xx00zzzz	1	—	—	—	○
DCT PCOPY Sy,Dz	DC = 1 时 Sy→Dz 为 0 时 nop	111110********** 1111101000yyzzzz	1	—	—	—	○

指令	操作	操作码	执行状态	DC 位	适用指令		
					SH-1	SH-2	SH-DSP
DCF PCOPY Sx,Dz	DC = 0 时 Sx→Dz 为 1 时 nop	111110********** 11011011xx00zzzz	1	—	—	—	○
DCF PCOPY Sy,Dz	DC = 0 时 Sy→Dz 为 1 时 nop	111110********** 1111101100yyzzzz	1	—	—	—	○
PDEC Sx,Dz	Sx 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字	111110********** 10001001xx00zzzz	1	更新	—	—	○
PDEC Sy,Dz	Sy 的高位字 –1→Dz 的高位字， 清除 Dz 的低位字	111110********** 10101001xx00zzzz	1	更新	—	—	○
DCT PDEC Sx,Dz	DC = 1 时 Sx 的高位字 –1→Dz 的高位字，清除 Dz 的低位字 为 0 时 nop	111110********** 10001010xx00zzzz	1	—	—	—	○
DCT PDEC Sy,Dz	DC = 1 时 Sy 的高位字 –1→Dz 的高位字，清除 Dz 的低位字 为 0 时 nop	111110********** 10101010xx00zzzz	1	—	—	—	○
DCF PDEC Sx,Dz	DC = 0 时 Sx 的高位字 –1→Dz 的高位字，清除 Dz 的低位字 为 1 时 nop	111110********** 10001011xx00zzzz	1	—	—	—	○
DCF PDEC Sy,Dz	DC = 0 时 Sy 的高位字 –1→Dz 的高位字，清除 Dz 的低位字 为 1 时 nop	111110********** 10101011xx00zzzz	1	—	—	—	○
PDMSB Sx,Dz	Sx 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字	111110********** 10011101xx00zzzz	1	更新	—	—	○
PDMSB Sy,Dz	Sy 数据的 MSB 位置 →Dz 的高位字， 清除 Dz 的低位字	111110********** 1011110100yyzzzz	1	更新	—	—	○
DCT PDMSB Sx,Dz	DC = 1 时 Sx 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 0 时 nop	111110********** 10011110xx00zzzz	1	—	—	—	○
DCT PDMSB Sy,Dz	DC = 1 时 Sy 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 0 时 nop	111110********** 1011111000yyzzzz	1	—	—	—	○
DCF PDMSB Sx,Dz	DC = 0 时 Sx 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 1 时 nop	111110********** 10011111xx00zzzz	1	—	—	—	○
DCF PDMSB Sy,Dz	DC = 0 时 Sy 数据的 MSB 位置 →Dz 的高位字，清除 Dz 的低位字 为 1 时 nop	111110********** 1011111100yyzzzz	1	—	—	—	○
PINC Sx,Dz	Sx 的高位字 +1→Dz 的高位字，清除 Dz 的低位字	111110********** 10011001xx00zzzz	1	更新	—	—	○
PINC Sy,Dz	Sy 的高位字 +1→Dz 的高位字，清除 Dz 的低位字	111110********** 1011100100yyzzzz	1	更新	—	—	○
DCT PINC Sx,Dz	DC = 1 时 Sx 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 0 时 nop	111110********** 10011010xx00zzzz	1	—	—	—	○

指令	操作	操作码	执行状态	DC 位	适用指令		
					SH-1	SH-2	SH-DSP
DCT PINC Sy,Dz	DC = 1 时 Sy 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 0 时 nop	111110********** 1011101000yyzzzz	1	—	—	—	○
DCF PINC Sx,Dz	DC = 0 时 Sx 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 1 时 nop	111110********** 10011011xx00zzzz	1	—	—	—	○
DCF PINC Sy,Dz	DC = 0 时 Sy 的高位字 +1→Dz 的高位字，清除 Dz 的低位字 为 1 时 nop	111110********** 1011101100yyzzzz	1	—	—	—	○
PLDS Dz,MACH	Dz→MACH	111110********** 1110110100000zzzz	1	—	—	—	○
PLDS Dz,MACL	Dz→MACL	111110********** 1111110100000zzzz	1	—	—	—	○
DCT PLDS Dz,MACH	DC = 1 时 Dz→MACH 为 0 时 nop	111110********** 1110111000000zzzz	1	—	—	—	○
DCT PLDS Dz,MACL	DC = 1 时 Dz→MACL 为 0 时 nop	111110********** 1111111000000zzzz	1	—	—	—	○
DCF PLDS Dz,MACH	DC = 0 时 Dz→MACH 为 1 时 nop	111110********** 1110111100000zzzz	1	—	—	—	○
DCF PLDS Dz,MACL	DC = 0 时 Dz→MACL 为 1 时 nop	111110********** 1111111100000zzzz	1	—	—	—	○
PMULS Se,Sf,Dg	Se 的高位字 ×sf 的高位字 →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	DC = 1 时 0 –Sx→Dz 为 0 时 nop	111110********** 11001010xx00zzzz	1	—	—	—	○
DCT PNEG Sy,Dz	DC = 1 时 0–Sy→Dz 为 0 时 nop	111110********** 1110101000yyzzzz	1	—	—	—	○
DCF PNEG Sx,Dz	DC = 0 时 0–Sx→Dz 为 1 时 nop	111110********** 11001011xx00zzzz	1	—	—	—	○
DCF PNEG Sy,Dz	DC = 0 时 0–Sy→Dz 为 1 时 nop	111110********** 1110101100yyzzzz	1	—	—	—	○
POR Sx,Sy,Dz	Sx | Sy→Dz，清除 Dz 的低位字	111110********** 10110101xxyyzzzz	1	更新	—	—	○
DCT POR Sx,Sy,Dz	DC = 1 时 Sx|Sy→Dz，清除 Dz 的低位字 为 0 时 nop	111110********** 10110110xxyyzzzz	1	—	—	—	○
DCF POR Sx,Sy,Dz	DC = 0 时 Sx|Sy→Dz，清除 Dz 的低位字 为 1 时 nop	111110********** 10110111xxyyzzzz	1	—	—	—	○

指令	操作	操作码	执行状态	DC 位	适用指令		
					SH-1	SH-2	SH-DSP
PRND Sx,Dz	Sx+H'00008000→Dz 清除 Dz 的低位字	111110********** 10011000xx00zzzz	1	更新	—	—	○
PRND Sy,Dz	Sy+H'00008000→Dz 清除 Dz 的低位字	111110********** 1011100000yyzzzz	1	更新	—	—	○
PSHA Sx,Sy,Dz	Sy ≥ 0 时 Sx << Sy→Dz Sy < 0 时 Sx >> Sy→Dz	111110********** 10010001xxyyzzzz	1	更新	—	—	○
DCT PSHA Sx,Sy,Dz	DC = 1 & Sy ≥ 0 时 Sx << Sy→Dz DC = 1 & Sy < 0 时 Sx >> Sy→Dz DC = 0 时 nop	111110********** 10010010xxyyzzzz	1	—	—	—	○
DCF PSHA Sx,Sy,Dz	DC = 0 & Sy ≥ 0 时 Sx << Sy→Dz DC = 0 & Sy < 0 时 Sx >> Sy→Dz DC = 1 时 nop	111110********** 10010011xxyyzzzz	1	—	—	—	○
PSHA #imm,Dz	imm ≥ 0 时 Dz << imm→Dz imm < 0 时 Dz >> imm→Dz	111110********** 00010iiiiiiizzzz	1	更新	—	—	○
PSHL Sx,Sy,Dz	Sy ≥ 0 时 Sx << Sy→Dz， 清除 Dz 的低位字 Sy < 0 时 Sx >> Sy→Dz， 清除 Dz 的低位字	111110********** 10000001xxyyzzzz	1	更新	—	—	○
DCT PSHL Sx,Sy,Dz	DC = 1 & Sy ≥ 0 时 Sx << Sy→Dz，清除 Dz 的低位字 DC = 1 & Sy < 0 时 Sx >> Sy→Dz，清除 Dz 的低位字 DC = 0 时 nop	111110********** 10000010xxyyzzzz	1	—	—	—	○
DCF PSHL Sx,Sy,Dz	DC = 0 & Sy ≥ 0 时 Sx << Sy→Dz，清除 Dz 的低位字 DC = 0 & Sy < 0 时 Sx >> Sy→Dz，清除 Dz 的低位字 DC = 1 时 nop	111110********** 10000011xxyyzzzz	1	—	—	—	○
PSHL #imm,Dz	imm ≥ 0 时 Dz << imm→Dz，清除 Dz 的低位字 imm < 0 时 Dz >> imm→Dz，清除 Dz 的低位字	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	DC = 1 时 MACH→Dz 为 0 时 nop	111110********** 1100111000000zzzz	1	—	—	—	○
DCT PSTS MACL,Dz	DC = 1 时 MACL→Dz 为 0 时 nop	111110********** 1101111000000zzzz	1	—	—	—	○
DCF PSTS MACH,Dz	DC = 0 时 MACH→Dz 为 1 时 nop	111110********** 1100111100000zzzz	1	—	—	—	○

指令	操作	操作码	执行状态	DC 位	适用指令		
					SH-1	SH-2	SH-DSP
DCF PSTS MACL,Dz	DC = 0 时 MACL→Dz 为 1 时 nop	111110********** 110111110000zzzz	1	—	—	—	○
PSUB Sx,Sy,Dz	Sx–Sy→Dz	111110********** 10100001xxyyzzzz	1	更新	—	—	○
DCT PSUB Sx,Sy,Dz	DC = 1 时 Sx-Sy→Dz 为 0 时 nop	111110********** 10100010xxyyzzzz	1	—	—	—	○
DCF PSUB Sx,Sy,Dz	DC = 0 时 Sx-Sy→Dz 为 1 时 nop	111110********** 10100011xxyyzzzz	1	—	—	—	○
PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx–Sy→Du Se 的高位字× sf 的高位字 →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，清除 Dz 的低位字	111110********** 10100101xxyyzzzz	1	更新	—	—	○
DCT PXOR Sx,Sy,Dz	DC = 1 时 Sx^Sy→Dz，清除 Dz 的低位字 为 0 时 nop	111110********** 10100110xxyyzzzz	1	—	—	—	○
DCF PXOR Sx,Sy,Dz	DC = 0 时 Sx^Sy→Dz，清除 Dz 的低位字 为 1 时 nop	111110********** 10100111xxyyzzzz	1	—	—	—	○

【注】 *1 根据 PADD 的运算结果更新。
*2 根据 PSUB 的运算结果更新。

附录 B. SH-DSP、 SH-2、 SH-1 的比较

- 功能比较

SH-DSP、 SH-2、 SH-1 的功能比较如表 B.1 所示。

表 B.1 SH-DSP、 SH-2、 SH-1 的功能比较

		SH-DSP	SH-2	SH-1
指令数	CPU 指令	65 种 182 条指令	62 种 142 条指令	56 种 133 条指令
	DSP 指令	26 种 135 条指令	—	—
已追加的 SH 指令		（对于 SH-2） 3 种 40 条指令 （参照表 A.2）	（对于 SH-1） 6 种 9 条指令 （参照表 A.3）	—
MAC.W 的功能		16×16+64→64 位 7 段流水线 （IF、 ID、 EX、 MA、 MA、 mm、 mm）	16×16+64→64 位 7 段流水线 （IF、 ID、 EX、 MA、 MA、 mm、 mm）	16×16+42→42 位 8 段流水线 （IF、 ID、 EX、 MA、 MA、 mm、 mm、 mm）
MULS.W MULU.W 的功能		6 段流水线 （IF、 ID、 EX、 MA、 mm、 mm）	6 段流水线 （IF、 ID、 EX、 MA、 mm、 mm）	7 段流水线 （IF、 ID、 EX、 MA、 mm、 mm、 mm）
PMULS 的功能		16×16→40 位 5 段流水线 （IF、ID、EX、MA、WB/DSP）	—	—

修订记录	SH-1/SH-2/SH-DSP 软件手册

Rev.	发行日	修订内容	
		页	修订处
1.00	2009.01.23	—	初版发行

瑞萨 32 位 RISC 单片机
软件手册
SH-1/SH-2/SH-DSP

Publication Date:	Rev1.00, Jan. 23, 2009
Published by:	Sales Strategic Planning Div. Renesas Technology Corp.
Edited by:	Customer Support Department Global Strategic Communication Div. Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.
Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7858/7898

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2377-3473

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 3518-3399

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.
Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> (2) 796-3115, Fax: <82> (2) 796-2145

Renesas Technology Malaysia Sdn. Bhd
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

SH-1/SH-2/SH-DSP

瑞萨电子株式会社

RCJ09B0065-0100